Test Doubles

- There will be times when we'll want to test code that interacts in complex ways with other pieces of software, e.g. by calling into a third-party library or by interacting with a network service.

- Imagine, for example, writing tests for a function that makes a request to an external server and then saves the result in a database. There are some important concerns we need to keep in mind with these tests:
    - They could take a long time to run, since network communication and database operations can be somewhat long-running operations.
    - They could be ***flaky*** because, for example, the network could randomly fail, or tests could overwrite each other's data in the database, depending on the order in which tests execute.

- ***Test Doubles*** are a mechanism that can be helpful in these situations. Test doubles give us more control over our tests by replacing one or more production software components with a different (usually simplified) component during testing.

- In particular, when writing tests, we want to focus on one specific element of the software within each test, sometimes referred to as the ***system under test*** or ***SUT***. **Test doubles are used to "stand in for" other software components the SUT interacts with** (sometimes called ***collaborators*** or ***depended-on components*** (***DOCs***)) whose behavior may impact the test but which aren't themselves being tested.

- Importantly, **test doubles provide the same API as the real software components they stand in for**. This means that, from the perspective of the SUT, a test double is indistinguishable from the real component.

- In the example described above where we want to test a function that makes a network request to an external server and stores the results in a database, the function itself is the SUT, and the external server and database are collaborators.

- Since it is the function itself that's being tested and not the collaborators, we could replace the network call to an external server with a test double that "mocked" that server's response without actually requiring a network call, or we could use a "fake" in-memory database implementation to replace a real

database.

- This would give us more control over the interactions between the function (i.e. the SUT) and its collaborators, so that our tests could focus on the behavior of the function itself.

- This is generally the rationale for using test doubles. In particular, the behavior of test doubles is *controlled* (by us, the test implementers), which can make tests that use doubles more predictable and faster to run while allowing us to focus on the behavior of the SUT.

- Here, we'll explore in more detail what test doubles are and how they work, and we'll see some mechanisms that will help us incorporate doubles into our tests. First, we'll look at the various kinds of test doubles.

Types of test doubles and when to use them

- There are several different types of test doubles that are distinguished primarily by how and why they are used. Here, we'll explore the main types of test doubles and talk about the role each type plays.
    - Note that there are different approaches to classifying the various types of test doubles, including this approach by Gerard Meszaros and this approach by Google's engineers. The categorization below is an attempt to synthesize these approaches.

Stubbing

- Sometimes, the SUT will receive ***indirect inputs***, i.e. inputs that aren't passed directly to the SUT (e.g. as function parameters) but instead come from some component within the environment/context in which the SUT executes.

- In order to gain control over these indirect inputs, we may wish to replace a part of the environment/context of the SUT with a ***stub***, which is a test-specific implementation of an interface on which the SUT depends that is configured to respond to calls by the SUT with predefined values.

- Stubs are useful when there is a behavior of the SUT that is difficult to test because we can't control the indirect inputs to the SUT, since a stub essentially allows us to "inject" indirect inputs into the SUT.

- As an example of a situation when a stub might be appropriate, imagine we have a function that generates an HTML string to display the current time of day, according to the system clock, and we want to test special functionality of that function that displays the string “noon” if the current time is exactly 12:00 pm.

- In this situation, the system clock provides the current time to the function as an indirect input. Since the system clock is difficult to control and it’s very challenging to try to execute our test exactly at noon, we could replace it in our test with a stub that is configured to send a predefined time value (i.e. 12:00 pm) to our function.

Spying

- Sometimes the SUT will have ***indirect outputs***, i.e. outputs that aren’t returned directly from the SUT (like function return values) but instead are passed as inputs to some component within the environment/context in which the SUT executes.

- In order to observe these indirect outputs, we may wish to replace a part of the environment/context of the SUT a ***spy***, which is designed to serve as an observation point by recording calls made to it by the SUT and providing access to the values passed to it by the SUT in those calls so we can make assertions on those values in our tests.

- A spy can also allow us to control (or prevent) side effects from actually happening.

- Spies are useful when we want to implement tests to validate side effects of the SUT, since they provide us with a mechanism that allows us to observe those side effects.

- As an example of a situation in which a spy might be appropriate, imagine we are working on a flight management system and want to test the function that cancels a flight. As part of our test, we want to verify that the function correctly triggers email alerts to all passengers on the flight (perhaps without actually sending any emails).

