Test Production Ready Apps with Cypress

Transcripts for Brett Cassette (https://egghead.io/instructors/brett-shollenberger) course on egghead.io (https://egghead.io/courses/test-production-ready-apps-with-cypress).

Description

One of the most important — but most ignored — practices for web developers is performing end-to-end testing on applications before they go live into production. You know: making sure they work like they're supposed to. But testing can be tedious, and definitely not fun.

In this course, Brett Cassette will show you how you can test all layers of your application stack, simultaneously, with Cypress. When you use Cypress, it's like having a robot that uses your app the way a real user would. Cypress reports if things work the way you designed them to — and if they don't — every step of the way. And it's fun.

After completing this course, you'll be ready to apply the same E2E testing principles to your own applications. Stop leaning on your QA department (if it exists) to stress-test your application, and ship your app knowing it's ready.

Course Introduction: Test Production Ready Apps with Cypress

End-to-end testing gets a bad name. It's flaky. It's unreliable. It's slow. All of that changed with Cypress.

`Cypress` sees the world like a real user. It knows what's visible, what's hidden. It automatically waits for objects to show up on the DOM. No more sleeps. No more waits. Cypress handles this out-of-the-box.

You can travel through time. You can take screenshots. That all just undersell the sales pitch. That says Cypress is better and faster than what came before, but Cypress is more than that. It's a paradigm shift in end-to-end testing. That's because it can test every layer of the stack, the database, the API, the XHR request, our UI, our frontend stores, everything.

With `Cypress`, you can interact with your frontend and your backend and everything in between. Confusing error codes can become a thing of the past because if you test every layer in order, you can pinpoint the exact location where the contract breaks down. You can communicate with new team members more effectively what the dataflow looks like.

Yes, we can still use diagrams and drawings, but you can also say it all with the test now. Now you can't quite do all of this out-of-the-box with Cypress. That's why, in this course, I'm going to

show you how you can build a transformative testing environment.

Install Cypress in a Production Application

In this lesson, we'll learn how to add Cypress to an existing application. First, let's `git clone` the sample repo.

Terminal

```
$ git clone
https://github.com/SamGrinis/cypress-egghead-
course.git

Cloning into 'cypress-egghead-course'...
remote: Enumerating objects: 157, done.
remote: Counting objects: 100% (157/157), done.
remote: Compressing objects: 100% (111/111),
done.
remote: Total 462 (delta 75), reused 113 (delta
42), pack-reused 305
Receiving objects: 100% (462/462), 512.79 KiB |
6.66 MiB/s, done.
Resolving deltas: 100% (218/218), done.
```

Next, `cd cypress-egghead-course` into the repo and `git checkout 01-cypress-install`.

```
$ cd cypress-egghead-course
$ git checkout 01-cypress-install
```

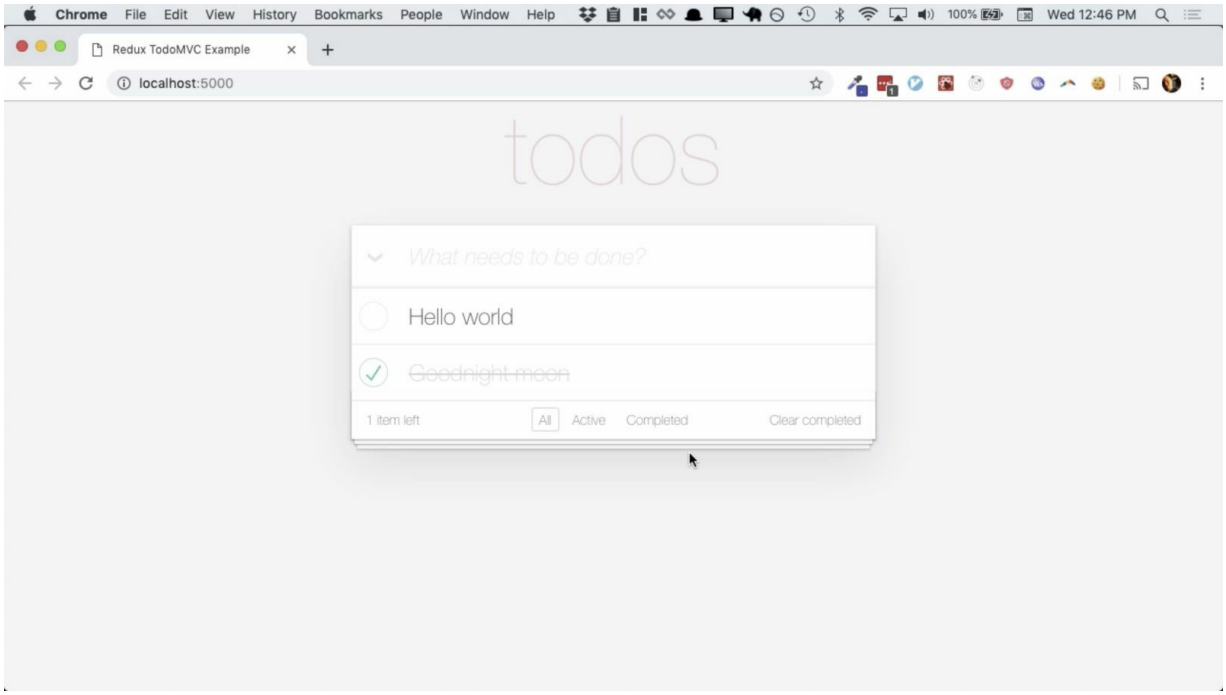
Next, run `npm install`.

```
$ npm install
```

When it's finished installing, you can start the demo application with `npm run start`.

```
$ npm run start
```

This command will automatically open the demo application on localhost 5000.



You can verify that your application is working by running `npm run test`.

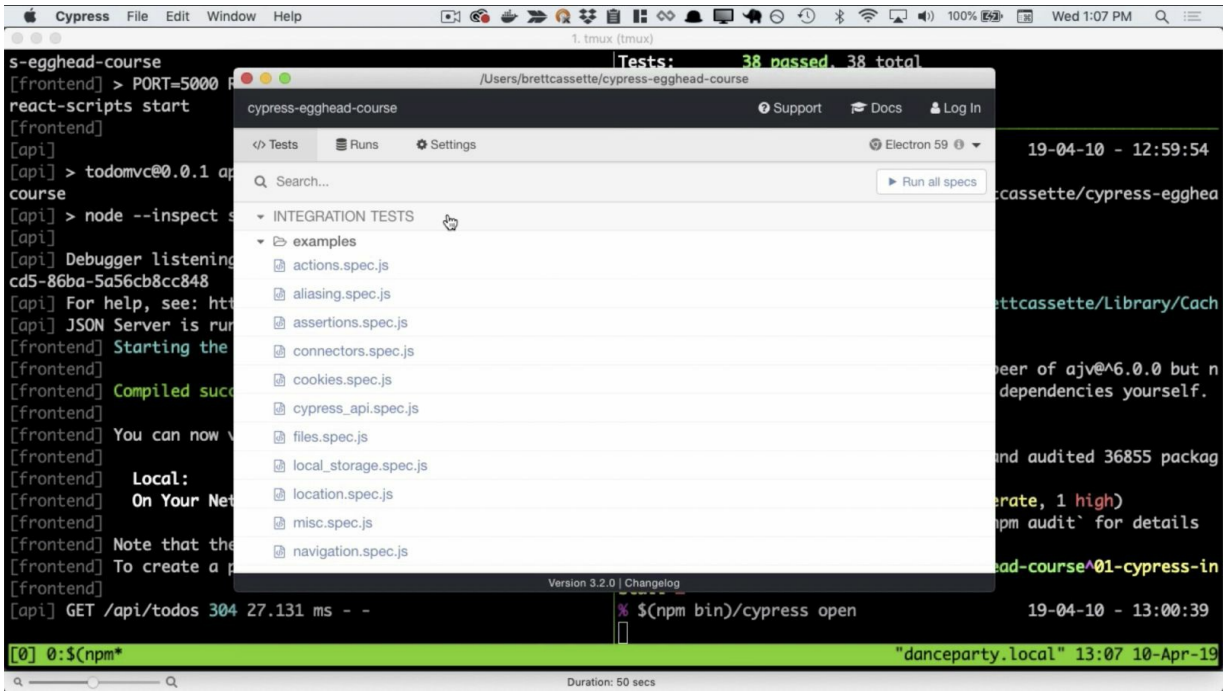
```
$ npm run test
```

Once our application is verified, we're prepared to `npm install cypress --save dev`.

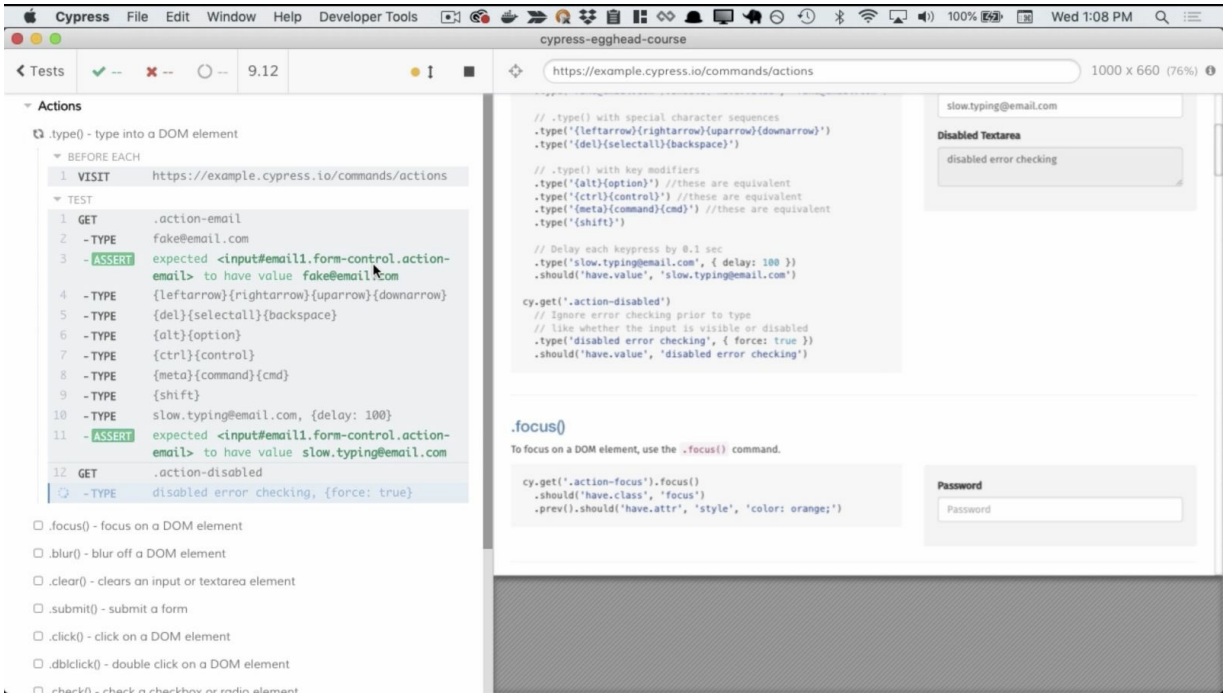
```
$ npm install cypress --save-dev
```

Once Cypress is installed, we're prepared to run it with `(npm bin)/cypress open`. This will pop open the Cypress interactive GUI which comes pre-seeded with a number of tests.

```
$ (npm bin)/cypress open
```



If we click on one, we can see an example of Cypress in action. This is Cypress running on the Cypress website.



We can see here that Cypress can either run against our local page or against a remote page.

All of these example files live in the Cypress directory under integration. You'll probably want an easier way to run Cypress, so open `package.json` and add a `cypress` command, which will be `cypress open`.

Package.json

```
  "scripts": {
    "start": "concurrently 'npm:frontend'
'npm:api'",
    "frontend": "PORT=5000
REACT_APP_API_URL=http://localhost:3000 react-
scripts start",
    "api": "node --inspect server.js",
    "build": "react-scripts build",
    "eject": "react-scripts eject",
    "test": "react-scripts test --env=node",
    "cypress": "cypress open"
  }
```

We can use our new command by running `npm run cypress`.

terminal

```
$ npm run cypress

> todomvc@0.0.1 cypress
/Users/samgrinis/cypress-egghead-course
> cypress open
```

Now you're all set up with Cypress and ready to roll. If you'd like to do an exercise before moving on to the next lesson, take a moment and familiarize yourself with some of the example tests.

Setup Your Cypress Dev Environment

In this lesson, we're going to set up the Cypress dev environment in order to make learning Cypress as easy as possible. To get started, check out the O2 Cypress Dev Environment branch.

terminal

```
$ git checkout 02-cypress-dev-enviroment
```

If you use VS Code, and you used `npm install` to install Cypress, then you already have IntelliSense set up.

The reason this works is because of what VS Code calls automatic type acquisition. This can work in one of two ways. The most common way is to use this triple slash directive, `///`, which describes which types are defined in this file.

The triple slash directive isn't limited to VS Code, and works in any editor that uses IntelliSense. If want to prove that the triple slash directive is what's loading this up for us, we can comment it out.

actions.spec.js

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

Once we save it and hover over `cy.visit` it has no idea where the type definitions are found.

Let's dive in a little deeper to see where these come from. Let's comment that back in.

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

Hover over `cy.visit` again, click Go To Definition, and see this `index.d.ts` file which we can see is located in this directory under the node_modules folder, under `cypress/types/index.d.ts`. We can also see this in the side bar.

What would happen if we moved this file from `node_modules/cypress/types/index.d` to the root of our project?

terminal

```
$ mv node_modules/cypress/types/index.d.ts .
```

Will the IntelliSense still work? We reopen our editor and see that the file we opened before has been deleted from the disc because we moved it to the project root. If we reopen and look this up, IntelliSense is still working. Let's go to definition and see what happens now. VS Code is still able to find the typescript definition file.

As I look at it, I can tell that this installed in my `global/node_modules` folder, which means that VS Code is able to resolve either a global installation of Cypress or a local installation of Cypress to give us these typescript definitions.

Now that we've learned where the file comes from, let's go ahead and move this back to our `node_modules/cypress/types` folder.

```
$ mv index.d.ts node_modules/cypress/types
```

In many editors the triple slash directive is the only way we can find type information for our Cypress files. However, in VS Code we have a global way to look these up. I'll comment this out, and we know that this means the `///` directive is no longer going to be used.

actions.spec.js

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

Now let's create a `tsconfig.json` file in the root of the `cypress` directory. If you visit the Cypress docs located in the show notes, you can copy this tsconfig and paste it in.

tsconfig.json

```
{
    "compilerOptions": {
        "allowJs": true,
        "baseUrl": "../node_modules",
        "types": [
            "cypress"
        ]
    },
    "include": [
        "**/*.*"
    ]
}
```

Now that we've configured the `ts.config`, we can still look up the `cy.visit` method, even without our triple slash directive.

There's just one more config file that we're going to want to add to the root of our project. Cypress itself accepts a lot of different configuration options. For instance, we can configure what the root URL is of our project.

If we close this out, go into our project root and add a new file called `cypress.json`, we can configure this here. We can add `baseURL` for instance, `http://localhost:5000` which is the root of our project.

cypress.json

```
{
    "baseURL": "http://localhost:5000",
}
```

VS Code can even be configured to tell us which options are accepted by the cypress.json file, or if we hover over, to tell us a little bit more about what option we're configuring.

To configure this, we can go to the user settings under `json.schemas`, edit in `settings.json`, and we can see that I've already configure my editor here to do this.

settings.json

```
{
    "workbench.editor.enablePreview": false,
    "json.schemas": [
        {
            "fileMatch": [
                "cypress.json"
            ],
            "url":
"https://on.cypress.io/cypress.schema.json"
        }
    ],
...
}
```

This is another configuration that's available on the Cypress documentation, and is included in the show notes.

Write Your First Cypress Integration Test

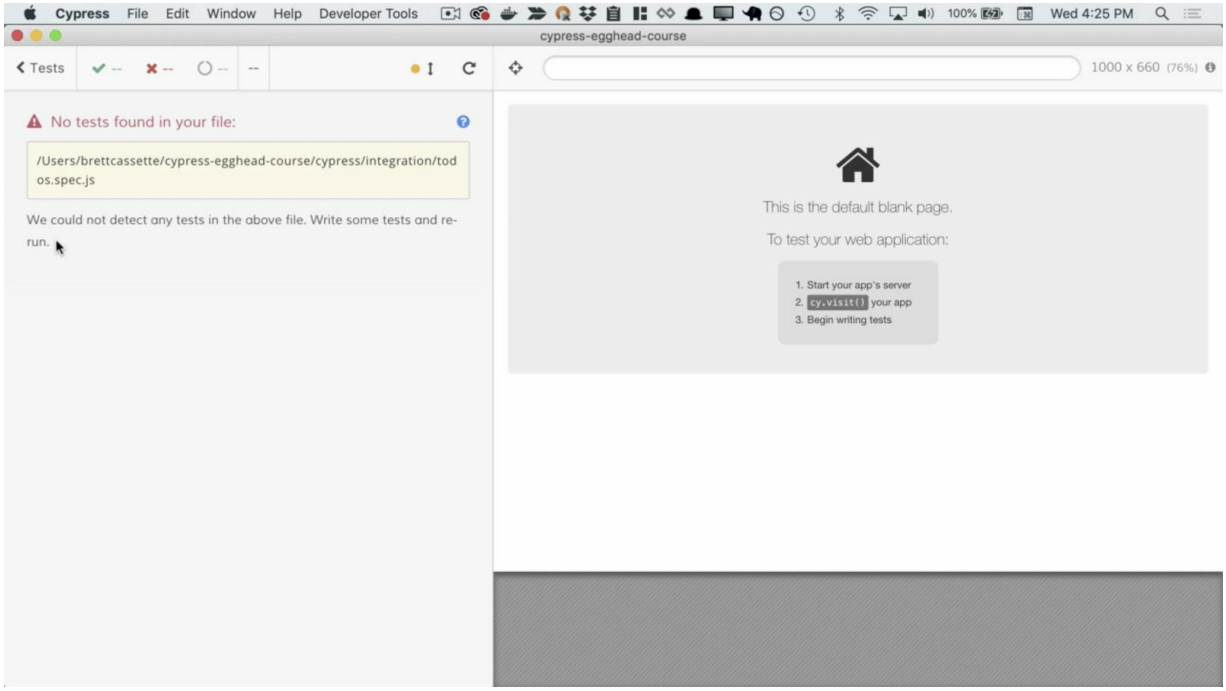
In this lesson, we're going to write our very first Cypress test. As usual, let's check out the branch associated with the lesson.

terminal

```
$ git checkout -b 03-first-integration-test
```

Up until now, we've been looking at the examples that were generated automatically when we first ran Cypress. Let's go ahead and blow this away and create our very first integration test. We can call it `todos.spec.js`.

We can see in our Cypress runner that `todos.spec.js` is created here. If we click on it, it will find no task in the file.



Let's go ahead and write one.

We'll start with a `describe` block and describe our todo application. Inside that, we can create an `it` statement. We can say `it('loads the page'`. Now we'll just visit it. Let's `cy.visit` at the root of our application. When we visit the page, we'll see that Cypress loads our todo app.

todos.spec.js

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')
```

`describe` and `it` are provided by the Mocha testing framework, which you can tell if you hover over them with your IntelliSense. Cypress itself is build upon a lot of existing best practice libraries

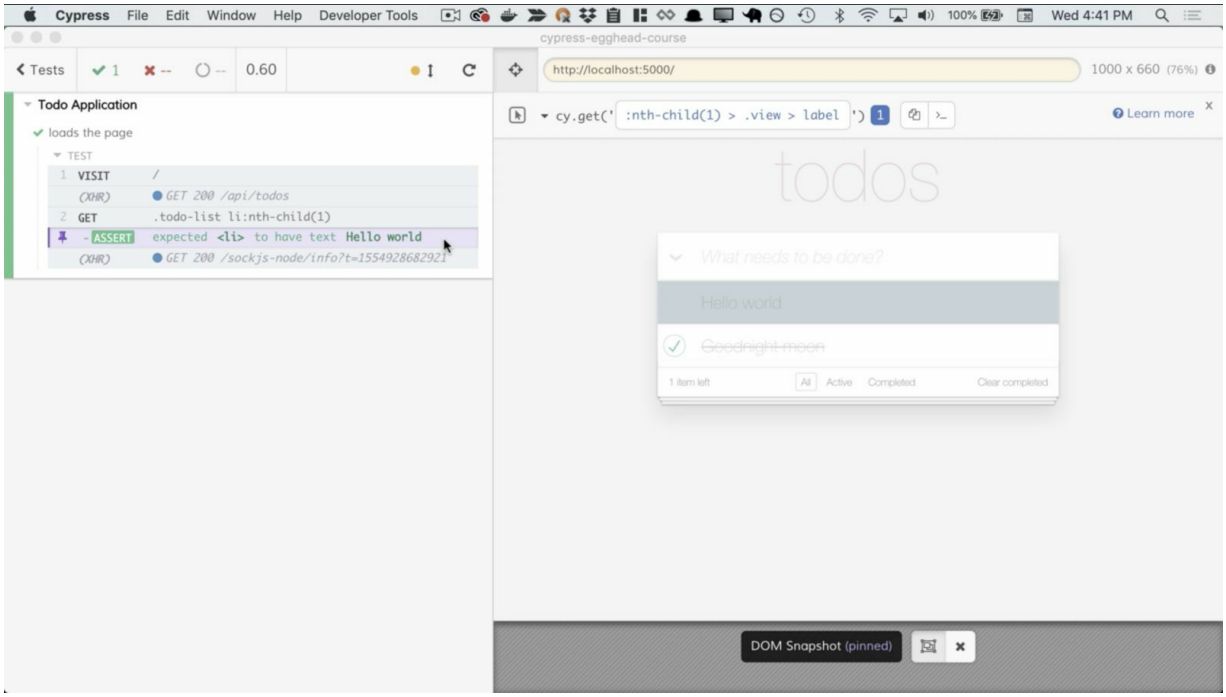
like Mocha and jQuery. You will be familiar with some of the nuts and bolts.

Let's learn how to use our `cy` commands. We've already seen how to use `cy.visit`. The next thing we'll learn is `cy.get`. Let's get the first item in our todo list. `cy.get`, we pass in the selector `.todo-list` which wraps all of our todos.

We implemented todo items as list items, and we can use `nth-child(1)` to select the first one. For assertions, Cypress includes the child library. If you're familiar, you can use `s.should('have.text', 'Hello world')`.

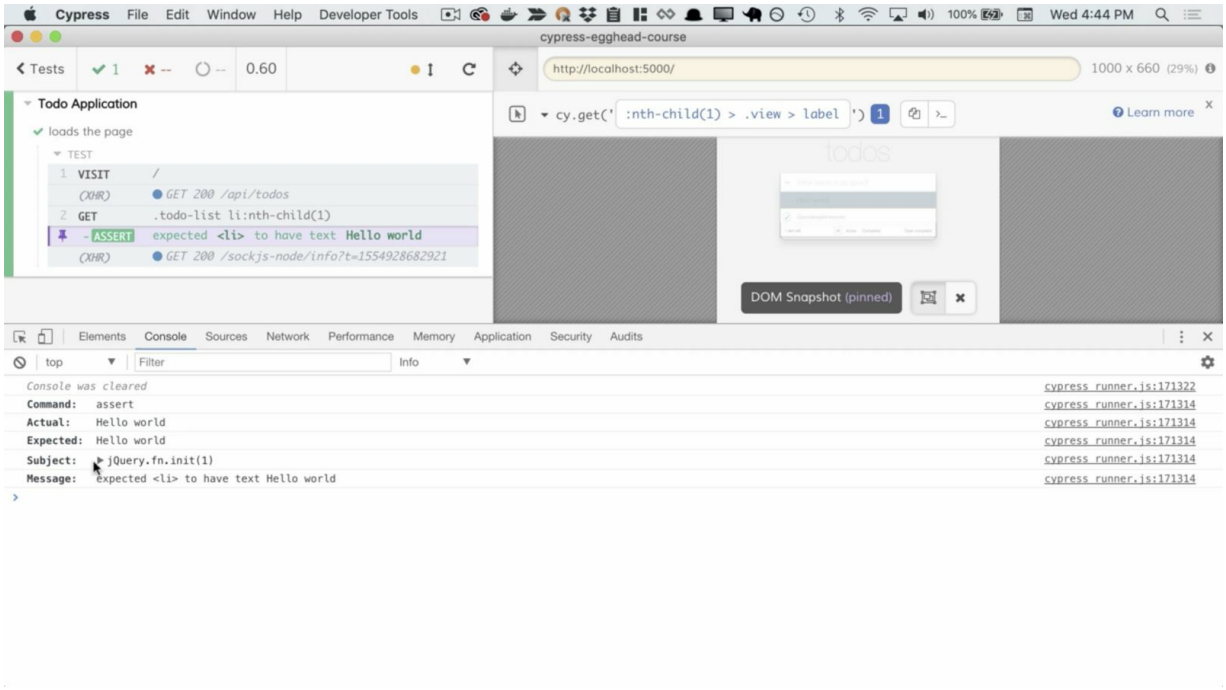
```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('.todo-list li:nth-child(1)')
          .should('have.text', 'Hello world')
```

If we head back into Cypress, we can see that task already ran and that it passed. We can step through each moment and time.



For instance, when we first visited the page, after we received our XHR request to the backend to load all our todos, when we targeted this first list item, and we can see what was targeted here on the screen, and then we can see our assertion which targeted this list item and expected it to have the text Hello World.

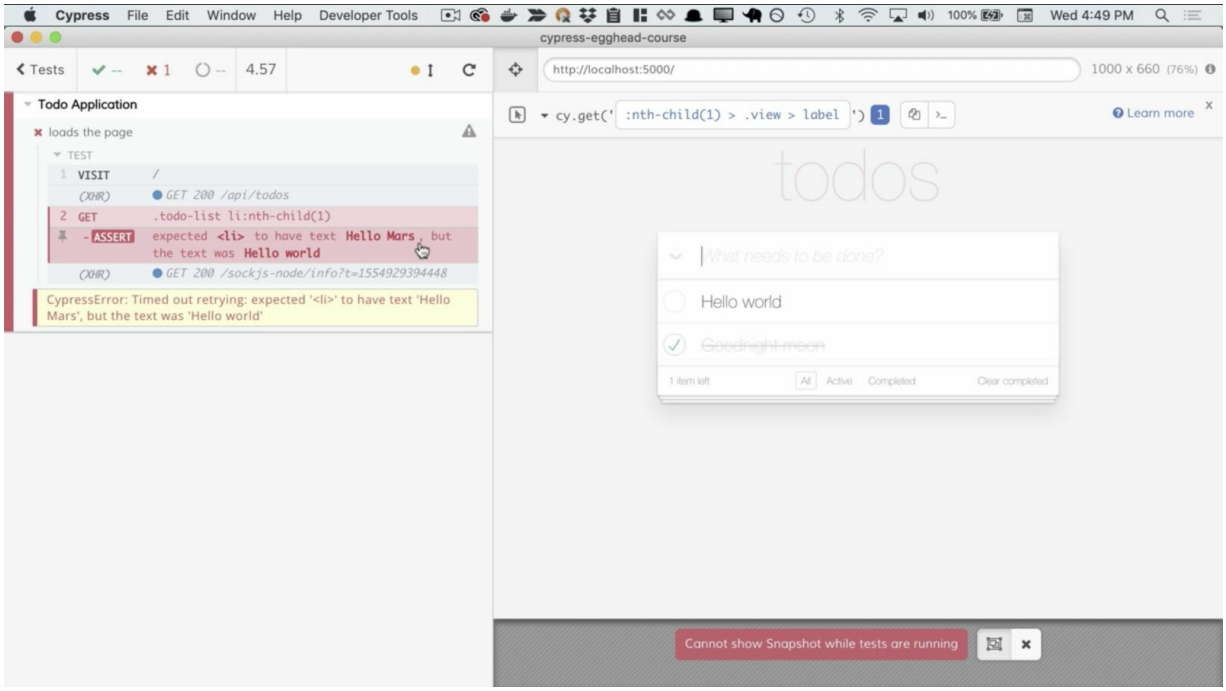
If we want more contextual information, we can pop open our console and see a little bit more. We can see, for instance, the command was asserted. We expected the text Hello World. That's what we got. We can see our subject, which was this jQuery selector of the list item.



We should also probably check that the testing just passed because of a fluke. Let's go ahead and change this to Hello Mars and reopen Cypress.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('.todo-list li:nth-child(1)')
          .should('have.text', 'Hello Mars')
```

We'll see here that the assertion hangs a lot longer this time. The reason it does that is because it's waiting for the text Hello World to change into `Hello Mars`.

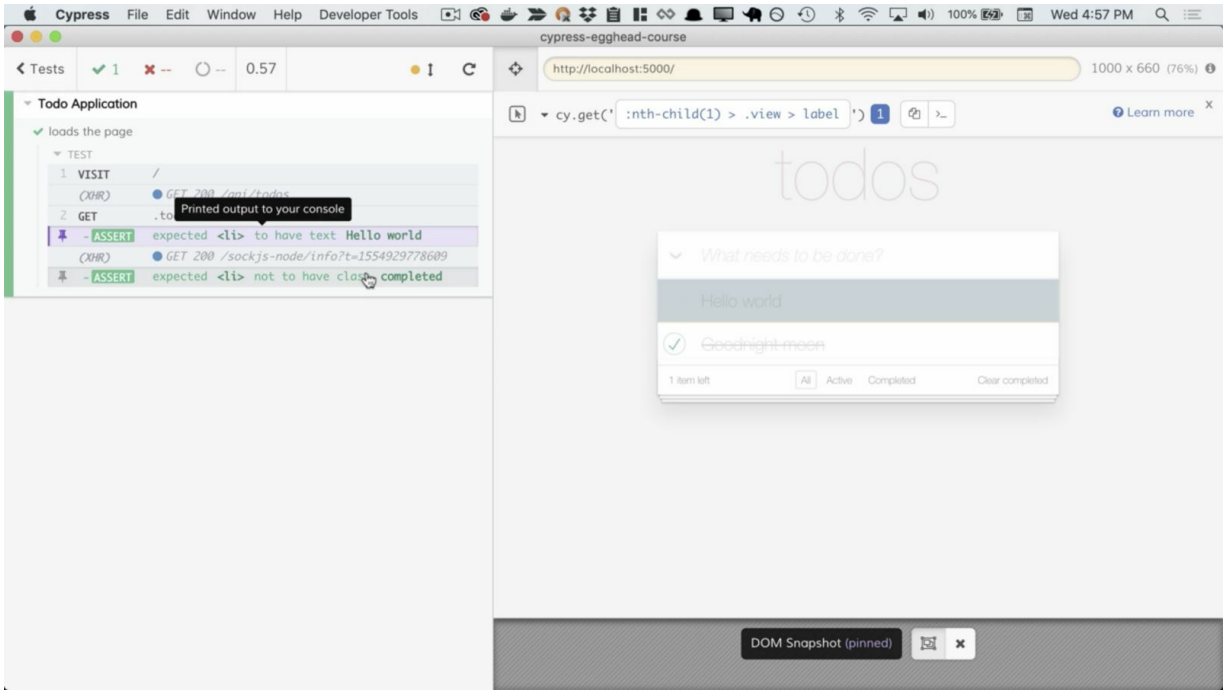


This is part of what makes Cypress so reliable. You don't have to write any code, sleeps or waits until an appropriate moment when the repan has happened. Cypress handles all of that for you.

Let's go ahead and undo that. We'll keep chaining some more `should`'s off of this to see how that works. Let's say it should not have the CSS class completed, `'not.have.class', 'completed'`, because we haven't yet completed our todo.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('.todo-list li:nth-child(1)')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
```

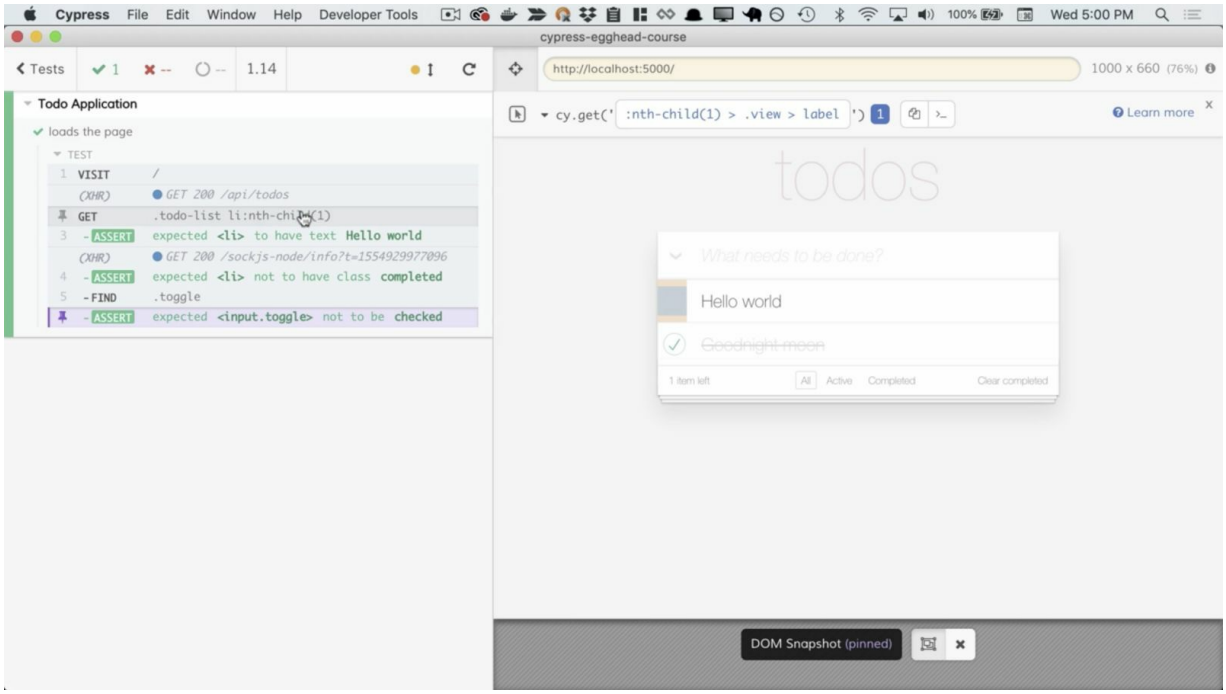
Back in Cypress, we see the chaining like this works. Both assertions ran one after the other, and both passed.



Cypress also bundles jQuery. We can use the jQuery method `find` or similar methods to navigate our UI. For instance, we can find the `toggle`, which is a child of this list item, and then we can say that it `.should('not.be.checked')`.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('.todo-list li:nth-child(1)')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')
```

Back in Cypress, we can see the way this gets targeted.

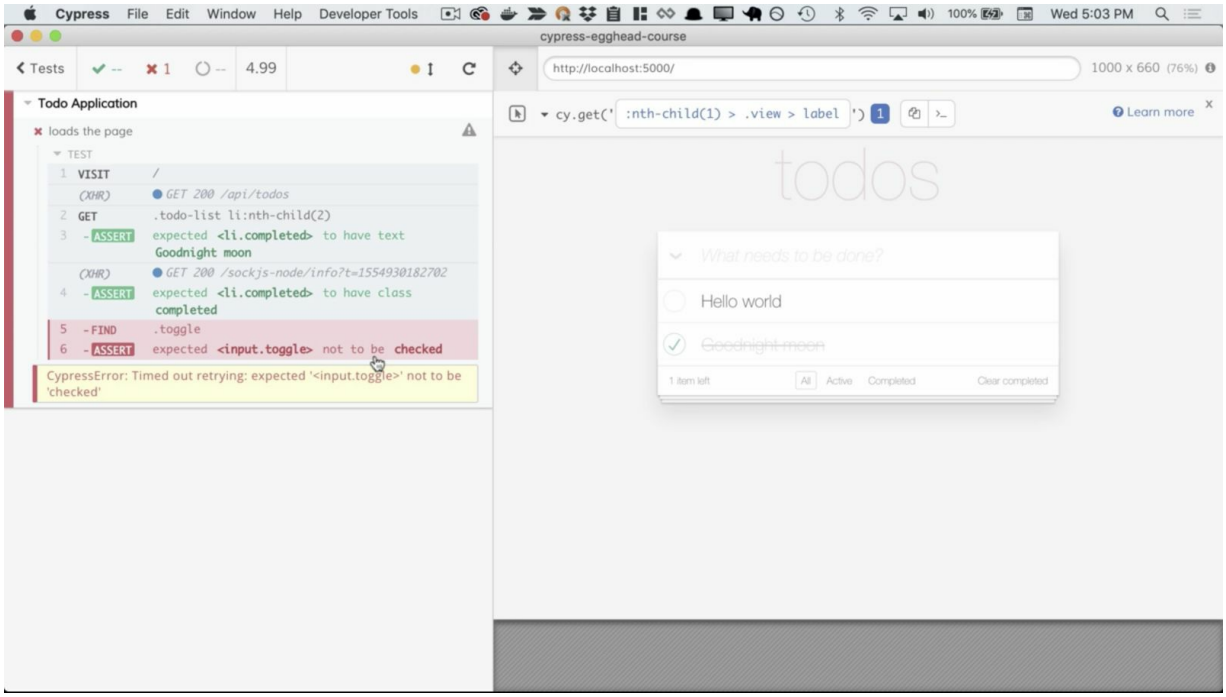


For instance, first, targeting this first list item, and then we can see it targeting the toggle, and asserting not checked on that. It's important that we run this check on the toggle because a toggle has a check or not check attribute, but the list item does not.

