Use dom-testing-library to test any JS framework

Transcripts for Kent C. Dodds (https://egghead.io/instructors/kentcdodds) course on egghead.io (https://egghead.io/courses/use-dom-testing-library-to-test-any-js-framework).

Description

The person using your application components shouldn't have to know or care what framework(s) you used to write your application. Guess what: Neither should your tests.

This course explores the dom-testing-library (https://github.com/kentcdodds/dom-testing-library) using 11 different frameworks, from React (https://reactjs.org/) to Svelte (https://svelte.technology/). You'll get hands-on experience writing tests for any JavaScript framework, giving you the confidence you need to ship your JavaScript application with your framework of choice.

Use dom-testing-library with React

Kent C Dodds: [00:00] We'll start by making a function called `render` that's going to accept our `ui` right here. We'll say,`ui` Then we need to make a `container` that's a `div`. That's what we're going to `render` using ReactDOM.

react.test.js

```
function render(ui) {
  const container =
document.createElement('div')
}
```

[00:16] We'll `import ReactDOM from 'react-dom'`. We'll say, `ReactDOM.render(ui, container)`. Then we'll `return getQueriesForElement`. We use that `getQueriesForElement` in this return on that `container`.

react.test.js

```
import {getQueriesForElement, fireEvent} from
'dom-testing-library'

function render(ui) {
  const container =
document.createElement('div')
  ReactDOM.render(ui, container)
  return getQueriesForElement(container)
}
```

[00:35] If we pop open here, we're going to see that our test is actually failing. That's because in React, all the DOM event listeners for listening to click events are actually bound to `document.body` with React.

[00:47] We need this `container` to be inside the body. We'll say, `document.body.appendChild(container)`. That'll get our test passing.

react.test.js

```
import {getQueriesForElement, fireEvent} from
'dom-testing-library'

function render(ui) {
  const container =
document.createElement('div')
  ReactDOM.render(ui, container)
  document.body.appendChild(container)
  return getQueriesForElement(container)
}
```

[00:55] There are a couple other optimizations here. I'm going to spread those utilities. I'm also going to pass along the `container`. I'm also going to pass along a `cleanup` method. That is going to say, `document.body.removeChild(container)`, so we can clean up that `container`.

[01:11] We also want to `unmount` the React component at the `container`. We'll say, `ReactDOM.unmountComponentAtNode(container)`. We'll `unmount` it. Then we'll remove it from the `container`.

react.test.js

```
function render(ui) {
  const container =
document.createElement('div')
  ReactDOM.render(ui, container)
  document.body.appendChild(container)
  return {
    ...getQueriesForElement(container),
    container,
    cleanup() {
      document.body.removeChild(container)
    }
  }
}
```

[01:22] In our `test` here, we can also add `cleanup` and `cleanup` at the end. If we look at `document.body.outerHTML`, we'll see that the body is now empty. That way, any other tests that are using this `render` method won't have a `document.body` that's all messed up with previous test components.

react.test.js

```
test('renders a counter', () => {
  const {getByText, cleanup} = render(<Counter
/>)
  const counter = getByText('0')
  fireEvent.click(counter)
  expect(counter).toHaveTextContent('1')
  fireEvent.click(counter)
  expect(counter).toHaveTextContent('2')
  cleanup()
  console.log(document.body.outerHTML)
})
```

[01:41] In review, for React, for the `render` method, we create a `container` that's a `div`. We `render` the `ui` to that `container`. We append that `container` to the body. Then we get all the utilities for that `container`.

[01:52] We also return the `container` just in case they want to do a query selector. Then we run this `cleanup` so that we can unmount the component and remove that `container` child.

Use dom-testing-library with Preact

Kent C Dodds: [00:00] Let's make our `render` function that'll accept some `ui`, and then here we're going to need a `const container = document.createElement('div')`. We're going to `render` our `ui` into that container using Preact, so we'll say `Preact.render(ui, container)`, and then we'll `return`.

