Install, Configure, and Script Cypress for JavaScript Web Applications

Transcripts for Kent C. Dodds (https://egghead.io/instructors/kentcdodds) course on egghead.io (https://egghead.io/courses/install-configure-and-script-cypress-for-javascript-web-applications).

Description

Cypress (https://www.cypress.io/) is an incredibly powerful web testing tool. It's capable of testing any web application. Its architecture places it a cut above similar end-to-end testing tools. Its developer experience is best-in-class. And because Cypress runs your tests in the same context as the rest of your application, you're able to get speed, reliability, and debuggability that are just a long-distant dream with similar tools. The catch? There is no catch. Cypress is exceptional.

In this course, we'll go over how you can install, configure, and script Cypress to test modern, real-world JavaScript web applications.

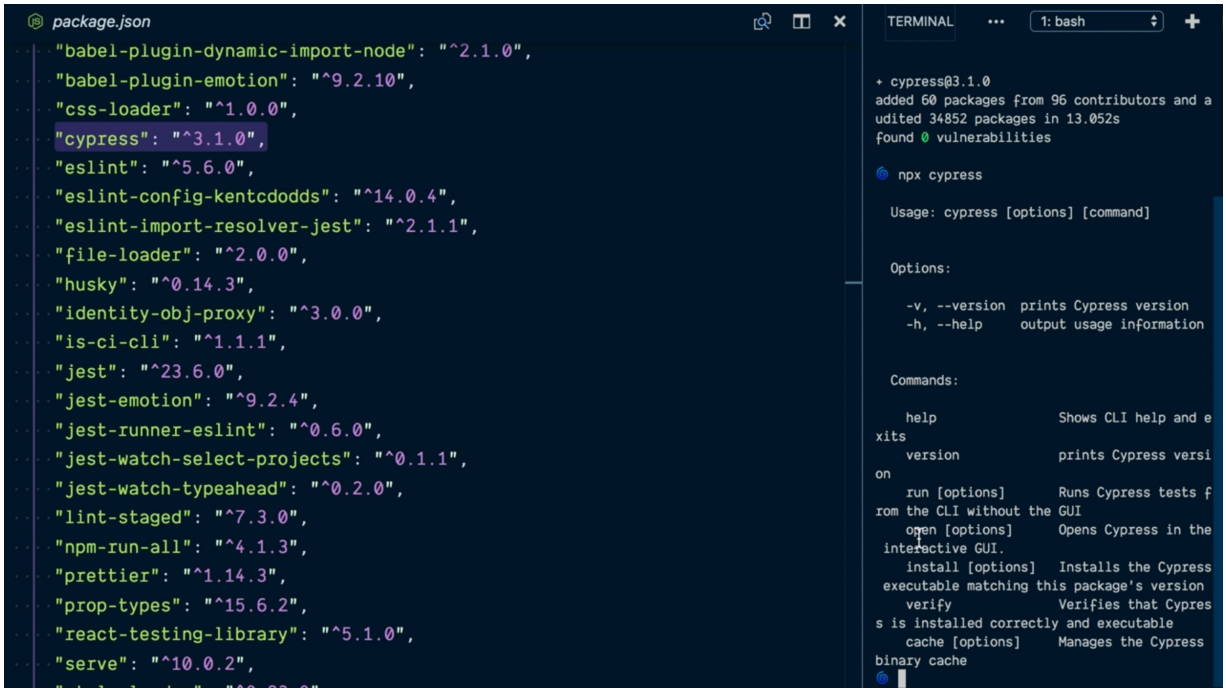
Install and run Cypress

To install Cypress into our project, we simply `npm install` as a dev dependency `cypress`.

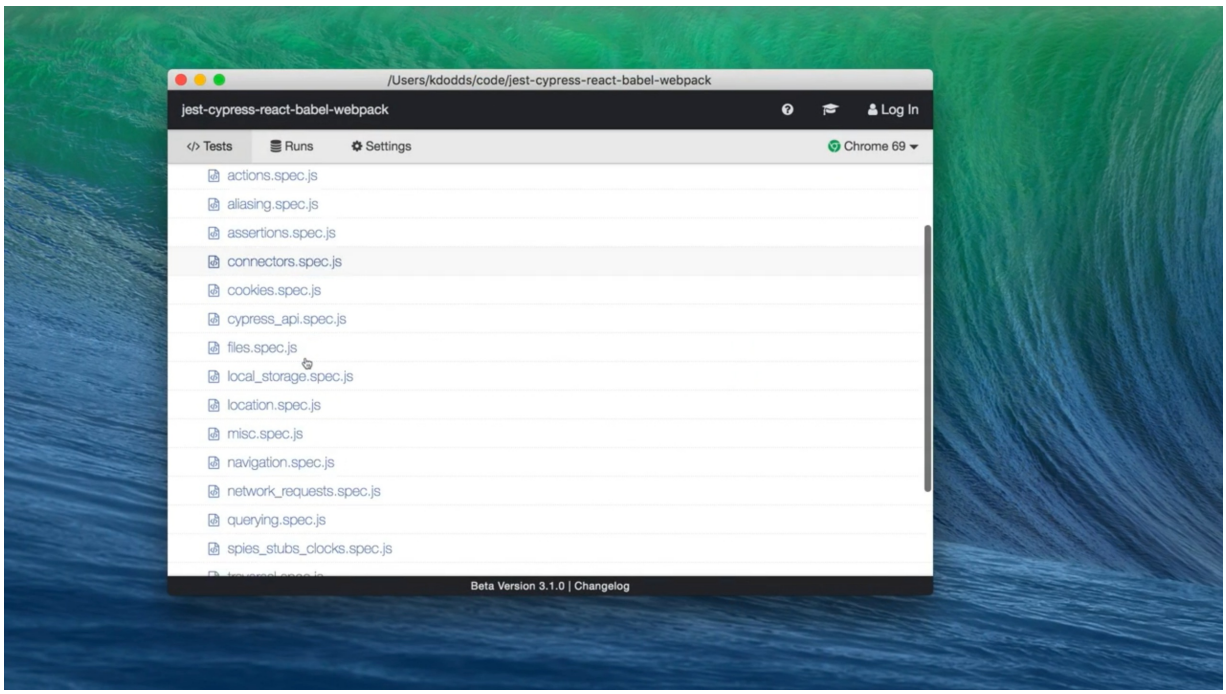
```
npm install --save-dev cypress
```

With Cypress installed, we can look at our `package.json` and we'll see that it has been added to our `devDependencies` right here.

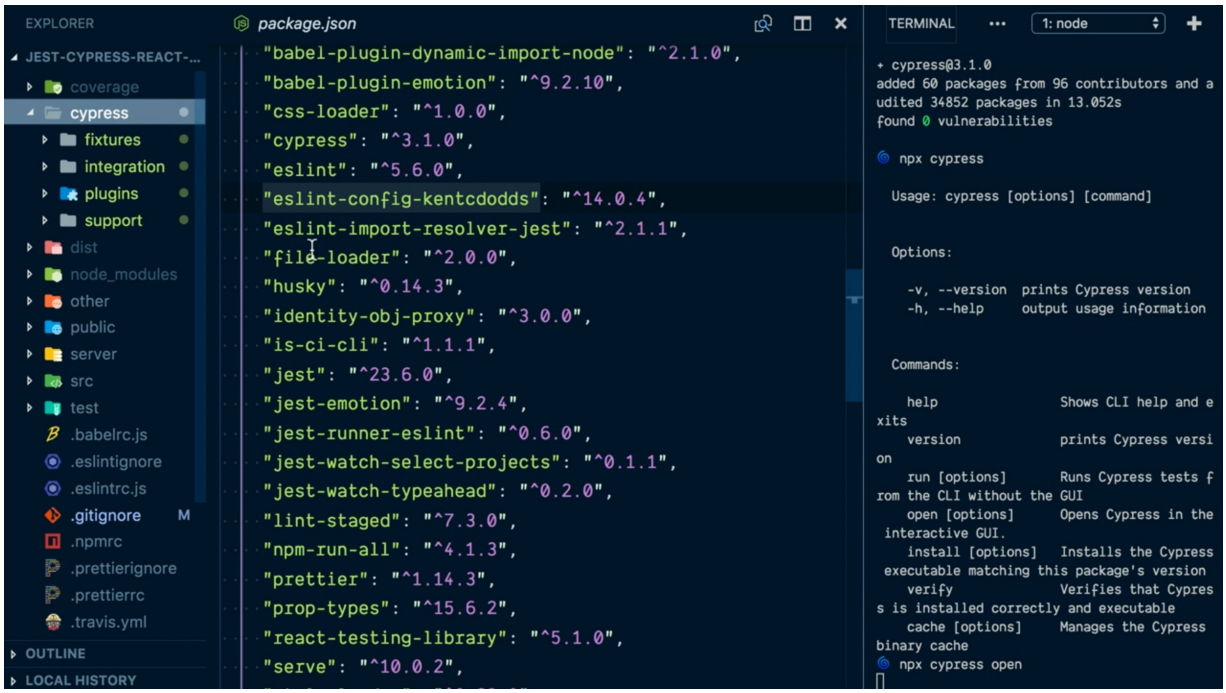
Now we can use `npx` to access the Cypress binary that has been added to our `node_modules` directory. Here we see a list of commands, `help`, `version`, `run`, and `open`.



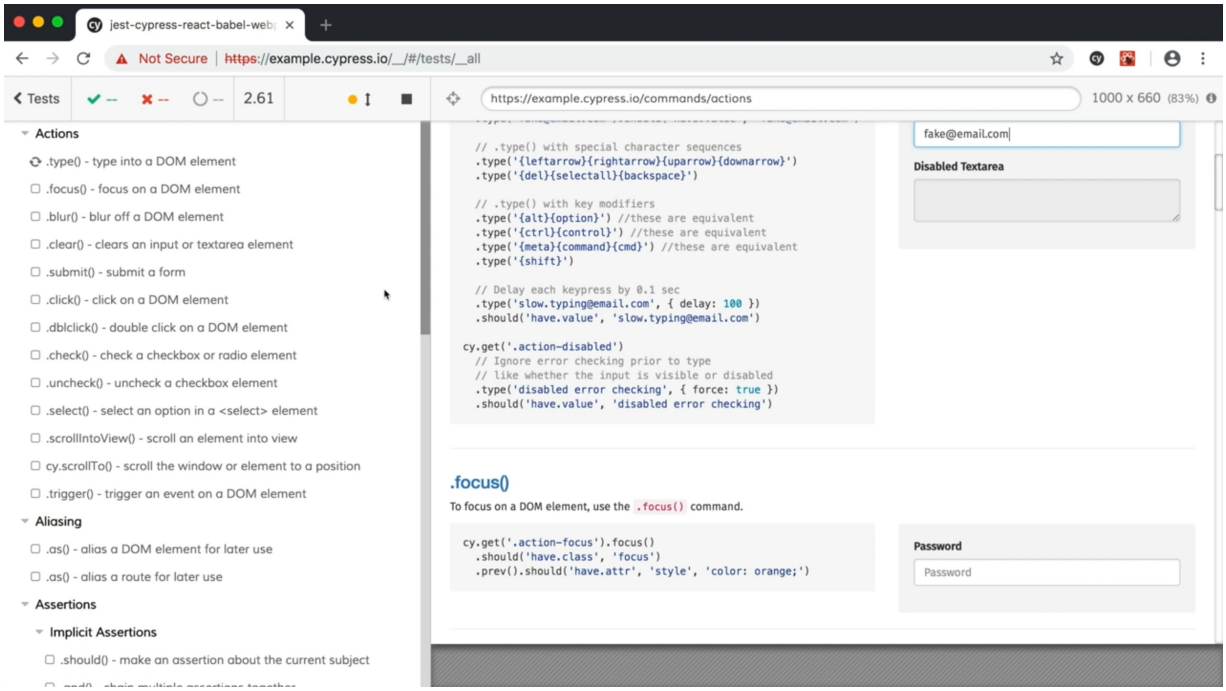
Let's go ahead and run `npx cypress open`. This is going to open up the Cypress application for us.



It automatically adds a whole bunch of tests to our project. We can see those tests added in this Cypress directory here.



We can go ahead and run all these specs and this is going to pull open the Chrome browser inside of the Cypress test runner.



Let's go ahead and stop this. Go back to our test, so we can stop those tests. These are all just some examples that we can use as we're getting started testing with Cypress.

You'll notice that we have some red underlines here. That's because our ESLint isn't liking Cypress very much. Let's go ahead and we'll fix that really quick by `npm install` as a dev dependency `eslint-plugin-cypress`.

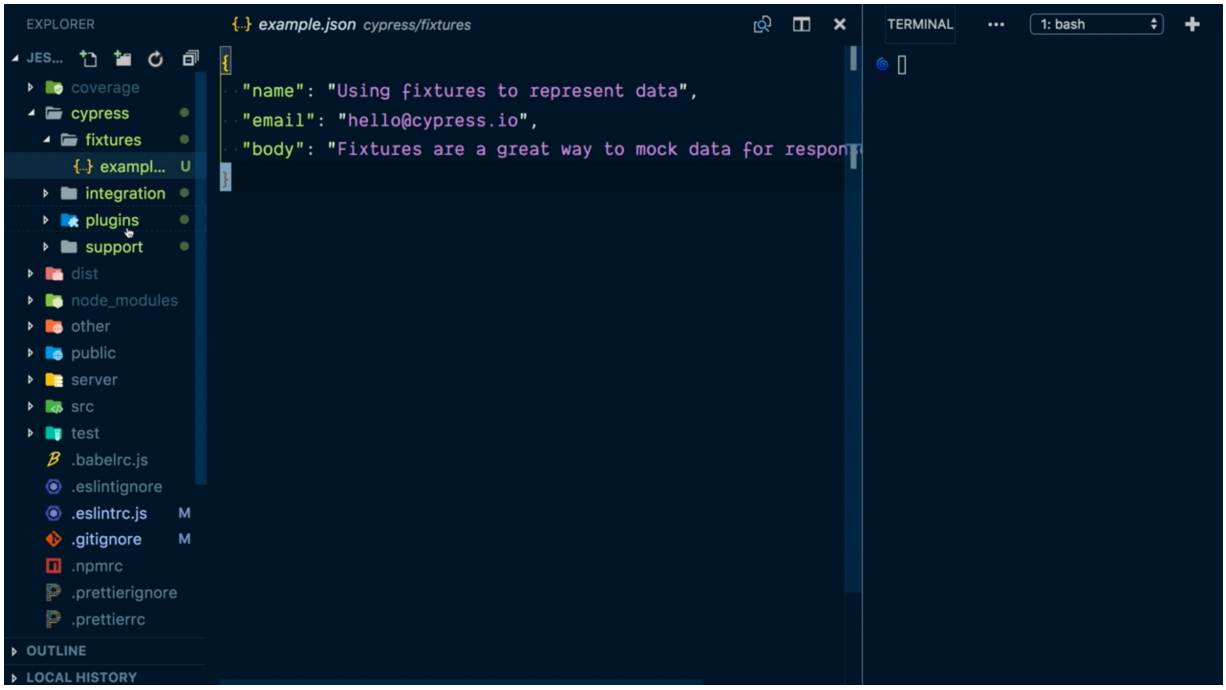
With that installed, we can check out our `devDependencies` here in our `package.json`. We can now go to the `.eslintrc.js` here. We'll say `plugins: ['eslint-plugin-cypress']`. We'll say `env: {'cypress/globals': true}`.

.eslintrc.js

```
plugins: ['eslint-plugin-cypress'],
env: {'cypress/globals': true},
```

If we open up the Cypress testing file, we're good with the ESLint.

Another thing that this creates for us is this `cypress.json` file. There's no configuration in here, but this is where we'll put configuration for Cypress in our project. It also creates a `fixtures example` here.



There is also `plugins` and `support`.

One last thing I'm going to do here is in my `.gitignore`, I'm going to add `cypress/videos` and `cypress/screenshots`, because as we're testing with Cypress, it will create two more directories for videos and screenshots. I don't want to commit those to source control.

In review, to get Cypress up and running in our project, we installed Cypress as a `devDependencies`, and that let us use this Cypress binary, which is right here. We can access it in our `npm` scripts or with `npx`. Then, we ran `cypress open` which initialized our project for Cypress by creating a Cypress directory with some default test and a `cypress.json` configuration file.

Then, we installed `eslint-plugin-cypress`. We can integrate our ESLint configuration with Cypress. We updated our `.gitignore` to exclude Cypress screenshots and Cypress videos.

Write the first Cypress Test

With Cypress installed, let's go ahead and we'll delete this `examples` directory that they generated for us. Then, I'll go ahead and go into our `fixtures`. I'm going to just leave this example, like it is or just empty it out like that.