To see this example in action, let's switch to the second list item which has the text, `Goodnight moon`. We know that this does have the class completed because we already completed the todo.

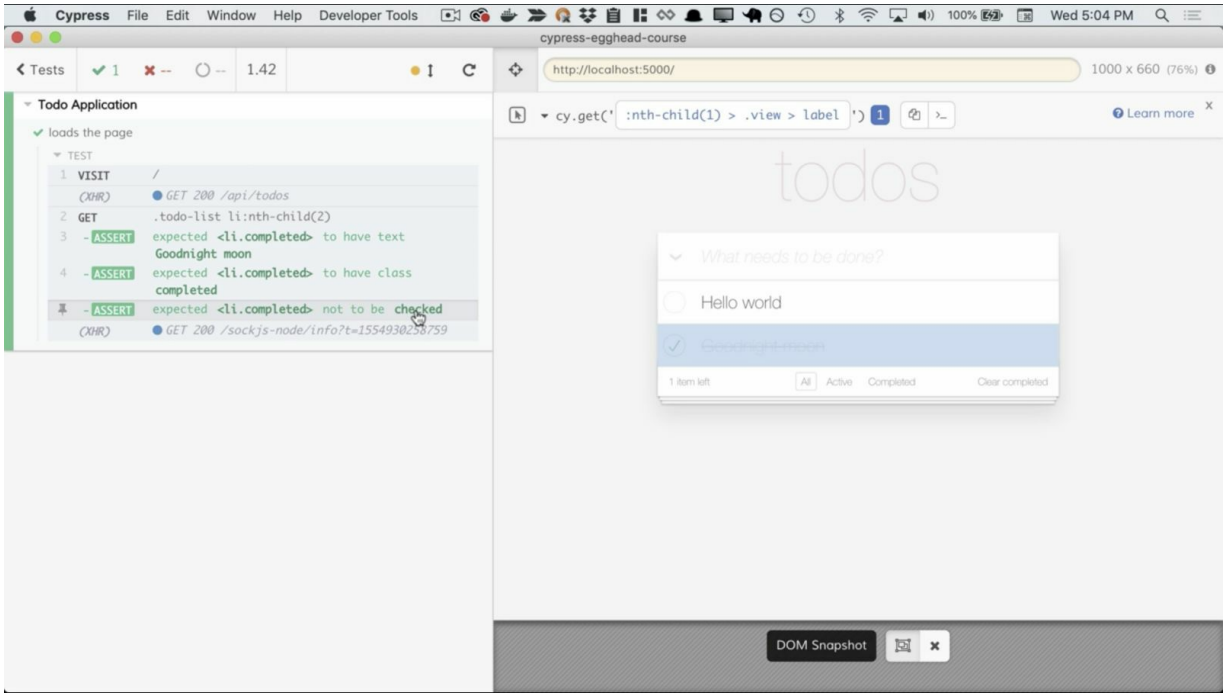
```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('.todo-list li:nth-child(1)')
          .should('have.text', 'Goodnight Moon')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')
```

What happens if we still say it should not be checked. Reopen Cypress. We expect the toggle not to be checked, and we see this fails.



However, if we run this same assertion on the list item and not the toggle, the test will pass.

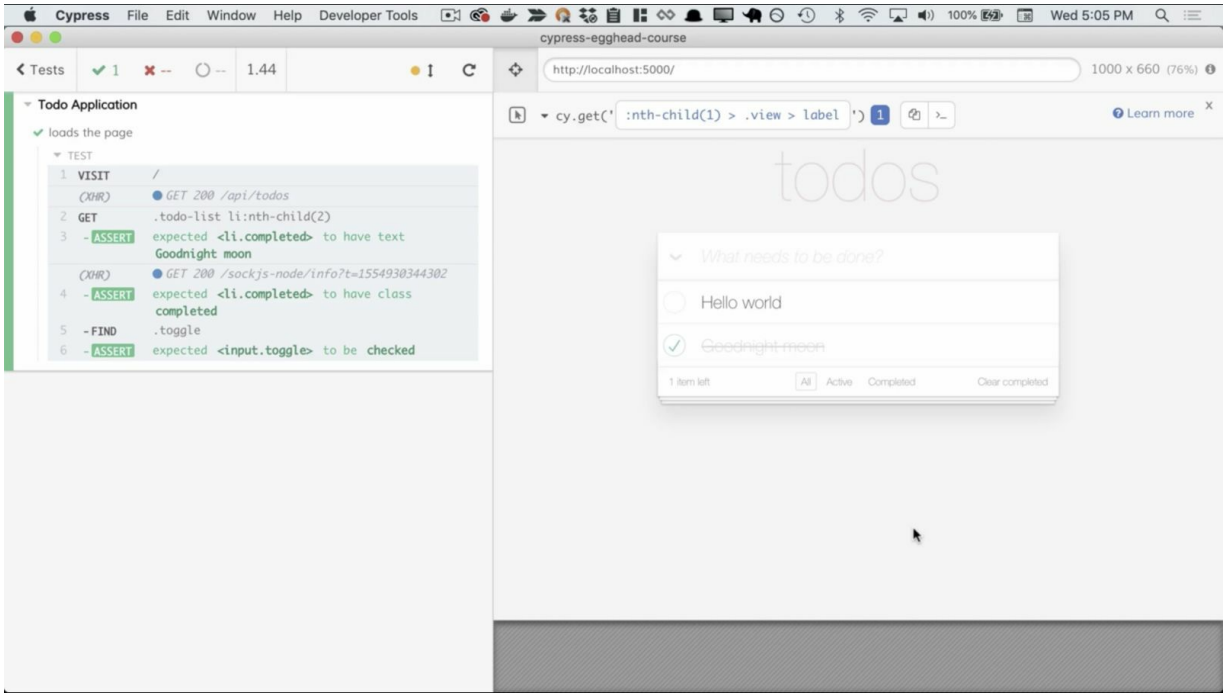
```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('.todo-list li:nth-child(1)')
          .should('have.text', 'Goodnight Moon')
          .should('not.have.class', 'completed')
          // .find('.toggle')
          .should('not.be.checked')
```



The reason is because the list item doesn't have a checked attribute in the first place, so of course, it's not checked. This is why it's important to double check that the inverse assertion of any assertion you make does in fact fail. Since we know the inverse fails in this case, we can retarget the toggle and make sure that `be.checked` passes.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('.todo-list li:nth-child(1)')
          .should('have.text', 'Goodnight Moon')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
```

Since it does, we know we've targeted the right element.



Use the Most Robust Selector for Cypress Tests

In this lesson, we're going to discuss best practices for using selectors. The selectors we used in the previous example are actually fairly brittle.

First, because they are dependent on their implementation. If we change the implementation of to-do items from something other than a list item to anything else, we would break our task.

Second, it's because they're coupled to CSS selectors which are traditionally used for styling and could reasonably be changed at any time.

Third, these are coupled to their specific location in the DOM. First, we know that the list item must be the child of a to-do list in order to be selected, and then it also must be a specific nth-child.

We want to be declarative about what we're selecting and that the selection is being used for testing so that if we change the underlying implementation of any of these objects, we won't

break our task inadvertently.

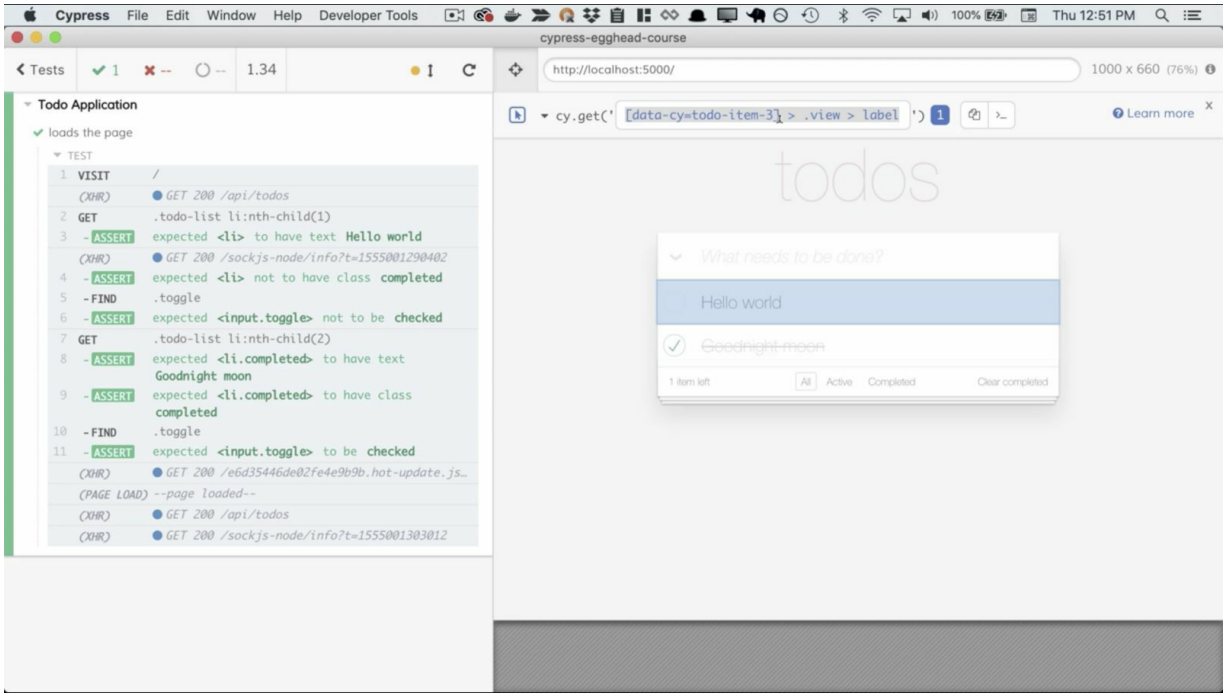
If we open up `src/components/Todoitem.js`, we can add a `data-cy` attribute. data-cy is exclusively used for testing. We can communicate to our teammates that we intended to be used in a task.

By using the unique `id` of the todo, we can guarantee that this will have a unique `data-cy` attribute and that it will be easy for us to target regardless of where it is in the DOM or what its implementation is.

todoitem.js

```
return (
  <li data-cy={`todo-item-${todo.id}`}
className={classnames({
    completed: todo.completed,
    editing: this.state.editing
  })}>
    {element}
  </li>
```

If we visit Cypress, we can click on this little target to the left of the search bar, to open the selector playground.



The selector playground will make recommendations as to the best way to target each element. For instance, if we want to target an item, we can click on it and see that it recommends the data-cy todo item-3, `[data-cy=todo-item-3] > .view > label`.

The selector playground isn't perfect. Because there're so many html elements on the page, we may not be able to target the exact element we want. In this instance, we see that we're targeting the label and not the higher level to-do item.

We still have to use some commonsense, know our html structure, and couple that with Cypress's recommendations in order to target the correct item in the most effective way.

If we return to our code, we know that the best way to target this now is to use `('[data-cy=todo-item-3]')`. If we save this, we can reopen Cypress and see that it's still targeted.

We can do the same thing on line 11 with `('[data-cy=todo-item-4]')`. If for some reason you're unable to use the data-cy attribute, the next best option is to use `contains` which looks for

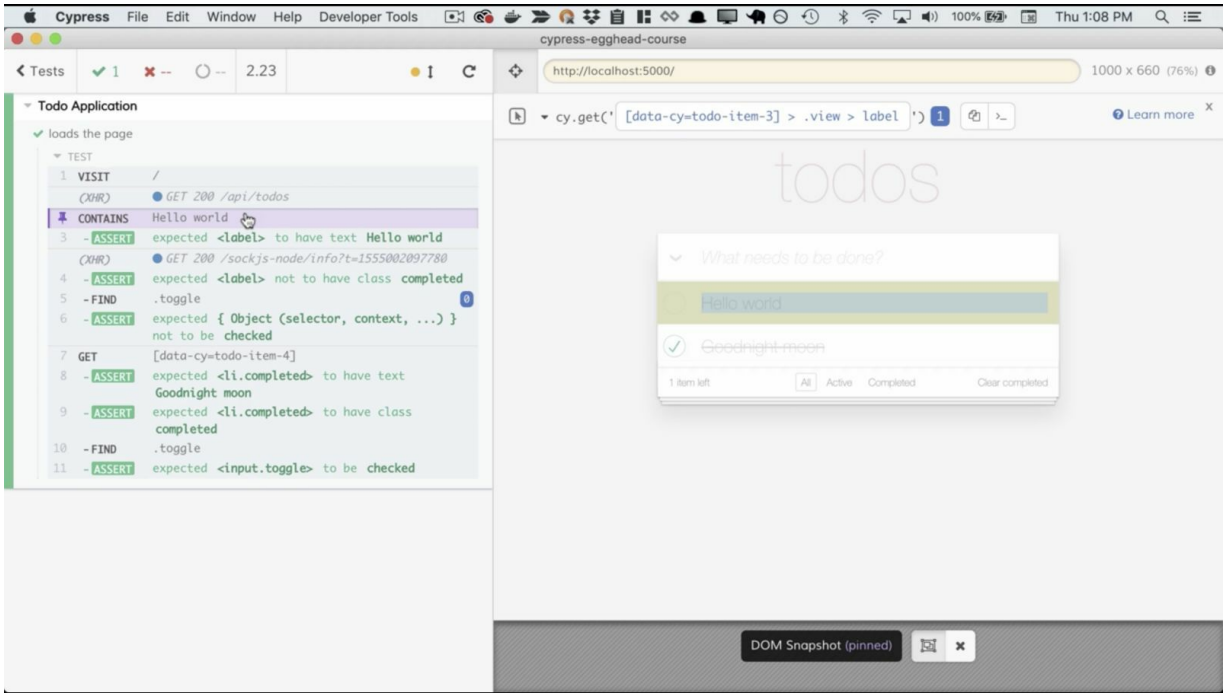
specific text on the page and targets it.

In this case, we'll target `'Hello world'`.

todos.spec.js

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('Hello world')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

When we return the Cypress, we'll see that we have in fact targeted an element containing the text Hello World.



Although if we take this approach, we really have to be careful and pay attention to what we've actually selected.

In this situation, we see that we selected the label, and not the overarching list item. This works when we're checking for the text Hello World but the list item, not the label, is what's able to have the class completed or not have it.

In our second assertion, we're actually looking at a false positive. Finally, when we try to find the toggle, which would be a child of the list item but isn't a child of the label, we discover that we found nothing.

When we run the assertion not to be checked or nothing, that's also a false p ositive. We can check array methods to navigate our DOM tree and find the specific element we want. We can see pretty quickly why it's much better to use a unique data-cy attribute than anything else.

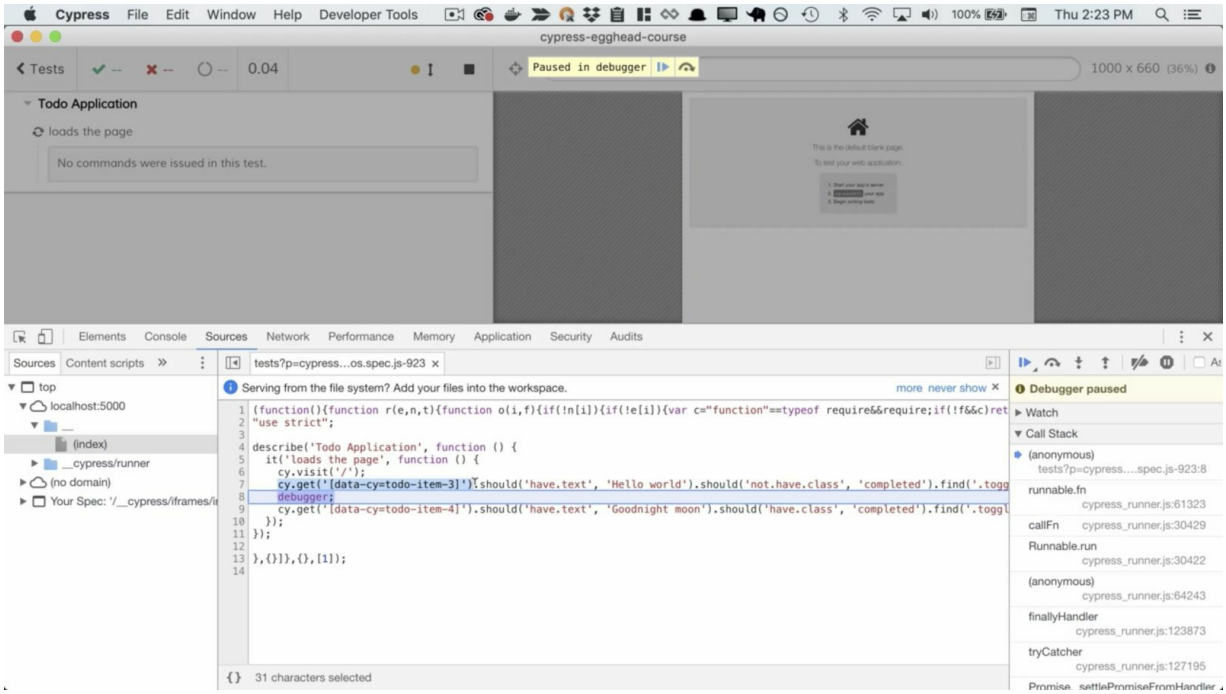
Debug and Log with Cypress

In this lesson, we're going to talk about debugging and locking, which are critical in any testing environment. We're going to use this as a way to take a look at Cypress' asynchronous nature. Let's start off by trying to add a `debugger` as we normally would.

todos.spec.js

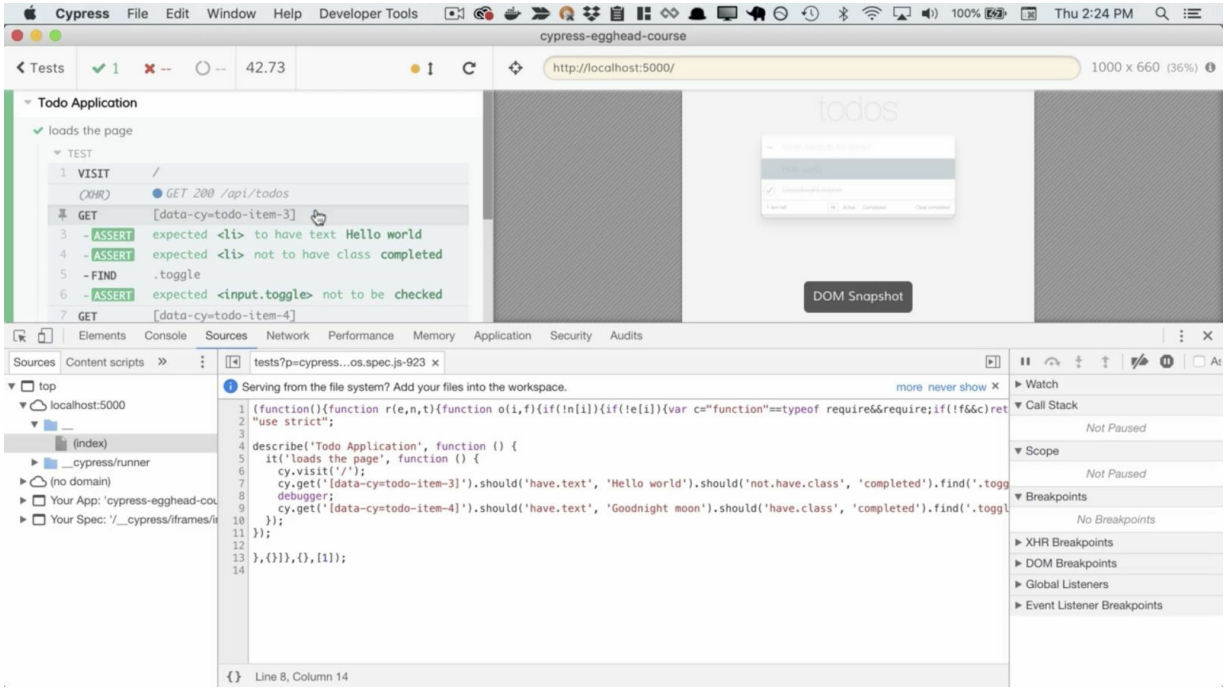
```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.get('[data-cy=todo-item-3]')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        debugger

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

If we visit Cyprus and pop open our console so that we make sure we hit our debugger, we'll see that the page actually hasn't loaded yet. This is strange because the debugger is the third statement.



The first statement is that we visit the page. The second is that we get the first to-do item. Then we should be hitting the debugger.

What's going on? If we press play, now we finally see Cypress load the page. We see it visit, get the first to-do item. This gives us a hint as to what's really happening with Cypress.



Cypress code is a lot like async and await. When we hit the `cy.visit` method, we don't actually visit the page yet. Instead, Cypress on-queues the event to happen later.

Think of this like pushing visit into an array and then moving on immediately to `cy.get`. cy.get is also asynchronous. It pushes that into the array, followed by the `should`, then the `should`, then the `find`, then the `should`. These will all be executed in order, but they won't be executed yet.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        // [ visit, get, should, should, find,
should, get... ]
        cy.visit('/')

        cy.get('[data-cy=todo-item-3]')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        debugger

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

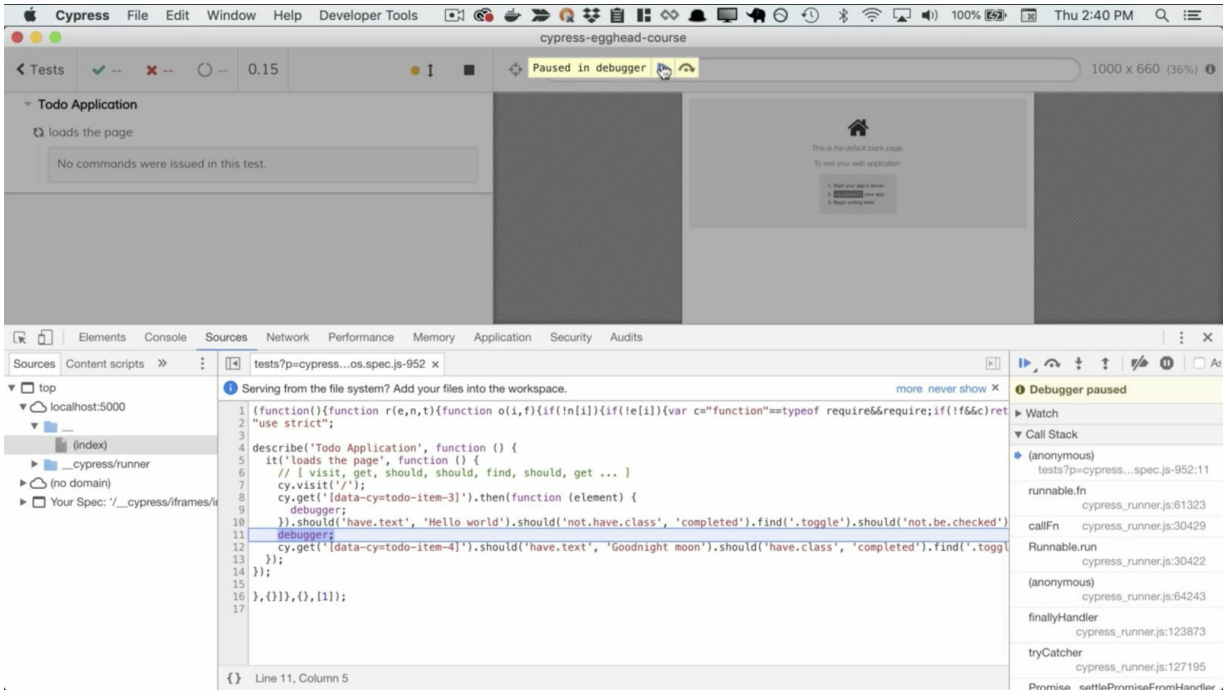
Next we move on to the debugger statement. This is the first synchronous line of code that actually runs. That's why when we hit the break point in our console, we haven't visited the page.

We haven't run a get, a should, or any of the other Cypress code. We hit that debugger. Nothing is loaded. Next we hit the second `cy.get`. We proceed on-queuing events in the same way as before. Finally, we start popping things off the queue. We visit the page first. Then we run cy.get.

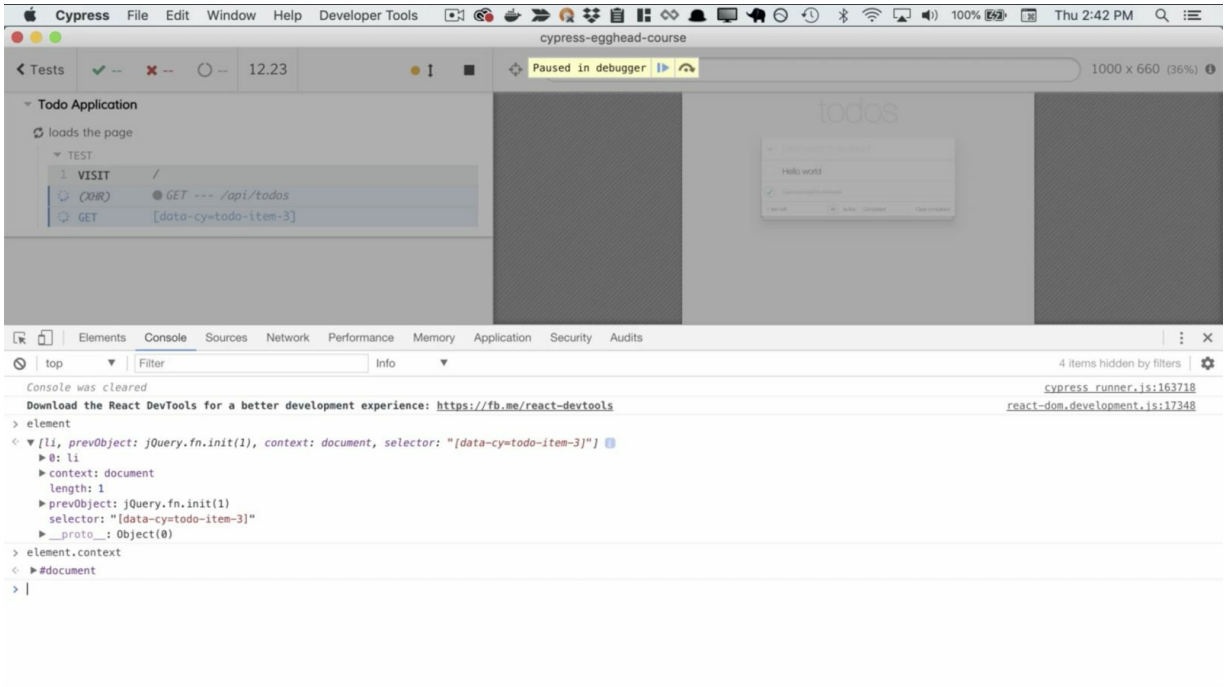
If we want to see what `cy.get` has actually gotten, we have to put our `debugger` in here. `then` takes a callback, so we can say that this will be the `element`. If we put our `debugger` statement in here and save it out, now we can reopen Cypress and see what happens.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        // [ visit, get, should, should, find,
should, get... ]
        cy.visit('/')

        cy.get('[data-cy=todo-item-3]')
          .then((element) => { debugger; })
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        debugger

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

We hit our first debugger from before. We see the page isn't loaded yet.



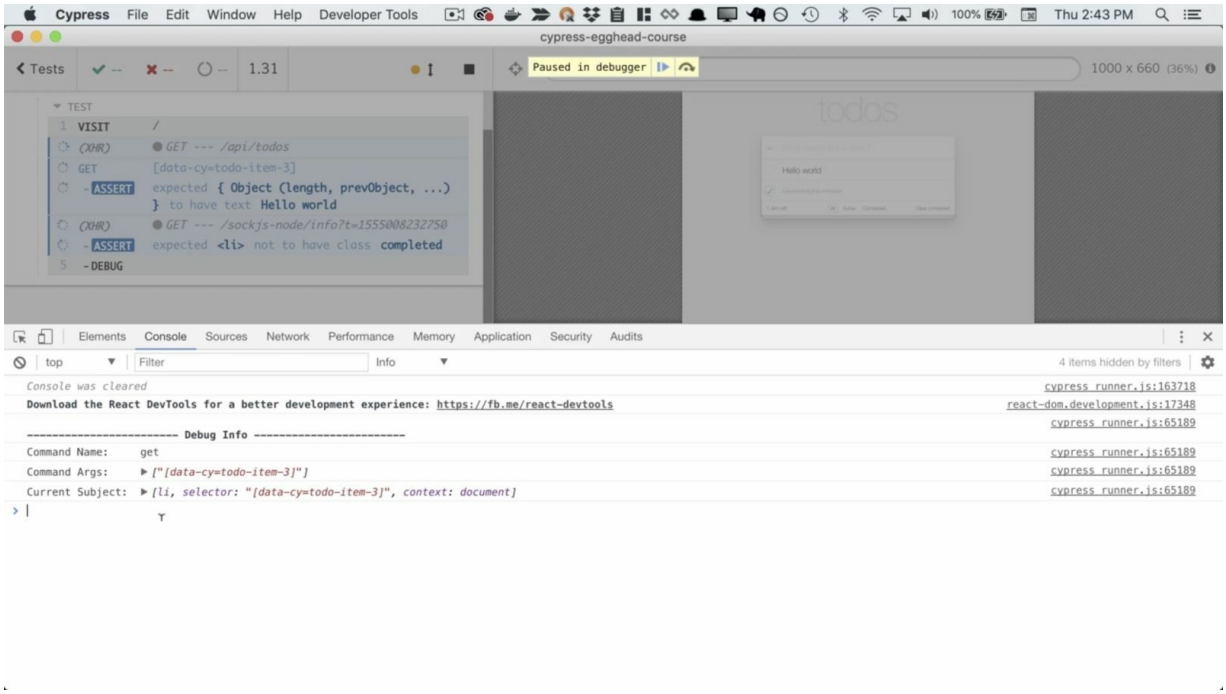
That's exactly what we expect. Let's press play and see what we get. We hit the second debugger statement and were fed the element. We can access the element, which we see as the jQuery selector. We can interact with it as we normally would.



Of course, this is a little tedious. Cypress provides a helper method called `.debug`.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        // [ visit, get, should, should, find,
should, get... ]
        cy.visit('/')

        cy.get('[data-cy=todo-item-3]')
          .debug
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

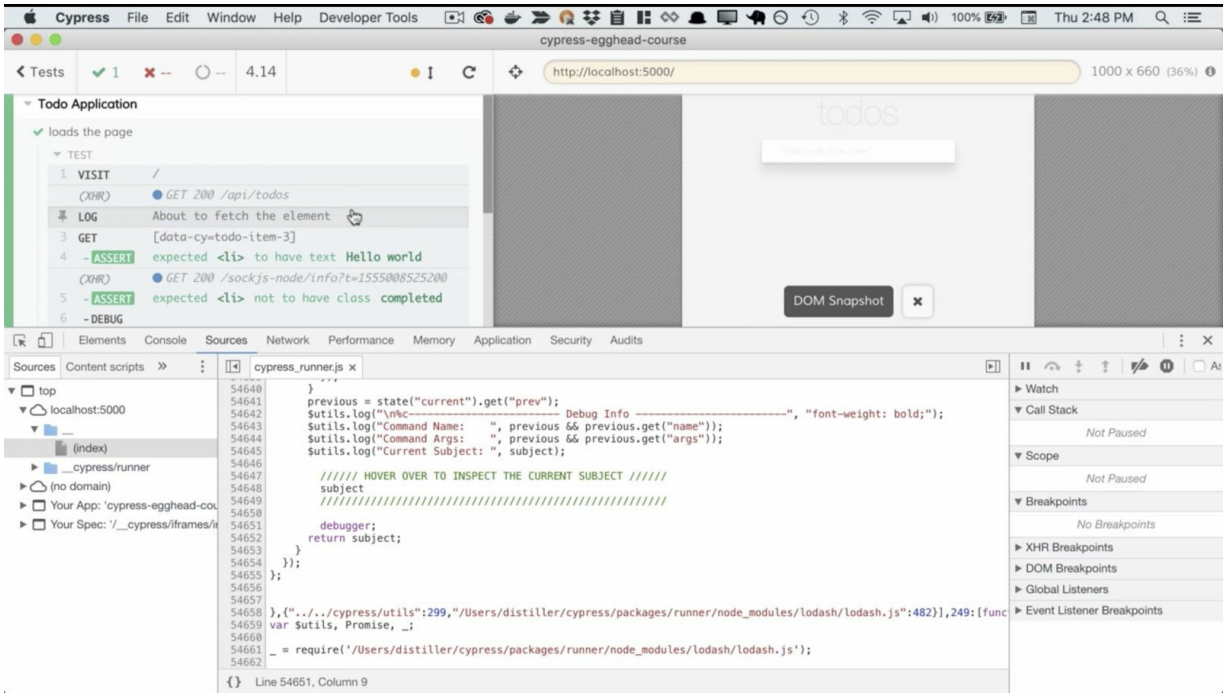
If we run `.debug`, it'll be the same thing as putting a debugger in, except now we'll be fed the subject by the name of subject, which we can inspect in the console, or as a named variable.



Similarly, if we want to log something out, we can use `cy.log` and say something like, `'about to fetch the element'`.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        // [ visit, get, should, should, find,
should, get... ]
        cy.visit('/')

        cy.log('about to fetch the element')

        cy.get('[data-cy=todo-item-3]')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

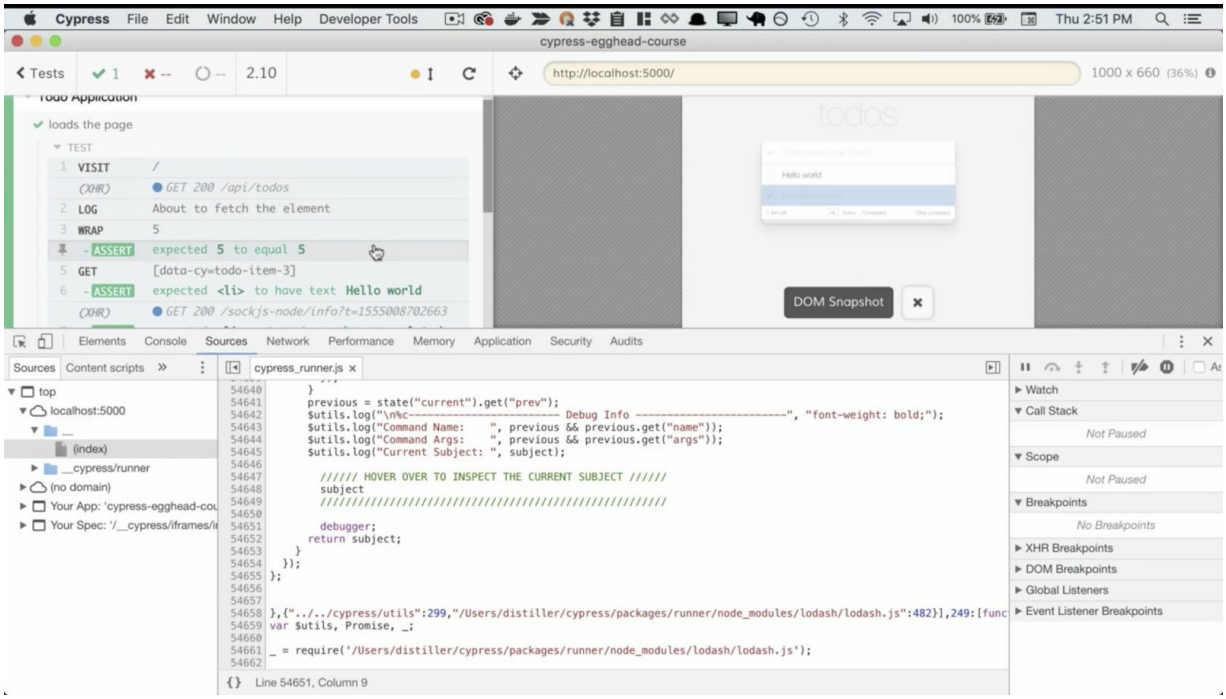
Now if we reload the page, we'll see cy.log in our list of statements.