[00:18] We need to `import getQueriesForElement from 'dom-testing-library'`, and then we'll `return getQueriesForElement(container)`. And with that, our tests

are actually failing, and the reason that they're failing is Preact does not `render` synchronously like React does.

preact.test.js

```
function rendex(ui) {
  const container =
document.createElement('div')
  Preact.render(ui, container)
  return getQueriesForElement(container)
}
```

[00:33] Preact will actually wait until the next tick of the event loop to go ahead and re-render. Our `counter` doesn't get updated until the next tick of the event loop, so `dom-testing-library` has the `wait` utility that we can use for this case.

[00:47] We're going to turn this into an `async` test, and after we fire that click, we're going to say `await wait()`, and that effectively just waits till the next tick of the event loop, and then we can go ahead and check if the `counter` has the text content of one and two.

preact.test.js

```
test('renders a counter', async () => {
  const {getByText} = render(<Counter />)
  const counter = getByText('0')
  fireEvent.click(counter)
  await wait()
  expect(counter).toHaveTextContent('1')
  fireEvent.click(counter)
  await wait()
  expect(counter).toHaveTextContent('2')
})
```

[01:00] With that, our test is passing, so let's just refactor this slightly. I want to also `return` the `container` as well as the `getQueriesForElement`, just in case people want to operate directly on the `container`. Then having to do this `wait` after every single time we call `fireEvent` is kind of annoying.

[01:17] What I did was I created this `fireEventAsync` which will bring in `fireEvent` and `wait` from `dom-testing-library`. And then it iterates through all the methods on `fireEvent`, and adds those two `fireEventAsync`. Effectively, it just takes the original `fireEvent` function, and makes it `async` and calls await.

fire-event-async.js

```
import {fireEvent, wait} from 'dom-testing-
library'

const fireEventAsync = {}

Object.entries(fireEvent).reduce((obj, [key,
val]) => {
  obj[key] = async (...args) => {
    const ret = val(...args)
    await wait()
    return ret
  }
  return obj
}, fireEventAsync)

export {fireEventAsync}
```

[01:35] It's effectively doing the same thing that we're doing right here except it does it all on one line. Instead of `fireEvent` from `dom-testing-library`, I'm going to `import {fireEventAsync} from './fire-event-async'`. Then instead of using `fireEvent` and then `wait`, we can just do `fireEventAsync`, `await` that and remove `await` there.

[01:57] We get effectively the same thing without having to add that `wait` manually ourselves. In review, all we had to do for Preact is we create a `container` to `render` our `ui` to, and then we `return getQueriesForElement` and that `container`, and then we can have our regular test.

preact.test.js

```
function render(ui) {
  const container =
document.createElement('div')
  Preact.render(ui, container)
  return {
    container,
    ...getQueriesForElement(container),
  }
}

test('renders a counter', async () => {
  const {getByText} = render(<Counter />)
  const counter = getByText('0')
  await fireEventAsync.click(counter)
  expect(counter).toHaveTextContent('1')
  await fireEventAsync.click(counter)
  expect(counter).toHaveTextContent('2')
}
```

Use dom-testing-library with jQuery

Kent C Dodds: [00:00] JQuery is actually pretty simple, so we're not even going to make a `render` method. What we're going to do is I'm going to use this jQuery plugin on an element that I create.

jquery.test.js

```
$.fn.countify = function countify() {
  this.html('
    <div>
      <button>0</button>
    </div>
  ')
  const $button = this.find('button')
  $button._count = 0
  $button.click(() => {
    $button._count++
    $button.text($button._count)
  })
}
```

[00:08] I'm going to say, `const div = document.createElement("div")` Then I'm going to turn this into a jQuery object. We'll call `countify` on that. Then we'll get our `getByText` from `getQueriesForElement` on that `div`. That gets my test passing.