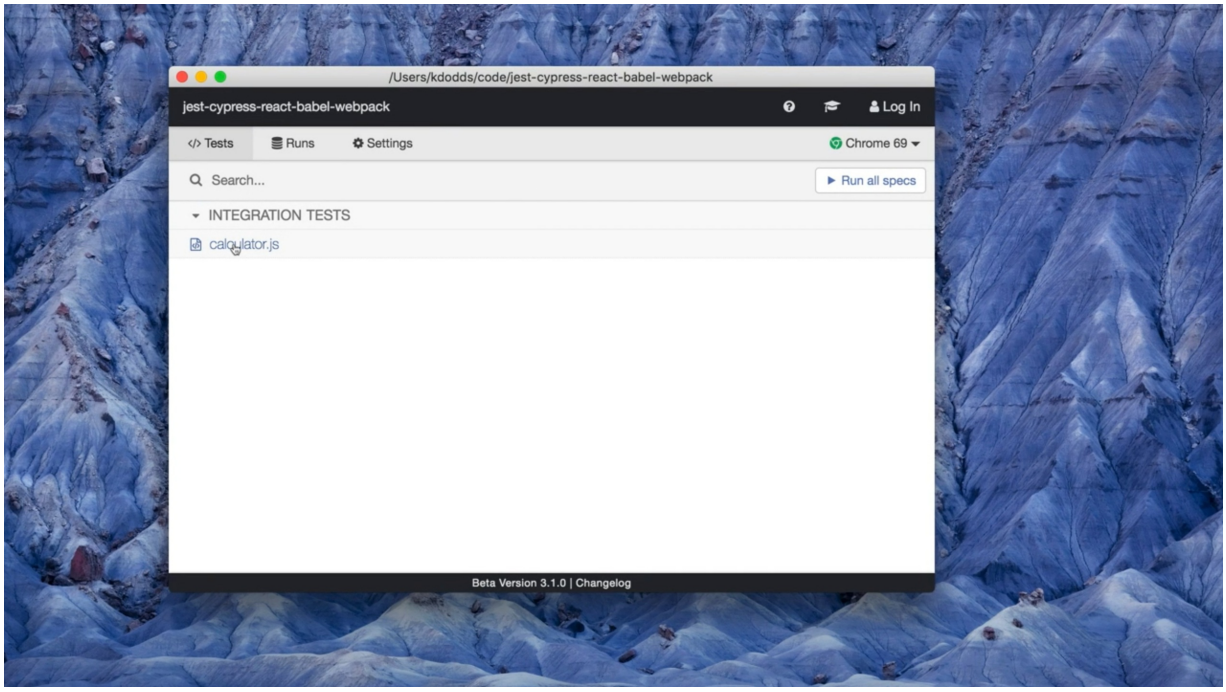
I'm going to add a new file in here called `calculator.js`. This is where we're going to test out our calculator. Cypress uses a mocha like framework for its testing. We have a `describe` block. Here, we'll just say `anonymous calculator`.

```
describe('anonymous calculator', () => {
})
```

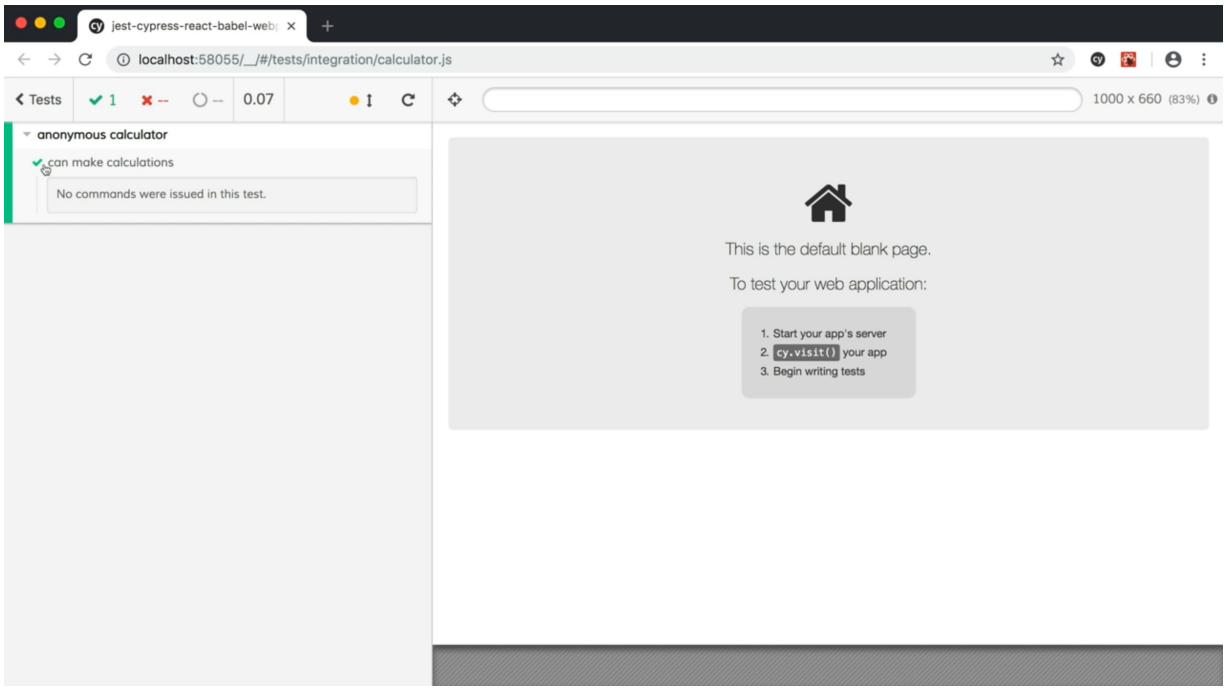
The experience of using the calculator without having logged in. We'll say `it can make calculations`.

```
describe('anonymous calculator', () => {
     it('can make calculations', () => {
     })
})
```

Let's go ahead and we'll save this as it is. We'll start up Cypress. Run `npx cypress open`. That will start up the Cypress app.

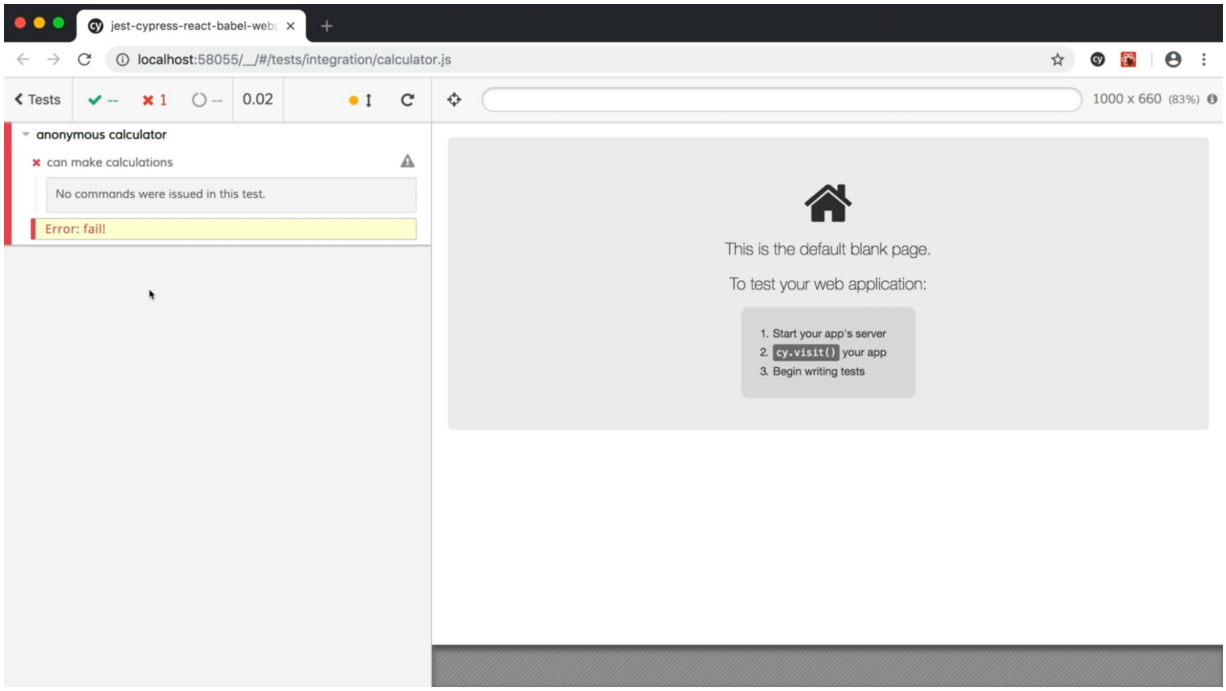


If I click on the `calculator.js` file, it will open up Cypress. Here, we'd see our test. No commands are issued during this test, but the test did pass.



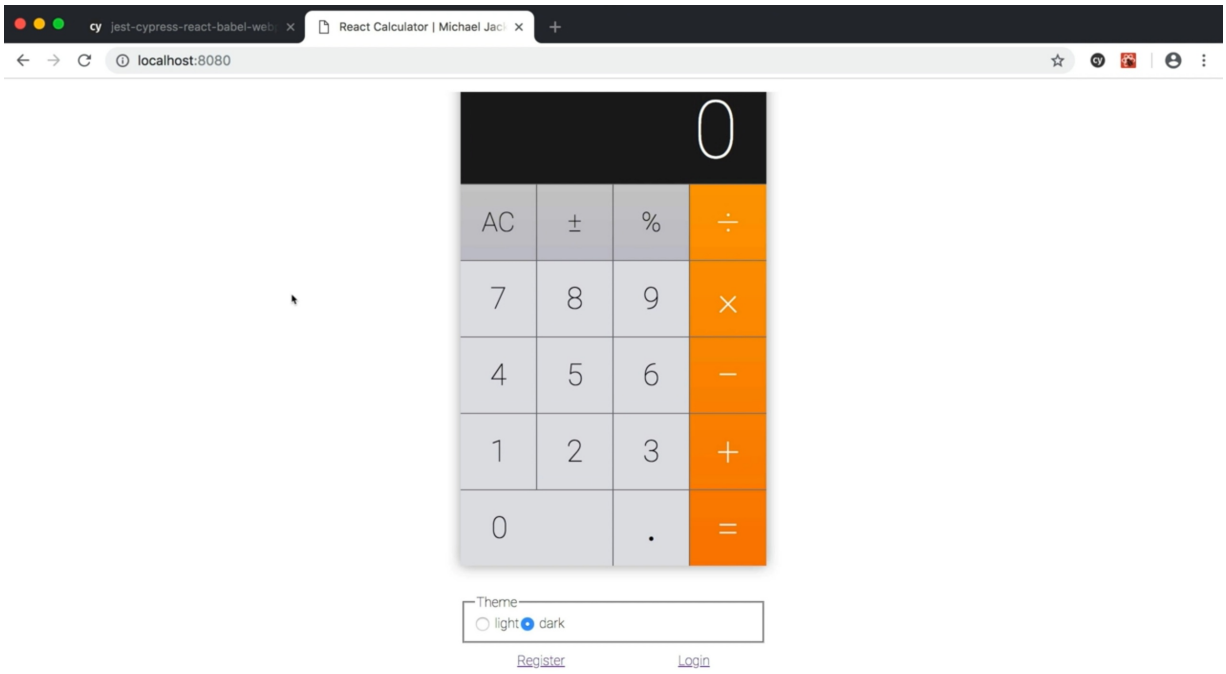
We can make this test fail by throwing an error, so we could say `throw new Error('fail')`.

We save that and Cypress will automatically rerun and it will fail the test.



Let's go ahead and start up our application now. I'm going to just open a new window here. Here, we'll run `npm run dev` to start our node server back end and our webpack-dev-server front end, which will start on http://localhost:8080 (http://localhost:8080).

Now if I just open up a new tab here, we'll go http://localhost:8080 (http://localhost:8080). We'll see our calculator right here.



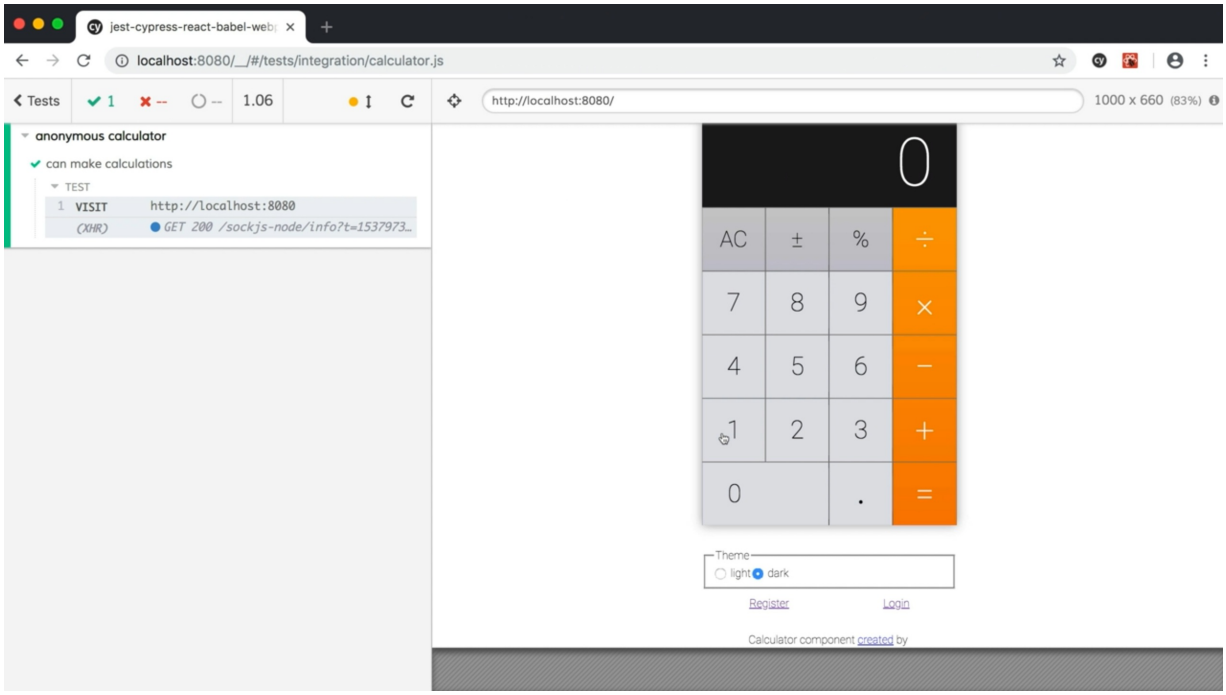
Perfect. We need to make Cypress, the little browser in here. Go to our calculator app. Here it says, "Start your app server."

We did that. Now, we need to do a `cy.visit()` to our app. We can begin writing our test. Let's do that.

We'll say `cy.visit('http://localhost:8080')`.

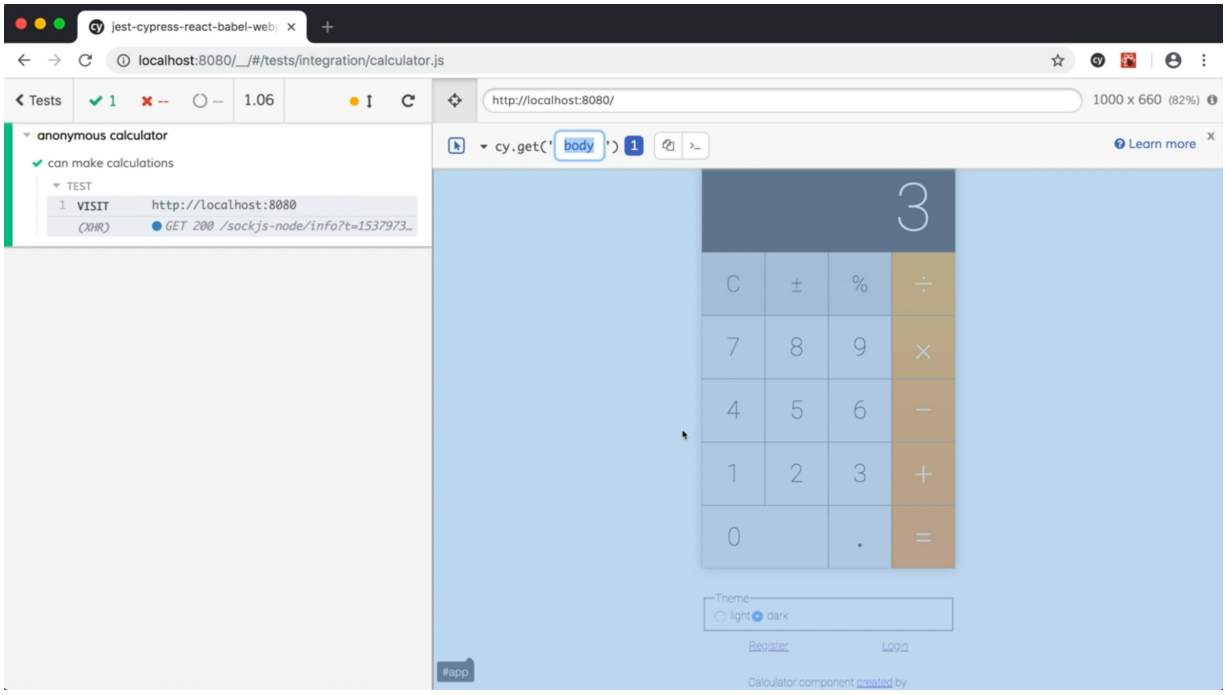
```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('http://localhost:8080')
  })
})
```

I'll save that and reruns pretty quick, but it visits http://localhost:8080 (http://localhost:8080). Now, we're in our app.



Let's say we wanted to do `1 + 2`, and `=`, and verify that `3` shows up. Just to verify that this thing actually can add two numbers together.

How do we make Cypress click on the 1, then +, then 2, then =, then verify that this has 3? What we can use this selector playground to go and select the DOM node that we want to click.



Here, it will show us a selector that will match that DOM node. The selectors are not my favorite, because we're using CSS modules. This class name is not very great, but we can look at a way to improve this later.

I'm going to go ahead and copy this in your CSS `cy.get` and the string here, so that's exactly what we can do in our code here.

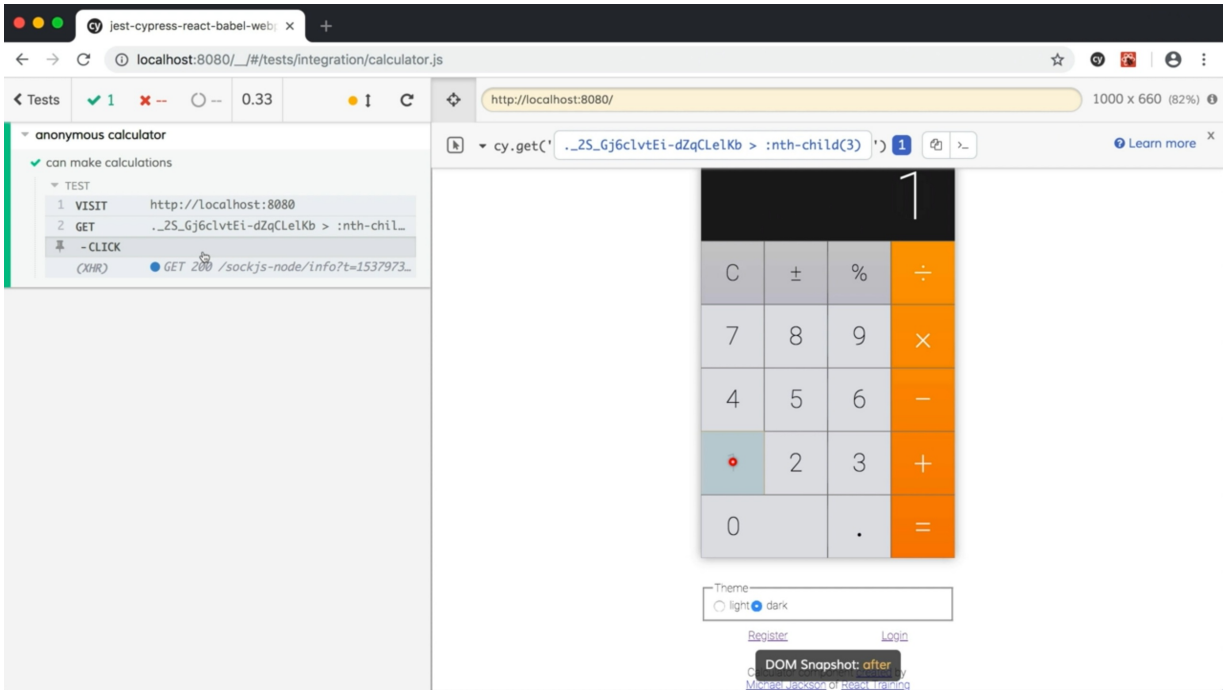
Cy's commands are trainable, so we could do `cy.get` and paste it right in there, or we can just do `.get` and we can just chain things on in this way.

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('http://localhost:8080')
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
```

Once we've gotten that, then we can `.click()`. Cypress will apply this command to the subject of Cypress. When we do a command like `get`, and there are various other commands that changes the subjects of the chain.

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('http://localhost:8080')
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
      .click()
```

When we execute the `click` command, it will execute on the most recent subject. If I save this, and going here, we're going to see that, we first visit, then we get that node. Then, we `click` on that button. You can see right here it's showing us a DOM snapshot, a before and after.



