Unit Testing

What are we going to cover

What is unit testing?

The Jest unit testing framework

AirBnB Enzyme
- Shallow rendering

@testing-library/react

Snapshot testing

What is unit testing?

"In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use."

-- Wikipedia --

Jest

Jest is a JavaScript unit testing framework

- Used by Facebook to test services and React applications

Not a requirement for React

- You can use other test frameworks/runners
  - Mocha
  - Karma
  - Jasmine

The default choice with React-Create-App

Jest has nice features

Fakes the **DOM** for tests

- Using JSDom

Runs tests in **parallel**

Works with **asynchronous** code

- Both promises and async/await

Reruns **only relevant tests** in development mode

- Faster feedback on large projects

Can do **snapshot testing**

- Guard against unexpected or accidental changes in components

Generate **coverage reports**

Creating a basic unit test

Create files with the name **<unit>.test.js**

- Or create a **__tests__** folder and put your tests there

Create a unit test by calling the **test()** function

- The first parameter is the test name
- The second parameter is the code under test

The **it()** function is an alias for test()

- Create-React-App uses it() for the default test it creates

A basic unit test

```
test("The calculator adds 2 + 3 = 5", () => {
  const calculator = new Calculator();
  const sum = calculator.add(2, 3);
  expect(sum).toBe(5);
});
```

Grouping tests together

The **describe()** function can optionally be used to group related unit tests

- Useful for creating a number of related test

Can be nested when required

Grouping tests together

```
describe("The calculator", () => {
  describe("adds", () => {
    test("2 + 3 = 5", () => {
      const calculator = new Calculator();
      const sum = calculator.add(2, 3);
      expect(sum).toBe(5);
    });
  });
  describe("subtracts", () => {
      // ...
  });
});
```

Expectations

The **expect()** function is used to verify that expected conditions are met

- Provides a number of matchers

Use **resolves** or **rejects** to resolve promises before asserting the value

- Can also be used with **async/await**

Can also be extended with **custom matchers** if needed

Using matchers to assert values

Matchers can be used to asserts results with expect()

There are **many matchers** available

- toBe() is a simple === comparison
- toEqual() tests if two objects are semantically the same
- toThrow() tests is an error is thrown
- toHaveProperty() checks if the specified property has a given value
- toMatch() validates a string against a regular expression
- toMatchObject() validates an object against a subset of that object
- See the documentation for the complete list

All matchers can be negated by prefixing with **.not.**

- Example: expect(1).not.toBe(2)

Validate a subset of an object

```
const getMovie = id => ({
  id,
  title: "Kill Bill: Volume 1",
  director: "Quentin Tarantino"
});

test("Can load Kill Bill", () => {
  const movie = getMovie(6534);

  expect(movie).toMatchObject({
    id: 6534,
    title: "Kill Bill: Volume 1"
  });
});
```

Setup and Teardown functions

Share common setup code between using the **beforeEach()** function

- Will be executed before each test executes

Use the **beforeAll()** function to run common setup code just once

The **afterEach()** or **afterAll()** functions can be used to clean up

Create a new calculator before each test

```
describe("The calculator", () => {
 let calculator;
 beforeEach(() => {
  calculator = new Calculator();
 });
 test("adds 2 + 3 = 5", () => {
  const sum = calculator.add(2, 3);
  expect(sum).toBe(5);
 });
});
```

Asynchronous tests

Jest can execute and test **asynchronous code**

Return a **Promise** from the test

- Use a then() function to execute matchers

Write an **async** test and use the **await** keyword

- Use the normal test pattern

Asynchronous test example

```
function doAsyncWork(value) {
  return new Promise(resolve =>
    setTimeout(() => resolve(value), 123));
}

test("Will eventually return 1", () => {
  return doAsyncWork(1)
    .then(result => expect(result).toBe(1));
});

test("Will eventually return 2", async () => {
  const result = await doAsyncWork(2);
  expect(result).toBe(2);
});
```