Arbitrary code can be on-queued into Cypress using `cy.wrap`. For instance, we can wrap the number `5` and say it `should` equal `5`.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        // [ visit, get, should, should, find,
should ]
        cy.visit('/')

        cy.log('about to fetch the element')

        cy.wrap(5).should('eq', 5)
        cy.get('[data-cy=todo-item-3]')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

If we reload, we'll see this assertion ran, and it ran in the same order we expected it to.

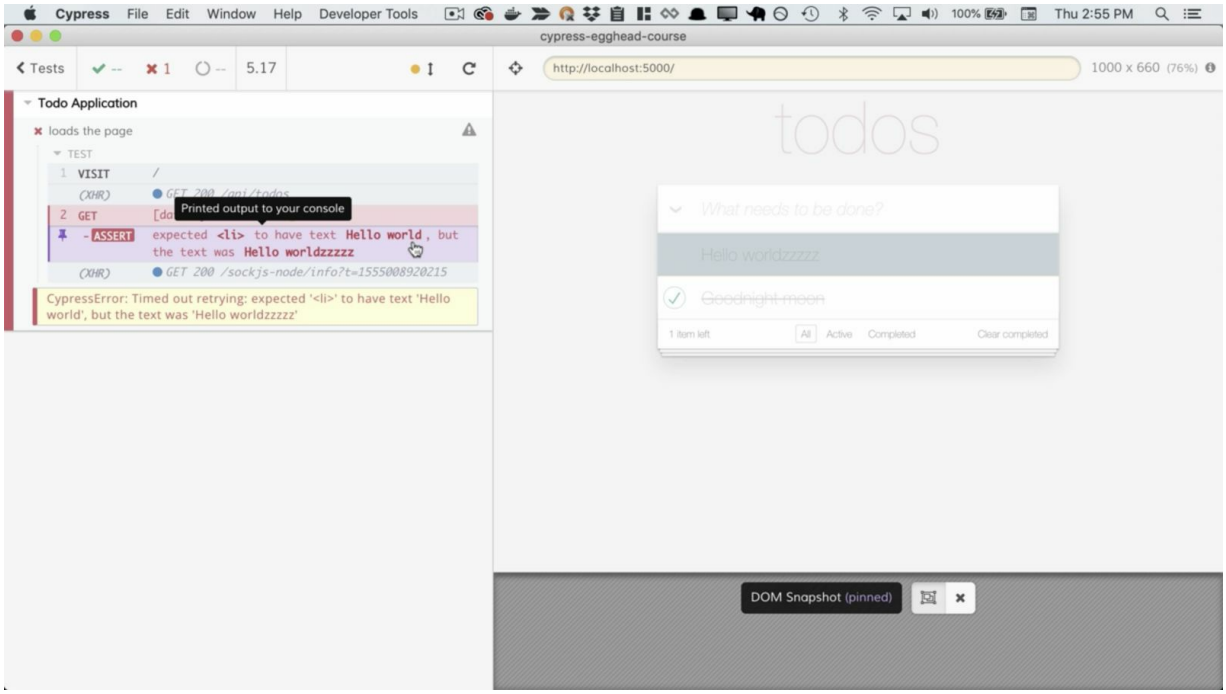


Mock Your Backend with Cypress

In this example, we're going to talk about mocking the backend. We can discover the reason we need to mock the backend by visiting our real page at `localhost:5000`. Let's switch over to Chrome.

Here, we can edit our first to-do. Let's make this Hello, World with a bunch of Zs, `Hello worldzzzzz`.

Now let's return to Cypress, refresh, and we'll see that we have a failing test. That's because we expected this to have the text Hello world, but it's actually using the real data from our real backend, which we've just changed.



There are two strategies we can use to address this. One is stubbing all of our network requests. The other is that we can interact with the real backend, but we have to have a database seeding abstraction, as well as a totally isolated testing environment.

In this lesson, we're just going to talk about the first one, and in later lessons, we'll show how to seed the database. To start, let's tell Cypress simply that we want to begin stubbing network requests.

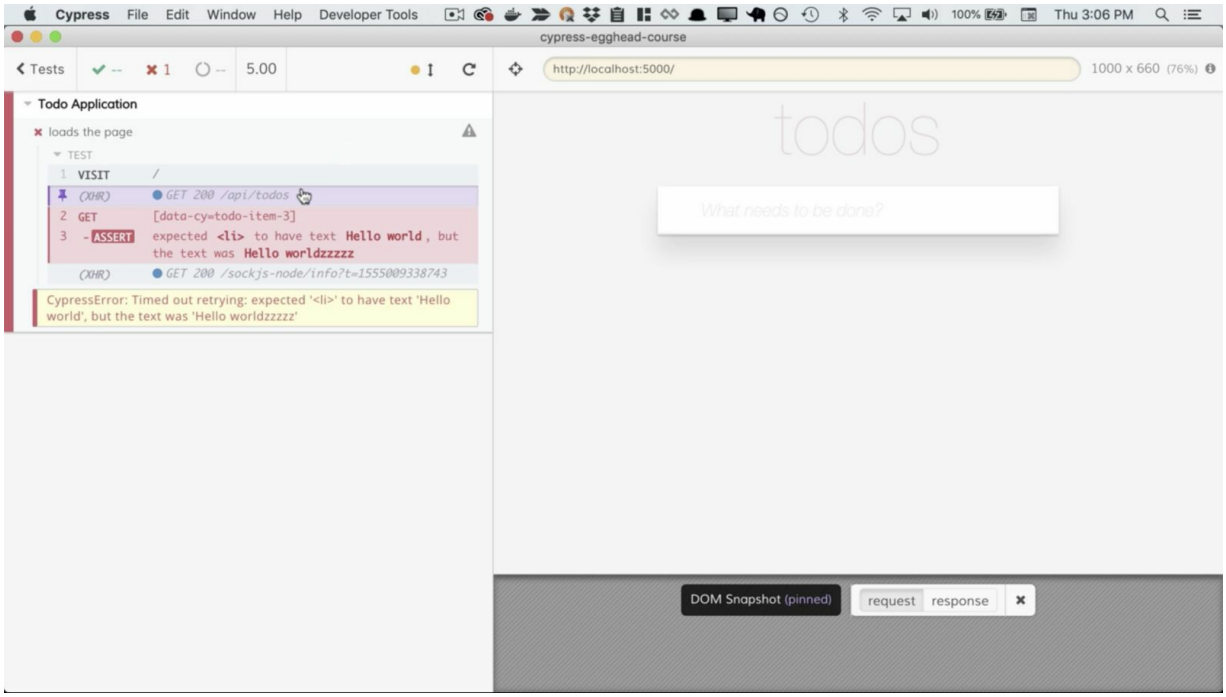
To do that, we can call `cy.server`.

todos.spec.js

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.server()
        cy.visit('/')

        cy.get('[data-cy=todo-item-3]')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

By default, this doesn't stub out any requests. They'll all still continue to pass through to the backend, but we do need to call cy.server before we can start mocking routes or spying on them.

To actually start mocking our routes, we need to know what we need to mock. Let's go back to Cypress and take a look at our XHR request. We see that we perform a `GET` to `api/todos`. That's hitting the real backend. Let's mock that out.



Let's call `cy.route('/api/todos')`. By default, this is a GET request.

Next, we'll return an array of hashes. We can actually see what's in our real backend by looking in `db.json`. Let's go ahead and just copy this out and paste it in. We did expect this not to have so many Zs or any at all, so let's save it.

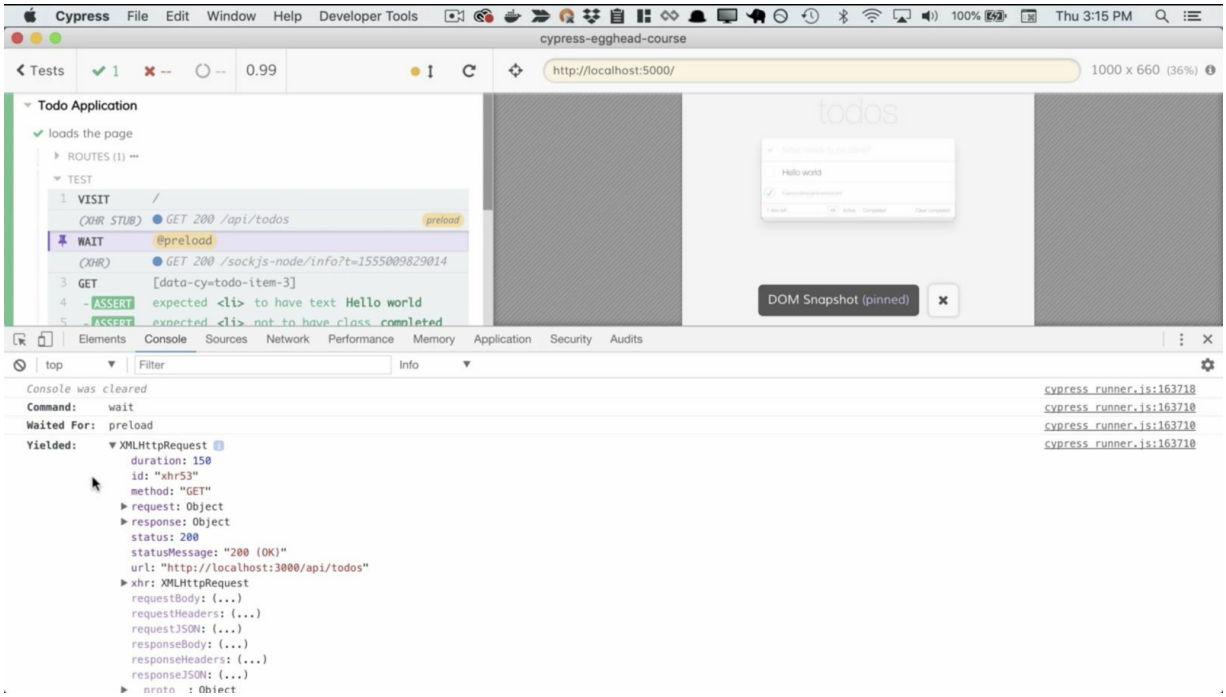
```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.server()

        cy.route('/api/todos', [
            {
                "text": "Hello world",
                "completed": false,
                "id": 3
            },
            {
                "id": 4,
                "completed": true,
                 "text": "Goodnight moon"
            }
        ])
        cy.visit('/')

        cy.get('[data-cy=todo-item-3]')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

Next, we have to `cy.wait` for this route to load, which means that we have to give it an alias. We can say `as('preload')` and we'll `cy.wait('@preload')`.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.server()

        cy.route('/api/todos', [
            {
                "text": "Hello world",
                "completed": false,
                "id": 3
            },
            {
                "id": 4,
                "completed": true,
                 "text": "Goodnight moon"
            }
        ]).as('preload')

        cy.visit('/')

        cy.wait('@preload')
...
```

We'll save it and return.

Now we can see that we waited for our preload, which returned our mock data and made all our tests pass again. We can also do a deep dive on this request by clicking on the preload, opening our console, and checking our XML HTTP request, where we can see everything like status code, response, body and headers, and we can inspect this to make sure it's what we just sent. It is.



Assert on Your Redux Store with Cypress

Right now we're not able to make assertions at all levels of the stack. We'd really like to be able to. When breakdowns happen, we want to know why they happened and where they happened. Is something not painted on the UI because of a UI issue? Is it because of the underlying representation in the redux store? Right now we don't know.

We can't make any assertions on our front-end stores of any kind. Let's go ahead and make that possible. We'll start by opening `src.index.js` which is where in our application we've defined the store. We'll say, `if(window.Cypress)` which means we're in a Cypress testing environment. Then we know we're safe to expose the store on the window.

index.js

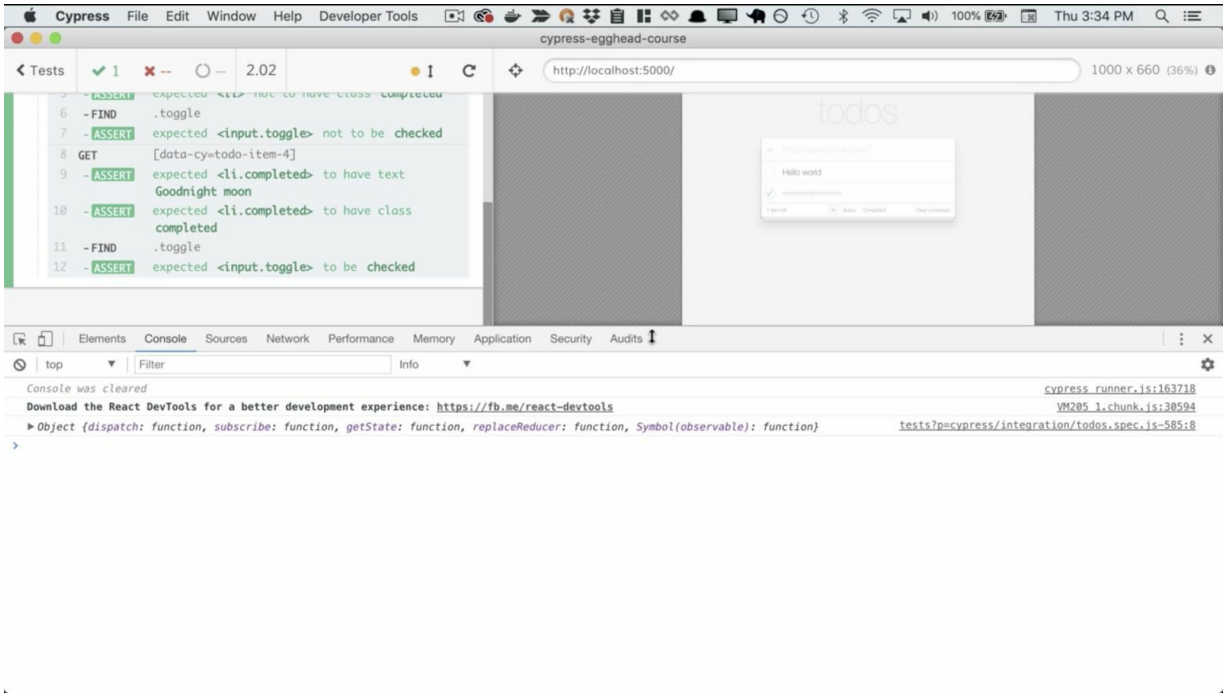
```
import React from 'react'
import { render } from 'react-dom'
import { createStore, applyMiddleware } from
'redux'
import createSagaMiddleware from 'redux-saga'
import { Provider } from 'react-redux'
import { rootSaga } from './sagas/TodoSagas'
import App from './components/App'
import reducer from './reducers'
import 'todomvc-app-css/index.css'

const sagaMiddleware = createSagaMiddleware()
const store = createStore(reducer,
applyMiddleware(sagaMiddleware))
sagaMiddleware.run(rootSaga)

if(window.Cypress) {
    window.store = store
}

render(
  <Provider store={store}>
    <App store={store} />
  </Provider>,
  document.getElementById('root')
)
```

We then go back to `todos.spec.js`. Cypress exposes the window as `cy.window` which will help us access it. Remember that Cypress code is asynchronous. We don't just have access to window here. Let's take a call back and `{console.log($window.store)}` just to make sure we've set everything up properly.

todos.spec.js

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')

        cy.window().then(($window) =>
{console.log($window)})

        cy.get('[data-cy=todo-item-3]')
          .should('have.text', 'Hello world')
          .should('not.have.class', 'completed')
          .find('.toggle')
          .should('not.be.checked')

        cy.get('[data-cy=todo-item-4]')
          .should('have.text', 'Goodnight moon')
          .should('have.class', 'completed')
          .find('.toggle')
          .should('be.checked')
    })
})
```

It looks like we have.



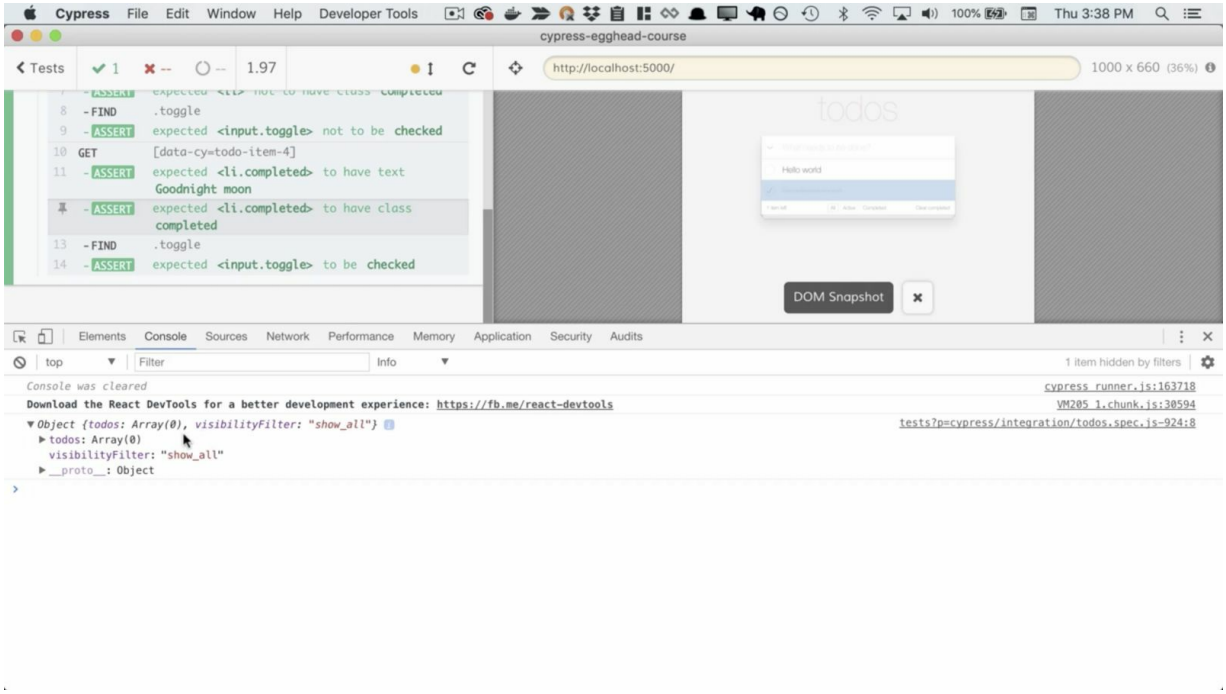
Cypress exposes a way to access these properties by using `its`. We can say, `.its('store')`.

What we want is to be able to make assertions against the state of the store. In order to get the state of the store, we would normally call, `getState` which is a function, not a property like store. In order to do this, we can call, `.invoke`.

We can call, `.invoke('getState')` Now we should have our state. Let's take a look. We'll `console.log($state)`.

```
describe('Todo Application', () => {
  it('loads the page', () => {
    cy.visit('/')


cy.window().its('store').invoke('getState').then(
($state) => { console.log($state)})
    ...
```

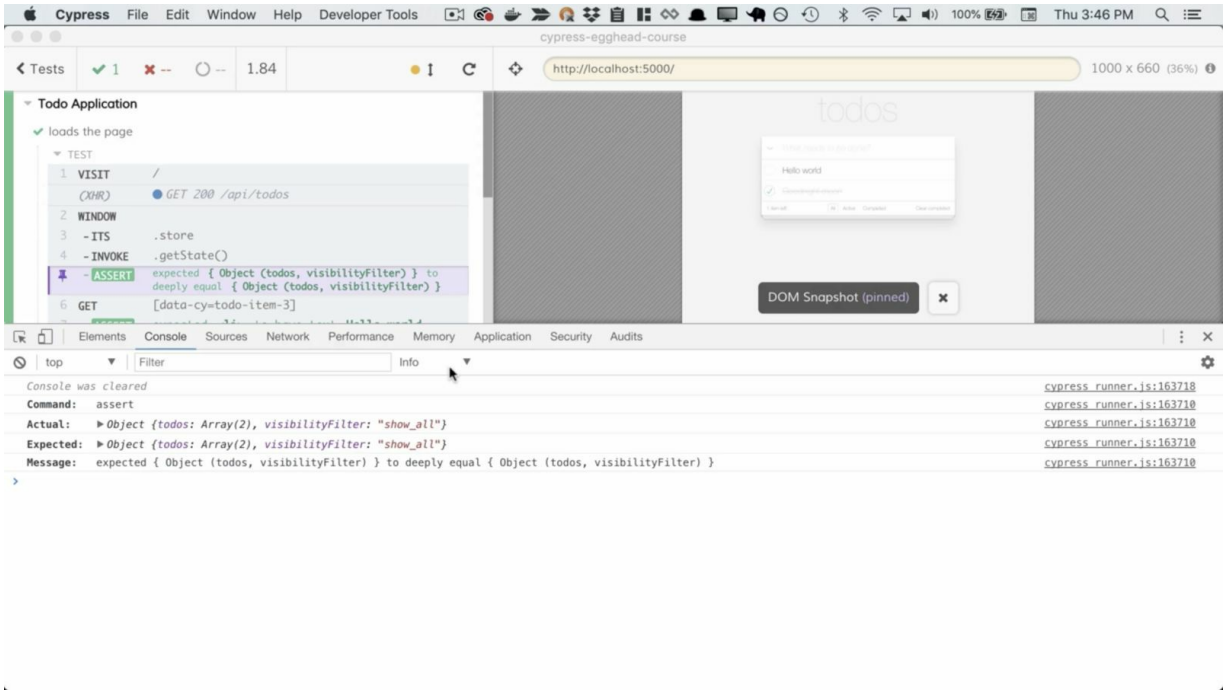
Great, this looks like the state we expect because this is before we've waited for any pre-load.



Instead, let's make an assertion on the store. We can say it `should('deep.equal', todos`, which will have the state of the to-dos array and `visibilityFilter`, which will be, `show_all`. We want to assert that the to-dos have loaded. Let's go ahead and paste in the to-dos that we expect to be there and save it out.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.visit('/')


cy.window().its('store').invoke('getState').shou
ld('deep.equal), {
            todos: [{
                "text": "Hello world",
                "completed": false,
                "id": 3
            }
            {
                "id": 4,
                "completed": true,
                "text": "Goodnight moon"
            }
          ],
          visibilityFilter: 'show_all'
        })
        ...
```

Let's revisit Cypress and see what happens. In fact, this test still passes because Cypress waits for the to-dos to load just like Cypress will wait until the specified time-out for UI elements to load.



It will also wait until the specified time-out for our stores state to match what we expect. This makes it really easy to wait for an assert on asynchronous behavior.

Create Custom Cypress Commands

In the previous video, we access the state of the Redux store with this rather cumbersome call. If we want to reuse this in our application over and over again, it would probably be easier if we could call `cy.store`.

todos.spec.js

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.server()
        cy.route('/api/todos', [
          {
            "id": 1,
            "text": "Hello world",
            "completed": false
          },
          {
            "id": 2,
            "text": "Goodnight moon",
            "completed": true
          }
        ]).as('preload')

        cy.visit('/')
        cy.wait('@preload')

        cy.store()

cy.window().its('store').invoke('getState').its(
'todos').should('deep.equal', [{
            id: 1,
            text: 'Hello world',
            completed: false
          }, {
            id: 2,
            text: 'Goodnight moon',
            completed: true
          }])
...
```

Let's turn this whole thing into a custom Cypress command. We can open up our side bar, go under `cypress/support/commands.js`.

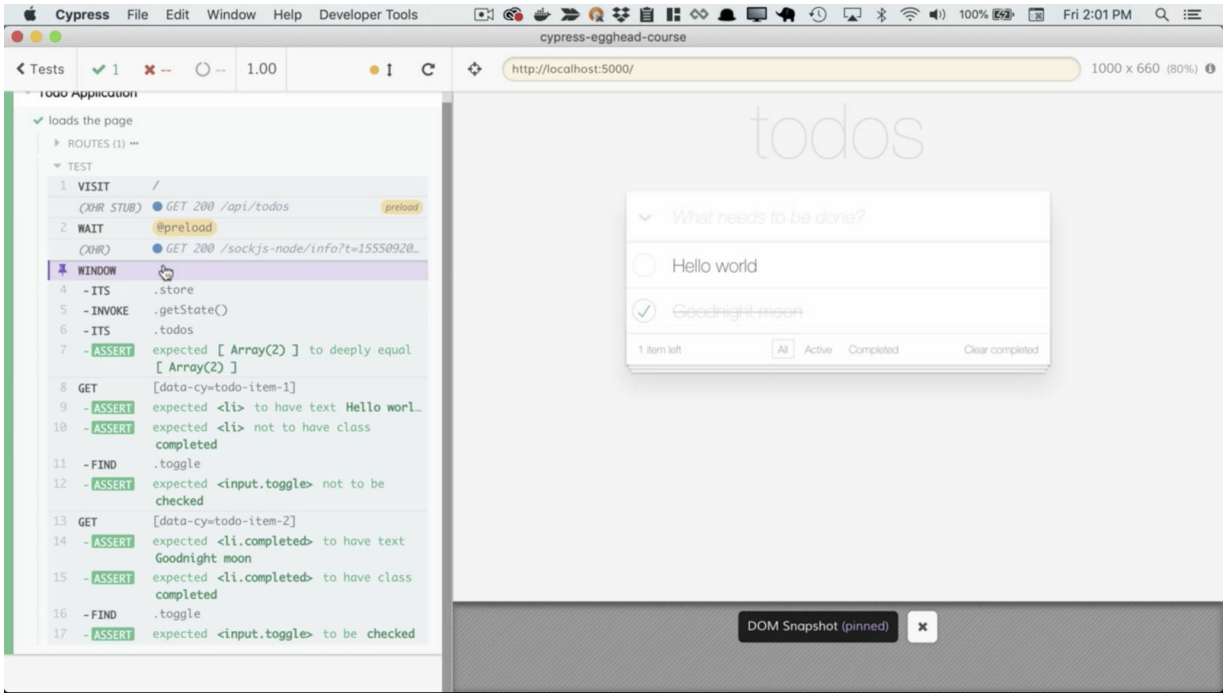
This file has a couple of examples. We can either have parent commands, child commands, or dual commands. Since we'll call store directly on cy, that'll make it a parent command. A child command would be chained off of some existing command.

Let's follow this formula. `Cypress.Commands.add("store")`. Since it's a parent command, we won't pass in the option for previous subject. Instead, our second argument is just the function we want to run when we run the command. Let's paste in our command and make sure we return it.

commands.js

```
Cypress.Commands.add("store", () => {
    return
cy.window().its('store').invoke('getState')
})
```

If we head back into Cypress and rerun our test, we'll see our custom command ran, or it seems like it did.



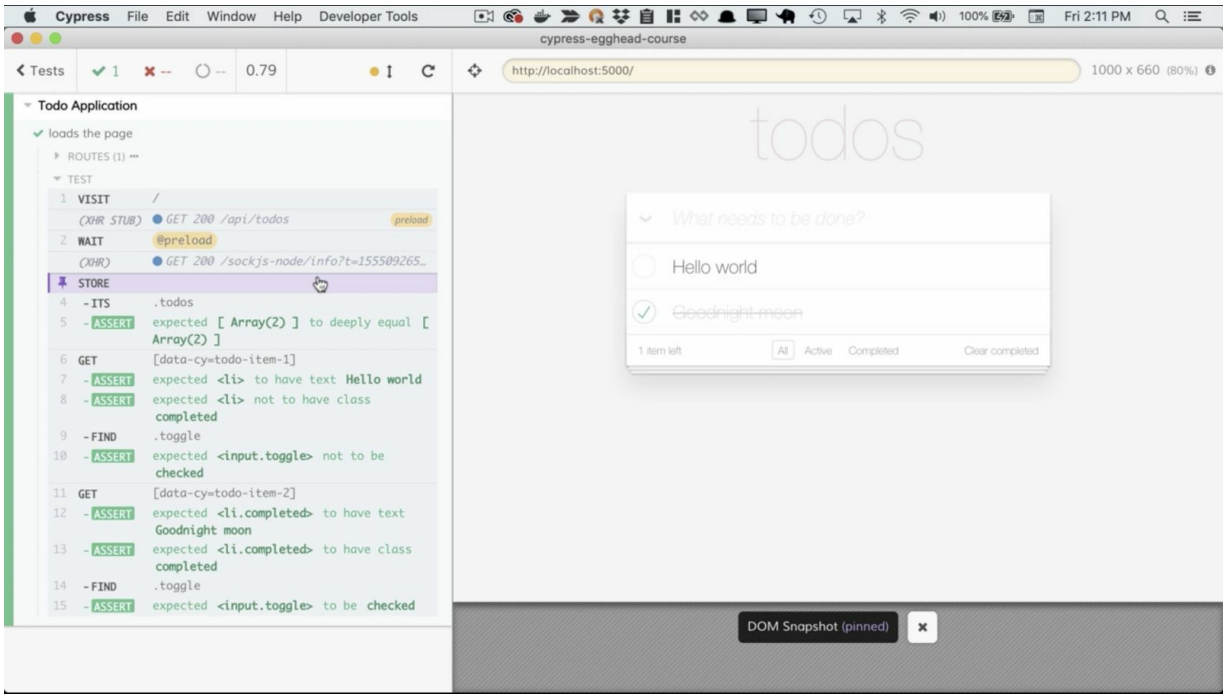
Instead of seeing store logged out here, we see all of the child commands, window, its, invoke, all of the gut contents of our method. That's going to be confusing to our end user.

It would be a lot more helpful if we could just see one log that said store and if when we clicked on it we could actually see the state of the store, the way we do with Cypress commands. How do we do that? Cypress will let us create a log by calling `Cypress.log`. We'll give it the `name` that we want, which is `store`.

Since we want to suppress the output of `cy.window`, we can pass in `log: false`. Most cy commands support log false, but `its` and `invoke` will not. Instead, let's just use a `.then`, which won't log anything. Instead, we'll grab our `$window`. We'll return `window.store.getState()`.

```
Cypress.Commands.add("store", (str = '') => {
  let log = Cypress.log({name: 'store'})
  return cy.window({log: false}).then(($window)
=> { return $window.store.getState() })
})
```

If we rerun our test, we'll see that we have logged out the name store.



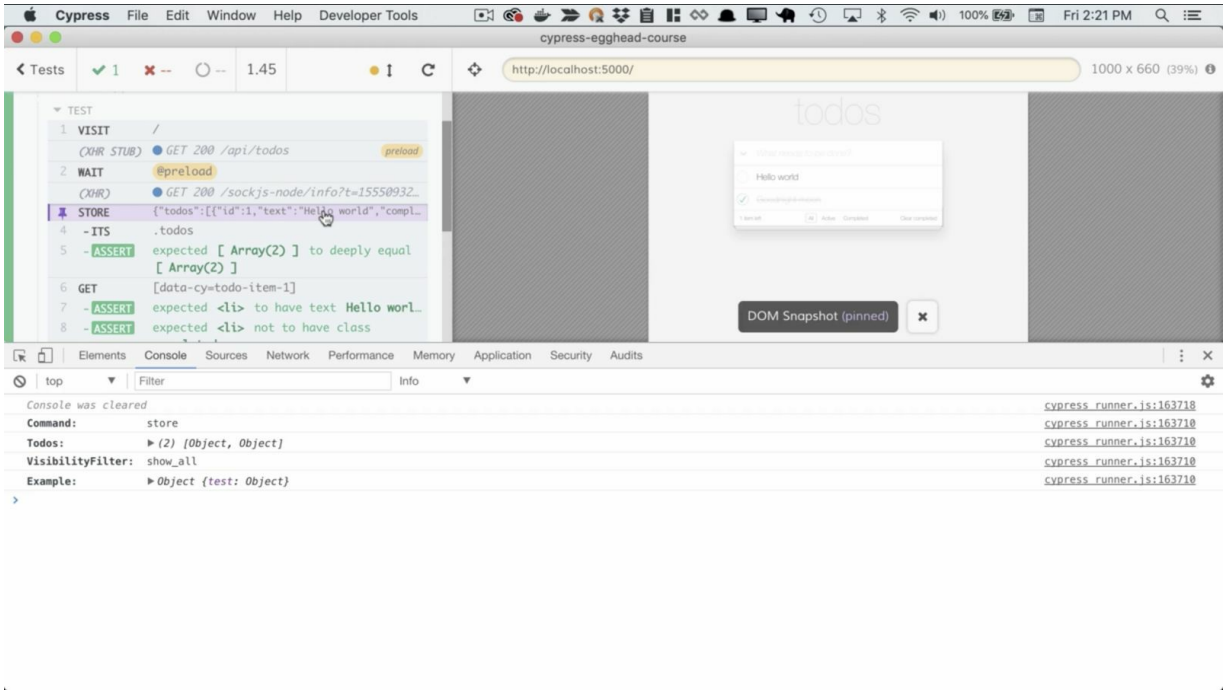
We have no child commands logged anymore. How do we actually log out the contents of the store inside the console? What we need is some way to attach the state that we just got to the log that we already created.

Fortunately, Cypress supports updating the log throughout the duration of the method. We can call `log.set`. We can pass in a `message`, which will always be a string. We'll call `JSON.stringify(state).` Then we'll pass in `consoleProps`, which will allow us to inspect the state in the console. Here, we'll

`return state`. Finally, we return the state from this whole method so that we can continue chaining Cypress commands off of it.