- We could accomplish this by installing a spy on the function that is called to trigger email alerts. The spy would record calls to this function, perhaps also preventing actual emails from being sent.

Faking

- Sometimes, the SUT will depend on a component with which interactions can be costly or where interactions could have unnecessary or undesirable side effects.

- In these cases, instead of using the real, production version of a component, we can use a ***fake***, which is a simpler, lighter-weight implementation of the same functionality the component provides, perhaps also without undesirable side-effects.

- As an example of a situation when a fake might be appropriate, imagine we want to test a function that interacts with a database.

- Interacting with a real, production database from the test could be costly, so we might choose to use a fake to replace the production database with a simple in-memory database based on hash tables. The fake unobtrusively allows the function being tested to perform all its normal behaviors while saving the expense of interacting with a real database.

Dummy objects

- Sometimes, the SUT will require inputs (e.g. function parameters) that won't actually impact the behavior of the SUT or the test but that may be nontrivial to construct, and we'll need to provide values for these inputs to conform to the API of the SUT.

- In these cases, instead of spending the time to come up with meaningful values for these inputs, we may just use a ***dummy object***, i.e. an instance of the required input type that is easy to construct. In some cases, a dummy object could even be a null value.

- As an example of a situation when a dummy object might be appropriate, imagine we have an `Invoice` class, and we want to test the method of that class that allows us to add a line item (i.e. a specific quantity of a specific product at a specific price) to an invoice, but in order to create an `Invoice` object, we need a `Customer` object, which in turn requires an `Address` object, which in turn requires a `City` object, etc., even though none of these objects will be used in or will impact our test.

- In this situation, instead of spending the time to build a `Customer`, an `Address`, a `City`, etc. (and probably cluttering up our test with this initialization) we could create a dummy `Customer` object to use to instantiate the `Invoice`.

A note about the word “mock”

- The word “mock” is used in various ways in the testing world. For example, Gerard Meszaros and Martin Fowler use “mock” to describe a separate type of test double that is much like a spy but that performs assertions within itself instead of providing access to the recorded parameters passed by the SUT for verification “outside” the mock.

- Here, we will use the term ***mock*** in a sense more similar to the way it is used by the Google engineers in Software Engineering at Google, by Kent C. Dodds, and by the developers of Jest to refer to any test double whose behavior is specified inline in a test. Under this usage, a mock could be either a stub or a spy. It could also be a hybrid between a stub and a spy that allows both the injection of indirect inputs into the SUT *and* the recording of indirect outputs from the SUT.

- We will use the term “mocking” to refer to the act of replacing real software components with mocks.

Basic mocking with Jest

- Let’s start to explore how to incorporate doubles into our tests. We’ll begin by exploring some of the mocking functionality that’s built directly into Jest.

- We’ll specifically implement some tests for the following function, which is designed to register an application user by storing a record of that user in a database:

```
module.exports = function registerUser(email, password) {
    const record = {
        email: email,
        password: bcrypt.hashSync(password, 8)
    }
    try {
        const userId = Database.save(record)
        return userId
```

```
    } catch {
        return null
    }
}
```

- Here are a couple things to note about this function:
    - The primary thing the function does is use the `Database` module to store a user record in a database. `Database` is not a real database module but the call here to `Database.save()` corresponds to the way we would use an actual database implementation like MySQL or MongoDB. The specific details about how the interaction with the database don't matter for our purposes here.
    - The user record stored in the database contains the user's email address and a ***hashed*** version of the user's password. The hashing is not an important part of the function from the perspective of test doubles (i.e. we won't mock it), but it is a security measure we'd take in a production application when saving a user's password to a database.
        - If you're curious about password hashing this is a good article about it.
    - The function here has some error handling built into it, in case communication with the database fails for some reason.

- We might want to write a test for the `registerUser()` function above that verifies the user record is correctly saved in the database. However, executing this test against a real, production database might be undesirable for several reasons:
    - Communication with a database typically happens over a network, so all of the potential testing pitfalls inherent in a network call (e.g. flakiness due to network unreliability) would apply to this test.
    - We may not want to store a record for a testing user in a production database, and we may not have a dedicated instance of the database available for testing.

- **Our preferred approach here would be to use a fake implementation of `Database`**, e.g. one that stored data in memory instead of communicating with a real database instance over the network. However, we may not have a fake available, and implementing one could be time-consuming.

- In this situation mocking can still allow us to implement tests that can give us confidence that `registerUser()` is working correctly.

