UI-Based Integration Testing

- Up to now, we've talked about testing in the context of library code that's not tied to any particular application. For this kind of code, we've focused on writing tests from the perspective of developers using our library.

- Often, though, we'll be working on implementing UI-based applications (e.g. web and mobile apps) rather than libraries. Writing tests for a UI-based application will necessarily look a little different than writing tests for a code library.

- One of the things we want to continue to focus on when implementing tests for a UI-based application is **making sure the tests interact with our software the way a user would**.

- Within the context of a UI-based application, the user we want to act like in our tests is the end user, who interacts with our application by looking, reading, scrolling, clicking, typing, etc. We'll need special tools to help us interact with the application like this.

- One option for implementing tests for a UI-based application that behave like the end user is to use end-to-end testing tools like Cypress to implement end-to-end tests that use a "robot" who scrolls, clicks, and types in the UI of our running application.

- End-to-end testing is an important form of testing, and we will explore it later in this course. However, as we'll see end-to-end tests can be expensive to implement and maintain and are usually reserved for testing large parts of an application, like an entire screen or user flow.

- Often, we'll want a more lightweight form of testing for our UI-based application that allows us to easily focus on validating the behavior of smaller parts of the app, like an individual form, button, or other interactive component.

- At the same time as we want a lightweight form of testing here, our tests will necessarily be larger in scope than unit tests, since they will need to validate interactions between the UI itself (i.e. the actual buttons, input fields, etc.) and the code attached to the UI that executes the behaviors associated with the user's typing, clicking, etc.

- For example, we may want to implement a test to verify that a specific dialog opens when a certain button in the UI is clicked.

- Because these tests involve interactions between different parts of our application, we will think of them as ***integration tests***.

- Below, we'll explore tools and techniques for performing this kind of testing specifically within the context of the client side of a web application (i.e. an app implemented with HTML and client-side JS), but these tools and techniques are based on principles that apply to testing any kind of UI-based application.

- We'll assume that we're working on a project with Jest already set up.

Setting up: Using JSDOM to render HTML for testing

- Our goal is to test parts of our application by interacting with its UI. To be able to do this, we'll need to somehow run the parts of the application we want to test, including its UI.

- Normally, the client side of a web application (i.e. the HTML, CSS, and client-side JS) runs within a web browser, but a browser is far too complex and bulky to use for the kind of testing we want to do here.

- Instead, we'll use a more lightweight runtime environment called JSDOM. JSDOM is a JavaScript implementation of web standards like the WHATWG DOM and HTML standards. It is designed to emulate enough of what a browser does to support testing (and scraping) web applications without including the bulky complexities of a browser.

- The most basic way to use JSDOM is to provide it with a string of HTML. JSDOM will parse and "render" that HTML into an in-memory representation called the ***DOM*** (the document object model), just like a browser would do. The DOM is simply a tree structure representing the hierarchy of elements described in the HTML.

- Unlike a browser, when JSDOM "renders" the HTML we give it, it does not draw a visual representation of it. Instead, it simply provides us with programmatic access to the in-memory DOM tree using the same mechanisms a browser would provide for interacting with the DOM.

- Let's see how this works. We'll need to start by installing JSDOM for use in the current project:

```
npm install --save-dev jsdom
```

- Once JSDOM is installed, we can create a new file in which to experiment to see how JSDOM works. Let's call this file `jsdomExperiment.js`. Inside, we'll begin by importing JSDOM:

```
const JSDOM = require("jsdom").JSDOM
```

- Once we have JSDOM imported, we can use it to render a basic HTML "page":

```
const dom = new JSDOM(
    "<!DOCTYPE html><p id='hello'>Hello world!</p>"
)
```

- Here, `dom` is an object representing the DOM rendered by JSDOM. We can print an HTML serialization of the DOM, for example, with the method `dom.serialize()`, e.g.:

```
console.log(dom.serialize())
```

- Note that this will print something like this (whitespace added for clarity):

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
        <p id="hello">Hello world!</p>
    </body>
</html>
```

- As you can see, in parsing the HTML and rendering the DOM, JSDOM adds implied `<html>`, `<head>`, and `<body>` elements, just like a browser would do.

- In addition, the `dom` object also gives us a programmatic interface into the rendered DOM it represents, just like the interface a browser would provide. The

entry point into the DOM is `dom.window`, which represents exactly what the browser would provide us in the global `window`.

- For example, we can access elements through `dom.window.document`, just like we can through `window.document` (or just `document`) in the browser, e.g.:

```
const paragraph = document.getElementById("hello")
```

- Now that we have an idea how JSDOM works, let's start to use it to set up tests for a client-side web application. To do this, we'll have to make a few small tweaks to the way we used JSDOM above.

Loading and rendering an HTML/JS application for testing

- When we're dealing with an actual client-side web application, it will typically be factored into multiple files. For example, let's say we have a very simple web application that's factored into two separate files, an HTML file and a JS file:

```
<!-- counter.html -->
<!DOCTYPE html>
<html>
    <head>
        <script src="counter.js" defer></script>
    </head>
    <body>
        <button id="counter">0</button>
    </body>
</html>

// counter.js
const counter = document.getElementById("counter")
counter._count = 0
counter.addEventListener("click", function () {
    counter._count++
    counter.textContent = counter._count
```

```
})
```

- Combined, these two files implement a very simple “counter” button, which displays an integer value that starts at 0 and increments by 1 each time the button is clicked.
    - To see how these files work, you could save them to the same directory on your machine and open `counter.html` in your browser.

- Let’s work on implementing some basic integration tests for this small application. We’ll start by creating a test file called `counter.test.js` in the same directory as `counter.html` and `counter.js`.

- We want to be able to use JSDOM easily within our tests to render our client-side applications into a testable form of DOM. Luckily, Jest is well set up to help us do this.

- In particular, Jest can be configured to use test environments that set up a specialized context in which to execute tests, and an official testing environment called `jest-environment-jsdom` is available for seamlessly incorporating JSDOM as part of our tests. To be able to use this test environment, we must install it using `npm`:

  ```
  npm install --save-dev jest-environment-jsdom
  ```

- At the top of our test file, `counter.test.js`, we can add the following `@jest-environment` “docblock” comment to indicate that the JSDOM environment should be used for this test file:

  ```
  /**
   * @jest-environment jsdom
   */
  ```

- Note that a docblock comment like this allows us to control the test environment on a file-by-file basis. If we wanted to use the same test environment across an entire project, we could use the options described in the Jest documentation for setting project-wide configurations. In this case, we would want to set the value of the `testEnvironment` configuration to `"jsdom"`.

- Using the JSDOM test environment provides us with a DOM we can interact with in our tests through global values like `document` and `window`, much like we'd interact with the DOM in a browser.

- To be able to test our counter application, we need to be able to load its HTML from `counter.html` and render it into this DOM while ensuring that the separate JS file `counter.js` is correctly executed during the construction of the DOM.

- We'll want to do this in a particular way. Specifically, if we implement multiple tests, we'll want to make sure each one starts with a fresh copy of the DOM that is unaffected by the actions of previously executed tests.

- This is a situation where implementing a helper function to perform setup work for our tests would be appropriate. As we talked about previously in this course, we want to be careful implementing helper functions and factoring our tests so they are too DRY. However, if our goal is to write tests that are clear and concise, factoring parts of the test (like shared setup) into a helper function can sometimes be helpful. The "Unit Testing" chapter of the book Software Engineering at Google gives the following good advice on how to approach this kind of code factoring in tests:

  > *Helper methods and test infrastructure can still help make tests clearer by making them more concise, factoring out repetitive steps whose details aren't relevant to the particular behavior being tested. The important point is that such refactoring should be done with an eye toward making tests more descriptive and meaningful, and not solely in the name of reducing repetition.*

- In other words, if we write a helper function to help factor code out of our tests, **it should not hide any details about how the tests work**.

- In our current situation, we can accomplish this with a helper function that simply initializes the DOM with HTML read from a specified file. By forcing the identity of the HTML to be passed into the function, we don't hide the details about what application the DOM represents. Here's what the function will look like:

```
const fs = require("fs")
function initDomFromFiles(htmlPath) {
    const html = fs.readFileSync(htmlPath, 'utf8')
```