```
Cypress.Commands.add("store", (str = '') => {
    let log = Cypress.log({name: 'store'})
    return cy.window({log:
false}).then(($window) => { return
$window.store.getState() }).then((state) => {
        log.set({
            message: JSON.stringify(state),
            consoleProps: () => {
                return state
            }
        })

        return state
    })
})
```

When we rerun Cypress, we can see our store command now has a stringified version of the state printed out as the message.



When it's clicked on in the console, we can look through any piece of state we like, including our to-dos, visibility filter and this example of a deeply nested piece of state.

Since application state can become sprawlingly huge in any application, we want to allow the user to define which piece of state they want logged in the console and what they want to test against, so let's include a `stateName` argument.

By default, we'll return the entire state, but if `stateName.length` is greater than zero, then we'll return just the subset of state that the user asked for. To continue chaining method calls, we'll call `cy.wrap`, passing in the `state`. Don't forget to pass `log: false`. Then we'll call its with the state name. That way, we grab just the piece of state that we're interested in.

Finally, we'll call a callback which will run our `log.set` and `return state`. We'll copy these out. Let's create our callback. This will receive the `state` and run the logging functions. Then in

the case where we don't want to run `its`, we'll wrap the state, `log: false`, and then just run the callback. We'll make sure to return the two of these and clean it up.

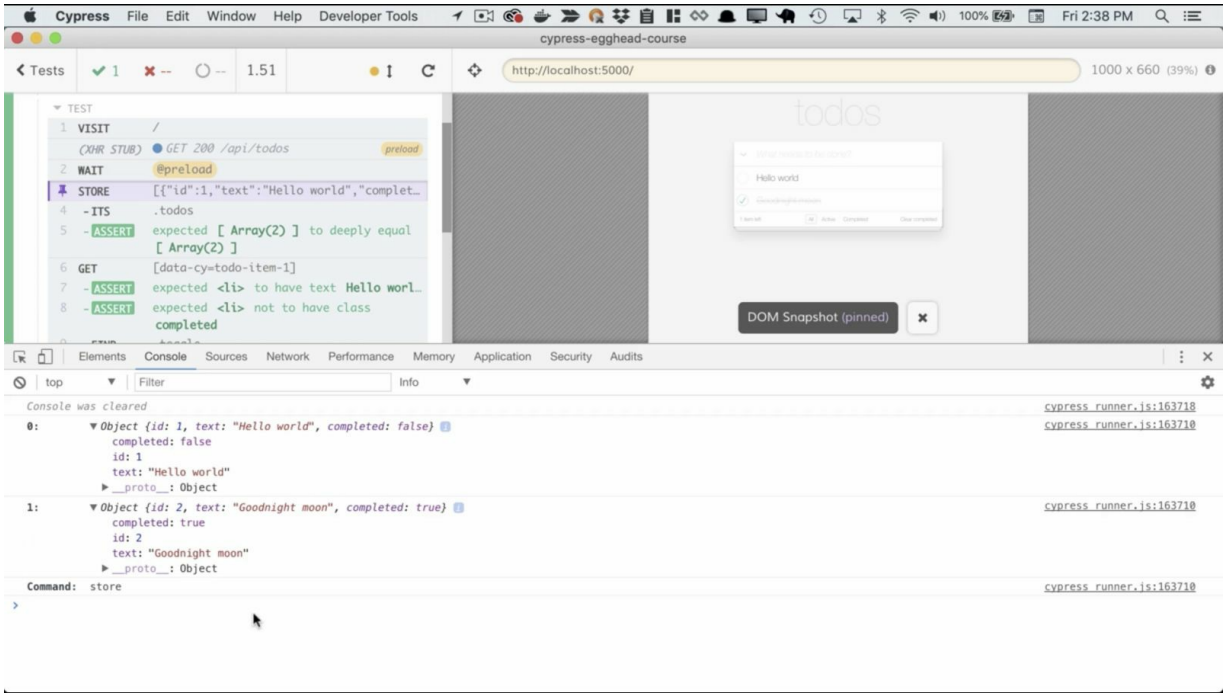
```
Cypress.Commands.add("store", (stateName = '')
=> {
    let log = Cypress.log({name: 'store'})

    const cb = (state) => {
        log.set({
            message: JSON.stringify(state),
            consoleProps: () => {
                return state
            }
        })

        return state
    }

    return cy.window({log:
false}).then(($window) => { return
$window.store.getState() }).then((state) => {
        if (stateName.length > 0) {
            return cy.wrap(state, {log:
false}).its(stateName).then(cb)
        } else {
            return cy.wrap(state, {log:
false}).then(cb)
        }
    })
})
```

Now, we can go back to our test in `todos.spec.js` and pass in `todos` to the store and remove the `its`.

todos.spec.js

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.server()
        cy.route('/api/todos', [
          {
            "id": 1,
            "text": "Hello world",
            "completed": false
          },
          {
            "id": 2,
            "text": "Goodnight moon",
            "completed": true
          }
        ]).as('preload')

        cy.visit('/')
        cy.wait('@preload')

        cy.store('todos').should('deep.equal',
[{
             id: 1,
            text: 'Hello world',
            completed: false
          }, {
            id: 2,
            text: 'Goodnight moon',
            completed: true
          }])
...
```

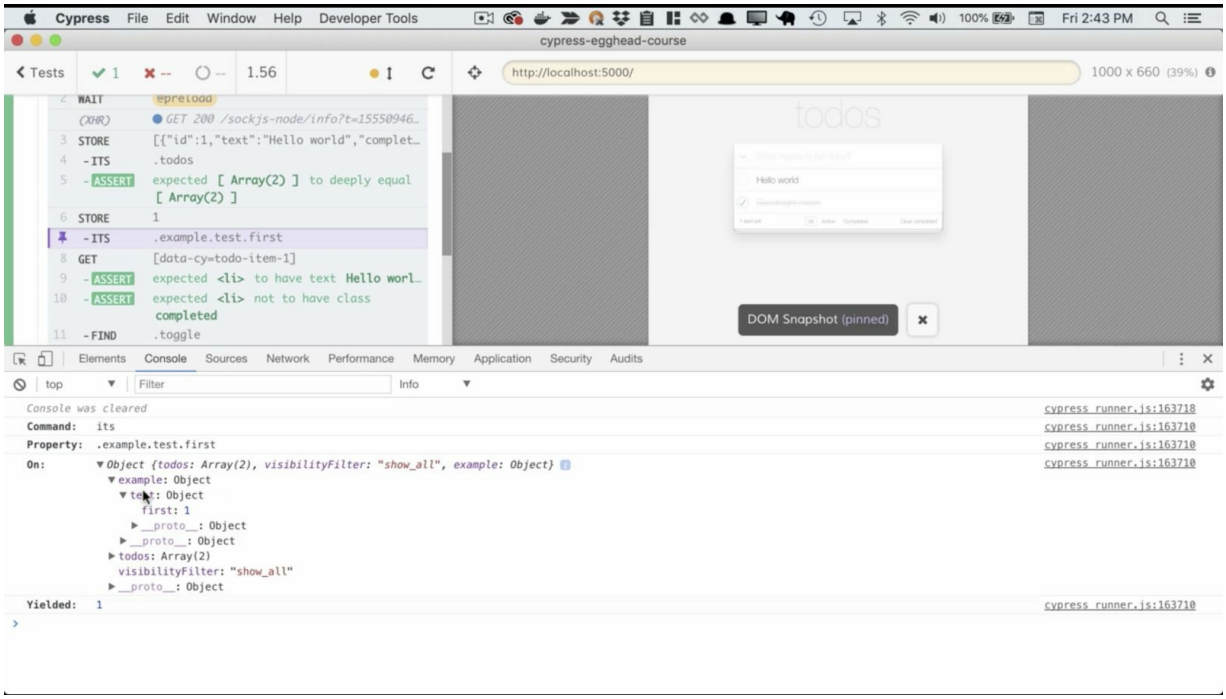
When we rerun our test, we'll see that in the store, the only part of the store that we logged out was the part of the store that we were interested in.



Because cy.its accepts dot-separated notation, we can drill into the store as deeply as we want. Remember this deeply nested piece of state, `example.test.first`? We can drill into that, too.

```
describe('Todo Application', () => {
    it('loads the page', () => {
        cy.server()
        cy.route('/api/todos', [
          {
            "id": 1,
            "text": "Hello world",
            "completed": false
          },
          {
            "id": 2,
            "text": "Goodnight moon",
            "completed": true
          }
        ]).as('preload')

        cy.visit('/')
        cy.wait('@preload')

        cy.store('todos').should('deep.equal',
[{
             id: 1,
            text: 'Hello world',
            completed: false
          }, {
            id: 2,
            text: 'Goodnight moon',
            completed: true
          }])

        cy.store('example.test.first')
...
```

When we load it up, we can see this. We can see that it yielded the value one.



We can see the piece of state that we drilled into. You can see pretty quickly how effective it is to create custom commands with custom logging.

Wrap External Libraries with Cypress

In this lesson, we're going to learn about wrapping external libraries in Cypress. Why might we want to do this? We can already see this is a more complex example. There's a lot more data.

For instance, we've run an assertion on our store, which has five todos in it. Maybe we only want to make an assertion against one of those.

When we look at our test, we can see that we've clearly had to assert too much. We've had to list out every single todo and all of its properties even though perhaps the only one we want to make an assertion on is this first one.

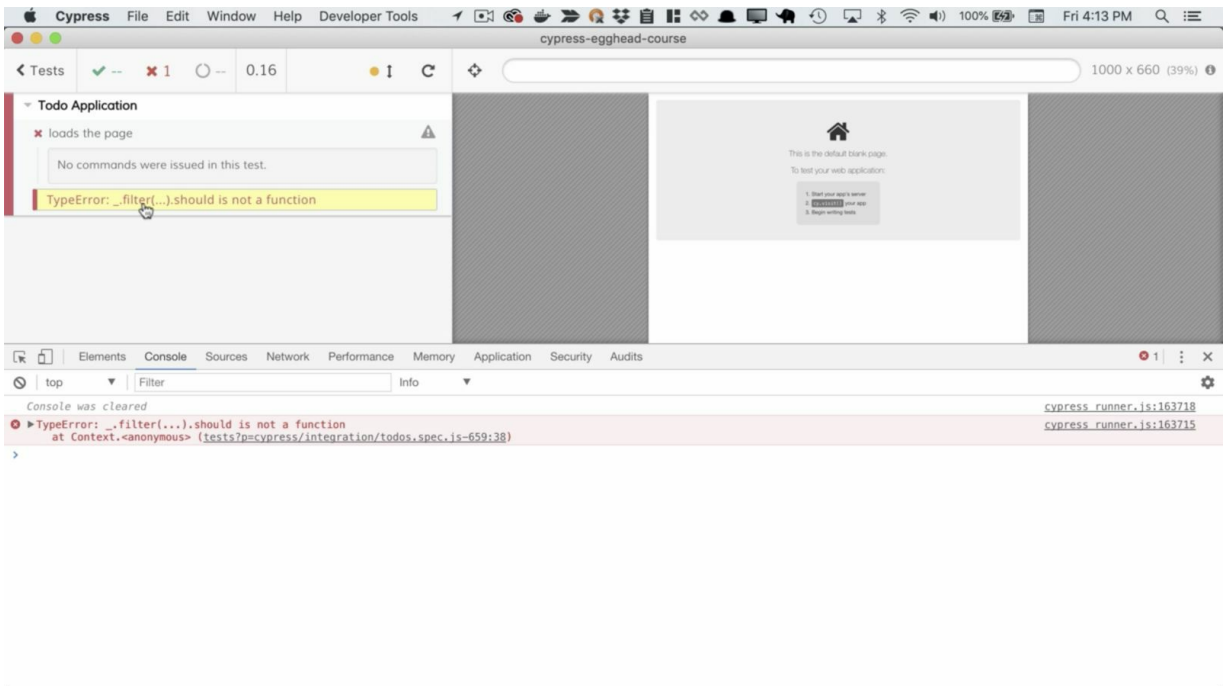
Of course, there are typical libraries we might try to use for this type of problem, like lodash. Let's see if we can use lodash `.filter`, passing in the todos from the store. We'll inspect each todo. If it has the ID of one, then we'll grab it. We hope this is just going to give us this first todo. Let's make sure we import `lodash`.

todos.spec.js

```
const _ = require('lodash')

describe('Todo Application', () => {
  it('loads the page', function () {
    cy.server()
    // Alias the fixture data
    let todos = [
      {
        id: 1,
        text: '1st Todo',
        completed: false
      },
      {
        id: 2,
        text: '2nd Todo',
        completed: true
      },
      {
        id: 3,
        text: '3rd Todo',
        completed: false
      },
      {
        id: 4,
        text: '4th Todo',
        completed: true
      },
```

```
        {
          id: 5,
          text: '5th Todo',
          completed: false
        },
      ]
      ...

      // Access the fixture data this.todos
      _.fliter(cy.store('todos'), (todo) => {
  todo.id == 1}).should('deep.equal', [
          {
              id: 1,
              text: '1st Todo',
              completed: false
          },
          ...
      ])
```

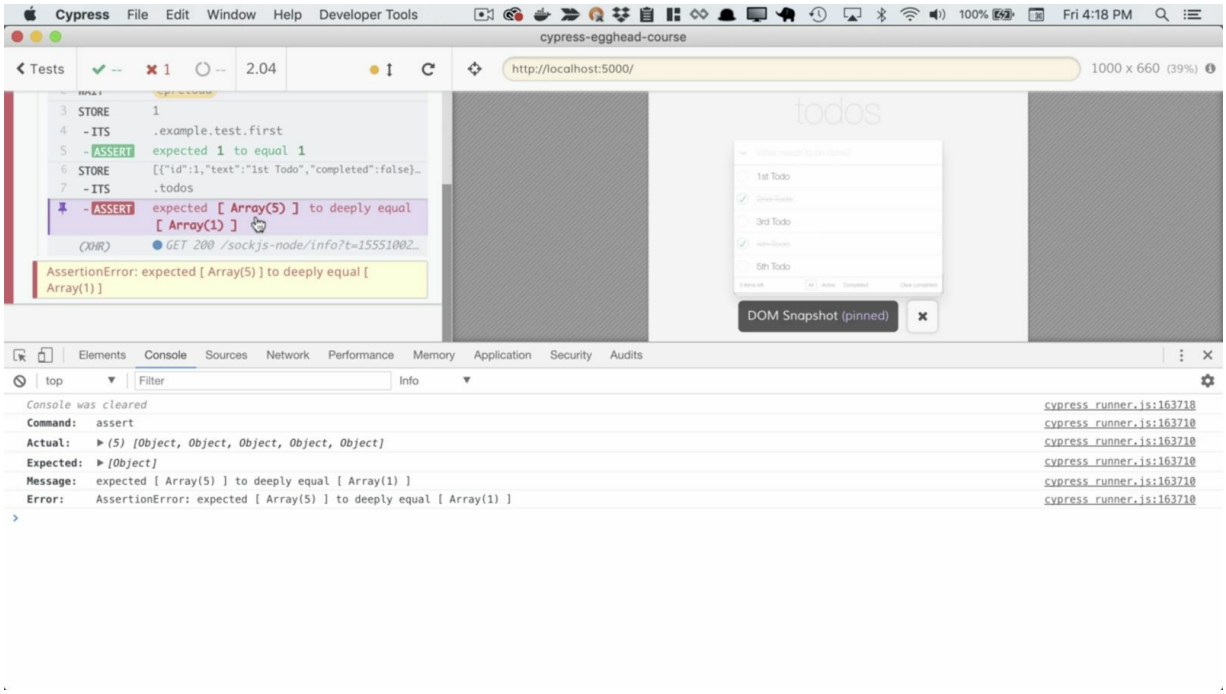
Of course, we see that `lodash.filter.should` is not a function because we haven't integrated lodash into Cypress.



Remember that Cypress is asynchronous. It produces its own command chain, so we'll pull off the filter from here and tag on a `.then`. Then we'll be past all the todos from the store. Then we can `.filter` them, passing in these `todos` and filtering out each individual todo in here. `return todo.id == 1`. Finally, we'll run our `.should`.

```
...
// Access the fixture data this.todos
cy.store('todos').then((todos) => {
    _.filter(todos, (todo) => {
        return todo.id == 1
    }).should('deep.equal', [
    {
        id: 1,
        text: '1st Todo',
        completed: false
    },
    ...
])
```

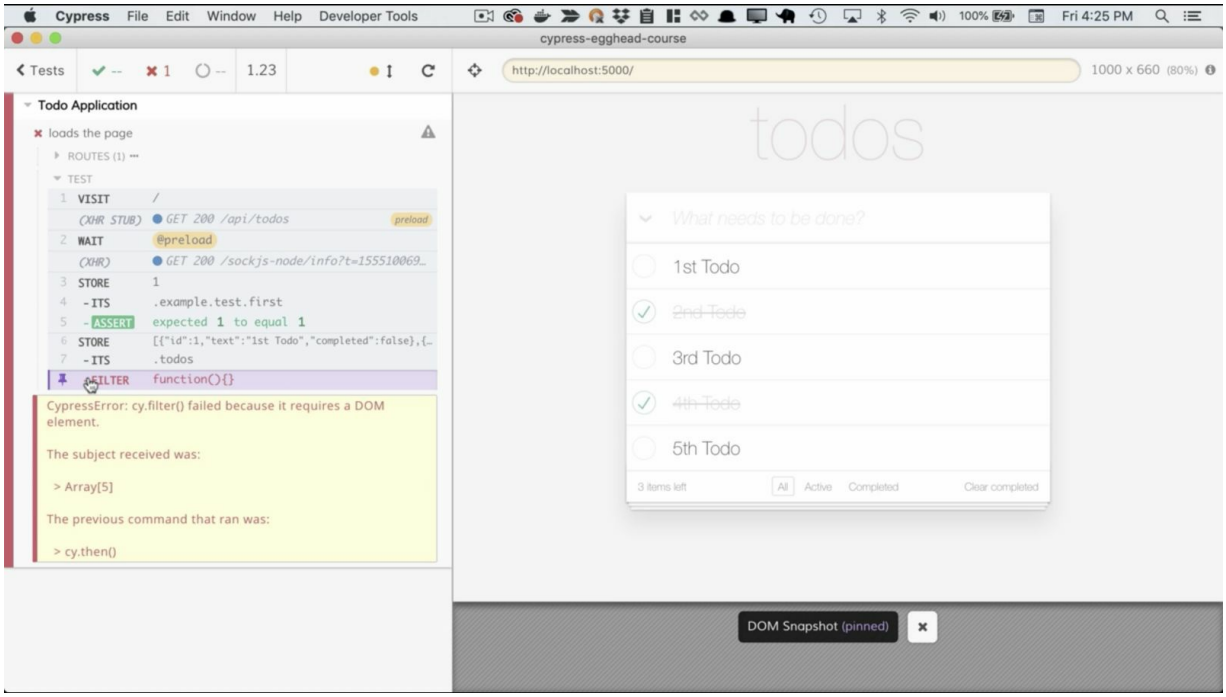
We'll see that we filtered out only one todo. This is a ton of overhead just to filter our todos.



Shouldn't we be able to incorporate filter into our Cypress command chain?

```
...
// Access the fixture data this.todos
cy.store('todos').filter((todo) => {
    return todo.id == 1}
).should('deep.equal', [
    {
        id: 1,
        text: '1st Todo',
        completed: false
    },
    ...
])
```

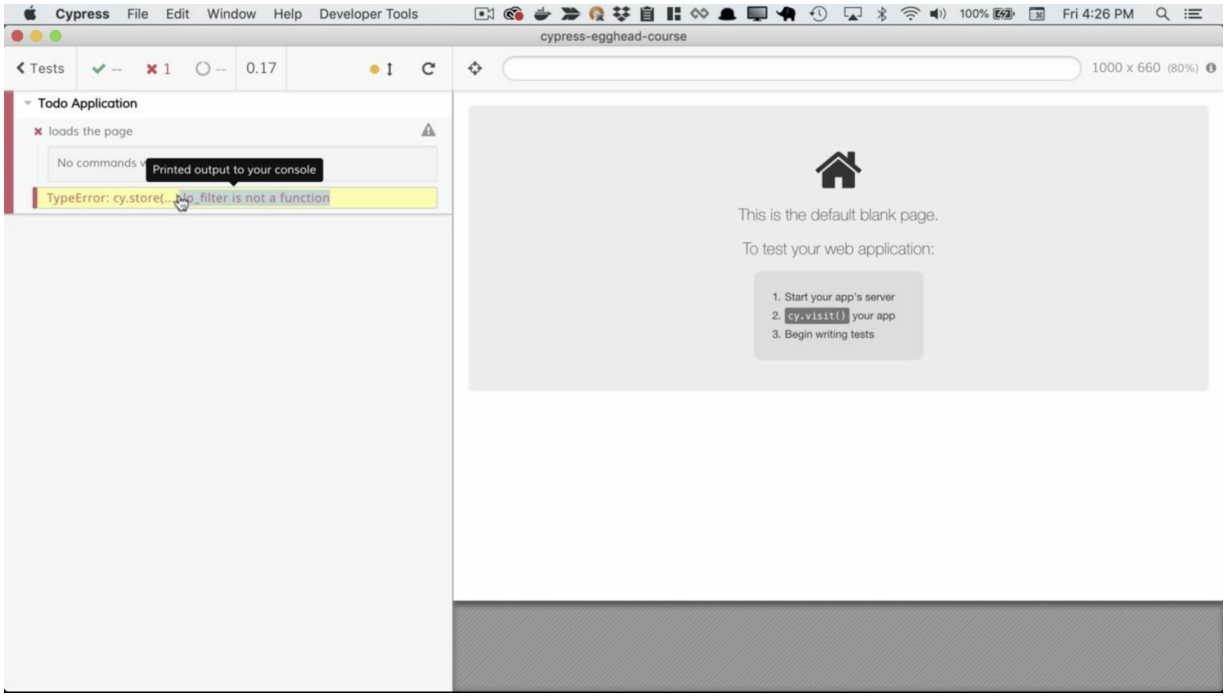
The first thing we'll notice if we try this is that filter is already an existing Cypress command.



Let's go ahead and rename this `lo_Filter`.

```
...
// Access the fixture data this.todos
cy.store('todos').lo_filter((todo) => { return
todo.id == 1}).should('deep.equal', [
    {
        id: 1,
        text: '1st Todo',
        completed: false
    },
    ...
])
```

Heading back into Cypress, we see that `lodashFilter` is not a function, which is what we want.

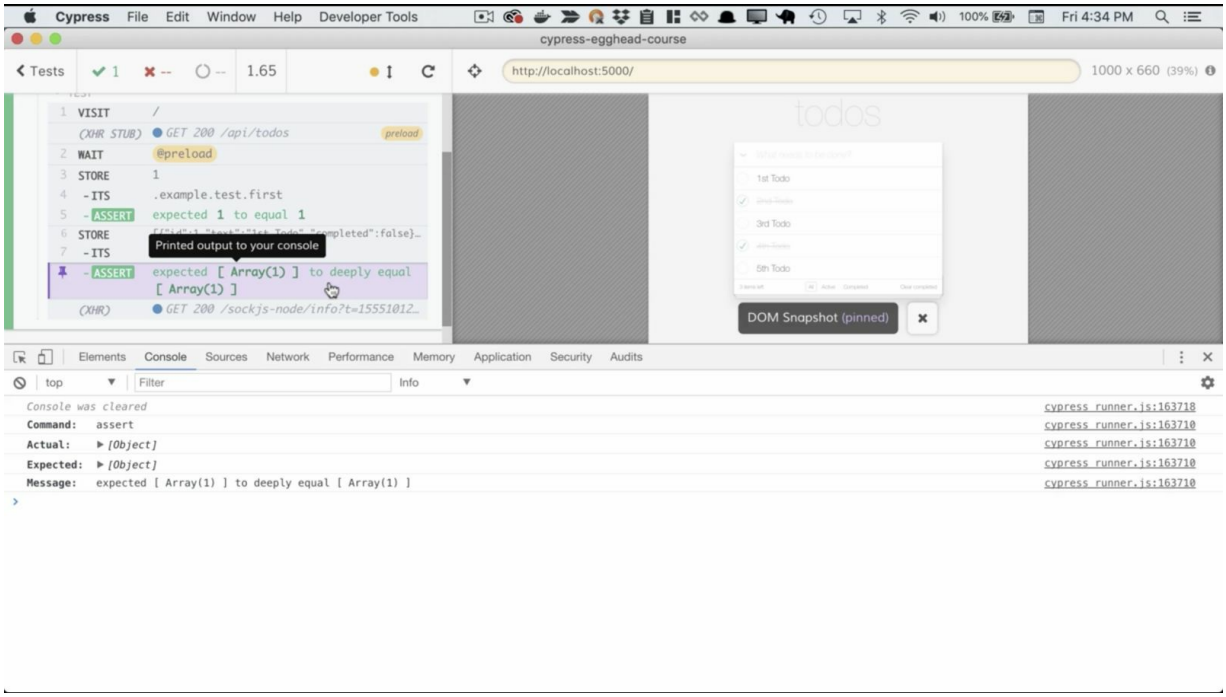


We're ready to add one. Let's hop back into our custom commands file and require lodash. We're ready to add a new command, `Cypress.Commands.add('lo_filter')`. This time, we are adding a child command, which means that we have a `{prevSubject: true}`. Since we have a previous subject, the first argument will be `subject`. The second will be the function that we use to filter. Next, we'll `return _.filter`, passing in the subject and the function to filter with.

commands.js

```
const _ = require('lodash')

Cypress.Commands.add("store", (str = '') => {
    let log = Cypress.log({ name: 'store' })

    const cb = (state) => {
        log.set({
            message: JSON.stringify(state),
            consoleProps: () => {
                return state
            }
        }).snapshot().end()

        return state
    }

    return cy.window({log:
false}).then(function($w) { return
$w.store.getState() }).then((state) => {
        if (str.length > 0) {
            return cy.wrap(state, {log:
false}).its(str).then(cb)
        } else {
            return cy.wrap(state, {log:
false}).then(cb)
        }

    })
})

Cypress.Commands.add('lo_filter', {prevSubject:
true}, (subject, fn) => {
    return _.filter(subject, fn)

})
```

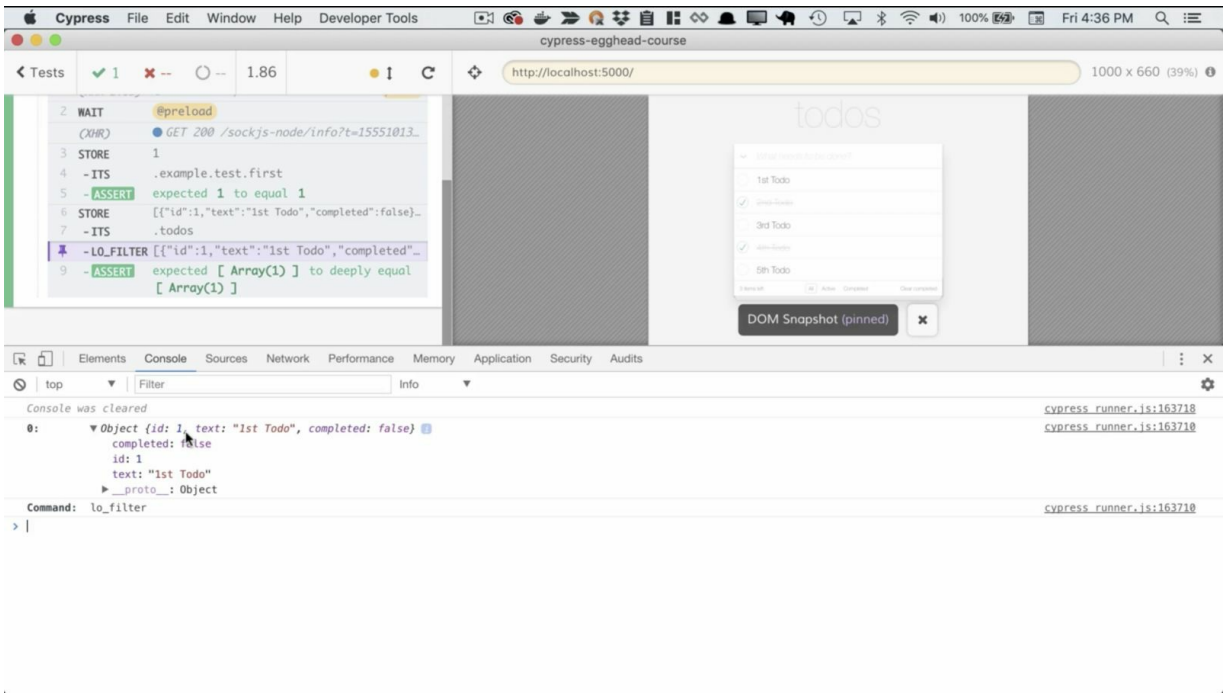
If we reopen Cypress, we'll see that we filtered out our subject.



Remember we're going to want to log this. Let's go back and make this the `result`. We'll run `Cypress.log`, passing in the name `lo_Filter`. The message will be `JSON.stringify` the `result`. We'll probably want `consoleProps`. We'll `return result`. Finally, we'll `return result` from all of this.

```
...
Cypress.Commands.add('lo_filter', {prevSubject:
true}, (subject, fn) => {
    return _.filter(subject, fn)
    Cypress.log({
        name: 'lo_filter',
        message: JSON.stringify(result),
        consoleProps: () => { return result }
    })

    return results
})
```

Now, in Cypress, we can see the filter. We can go in the console to see what it looks like.



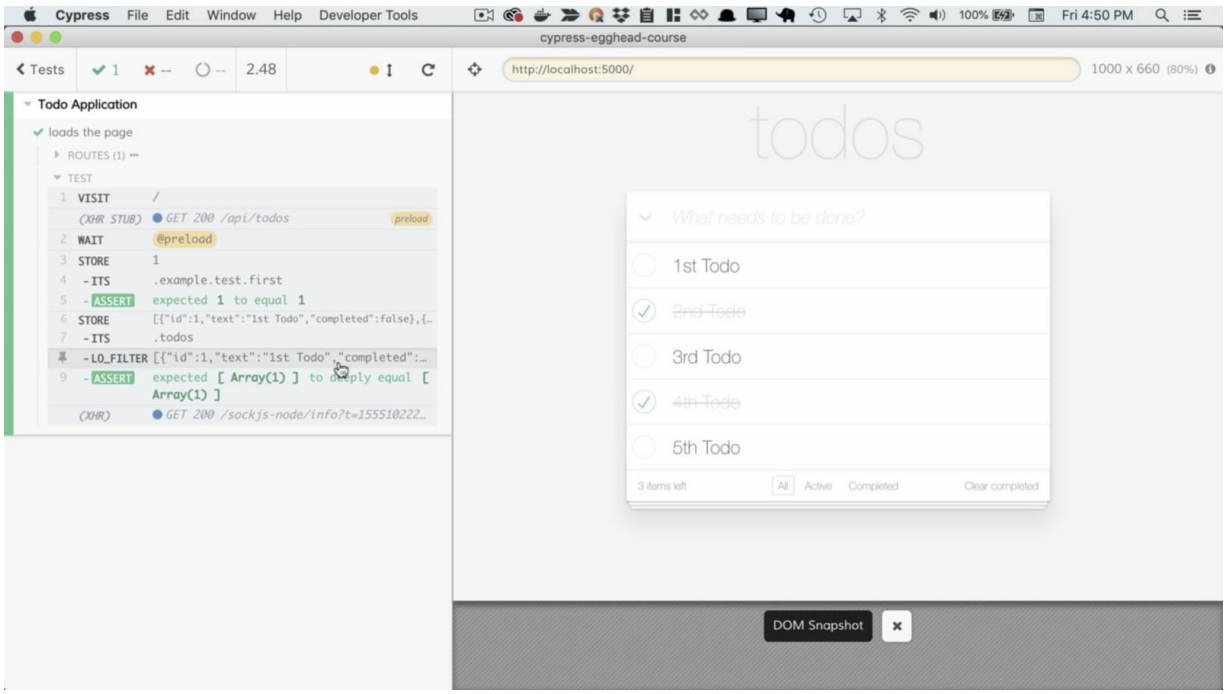
What if we want to incorporate every lodash method into Cypress? What if we want to wrap the whole library?

Let's start by defining each of the lodash methods. We'll grab each of the function names from lodash. Then we'll `map` them because we know there's sometimes conflicts. We'll preface them each with `lo` and then the name of the real function. Then for each of those, we will define a Cypress command. We'll copy out the Cypress command. We'll get the real name of the lodash method by removing the lo preface because we're going to need to call the real method in lodash. There we go.

Next we will define the name of the command. Some lodash methods have args. We'll just pass these through. Instead of filter, we'll call the lodash method that is defined. We'll change the `name` so it updates in the log.

```
...
let loMethods = _.functions(_).map((fn) => {
return 'lo_${fn}'})

loMethods.forEach((loFn) => {
    let loName = loFn.replace(/lo_/, '')
Cypress.Commands.add(loFn, {prevSubject: true},
(subject, fn, ...args) => {
    let result = _[loName](subject, fn, ...args)
    Cypress.log({
        name: 'lo_filter',
        message: JSON.stringify(result),
        consoleProps: () => { return result }
    })

    return result
})
```

Let's check this out in Cypress. Make sure everything still works the same way, and it does.



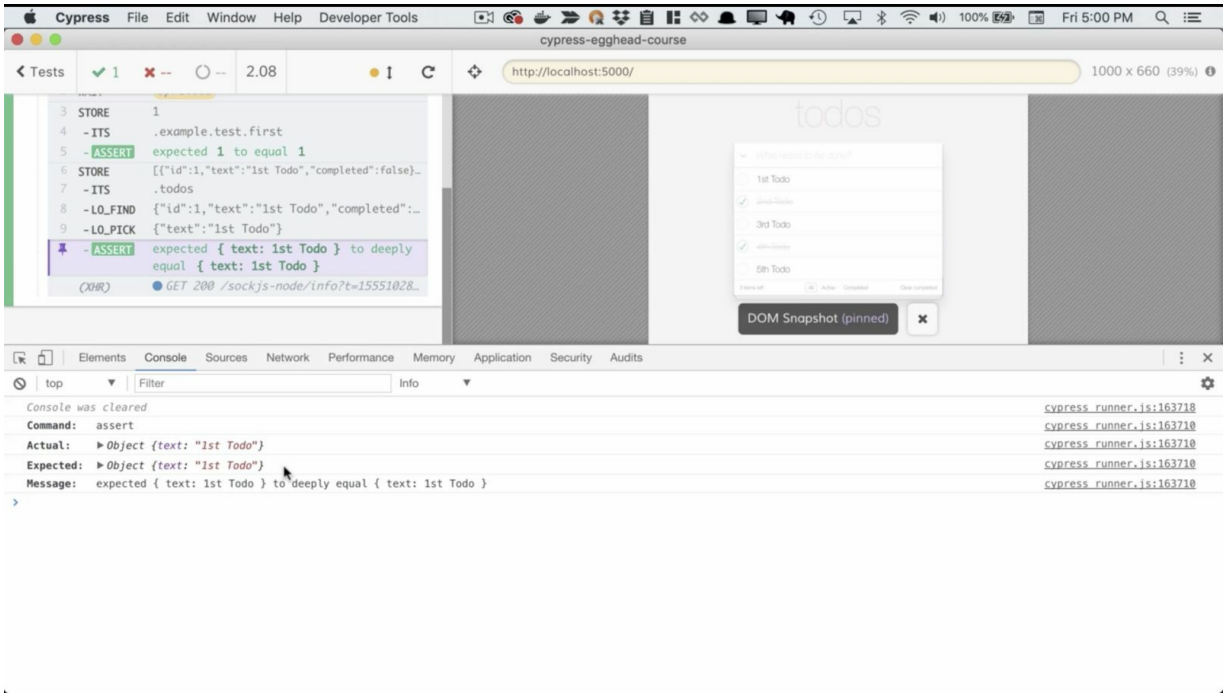
Let's go ahead and show the power of using our lodash methods together. We'll use `.lo_find` to pick off the first todo.

Then we'll use `.lo_pick` to pick off just the text. This is no longer an array. It no longer has any other attributes. We're just asserting on the parts that we care about.

todos.spec.js

```
...
    // Access the fixture data this.todos
    cy.store('todos')
    .lo_find((todo) => { return todo.id == 1})
    .lo_pick('text')
    .should('deep.equal', [
        {
            text: '1st Todo',
        },
        ...
    ])
```

We'll rerun this and see that find found us the entire todo. Pick picked off just the text. Then our assertion runs only the assertion we care about. You can see that by wrapping this external library, we've given Cypress superpowers.



Reuse Data with Cypress Fixtures

Thus far, whenever we've stubbed out a network request, we've written our todos in the body of the test and then passed them into `cy.route`. What if we created another test, as we have in this example, and we want to share the same starting state between the two examples?

Let's go ahead and move our data into a fixture file. Let's go ahead and create a new folder under `cypress/fixtures/todos/all.json`, and go ahead and paste in your fixture.

all.json

```
[
    {
        "id": 1,
        "text": "Hello world",
        "completed": false
    },
    {
        "id": 2,
        "text": "Goodnight moon",
        "completed": true
    }
]
```

Make sure to remove the variable assignment since we just need to return the raw data.

Now we can reference our fixture by returning to our test and using `cy.fixture`, and then the path that's relative to the fixtures folder, `todos/all.json`. Then we can use `then`, which will asynchronously load our JSON. Since the JSON is going to be loaded asynchronously, we can take the rest of our test and place it inside the callback. Anywhere we had previously used our variable todos, we should update it to use the `cy.fixture` data.

todos.spec.js

```
...
it('creates new todos', function () {
    cy.server()

    cy.fixture('todos/all.json').then((json) =>
{
        cy.route('/api/todos',
json).as('preload')

        cy.visit('/')
        cy.wait('@preload')
        ...
    })
})
```

We can also share data between our tests without using callbacks at all. It starts with a `beforeEach` block, which we will run before each of our tests. Next, call `cy.fixture`, passing in the same path as before. Then we'll add an alias to it, so it can be referenced from other locations in our file. Let's call it `todosPreload`.

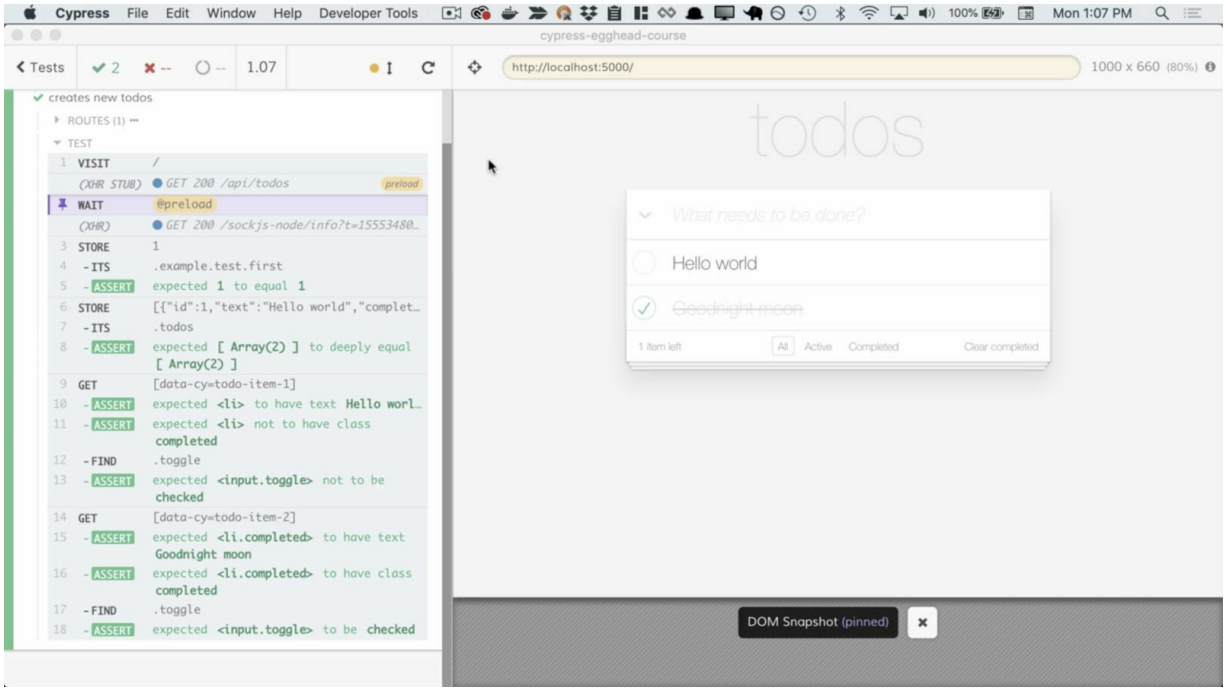
```
describe('Todo Application', () => {
    beforeEach(function() {

cy.fixture('todos/all.json').as('todosPreload')
    })
    it('loads the page', function () { ...

    })
    it('creates new todos', function () {
        cy.server()
    })
})
```

Now we can go ahead and remove our fixture, and inside `cy.route`, we're capable of referencing todos-preload as `@todosPreload`, because `cy.route` is the special method that actually recognizes that.

```
describe('Todo Application', () => {
    beforeEach(function() {

cy.fixture('todos/all.json').as('todosPreload')
    })
    it('loads the page', function () { ...

    })
    it('creates new todos', function () {
        cy.server()

        cy.route('/api/todos',
'@todosPreload').as('preload')

        cy.visit('/')
        ...
    })
})
```

Otherwise, we have to use `this.todosPreload`, and we get todosPreload attached to this if we use `cy.fixture` with an alias inside a `beforeEach` function, but not if we just do it in the body of our test.