Before, the number is still `0`, and after, the number is `1`, and `AC` turns to `C`. We could add some assertions to make sure that's exactly what's happening, but let's continue on.

Now, we need to change the subject to this number `2`. So we can click on it. I'm going to go ahead and we'll open this up again. I'll click on that. I'll copy this. We'll say `.get`. That changes the subject. Now, we can `.click`.

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('http://localhost:8080')
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(4)')
      .click()
```

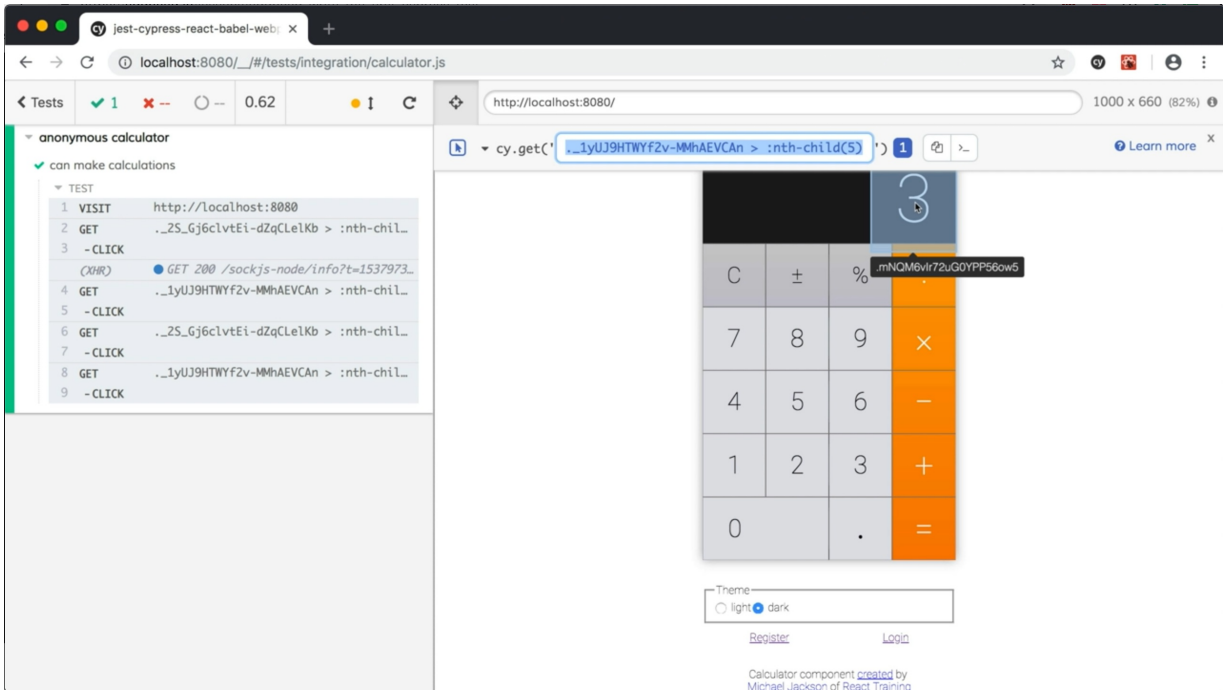
If I save this, let's refresh this and try that again. There we go.

Oh oops. We actually don't want to click on that. We want to click on the `+` first. Let's go ahead and we'll select the `+`. I'll get that. Turn off the playground. We'll do `.get` here first and `.click` on that.

If we refresh this, we can watch it saying `1 + 2`. Now, we got a click on this `=`. Let's do this again. We'll copy this. We'll say `get`, which changes the subject to that `=` sign and click on that `=` sign.

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('http://localhost:8080')
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(4)')
      .click()
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(4)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(5)')
      .click()
```

Cool. Now, we got `3`.



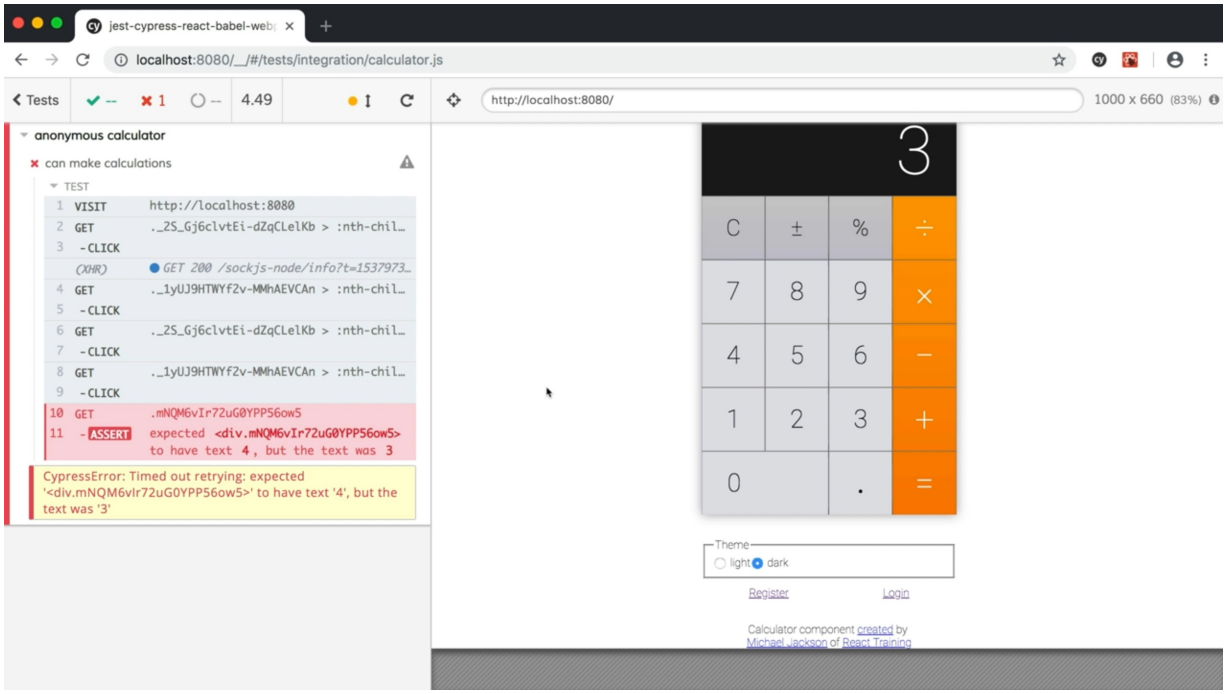
Now, we got to verify that this actually does say `3`, so we'll select this. We got a class there. We'll say `get`. And now we have made the subject the total amount displayed here. Now, we can make an assertion.

We'll say `.should('have.text', '3')`.

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('http://localhost:8080')
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(4)')
      .click()
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(4)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(5)')
      .click()
      .get('.mNQM6vIr72uG0YPP56ow5')
      .should('have.text', '3')
  })
})
```

That assertion works. I can fail that assertion with saying it should equal 4.

It'll wait for four seconds to see if maybe the app is going to update it to 4 eventually. After four seconds, it will say no.



I think the app is settled, and that number is not going to change to `4`. Let's change this back to `3`. We have our first test running.

In review, the way that we made this work is we started out by describing a series of test that we're going to be making, so what that calculator does when there is an anonymous user.

Then, we have a specific test case. It can make calculations. At the very first, we visit our application which we do have to have running on port `8080`. First, we get the `1` and we `click` on that. Then, we get the `+` and we `click` on that. We get the `2`, we `click` on that. We get `=`, `click` on that.

Then, we get the display. We verify that it `should`` have text` of `3`. The way that I'd like to think about this `cy` object is I almost would rather call it a user, and we can say `const user = cy`, because I think of this as a user being instructed on what things to do.

We write out our commands. Then, the `user` goes to execute them. None of this is happening synchronously. The user can't do two things at once or can't do two things in quick succession.

Everything in cy's world is happening asynchronously.

It's not actually clicking on an element when you call `.click`. You're just queuing this command up in the list of actions that you want the user or Cypress to execute.

That's an important distinction to remember when you're using Cypress is that, each one of these commands that you're telling Cypress to do are going to happen in the future and are not happening right now. Just think of these as a list of commands that you're giving to a user to manually test your application.

Configure Cypress in cypress.json

In our Cypress test here, we call `cy.visit`. We have to give the full URL for our application.

calculator.js

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('http://localhost:8080')
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(4)')
      .click()
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(4)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(5)')
      .click()
      .get('.mNQM6vIr72uG0YPP56ow5')
      .should('have.text', '3')
  })
})
```

I don't really like that a whole lot. This actually causes another problem. If I pop open this, you'll notice that this is going to refresh.

It loads the application and then it refreshes. That's because when Cypress realizes that this is where we're going to go, it has to refresh so that it goes to that same place as well. This is a technical limitation of Cypress. It's pretty easy to overcome.

Let's go ahead and solve both problems. I want to go to `/`, to the root of my application in my test.

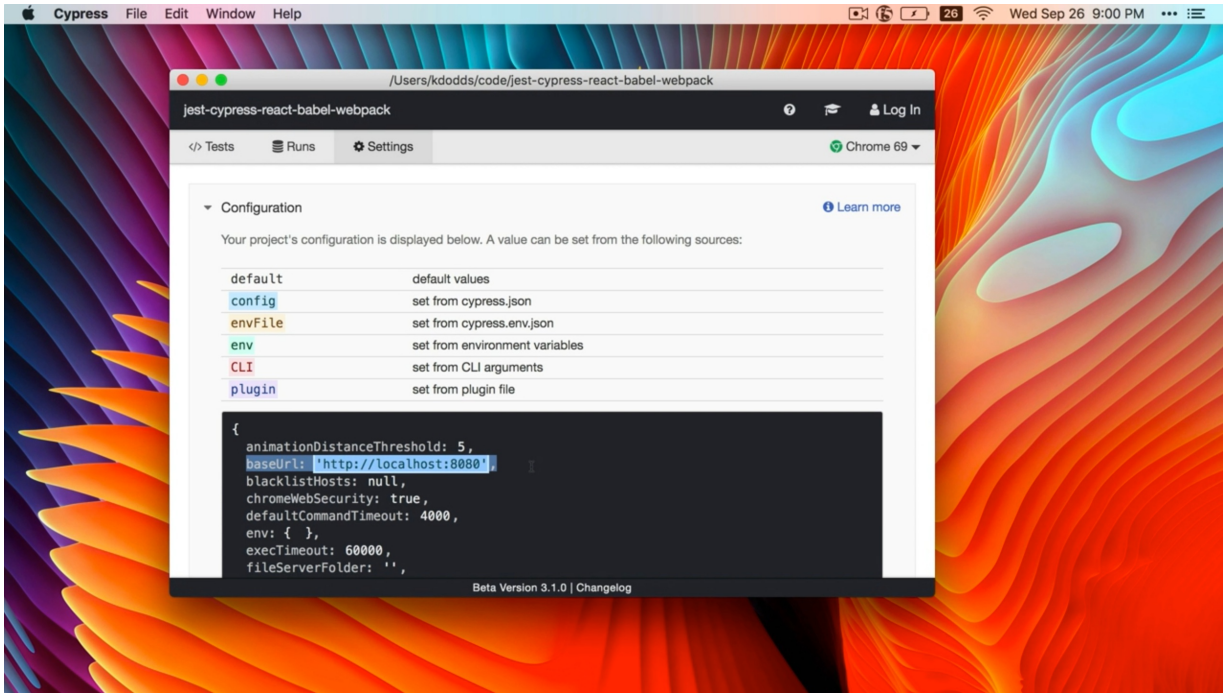
```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
```

In my `cypress.json` is where I'm going to configure where the `baseUrl` for my application is. I'll say, `baseUrl` That'll be `http://localhost:8080`.

cypress.json

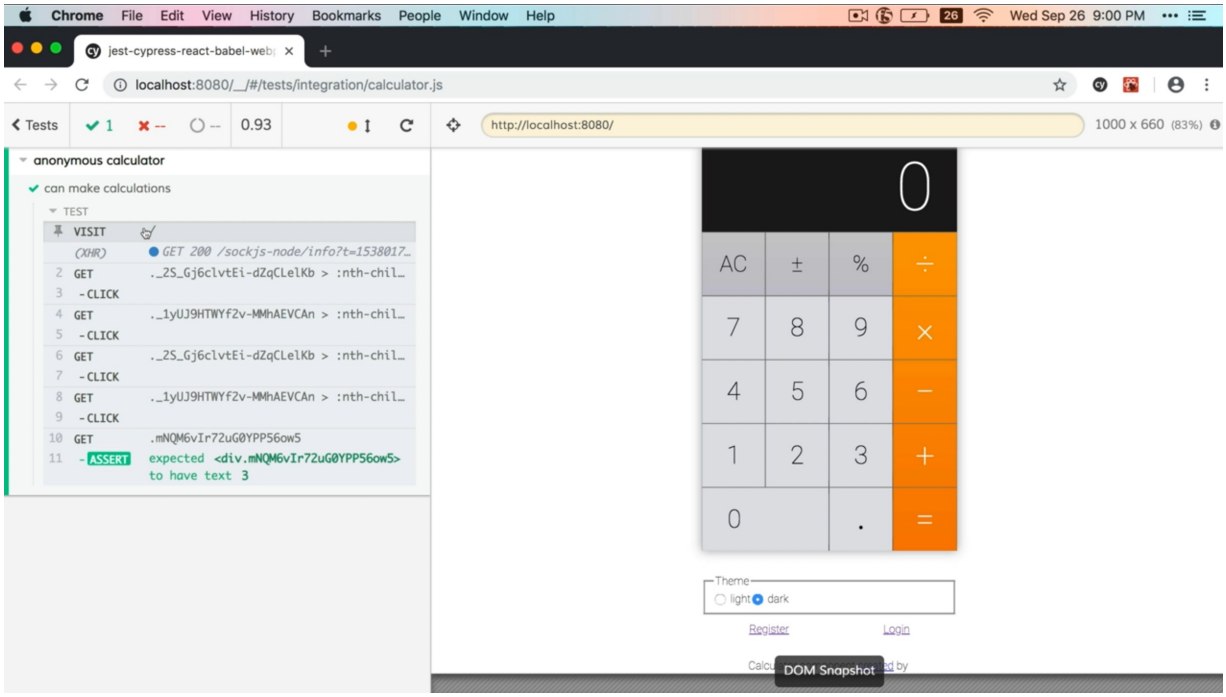
```
{
  "baseUrl": "http://localhost:8080"
}
```

With that, Cypress will actually reload because we've changed the configuration. It'll close down Chrome, reload the app. Now our configuration has a `baseUrl` which has been set from the config file.



There's also an `env` file which you can use to configure Cypress. You can use environment variables, `CLI` arguments, when we open up Cypress in the first place, as well as a `plugin` file. We're going to use this `cypress.json`.

Let's go to our test now and open up the calculator. You'll notice it does not refresh this time. It starts a bit quicker. We also don't have to give the full URL for when we issue a visit command.



The next thing that I want to configure with Cypress is I'm not a huge fan of this `integration` folder. I'd much rather communicate that the tests inside of here are end-to-end tests. I'll do my integration tests with react-testing-library and Jest.

I'm going to rename this to `e2e`, for end-to-end. Here I'm going to set my `integrationFolder` to `cypress/e2e`.

e2e/cypress.json

```
{
  "baseUrl": "http://localhost:8080",
  "integrationFolder": "cypress/e2e"
}
```

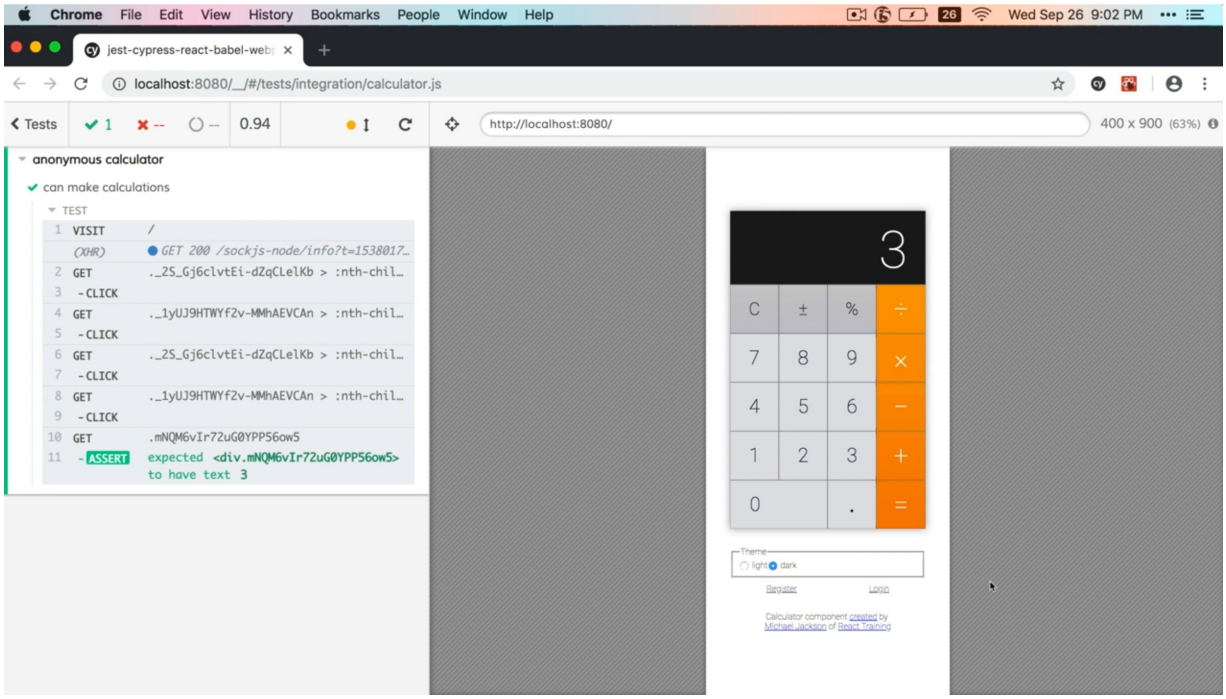
With that, it's going to reload everything. Our tests still work just fine.

The last thing that I want to configure is the viewport. You'll notice the default is 1000x660. Cypress will make sure that our viewport stays the same. It'll just scale down depending on the width of our window so we don't break any tests by changing the size of this window.

For my test, I want to change that default viewport. Let's go ahead and we'll set the `viewportHeight` to `900` and the `viewportWidth` to `400`.

```
{
  "baseUrl": "http://localhost:8080",
  "integrationFolder": "cypress/e2e",
  "viewportHeight": 900,
  "viewportWidth": 400
}
```

It'll reset everything. I can open it up again. Now it's the viewport that I want.



In review, what we did here is we set the `baseUrl` in our `cypress.json` so that we didn't have to provide the full URL when we issue a visit command. Cypress knows ahead of time which URL it should be loading when it starts up.