```
    document.open()
    document.write(html)
    document.close()
}
```

- Here, we read the HTML file synchronously using the `readFileSync()` function from the `fs` module that's built into Node.js (which provides functionality for interacting with the local file system, e.g. reading and writing files).

- The `initDomFromFiles()` function also uses `document.open()`, `document.write()`, and `document.close()` to completely replace whatever are the current contents of the DOM with the contents of the file at `htmlPath`. It would also be possible to add content to the DOM using any mechanism we would normally use to do so in a browser, e.g. `document.createElement()`, etc.

- Unfortunately, some limitations on the way we can interact with JSDOM from Jest will prevent it from automatically loading a JS file based on a `<script>` tag in the HTML. We'll need to work around this.

- Luckily, we can execute a client-side JS file against the current DOM by simply importing it using `require()`. We must do this in a special way. In particular, by default, `require()` caches every file it loads and doesn't re-execute a file the second time it's imported (or the third, etc.). We, on the other hand, want to ensure that our client-side JS file is executed every time `initDomFromFiles()` is called to reset the DOM.

- Luckily, Jest provides a function called `isolateModules()` that allows us to import a "fresh" copy of a file for each test. We can use this function to load the client-side JS file in `initDomFromFiles()`. In particular, we'll accept the path to the JS file as an argument to `initDomFromFiles()`, and we'll load it like this:

```
function initDomFromFiles(htmlPath, jsPath) {
    ...
    jest.isolateModules(function () {
        require(jsPath)
```

```
    })
}
```

- The `initDomFromFiles()` function is now ready to be used to completely initialize the DOM for each test we want to run.

Testing a simple DOM-based application with the DOM Testing Library

- As we explored above, we can interact with the DOM rendered by JSDOM using the same mechanisms we use to interact with the DOM in an application running in the browser. For example, we can access nodes in the DOM using methods like `getElementById()`, `getElementsByClassName()`, etc.

- It is possible to implement tests based on these methods. For example, we could implement a test by using `getElementById()` to find a particular node in the DOM and then accessing that node's properties to construct the test's assertions.

- However, it's important to remind ourselves here of our goal to **make sure our tests interact with our code in the same way a user would**. In particular, end users don't find elements in an application based on the IDs assigned to those elements in the HTML code. Most users won't ever look at the HTML code for an application.

- Instead, **users find the elements they want to interact with based on the visual characteristics of those elements**. For example, a user might find a button based on the fact that it looks like a button or based on the text inside the button, or they might find a text input field based on its label or placeholder.

- Similarly, the user will assess the results of any particular interaction based on visual characteristics of the page, not by reading properties of DOM nodes in JS.

- We want our tests to interact with the application in these ways, too. To accomplish this, we'll use a library called the DOM Testing Library.

- The DOM Testing Library is part of a larger collection of libraries collectively called the Testing Library, whose purpose is to provide utilities for testing

client-side web applications by simulating interactions that resemble real end-user interactions.
    - In addition to the DOM Testing Library, which is designed for use as a general framework for testing web applications of any type, the Testing Library also includes sub-libraries specifically designed for testing applications implemented using React, Vue, Angular, and more.

- Conveniently, all of the libraries within the Testing Library collection are designed to integrate easily with Jest. In fact, one part of the Testing Library ecosystem is a companion library called `jest-dom` that provides custom matchers for use in Jest-based tests.

- Let's start to explore how these tools work by implementing a few tests for our little counter application. To be able to use the DOM Testing Library and `jest-dom`, we'll need to install them into the current project. We'll also install a companion library from the Testing Library collection called `user-event`, which we'll eventually use to simulate user interactions (more on this in a bit):

```
npm install --save-dev @testing-library/dom    \
    @testing-library/jest-dom @testing-library/user-event
```

- Now, we can go back to `counter.test.js` and implement a test. We'll start by importing the functionality provided by the DOM Testing Library, `jest-dom`, and `user-event`:

```
require("@testing-library/jest-dom/extend-expect")
const domTesting = require("@testing-library/dom")
const userEvent =
require("@testing-library/user-event").default
```

- Note that the first `require()` statement here for `jest-dom` is a little different from ones we've used up until now, since we're not assigning a variable from it. Instead, this `require()` statement will augment Jest's `expect()` function to add a set of custom matchers that we'll make use of in a second.

- The `require()` statement for `user-event` is a little different than normal, too, since we need to specifically import the `.default` field. This is a quirk of the fact that the `user-event` library is set up to be more easily imported with ES6

`import` statements, which we can't use here.

- Now, we can begin writing a test. We'll start with a simple test to verify that the value displayed by the counter increments when the user clicks it. Inside this test, we'll start by using our setup function to render `counter.html` and `counter.js` into the DOM:

```
test("counter increments when clicked", function () {
    initDomFromFiles(
        `${__dirname}/counter.html`,
        `${__dirname}/counter.js`
    )
})
```

- Note that we're using the value `__dirname` here, which is a value available within any file being executed by Node.js to provide the full path to the directory where that file lives. Here, we use it to find the location of `counter.html` and `counter.js`, assuming those files are in the same directory as the current file, `counter.test.js`.

- Once the DOM is rendered, we can grab a reference to the counter button itself. Remember, we want to do this in a way that resembles the way an end user might locate the counter button. The DOM Testing Library provides a few different mechanisms for doing this, and we'll explore a couple of them here. The first approach we'll take will grab the counter button based on the text inside it (i.e. "0"):

```
const counter = domTesting.getByText(document, "0")
```

- Once we've found the counter button, we want to simulate the user clicking on it. This is where we'll use the `user-event` library we set up a minute ago. This library is designed to help us emulate user interactions in a way that resembles what those interactions would look like if a real user was interacting with our application through the browser.

- In particular, real user interactions in the browser often result in a complex sequence of individual JS events being fired. For example, when a user clicks

on an element like a button, it actually generates this sequence of six different events:
    - A `mouseOver` event
    - A `mouseMove` event
    - A `mouseDown` event
    - A `focus` event (if the element is focusable)
    - A `mouseUp` event
    - A `click` event

- The `user-event` library is designed to simulate these complete interactions during testing.

- To use the `user-event` library, we'll begin by setting up a simulated user inside our test:

```
const user = userEvent.setup()
```

- Now, we can use that simulated user to make simulated interactions with the application. For example, we can simulate a click on the counter button:

```
user.click(counter)
```

- Note that this call to `user.click()` is *asynchronous*, since it takes a bit of time to execute all the individual events that comprise a click interaction. That means the result of the click might not immediately be available directly after the call to `click()`.

- We want to wait to make assertions on the state of the DOM until after the click interaction completes, so we need to wait for the asynchronous operation associated with `user.click()` to complete.

- Luckily, `user.click()` returns a JS promise to represent its eventual completion. Thus, for syntactic simplicity, we can use the async/await framework to wait for `user.click()` to complete.

- To do this, we must first mark the test function with the `async` keyword. Then we can prefix the call to `user.click()` with the `await` keyword:

```
test("counter increments when clicked", async function () {
    ...
    await user.click(counter)
})
```

- This will cause the test to pause at the call to `user.click()` until it completes, i.e. after the click interaction is executed. Then it will proceed with the rest of our test code.

- In this case, after the click interaction completes, we'll assert that the value displayed in counter button did indeed increment:

```
expect(counter).toHaveTextContent("1")
```

- In our assertion here, we're using the `toHaveTextContent()` matcher provided by the `jest-dom` library to verify that the button has the correct text content. This approach again simulates the way the user experiences an application, i.e. making assessments about the changing state of the application based on the text it displays.

- This gives us a complete test to make sure the counter behaves the way we want it to behave, and the test is implemented to interact with the app the way an end user would.

An alternative version of the same test

- In the test we implemented above, we used the DOM Testing Library's `getByText()` query to grab a reference to the counter button based on the text it contains. This may be appropriate in many situations, since finding an element based on its text content is one way an end user might locate an element in an application. However, the DOM Testing Library provides alternative query mechanisms, too.

- For example, here's a similar test that also verifies that the counter button correctly increments when clicked, but this one instead grabs a reference to the counter button based on its role (as a button):