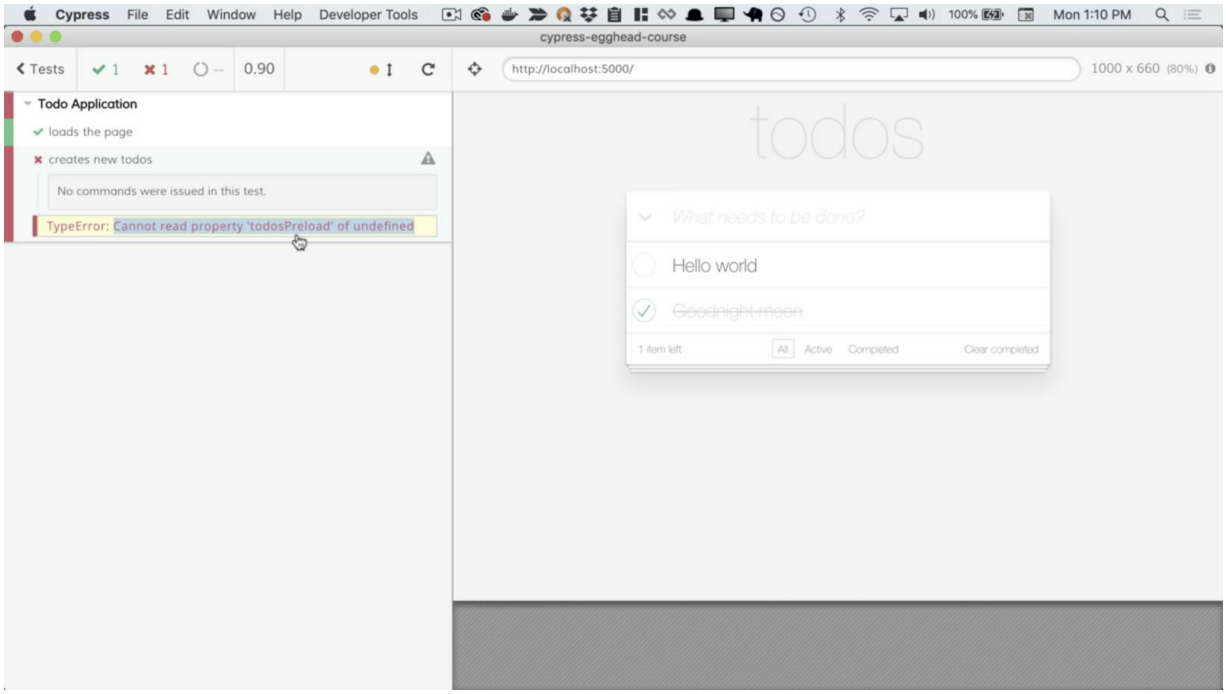
We can go ahead and save this out. Let's open up Cypress. We'll see that Cypress doesn't actually log out any information about our fixture file, but it had no problem with using it.



One gotcha that I do want to make you aware of is that if you're going to reference any of your aliases using this, for instance `this.todosPreload`, don't use a fat arrow function for your test, because fat arrow functions don't have a `this` context. This may be assigned in your `beforeEach`, but it's not going to be available here.

```
describe('Todo Application', () => {
    beforeEach(function() {

cy.fixture('todos/all.json').as('todosPreload')
    })
    it('loads the page', function () { ...

    })
    it('creates new todos', () => {
        cy.server()

        cy.route('/api/todos',
'@todosPreload').as('preload')

        cy.visit('/')
        ...
    })
})
```

If we reopen our test, we'll see that we can't read the property todosPreload of undefined. That's because we're not using a standard function. That's because of the fat arrow function.



That's how you can use fixtures in your tests.

Mock Network Retries with Cypress

Networks fail for all kinds of reasons, and it's common for our frontend code to retry in the face of network errors. Thanks to Cypress, we can test this behavior easily.

Let's start by creating a new context for `Todo creation retries`. In here, we'll make two tests. The first we'll say is `it('retries 3 times', function()`. This will be the success case, where on the third time the retry is successful. Then, we'll say `it('fails after 3 unsuccessful attempts', function()`, and we'll go ahead and attach `.only` to our `context`, so this will be the only series of tests that we'll run.

todos.spec.js

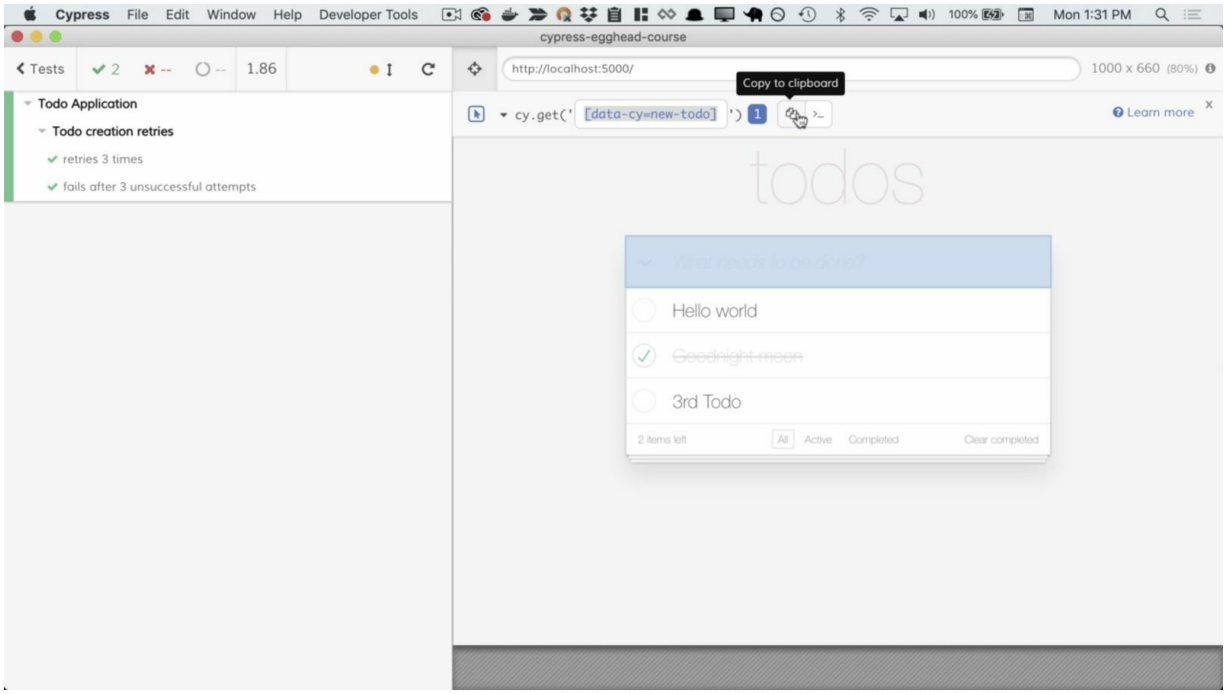
```
describe('Todo Application', () => {
  beforeEach(function() {
    cy.fixture('todos/all.json').as('todos')
  })

  it('loads the page', function () {...
  })

  context.only('Todo creation retries',
function() {
      it('retries 3 times', function() {

      })

      it('fails after 3 unsuccessful attempts',
function () {

      })
    })
  })
```

Let's go ahead and start off each test with a `beforeEach` context, which loads up the page in a known state. In this case we have the two todos that we've typically been using. Now we're ready to get started writing our test.

```
describe('Todo Application', () => {
  beforeEach(function() {
    cy.fixture('todos/all.json').as('todos')
  })

  it('loads the page', function () {...
  })

context.only('Todo creation retries', function()
{
    beforeEach(function() {
        cy.server()
        // Alias the fixture data
        cy.route('/api/todos',
'@todos').as('preload')

        cy.visit('/')
        cy.wait('@preload')
    })
```

Let's head into Cypress and see what the interaction looks like to create a new todo. We'll type "third todo," and press enter, and we see that shows up on the page, so let's replicate that.

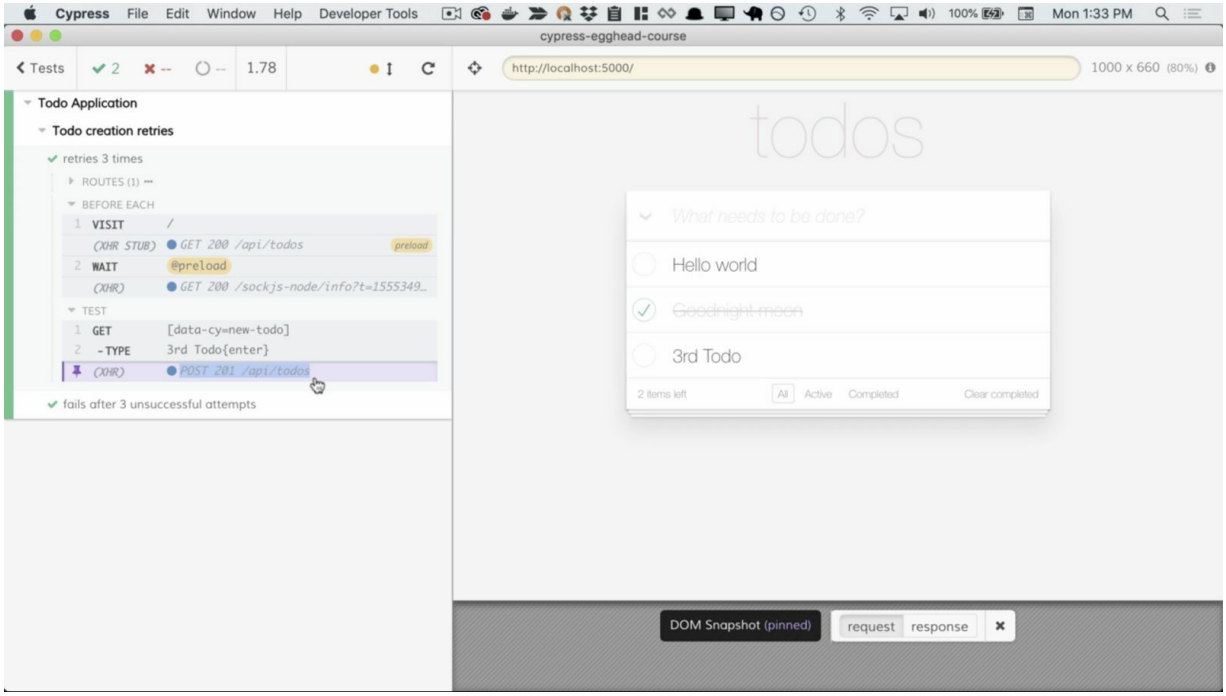
We'll click on our Selector Playground, and we'll see that this is called "data-cy new todo," so we'll go ahead and copy that.



We'll paste it in and say that we type `'3rd Todo'` and then we'll use the special key `enter`, and that will interact with the page for us.

```
context.only('Todo creation retries', function()
{

    ...

      it('retries 3 times', function() {
          cy,get('[data-cy+new-todo]').type('3rd
Todo{enter}')

      })

      it('fails after 3 unsuccessful attempts',
function () {

      })
    })
  })
```

If we head back in the Cypress, we can see what the POST request is that's created here.



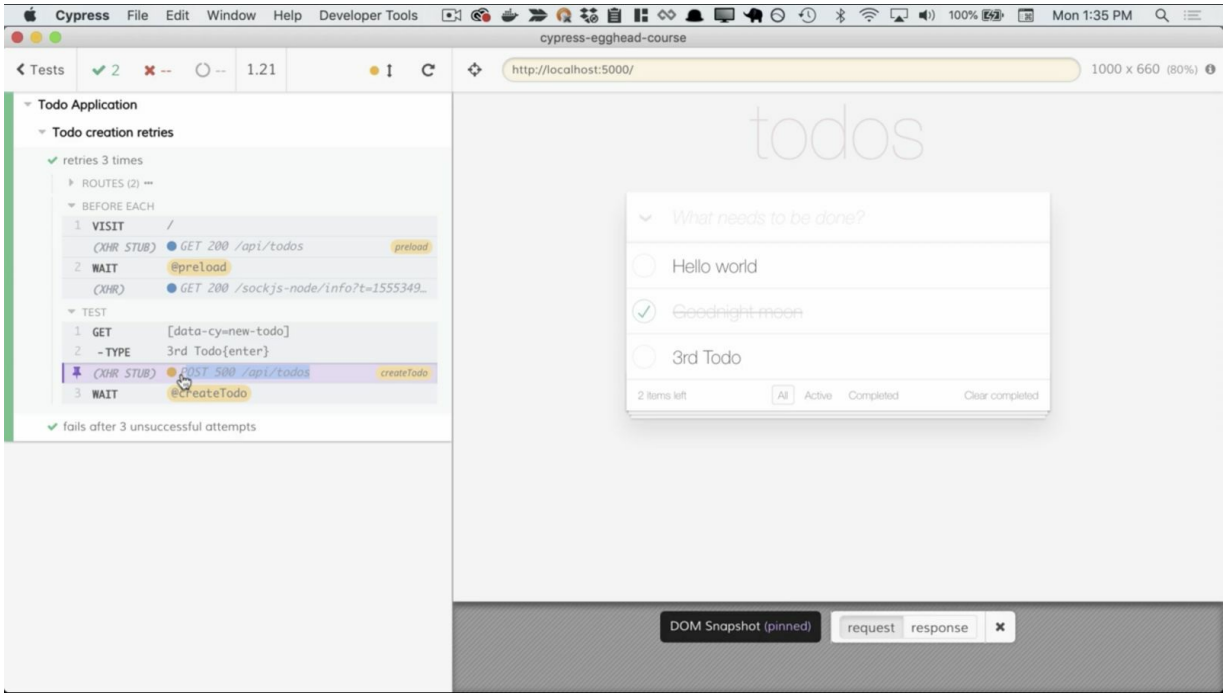
We post `api/todo`, so let's go ahead and mock that out. That's going to be `cy.route`. The `method` is going to be `'POST`. The `url` is going to be `/api/todos`. The first response is going to be a `500` status code, and we'll just reply with an empty `response`. Let's say this is `createTodo`, so we give it an alias.

```
context.only('Todo creation retries', function()
{

    ...

      it('retries 3 times', function() {
          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo')

          cy,get('[data-cy+new-todo]').type('3rd
Todo{enter}')

      })
```

Then, we'll scroll down here and say `cy.wait` for `('@createTodo')`.

```
context.only('Todo creation retries', function()
{

    ...

      it('retries 3 times', function() {
          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo')

          cy,get('[data-cy+new-todo]').type('3rd
Todo{enter}')

          cy.wait('@createTodo')
      })
```

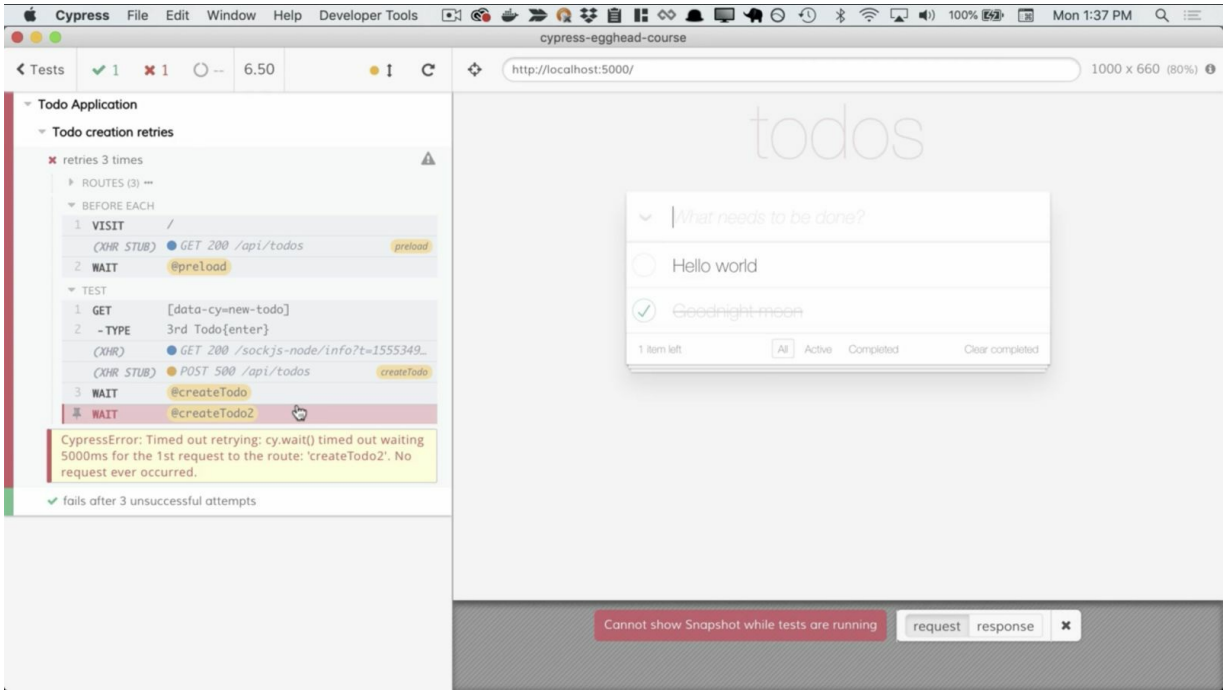
Let's head back to Cypress. We'll see that we successfully waited on createTodo, and instead of hitting our real backend, which sent us a 201, now we're sending back a 500, which means that our frontend should start retrying.



Let's go ahead and say that we're going to `cy.wait` for this again, so we'll call this `.as('createTodo2')`, this will be the second version of the request. We can alternatively just wait on `createTodo` a second time, but this makes it a little more explicit what we intend.

```
context.only('Todo creation retries', function()
{
    beforeEach(function() {
        cy.server()
        // Alias the fixture data
        cy.route('/api/todos',
'@todos').as('preload')

        cy.visit('/')
        cy.wait('@preload')

        cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo')

        cy,get('[data-cy+new-todo]').type('3rd
Todo{enter}')

        cy.wait('@createTodo')

        cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo2')

        cy.wait('@createTodo2')
    })
```

We'll come back to our test and see that createTodo two hangs indefinitely, and that's because our frontend doesn't yet do any retrying.



Let's head back to our code and make sure we do some retries.

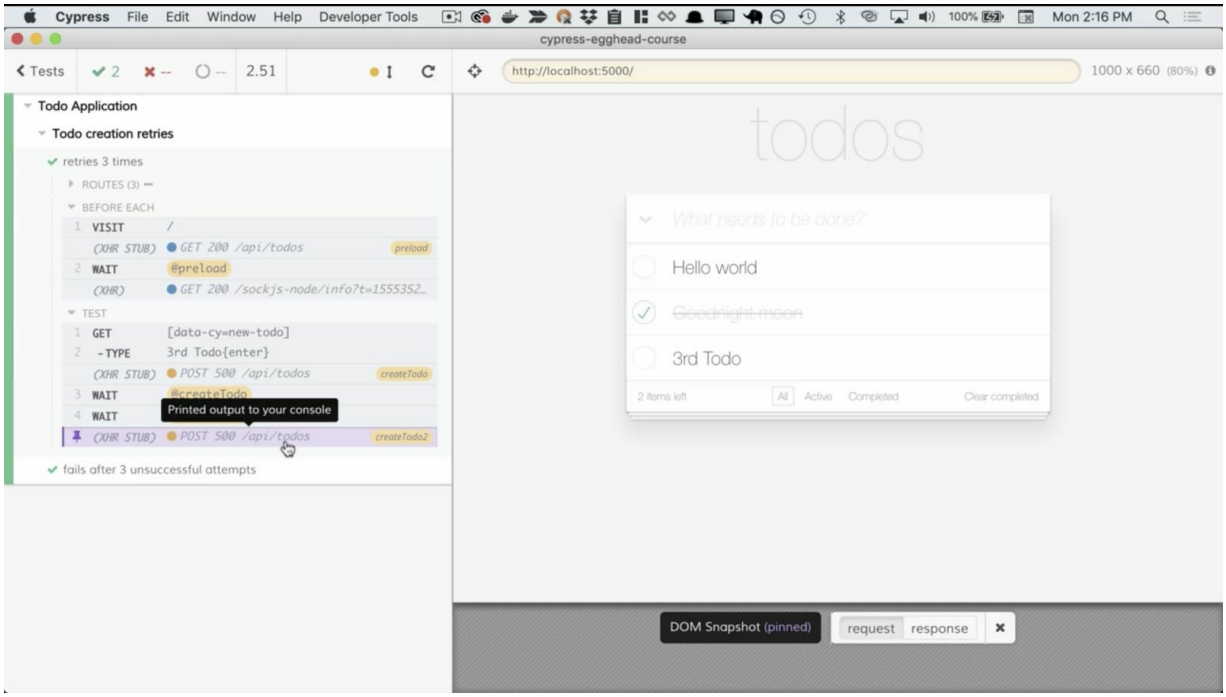
This will be in `TodoSagas.js`. We'll import the `retry` from `"redex-saga/effects"`. The `createTodo` function is what actually creates the todo-now, so let's call this `createTodoAttempt`, and we'll make sure this retries three times.

Let's go ahead and use a try, because after three unsuccessful attempts, this will raise an error, and we'll `yield` using the retry-saga. We'll `retry` three times for one second each, and we'll retry using the `createTodoAttempt` method, and then we'll pass it our action. If we're successful, we will yield the action `'ADD_TODO_SUCCESS'`, and after three unsuccessful retries, we'll catch that error, and instead we'll yield a new action called `'ADD_TODO_FAIL'`.

TodoSagas.js

```
import { takeEvery, takeLatest, put, all, select
} from "redux-saga/effects";
import axios from "axios";

function getBaseUrl() {
  return 'http://localhost:3000'
}

function* createTodo(action) {
    try {
      yield retry(3, 1000, createTodoAttempt,
action)
      yield put({type: 'ADD_TODO_SUCCESS'})
    } catch(e) {
      yield put({...action, type:
'ADD_TODO_FAIL'})
    }
}

function* createTodoAttempt(action) {
  yield axios.post(`${getBaseUrl()}/api/todos`,
{text: action.text, completed: false})
}
```

If we return to Cypress, we'll see that we've now waited for both our first and second createTodo attempts and replied in both cases with a 500.



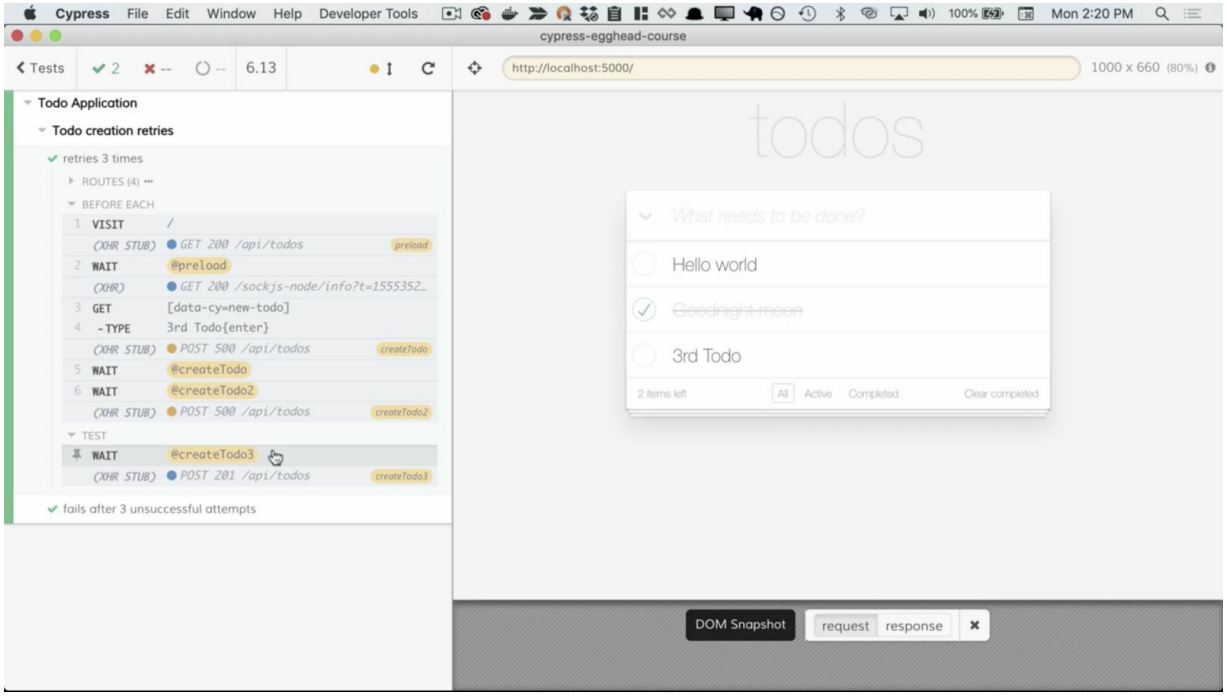
Let's go ahead and finish our test. We'll move out the first two `createTodo` attempts to the `beforeEach` hook, because in both cases we're going to be using that, and then in the third `createTodo` attempt, in one case it'll be successful, and in the second case it will be unsuccessful. We can leave this 500 for the third createTodo for the unsuccessful attempt, and in the successful attempt, we'll just go ahead and change our status code to 201.

todos.spec.js

```
it('retries 3 times', function() {
          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo')

          cy,get('[data-cy+new-todo]').type('3rd
Todo{enter}')
```

```
          cy.wait('@createTodo')

          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo2')

           cy.wait('@createTodo2')
      })

       it('retries 3 times', function() {
          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 201,
            response: ''
          }).as('createTodo3')

          cy.wait('@createTodo3')
       })

        it('fails after 3 unsuccesful attempts',
function() {
          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo3')

          cy.wait('@createTodo3')
```

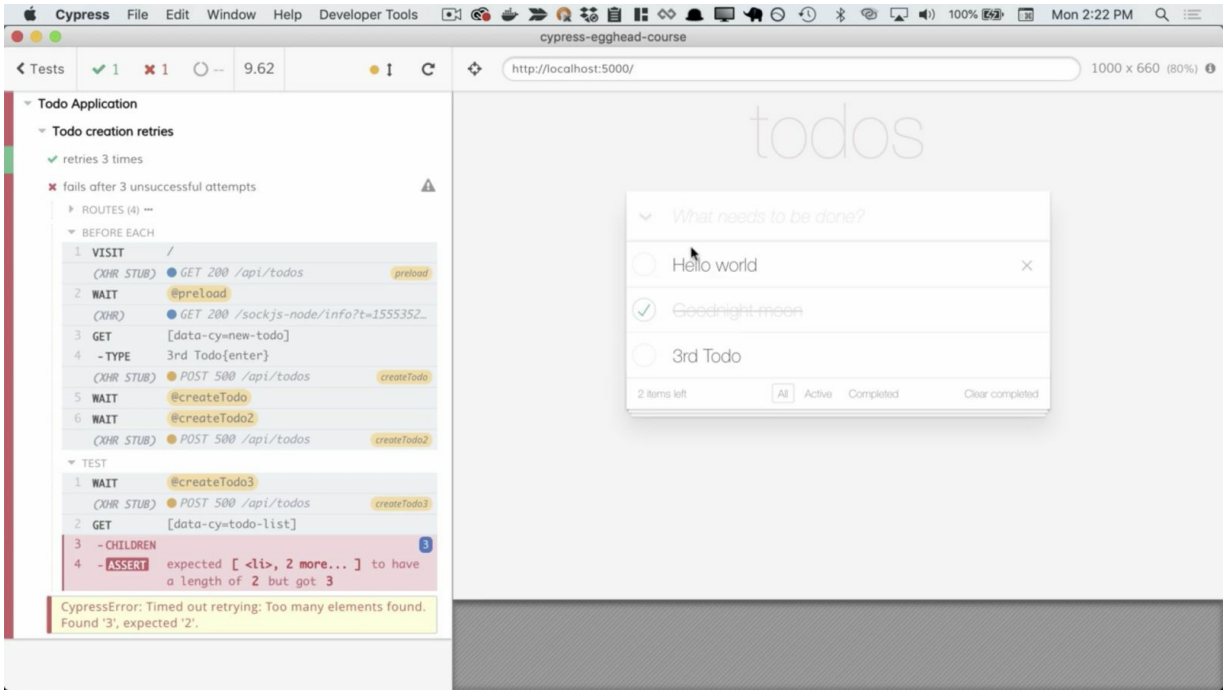
We'll see that Cypress does in fact wait for each subsequent createTodo attempt, but we still haven't made any assertions.



Let's go ahead and say that in the successful case, we will get the `todo-list`, and the todo-list `.children`, `.should` have length `3`. In the unsuccessful case, let's just assume that it gets removed from the DOM, so it'll have length `2`.

```
...
        it('retries 3 times', function() {
          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 201,
            response: ''
          }).as('createTodo3')

          cy.wait('@createTodo3')

          cy,get('[data-cy=todo-list]')
            .children()
            .should('have.length', 3)
      })

        it('fails after 3 unsuccesful attempts',
function() {
          cy.route({
            method: 'POST',
            url: '/api/todos',
            status: 500,
            response: ''
          }).as('createTodo3')

          cy.wait('@createTodo3')

          cy,get('[data-cy=todo-list]')
            .children()
            .should('have.length', 2)
      })
```

Of course, this fails in our Cypress test, because we still have all three children on the DOM.



The reason for that is because our reducer doesn't yet respond to the AddToDoFail method, which we added in our saga after three unsuccessful attempts. Let's go ahead and just finish this interaction.

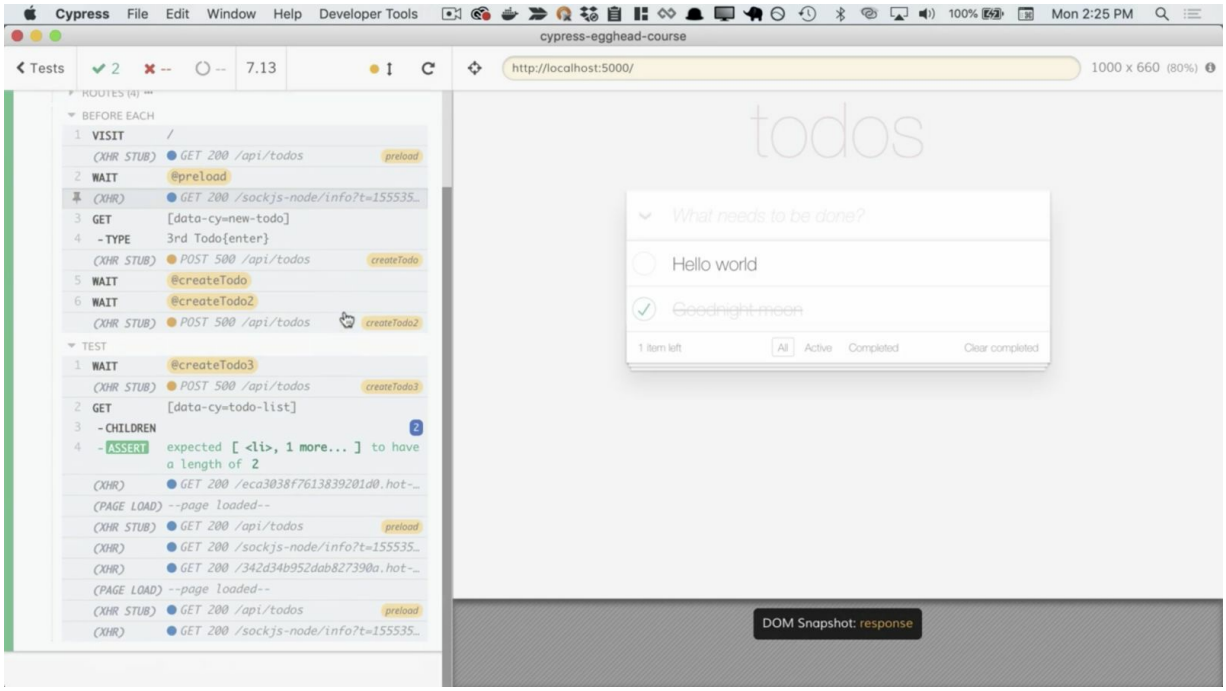
todos.js

```
import {
  TODOS_LOADED,
  ADD_TODO,
  DELETE_TODO,
  EDIT_TODO,
  LOCAL_CLEAR_COMPLETED,
  BULK_EDIT_TODOS,
  ADD_TODO_FAIL,
} from '../constants/ActionTypes'

...

    case ADD_TODO_FAIL:
      return state.filter(todo => todo.text !==
action.text)
```

We'll uncomment this so it filters out the Todo.

```
case ADD_TODO_FAIL:
  return state.filter(todo => todo.text !==
action.text)
```

We will go back to our test and see that it passed. With Cypress, network retries are easy to test, because for each XHR request, we can stub out a different response and then wait for them in order.



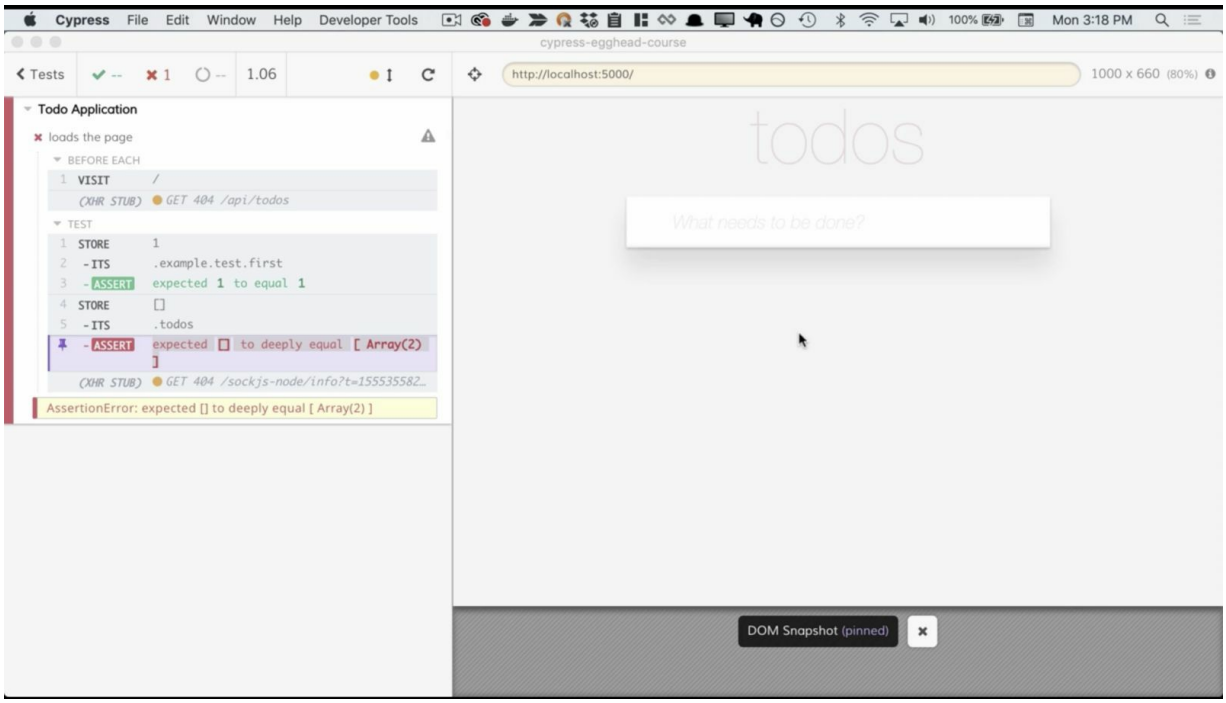
Find Unstubbed Cypress Requests with Force 404

By default, `cy.server` will pass all requests to your back end. Now, that's a great configuration when we want to do full end-to-end testing, but often times, we'll want to mock our back end to ensure that we get known data for our tests. How do we make sure we stub out every request our application makes?