Spying with Jest

- Let's first explore how we can test the `registerUser()` function above using a spy to verify that the user record was saved to the database.

- Let's start by setting up the test by setting an email address and a password and passing them into a call to `registerUser()`:

```
test("saves user record in database", function () {
    const email = "iamfake@oregonstate.edu"
    const password = "pa$$Word123"

    registerUser(email, password)
})
```

- In order to verify that the call to `registerUser()` here correctly interacts with the database, we will attach a spy to `Database.save()`. Jest provides a method called `spyOn()` that will allow us to do this:

```
const Database = require("...")
test("saves user record in database", function () {
    const email = "iamfake@oregonstate.edu"
    const password = "pa$$Word123"
    const spy = jest.spyOn(Database, "save")

    registerUser(email, password)
})
```

- Once we've attached a spy to `Database.save()`, we can use that spy to assert that `Database.save()` was indeed called:

```
expect(spy).toHaveBeenCalled()
```

- If we wanted to, we could even verify that `Database.save()` was called the correct number of times (one, in this case):

  ```
  expect(spy).toHaveBeenCalledTimes(1)
  ```

- There's more that we can do once we've attached a spy to a function, and we'll explore this in just a minute.

- After we're finished using the spy, we need to restore `Database.save()` back to its original, unspied version:

  ```
  spy.mockRestore()
  ```

- Make note here that **spying in this way requires us to incorporate implementation details into our test**. Here, for example, we have to reference one of the dependencies of `registerUser()` (i.e. `Database.save()`) in the test.

- **As we've discussed, relying on implementation details like this in a test is undesirable**. However, here, this is the only way we can reasonably verify that `registerUser()` has the correct side effect (i.e. saving the user record in the database).

- As we mentioned above, using a fake would be a better approach to verifying the database interaction here because it wouldn't require us to rely on implementation details in the test. We'll discuss this in more detail later along with other best practices and guidelines for using test doubles.

Accessing call information from a spy

- One of the benefits of using a spy is that we not only verify that the spied function is called, but we can also access information about calls made to the spied function.

- To do this in Jest, we can use the value returned by `jest.spyOn()`, which is an instance of the Jest mock function class. This class provides access to information about calls to the spied function through its `.mock` property.

- For example, in the test above, the field `spy.mock.calls` provides an array containing the call arguments for all the calls to `Database.save()`. If we assume that the user record to be inserted into the database is the first argument to `Database.save()`, then we could access it like this:

```
const userRecord = spy.mock.calls[0][0]
```

- Here, we use the double index `calls[0][0]` because `calls` itself is an array whose length corresponds to the number of calls made to the spied function, and each element of `calls` is also an array whose length corresponds to the number of arguments passed to that particular call to the spied function. Thus, `calls[0][0]` represents the first argument to the first call to the spied function.

- The `.mock` field of Jest's mock function class also provides a value called `lastCall` as a convenience for accessing the arguments of the last call to the spied function. Thus, since our spied function `Database.save()` is only called once (as verified by the test we wrote above), we could also access the user record as the first argument to the last call to the spied function, like this:

```
const userRecord = spy.mock.calls[0][0]
```

- The documentation for Jest's mock function class describes more of the functionality it provides. We'll use a bit more of this functionality in a minute.

- Let's use the ability to access information about calls to the spied function in order to write a test that verifies that the user's password is successfully hashed by `registerUser()` before it is saved in the database. We can accomplish this by making assertions about the user record that's saved in the database, i.e. the value that's passed as an argument to `Database.save()`.

- Here's the outline of our test, without any assertions implemented yet:

```
test("password is hashed before stored in database",
function () {
    const email = "iamfake@oregonstate.edu"
    const password = "pa$$Word123"
    const spy = jest.spyOn(Database, "save")
```

```
    registerUser(email, password)

    const userRecord = spy.mock.lastCall[0]

    spy.mockRestore()
})
```

- After grabbing the user record here (but before the call to `spy.mockRestore()`), let's add some assertions to verify that the password in the user record stored in the database is hashed.