Then we moved this test to an `e2e` folder, rather than an integration folder, to communicate more clearly what kinds of tests should live in here. We configured that with `integrationFolder`.

calculator.js

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(4)')
      .click()
      .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(4)')
      .click()
      .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(5)')
      .click()
      .get('.mNQM6vIr72uG0YPP56ow5')
      .should('have.text', '3')
  })
})
```

Then we set the `viewportHeight` and `viewportWidth`, so it can match more accurately the type of experience that we're trying to build with this application.

cypress.json

```
{
  "baseUrl": "http://localhost:8080",
  "integrationFolder": "cypress/e2e",
  "viewportHeight": 900,
  "viewportWidth": 400
}
```

Installing cypress-testing-library

I really don't like these selectors.

calculator.js

```
describe('anonymous calculator', () => {
it('can make calculations', () => {
  cy.visit('/')
    .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(3)')
    .click()
    .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(4)')
    .click()
    .get('._2S_Gj6clvtEi-dZqCLelKb > :nth-
child(4)')
    .click()
    .get('._1yUJ9HTWYf2v-MMhAEVCAn > :nth-
child(5)')
    .click()
    .get('.mNQM6vIr72uG0YPP56ow5')
    .should('have.text', '3')
  })
})
```

They're pretty confusing. I can look at things, and I guess this is `1`, and then this is `+`, and then this is `2`, that's `=`, and that's the `display`, but that's pretty hard to read, and I would definitely not want to maintain a test that looked like this for sure.

We're going to install `cypress-testing-library` that will make our selectors a lot better. I'm going to `npm install --save-dev cypress-testing-library`, and once we get that installed into

our `package.json` right here, then we can go ahead and use that.

Terminal Input

```
npm install --save-dev cypress-testing-library
```

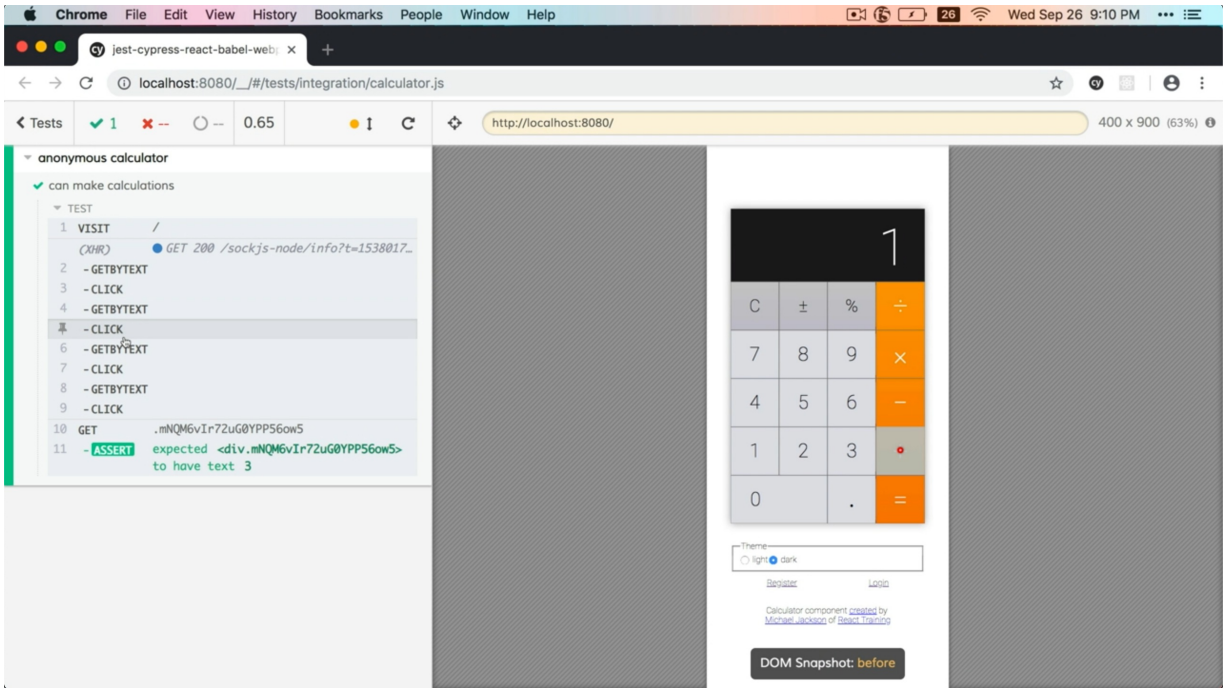
I'm going to put that in `support/index.js` here, and we'll just `import cypress-testing-library/add-commands`. This is going to add custom commands to Cypress that we can use in our tests. With that added, we can now use `getByText`, and if you're familiar with `react-testing-library`, you'll be pretty comfortable here.

Here we'll enter a regex that matches `1`, and then we'll click on `1`, and then we'll go ahead and find the text that matches `+`, and then we'll click on `+`. Then we'll find the text that matches `2`, we'll click on `2`, find the text that matches `=`, and click on `=`, and then we define the total amount.

calculator.js

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
      .getByText(/^1$/)
      .click()
      .getByText(/^\+$/)
      .click()
      .getByText(/^2$/)
      .click()
      .getByText(/^=$/)
      .click()
      get('.mNQM6vIr72uG0YPP56ow5')
      .should('have.text', '3')
  })
})
```

Let's go ahead and leave that for a second. Let's make sure that this is going to work. If we save that, it runs again really quickly, and we get all of the elements that we need.



How are we going to find this display here? If we use our selector playground, the only thing that it has on it that's really identifiable is the `className`, and that's not super helpful for us.

What we're going to do, I'm going to open up where that is rendered, that's `auto-scaling-text`, and right here I'm going to add a `data-testid` for a `total`, and then I can say `getByTestId('total')`.

shared/auto-scaling-text.js

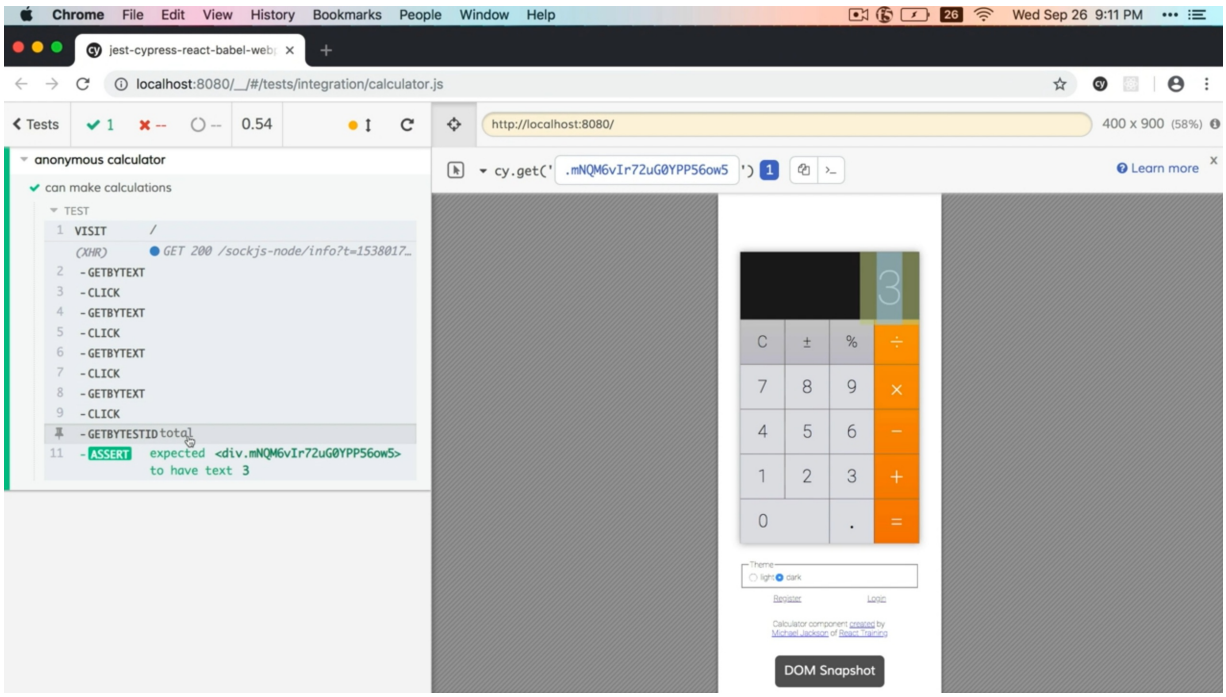
```
render() {
  const scale = this.getScale()

  return (
    <div
      className={styles.autoScalingText}
      style={{transform:
`scale(${scale},${scale})`}}
      ref={this.node}
      data-testid="total"
    >
      {this.props.children}
    </div>
  )
}
```

calculator.js

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
      .getByText(/^1$/)
      .click()
      .getByText(/^\+$/)
      .click()
      .getByText(/^2$/)
      .click()
      .getByText(/^=$/)
      .click()
      .getByTestId('total')
      .should('have.text', '3')
  })
})
```

With that we'll refresh here and it's working great.



`getByTestId('total')`, and it expected that div to have the text of `3`, and this is so much easier to read and understand what's going on.

In review, to make all of this work, we first had to install `cypress-testing-library` in our `package.json`, and then we had to apply the commands that come from `cypress-testing-library`, so we `import cypress-testing-library/add-commands`. All that's really doing is it goes through all of the commands that Cypress has, and for each one of those it's going to add that command by its name to Cypress commands.

With that done, we can go into `e2e\calculator.js` and update these to use these custom commands that come from `cypress-testing-library` like `getByText` and `getByTestId`. This makes our tests a lot easier to both write and read.

calculator.js

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
      .getByText(/^1$/)
      .click()
      .getByText(/^\+$/)
      .click()
      .getByText(/^2$/)
      .click()
      .getByText(/^=$/)
      .click()
      .getByTestId('total')
      .should('have.text', '3')
  })
})
```

Scripting Cypress for local development and Continuous Integration

To run Cypress, we have to first run `npm run dev`, and then in another tab, we have to run `npx cypress open` to start Cypress up.

It would be great if we could have a single script that could start up Cypress and our dev server, so we only have one script to run. For our continuous integration, we also should have one script that will run both as well.

What we're going to do is I'm going to `npm install --save-dev start-server-and-test`.

Terminal Input

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

With that installed in our `devDependencies` here in our `package.json, we can go ahead and use this. I'm going to create a couple of scripts here.

First, we'll make a `cy:run`. That's going to be `cypress run`. That will run Cypress in headless mode, so you won't actually see the browser or the Cypress application running. This is for CI.

package.json

```
"scripts": {
    ...
    "cy:run": "cypress run",
}
```

Then we'll have a `cy:open` that will run `cypress open`.

```
"scripts": {
    "cy:run": "cypress run",
    "cy:open": "cypress open",
}
```

Then we'll have a `test:e2e`. For this one, we're going to do a similar thing that we did up here with the `test`, where we'll use the `is-ci`.

We'll say run `test:e2e:run` for CI, and `test:e2e:dev` for local.

```
"scripts": {
    "cy:run": "cypress run",
    "cy:open": "cypress open",
    "test:e2e": "is-ci \"test:e2e:run\"
\"test:e2e:dev\"",
}
```

Then for `test:e2e:run`, we'll use `start-server-and-test start`. What `start-sever-and-test` is going to do is it will run this command, and then it will wait for our server to start responding.

We'll say `http://localhost:8080`. Once this starts responding to requests, then it will run this script, `cy:run`.

```
"scripts": {
    "cy:run": "cypress run",
    "cy:open": "cypress open",
    "test:e2e": "is-ci \"test:e2e:run\"
\"test:e2e:dev\"",
    "test:e2e:run": "start-server-and-test start
http://localhost:8080 cy:run",
}
```

We'll do a similar thing for `dev`. We'll say `cy:dev` and instead of `start`, we'll have it run the `dev` script.

```
"scripts": {
    "cy:run": "cypress run",
    "cy:open": "cypress open",
    "test:e2e": "is-ci \"test:e2e:run\"
\"test:e2e:dev\"",
    "test:e2e:run": "start-server-and-test start
http://localhost:8080 cy:run",
    "test:e2e:dev": "start-server-and-test dev
http://localhost:8080 cy:dev",
}
```

For our `dev` script, as we're making changes, our webpack configuration will reload the app and we can rerun the test, then it will have our most recent changes. For our `run` script, it's running this `start` script. That's going to run the built code.

Before we run our end-to-end test, we should make sure that we're building the code. What I'm going to do is I'll also add a `pretest:e2e:run`.

This script, because it has the `pre` prefix will run the

`test:e2e:run` script, and before we run the `test:e2e:run` script, we're going to run `npm run build`. That way, when we do end up running the script, it's using the latest of our code changes.

```
"scripts": {
    "cy:run": "cypress run",
    "cy:open": "cypress open",
    "test:e2e": "is-ci \"test:e2e:run\"
\"test:e2e:dev\"",
    "pretest:e2e:run": "npm run build",
    "test:e2e:run": "start-server-and-test start
http://localhost:8080 cy:run",
    "test:e2e:dev": "start-server-and-test dev
http://localhost:8080 cy:dev",
}
```

One other quick thing that I'm going to do for some other scripts in `precommit`, I'm going to go ahead and add and `npm run test:e2e:run`. This way, I know that before I commit any code, I'm not breaking any of my end-to-end test.

```
"precommit": "lint-staged && npm run
test:e2e:run",
```

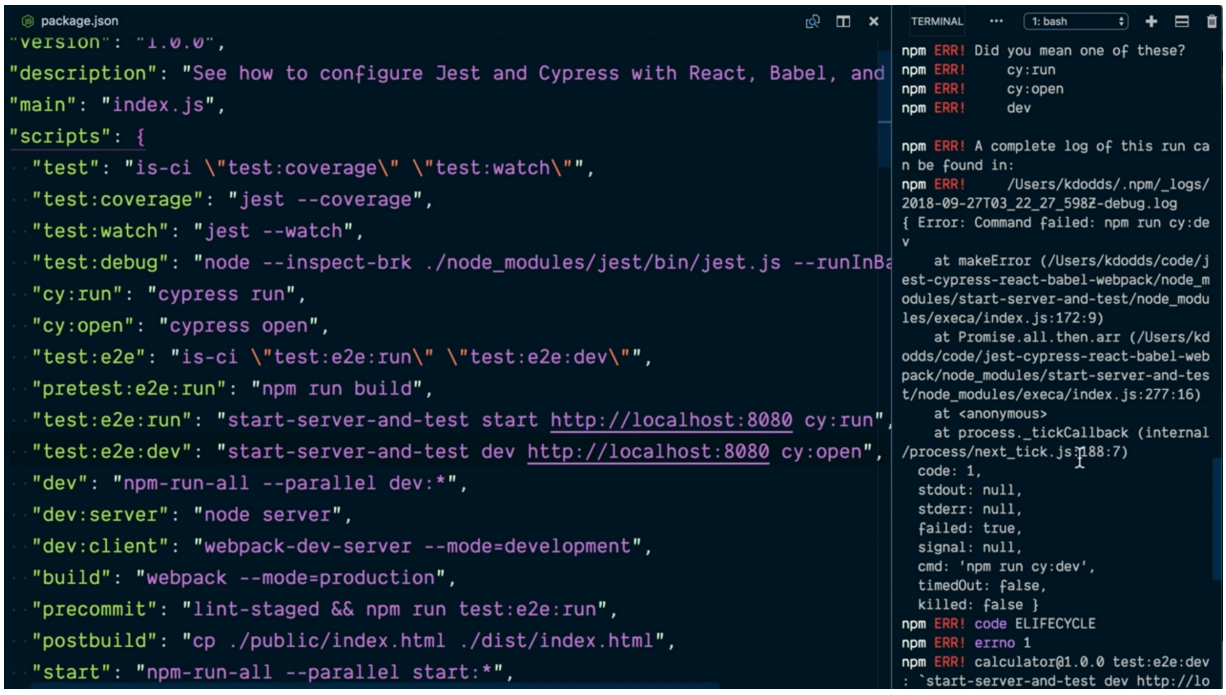
You might not want to do this if your end-to-end test takes several minutes, but for this small app, I'm going to go ahead and put it in there. If this app grows and the end-to-end test suite starts taking a little bit of time, then I'll probably remove this and rely more heavily on CI.

In any case, I definitely am going to want to have my `validate` script to run the end-to-end test, because this is what I'm running in CI. Run `npm run test:e2e:run`.

```
 "validate": "npm run test:coverage && npm run
test:e2e:run",
```

With that, now I can go ahead and run `npm run test:e2e`, and because I am local, it's going to run `test:e2e:dev`. It's running my dev server.

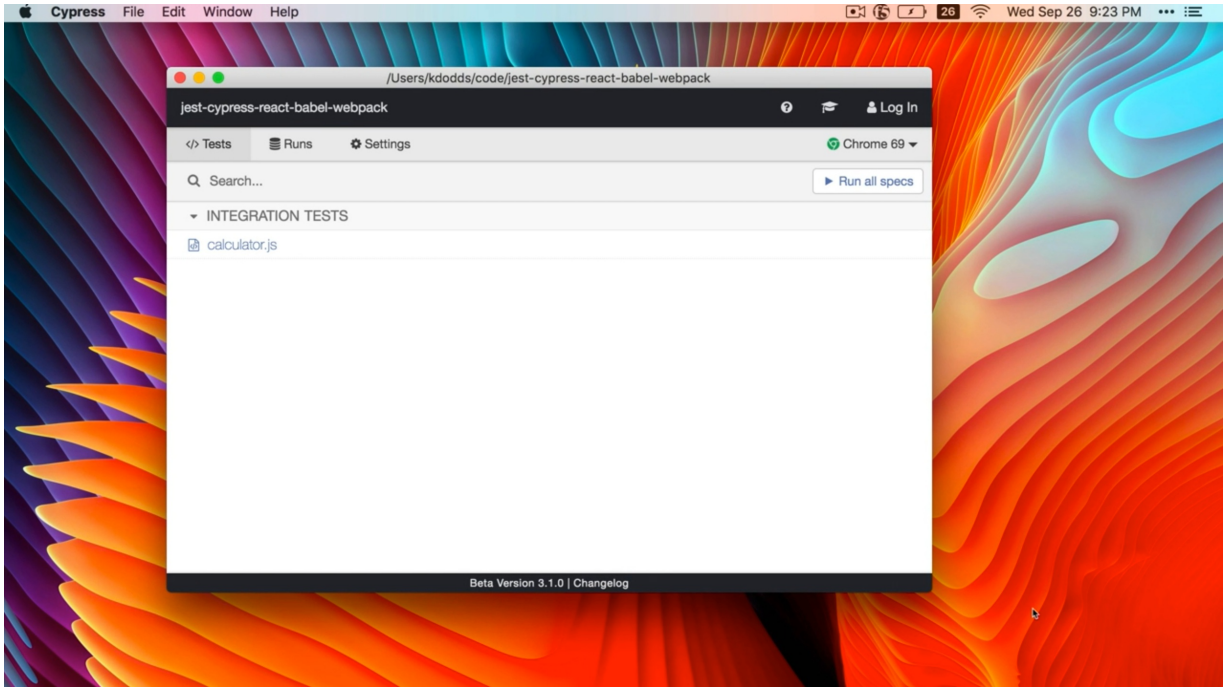
It looks like I made a little typo here.