[00:25] With jQuery, it's a little bit unique. We're treating this as a jQuery plugin. We create an element. We apply our jQuery plugin. Then we get our queries for that element. It wouldn't make a whole lot of sense to create a `render` method for this.

jquery.test.js

```
test('counter increments', () => {
  const div = document.createElement('div')
  $(div).countify()
  const {getByText} = getQueriesForElement(div)
  const counter = getByText('0')
  fireEvent.click(counter)
  expect(counter).toHaveTextContent('1')

  fireEvent.click(counter)
  expect(counter).toHaveTextContent('2')
})
```

Use dom-testing-library with Dojo

Kent C Dodds: [00:00] Let's go ahead. We'll start by making a `render` function for rendering a Dojo `WidgetBase` component that's going to accept our `ui`. The type is going to be a `Constructor` of `WidgetBase`. It's a `Constructor` of something that extends the `WidgetBase`.

dojo.test.js

```
class Counter extends WidgetBase {
  count = 0
  increment() {
    this.count++
    this.invalidate()
  }
  render() {
    return v('div', [v('button', {onclick:
this.increment}, [`${this.count}`])])
  }
}
```

[00:16] Then we'll go ahead and make our `container = document.createElement("div")`. Then we'll make our `Projector` as a `ProjectorMixin` of that `ui`. Then we'll make our `projector` instance, `new Projector()`.

[00:31] We'll set this `projector` instance `async` to be `false` to make our tests a little bit easier to work with and `projector.append` to that `container`. Then we can `return` our `getQueriesForElement` on that `container`. Our tests are passing.

dojo.test.js

```
function render(ui: Constructor<WidgetBase>) {
  const container =
document.createElement('div')
  const Projector = ProjectorMixin(ui)
  const projector = new Projector()
  projector.async = false
  projector.append(container)
  return getQueriesForElement(container)
}
```

[00:47] Let's also add the `container`. We'll spread across those `getQueriesForElement` so that people can access the `container` if they need. That gets things working.

dojo.test.js

```
function render(ui: Constructor<WidgetBase>) {
  const container =
document.createElement('div')
  const Projector = ProjectorMixin(ui)
  const projector = new Projector()
  projector.async = false
  projector.append(container)
  return {
    container,
    ...getQueriesForElement(container)
  }
}
```

[00:56] In review, we've got this `render` method that creates a `container` and then uses `ProjectorMixin` from the Dojo framework and passes in our `ui`, which is a `Constructor` based

on the `WidgetBase`.

[01:09] Then we create our instance of the `projector`. We make `projector.async` as `false` to make testing a little bit easier. Then we `append` the `projector` to that `container`. Then we return `getQueriesForElement` so that we can use it in our typical `dom-testing-library` test.

Use dom-testing-library with HyperApp

Kent C Dodds: [00:00] For `hyperapp`, we're going to make a function called `render`. This one's going to accept an object as the first argument, and it's going to accept `state`, `view`, `actions`, and then we'll make our `container`, `document.createElement("div")`.

hyperapp.test.js

```
function render({state, view, actions}) {
  const container =
document.createElement('div')
}
```

[00:15] Then we're going to use `hyperapp.app`, and we'll pass `state`, `actions`, `view`, and `container`. Then we'll go ahead and `return getQueriesForElement`, and we'll pass `container` to that. Now, `hyperapp` actually will `render` everything asynchronously, and all of our events that we're firing are asynchronous, meaning it's going to wait until the next tick of the event loop before it continues on for the rest of the test.

hyperapp.test.js

```
function render({state, view, actions}) {
  const container =
document.createElement('div')
  hyperapp.app(state, actions, view, container)
  return getQueriesForElement(container)
}
```

[00:41] This is because `hyperapp` is implemented to update the app on the next tick of the event loop any time a `state` update happens. That applies also when you initially `render`. We need to actually turn this into an `async` function, which we can `await` the `wait` utility from `dom-testing-library`.

hyperapp.test.js

```
function render({state, view, actions}) {
  const container =
document.createElement('div')
  hyperapp.app(state, actions, view, container)
  return getQueriesForElement(container)
}
```

[00:57] With that, our tests are now passing.

