Learning Objectives

Learners will be able to…

- **Differentiate mocks and stubs**
- **Create tests that utilize mocks**
- **Create tests that utilize stubs**

info

Make Sure You Know

Students should know JavaScript and HTML basics, including React and Axios.

Limitations

This is not an assignment on React or Axios. These technologies are referenced only in the context of testing with mocks and stubs.

Module and Unit Testing

Defining Module Testing

Module testing is a procedure for testing the individual subprograms, subroutines, classes, or strategies in a program. In the module test pattern, tests are composed as small, function-based tests executed inside the test suite.

This is done in contrast to testing the entire software program at once. The goal of module testing is not to prove the proper working of the module. Rather, module testing shows the presence of errors in a module.

Test teams frequently turn to module testing when testing integrated and complicated applications. The modular pattern provides the simplest testing roadmap. This testing methodology also incorporates both composition and organizing tests, which makes it a compelling choice.

Making a module unit regression test suite, however, requires a fair amount of time. But this methodology gives full application test inclusion to use in regression testing and in additional functional testing. It also introduces parallelism into the testing procedure by evaluating several modules at the same time. This makes a perfect, powerful base for automated test development.

Differentiating Module and Unit Testing

A typical source of confusion for new software testers is the difference between unit testing and module testing. Generally, unit tests are a collection of tests composed by an engineer during the software development life cycle. Module tests are a collection of tests composed by a tester after a developer wrote some code.

There are numerous special cases to this generalization yet the key point is that unit testing is fundamentally a development-related activity, while module testing is essentially a testing-related activity. At the point when a developer makes unit tests during development, this methodology is sometimes referred to as test-driven development.

So which is better, module or unit testing? The two methodologies are correlative, not selective. Much of the time, neither unit testing nor module testing is adequate by themselves. The two methodologies should be utilized together.

While there is not a big difference between unit testing and module testing, it should be mentioned that in unit testing we use real objects and drivers as the function parameters. In contrast, module testing uses “fake” objects for testing. These “fake” objects are often referred to as mocks or stubs.

Module testing is also a part of integration testing (which we will cover in a later assignment). In this assignment we will talk about modules and their interaction between components or functions.

Increasing Complexity

Simple Classes

There is a category of classes that are very easy to test — those that only depend on primitive data types and have no relation to other business entities. In these cases, it is sufficient to create an instance of that class and test it by changing a property, calling a method, etc. and checking the expected state.

This is the simplest and most efficient way to test. Any good design starts from these classes, which are the lower-level “building blocks” from which more complex abstractions are then built.

Complex Classes

However, the number of classes that live in “isolation” is limited. Any sufficiently large piece of software interacts with other systems or software. Even if we isolate all the logic for working with a database (or any external software or service), we will eventually have to test the higher-level behavior.

So, how do you test these increasingly complex classes? Do you have to test the external entities as well? Do you have to recreate the entire collection of external software and services in your testing environment? No, this would be incredibly inefficient.

Instead, testers use mocks and stubs to imitate the higher-level behavior. This way you can quickly and easily test a class that interacts with these external entities.

Mocks and Stubs

Defining Mocks and Stubs

As previously discussed, mocks and stubs are used to replicate higher-level behavior for testing purposes. Mocks and stubs are often used together so one may think the two are synonymous, but there are important differences between the two.

Mock - is an imitation of an object. It replicates chosen component with the required precision and implements its interface or API. Mocks are only used in a test environment. For example, for backend tests, you can mock the repository so that it writes and reads data from RAM, while the real repository works with the database.

Stub - is an object with an interface of related components or API, but without logic. For example, an object whose method returns the same result on its call. Or with a predefined set of parameters: with a unit at the input, it returns one object, with a deuce, another.

Usually stubs are used in place of an API. For example, a `GET api/user/1` request will return the same user without the need to run a real backend.

An important distinction is that while you can read data with both mocks and stubs, only mocks can write data. In most cases, stubs are preferable because their logic does not change and they are faster to write than mocks.

Mocks and stubs (and to a lesser extent spies) are some of the most common ways to provide fake testing data. In this assignment, we are going to cover just mocks and stubs. Other forms of fake testing data are used in a fairly similar manner.

Writing Tests with Mocks

In this example, we are going to create a React component with two buttons and a counter. The buttons will either increment or decrement the counter. Then, we will test the counter using mock button clicks to simulate user interaction.

Setup

The examples on this page use React and Axios. Run the following commands in the terminal to set up the testing environment.

```
cd react-testing-tutorial && npm install
```

▼ **Installed Packages**

For a detailed look at the installed packages, here is the `package.json` file used for these examples. Feel free to copy it and use it locally with the `npm install` command.

```json
{
  "name": "react-testing-tutorial",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "axios": "^1.2.2",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --transformIgnorePatterns
        \"node_modules/(?!axios)/\"",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  },
  "devDependencies": {
    "jest-watch-typeahead": "^0.6.5"
  }
}
```

Creating the Component

Use the button below to open the `App.js` file. This is where we will create our React component.