Instead of `cy:dev`, this should be `cy:open`.

```
"test:e2e:dev": "start-server-and-test dev
http://localhost:8080 cy:open"
```

Let's go ahead and try that again. `npm run test:e2e`. It starts my webpack dev server and the backend server at the same time, and then it runs `cypress open` when that `localhost:8080` starts accepting requests.



Once that's started, we know that we can start our Cypress test app and we can run calculator, and it will run against that server that we started.

In review, what we had to do to make all of this work is we installed `start-server-and-test` and then we created a couple of scripts to run and open Cypress.

Then we created our `test:e2e` scripts here, and we used `start-server-and-test` for both `start` and `dev`, and in both cases, when `localhost:8080` is ready, it's going to run `cy:run` to run our test heedlessly with Cypress and `cy:open` to open the Cypress app.

```
"scripts": {
    "cy:run": "cypress run",
    "cy:open": "cypress open",
    "test:e2e": "is-ci \"test:e2e:run\"
\"test:e2e:dev\"",
    "pretest:e2e:run": "npm run build",
    "test:e2e:run": "start-server-and-test start
http://localhost:8080 cy:run",
    "test:e2e:dev": "start-server-and-test dev
http://localhost:8080 cy:open"
}
```

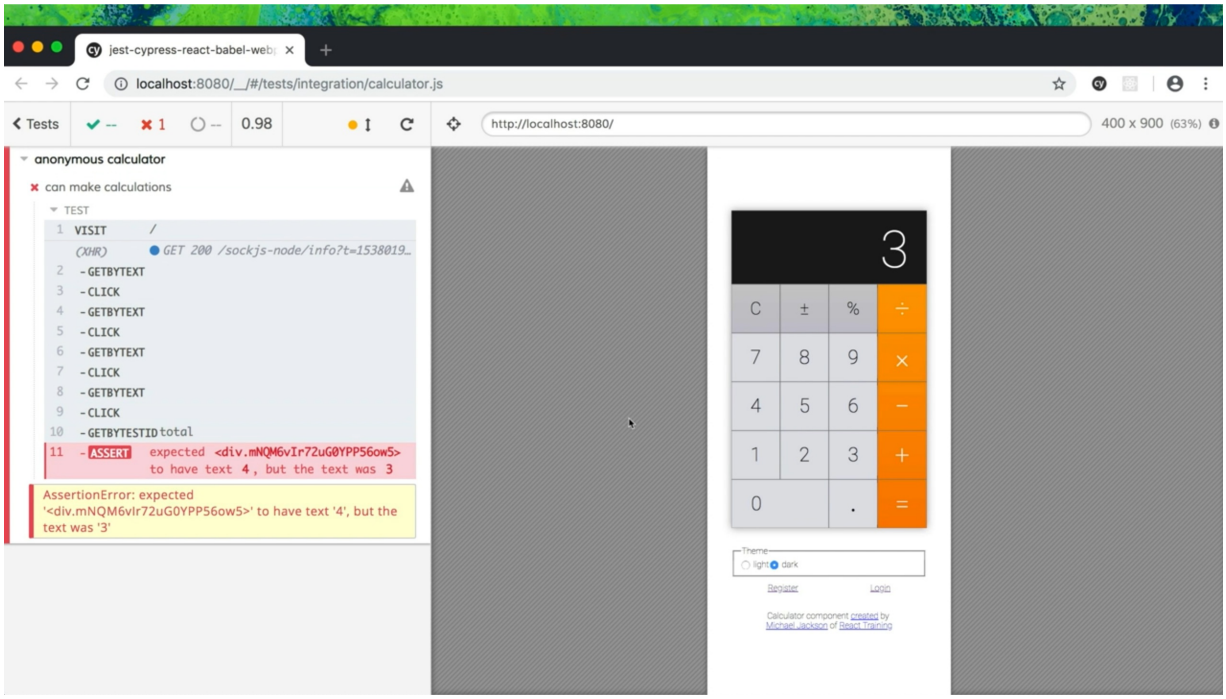
Let's go ahead and see what happens in our console now when we run `npm run test:e2e:run`. This is what it's going to look like in CI. First, it's going to run our build, because we have that `pretest:e2e:run` there. Once the build is finished, it's going to start our server with the `start` script.

Now that the server is starting to take traffic, it's going to run `cypress run`, and cypress runs all the tests. Once Cypress is finished, then it's going to close down our server and our script executes with an exit code of 0 so it was successful.

If we were to break one of our tests, then our script would fail, and CI would fail.

Debug a test with Cypress

Let's say that we have a bug here, where we expect this to equal `4` when we added `1` and `2`, but it actually equals `3`. We open up our cypress test and this thing's totally broken.

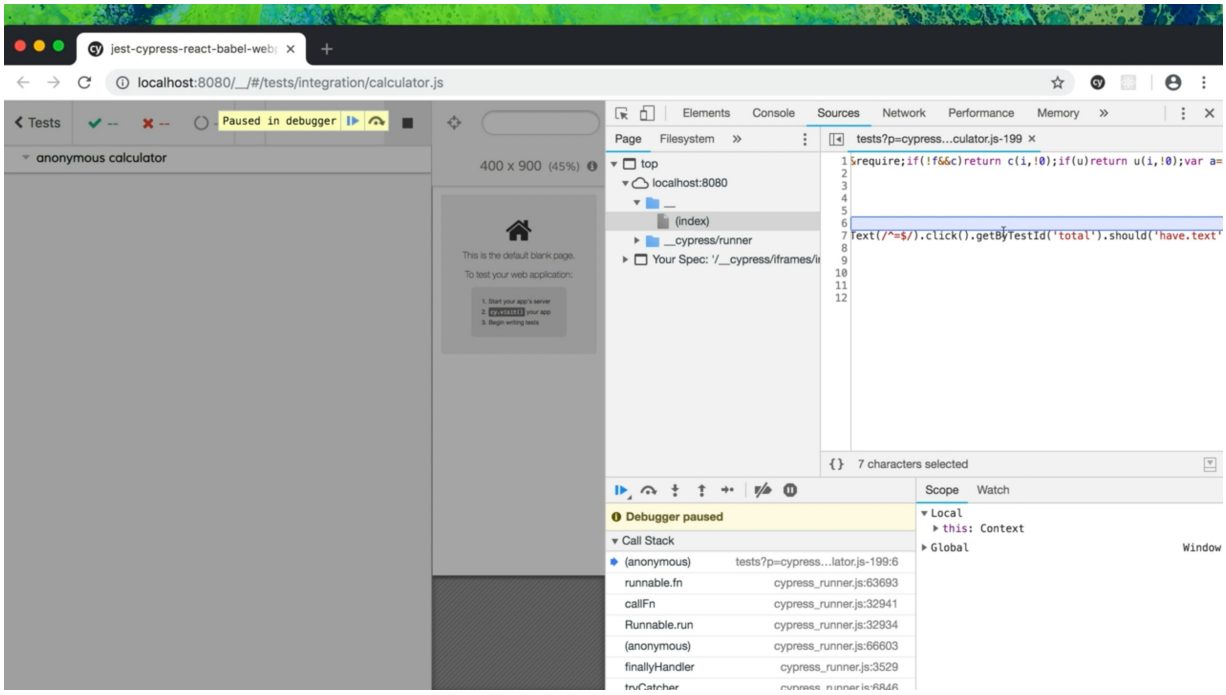


How do we go about debugging this? Maybe clicking on the wrong button or something is going on wrong. What do we do to debug this? What we would traditionally do is, we'd add a `debugger` statement here and cypress will rerun our test. We can actually open up our developer tools.

calculator.js

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    debugger
    cy.visit('/')
      .getByText(/^1$/)
      .click()
      .getByText(/^\+$/)
      .click()
      .getByText(/^2$/)
      .click()
      .getByText(/^=$/)
      .click()
      .getByTestId('total')
      .should('have.text', '3')
  })
})
```

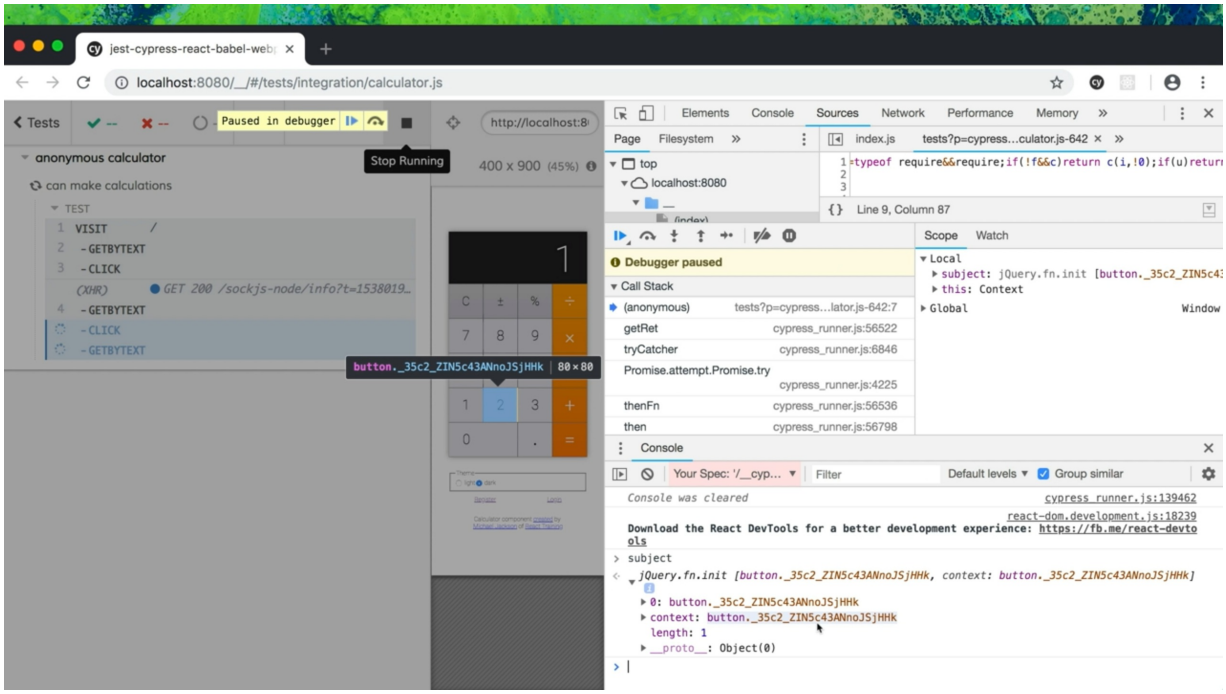
When we have that opened up and we can look in that `debugger` statement, and we can inspect the world at this point in time. The problem is that we're queuing up cypress to execute these commands that will happen at a later time.



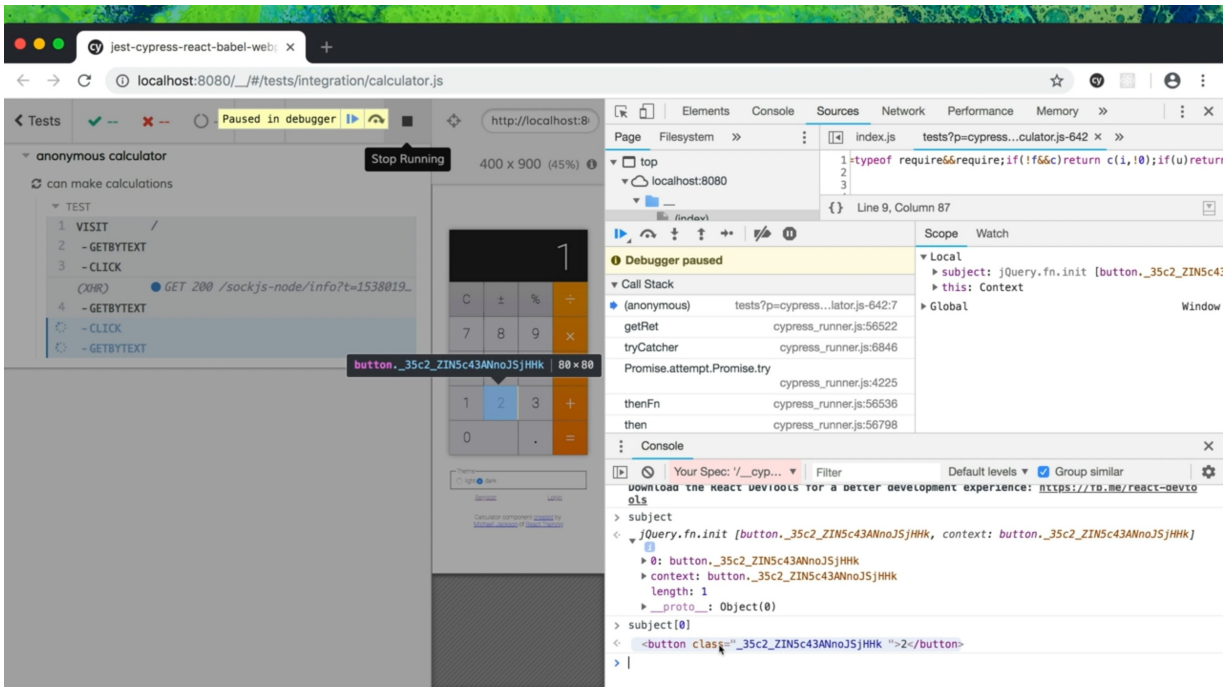
If we wanted to know what this subject that were clicking on here is, then we have to do a little bit more work. What we're going to do is we'll add a `.then`. This is going to give us our `subject`. Then, we'll have to `return` the `subject` to keep the chain going, but then we can add our `debugger` right here.

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
      .getByText(/^1$/)
      .click()
      .getByText(/^\+$/)
      .click()
      .getByText(/^2$/)
      .then(subject => {
          debugger
          return subject
      })
      .click()
      .getByText(/^=$/)
      .click()
      .getByTestId('total')
      .should('have.text', '4')
  })
})
```

If I save that and we'll play this through, then we're going to get our `debugger` statement at this point. We have our `subject`. We can open up our console here. We'll look in that `subject` variable. We'll see it's a jQuery wrapped button.



If we say `subject[0]` that's our `button` node, then we can see exactly what node is being interacted with at this point in time.



Executing a `.then` here, which is promise like, but it's not exactly the same thing as a promise, so you can't do `async await`.

But using `.then` here to say, "Hey cypress, after you've gotten this `subject`, I want to inspect that, and make sure to return the `subject`, so that the test can continue with that `subject` as if you

didn't do anything to change that `subject`. Then, you can put your `debugger` statement and inspect what's going on at this point in time."

We can move this around at any part in or test here to inspect what the `subject` is at any point in time. That can be really, really helpful in debugging our tests.

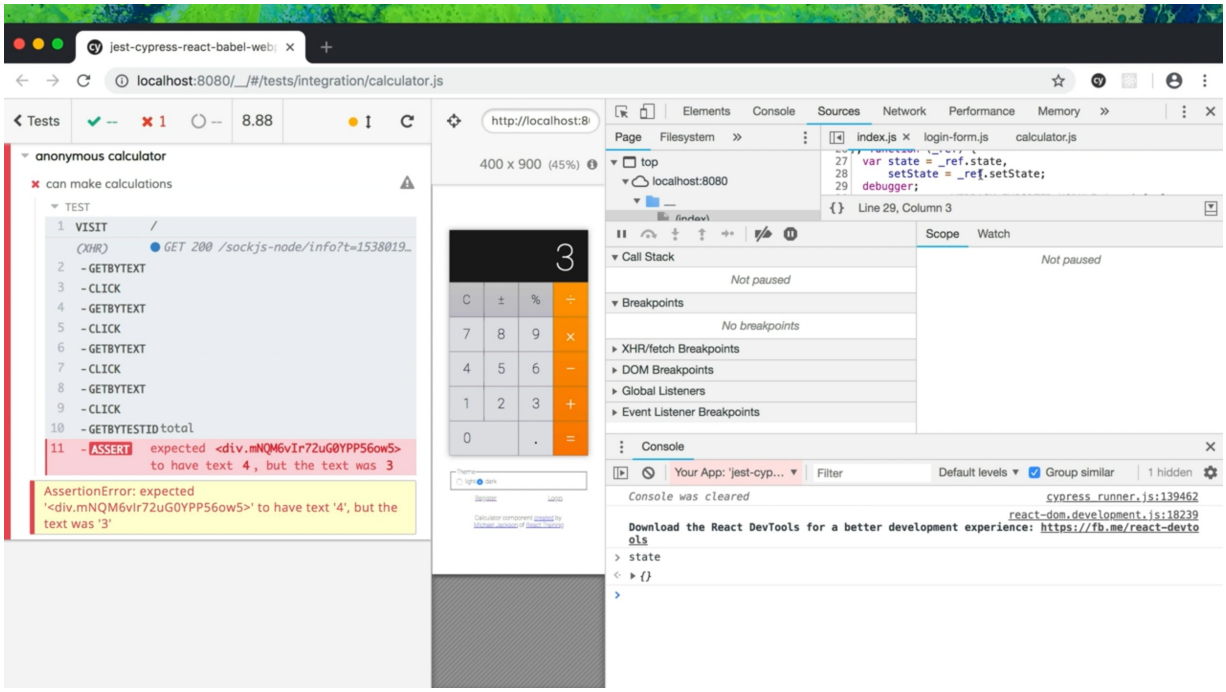
```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
      .getByText(/^1$/)
      .click()
      .getByText(/^\+$/)
      .click()
      .getByText(/^2$/)
      .click()
      .getByText(/^=$/)
      .click()
      .then(subject => {
          debugger
          return subject
      })
      .getByTestId('total')
      .should('have.text', '4')
  })
})
```

If we wanted to debug some code that's in our source code, then we can actually do the same thing. If I wanted to, for example, going to this `index.js` and I wanted to debug here, then I can do that.

index.js

```
ReactDOM.render(
  <Component initialState={{}}>
    {({state, setState}) => {
      debugger
      return (
        <LoadUser
        user={state.user}
        setUser={loadedUser => setState({user:
loadedUser})}
      >
```

I'll go and play this through. I get my `debugger` statement right here which is perfect. I can look at the `state`. I see the `state` is an empty object or play through, and the things are still broken.



What if I wanted to see what the `state` was here now, when we don't have access to the `state`? I'm going to do a little trick here. Anywhere inside of your source code, you can actually do

`if(window.Cypress){` and anything inside of this `if` statement will actually execute when we're running in our cypress test, but not when your application is actually running.

So we can add `window.appState = state` and `window.setAppState = setState`.