If we pass the option `cy.server({force404: true})`, Cypress will ensure that any route that isn't stubbed returns a 404, breaking our app. Let's go ahead and pass it. We'll comment out our `preload` call.

```
describe('Todo Application', () => {
  beforeEach(function() {
    cy.fixture('todos/all.json').as('todos')

    cy.server({force404: true})
    // Alias the fixture data
    // cy.route('/api/todos',
'@todos').as('preload')

    cy.visit('/')
    // cy.wait('@preload')
  })
```

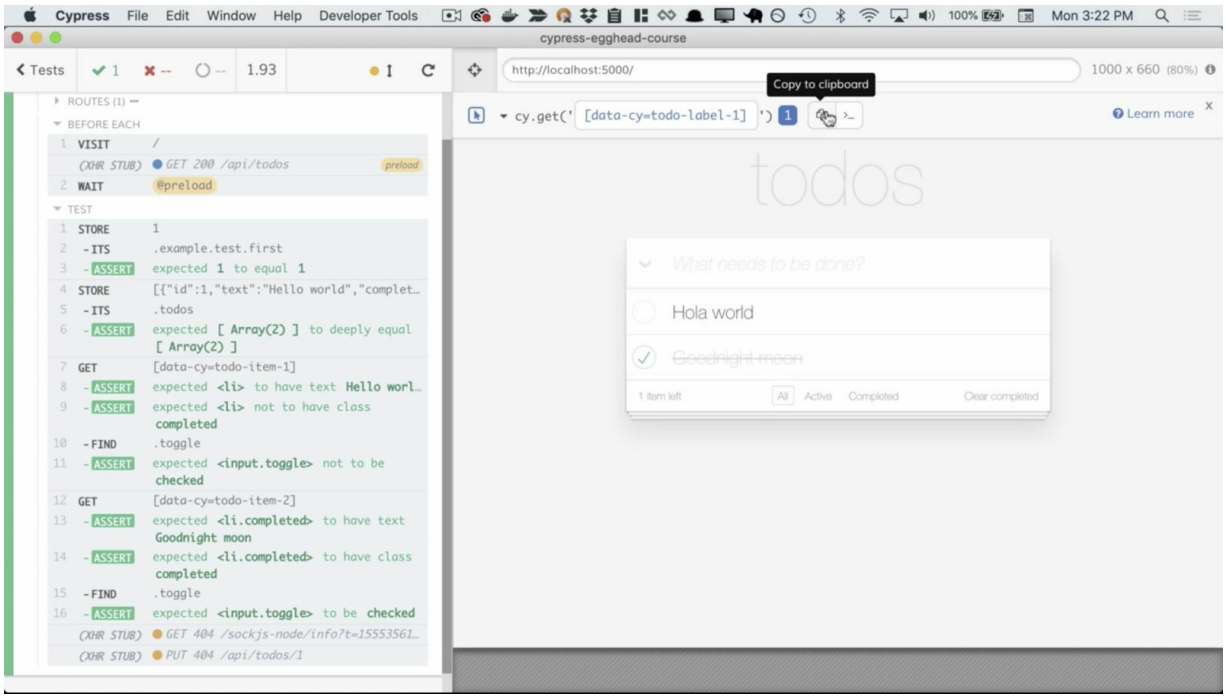
When Cypress tries to preload our todos, it returns a 404, which breaks our tests because the test expects two todos to be visible on the page.



This is a great way to identify stubs that we haven't yet configured. To see this in action, let's comment these back in and move on to creating a test where we edit one of our todos.

```
describe('Todo Application', () => {
  beforeEach(function() {
    cy.fixture('todos/all.json').as('todos')

    cy.server({force404: true})
    Alias the fixture data
    cy.route('/api/todos',
'@todos').as('preload')

    cy.visit('/')
    cy.wait('@preload')
  })
```

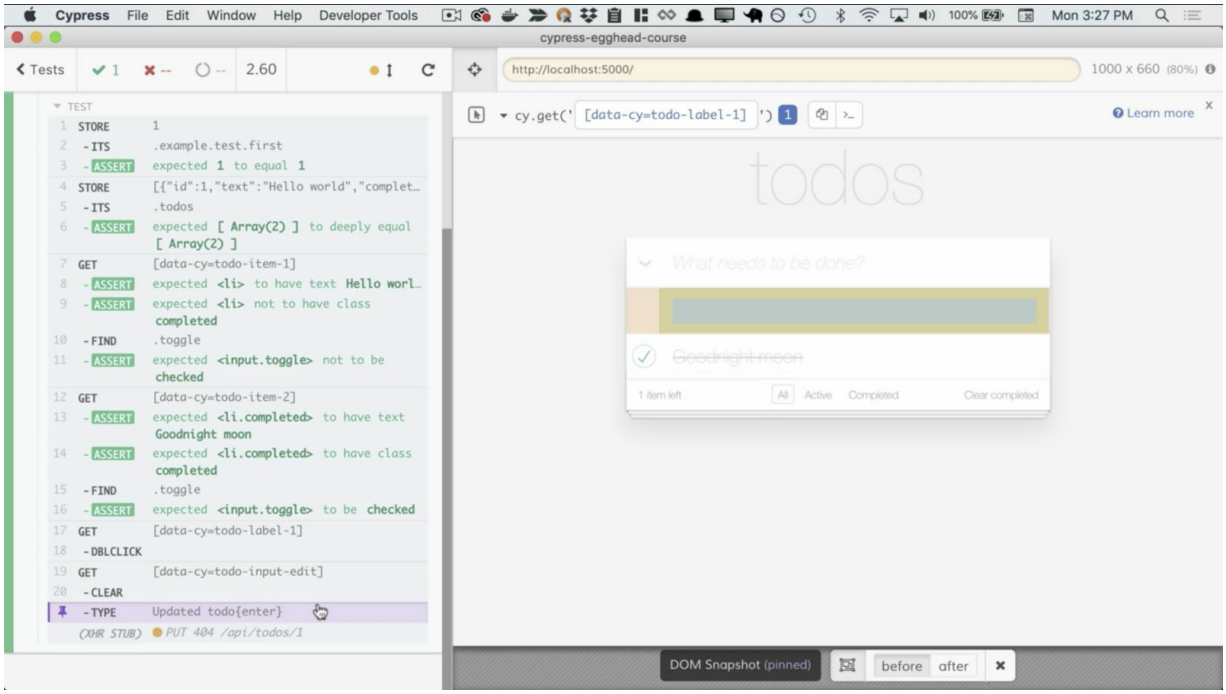
When we edit a todo, we double-click on the text field, change the text, and press enter. In order to replicate this, let's hit our selector playground, find the selector we need to target, and copy it to the clipboard.



Let's append the selector to our test and double-click it. In our application, when we `dblclick()` on a todo, a new input appears on the page. That input is called `todo-input-edit`. If we want to clear the text in it, we can call `.clear`. Then we can `.type('Updated todo{enter}')`.

```
it('loads the page', function () {

cy.store('example.test.first').should('equal',
1)

    // Access the fixture data as this.todos
    cy.store('todos').should('deep.equal',
this.todos)

    cy.get('[data-cy=todo-item-1]')
      .should('have.text', 'Hello world')
      .should('not.have.class', 'completed')
      .find('.toggle')
      .should('not.be.checked')

    cy.get('[data-cy=todo-item-2]')
      .should('have.text', 'Goodnight moon')
      .should('have.class', 'completed')
      .find('.toggle')
      .should('be.checked')

    cy.get('[data-cy=todo-label-1]').dblclick()
    cy.get('[data-cy=todo-input-
edit]').clear().type('Updated todo{enter}')
  })
```

Back in Cypress, we see that we targeted the label. We doubled-clicked it, which caused the new input edit to appear on the page. We targeted that, cleared it, then typed updated todo enter.

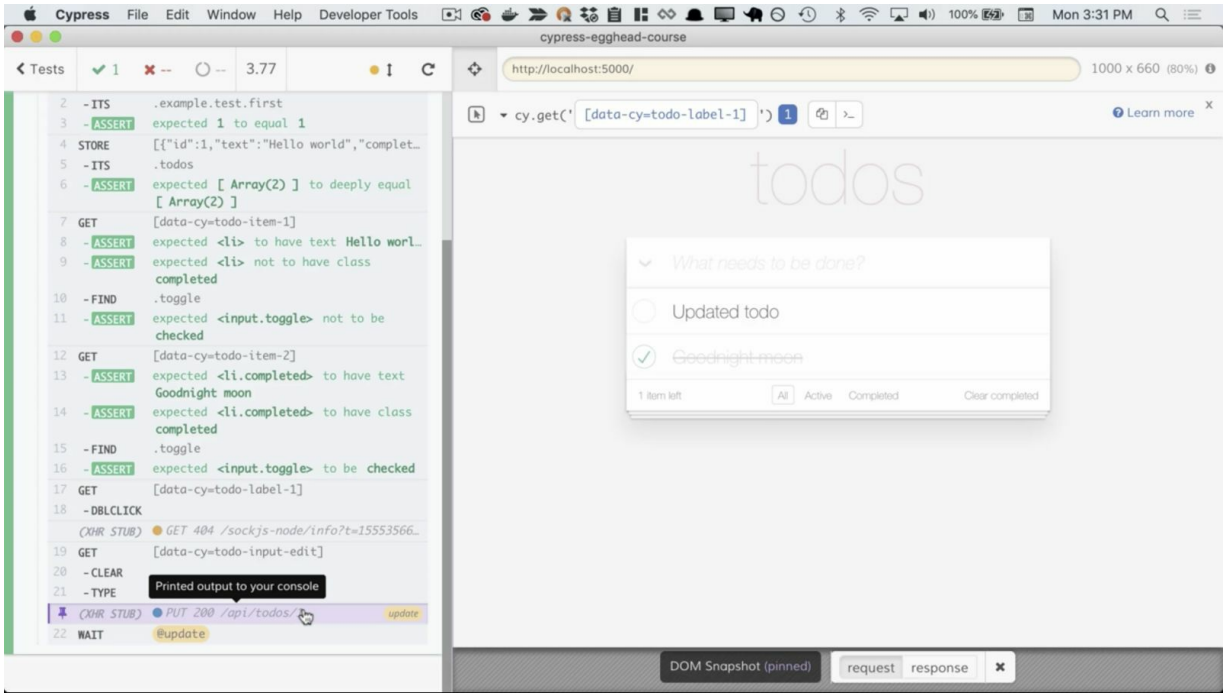


Finally, we see we have a 404 to the request `api/todos/1`, which is the update endpoint. The 404 alerts us that we haven't yet stubbed out our endpoint. Let's stub it out.

This will be a `cy.route`. It'll be a `cy.route('PUT', '/api/todos/1', 'ok').as('update')`. Now all we need to do is `cy.wait('@update')`.

```
    cy.route('PUT', '/api/todos/1',
'ok').as('update')
    cy.get('[data-cy=todo-label-1]').dblclick()
    cy.get('[data-cy=todo-input-
edit]').clear().type('Updated todo{enter}')
    cy.wait('@update')
  })
```

If we return to our test, we'll see that we no longer have a 404, but a 200 for the update.



I find it's helpful to configure cy.server with force 404 anytime we aren't directly interacting with our back end.

Extend Cypress with Plugins

Most of your Cypress code will be executed in the browser. For certain types of challenges like seeding a database, sending emails, or otherwise testing the impact of jobs that might be queued as a result of your frontend interactions, you'll want to write a Cypress plugin.

Plugins are a scene for you to write your own custom code that executes during particular stages of the Cypress lifecycle.

Let's go ahead and write a Hello World version of a Cypress plugin to see how it works. In an `index.js` file, let's say `on('task')` which we'll call with `cy.task`. We'll call the task `hello`. Hello will accept a `name`. Finally, we'll `console.log`. The log will say `('hello ${name}')`. We'll `return null`, which for a Cypress plugin means it executed successfully.

index.js

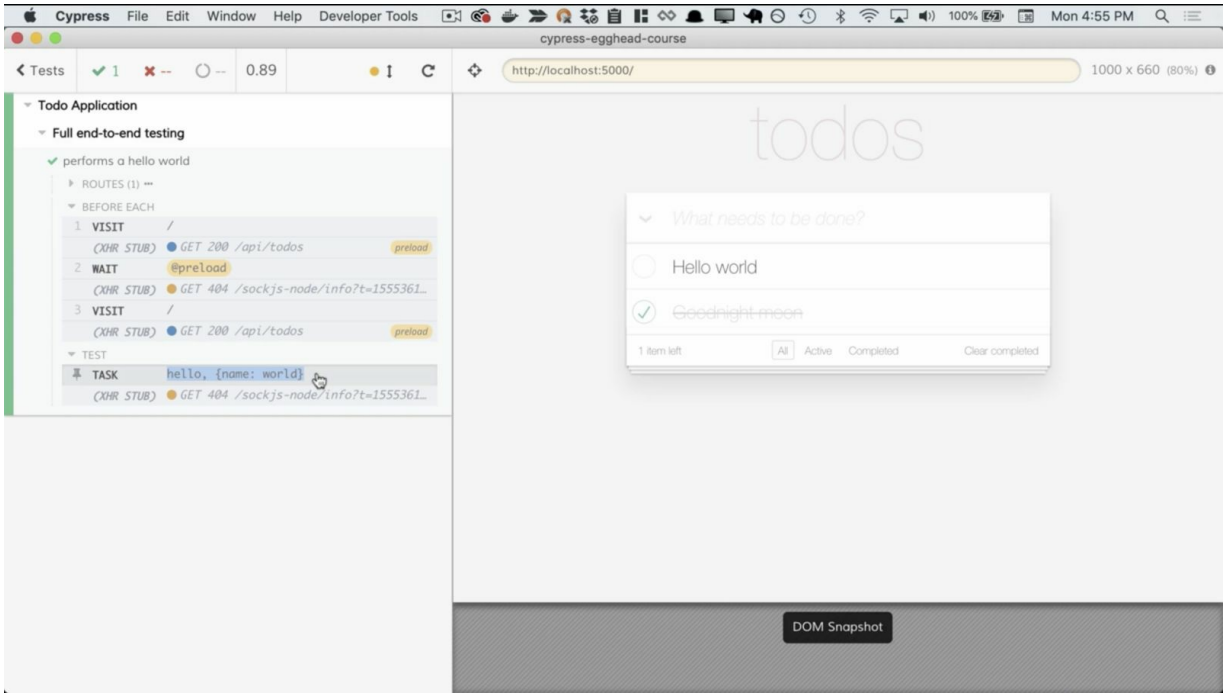
```
module.exports = (on, config) => {

    on('task', {
        hello ({name}) {
            console.log('hello ${name}')
            return null
        }
    })
}
```

If we switch over to our test file, we can call this with `cy.task` calling the task `hello` and passing a `name: world`.

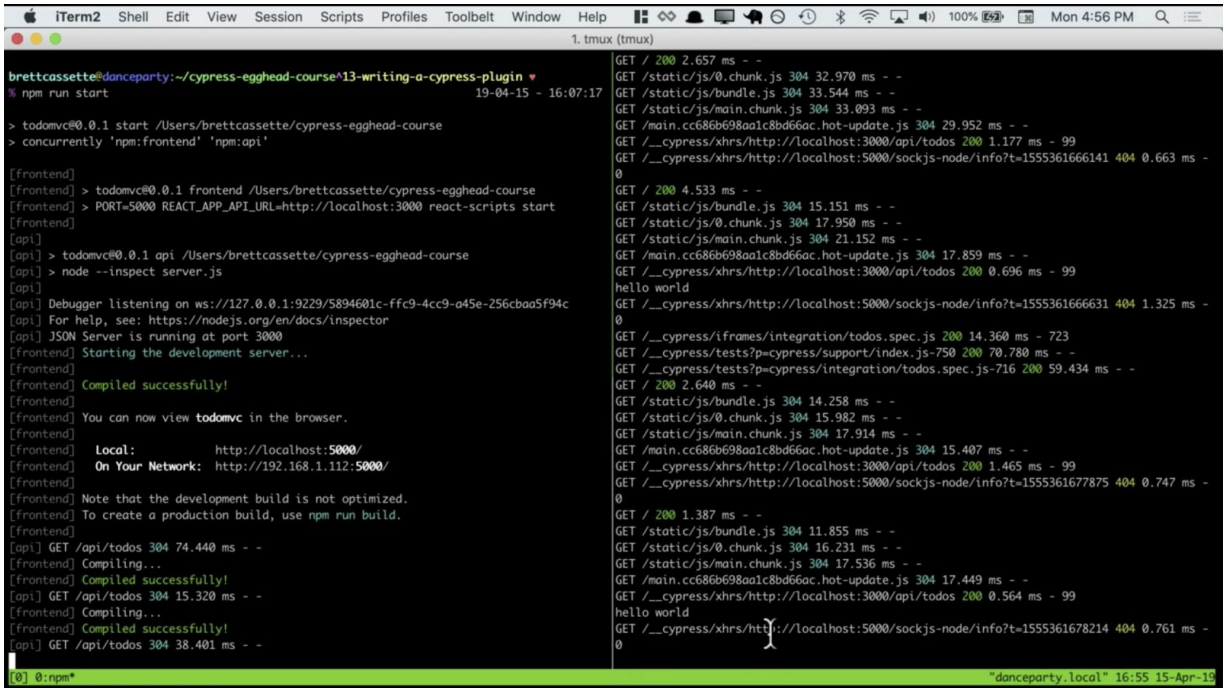
todos.spec.js

```
context.only('Full end-to-end testing',
function() {
    beforeEach(function() {
        cy.visit('/')
    })

    it('performs a hello world', function() {
        cy.task('hello', {name: 'world'})
    })
}
```

From Cypress, we'll see the task logged out.



Logging out the name of the task and the arguments we passed. If we look at our Cypress runner, we can see that hello world gets logged out in our console.



We take notice that the task took place in the command line, not in the browser and that we're going to use task to interact with our server environment.

Also, that Cypress isn't running in our web pack server or our backend server. Cypress is running in its own execution context with its own version of node. It can't just happen to the running server. We have to trigger tasks that will do the work for us on the backend.

In the next lesson, we'll see this in action, when we write a task to see their database.

Seed Your Database in Cypress

Up until now, we've been stubbing out our backend in tests with `cy.fixture` and `cy.route`. In order to do full end-to-end tests, we're going to want to start seeding our database. Let's write a Cypress task that seeds the database.

We can call the `cy.task('db:seed'` and that'll take an argument of a hash whose keys are the names of our model. `todos` is our first model right now.

We can pass an array of todos. In the future, we could add a users model and pass an array of users or any other models we might think of, but for right now, we just have todos. We'll pass an array of hashes. In fact, we can just copy in the data from our `fixtures/all.json` file and paste that in.

todos.spec.js

```
context.only('Full end to end testing', function
() {
    beforeEach(function() {
        cy.visit('/')
    })

    it('Performs a hello world', function() {
        cy,task('hello', {name: "world"})
    })

    it('seeds the database', function() {
        cy.task('db:seed', {
            todos: [{
                "id": 1,
                "text": "Hello world",
                "completed": false
            },
            {
                "id": 2,
                "text": "Goodnight moon",
                "completed": true
            }]
        })
    })
})
```

Now that we know what our task will look like, let's go ahead and define it in `cypress/plugins/index.js` where we defined our Hello World task.

This task is called `'db:seed'` and it takes an argument which is our seeds. We'll assume that there's going to be some kind of a default case. The `defaultSeed` would be an empty `todos` array and that's just us resetting the database to a clean, empty state.

Then we're either going to use the `seeds` if they were passed in or the default seed. We'll say the `seedsToUse = seeds ? seeds : defaultSeed`.

index.js

```
module.exports = (on, config) => {

    on('task', {
        hello ({name}) {
            console.log('hello ${name}')
            return null
        },

        'db:seed': (seeds) => {
            let defaultSeed = {todos: []}
            let seedsToUse = seeds ? seeds :
defaultSeed
        }
    })
}
```

We're going to write a library. We'll call it `db`. We'll import that and we'll say that it has a `seed` method. We'll pass it into `seedsToUse`. We `return null`, remembering that null is the case where the plugin is used successfully.

```
module.exports = (on, config) => {

    on('task', {
        hello ({name}) {
            console.log('hello ${name}')
            return null
        },

        'db:seed': (seeds) => {
            let defaultSeed = {todos: []}
            let seedsToUse = seeds ? seeds :
defaultSeed

            db.seed(seedsToUse)

            return null
        }
    })
}
```

Then we need to import our db library. We'll say `const db = require()` and this will be relative to our path, so go up to Cypress, plugins, yeah. If we want it to be in our route to this, we'll call it `db-seeder.js`.

```
const db = require('../../db-seeder.js')
module.exports = (on, config) => {

    on('task', {
        hello ({name}) {
            console.log('hello ${name}')
            return null
        },

        'db:seed': (seeds) => {
            let defaultSeed = {todos: []}
            let seedsToUse = seeds ? seeds :
defaultSeed

            db.seed(seedsToUse)

            return null
        }
    })
}
```

Now we have to write our db seeder, so we'll create a new file, call it db seeder. For this project, we used the `lowdb` library. I'm going to copy in some boilerplate for writing to that particular database. If you happen to use Postgres or MySQL, you'll have your own version of this code.

Here's our `module`. We had a `seed` function that takes some `state`. That's the seeds that we passed in in the plug-in. This library writes to a JSON file which is located at the root of our project. This is just some more boilerplate for writing to this particular database. Again, use your own version of this code and `setState`. That's it. That's all the boilerplate.

db-seeder.js

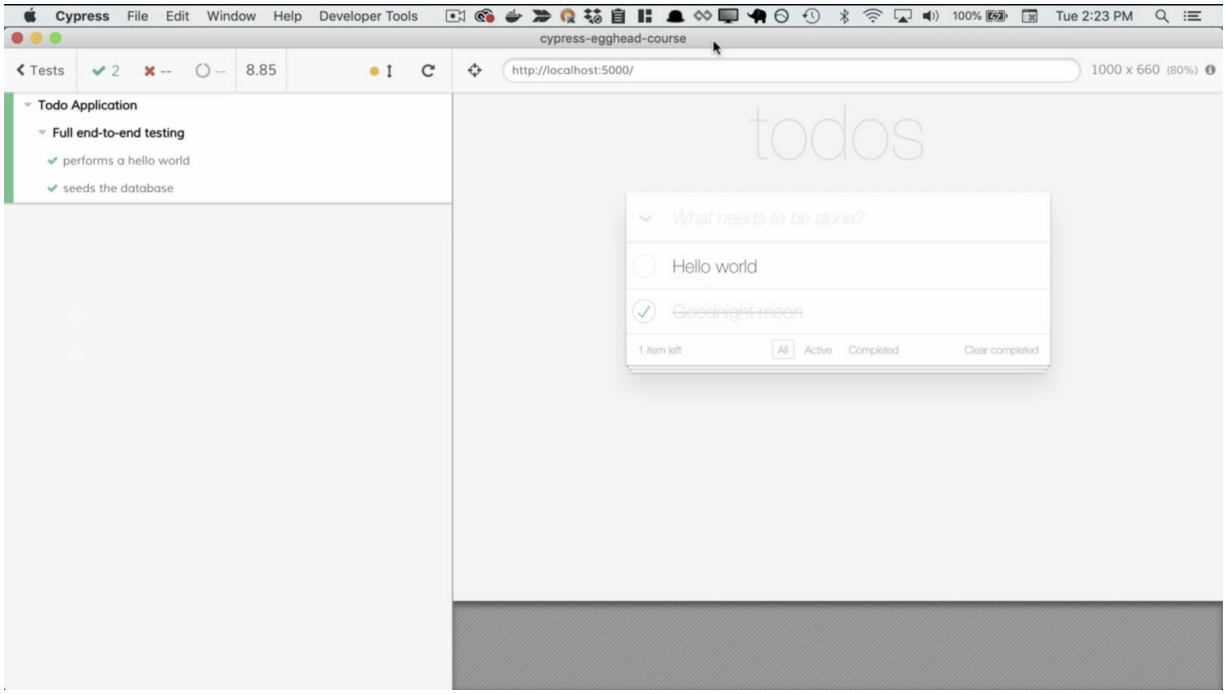
```
const low = require('lowdb')
const FileSync =
require('lowdb/adapters/FileSync')

module.exports = {
    seed: function(state) {
        let adapter = new FileSync('db.json')
        let db = low(adapter)
        db.setState(state).write()
    }
}
```

We just have to rerun Cypress so it can pick up our new plug-in.

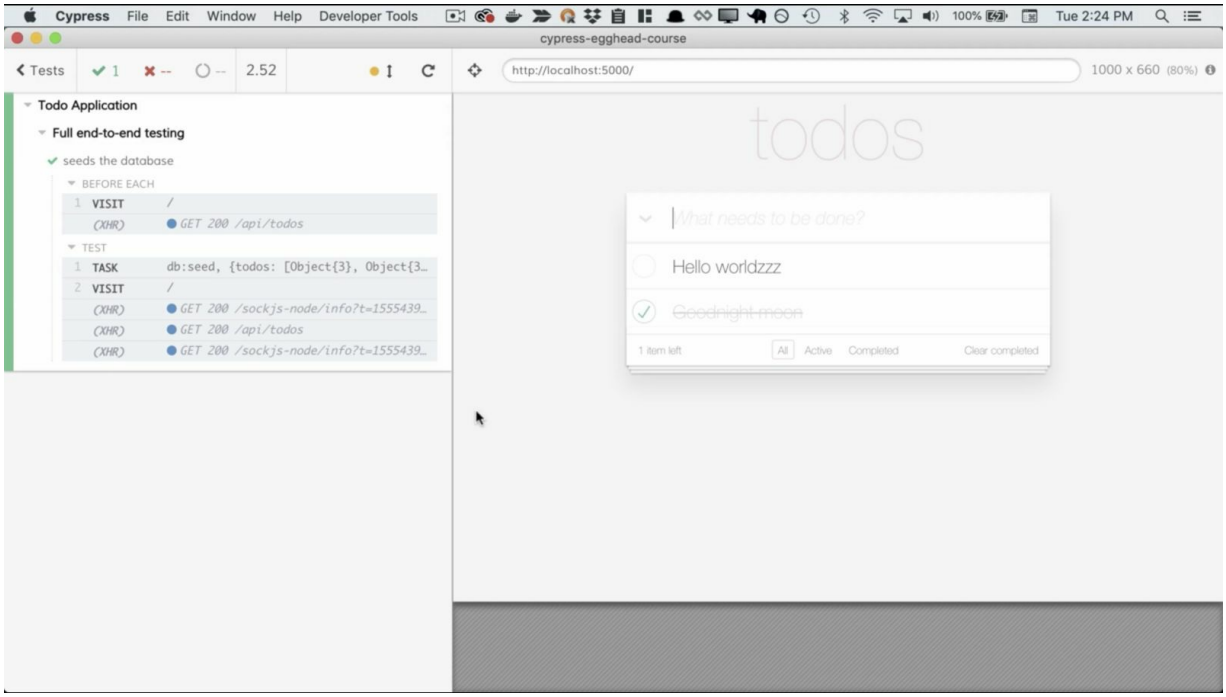
terminal

```
$ npm run cypress
```

Let's go ahead and make sure that we visit the page after we've seeded the database. That way, we can see the changes on the page, and we do see them there.

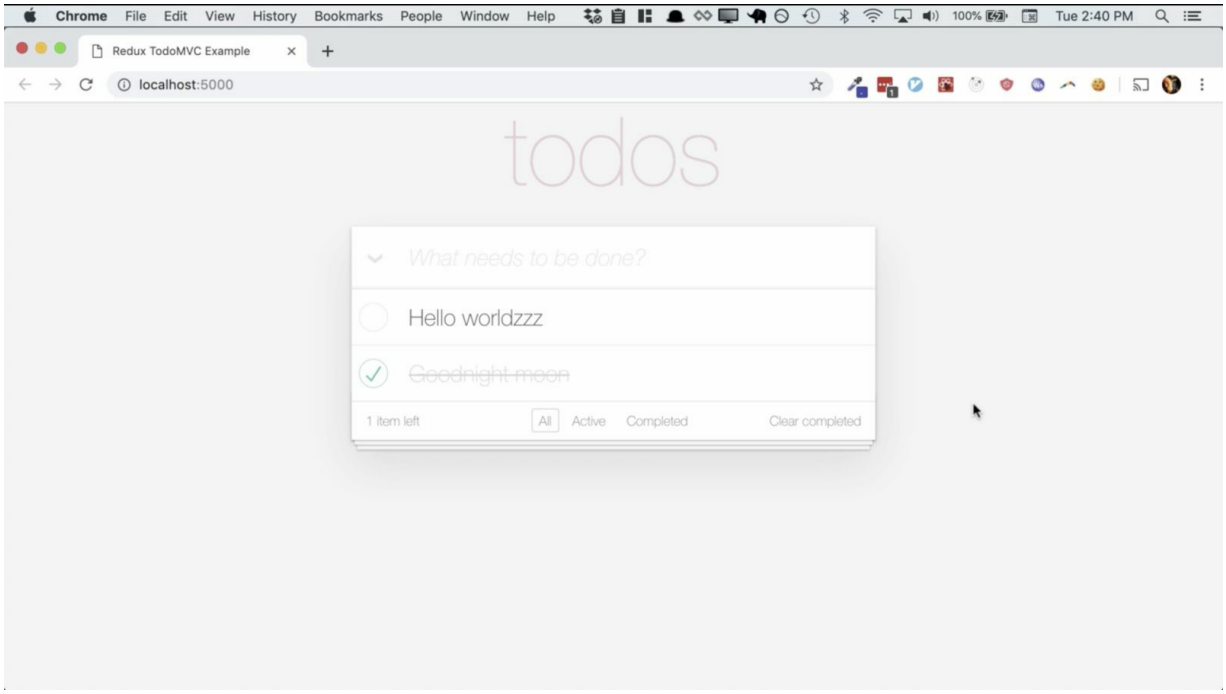


Just to double-check that this is in fact working the way we expect it to, let's update one of our seeds, and we see the changes reflected. That's great.



Let's go into our database file, and we can see the changes reflected in our database. We can see that in our test file, we're actually able to interact with and update our backend. That's

pretty cool. But one problem that we are going to notice is that if we visit our actual development environment, oh no, the changes are reflected here as well.



The reason for this is because we have no separate testing environment. We're using the same db.json file in both our tests and our dev workspace.

While every application will have different requirements for separating the dev and test environments, let's go ahead and take a look at some of the basics.

Since we're going to have a separate version of the server running for both dev and test, we'll need to configure the port and the database file for each so we can separate them. We can detect the node environment. If it's `test` then we will configure the port to be `3001` and the `dbFile` to be `db.test.json`. Otherwise, we'll just use the defaults that we've used before.

server.js

```
let port, dbFile;

if (process.env.NODE_ENV == 'test') {
    port = '3001'
    dbFile = 'db.test.json'
} else {
  port = '3000'
  dbFile = 'db.json'
}
```

Then all we have to do is go through the file, find any place the database has used and replace that with `dbFile`. You can see it's hard-coded in a couple of places, so we just replace these. The same thing with the port that's used down here. We'll just pass in the port.

Let's go ahead and create a new file called `db.test.json`. We can preset that up with an empty todos array, save that out.

db.test.json

```
{
    "todos": []
}
```

In our `package.json`, we have the scripts that we run when we actually run our project.

Let's create a separate frontend test environment and API test environment. We have to pass `NODE_ENV=test`, change the port to `5001` and `3001`. When we run npm-run-start, which starts all of

our processes, we also want it to start `"frontend-test"` and `"api-test"`. We'll just go ahead and copy these in and there we go.

package.json

```
...

  "scripts": {
    "start": "concurrently 'npm:frontend'
'npm:api' 'npm:frontend-test' 'npm:api-test'",
    "frontend": "PORT=5000
REACT_APP_API_URL=http://localhost:3000 react-
scripts start",
    "api": "node server.js",
    "frontend-test": "PORT=5001
REACT_APP_API_URL=http://localhost:3001 react-
scripts start",
    "api-test": "NODE_ENV=test node server.js",
    "build": "react-scripts build",
    "eject": "react-scripts eject",
    "test": "react-scripts test --env=node",
    "cypress": "cypress open"
  },

  ...
```

Our frontend needs to know what the `REACT_APP_API_URL` is, so that whenever we make a call to the backend, we call this URL.