hyperapp.test.js

```
test('renders a counter', async () => {
  const {getByText, getByTestId} = await
render({state, view, actions})
  const counter = getByText('0')
  await fireEventAsync.click(counter)
  expect(counter).toHaveTextContent('1')
  await fireEventAsync.click(counter)
  expect(counter).toHaveTextContent('2')
}
```

In review, for `hyperapp`, we use `async`, and we use our `fireEventAsync` utility here, which effectively is `fireEvent` with an `await` inside of each one of those events.

fire-event-async.js

```
const fireEventAsync = {}

Object.entries(fireEvent).reduce((obj, [key,
val]) => {
  obj[key] = async (...args) => {
    const ret = val(...args)
    await wait()
    return ret
  }
  return obj
}, fireEventAsync)
```

[01:11] Then it waits to the next tick of the event loop. Then we create our `container`. We start `hyperapp` with the `state`, `actions`, and `view`, and then we wait for the next tick of the event loop before returning the `getQueriesForElement`.

[01:24] Then when people use our `render` method, they're going to `await` it to make sure that the DOM is ready for us to start making our queries with `dom-testing-library`.

Use dom-testing-library with AngularJS

Kent C Dodds: [00:00] Let's go ahead and make a function called `render`. For our `render` method here with AngularJS, we're going to accept a template and an options object for the modules. We'll just call that `html` and `config`.

angularjs.test.js

```
function render(html, config) {

}

test('renders a counter', () => {
  const {getByText} = render(`<my-counter></my-
counter>`, {modules: ['myApp']})
  const counter = getByText('0')
  fireEvent.click(counter)
  expect(counter).toHaveTextContent('1')

  fireEvent.click(counter)
  expect(counter).toHaveTextContent('2')
})
```

[00:12] Let's go ahead and make our `container = document.createElement("div")`. Then we'll set the `container.innerHTML` to be that `html`.

[00:22] We'll say `angular.bootstrap(container, config.modules)`. Then we can `return` this `getQueriesForElement` on that `container`. That'll get our test passing.

[00:34] Let's go ahead and add, as a convenience, we'll spread the `getQueriesForElement` and that `container` just in case we want to make queries directly on that `container`.

angularjs.test.js

```
function render(html, config) {
  const container =
document.createElement('div')
  container.innerHTML = html
  angular.bootstrap(container, config.modules)
  return {
    container,
    ...getQueriesForElement(container),
  }
}
```

[00:44] With that, we get our standard `dom-testing-library` test working by passing the `html` and then the modules for our AngularJS app. Then we create a `div`, which is our `container`.

[00:54] We set the `container.innerHTML`. Then we `.bootstrap` Angular on that `container` with the specific modules that have our direct `div` defined.

Use dom-testing-library with Angular

Kent C Dodds: [00:00] For our Angular test, we're going to need `import {TestBed, ComponentFixtureAutoDetect} from '@angular/core/testing'`. Let's go ahead and create our function for rendering a `component` of `any` type here.

[00:13] We'll use `TestBed.configureTestingModule`. We'll pass in `declaration: [component]`, and `providers` is this array with the object of `provide: ComponentFixtureAutoDetect`, and `useValue: true`. Then we'll `compileComponents`.

angular.test.ts

```
function render(component: any ) {
  TestBed.configureTestingModule({
    declarations: [component],
    providers: [{provide:
ComponentFixtureAutoDetect, useValue: true}]
  }).compileComponents()

}
```

[00:32] Then we need to get a DOM node to query. Let's make our `fixture` with `TestBed.createComponent`. We'll pass our `component`, and then we'll get our `container` from `fixture.debugElement.nativeElement`. Then we can return `getQueriesForElement(container)`, and let's go ahead and run our test.

angular.test.ts

```
function render(component: any ) {
  TestBed.configureTestingModule({
    declarations: [component],
    providers: [{provide:
ComponentFixtureAutoDetect, useValue: true}]
  }).compileComponents()

  const fixture =
TestBed.createComponent(component)
  const container =
fixture.debugElement.nativeElement

  return getQueriesForElement(container)
}
```

[00:53] Cool. That gets our test to pass. In review, for testing an Angular component with `dom-testing-library`, you can create this `render` function that will configure the testing module to have the component as the declaration and `ComponentFixtureAutoDetect` with `useValue: true` as one of the providers.