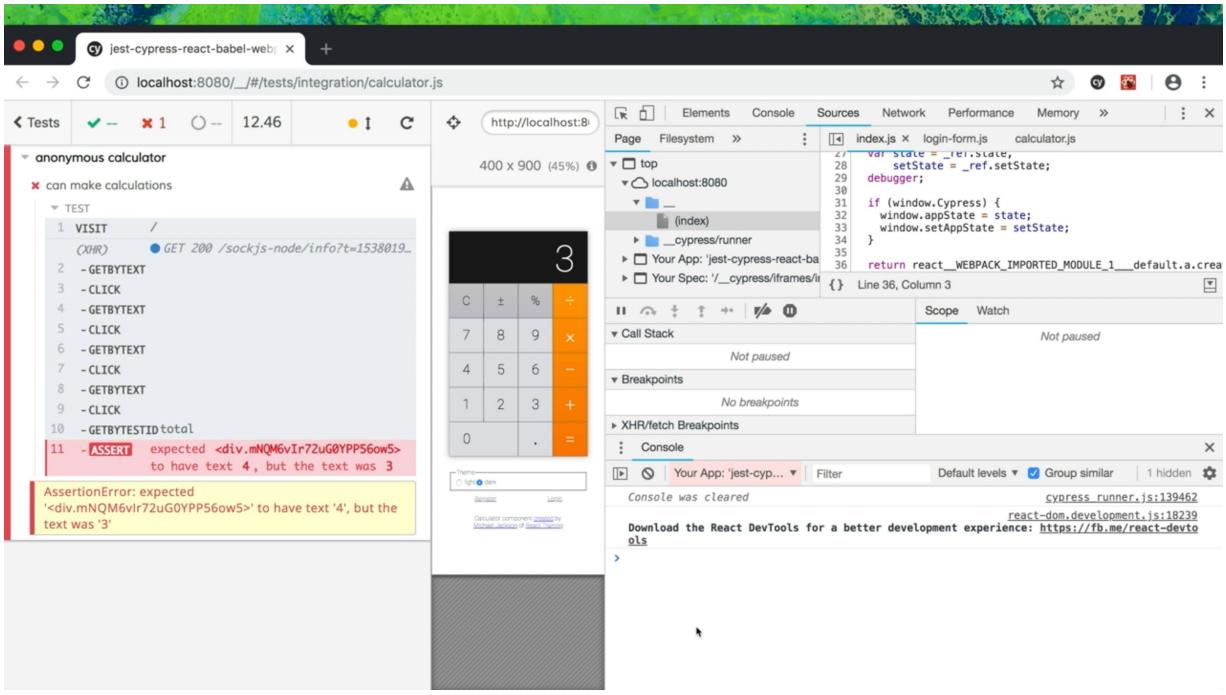
```
ReactDOM.render(
  <Component initialState={{}}>
    {({state, setState}) => {
      if(window.Cypress) {
        window.appState = state
        window.setAppState = setState
      }
```

You could do this with your redux store, or just about anything you can think of to provide some better debugging tools for your cypress test. I'll go ahead and save that.

Just to see what's going on, we'll add a `debugger` right here.

```
ReactDOM.render(
  <Component initialState={{}}>
    {({state, setState}) => {
      debugger
      if(window.Cypress) {
        window.appState = state
        window.setAppState = setState
      }
```

Then, let's go ahead and open our console. We'll refresh, and we'll get that stopped right there. We'll step over. We're inside of this `if` statement setting `appState` and this `setAppState`.



We'll play through. We can look at `window.appState`. We'll see that's an empty object and we can call `setAppState`. We can call that with a `user` with `name` of `someone`.

Chrome Console

```
window.appState
> {}
window.setAppState({user: {name: 'someone'}})
```

That's going to rerender our app, so we'll hit that debugger again. Look right here, we have `register` and `log in`.

When I click play through, now that we have a `user`, we have `log out`. We can debug a lot of things. If you were to put your redux store in here, you can do a bunch of things and set your app into

a certain state to make things a lot easier.

In fact, we could also use this in our cypress test as well to access some variables if that's at all helpful.

In review, to debug your test with cypress, you can add a `.then`, you'll get access to the subjects. You can add a `debugger` there.

calculator.js

```
describe('anonymous calculator', () => {
  it('can make calculations', () => {
    cy.visit('/')
      .getByText(/^1$/)
      .click()
      .getByText(/^\+$/)
      .click()
      .getByText(/^2$/)
      .click()
      .getByText(/^=$/)
      .click()
      .then(subject => {
          debugger
          return subject
      })
      .getByTestId('total')
      .should('have.text', '4')
  })
})
```

If you wanted to debug in your source code, you add a `debugger` anywhere in your source code and that will also be stopped in your developer tools.

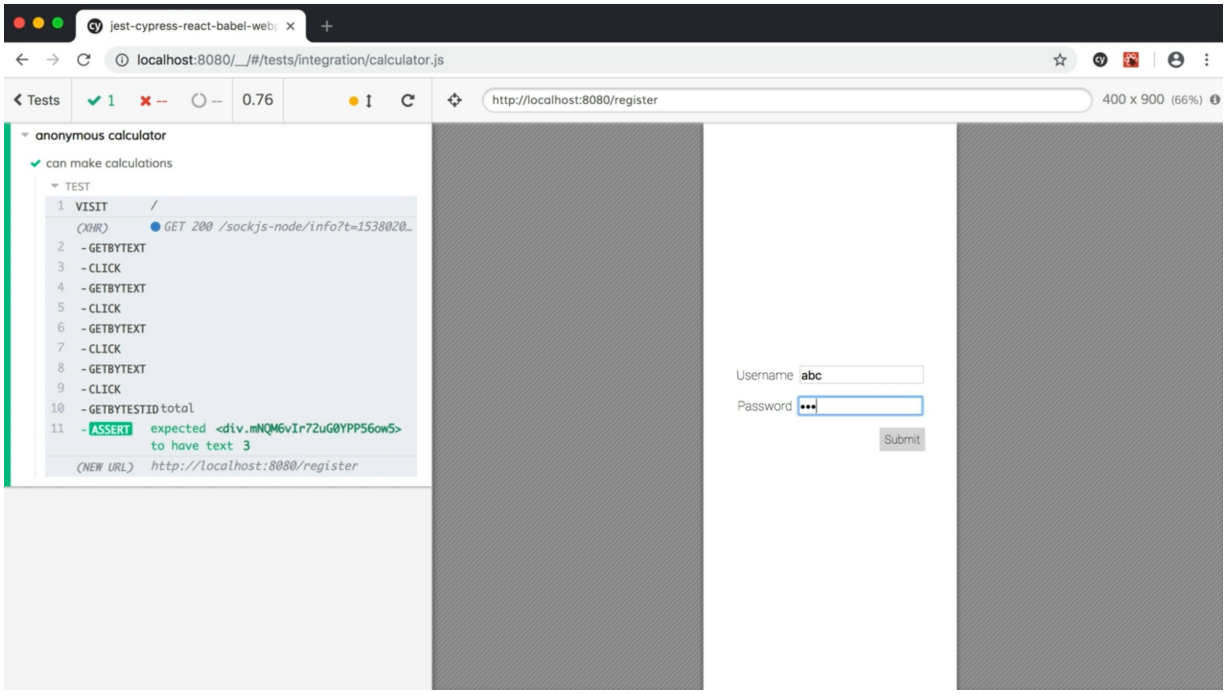
You can also run some code that's specific to cypress to expose some internal state of your application to make it easier to develop and debug your cypress test.

index.js

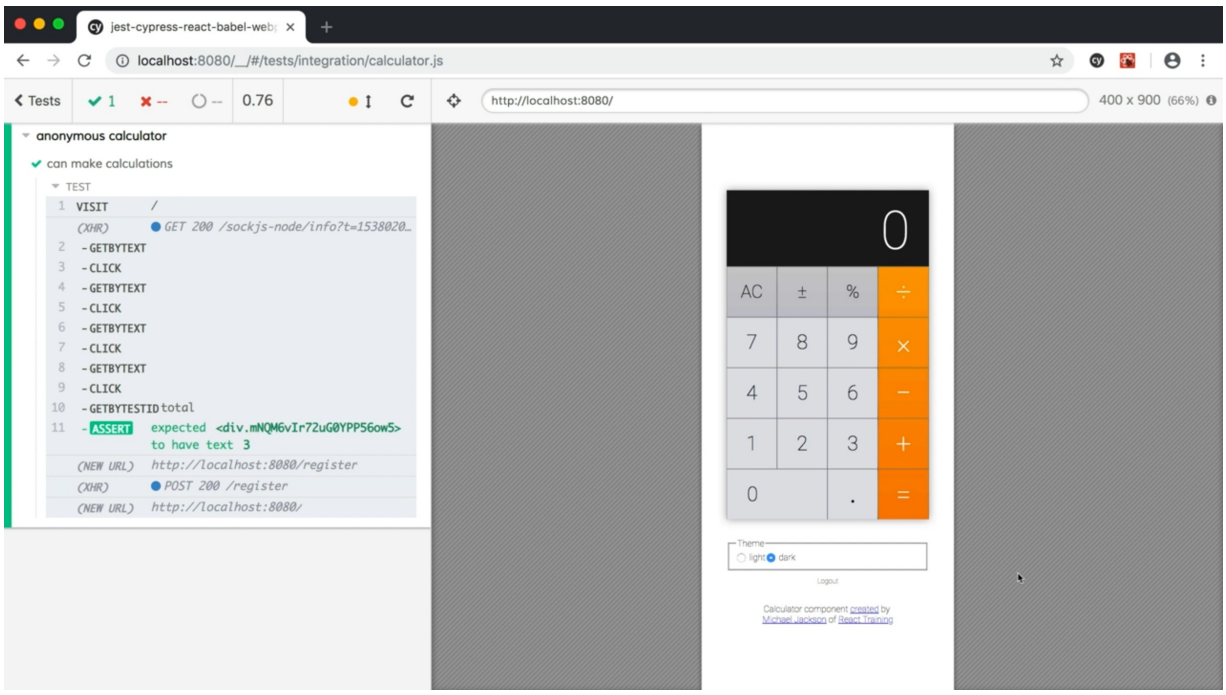
```
ReactDOM.render(
  <Component initialState={{}}>
    {({state, setState}) => {
      debugger
      if(window.Cypress) {
        window.appState = state
        window.setAppState = setState
      }
```

Use Cypress to test user registration

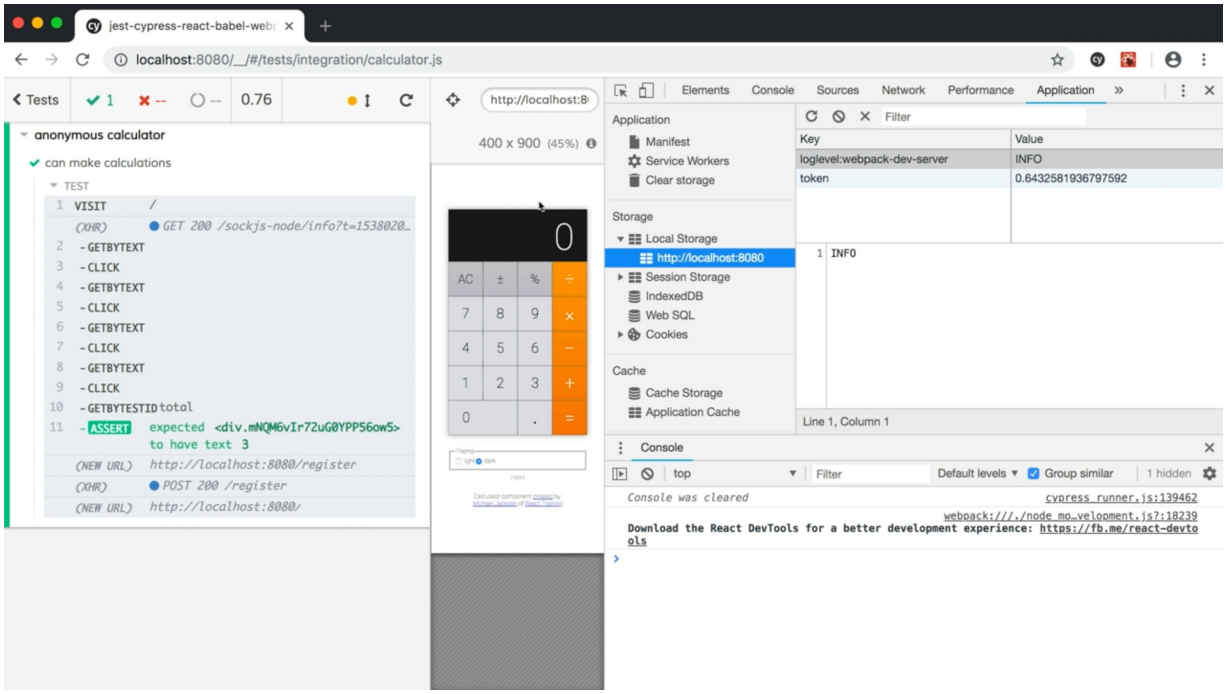
In our application, we have this little register link that we can use to register as a user. I'll say, `ABC` and a password, `123` I'll submit that.



Now I have this logout button I can click on to log out of the application.



There is also some local storage set in my application here. If I go to `Local Storage`, I get my `token` right there in my local storage so that if I refresh the application as the user, I am still logged into the app.



When I log in, I'm redirected to this home page.

Let's go ahead and test this entire experience with Cypress. I'm going to create a new file in this `e2e` directory called `register.js`. Here I'll `describe registration`. I'll say, `It should register a new user`

register.js

```
describe('registration', () => {
  it('should register a new user', () => {

  })
})
```

Then we'll go ahead and create a `user` object here with a `username` of `blah` and a `password` of `Yo`.

```
describe('registration', () => {
  it('should register a new user', () => {
      const user ={username: 'blah', password:
'yo'}
  })
})
```

Actually, that's not going to work very well. If we run this test more than one time, we're going to have a problem where we're registering a `user` that actually already exists. We're going to need to generate this information.

I'm going to go ahead and in this `support` directory, I'm going to make a `generate.js` file. Here we're going to `import {build, fake} from 'test-data-bot'`, which we already have installed

in this project.

generate.js

```
import {build, fake} from 'test-data-bot'
```

I'm going to make a `userBuilder`. We'll `build('User')`. That'll have `.fields` of `username`, which will be `fake`, using `f.internet.userName()`. `password` will be `fake(f => f.internet.password())`.

Then we'll go ahead and `export {userBuilder}`.

generate.js

```
import {build, fake} from 'test-data-bot'

const userBuilder = build('User').fields({
  username: fake(f => f.internet.userName()),
  password: fake(f => f.internet.password()),
})

export {userBuilder}
```

With that, let's go ahead and `import {userBuilder} from '../support/generate'`.

register.js

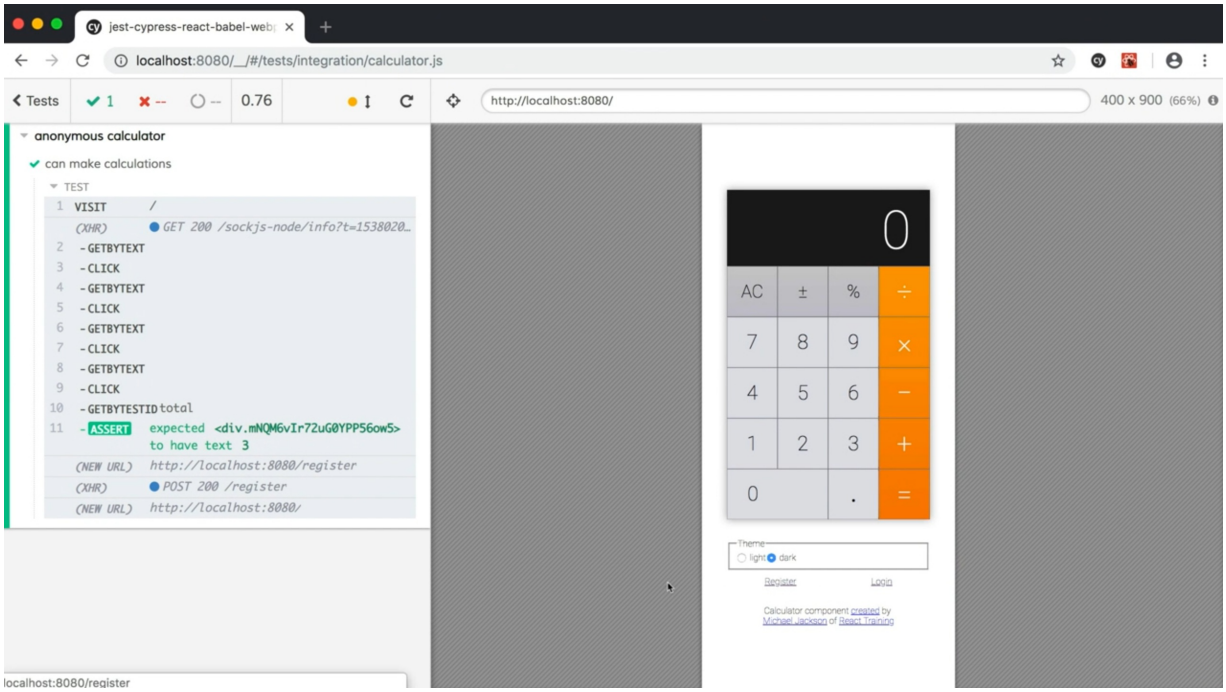
```
import {userBuilder} from '../support/generate'
```

Cool. Now this user can be unique every single time we run this `userBuilder`. Then we'll do `cy.visit('/')`.

```
import {userBuilder} from '../support/generate'

describe('registration', () => {
  it('should register a new user', () => {
    const user = userBuilder()
    cy.visit('/')
  })
})
```

Then the first thing that we're going to want to do is we'll click on this register link.



We'll say, `getByText(/register/i)` We'll ignore case there. Case doesn't matter. We'll `click` on it.

```
    cy.visit('/')
       .getByText(/register/i)
       .click()
```

Go ahead and do that. We'll `getByLabelText(/username/)`.

We want to `type` the value from `user.username`.

```
     cy.visit('/')
       .getByText(/register/i)
       .click()
       .getByLabelText(/username/i)
       .type(user.username)
```

Then we'll `getByLabelText(/password/i)`. We'll `type` the `user.password`.

```
     cy.visit('/')
       .getByText(/register/i)
       .click()
       .getByLabelText(/username/i)
       .type(user.username)
       .getByLabelText(/password/i)
       .type(user.password)
```

Then we'll `getByText(/submit/i)`. We've got that submit button. We'll go ahead and `click` on that.

```
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
```

Then we want to make some assertions. I'm going to change the subject to our `url`. We can make an assumption that it `should` equal, `'http://localhost:8080'`.