- If we assume that the user record contains a field called `password` that should hold the hashed password, then we can start with an assertion that verifies that the `password` field of the user record *does not* match the plain text password passed to `registerUser()`:

```
expect(userRecord).toMatchObject({
    password: expect.not.stringContaining(password)
})
```

- We are using two new mechanisms within this assertion that bear a little more explanation:
    - The `toMatchObject()` matcher is built into Jest and allows us to verify that a JavaScript object contains some subset of specified properties with specified values. Here we are using it to verify that the user record contains a property called `password` with a particular value.
    - We are using Jest's "asymmetric matcher" `expect.not.stringContaining()` to specifically verify that the `password` field of the user record *does not* contain the original plain text password, as we would expect of a password hash.

- If we wanted to, we could make additional assertions to help confirm that the password is correctly hashed. For example, we could add the following two assertions to verify that the password hash has the correct length (according to the documentation for the bcryptjs module) and that is has the correct prefix (all hashes computed by the bcryptjs module start with the prefix $2a$, which indicates the specific hashing algorithm used):

```
expect(userRecord.password).toHaveLength(60)
expect(userRecord.password.startsWith("$2a$")).toBeTruthy()
```

Adding a stub with Jest

- The default behavior of spies created with `jest.spyOn()` is to continue to call the original spied function. In other words, the real `Database.save()` is still being called in the tests above.

- This is not what we want in our situation. Specifically, one of our main purposes for mocking `Database.save()` is to avoid calling the real version, since it might, for example, need to communicate with the database over the network.

- In addition to just spying, `jest.spyOn()` also allows us to stub the spied function, replacing its actual functionality with predefined functionality for the purposes of testing.

- In particular, once the spy is created, we can call `mockImplementation()` on it. `mockImplementation()` takes a single argument, which should be a function specifying the behavior of the stub with which we're replacing the original spied function.

- For example, in both of the above tests, we could prevent the original version of `Database.save()` from being called by replacing it with an empty stub, like this:

  ```
  spy.mockImplementation(function () {})
  ```

- Here, using an empty stub is fine, because we don't care about any specific behavior of `Database.save()`, only preventing the real version of `Database.save()` from executing.

Using a stub with a specific behavior

- Sometimes, we'll want to use a stub that behaves in a specific way. For example, we might want to test that our `registerUser()` function correctly returns `null` if there's any problem saving the user record in the database.

- In particular, recall that `registerUser()` is set up to catch an error thrown by `Database.save()` and to return `null` in this situation.
    - There are potentially many reasons why `Database.save()` could throw an error, e.g. a failure to communicate with the database itself. For our purposes here, we just care that `registerUser()` responds correctly when an error is thrown, whatever its cause.

- In order to test this behavior, we can replace `Database.save()` with a stub that throws an error. Here's what a test for this might look like:

```
test("returns null on database error", function () {
    const email = "iamfake@oregonstate.edu"
    const password = "pa$$Word123"
    const spy = jest.spyOn(Database, "save")
    spy.mockImplementation(function () { throw new Error()
})

    const response = registerUser(email, password)
    expect(response).toBeNull()

    spy.mockRestore()
})
```

- Note that the stub here doesn't have to be complicated (and it's not). We only need it to perform the specific behavior we want to test against.

Other ways to mock with Jest: Faking the system clock

- Jest provides other mechanisms for mocking, as well. We'll explore one more of these here, which is Jest's ability to fake the system clock. This is useful, since we may occasionally need to write tests for software components that use the system clock as an indirect input.

- Let's say we want to implement a test for a UI-based application that simply displays the current time. If it is exactly 12:00 am, the application displays the string "midnight". If it is exactly, 12:00 pm, the application displays the string "noon", and otherwise, the application displays the am/pm time.

- Here's HTML (just the contents of the `<body>`) and client-side JS for this simple application:

```
<!-- currentTime.html -->
<p>The current time is: <span id="time-span"></span></p>

// currentTime.js
const timeSpan = document.getElementById("time-span")
const currentTime = new Date()
const hours = currentTime.getHours()
const minutes = currentTime.getMinutes()

let timeString = ""
if (hours === 0 && minutes === 0) {
    timeString = "midnight"
} else if (hours === 12 && minutes === 0) {
    timeString = "noon"
} else if (hours === 0) {
    timeString = `12:${minutes} am`
} else if (hours === 12) {
    timeString = `12:${minutes} pm`
} else if (hours > 12) {
    timeString = `${hours % 12}:${minutes} pm`
} else {
    timeString = `${hours}:${minutes} am`
}

timeSpan.textContent = timeString
```

- Importantly, notice that in the JS here that the `Date` class is used to read the current time from the system clock. In particular, by default, the `Date` class constructor creates a new `Date` object representing the current time, based on the system clock.