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore, applyMiddleware } from
'redux'
import createSagaMiddleware from 'redux-saga'
import { Provider } from 'react-redux'
import { rootSaga } from './sagas/TodoSagas'
import App from './components/App'
import reducer from './reducers'
import 'todomvc-app-css/index.css'

const sagaMiddleware = createSagaMiddleware()
const store = createStore(reducer,
applyMiddleware(sagaMiddleware))
sagaMiddleware.run(rootSaga)

window.REACT_APP_API_URL =
process.env.REACT_APP_API_URL

...
```

We want to make sure that we have isolated the locations in our application that actually call the backend. For us, they exist in this sagas file. We have this `getBaseUrl` which knows the application base URL and returns it. For us, we'll just say it's this `REACT_APP_API_URL`. If it's present, we will return that. Otherwise, we'll just return this default which was our dev environment URL.

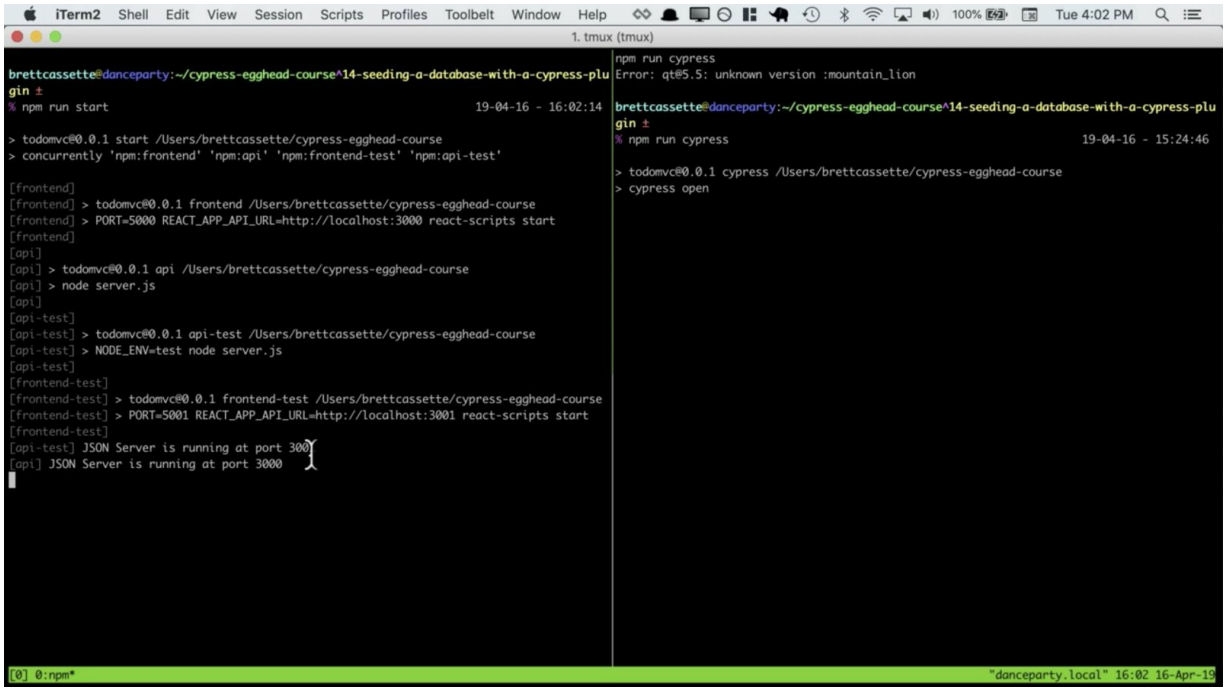
TodoSagas.js

```
import { takeEvery, takeLatest, put, all, select
} from "redux-saga/effects";
import axios from "axios";

function getBaseUrl() {
  return window.REACT_APP_API_URL =
process.env.REACT_APP_API_URL :
'http://localhost:3000'
}

...
```

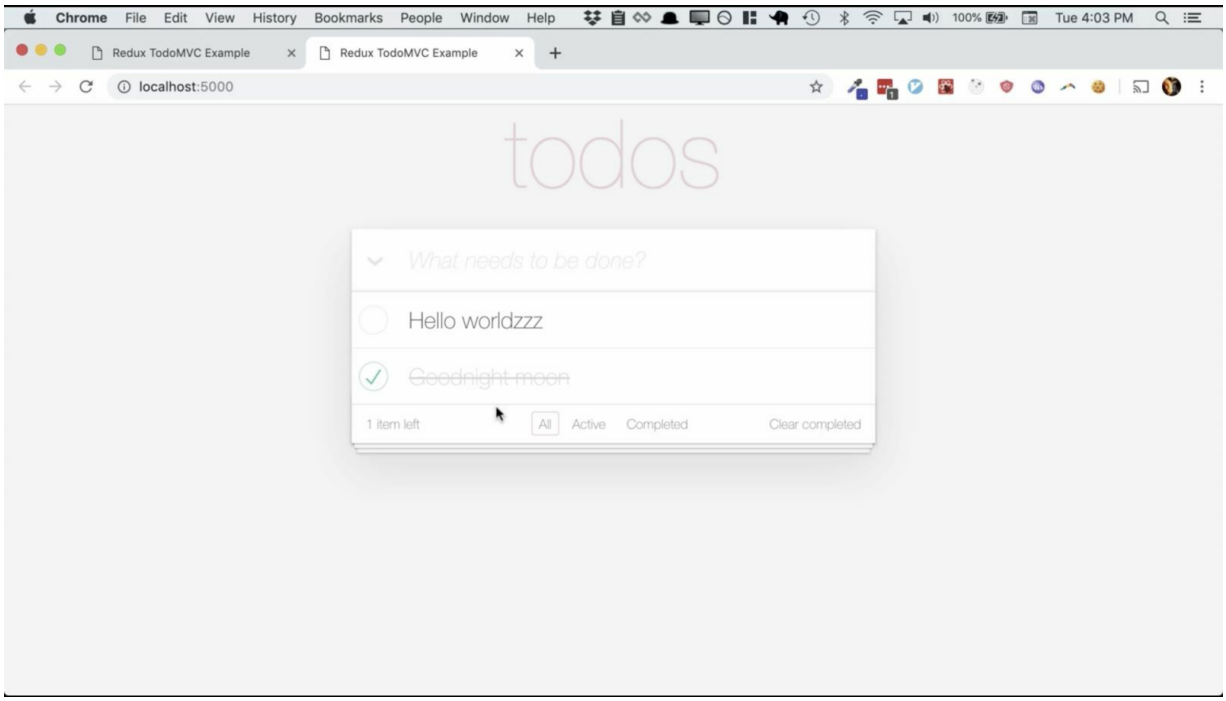
When we actually run `npm run start`, we'll see that we start up four separate processes -- a frontend and frontend test, an API and API test.

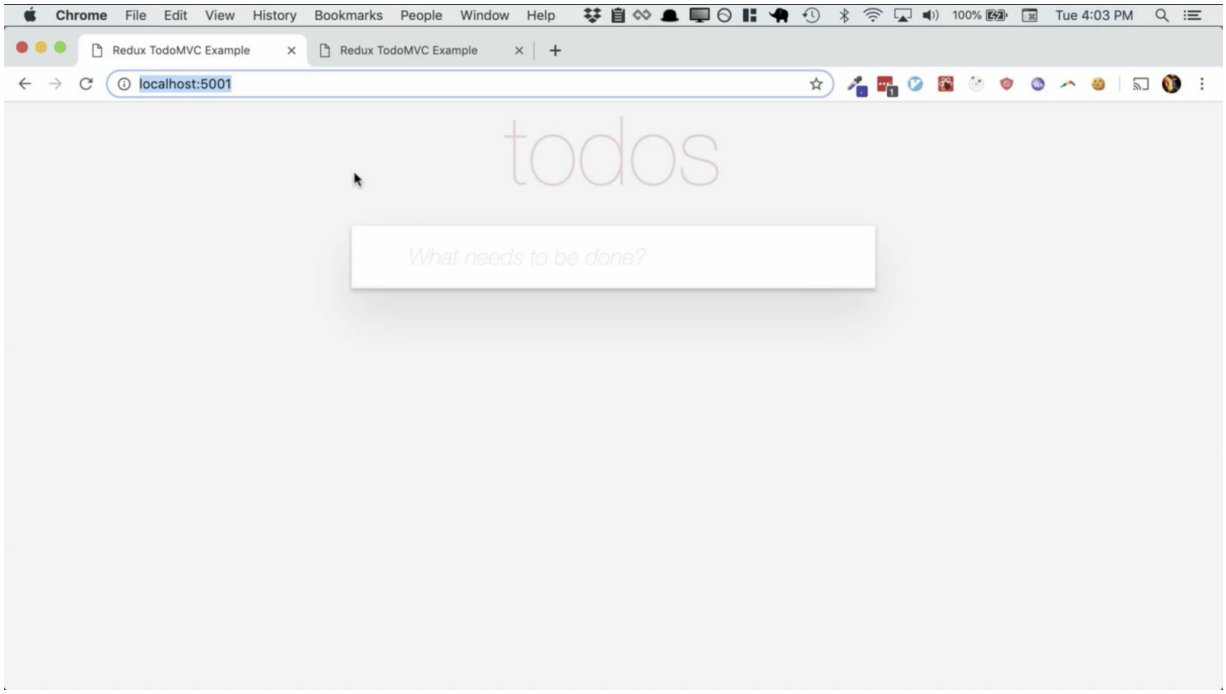
terminal

```
$ npm run start
```



These will all run on separate ports and using separate database files.When these load up, we'll see that we have test environment running on port 5001 and a dev environment running on port 5000.





Clearly, they have different databases here.

You can say that this is the dev todo one. This has not impacted our testing environment against a test todo one. These are totally isolated.

Productionize Your Database Seeder in Cypress

In the last lesson, we saw how to make a basic database seeding abstraction that's suitable to our test app. In the real world, we'll want something a little more robust. For instance, it would allow us to set defaults. Completed is false by default, so it doesn't make sense to have to specify that every single time.

Next, id numbers will be a little bit tedious to maintain. It would make more sense for these to be auto-generated for us. Finally, if we wanted to override an attribute, like we need `completed` to be `true` in one case, then we just need to specify that, and our abstraction will respect it.

todos.spec.js

```
it.only('seeds the database', function() P{
    cy.task('db.seed', {todos: [{text: 'Seed the
database', completed: true}]})

    ...

})
```

One way to think about what we're aiming at is that we want our abstraction to provide some sensible defaults while still giving the end user complete control to override whatever they'd like.

Let's make a new abstraction called a factory. A factory will define the defaults for each of our object types. We can make a manifest file under Cypress. We'll make a new folder called `factories`, and then a new file called `factories.js`. This will be our manifest.

We'll say `module.exports` equals, and then create a hash. Our manifest file will simply require the related files that it needs. We know that we'll define todos under a file we will created in the factories folder, `('./todos.js')`

factories.js

```
module.exports = {
    todos: require('./todos.js')
}
```

This is where we'll define our defaults. We'll say `module.exports`. We'll say the default text is `'Hello World.'` The default `completed: false`. We want our seeding abstraction to support these factory defaults.

factories/todos,js

```
module.exports = {
    text: "Hello World",
    completed: false
}
```

Let's open up `cypress/support/commands.js`. We'll create a new Cypress command to do this. Let's go ahead and scroll to the top of our file, and require our manifest file as `const factories = require('../factories/factories.js')`.

commands.js

```
const _ require('lodash')
const factories =
require('../factories/factories.js')
```

Let's add our new Cypress command. It's called `seed`. It accepts a hash of `seeds`, as well as some options, so we configure it not to log if we don't want it to. In order to apply our defaults, which we'll call `mappedSeeds`, we'll need to `_.reduce` our `seeds`. The reducer function will have an `output`. That's the seeds with the defaults applied. It will have an array of `seeds` that we're currently applying defaults to. It will have a current `key`, which is the name of the model, aka todos in this case.

```
const _ require('lodash')
const factories =
require('../factories/factories.js')

...

Cypress.Commands.add('seed', (seeds, options =
{}) => {
    let mappedSeeds = _.reduce(seeds, (output,
seeds, key))
})
```

Then we need to pass in a hash that will serve as our output to the function, which will hold the seeds with defaults applied. Let's start writing the reducer function. First, we find the `factory` for the current model. We'll use our manifest file to look up the correct factory. In this case, it's the todos factory. Otherwise, we can explicitly label that the factory is undefined.

```
const _ require('lodash')
const factories =
require('../factories/factories.js')

...

Cypress.Commands.add('seed', (seeds, options =
{}) => {
    let mappedSeeds = _.reduce(seeds, (output,
seeds, key) => {
        let factory = factories[key] []
undefined;
    })
})
```

If the factory is undefined, we'll go ahead and specify that the seeds with defaults applied is just the user-specified seeds because we haven't defined any factory defaults yet. If we have defined a factory, we'll go ahead and `seeds.map((seed)`. We'll use `_.defaults(seed, factory)` to apply the factory defaults.

```
...

Cypress.Commands.add('seed', (seeds, options =
{}) => {
    let mappedSeeds = _.reduce(seeds, (output,
seeds, key) => {
        let factory = factories[key] []
undefined;

        if (_.isUndefined(factory)) {
            output[key] [] seeds;
        } else {
            output[key] = seeds.map((seed) => {
                return _.defaults(seed, factory)
            })
        }

        return output
    }, {})
})
```

If `options.log` is not `false`, then we'll `log` out to Cypress. The name of the log is `seed`. `message` is the seeds with defaults applied. `consoleProps`, same thing.

```
...

Cypress.Commands.add('seed', (seeds, options =
{}) => {
    let mappedSeeds = _.reduce(seeds, (output,
seeds, key) => {
        let factory = factories[key] []
undefined;

        if (_.isUndefined(factory)) {
            output[key] [] seeds;
        } else {
            output[key] = seeds.map((seed) => {
                return _.defaults(seed, factory)
            })
        }

        return output
    }, {})

    if(options.log != false) {
        Cypress.log({
            name: 'seed',
            message:
JSON.stringify(mappedSeeds),
            consoleProps: () => { return
mappedSeeds }
        })
    }
})
```

Finally, we'll actually call our `cy.task` that we defined in the last episode. We pass it the seeds with defaults applied. We make sure it doesn't `log` because we already created our own logging functionality here. Great.

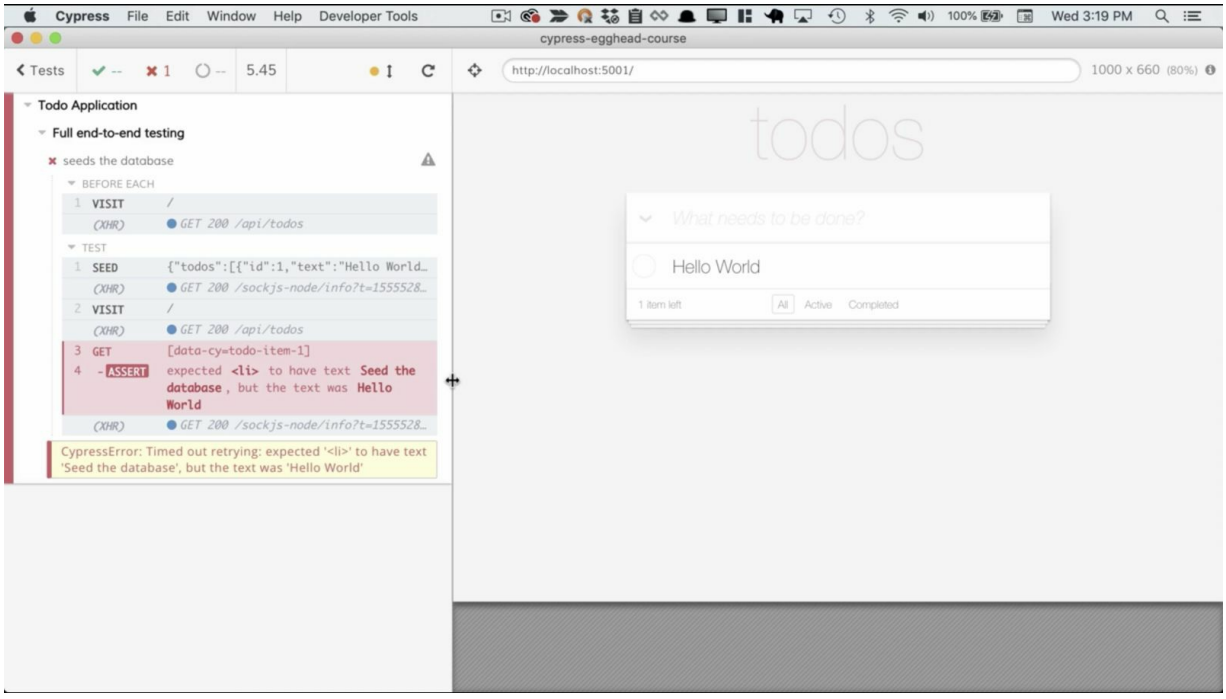
```
...

Cypress.Commands.add('seed', (seeds, options =
{}) => {
    let mappedSeeds = _.reduce(seeds, (output,
seeds, key) => {
        let factory = factories[key] []
undefined;

        if (_.isUndefined(factory)) {
            output[key] [] seeds;
        } else {
            output[key] = seeds.map((seed) => {
                return _.defaults(seed, factory)
            })
        }

        return output
    }, {})

    if(options.log != false) {
        Cypress.log({
            name: 'seed',
            message:
JSON.stringify(mappedSeeds),
            consoleProps: () => { return
mappedSeeds }
        })
    }

    cy.task('db:seed', mappedSeeds, {log:
false})
})
```

Let's go ahead and use our new command in our test. We can pop back to our test file, change `cy.task` to `cy.seed`. No longer need to call the task. Let's delete everything that's not default. Obviously, we haven't set primary keys yet, so we still have to add a primary key.

todos.spec.js

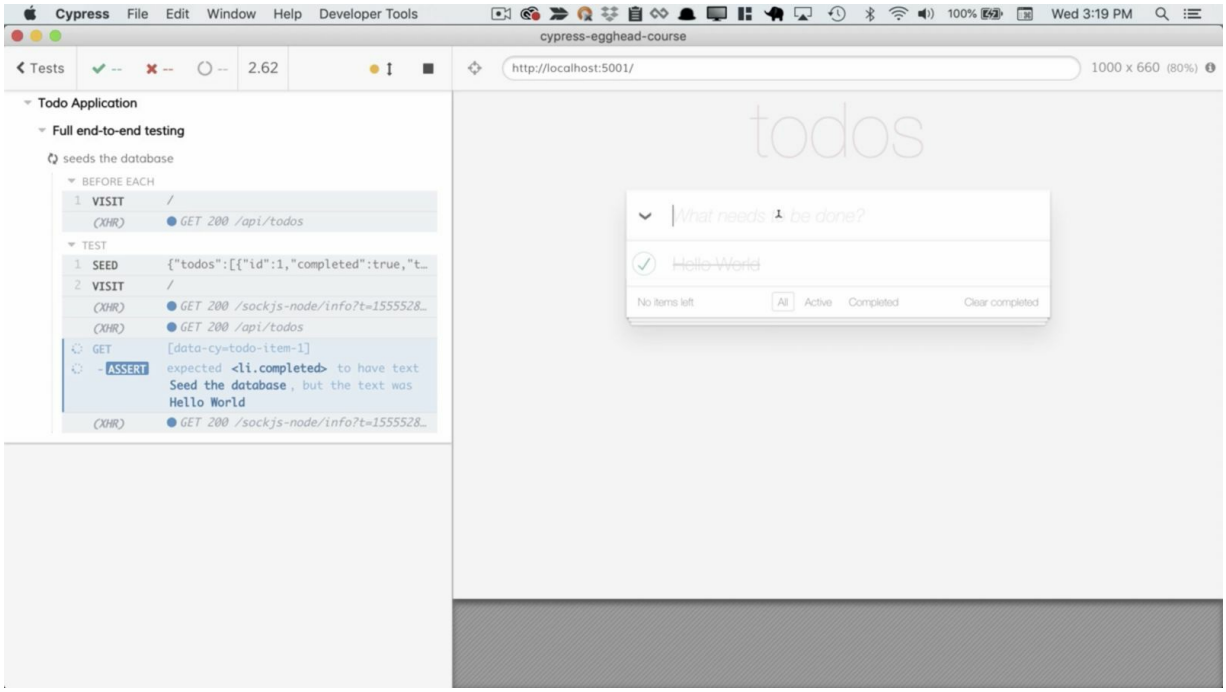
```
it.only('seeds the database', function() P{
    cy.seed({todos: [{id: 1}]})

    ...

})
```

Let's do this and reopen Cypress. We'll see that we have the text, "Hello World," which was our default. We don't have completed set. That's great.



We can override completed and see that that should override it for us. We see that it does, awesome.

```
it.only('seeds the database', function() P{
    cy.seed({todos: [{id: 1, completed: true}]})

    ...

})
```



The one thing we have left to do is figure out primary keys. Let's look into that.

Let's pop up in our console. We'll create a generator function. A `*gen` function is a function that can stop midway through, and then continue where it left off. We ca use a generator to generate a series of numbers that continues increasing.

First, we'll define the `ID` variable. This is the variable that we're going to increment.

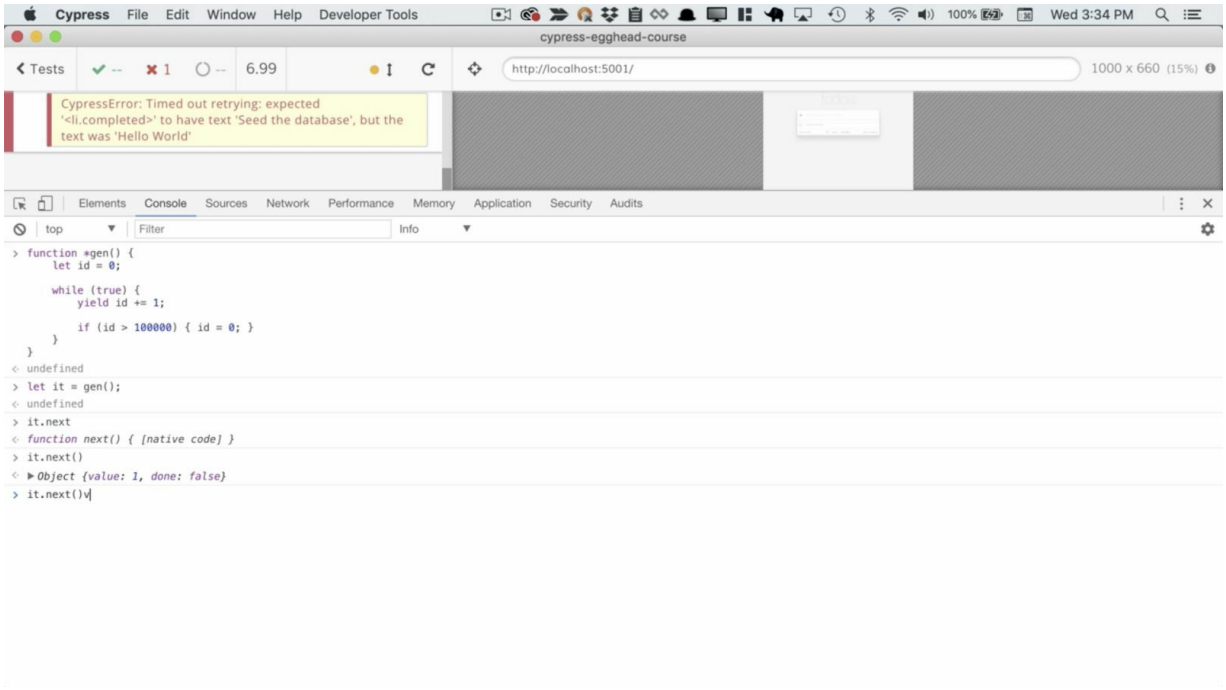
Next, we'll define a loop that never exits. Whenever we call our generator, it will always yield the next number in the series. We'll say `while (true)`, then this is where we pause the function. We'll say `yield` when we're yielding the next number in the series, and then we pause. Next time, when we run the function, we un-pause and yield the next value, then re-pause, and repeat over and over.

Finally, we'll just check if the value has gotten too large, at which point, we'll just reset back to zero so it will go up to `100,000`, then it will go back to zero and start over again.

console

```
function *gen() {
    let id = 0;

    while (true) {
        yield id += 1;

        if(id> 10000) { id = 0; }
    }
}
```

We can just test this out and show that it works.



Let's hop back into our commands and create a hash of `generators`. We'll look through each of the keys of the factories and just create a new one for each. Let's paste in our generator function.

commands.js

```
const _ require('lodash')
const factories =
require('../factories/factories.js')
let generators = {}

function *gen() {
    let id = 0;

    while (true) {
        yield id += 1;

        if(id> 10000) { id = 0; }
    }
}
```

We'll say `_.each`. For each of the factories, we'll get the `factory` and the `key`, and we'll say `{ generators[key] = gen()}`.

```
const _ require('lodash')
const factories =
require('../factories/factories.js')
let generators = {}

function *gen() {
    let id = 0;

    while (true) {
        yield id += 1;

        if(id> 10000) { id = 0; }
    }
}

_.each(factories, (factory, key) => {
generators[key] = gen()}

...
```

Then we'll come down here to the location where we'll actually use this function. We'll keep applying the defaults in order. We'll just add an `id`. We'll say that that is the `generator` for this particular. We'll call `.next().value`. That will give us that value.

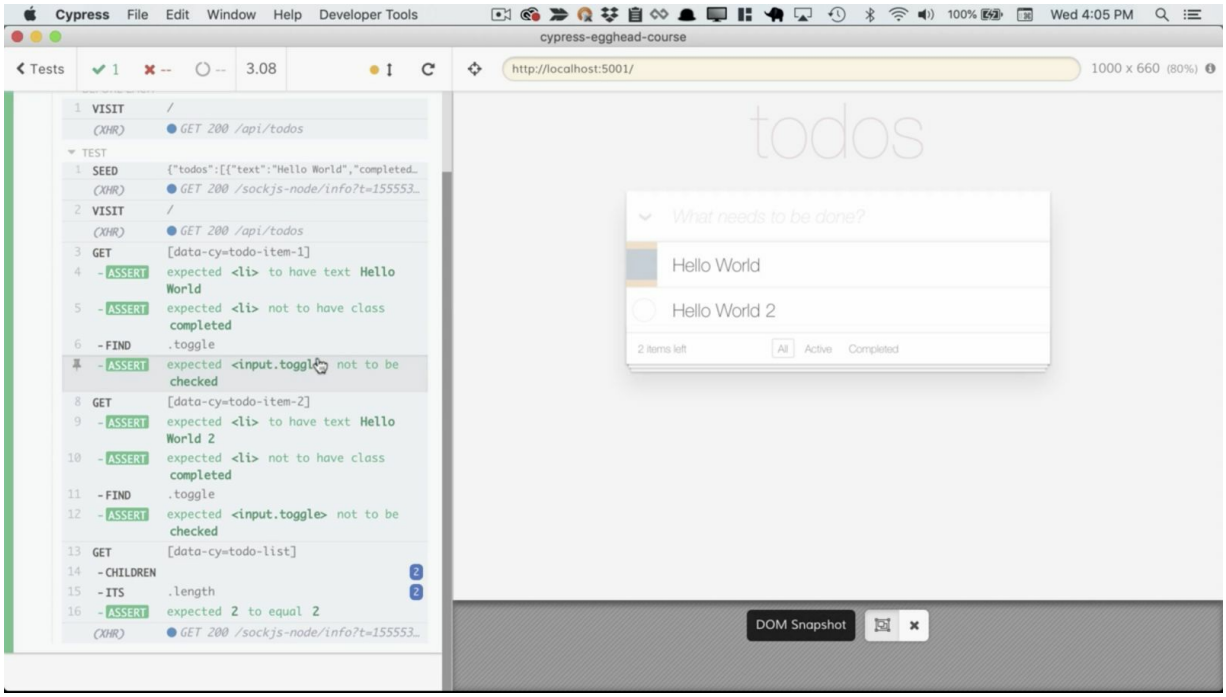
```
const _ require('lodash')
const factories =
require('../factories/factories.js')
let generators = {}
```

We hop back into our test. We can just delete any of these defaults now. We can create as second `todo`. We'll just say that it has the text, `"Hello World 2"`. We can copy out this assertion. We'll just change its primary key and its text to "`Hello World 2`". Again, it's getting this primary key `2` and one just based on the order these are created in. We'll say that the length of the todo list will now be two. Go ahead and update this to be `'Hello World'` for a default.

todos.spec.js

```
it.only('seeds the database', function() P{
    cy.seed({todos: [{}, {text: 'Hello World
2'}]})
    cy.visit('/')

    cy.get('[data-cy=todo-item-1]')
      .should('have.text', 'Hello World')
      .should('not.have.class', 'completed')
      .find('.toggle')
      .should('not.be.checked')

    cy.get('[data-cy=todo-item-1]')
      .should('have.text', 'Hello World 2')
      .should('not.have.class', 'completed')
      .find('.toggle')
      .should('not.be.checked')
    ...

})
```

Let's run our test. We'll see that it passes.

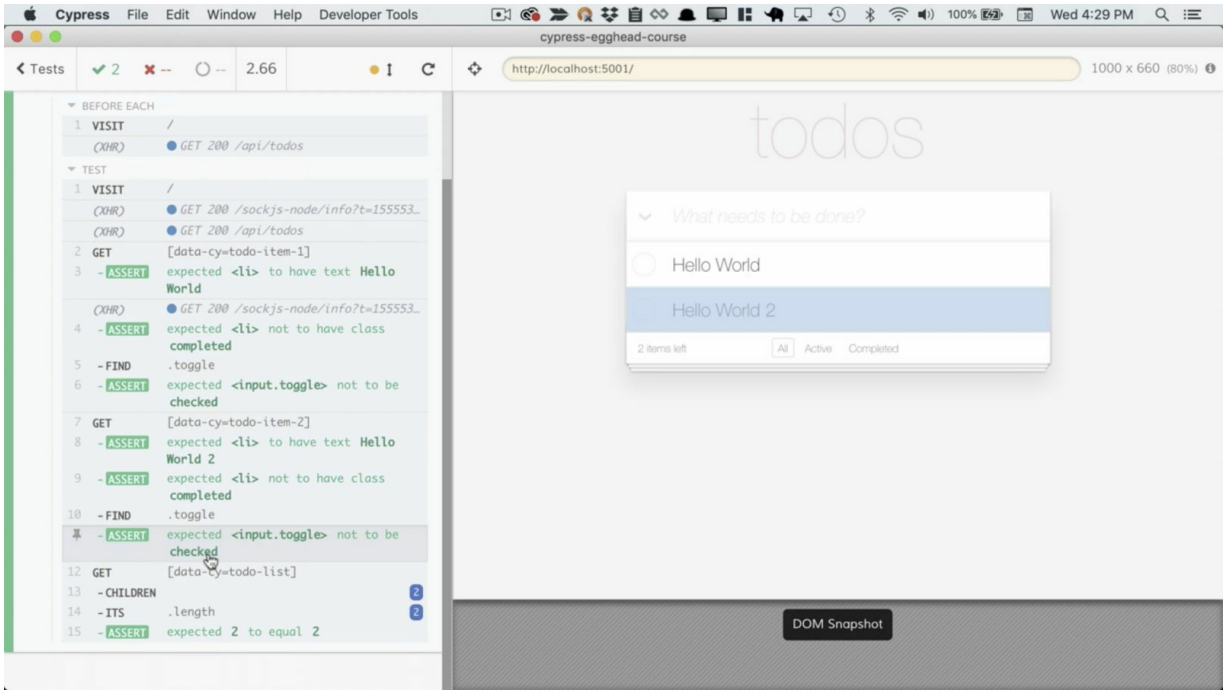


We just have a few more hiccups that we want to think about. For instance, if we run two versions of this test back to back, but let's say we comment out our `cy.seed` at the start of the second test, which means that we shouldn't have any todos, we should have an empty database in the second test.

todos.spec.js

```
it.only('seeds the database', function() P{
   // cy.seed({todos: [{}, {text: 'Hello World
2'}]})
    cy.visit('/')

    cy.get('[data-cy=todo-item-1]')
      .should('have.text', 'Hello World')
      .should('not.have.class', 'completed')
      .find('.toggle')
      .should('not.be.checked')

    cy.get('[data-cy=todo-item-1]')
      .should('have.text', 'Hello World 2')
      .should('not.have.class', 'completed')
      .find('.toggle')
      .should('not.be.checked')
    ...

})
```

We go and see actually that the second test passed and we do have all of our todos loaded in.

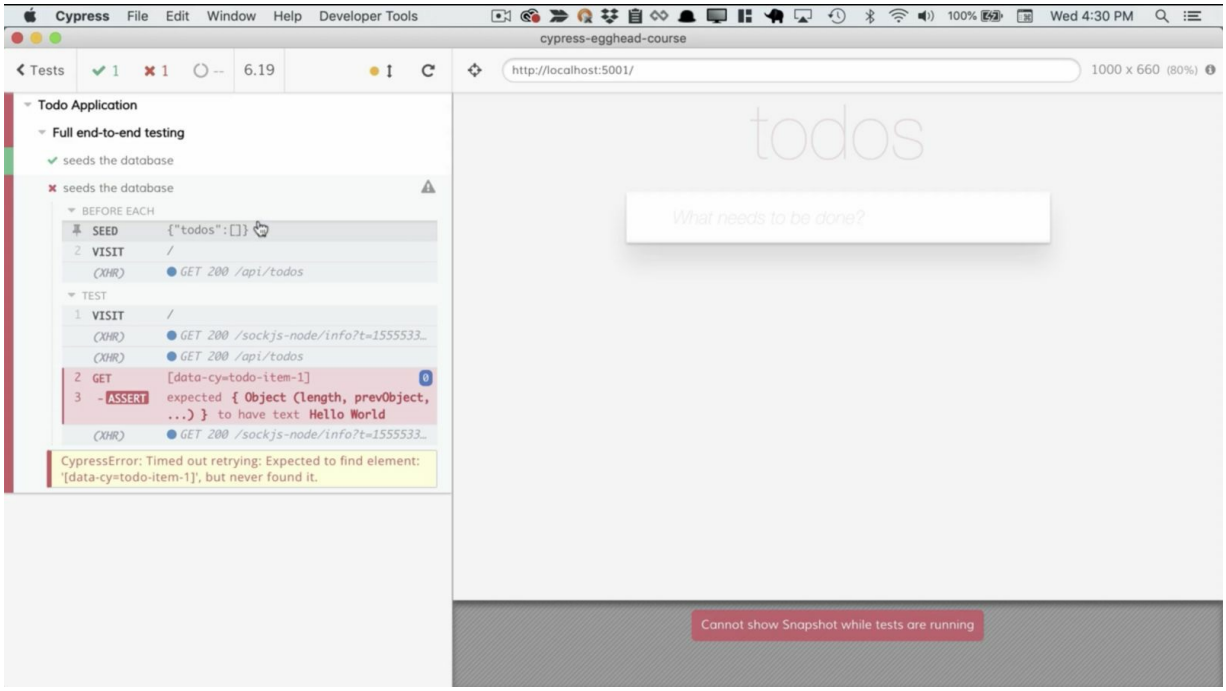


That's because we haven't reset the database in between every test. If we want to reset the database, what I like to do is go into our `commands.js`. We'll say `beforeEach`, and then in the beforeEach, we'll call `cy.seed`. Then we'll pass it our reset state.

```
const _ require('lodash')
const factories =
require('../factories/factories.js')
let generators = {}

...

Cypress.Commands.add('seed', (seeds, options =
{}) => {
    let mappedSeeds = _.reduce(seeds, (output,
seeds, key) => {
        let factory = factories[key] []
undefined;

        if (_.isUndefined(factory)) {
            output[key] [] seeds;
        } else {
```

```
            output[key] = seeds.map((seed) => {
                return _.defaults(seed, factory)
            })
        }

        return output
    }, {})

    if(options.log != false) {
        Cypress.log({
            name: 'seed',
            message:
JSON.stringify(mappedSeeds),
            consoleProps: () => { return
mappedSeeds }
        })
    }

    cy.task('db:seed', mappedSeeds, {log:
false})
})

beforeEach(function() {
    cy.seed({todos: []})
})
```

We can take a look at that. We'll see that the second time, we did actually seed an empty todos array.



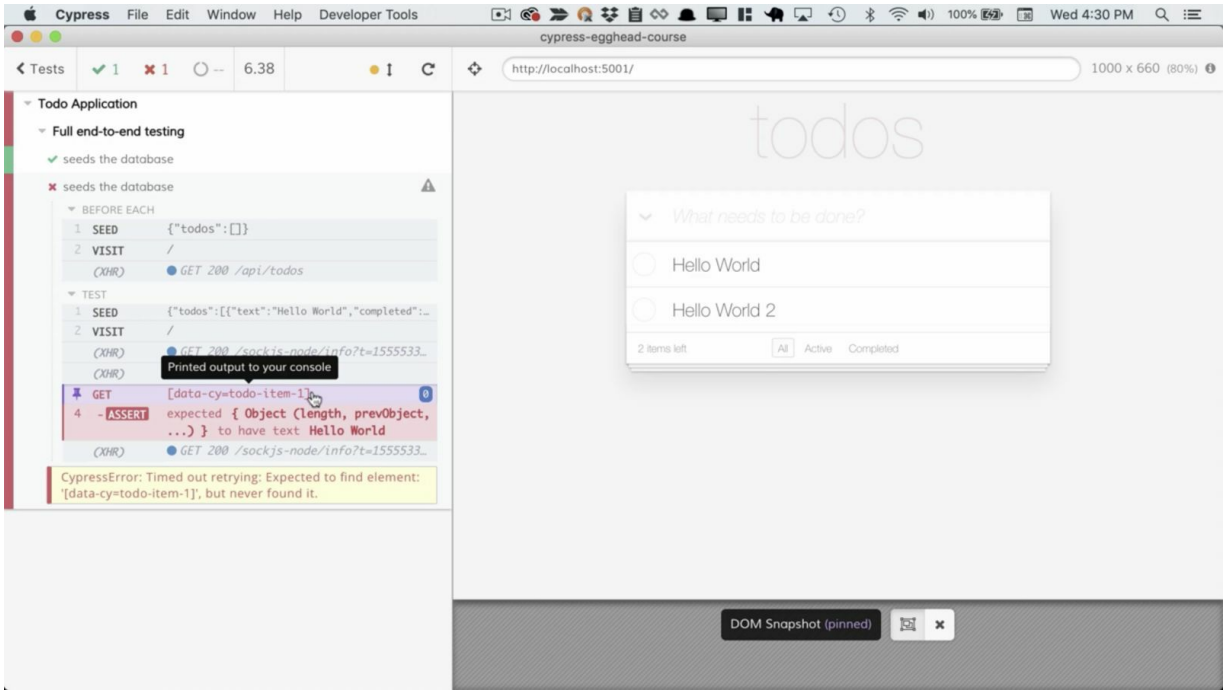
We do that actually the first time as well. We seed the empty todos array, but then in the test, we run our own seeds. That's what actually gets us the data that we want. This resets us to a totally blank state, which is great.

The next thing to think about is that if we do call two versions of this test back to back, the second one will fail.

todos.spec.js

```
it.only('seeds the database', function() P{
    cy.seed({todos: [{}, {text: 'Hello World
2'}]})
    cy.visit('/')

    cy.get('[data-cy=todo-item-1]')
      .should('have.text', 'Hello World')
      .should('not.have.class', 'completed')
      .find('.toggle')
      .should('not.be.checked')

    cy.get('[data-cy=todo-item-1]')
      .should('have.text', 'Hello World 2')
      .should('not.have.class', 'completed')
      .find('.toggle')
      .should('not.be.checked')
    ...