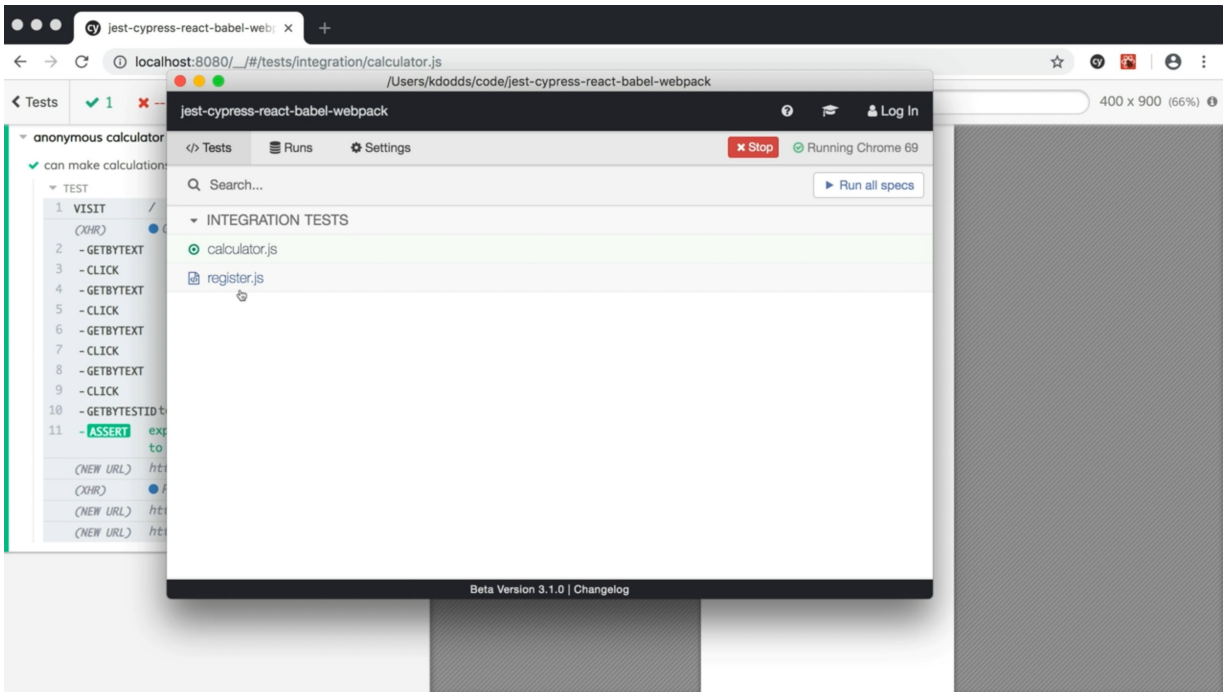
```
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      .url()
      .should('eq', `http://localhost:8080`)
```

Then we'll switch our subject to `window`, so we can verify that `.its('localStorage.token')` value `.should('be.a', 'string')`.

```
        .click()
        .getByLabelText(/username/i)
        .type(user.username)
        .getByLabelText(/password/i)
        .type(user.password)
        .getByText(/submit/i)
        .click()
        .url()
        .should('eq',
`${Cypress.config().baseUrl}/`)
        .window()
        .its('localStorage.token')
        .should('be.a', 'string')
```

We don't really need to worry about verifying what the `token` actually is. I will be pretty satisfied that there's just a `token` in there that's a string. Let's go ahead and save that.

We'll go back to our test. We have this new `register.js` file.



Let's run this one instead. It'll pull open Chrome. It'll run through our test. It runs pretty quick. Let's watch that again. Every time, that `user` is going to be unique.

Let's go ahead and refactor some of this stuff. We can actually get rid of all of this by doing Cypress.config. This will get all of our config `baseUrl`. Then we'll add the slash at the end there.

```
.url()
.should('eq', `${Cypress.config().baseUrl}/`)
```

Now if we make changes to our `baseUrl`, we don't have to update this test, which is great. Everything else, I feel pretty good about.

register.js

```
describe('registration', () => {
  it('should register a new user', () => {
    const user = userBuilder()
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      .url()
      .should('eq',
`${Cypress.config().baseUrl}/`)
      .window()
      .its('localStorage.token')
      .should('be.a', 'string')
  })
})
```

In review, what we had to do here is we created this `generate.js` file that can generate us a `user` that has a `username` and a `password`. Then, in `register.js`, we use that to create a `user`.

generate.js

```
import {build, fake} from 'test-data-bot'

const userBuilder = build('User').fields({
  username: fake(f => f.internet.userName()),
  password: fake(f => f.internet.password()),
})

export {userBuilder}
```

Then we visit our app. We `click` on `register`. We fill in the `username`, fill in the `password`. We `click` on the `submit` button. Then we verify that the `url` is our home route and that the window's `localStorage.token` is a `string` value.

I should add that if your app requires email validation when a new user registers, that it would probably be a good idea to mock that with some sort of service so that you can circumvent the email process. That would be a pretty hard thing to actually test and not provide as much value for you. It's definitely worth mocking for that scenario.

Cypress Driven Development

One really cool thing about Cypress is that it allows you to get your application into a certain state really quickly and repetitively so that, as you're developing your application, you can get yourself into that state as you're making changes. This actually pairs really nicely with test driven development.

Let's go ahead and say that after the `user` has been registered and logs in, then instead of just showing this `logout`, we also want to show their `username`.

Rather than developing the feature and refreshing the browser every single time and going through the whole registration process, we can just let Cypress go through that registration process for us, and then we can assert that our feature's done before it actually is. Let's go ahead and do that.

I'm going to add some assertions here, with `getByTestId('username-display')`, and we'll say that `.should('have.text', user.username)`.

register.js

```
import {userBuilder} from '../support/generate'

describe('registration', () => {
  it('should register a new user', () => {
    const user = userBuilder()
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      .url()
      .should('eq',
`${Cypress.config().baseUrl}/`)
      .window()
      .its('localStorage.token')
      .should('be.a', 'string')
      .getByTestId('username-display')
      .should('have.text', user.username)
  })
})
```

Now our test is going to fail, because it's looking for that `username-display` node and it can't find it.

We can also add a `timeout` here to say if it doesn't show up within `500` milliseconds, then it probably never going to show up, and we'll get that failure a little bit sooner, so next let's go ahead and implement this feature.

```
.getByTestId('username-display', {timeout: 500})
```

I'm just building my app here, so I'm going to go into `app.js`, and right here toward the bottom is where that `Logout` button appears. We're going to not only render this `button`, but also want to render a `div` that has the user's username.

app.js

```
{this.props.user ? (
    <button type="button" onClick=
{this.props.logout}>
        Logout
    </button>
    ) : (...
```

We're going to add a React fragment right here, and we'll say a `div` with `data-testid="username-display"`, and then we'll just render `this.props.user.username`.

```
{this.props.user ? (
    <>
    <div data-testid="username-display">
        {this.props.user.username}
    </div>
    <button type="button" onClick=
{this.props.logout}>
        Logout
    </button>
    </>
```

With that, let's go and rerun all of our tests, and it worked.

This test-driven development with Cypress really enhances your development workflow, and because you have access to the developer tools right in your testing tool, there's really not a whole lot of reason to use a regular browser to develop your application.

Just use Cypress, and when you need to get the application in a certain state, you can actually automate that, and you don't have to actually commit the tests that you create when you're doing this test-driven development if they're not really all that useful or very good tests.

They definitely provide a great deal of value when you're developing your application, by letting Cypress do the hard work to get you to the state of the application that you need to to verify that your changes were successful.

In our case, we're going to go and leave this as it is, because I do like this test. I think it is useful, and we'll go and commit this. We can continue to verify that we haven't broken the behavior where the username appears when the user's been logged in.

Simulate HTTP Errors in Cypress Tests

It's great that we have this registration test here, and now, I can log out. If I `register` again and type `some user` and `whatever`, what if I click on `submit`, and there's an error on the server or something? I actually in my code, I'm going to show an error message here, but how do I simulate that in this type of environment so that I can get some coverage for that experience?

Let's go ahead and write that. I'm going to write here at the bottom, I'll add, `it(`should show an error message if there's an error registering`` It looks like I'm going to want a template literal there, so I can do an apostrophe.

register.js

```
it(`should show an error message if there's an
error registering`, () => {
})
```

Now, the first thing I'm going to do is I'll `visit register` directly.

```
it(`should show an error message if there's an
error registering`, () => {
    cy.visit('/register')
})
```

I've already tested that this clicking on `register` link is going to take me to the `register` page. I don't need to really test that again, so I'll just go directly to the `register` page.

Then I want to `.getByText(/submit/i)`. I don't really care what the values are here. In fact, we can just leave them blank, because I'm going to mocking out the request, anyway. I'm just going to go ahead and `click` on the submit button directly.

```
cy.visit('/register')
  .getByText(/submit/i)
```

I'll `click` on that submit button, and then I want to verify that there is some text that says, "There was an error. You need to try again." We'll say `getByText`, and we'll just pass a regex `/error.*try again/i` I don't really care about the particular message.

```
cy.visit('/register')
  .getByText(/submit/i)
  .click()
  .getByText(/error.*try again/i)
```

It just needs to say, "There was an error. You need to try again." If I do this, I'm going to refresh the browser here, and it's going to run my first test. Then it'll run the second test. We actually didn't see what happened, it happened so fast.

First, we visited the register page. We clicked on the submit button, and it just went forward with that. Now, probably in a real application, you're going to want to have some logic. We may have another test to verify that you can't just do what I just did.

For our situation, we want to show an error message, because there is a server-side error. What we need to do is we need to mock out the request that is made when we make a request to the server to register.

I'm going to pop open the developer tools here. If we go to `network`, and we'll clear all this, I'm going to change this to an `it.only`, so that we only run this one test, so we don't run that one. I'll save that, and our test should re-run.

```
it.only(`should show an error message if there's
an error registering`, () => {
  cy.visit('/register')
    .getByText(/submit/i)
    .click()
    .getByText(/error.*try again/i)
})
```

We can see what's going on here.

If we go just to XHRs, we have a `register`. This is our `OPTIONS` request to see whether this supports CORS, because this is a different domain. Then we have our actual `POST`, where we're actually posting a blank `username` and `password`.

Again, it really doesn't matter what those values are. What we need to do is, we need to get this `Request URL`, and we're going to mock that out.

The first thing we're going to do is we'll say `cy.server`. We're indicating to Cypress that we want to start up a mock server.

```
it.only(`should show an error message if there's
an error registering`, () => {
  cy.server()

  cy.visit('/register')
    .getByText(/submit/i)
    .click()
    .getByText(/error.*try again/i)
})
```

Then for that mock server, we're going to specify a `cy.route`. The method here is a post method. We'll say `method` is `'POST'`. The `url` is what I just copied, that register URL. Then we can specify what we want the `status` code response to be.

We'll say `status: 500`, we'll just give it a `response` of whatever we actually want that response to be. We can just leave it blank, because the way that our code is written, it doesn't really matter what the response is.

```
it.only(`should show an error message if there's
an error registering`, () => {
  cy.server()
  cy.route({
      method: 'POST',
      url: 'http://localhost:3000/register',
      status: 500,
      response: {},
   })
  cy.visit('/register')
    .getByText(/submit/i)
    .click()
    .getByText(/error.*try again/i)
})
```

That's sufficient for us. Let's go ahead and save this, and we'll see what happens now. We see the browser does actually make that request, but Cypress is intercepting it, so that it can return a `500` internal server error. In our code, we're handling that by saying, "There was an error. Please try again."

As far as our application is concerned, it really did make that `500` request, but as you can see here, `XHR STUB` is what it's showing. That's because that post to the register URL was stubbed out, and it returned a status of 500 because of our mock.

Then our application correctly showed, `There was an error. Please try again.` Our test was successful. In review, to mock out an HTTP call in your client application, you create a Cypress `.server`. Then you supply a Cypress `.route` with the `method` and

`url`, and this can be a regex For us, it worked to be just a full string. Then you can specify the `status` and `response` that you want to have, rather than actually hitting the back-end server.

```
it.only(`should show an error message if there's
an error registering`, () => {
  cy.server()
  cy.route({
      method: 'POST',
      url: 'http://localhost:3000/register',
      status: 500,
      response: {},
   })
  cy.visit('/register')
    .getByText(/submit/i)
    .click()
    .getByText(/error.*try again/i)
})
```

Test user login with Cypress

I'm going to go ahead and create a new file called `login.js`. We're going to do pretty much a lot of the same thing that we're doing in a `registration`. I'm going to copy this over. We'll get rid of this error testing here. We don't need to worry about that for this.

Instead of `registration`, we'll say `login` and `should log in an existing user`.

login.js

```
import {userBuilder} from '../support/generate'

describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      .url()
      .should('eq',
`${Cypress.config().baseUrl}/`)
      .window()
      .its('localStorage.token')
      .should('be.a', 'string')
      .getByTestId('username-display', {timeout:
500})
      .should('have.text', user.username)
  })
})
```

How do we have an existing user? That user needs to be registered first. This is the registration process right here. Having done this, we now have a registered user. Now our test can start.

```
import {userBuilder} from '../support/generate'

describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()

      // Now our test can start
  })
})
```

Actually, we want to log out. We'll create a new user. We'll log out, and then we'll go log in as that user.

We'll `.getByText(/logout/i)t` and then `.click()`.

```
import {userBuilder} from '../support/generate'

describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      .getByText(/logout/i)
      .click()

      // Now our test can start

      // Now let's verify things after login
  })
})
```

Now our test will live right in here.

The first thing that we're going to do is we'll `.getByText(/login/i)`. We'll `.click()` on that. We'll do lots of the same thing that we did here actually. We'll fill in the `username` and `password` and `click` on the Submit button. I'll add that .click there.

Then we'll `.getByLabelText(/username/i)`, fill in the `username` and the `password`, and `submit` because our login form is pretty much exactly the same as our registration form. Once we've clicked, we should be logged in as that user.

```
describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      .getByText(/logout/i)
      .click()

      // Now our test can start
      .getByText(/login/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()

      // Now let's verify things after login
      .url()
      .should('eq',
`${Cypress.config().baseUrl}/`)
      .window()
      .its('localStorage.token')
      .should('be.a', 'string')
      .getByTestId('username-display', {timeout:
500})
      .should('have.text', user.username)
  })
```

```
})
```

Let's go ahead and save this. With this new file, Cypress will show that login test.

Awesome. We go through here. We click on the log in. We click on the register. We fill in a username and password. We click on log out. Then we click on log in. We fill in that same username and that same password. Then we verify that this user is logged in.

Create a user with cy.request from Cypress

It's great that we're testing the entire login flow, but this `'login'` test and this `'registration'` test look very, very similar. It makes me a little bit uncomfortable, because this means that we're doubly covering the exact same code.

It's not giving us a whole lot more confidence, but it taking our tests longer to complete. If something breaks, more of our tests are going to break. It'll take longer to identify what exactly it is that broke. Since we've already tested the registration, what if we were to simulate exactly what our registration code is doing so that we can register a new user, and then log in with that user.

Rather than clicking through the UI, we just do the same thing that our application when we register a new user. What is our application doing? It's going to the registration, it's filling out this form. Then when we click on this form, we POST something to the server.

What we were just to make that `POST` directly? I'm going to pin that, and I'll open my console here.

We'll go to console, and we can see we have this `Request` object with a `body` and `headers`. What if we just made that request ourselves?

That's exactly what we're going to do. Here at the top, I'll say `cy.request`, and the `url` is going to be this URL right here. I'll say `url: 'http://localhost:3000/register'`, `method: POST`, and the `body` needs to just be that `user` object, a `username` and `password`. I'll say `user`.

login.js

```
describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.request({
        url: 'http://localhost:3000/register',
        method: 'POST',
        body: user,
    })
    ...
```

With that, this `user` has been registered. They haven't been logged in, but that's what we're testing. Let's go ahead, and we'll clear out all the registration nonsense here, right down to where

our tests can start. Once we have a `user` that's been registered in the database, we can visit our app, go ahead to the `login` page, and fill out the `login` process.

```
describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.request({
      url: 'http://localhost:3000/register',
      method: 'POST',
      body: user,
    })
    cy.visit('/')
      .getByText(/login/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      // now let's verify things are set after
login.
      .url()
      .should('eq',
`${Cypress.config().baseUrl}/`)
      .window()
      .its('localStorage.token')
      .should('be.a', 'string')
      .getByTestId('username-display')
      .should('have.text', user.username)
  })
})
```

Then we can make our assertion that our user has been logged in. Let's go ahead and try that. I'll get rid of this thing. We'll save this, and we'll get a refresh in our tests. Perfect.

We start out with this `request` before we even visit the app to register our new user with the database.

Then we visit our app. We click on that `login` button. We get that `username` and type in the user's `username`, the same one that we requested with, and their password. Then we submit, and then we can verify that the user's logged in.

In review, to do this, we simply use the `cy.request` command to tell Cypress to make an HTTP request with this configuration, specifying the `user` as the `body`. The `user` has a `username` and `password`, and then we could go to the login screen and fill in that `username` and `password`.

```
describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.request({
      url: 'http://localhost:3000/register',
      method: 'POST',
      body: user,
    })
    cy.visit('/')
      .getByText(/login/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
    ...
```

What we had before was great, because it was testing more things, but we were already testing those things elsewhere. It was just adding extra things to do without increasing any of our confidence. It's nice to circumvent that, because we wouldn't be getting any additional confidence, anyway.

Keep tests isolated and focused with custom Cypress commands

A lot of my tests are going to require creating this new `user` that I can use to log in. I'm going to take this and create a custom command to generate a brand-new user. I'll go ahead and copy this.

login.js

```
import {userBuilder} from '../support/generate'

describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.request({
      url: 'http://localhost:3000/register',
      method: 'POST',
      body: user,
    })
```

I'm going into the `commands.js` file here. I'll use `Cypress.Commands.add`. We'll call this custom command `createUser`. This will be an arrow function. Our command logic will live inside of here.

commands.js

```
import {userBuilder} from '../support/generate'

Cypress.Commands.add('createUser', ()) => {

describe('login', () => {
  it('should login an existing user', () => {
    const user = userBuilder()
    cy.request({
    url: 'http://localhost:3000/register',
    method: 'POST',
    body: user,
    })
})
```

Let's clean this up a little bit. That `import` should be at the top. We'll get rid of `describe` and `it`, but we will want to create this `user`. We'll make this `request`. I'm also going to allow people to pass in `overrides` if they have something specific about the user that they want to have added to the created `user`.

```
import {userBuilder} from './generate'

Cypress.Commands.add('createUser', overrides =>
{
  const user = userBuilder(overrides)
  cy.request({
      url: 'http://localhost:3000/register',
      method: 'POST',
      body: user,
    })
})
```

Now we can go into our `login`. Instead of doing all of this stuff, we can say `cy.createUser`. We can get rid of all of this.

login.js

```
describe('login', () => {
  it('should login an existing user', () => {
    cy.createUser()
    cy.visit('/')
      .getByText(/login/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      // now let's verify things are set after
login.
      .url()
      .should('eq',
`${Cypress.config().baseUrl}/`)
      .window()
      .its('localStorage.token')
      .should('be.a', 'string')
      .getByTestId('username-display')
      .should('have.text', user.username)
  })
})
```

Now we've got a missing `user`. We can get rid of that builder, but where's that `user` coming from? We could create that ourselves and then pass in those overrides, but it would be nice if this thing takes care of all that for us, and we'd get access to the `user` somehow.