Add the following code to the IDE. This creates two buttons and a counter. The buttons have `+` and `-` as their text. The counter increments or decrements a numerical value based on the button pressed.

```
import React, {
  useState
} from "react";
const Counter = () => {
  const [counter, setCounter] = useState(0);
  const incrementCounter = () => {
    setCounter((prevCounter) => prevCounter + 1);
  };
  const decrementCounter = () => {
    setCounter((prevCounter) => prevCounter - 1);
  };
  return ( < > < button data-testid = "increment"
    onClick = {
    incrementCounter
    } > + < /button> < p data-testid = "counter"> {
    counter
  } < /p> < button disabled data-testid = "decrement"
  onClick = {
    decrementCounter
  } > - < /button> </ > );
};
export default Counter;
```

Creating the Test

Now that our simple React component is made, we are ready to test it.
Open the `App.test.js` file, which will contain our tests.
Verify that you are using a version of Node that is greater than 14.04 with the `node -v` command in the terminal. This project uses React. However, all of the scaffolding for you has already been done.

▼ React scaffolding

The scaffolding for React was done using the command below. You would need to do something similar if you want to work with React on your own system.

```
npx create-react-app react-testing-tutorial
```

Create two tests; one to test the increment button and another to test the decrement button. Notice how we use `fireEvent.click` to create a mock button click for each button.

```
import { render, fireEvent, screen } from "@testing-
        library/react";
import Counter from "./App.js";

//test block
test("increments counter", () => {
  // render the component on virtual dom
  render(<Counter />);
  //select the elements you want to interact with
  const counter = screen.getByTestId("counter");
  const incrementBtn = screen.getByTestId("increment");
  //interact with those elements
  fireEvent.click(incrementBtn);
  //assert the expected result
  expect(counter).toHaveTextContent("1");
});

test("decrements counter", () => {
  // render this component again just to be sure test is
        launched from the default state
  render(<Counter />);
  //now we select decrement button
  const counter = screen.getByTestId("counter");
  const decrementBtn = screen.getByTestId("decrement");
  fireEvent.click(decrementBtn);
  //assert the expected result
  expect(counter).toHaveTextContent("0");
});
```

Run the test; all cases should pass.

Exploring Stubs in Sinon

Stubs let you easily isolate a component you are testing from other components that it calls. A stub is a small piece of code that takes the place of another component during testing. The benefit of using a stub is that it returns consistent results, making the test easier to write. And you can run tests even if the other components are not working yet.

To use stubs, you have to write your component so that it uses only interfaces, not classes, to refer to other parts of the application. This is a good design practice because it makes changes in one part less likely to require changes in another. For testing, it allows you to substitute a stub for a real component.

Setup

The examples on this page use a combination of Mocha, Chai, and Sinon instead of Jest for testing. Run the following commands in the terminal to set up the testing environment.

```
cd userComments && npm install
```

▼ **Installed Packages**

For a detailed look at the installed packages, here is the `package.json` file used for these examples. Feel free to copy it and use it locally with the `npm install` command.

```json
{
  "name": "userComments",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test" : "mocha index.test.js",
    "test_1": "mocha indexstubs.test.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "request": "^2.88.2"
  },
  "devDependencies": {
    "chai": "^4.3.7",
    "mocha": "^10.2.0",
    "sinon": "^15.0.1"
  }
}
```

Create the Index File

The `index.js` file is the entry point for the project. Click the link to open it.

Create the function `getCommentsById` which takes the post ID number and requests the first three comments for that post. Just as before, we will use JSON Placeholder as the source for the comments.

```
const request = require('request');

const getCommentsById = (id) => {
    const requestUrl =
        `https://jsonplaceholder.typicode.com/posts/${id}/commen
        ts?_limit=3`;
    return new Promise((resolve, reject) => {
        request.get(requestUrl, (err, res, body) => {
            if (err) {
                return reject(err);
            }
            resolve(JSON.parse(body));
        });
    });
};

module.exports = getCommentsById;
```

Creating Tests without Stubs

Open the test file for `index.js` with the link below so we can create some tests for the `getCommentsById` function.

Let's first create a test **without** any stubs. Since we are using Chai instead of Jest, the syntax is slightly different. However, you still have the same basic concepts. The test requests information from the first post. It checks that 3 comments were returned. In addition, each comment should have the properties `id`, `name`, and `email`.

```
const expect = require('chai').expect;
const getCommentsById = require('./index');

describe('withoutStub: getCommentsById', () => {
    it('should getCommentsById', (done) => {
        getCommentsById(1).then((comments) => {
            expect(comments.length).to.equal(3);
            comments.forEach(comment => {
                expect(comment).to.have.property('id');
                expect(comment).to.have.property('name');
                expect(comment).to.have.property('email');
            });
            done();
        });
    });
});
```

Run the test; all cases should pass. Notice the time it takes for the test to complete. Your results should look something like this:

```
> mocha index.test.js



  withoutStub: getCommentsById
    ✔ should getCommentsById (75ms)


  1 passing (82ms)
```

Creating Tests with Stubs

Open the test file once again with the link below.