[01:09] Then we compile those components. We create a `fixture`. We get our `container` from the `nativeElement`, from the `debugElement`, from that `fixture`. Then we `getQueriesForElement`. Just for convenience, we'll go ahead and provide the `container`, and we'll spread across the `getQueriesForElement`. That allows us to write our typical DOM testing library test.

angular.test.ts

```
function render(component: any ) {
  TestBed.configureTestingModule({
    declarations: [component],
    providers: [{provide:
ComponentFixtureAutoDetect, useValue: true}]
  }).compileComponents()

  const fixture =
TestBed.createComponent(component)
  const container =
fixture.debugElement.nativeElement

  return {
    container,
    ...getQueriesForElement(container)
  }
}
```

Use dom-testing-library with VueJS

Kent C Dodds: [00:00] For Vue, let's go ahead and create a function that is a `render` function, it takes a `Component`. Then we'll make `const vm = new Vue(Component).$mount()`. We'll go ahead and `$mount` that. Then we can `return getQueriesForElement(vm.$el)`. We'll save that. Our tests are passing.

vue.test.js

```
function render(Component) {
  const vm = new Vue(Component).$mount()
  return getQueriesForElement(vm.$el)
}
```

[00:21] Vue is another one of those libraries that doesn't synchronously re-render on state changes. We're using our `fireEventAsync` utility here that effectively takes the `fireEvent` and adds `await` as part of the function.

fire-event-async.js

```
const fireEventAsync = {}

Object.entries(fireEvent).reduce((obj, [key,
val]) => {
  obj[key] = async (...args) => {
    const ret = val(...args)
    await wait()
    return ret
  }
  return obj
}, fireEventAsync)
```

[00:36] From there, we just `await fireEventAsync`. That waits for the next tick of the event loop. Then we can make our assertion about the counter being updated.

[00:45] From here, let's just go ahead and add the `container` as `vm.$el`. Then we'll spread across the `...getQueriesForElement` just as a helper here.

vue.test.js

```
function render(Component) {
  const vm = new Vue(Component).$mount()
  return {
    container: vm.$el,
    ...getQueriesForElement(vm.$el),
  }
}

test('counter increments', async () => {
  const {getByText} = render(Counter)
  const counter = getByText('0')
  await fireEventAsync.click(counter)
  expect(counter).toHaveTextContent('1')

  await fireEventAsync.click(counter)
  expect(counter).toHaveTextContent('2')
})
```

Use dom-testing-library with Mithril

Kent C Dodds: [00:01] For Mithril, let's go ahead and we'll create a function to render a Mithril `component`. This is going to create a `container`. That'll be `document.createElement('div')`, and then we'll use `m.mount` to mount to that `container`, and that `component`. Then we'll return `getQueriesForElement(container)`. We'll save that, and our tests are passing.

mithril.test.js

```
function render(component) {
  const container =
document.createElement('div')
  m.mount(container, component)
  return getQueriesForElement(container)
}
```

[00:24] There are a couple unique things about Mithril. That is that our test does have to be async, because we're going to be awaiting this `wait` call. Mithril is unique in that it doesn't update the DOM in a deterministic amount of time.

[00:36] Rather than using our `fireEventAsync` utility that we have local to our project, we're going to use the `wait` utility, and pass a callback, so that it can continuously check this assertion until it passes. Let's go ahead, and we'll also provide the `container`, just in case that's useful for people.

mithril.test.js

```
function render(component) {
  const container =
document.createElement('div')
  m.mount(container, component)
  return {
    container,
    ...getQueriesForElement(container),
  }
}

test('counter increments', async () => {
  const {getByText} = render(Counter)
  const counter = getByText('0')
  fireEvent.click(counter)
  await wait(() =>
expect(counter).toHaveTextContent('1'))

  fireEvent.click(counter)
  await wait(() =>
expect(counter).toHaveTextContent('2'))
})
```

[00:55] Now, our render method creates an element. It uses Mithril to mount to that `container` that `component` that we're passing, and then returns those queries in the `container`. From there, we can write a regular DOM testing library test.