```
test("...", async function () {
    initDomFromFiles(
```

```
        `${__dirname}/counter.html`,
        `${__dirname}/counter.js`
    )
    const counter = domTesting.getByRole(document,
"button")
    const user = userEvent.setup()
    await user.click(counter)
    expect(counter).toHaveTextContent("1")
})
```

- Here, the role `"button"` is an accessibility property (part of the WAI-ARIA specification) that is implicitly assigned to `<button>` elements in HTML. Many HTML elements have roles implicitly assigned. These roles often represent a useful, user-like way to access these elements for testing.

Testing a more complex application

- Let's continue to explore the way the DOM Testing Library works with a slightly more complex example than the counter application above.

- In particular, let's work on testing an application that adds new content to the DOM based on input from the user. The specific app we'll work on will allow a user to input the URL for a photo and a caption for the photo, and it will display that photo in a "card" element in the DOM.

- Here's the body of the HTML for this application, which we'll assume is stored in the file `photos.html`:

```
<form id="add-photo-form">
    <label>
        Photo URL
        <input id="url" name="url" />
    </label>
    <label>
        Caption
        <input id="caption" name="caption" />
    </label>
```

```
    <button>Add photo</button>
</form>
<ul id="photo-card-list"></ul>
```

- The body is divided into two main parts, a `<form>` where the user can enter the URL and caption for a photo and an empty `<ul>` where cards representing new photos will be added to the DOM.

- Assume the `<head>` element of the HTML contains a `<script>` tag that incorporates the client-side JS below (in a file named `photos.js`):

```
const form = document.getElementById("add-photo-form")
const urlInput = document.getElementById("url")
const captionInput = document.getElementById("caption")

form.addEventListener("submit", function (event) {
    event.preventDefault()
    const url = urlInput.value
    const caption = captionInput.value
    if (url && caption) {
        insertNewPhotoCard(url, caption)
        urlInput.value = ""
        captionInput.value = ""
    }
})
```

- The key thing to understand about this JS is that it attaches a listener to the `<form>` element in the HTML that grabs the URL and caption entered by the user and passes them to a function called `insertNewPhotoCard()`.

- The details of `insertNewPhotoCard()` are not included here, but assume that it inserts a new photo “card” into the DOM within the `<ul>` element in the HTML. Assume each photo card looks like this (where `{url}` and `{caption}` represent the URL and caption input by the user):

```
<li class="photo-card">
```

```
    <img src={url} alt={caption} />
    <p>{caption}</p>
</li>
```

- The additional complexity of this application will require additional testing to make sure it works the way we want it to. However, we can start the same way as we did for our simple counter application above, importing the needed dependencies and setting up the same `initDomFromFiles()` function to render the DOM at the beginning of each test (we could factor this function out into a separate file if we wanted to instead of duplicating it):

```
/**
 * @jest-environment jsdom
 */

require("@testing-library/jest-dom/extend-expect")
const domTesting = require("@testing-library/dom")
const userEvent =
require("@testing-library/user-event").default

const fs = require("fs")

function initDomFromFiles(htmlPath, jsPath) {...}
```

- Now we're ready to test our application.

Testing new DOM content generation

- Let's start working on our first test of the photos application. Our first test will verify that a photo card is correctly generated and inserted into the DOM when the user enters and submits a photo URL and caption. Here's the start of that test, initializing the DOM from the appropriate HTML and JS file:

```
test("counter increments when clicked", async function () {
    initDomFromFiles(
        `${__dirname}/photos.html`,
        `${__dirname}/photos.js`
    )
```

```
})
```

- Note that the test function is marked with the `async` keyword, since we'll use `await` inside to wait for asynchronous operations to complete.

- Once the DOM is set up, we'll grab references to the major elements of the application based on their visual properties using functions from the DOM Testing Library:

```
const urlInput = domTesting.getByLabelText(document, "Photo
URL")
const captionInput =
    domTesting.getByLabelText(document, "Caption")
const addPhotoButton = domTesting.getByRole(document,
"button")
const photoCardList = domTesting.getByRole(document,
"list")
```

- Note that we're using the `getByLabelText()` function to grab the URL and caption text input fields based on the text of their labels.

- Next, we can simulate the user's interaction with our application using the `user-event` library. Here, the interaction will be more complex. We want to simulate the user typing a URL into the URL text field, then typing a caption into the caption text field, and finally clicking the "add photo" button to submit those values:

```
await userEvent.type(urlInput,
"http://placekitten.com/480/480")
await userEvent.type(captionInput, "Cute kitty")
await userEvent.click(addPhotoButton)
```

- As before, we use `await` to wait for each interaction to complete, since each one consists of multiple individual actions. The `type()` function we're using here simulates the user clicking the input field to focus it and typing the individual characters in the specified string.

- We'll perform a number of assertions to ensure that the photo card element was correctly added. It may not be strictly necessary to use all of these assertions for the test to be useful, but they'll each provide us with an opportunity to explore different ways to test DOM content with the DOM Testing Library.

- We'll start with a simple assertion to ensure *any* element was added into the photo card list:

```
expect(photoCardList).not.toBeEmptyDOMElement()
```

- Here, we're using a couple new mechanisms: the `.not` modifier is an element of Jest that allows us to test the opposite of the assertion encoded by a matcher, and `.toBeEmptyDOMElement()` is a matcher from `jest-dom` that asserts that a given DOM element has no content within it. Together, they ensure that the photo card list has *something* inside it.

- Next, we'll make some assertions about *what* is in the photo card list. We'll start by trying to grab all the photo cards based on their `"listitem"` role and searching *only* within the photo card list:

```
const photoCards =
    domTesting.queryAllByRole(photoCardList, "listitem")
```

- Here, we're using a different DOM Testing Library query than we've used before. Specifically, we're using the `queryAllByRole()` query. The important distinction to notice here is the prefix to the query, `queryAllBy`.

- The DOM Testing Library allows us to execute queries slightly differently by using different prefixes. Here, the `queryAllBy` prefix differs from the `getBy` prefix we've used before in two important ways
    - It returns an array of all elements that match the query instead of just a single element (this is what the `All` portion of the query means).
    - It returns an empty array if no elements match the query. This differs from a `getBy` query (or `getAllBy` query), which throws an error if no elements match the query.

- We're querying for photo cards this way because we want to verify that the right number of photo cards were added to the DOM (i.e. 1). We can do this by asserting the length of the array returned by `queryAllByRole()` using Jest's

toHaveLength() matcher:

```
expect(photoCards).toHaveLength(1)
```

- Once we know exactly one photo card was inserted into the DOM, we can make some assertions about what that photo card looks like. We'll start by making sure the photo card contains an image that displays the correct thing (i.e. uses the correct URL):

```
const img = domTesting.queryByRole(photoCards[0], "img")
expect(img).toBeTruthy()
expect(img)
    .toHaveAttribute("src",
"http://placekitten.com/480/480")
```

- Note that we're again using a `queryBy` query to make sure we get a `null` back if the query doesn't match anything (instead of throwing an error, like a `getBy` query).

- Finally, we'll assert that the photo card contains the correct caption:

```
expect(
    domTesting.queryByText(photoCards[0], "Cute kitty")
).toBeTruthy()
```

- This test should pass when we run it, and it'll give us some confidence that our application correctly inserts a photo card when the user submits a URL and caption. Let's move on to test a few more of our application's behaviors.

Testing correct resetting of form values

- The next behavior we'll test is the correct resetting of the "url" and "caption" form fields when the user enters values and submits them. This is done in the client-side JS code for the application by setting the `value` field of the form fields to the empty string `""`.

- We can set this test up mostly the same way as we set up the previous test, rendering the DOM, grabbing references to relevant DOM elements, and simulating user interactions to enter a URL and caption and then submit them

(function parameters are omitted here for clarity and use the same values as the test above):

```
test("clears form when correctly submitted", async function
() {
    initDomFromFiles(...)
    const urlInput = domTesting.getByLabelText(...)
    const captionInput = domTesting.getByLabelText(...)
    const addPhotoButton = domTesting.getByRole(...)

    await userEvent.type(urlInput, "...")
    await userEvent.type(captionInput, "...")
    await userEvent.click(addPhotoButton)
})
```

- We can make two simple assertions in this test using [jest-dom's toHaveValue() matcher] to ensure that the form values are cleared:

```
expect(urlInput).not.toHaveValue()
expect(captionInput).not.toHaveValue()
```

Testing correct error handling

- Our application is set up so that if the user submits the form without providing both a URL and caption, no new photo card will be added, and the form values they entered are not cleared. This is a useful behavior to validate through a test.

- Again, we'll set up the test in a similar way to the ones above, but we'll only simulate a user interaction that types a caption and clicks the "add photo" button without typing a URL:

```
test("clears form when correctly submitted", async function
() {
    initDomFromFiles(...)
    const captionInput = domTesting.getByLabelText(...)
    const addPhotoButton = domTesting.getByRole(...)
    const photoCardList = domTesting.getByRole(...)
```

```
    await userEvent.type(captionInput, "Lonely caption")
    await userEvent.click(addPhotoButton)
})
```

- For this test, we'll make two different assertions: one that asserts that the photo card list is still empty after the button was clicked and another that asserts that the caption typed by the user is not cleared:

```
expect(photoCardList).toBeEmptyDOMElement()
expect(captionInput).toHaveValue("Lonely caption")
```

Using snapshot testing to validate more complex DOM contents

- As a last test, let's make sure that our application correctly handles the submission of multiple photo cards by the user. In particular, if the user correctly submits multiple URLs and captions, we'd expect multiple photo cards to be added to the DOM.

- With the methods we've used so far, such a test could become quite complex. Indeed, we saw that validating the correct insertion of just a single photo card above required several assertions.

- As an alternative to using many individual assertions to validate the state of the UI, Jest supports a mechanism called ***snapshot testing***.

- A snapshot test works just like its name implies: the UI is rendered, actions are taken on the UI, and then a snapshot of the UI is taken. This snapshot is then compared with a reference snapshot that is stored in a file next to the test, and if the two snapshots match, then the test passes. If the two snapshots are different, then the test fails.

- Let's see how to use snapshot testing to implement our test for multiple photo card insertion. The test will start similarly to the tests above, rendering the UI and simulating user interactions to submit *two* photo cards:

```
test("inserts multiple photo cards", async function () {
    initDomFromFiles(...)
    const urlInput = domTesting.getByLabelText(...)
```

```
    const captionInput = domTesting.getByLabelText(...)
    const addPhotoButton = domTesting.getByRole(...)
    const photoCardList = domTesting.getByRole(...)

    // Submit first photo card.
    await userEvent.type(urlInput, "...")
    await userEvent.type(captionInput, "...")
    await userEvent.click(addPhotoButton)

    // Submit second photo card.
    await userEvent.type(urlInput, "...")
    await userEvent.type(captionInput, "...")
    await userEvent.click(addPhotoButton)
})
```

- Now, we'll make an assertion to verify that the contents of the photo card list match the reference snapshot (more on the creation of the reference snapshot in just a second) using [Jest's `toMatchSnapshot()` matcher]:

  ```
  expect(photoCardList).toMatchSnapshot()
  ```

- The first time this test is executed, Jest will generate the reference snapshot, which will capture the DOM contents of the photo card list. The reference snapshot will be stored in a file within a directory called `__snapshots__/` that will be created by Jest next to the test file. If you open that file to inspect it, it will look mostly like HTML.

- This snapshot file should be a part of our test implementation. For example, it should be committed into our Git repository, just like our test code.

- Once the reference snapshot file is created, Jest will compare against it every time the test is executed in the future. If the current snapshot doesn't match the reference snapshot, then Jest will treat it as a test failure, and it will output the differences between the two snapshots.

- In this way, Jest will catch any changes to the way the UI is rendered.

- Of course, there may occasionally be valid reasons why our UI might change. For example, we may decide we want to change the way a photo card looks. Jest would still consider this a test failure.

- We can resolve test failures that result from valid changes to our code by rerunning Jest with the `--updateSnapshot` option. If we're running our tests using `npm`, we'd do this as follows:

  ```
  npm test -- --updateSnapshot
  ```

- Note that this would regenerate the reference snapshots (i.e. update the snapshot files) for *all* failing snapshot tests. Thus, we'd want to make sure we fix any snapshot tests that are falling because of a bug in the code before regenerating reference snapshots. Alternatively, we could run Jest with an option that limits which tests are executed, like the `--testNamePattern` option.

- These are all the tests we'll implement for our photos application, though there are more tests we could add. For example, we could add additional tests to cover all the possible ways a user could omit a value before clicking the submit button (i.e. no URL, no caption, no URL or caption). We could also add a test to make sure the form is correctly submitted when the user hits Enter key in one of the form fields (this should work as the app is currently implemented. These tests are left as an exercise for you.