The way we're going to do this is we're going to add `.then()`. This will take our `subject`. Then we can take everything that we have out here and put it inside of our`.then()`. That has access to

the `subject`, which in our case is going to be the `user`.

```
describe('login', () => {
  it('should login an existing user', () => {
    cy.createUser().then(user => {
      cy.visit('/')
        .getByText(/login/i)
        .click()
        .getByLabelText(/username/i)
        .type(user.username)
        .getByLabelText(/password/i)
        .type(user.password)
        .getByText(/submit/i)
        .click()
        // now let's verify things are set after
login.
        .url()
        .should('eq',
`${Cypress.config().baseUrl}/`)
        .window()
        .its('localStorage.token')
        .should('be.a', 'string')
        .getByTestId('username-display')
        .should('have.text', user.username)
    })
  })
})
```

We have to set the `subjejct` in Cypress to be that `user`. What we're going to do is we'll say `.then()`. The subject that `request` yields is the `response` obect. We're going to take that `response` object, and we'll say `response.body.user`. This changes the subject from the `response` object to the `user` object on that `response`.

commands.js

```
import {userBuilder} from './generate'

Cypress.Commands.add('createUser', overrides =>
{
  const user = userBuilder(overrides)
  return cy
    .request({
      url: 'http://localhost:3000/register',
      method: 'POST',
      body: user,
    })
    .then(({body}) => body.user)
})
```

With that, I can create the `user` entirely in another command and use the same command in any other test that needs a new `user` created in the database without having to duplicate a bunch of code anywhere that's needed. In our output, we still do that same request that we had before.

In review, to create a custom command, you're going to do that in this `commands.js` file which is in this `support` directory, and it's imported in that `index.js` file. Inside of the commands, you can say `Cypress.Commands.add`. Then you give the name of the command and a function to run when that command is run.

commands.js

```
import {userBuilder} from './generate'

Cypress.Commands.add('createUser', overrides =>
{
  const user = userBuilder(overrides)
  return cy
    .request({
      url: 'http://localhost:3000/register',
      method: 'POST',
      body: user,
    })
    .then(({body}) => body.user)
})
```

Inside of here, you can queue up other Cypress commands. The last Cypress command that you execute can change the subject by using this `.then()` API. Then we used it in our `login` test where we said `cy.createUser`. We got access to that new subject, the `user`. We used that new subject in the rest of the commands that we execute with Cypress.

login.js

```
describe('login', () => {
  it('should login an existing user', () => {
    cy.createUser().then(user => {
      cy.visit('/')
        .getByText(/login/i)
        .click()
        .getByLabelText(/username/i)
        .type(user.username)
        .getByLabelText(/password/i)
        .type(user.password)
        .getByText(/submit/i)
        .click()
        // now let's verify things are set after
login.
        .url()
        .should('eq',
`${Cypress.config().baseUrl}/`)
        .window()
        .its('localStorage.token')
        .should('be.a', 'string')
        .getByTestId('username-display')
        .should('have.text', user.username)
    })
  })
})
```

Use custom Cypress command for reusable assertions

In our `login`, we were able to clean up a bunch of duplicate code from the registration test by using this `createUser` command, but we still have a whole bunch of duplicate code for our assertion. If we look at `register` here, that looks awfully like what we have in our `login`.

login.js + register.js

```
.url()
.should('eq', `${Cypress.config().baseUrl}/`)
.window()
.its('localStorage.token')
.should('be.a', 'string')
.getByTestId('username-display')
.should('have.text', user.username)
```

I don't want to remove that from the `login`, because there could be something that I change with the `login` that breaks the `login` but not the registration, so I'm not actually testing the exact same thing. Let's see if we reduce this duplication by using another Cypress custom command.

I'm going to copy this, and we're going to go to our `commands.js`, and I'm going to make a new command. `Cypress.Commands.add`. The first one I'm going to make is `assertHome`. This one is going to be pretty simple. I'll just paste this down here in a comment.

commands.js

```
Cypress.Commands.add('assertHome', () => {

})
```

What it's going to do is I'll say `cy` and then we'll bring up the `url should` equal the home URL.

```
Cypress.Commands.add('assertHome', () => {
  cy.url().should('eq',
`${Cypress.config().baseUrl}/`)
})
```

Then, in both `register.js` and `login.js` I can get rid of this,

```
.url()
.should('eq', `${Cypress.config().baseUrl}/`)
```

and I could say `assertHome()`, and use that custom command in `register`, and then use it in `login` as well.

Let's see what we can do about this duplication.

```
.window()
.its('localStorage.token')
.should('be.a', 'string')
.getByTestId('username-display')
.should('have.text', user.username)
```

I'll make another custom command, `Cypress.Commands.add('assertLoggedInAs')`, and this one will take a `user`. Here we'll use `cy` and let's just bring all of these commands up here.

It's going to take that user and verify that the username is what is being displayed right there.

```
Cypress.Commands.add('assertLoggedInAs', user =>
{
  cy.window()
    .its('localStorage.token')
    .should('be.a', 'string')
    .getByTestId('username-display', {timeout:
500})
    .should('have.text', user.username)
})
```

Let's go and save this, and instead of all this stuff in `regsiter`, we can say `assertLoggedInAs` and pass the `user`.

register.js

```
describe('registration', () => {
  it('should register a new user', () => {
    const user = userBuilder()
    cy.visit('/')
      .getByText(/register/i)
      .click()
      .getByLabelText(/username/i)
      .type(user.username)
      .getByLabelText(/password/i)
      .type(user.password)
      .getByText(/submit/i)
      .click()
      .assertHome()
      .assertLoggedInAs(user)
  })
```

We'll do the same thing in our `login`, and we get the exact same output as we did before.

login.js

```
describe('login', () => {
  it('should login an existing user', () => {
    cy.createUser().then(user => {
      cy.visit('/')
        .getByText(/login/i)
        .click()
        .getByLabelText(/username/i)
        .type(user.username)
        .getByLabelText(/password/i)
        .type(user.password)
        .getByText(/submit/i)
        .click()
        .assertHome()
        .assertLoggedInAs(user)
    })
  })
})
```

Now our `login` test is pretty lean. It doesn't do any more than it should, and its shared assertions are unique commands that we can execute.

Normally, I might not make an abstraction for just two duplications of code, but I'm probably going to have a couple more tests, where I want to assert that we're logged in as a specific user, or assert that we were redirected to the home page. Having these special commands for `assertHome` and `assertLoggedInAs` will be really valuable to me in the long term.

In review, what we did here was we noticed we had some duplicate assertions between our two different tests, and so we made a custom command `assertHome` and `assertLoggedInAs`,

that encapsulated those commands. Then, we were able to exchange all of those commands in both of our tests for our custom commands.

Run tests as an authenticated user with Cypress

We have a test for an `anonymous calculator`. Let's go ahead and make a test for a logged-in calculator. We want to have a user who's logged in and verify some features that the user is able to do when they're logged in.

In our contrived example, we're just going to verify that the user's display name is displayed and that they can log out, just to show how to run tests as a logged-in user. Here we're going to make a `describe('authenticated calculator',`. We'll say, `it('displays the username',`

Here we can use `cy.createUser`, our custom command. We'll save that.

calculator.js

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser()
  })
})
```

We'll say `then(user => {...`. Then we'll start executing our commands. We'll say, `cy.visit` our application at home.

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
        cy.visit('/')
    })
  })
})
```

Then we need to log in.

Let's go ahead and we'll get our `login` test. We'll pretty much do all the same stuff that we did here. Then we'll go ahead and say, `assertLoggedInAsUser`. Then we'll `.getByText(/logout/i)`. We'll `.click()` on the logout button.

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
        cy.visit('/')
          .getByText(/login/i)
          .click()
          .getByLabelText(/username/i)
          .type(user.username)
          .getByLabelText(/password/i)
          .type(user.password)
          .getByText(/submit/i)
          .click()
          .assertLoggedInAs(user)
          .getByText(/logout/i)
          .click()
    })
  })
})
```

Then we need to verify that something does not exist in the DOM. With `cypress-testing-library`, you have these `getBy` queries. If something does not exist, a `getBy` query is actually going to throw an error after four seconds.

We need to assert that something does not exist. We'll use a `queryByTestId`, which will not throw an error. We're going to look for the `username-display`. We'll go ahead and have it `timeout` after `300` milliseconds because we know that it's not going to be there.

```
.queryByTestId('username-display', {timeout:
300})
```

If it's not there after 300 milliseconds, we can feel pretty confident that it's not going to appear there at all. By using `timeout`, it means that our tests will run a little bit faster. Then we can say, `.should('not.exist')`.

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
      cy.visit('/')
        .getByText(/login/i)
        .click()
        .getByLabelText(/username/i)
        .type(user.username)
        .getByLabelText(/password/i)
        .type(user.password)
        .getByText(/submit/i)
        .click()
        .getByTestId('username-display')
        .should('have.text', user.username)
        .getByText(/logout/i)
        .click()
        .queryByTestId('username-display',
{timeout: 300})
        .should('not.exist')
    })
  })
})
```

We'll go ahead and save that. Now our test will run. We're all set.

In review, here we created a new user. Then we used that user to go through the login flow. Then we asserted that we were logged in as that user. We clicked on the logout. Then we asserted that the `username-display` no longer exists.

Use cy.request from Cypress to authenticate as a new user

All the stuff that we're doing to log the user in is a little bit verbose. It's doing some extra work that we're already testing in our `login` test. Let's go ahead and expand this and see what actually happens. Maybe we can simulate the same type of thing using `cy.request` like we did with registration.

Here we go on and fill this form where you send a `POST` to the `login`, so I'm going to click on that. We'll pop this open. We can see the `Request` is just the `body: {username: "...", password: "..."}`. We're making that request to log in.

Let's go ahead and do this using `cy.request`. We'll say `cy.request`. The `url` is this URL here. I'm going to snag that really quick. The `method`, that's a `POST` right there. The `body` it's going to be our `user`, just that object that has a `username` and a `password`.

calculator.js

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
      // login as the new user
      cy.request({
        url: 'http://localhost:3000/login',
        method: 'POST',
        body: user,
      })
```

Something that we don't actually see here but is an important aspect of our authentication for our application is the `response` comes back with a `user` that has a `token` on it. That `token` needs to get set into local storage.

We're going to add a `.then()`. We'll get the `response`. We'll say `window.localStorage.setItem('token')`. This will be `response.body.user.token`. With that, our user is authenticated and logged in as a user when they first start the application.

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
      // login as the new user
      cy.request({
        url: 'http://localhost:3000/login',
        method: 'POST',
        body: user,
      }).then(({body}) => {
        window.localStorage.setItem('token',
body.user.token)
      })
```

Let's go ahead and update our test. We no longer need to go to `login`. In fact, we can't because we're actually authenticated. We'll get rid of all that stuff. We can start out our test by verifying that the `username-display` is there and that it has our `username` text. Then `click` on the `logout` and verify that the `username-display` is not there.

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
      // login as the new user
      cy.request({
        url: 'http://localhost:3000/login',
        method: 'POST',
        body: user,
      }).then(({body}) => {
        window.localStorage.setItem('token',
body.user.token)
      })

      cy.visit('/')
        .getByTestId('username-display')
        .should('have.text', user.username)
        .getByText(/logout/i)
        .click()
        .queryByTestId('username-display',
{timeout: 300})
        .should('not.exist')
    })
  })
})
```

With all that, we can have our test run. It runs a lot faster.

We first `POST` to register a new `user`. Then we `POST` to log in as that `user` and get the `token`. Then we visit our app. Our app actually makes a `/me` request when it gets started so that it can get the user's information. Then it renders with that user's information.

We can assert that the `username-display` is showing our user's `username`. We go ahead and `click` on the `logout`. We can verify that the `username-display` is no longer on the page.

We do that login by making a `cy.request` command that posts to this login URL with our user's information. Then we set the `token` in local storage to be our `respose.body.user.token` so that when we visit our application, the application will see that `token` exists in local storage and will make the `/me` request to get the user's information. Then it will render the user's `username`.

Use a custom Cypress command to login as a user

It's probably likely that we're going to want to do this `login` request for many tests, so I'm going to make a command out of this. I'll just copy this, we'll go to our `commands.js`, and I'll add a custom `Cypress.Commands.add`, and we'll just call this simply `login`. That will accept a `user`, and then I'm just going to paste in those commands.

commands.js

```
Cypress.Commands.add('login', user => {
  cy.request({
      url: 'http://localhost:3000/login',
      method: 'POST',
      body: user,
  }).then(({body}) => {
    window.localStorage.setItem('token',
body.user.token)
  })
})
```

Then we'll go back to our `calculator.js` and we'll swap this out for a `cy.login` with that `user`. We'll save that.

calculator.js

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
      // login as the new user
      cy.login(user)

      cy.visit('/')
        .getByTestId('username-display')
        .should('have.text', user.username)
        .getByText(/logout/i)
        .click()
        .queryByTestId('username-display',
{timeout: 300})
        .should('not.exist')
      })
    })
  })
})
```

Once we've logged in we'll visit the app and execute all of our other commands, and everything is working great.

In review, we just copied a bunch of commands, put them in a custom command here. Then we're able to use it both in this test and any other test that needs to log a user in.

Combine custom Cypress commands into a single custom command

When testing, it's a good idea to make sure that your tests are isolated. That's why we're creating a brand-new `user` for each one of our tests that need a `user`. Most of the time when we

create this `user`, we're probably going to need to be logged in as this `user`.

calculator.js

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.createUser().then(user => {
      // login as the new user
      cy.login(user)

      cy.visit('/')
        .getByTestId('username-display')
        .should('have.text', user.username)
        .getByText(/logout/i)
        .click()
        .queryByTestId('username-display',
{timeout: 300})
        .should('not.exist')
    })
  })
})
```

What I'm going to do is I'm going to take this and make it a single command, `loginAsNewUser()`. We can use that in all the tests that need an authenticated user. I'll copy this. We'll use `Cypress.Commands.add('loginAsNewUser')`. Then we'll provide that arrow function with exactly what we were doing before.

commands.js

```
Cypress.Commands.add('loginAsNewUser', () => {
  cy.createUser().then(user => {
    cy.login(user)
  })
})
```

If I go back to my test, I can say `cy.loginAsNewUser()`, get the `user`. Then we'll move all these commands inside of there. One last of step I'm going to do here is make sure that that subject is `body.user`. We'll `return body.user` to keep the subject of `Cypress` to be that `user`. With that, our tests are passing.

calculator.js

```
describe('authenticated calculator', () => {
  it('displays the username', () => {
    cy.loginAsNewUser().then(user => {
      cy.visit('/')
        .getByTestId('username-display')
        .should('have.text', user.username)
        .getByText(/logout/i)
        .click()
        .queryByTestId('username-display',
{timeout: 300})
        .should('not.exist')
    })
  })
})
```

commands.js

```
Cypress.Commands.add('login', user => {
  return cy
    .request({
      url: 'http://localhost:3000/login',
      method: 'POST',
      body: user,
    })
    .then(({body}) => {
      window.localStorage.setItem('token',
body.user.token)
      return body.user
    })
})
```

In review, we had this really common use case where we need to log in as a new `user`, which we'll create a new `user`, register that `user` to the database, and then log that `user` into the application. Then we can use that `user` in our test. We did this by creating a new custom command that itself used other custom commands composing those commands together to make us a nice, useful `loginAsNewUser()` command.

Install React DevTools with Cypress

Because our application is a React application and we have access to the Chrome's DevTools, it sure would be nice to have access to React Developer Tools, so we can access React component instances, see what the React tree actually looks like, and really use this Cypress tool as a replacement for our development workflow with Chrome.

This actually is a Chrome browser, so I can open up a new tab and we're going to Google `react developer tools chrome`. Right here at the top, I can open that up and I can add it to Chrome. I'll add that extension. It gets added to our Cypress Chrome browser, so then I can close this out.

I'll go ahead and refresh. We can see that React is active on this page. To get React in our DevTools, I'm going to close those and open it up again, and there is the React tab in our DevTools!

It connects to React and we've got a `Container`, and our `App`, and...Oh, it looks like Cypress is actually built with React itself. That's what we're getting in our React developer tools. That's not all that useful, so let's see how we can make certain the React developer tools references our application rather than the Cypress application.

React DevTools Tab

Let's go to this `public` directory. We'll open up our `index`. This is the `index.html` that served to Cypress for our application. We have our `script` tag loading up our React `bundle.js` here. All that matters is that this code that we're going to execute happens before React is loaded onto the page.

index.html

```
<head>
  <meta charset="utf-8">
  <title>React Calculator | Michael Jackson |
React Training</title>
  <meta name="viewport" content="width=400,
height=600, initial-scale=0.8, maximum-scale=1">
</head>

<body>
  <div id="wrapper">
    <div id="app"></div>
  </div>
  <script type="text/javascript"
src="bundle.js"></script>
</body>
```

I'm going to add a `script` tag right here `script`, and we'll say `if (window.Cypress)`. Then, we're running in a Cypress environment. Then, I want to say `window.__REACT_DEVTOOLS_GLOBAL_HOOK__`. That's a variable that the React developer tool sets onto the page. React internally will reference this variable to register components with it.

We're going to create that variable in our application. We'll assign it to the one that comes from our `parent`, because the `parent` is where React DevTools is installed. We'll just forward that along to ourselves, so that React running within the iframe has access to it.

index.html

```
<head>
  <meta charset="utf-8">
  <title>React Calculator | Michael Jackson |
React Training</title>
  <meta name="viewport" content="width=400,
height=600, initial-scale=0.8, maximum-scale=1">
  <script>
    if (window.Cypress) {
      window.__REACT_DEVTOOLS_GLOBAL_HOOK__ =
window.parent.__REACT_DEVTOOLS_GLOBAL_HOOK__
    }
  </script>
</head>
```

I'll save that and we'll refresh. Actually, we'll refresh the entire browser and open up my DevTools. We'll look at React. We've got our `Container` that's coming from React. We also have this `tooltip`. Apparently, lots of these tool tips are individual components in themselves, probably React portals.

Tooltips

We have our own component from our application. We can look at `LoadUser`, `Router`, and so on and so forth. Let's find `App`. Here we are, `<App path="/" uri="/">` there is our `ThemeProvider` and our `Calculator`. We can access all of these things right here.

I can look at the `State`. I can change the `State`. I can do all of the things that I would be used to doing in my regular Chrome browser right within Cypress, making it a lot easier to develop my React application with Cypress.

Cypress Browser

In review, to get this to work, we simply Googled for the React extension and installed it. Then, we have to register our application from within the iframe to the React developer tools in the browser.

We did that in our `index.html`, but you could do that from within your bundle, and just needs to happen before React is loaded onto the page. An alternative method that we could have used here is added a new `script` called `react-dev-tools.js`.

We could take this `script` here, remove that from our `index.html`, stuck it right in here. In our `index.js`, just at the very top `import './react-dev-tools'`, so that could have worked as well. All that really matters is that this happens before React is imported.

react-dev-tools.js

```
if (window.Cypress) {
  window.__REACT_DEVTOOLS_GLOBAL_HOOK__ =
window.parent.__REACT_DEVTOOLS_GLOBAL_HOOK__
}
```

index.js

```
import './react-dev-tools'
```

With that registration, when React is loaded onto the page for our application, it registers itself with the DevTools from the Cypress browser.