Use dom-testing-library with Svelte

Kent C Dodds: [00:00] For Svelte, we're going to go ahead and create a function called `render`. That's going to take our `template`. Then we'll also take some `options`. Here, we're passing the `counterTemplate` and taking `data` as one of the options.

svelte.test.js

```
function render(template, options) {

}

test('counter increments', () => {
  const {getByText} = render(counterTemplate,
{data: {count: 0}})
  const counter = getByText('0')
  fireEvent.click(counter)
  expect(counter).toHaveTextContent('1')

  fireEvent.click(counter)
  expect(counter).toHaveTextContent('2')
})
```

[00:12] Then we're going to make our `container`. That is going to be `document.createElement("div")`. Then we'll create a `Constructor`. That's going to come from `svelte.create` with that `template`. We'll go ahead and initialize that `Constructor` with the target set to our `container`.

[00:31] Then we'll forward along the other `options`. In our case, that includes the `data` for our `template` to interpolate. Then we'll go ahead and return `getQueriesForElement` on that `container`. That gets our test passing.

svelte.test.js

```
function render(template, options) {
  const container =
document.createElement('div')
  const Constructor = svelte.create(template)
  new Constructor({
    target: container,
    ...options
  })
  return getQueriesForElement(container)
})
```

[00:44] With Svelte, we pass in the Svelte `template`. We pass along the `data` that we want to interpolate. Then we create a `container` element. We create a Svelte `Constructor`.

[00:54] We initialize that `Constructor` to have the `target` set to our `container` and forward along the `options` that we've been provided and then get queries for the `container` element.

[01:03] Just for fun, we'll go ahead and also add the `container` here and spread across those `getQueriesForElement`. From there, we can do our regular `dom-testing-library` test.

svelte.test.js

```
function render(template, options) {
  const container =
document.createElement('div')
  const Constructor = svelte.create(template)
  new Constructor({
    target: container,
    ...options,
  })
  return {
    container,
    ...getQueriesForElement(container),
  }
}
```

Use dom-testing-library with from-html

Kent C Dodds: [00:00] Let's go ahead and start by making this `render` method. I'm going to say `function render` That's going to take my `FromHtmlClass`. I'm going to create my `instance` of that new `FromHtmlClass`.

[00:13] Then I'm going to make my `container` a div. Then we're going to call this `mount` method on that `container`. We'll say, `instance.mount(container)` Then I'm going to `return` my `getQueriesForElement` from dom-testing-library. The element is the `container`.

from-html.test.js

```
function render(FromHtmlClass) {
  const instance = new FromHtmlClass()
  const container =
document.createElement('div')
  instance.mount(container)
  return getQueriesForElement(container)
}
```

[00:33] With that, my tests are passing. We can refactor this a little bit, make it a little more useful, spread all those, and also expose the `container` and even the `instance`, though I wouldn't recommend using that directly.

from-html.test.js

```
function render(FromHtmlClass) {
  const instance = new FromHtmlClass()
  const container =
document.createElement('div')
  instance.mount(container)
  return {
    container,
    instance,
    ...getQueriesForElement(container),
  }
}
```

[00:46] That's an implementation detail. Maybe we'll just leave `instances` out, but the `container` could be useful for certain tests.

from-html.test.js

```
function render(FromHtmlClass) {
  const instance = new FromHtmlClass()
  const container =
document.createElement('div')
  instance.mount(container)
  return {
    container,
    ...getQueriesForElement(container),
  }
}
```

[00:52] In review, for this use case of the `FromHtml` library, we can take a `FromHtmlClass`, get an `instance`, and `mount` that `instance` to a `container` div and then return the queries for that `container` element.