Before the tests run, create a stub with Sinon that yields 3 objects, each one representing a comment for a post. The stub will be used in place of requesting data from JSON Placeholder.

```
const expect = require('chai').expect;
const request = require('request');
const sinon = require('sinon');
const getCommentsById = require('./index');

describe('with Stub: getCommentsById', () => {
    before(() => {
        sinon.stub(request, 'get')
            .yields(null, null, JSON.stringify([
                {
                    "postId": 1,
                    "id": 1,
                    "name": "user1",
                    "email": "user1@fake.com",
                    "body": "lol"
                },
                 {
                    "postId": 1,
                    "id": 2,
                    "name": "user2",
                    "email": "user2@fake.com",
                    "body": "lollol"
                },
                {
                    "postId": 1,
                    "id": 3,
                    "name": "user3",
                    "email": "user3@fake.com",
                    "body": "lollollol"
                }
            ]));
    });

    after(() => {
        request.get.restore();
    });

    it('should getCommentsById', (done) => {
        getCommentsById(1).then((comments) => {
            expect(comments.length).to.equal(3);
            comments.forEach(comment => {
                expect(comment).to.have.property('id');
                expect(comment).to.have.property('name');
                expect(comment).to.have.property('email');
            });
            done();
        });
    });
});
```

Run the test; all cases should pass. Notice the time it takes for the test to complete. It should be much faster now. Your results should look something like this:

```
> mocha indexstubs.test.js



  with Stub: getCommentsById
    ✔ should getCommentsById


  1 passing (12ms)
```

Exploring Mocks

Setup

The examples on this page use the Jest testing framework. Run the following commands in the terminal to set up the testing environment.

```
cd exploring-mocks && npm install
```

▼ Installed Packages

For a detailed look at the installed packages, here is the `package.json` file used for these examples. Feel free to copy it and use it locally with the `npm install` command.

```json
{
  "name": "exploring-mocks",
  "version": "1.0.0",
  "description": "",
  "main": "mapper.js",
  "scripts": {
    "test": "jest"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "jest": "^29.3.1"
  }
}
```

Mock Functions

Assume we have the following function `mapper`. It takes an array and a function as parameters. The function iterates over the array, passing each element to the function. How would we test this? The `mapper` function depends on another function which we do not have. We only have the `mapper` function.

```
function mapper(arr, func) {
  arr.forEach(element => {
    func(element);
  });
}

module.exports = mapper;
```

This is where mocks come into play. We are going to create a mock function to pass to `mapper` (along with an array) so that we can conduct tests. Start by opening the `mapper.test.js` file with the link below.

We are going to import the `mapper` function for testing as we have done in the past. We are also going to create the `mockFunc` function that takes a value and returns that value multiplied by `2`. This is the mock function we will be using for our testing. Notice that the function was created with `jest.fn`.

```
const mapper = require('./mapper');

const mockFunc = jest.fn(x => 2 * x);
```

Now, add a test to the file. The first thing the test should do is call the `mapper` function, passing it the array `[5, 10, 15]` and `mockFunc`. Because `mockFunc` was created with Jest, it has a special `mock` property. Log this property to the console and run the test.

```
test('Test mapper', () => {
    mapper([5, 10, 15], mockFunc);
    console.log(mockFunc.mock)
});
```

Jest provides us with all kinds of information about the mock function calls. The `calls` key contains an array of the values passed to `mockFunc` each time it was called. The `results` array contains the objects for each time `mockFunc` was called.

```
{
  calls: [ [ 5 ], [ 10 ], [ 15 ] ],
  contexts: [ undefined, undefined, undefined ],
  instances: [ undefined, undefined, undefined ],
  invocationCallOrder: [ 1, 2, 3 ],
  results: [
    { type: 'return', value: 10 },
    { type: 'return', value: 20 },
    { type: 'return', value: 30 }
  ],
  lastCall: [ 15 ]
}
```

Testing with Mocks

Open the test file once again with the link below and update the test with the code below.

We are going to use the information store in the `mock` property to write some test cases. We expect `mockFunc` to run three times (line 8). The first time it runs, we pass `5` to `mockFunc` and it returns `10` (lines 10 and 11). On the second run, we pass it `10` and receive `20` (lines 13 and 14). Finally, we pass `mockFunc` the value `15` and receive `30` (lines 16 and 17).

```javascript
const mapper = require('./mapper');

const mockFunc = jest.fn(x => 2 * x);

test('Test mapper', () => {
    mapper([5, 10, 15], mockFunc);

    expect(mockFunc.mock.calls).toHaveLength(3);

    expect(mockFunc.mock.calls[0][0]).toBe(5);
    expect(mockFunc.mock.results[0].value).toBe(10);

    expect(mockFunc.mock.calls[1][0]).toBe(10);
    expect(mockFunc.mock.results[1].value).toBe(20);

    expect(mockFunc.mock.calls[2][0]).toBe(15);
    expect(mockFunc.mock.results[2].value).toBe(30);
});
```

Run the test; all cases should pass.