- In order to adequately test this application, we'll need to have a way to control the time it displays. In other words, we don't want to try to time our tests to

execute *exactly* at midnight to verify that the application correctly displays the string “midnight”.

- To do this, we can fake the system clock with Jest to have it return a specific, predefined time that we control (e.g. midnight).

- Let’s start by setting up the test. Since this is an HTML/JS app, we’ll want to make sure we’re set up to use some of the tools we’ve previously studied for testing DOM-based apps. For example, we’ll want to make sure to install the JSDOM testing environment for Jest along with the DOM Testing Library tools we’ve previously used:

```
npm install --save-dev jest-environment-jsdom                \
    @testing-library/dom @testing-library/jest-dom
```

- We’ll also want to make sure to set up the testing environment, import the needed libraries, and implement a function for initializing the JSDOM-based DOM from HTML and JS, just like we did previously:

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

require("@testing-library/jest-dom/extend-expect")
const domTesting = require("@testing-library/dom")

const fs = require("fs")

function initDomFromFiles(htmlPath, jsPath) {
    const html = fs.readFileSync(htmlPath, 'utf8')
    document.open()
    document.write(html)
    document.close()
    jest.isolateModules(function () {
        require(jsPath)
    })
}
```

- Then, we can begin our test. For now, we'll just initialize the DOM from the HTML and JS (assuming they're in files named `currentTime.html` and `currentTime.js` in the same directory as the test file):

```
test("displays 'midnight' at 00:00", function () {
    initDomFromFiles(
        __dirname + "/currentTime.html",
        __dirname + "/currentTime.js"
    )
})
```

- To control the system clock, we'll use Jest's fake timers API. This actually allows us to control many different kinds of time, including the system clock.

- At the top of our test, we'll start by instructing Jest to use fake timers, and then we'll set the system time to be exactly midnight:

```
test("displays 'midnight' at 00:00", function () {
    jest.useFakeTimers()
    jest.setSystemTime(new Date(2023, 4, 8, 0, 0))
    ...
})
```

- Here, the call to `jest.setSystemTime()` will specifically cause the system clock to always return "May 8, 2023, 00:00" (i.e. the last two arguments signify 0 hours, 0 minutes; the 4 indicates May, since the month indices are 0-based, i.e. 0 indicates January). This will, for example, be the date and time assigned to the new `Date` object that's used to query the time in `currentTime.js`.

- Thus, we should be able to set up an assertion that verifies that the application does indeed display midnight using mechanisms from the DOM Testing Library:

```
expect(domTesting.queryByText(document, "midnight"))
    .toBeInTheDocument()
```

- Finally, we can "reset" Jest to make it go back to using the real system clock at the end of the test:

```
jest.useRealTimers()
```

Mocking network calls with Mock Service Worker

- We'll frequently need to write tests for applications that communicate with some external service over the network, e.g. fetching data from an external server to display in the application or sending data to an external server to store there.

- As we've mentioned, testing against network operations can be very unreliable and is a prototypical use case for test doubles. In particular, doubles are often used to avoid actually making network calls during testing.

- Let's explore here how to write tests that use doubles to mock network communication.

- We'll write tests against an application that allows the user to enter a search query and then sends that query to the GitHub API to search for repositories on GitHub that match the search query.

- The specific method from the GitHub API that the application will use is the "search repositories" method. To use this operation, the application must make an HTTP call (specifically an HTTP GET request) to the following URL, plugging the specific search query into the indicated location:

  ```
  https://api.github.com/search/repositories?q={search_query}
  ```

- For example, to search for the search query "jest", the application would make a call to the following specific URL:

  https://api.github.com/search/repositories?q=jest

- You can see what the data returned by GitHub API looks like by visiting the URL above in your browser (you can just click on it from this document). The data itself is formatted in the JSON format, which is a widely-used format for representing structured data that resembles a JavaScript object (the acronym JSON stands for JavaScript Object Notation).

- Here is a very abbreviated depiction of what the data returned by GitHub looks like:

```
{
    "total_count": 65820,
    "incomplete_results": false,
    "items": [
        {
            "name": "jest",
            "full_name": "jestjs/jest",
            "owner": {...},
            "html_url": "https://github.com/jestjs/jest",
            ...
        },
        ...
    ]
}
```

- Here, the `"items"` field contains the actual search results, with each GitHub repo that matches the search query represented there as an object with fields like `"full_name"`, `"html_url"`, etc. representing the different properties of the GitHub repo.

- Here's an (abbreviated) HTML/JS application that displays a text box where the user can type a search query and sends that query to the GitHub API when the user submits the search query:

```
<!-- githubSearch.html -->
<form id="search-form">
    <input placeholder="Search GitHub" id="query"
name="query" />
    <button>Search</button>
</form>
<ul id="results-list"></ul>

// githubSearch.js
const form = document.getElementById("search-form")
const queryInput = document.getElementById("query")
```

```
form.addEventListener("submit", async function (event) {
    event.preventDefault()
    const q = queryInput.value
    if (q) {
        const response = await fetch(

`https://api.github.com/search/repositories?q=${q}`
        )
        const results = await response.json()
        displaySearchResults(results)
    }
})
```

- The important thing to notice about the code above is that the `fetch()` function is used to make an HTTP call to the GitHub API using the URL for the "search repositories" method. The function waits for the response from GitHub and then parses the JSON-formatted results.

- We'll leave the definition of `displaySearchResults()` unspecified here, since they're not that important. We'll just assume that this function inserts `<li>` elements into the `<ul>` in the HTML to display each of the elements in the `items` array returned by GitHub. We'll further assume that each of these `<li>` elements contains text displaying the `full_name` property of its corresponding entry in `items`.

- Let's start to implement a test for this application to see how we can use a double to mock the network call to the GitHub API. We'll start our test in a new file that begins with the same setup we've been using for testing DOM-based applications:

```
/**
 * @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) {...}
```

- Note that we include an additional `require()` here to import the User Event library to simulate user interactions with the application. You'll need to make sure this is installed to be able to use it:

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

- We'll begin our test itself like we often do, rendering the application into a testable DOM, grabbing references to key elements in the application, and simulating the relevant user interactions:

```
test("renders GitHub search results", async function () {
    initDomFromFiles(
        __dirname + "/githubSearch.html",
        __dirname + "/githubSearch.js"
    )
    const queryInput =
        domTesting.getByPlaceholderText(document, "Search
GitHub")
    const searchButton = domTesting.getByRole(document,
"button")

    const user = userEvent.setup()
    await user.type(queryInput, "Jest")
    await user.click(searchButton)
})
```

- We're not using a test double for the network call here yet, but let's still run the test, anyway. If we do, we'll see that the test will fail, even though we have no assertions. This is a sign that there's a problem running the application code. In this case, we see the following error when Jest reports the test results:

```
ReferenceError: fetch is not defined
```

- The issue causing this error is that there's no available implementation of the function `fetch()`, which, remember, is used in the application itself to execute an HTTP call to the GitHub API.

- Normally, `fetch()` would be supplied by the browser, and we wouldn't need to do anything special. However, here in our testing environment, we need to provide an implementation of `fetch()` for the application to use. A good bet is the package `whatwg-fetch`, which implements (a subset of) the official `fetch()` standard. We can install this package like this:

  ```
  npm install --save-dev whatwg-fetch
  ```

- Once the package is installed, we can import it by including the following `require()` statement at the top of our test file:

  ```
  require("whatwg-fetch")
  ```

- Note that `whatwg-fetch` is what's known as a ***polyfill***. It will "fill in" an API that would normally be provided natively by the browser. In this case, it will make `fetch()` available as a global, just like it would be in a browser.

- With that fix, our test will no longer fail. If we wanted to, we could even include an assertion to make sure that some search results are being displayed, based on the data that's fetched from the actual GitHub API.

- There's one important thing to keep in mind before we write this assertion, which is that the call to `fetch()` will take time to execute, since it requires data to make a round-trip over the network to the GitHub API and back.

- This means that the search results won't immediately be displayed in the application as soon as the user clicks the search button, so if we query for them immediately, we won't find them.

- For these situations, where we need to wait for something to appear, the DOM Testing Library includes special versions of its query functions that start with the prefix `findBy...` (for querying for a single element) or `findAllBy...` (for

querying for multiple elements). These queries wait for the specified elements to appear in the DOM. If they are not found after a specified timeout (which defaults to 1 second), then the query fails.

- This is ideal for our situation here, where we want to wait for the search results to appear. Assuming again that the search results are each displayed as `<li>` items, we can make the following query to find them when they appear:

```
const resultItems =
    await domTesting.findAllByRole(document, "listitem")
```

- Note here that we use `await` to wait for the promise returned by `findAllByRoile()` to resolve.

- We could include a simple assertion now, and it should pass, e.g.:

```
expect(resultItems).not.toHaveLength(0)
```

- Again, though, this assertion passes because we're fetching real search results from the real GitHub API over the network. We'd like not to base our test on a network call. Let's start to explore how to use a double to mock this network call.

Setting up Mock Service Worker to fake the GitHub API

- There are different ways we could apply a double to avoid making an actual network call to the GitHub API here. One option, for example, would be to use Jest's `spyOn()` functionality, which we explored above, to stub the `fetch()` function itself.

- There are some drawbacks to this approach, though. For example, as we mentioned above, mocking requires us to incorporate implementation details of our code into our tests. We'd like to avoid that if we can. It would be better to use a fake somehow, which would allow us to keep the implementation details out of the tests, isolated within the fake.

- In addition, there are many different HTTP clients we could use in our application code, and `fetch()` is just one of them. At some point, we might decide we'd rather use Axios or Superagent, and if we did that, we'd have to refactor our tests

to update `fetch()` mocks into Axios or Superagent mocks.

- The approach we'll take instead of mocking `fetch()` will be to fake the GitHub API itself using a tool called Mock Service Worker (or just MSW). MSW is very cool because it is designed to *intercept* network requests made with *any* HTTP client and send those requests to a fake server we set up.

- Because the fake server is actually just another function defined in our testing code, MSW allows us to avoid network calls during testing while at the same time avoiding stubs or modifications to our application code.

- We'll need to start by installing MSW:

  ```
  npm install --save-dev msw
  ```

- Before proceeding with MSW, we'll need some data to power our fake GitHub API. We can get this by following the GitHub API URL above and saving the entire JSON data it sends us as a file called `searchRepositoriesResults.json` right next to our testing file. Here's the URL again:

  https://api.github.com/search/repositories?q=jest

- Then, we can import that file into our testing code along with the needed pieces from MSW:

  ```
  const rest = require("msw").rest
  const setupServer = require("msw/node").setupServer

  const searchRepositoriesResults =
      require("./searchRepositoriesResults.json")
  ```

- Here, the `rest` package we're importing from MSW is specifically a tool for faking "RESTful" HTTP APIs. For this course, you don't need to know too much about RESTful APIs other than the following:
  - Methods in a RESTful API are represented with URLs (like the GitHub "search repositories" URL above) and HTTP methods (like HTTP GET, which is the method that's implicitly used by the `fetch()` call in our application).

    - RESTful API methods are called by sending an HTTP request to the API server (which is what `fetch()` does). The API server then sends a response back to whoever sent the request, and that response contains the results of the request.

- Keeping these things in mind, we can set up a fake GitHub API server with MSW that specifically intercepts HTTP GET requests to the GitHub API's "search repositories" URL, and we'll provide a function that specifies what to do when these requests are intercepted. This will look like this:

```
const server = setupServer(
    rest.get(
        "https://api.github.com/search/repositories",
        function (req, res, ctx) {
            return res(ctx.json(searchRepositoriesResults))
        }
    )
)
```

- Here, the function we supply to specify what to do when a request is intercepted simply returns a mocked response that contains the JSON data we downloaded from the GitHub API and saved locally.
    - The `ctx` argument to this function just contains a set of helper functions that make it easy for us to send a mocked response.

- All we need to do to "install" our fake server is to call `server.listen()` before the tests execute. We can do this using Jest's `beforeAll()` function, which allows us to perform one-time setup before the tests begin to execute:

```
beforeAll(function () {
    server.listen()
})
```

- We'll also use Jest's `afterAll()` function to close down the fake server after all the tests have finished:

```
afterAll(function () {
    server.close()
```

```
})
```

- Now, when we run our test, it will use our fake server instead of sending a network call to the real GitHub API. This means we can now make more meaningful assertions about what's displayed in the application.

- For example, we could verify that the correct number of results is displayed (based on the contents of our saved JSON data). We could even verify that some of the correct text is displayed:

```
expect(resultItems).toHaveLength(
    searchRepositoriesResults.items.length
)
expect(resultItems[0]).toHaveTextContent(
    searchRepositoriesResults.items[0].full_name
)
```

- We could include more assertions if we wanted to, or we could even perform a snapshot test to verify that the application was rendering the correct results based on the data we sent it from our fake server.

Best practices and guidelines for using test doubles

- TBD