})
```

The reason for that is because we actually keep the generator function running. Instead of this being ID number one, it ends up being ID number three.



We don't want that because we want each of these tests to be totally isolated from each other. It shouldn't matter whether both of these tests run one after another. We want them to be independent. What we have to do for that is to reset our `generator`.

Let's go back up here. We can create a function called `resetGenerators`. In here, just go ahead and paste this reset.

commands.js

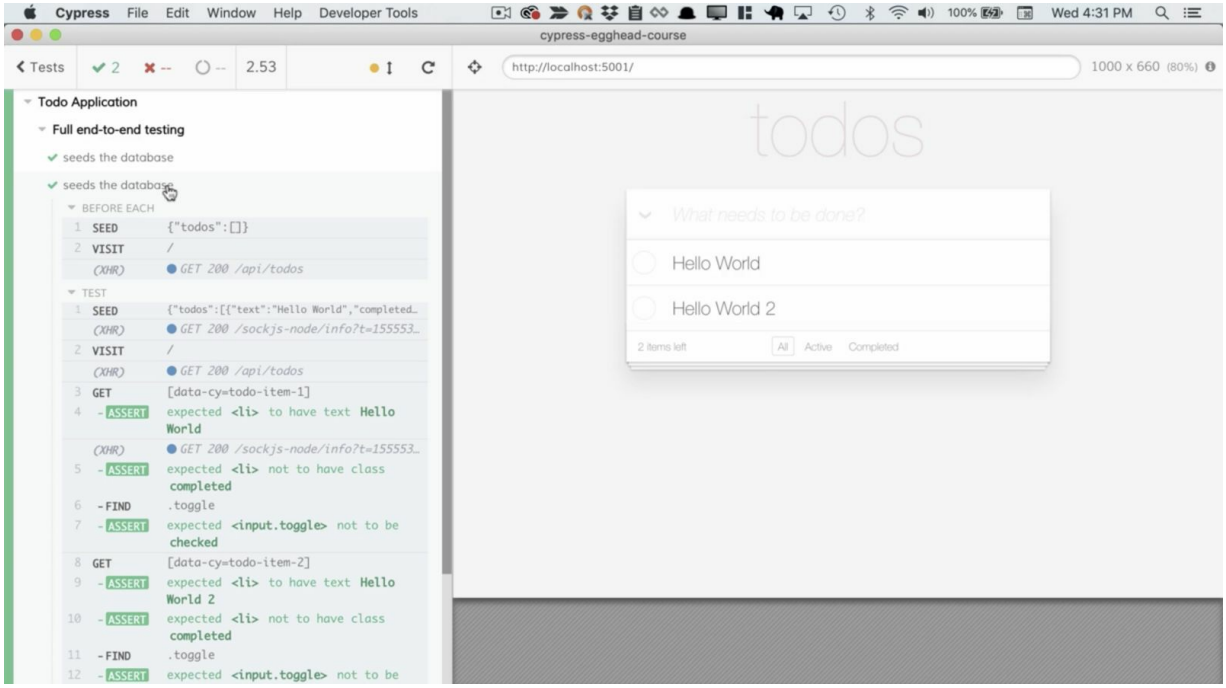
```
const _ require('lodash')
const factories =
require('../factories/factories.js')
let generators = {}

function *gen() {
    let id = 0;

    while (true) {
        yield id += 1;

        if(id> 10000) { id = 0; }
    }
}

function resetGenerators() {
    _.each(factories, (factory, key) => {
generators[key] = gen() })
}
```

Then in our `beforeEach`, we'll just make sure that we also call `resetGenerators`.

commands.js

```
const _ require('lodash')
const factories =
require('../factories/factories.js')
let generators = {}

...

beforeEach(function() {
    cy.seed({todos: []})
    resetGenerators();
})
```

We'll reset those generators between each test. We have true isolation, a much more robust, and production-ready database seeding tool.



Assert on Database Snapshots in Cypress

One of the layers of this stack that we still haven't seen how to assert against is the database. We've seen how to test against the UI, how to assert on the front-end stores, but it's important for us to test against all layers of the stack. To illustrate why, let's see an example.

Let's say that we go ahead and create a new todo. We've seen this code before. What I'm suggesting we want to do is that before this, we want to say that the `// database is empty`. Afterwards, we want to assert that the `// database contains 1 todo, todo contains: 3rd Todo`.

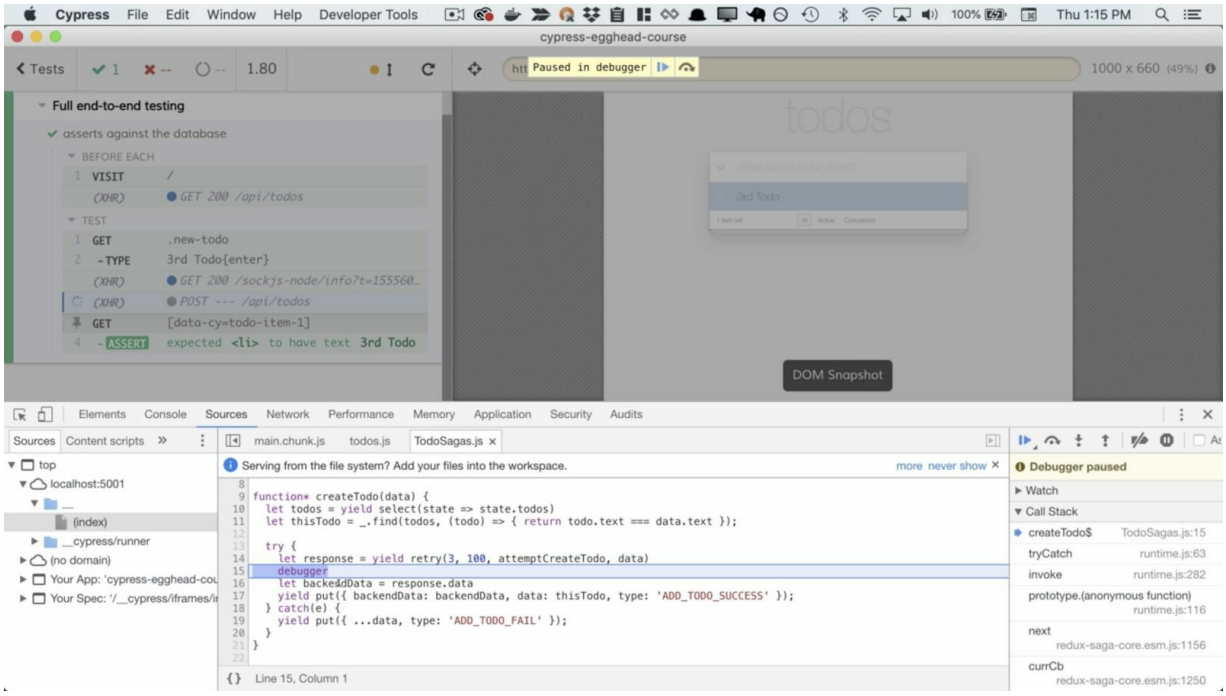
todos.spec.js

```
it.only('asserts against the database', function
(){
    // database is empty
    cy.get('.new-todo').type('3rd Todo{enter}')
    // database contains 1 todo, todo contains
3rd Todo
})
```

Why is it not enough simply for us to assert that the todo shows up on the UI? Well, let's go ahead and paste in that assertion.

```
it.only('asserts against the database', function
(){
    // database is empty
    cy.get('.new-todo').type('3rd Todo{enter}')
    // database contains 1 todo, todo contains
3rd Todo

    cy.get('[data-cy=todo-item-1]')
        .should('have.text', '3rd Todo')
})
```

We'll reopen Cypress and see that the assertion passes, even though we haven't yet gotten the response back from the back end.



The reason for that is because I've inserted an artificial delay in the server process to further illustrate this point.

The point is that we commonly show the content as created on the UI while the network request is still in flight. We do that so that the UI doesn't feel sluggish, but anything could happen. The

network could time out. Maybe the request is malformed. Maybe even the API says that the todo was created and it hasn't actually been inserted into the database.

The point is that our assertion here can give us a false sense of security, even though we haven't tested the entire contract. It's part of our contract for the data to be in the database, to be persisted, or else we're going to have a fundamentally broken experience.

Let's go ahead and make it possible to assert on our database.

We've seen how to write Cypress code in the past that interacts with our back end. We write a `cy.task`. We can call it `db:snapshot`. Specifically, we're going to look for the todos on the snapshot. We can say that it should `.should('be.empty')` at first. Then what we want to do after we know that we've created it is say that it `.should('deep.equal'`. We'll pass in the array and just use these keys that we created, so `3rd Todo` and `completed` is `false`.

```
it.only('asserts against the database', function
(){
    cy.task('db:snapshot',
'todos').should('be.empty')

    cy.get('.new-todo').type('3rd Todo{enter}')

    cy.get('[data-cy=todo-item-1]')
        .should('have.text', '3rd Todo')

    cy.task('db:snapshot',
'todos').should('deep.equal'), [
        {
            id: 1,
            text: '3rd Todo'
            completed: false
        }
    ]

})
```

Let's go ahead and pop into our plugins file again. We will add a `db:snapshot` method. This takes the name of the `table` that we want to get, which in our case was todos, so overturn `db.snapshot`, passing in the `table`, remembering that db is the library that we created to seed our database.

index.js

```
'db:snapshot': function(table) {
    return db.snapshot(table)
}
```

Let's go ahead and open that up again and add a `snapshot` method. In here, we just add snapshot. This takes the name of the table that we want to pass the snapshot of, remembering that all of this code is fairly specific to our specific database.

db-seeder.js

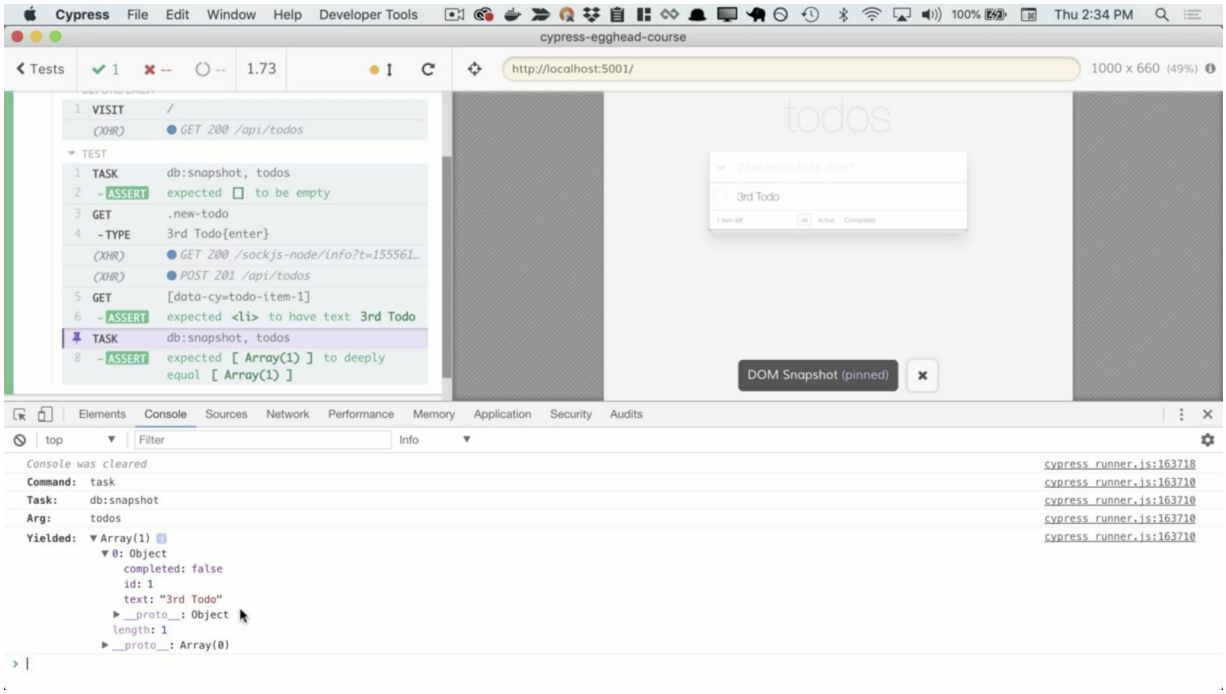
```
snapshot: function(table) {
    let adapter = new FileSync('db.test.json')
    let db = low(adapter)
}
```

You have MySQL, Postgres, RDS, whatever you have. Just use your own version of this code. This is how to do the integration so it'll just `return db.get`, passing in the `table` name, because that's how we do it in lowdb.

```
snapshot: function(table) {
    let adapter = new FileSync('db.test.json')
    let db = low(adapter)
    return db.get(table)
}
```

We'll head back to our test.

Let's go run it in Cypress. Here, we can see the task that ran. This is the snapshot. We can actually see the database snapshot with that one todo. Then we can see the assertion that ran on that object.



[3:41] Great. Now we have a lightweight way to test a new layer of the stack. In the next lesson, we'll keep expanding upon this by testing our XHR requests.

Assert on XHR Requests in Cypress

In this lesson, let's learn how to assert on XHR request by starting in `todos.spec.js`. It starts with the `cy.server`. Typically, when we've used this in the past, it's because we were going to mark our backend request. In this case, we're not marking any request, but we do have to start the cy.server in order to use `cy.route` which will allow us to spy on and assert on the XHR request.

Also, when we view `cy.route` in the past, it's been because we are returning response, but in this case, we're just not going to specify response. It's going to still hit the backend as it normally would. We will use an alias as we normally do.

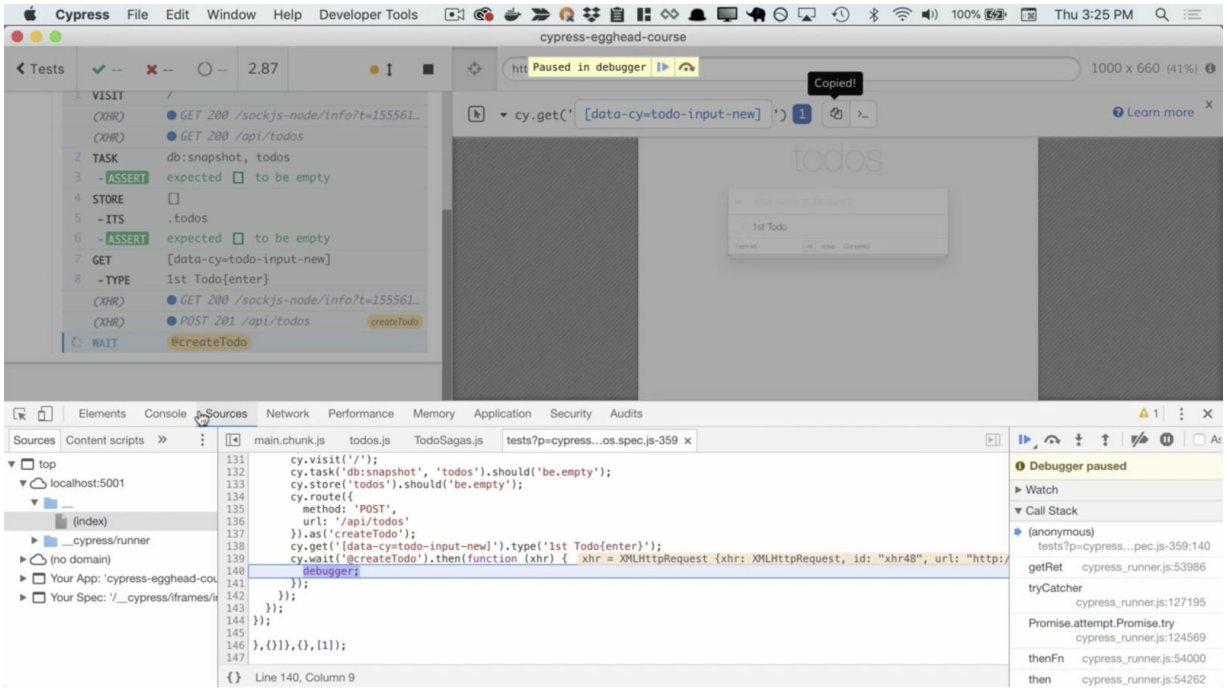
todos.spec.js

```
it.only("tests XHR requests", function() {
    cy.server()
    cy.visit('/')

    cy.task('db:snapshot',
'todos').should('be.empty')
    cy.store('todos').should('be.empty')

    cy.route({
        method: 'POST',
        url: '/api.todos'
    })
})
```

This will be our create todo request. We need to get our new todo input. We'll type into it `1st Todo`, and we'll press enter. We'll do `cy.wait`, we'll wait for that `('@createTodo')` request which will hit the real backend. Then we'll receive our `XHR` response, and we can make assertions on it here. Let's press `debugger` so we can take a look at what that looks like.

```
it.only("tests XHR requests", function() {
    cy.server()
    cy.visit('/')

    cy.task('db:snapshot',
'todos').should('be.empty')
    cy.store('todos').should('be.empty')

    cy.route({
        method: 'POST',
        url: '/api.todos'
    }).as('createTodo')

    cy.get('[data-cy=todo-input-new]').type('1st
Todo{enter}')

    cy.wait('@createTodo').then((xhr) => {
        debugger
    })
})
```

Here we see it. XHR is our subject.

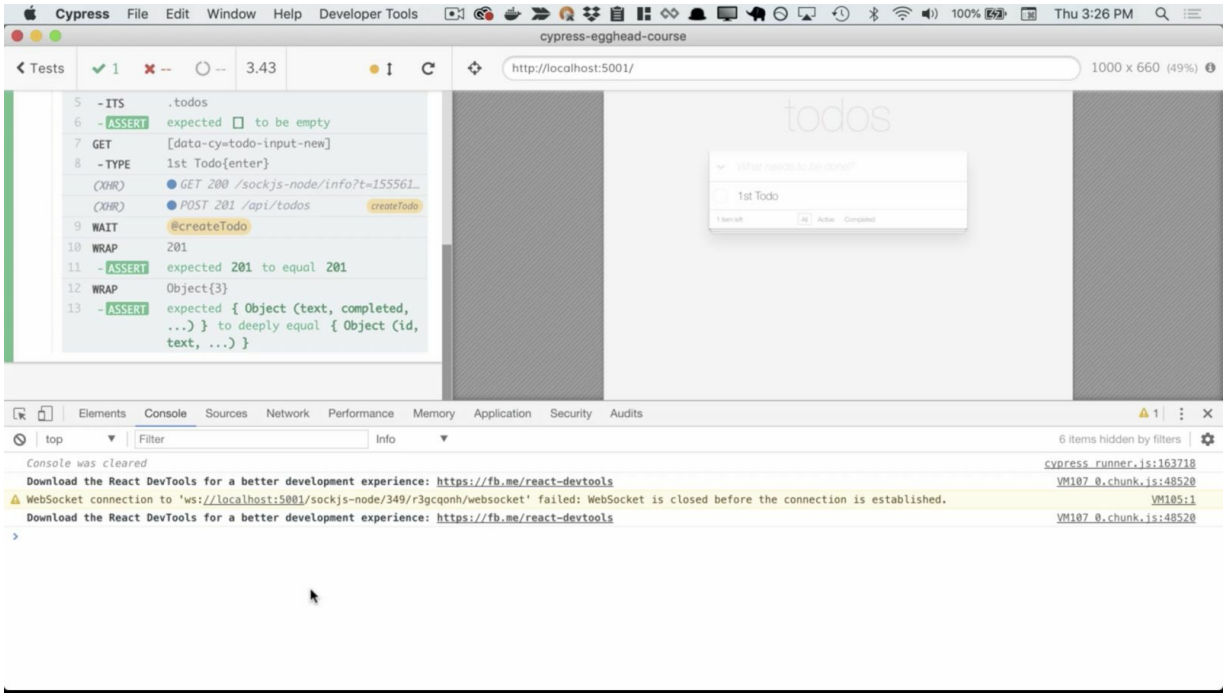


We see that there is a request object, which has a body and its headers, and the response object which has body and headers.

What we can do here, we can say `cy.wrap(xhr.response.body)`, and then we can chain that as we normally would. We can say it should `deep.equal` and then we'll give it `1`, `text`. `{id: 1, text: '1st Todo', completed: false})`, that kind of thing, we can do the same thing for our `request` if we're interested or we could do it for our `status` code and say that the `(xhr.status).should('equal', 201)`, which will be our created code.

```
it.only("tests XHR requests", function() {
    cy.server()
    cy.visit('/')

    cy.task('db:snapshot',
'todos').should('be.empty')
    cy.store('todos').should('be.empty')

    cy.route({
        method: 'POST',
        url: '/api.todos'
    }).as('createTodo')

    cy.get('[data-cy=todo-input-new]').type('1st
Todo{enter}')

    cy.wait('@createTodo').then((xhr) => {
        cy.wrap(xhr.status).should('equal', 201)

cy.wrap(xhr.response.body).should('deep.equal',
{id: 1, text: '1st Todo', completed: false})
    })
})
```

We can make both of those assertions and rerun this. That's it. That's how you assert on the XHR request.



Full End-To-End Testing in Cypress

In this lesson, we're finally going to bring all of the pieces of the puzzle together that we've been building in order to do a full end-to-end test.

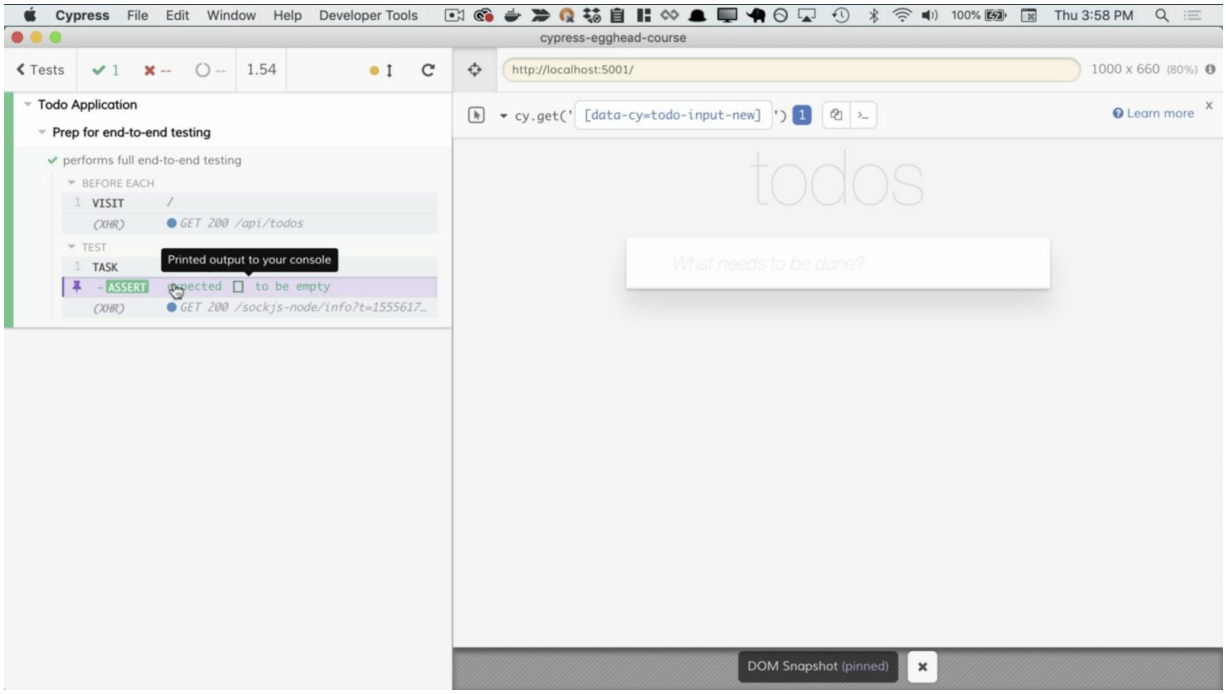
This type of test is guaranteed to work in production and will test all levels of the stack -- the database, the API responses, XHR requests, frontend stores, UI -- all of it.

To get started, we'll add a `cy.task` for our `db:snapshot`. We'll take a snapshot of our `todos` in the backend. Basically, we are just asserting that we have an empty database.

todos.spec.js

```
it('performs full end-to-end testing') {
    cy.task('db:snapshot',
'todos').should('be.empty')
}
```

This is a clean slate that we should be working from and this should be true for all of our tests. We'll quickly pop open Cypress and see that our snapshot is indeed empty, so we're ready to get working.



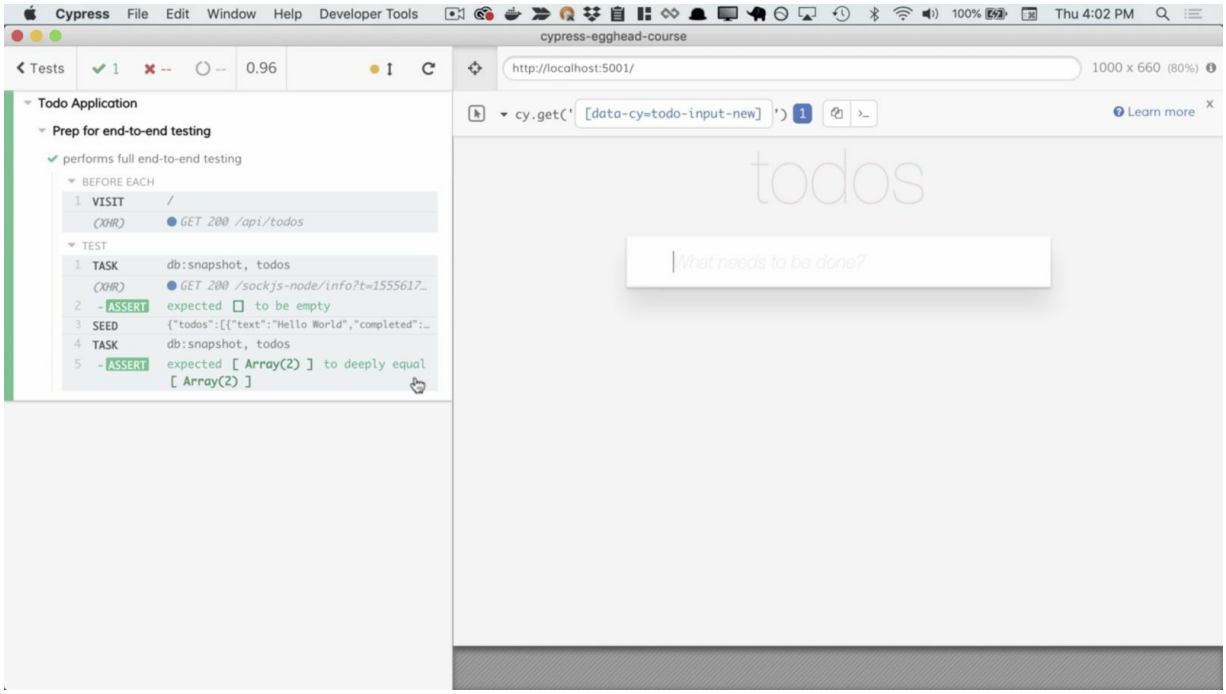
We always start our test with `cy.server` that way we can spy on our XHR request. Then we'll run a `cy.seed` so that we can get our backend into a predictable opening state. We'll say we have our todos.

The first todo can have the text, `Hello World`, which is the default. The second one can have the text, `Goodnight Moon`. We can say that it is `completed` so we can override that default.

Then we can take a `db:snapshot` and assert that we are in the working state that we would expect to be in at this point, so it should `.should('deep.equal')`.

```
it.only('performs full end-to-end testing')
function () {
    cy.task('db:snapshot',
'todos').should('be.empty')

    cy.server()

    cy.seed({todos: [{}, {text: 'Goodnight
Moon', completed: true}]})

    cy.task('db:snapshot',
'todos').should('deep.equal', [
    {
        "id": 1,
        "text": "Hello World",
        "completed": false
    },
    {
        "id": 2,
        "text": "Goodnight moon",
        "completed": true
    }
    ])
}
```

We'll pass in an array. We've seen these todos before, so I'm just copying them in, although we have different capitalization here this time. Let's save that out, take a look at our test. Everything is working as expected.



We're ready to start visiting our page and asserting on each layer of the stack in order. We'll add a `cy.route` for our main page. This will be our preload API todos `.as('preload')`. We will `cy.visit('/')`. Then we will run a `cy.wait('@preload')`.

```
...

    cy.task('db:snapshot',
'todos').should('deep.equal', [
    {
        "id": 1,
        "text": "Hello World",
        "completed": false
    },
    {
        "id": 2,
        "text": "Goodnight moon",
        "completed": true
    }
    ])

    cy.route('GET', '/api/todos').as('preload')

    cy.visit('/')

    cy.wait('@preload')
}
```

Before we get the preload, we want to assert that there is no data in the store because we haven't actually loaded the preload yet. We can do `cy.store` on our `todos`, and we can say that it should be empty. After we wait for the `preload`, we can say the `store` on the `todos` should deep equal. This is the same snapshot up here, so I'm actually just going to make a `const` with our `todos`. There we go.

```
```

```
it.only('performs full end-to-end testing')
function () {

    const todos = [{
        "id": 1,
        "text": "Hello World",
        "completed": false
    },
    {
        "id": 2,
        "text": "Goodnight moon",
        "completed": true
    }]
    cy.task('db:snapshot',
'todos').should('be.empty')

    cy.server()

    cy.seed({todos: [{}, {text: 'Goodnight
Moon', completed: true}]})

    cy.task('db:snapshot',
'todos').should('deep.equal', [
    {
        "id": 1,
        "text": "Hello World",
        "completed": false
    },
    {
        "id": 2,
        "text": "Goodnight moon",
        "completed": true
    }
    ])

    cy.route('GET', '/api/todos').as('preload')

    cy.visit('/')
```

```
    cy.store('todos').should('be.empty')

    cy.wait('@preload')

    cy.store('todos').should('deep.equal')
}
```

Let's say that this is our `todos`, this is our `todos` here. You can see that we're stepping through the stack layer by layer. The database we know has these todos in it, we go visit the page.

```
...

    cy.task('db:snapshot',
'todos').should('deep.equal', todos)

    cy.route('GET', '/api/todos').as('preload')

    cy.visit('/')

    cy.store('todos').should('be.empty')

    cy.wait('@preload')

    cy.store('todos').should('deep.equal',
todos)
}
```

Before we've gotten the data from the backend, our frontend store is empty. We populate the frontend store with the data from the backend, then we can start making assertions on the UI. We can say `('data-cy=todo-list')`. Then we can say `.children().should('have.length', 2)`.

```
...

    cy.task('db:snapshot',
'todos').should('deep.equal', todos)

    cy.route('GET', '/api/todos').as('preload')

    cy.visit('/')

    cy.store('todos').should('be.empty')

    cy.wait('@preload')

    cy.store('todos').should('deep.equal',
todos)

    cy.get('data-cy=todo-
list').children().should('have.length', 2)
}
```

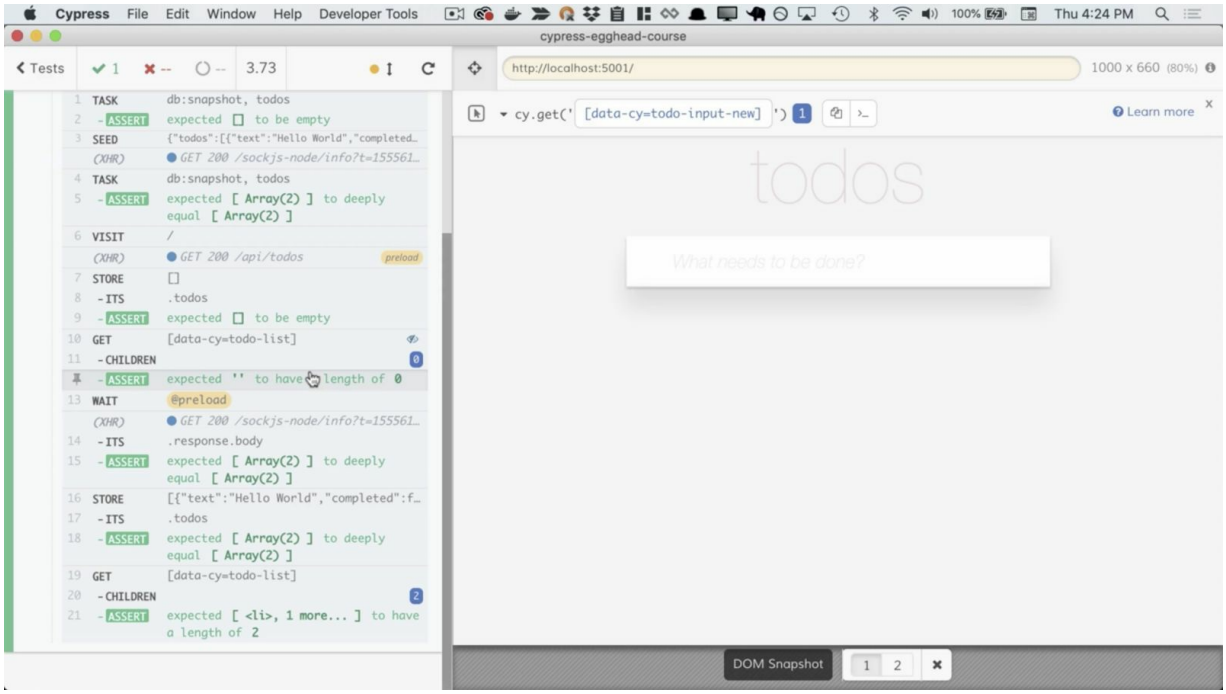
We can make sure that before we wait for the preload, we also have length `zero`. We're building this up bit by bit here.

```
...

    cy.task('db:snapshot',
'todos').should('deep.equal', todos)

    cy.route('GET', '/api/todos').as('preload')

    cy.visit('/')

    cy.store('todos').should('be.empty')
    cy.get('data-cy=todo-
list').children().should('have.length', 0)


    cy.wait('@preload')

    cy.store('todos').should('deep.equal',
todos)

    cy.get('data-cy=todo-
list').children().should('have.length', 2)
}
```

The next layer that I would test here would be RXHR request. We could say `its('response.body').should("deep.equal")`. It's going to deep equal these `todos` because here's where we're passing the todos from our database through to our frontend.

```
...

    cy.task('db:snapshot',
'todos').should('deep.equal', todos)

    cy.route('GET', '/api/todos').as('preload')

    cy.visit('/')

    cy.store('todos').should('be.empty')
    cy.get('data-cy=todo-
list').children().should('have.length', 0)



cy.wait('@preload').its('response.body').should(
"deep.equal", todos)

    cy.store('todos').should('deep.equal',
todos)

    cy.get('data-cy=todo-
list').children().should('have.length', 2)
}
```

We see this all happening in order. If we looked at our test, and we saw any location where the data flow broke down, we would be able to pinpoint it.



We wouldn't have to wonder at the end here why aren't the todos painted on the UI. We wouldn't have some head-scratching error code here that doesn't make any sense because we had some breakdown in the store.

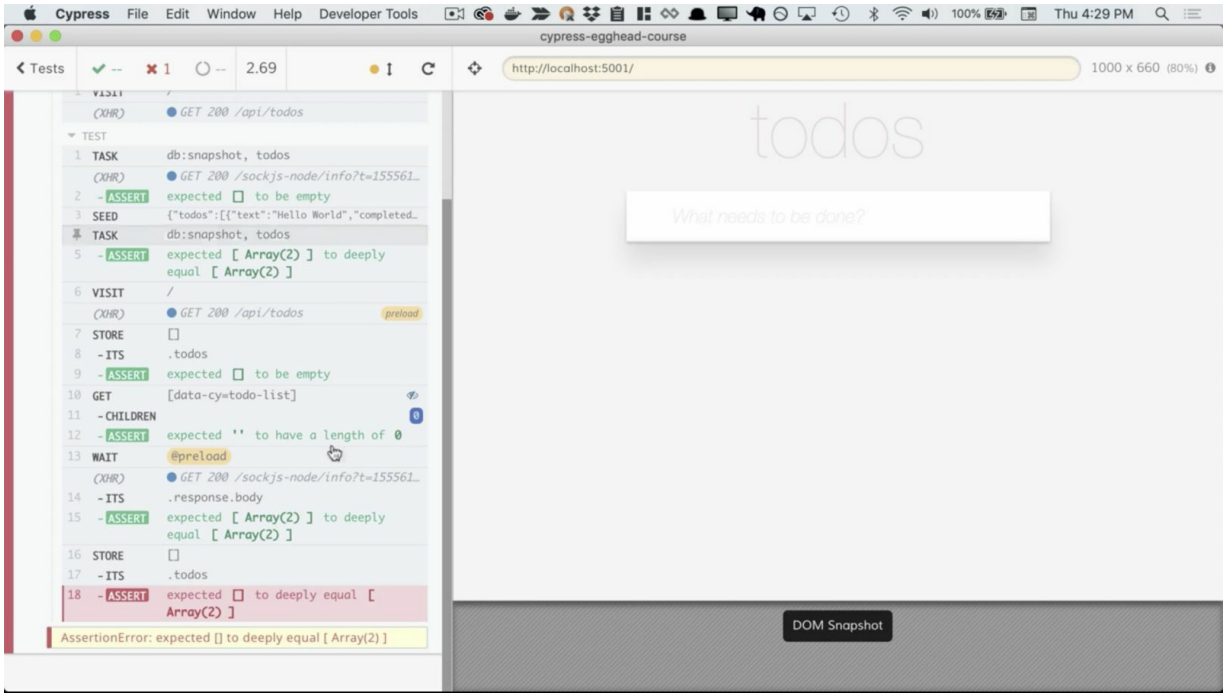
If we don't see the data in the store, we will say, hmm, I guess something broke down in my reducer. Let's go take a look at that. In fact, let's actually just look at an example of what that would look like so we can see how valuable this is.

Let's go to our reducer. Here's our `todos` loaded method where we actually get the todos from the backend in the preload. Let's just say that we returned the previous state so we didn't actually add the todos now to our page.

todos.js

```
export default function todos(state =
initialState, action) {
    switch (action.type) {
        case TODOS_LOADED:
        //return action.todos;
        return state
    }
}
```

Let's go back to our Cypress test. We'll run it, and we'll see that it's actually the store assertion that breaks down.



Because we do wait, we do see the right response from the API, but we don't see it load into the store. We don't even make it to the UI layer because we're not expecting something that hasn't made it into the store to end up on our UI.

This kind of end-to-end testing that emphasizes the flow of data through the application also has the benefit of making it a lot easier to onboard new team members because there's no

question of where the data is going in the application or which parts of the application are touched by each individual feature.