Mocking dependencies

Jest can mock **individual functions** or **complete modules**

Create an individual mock function using **jest.fn(implementation)**

Use **jest.mock(moduleName, factoryFn, options)** to mock a complete module

- Mock modules are **hoisted** above the ES2015 imports

Mocking an HTTP fetch request

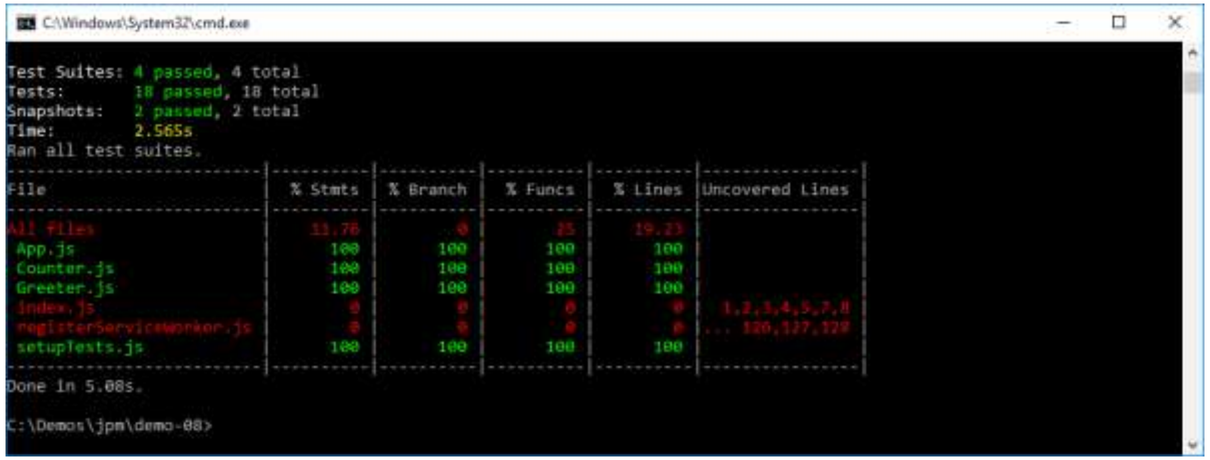
```
function fetchMovie(id) {
  return fetch(`/api/movies/${id}`).then(rsp => rsp.json());
}
describe("Can do an HTTP fetch", () => {
  beforeAll(() => {
    global.fetch = jest.fn(() =>
      Promise.resolve({
        json: () => Promise.resolve({ id: 6534, title: "Kill Bill: Volume 1" })
      })
    );
  });

  test("for Kill Bill", async () => {
    expect(await fetchMovie(6534)).toMatchObject({
      id: 6534,
      title: "Kill Bill: Volume 1"
    });
  });
});
```

Code coverage

Jest can generate a **coverage report** of the code under test

- Runs all the tests once and prints a report to the console

Use the **--coverage** option when starting Jest



C:\Windows\System32\cmd.exe

```
Test Suites: 4 passed, 4 total
Tests:       18 passed, 18 total
Snapshots:   2 passed, 2 total
Time:        2.565s
Ran all test suites.
---------------------------|----------|----------|----------|----------|-------------------|
File                       |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Lines   |
---------------------------|----------|----------|----------|----------|-------------------|
All files                  |    11.76 |        0 |       25 |    19.23 |                   |
 App.js                    |      100 |      100 |      100 |      100 |                   |
 Counter.js                |      100 |      100 |      100 |      100 |                   |
 Greeter.js                |      100 |      100 |      100 |      100 |                   |
 index.js                  |        0 |        0 |        0 |        0 |   1,2,3,4,5,7,8   |
 registerServiceWorker.js  |        0 |        0 |        0 |        0 |... 120,127,128    |
 setupTests.js             |      100 |      100 |      100 |      100 |                   |
---------------------------|----------|----------|----------|----------|-------------------|
Done in 5.08s.

C:\Demos\jpm\demo-08>
```

Unit testing
React components

Enzyme

A testing utility from AirBNB that provides additional functionality

- Install enzyme and the enzyme adaptor for the appropriate version of React

Only render the actual component, not any of its children

- Treat a component as a unit instead of a tree
- Using the **shallow()** function

Create a **setupTests.js** for Jest to configure Enzyme

```
import Enzyme from "enzyme";
import Adapter from "enzyme-adapter-react-16";
Enzyme.configure({ adapter: new Adapter() });
```

Enzyme Shallow rendering

Shallow rendering lets you treat a component as a single unit

- Any child component in the render() will not be executed

The rendered component can be **manipulated** in a number of ways

- Find a DOM element and their properties
- Find a DOM element and simulate an event
- Get or update the components state
- Change the value of the properties

Shallow testing a component

```
import React from "react";
import { shallow } from "enzyme";
import Greeter from "./Greeter";

describe("The Greeter component", () => {
  let greeter;
  beforeEach(() => {
    greeter = shallow(<Greeter name="Maurice" />);
  });
  it("should render the name in lowercase", () => {
    expect(greeter.find("span").text()).toBe("maurice");
  });
  it("should increment the count in click", () => {
    greeter.find("button").simulate("click");
    expect(greeter.state("count")).toBe(5);
  });
});
```

Enzyme Full rendering

Full rendering lets you test the interaction between components

- All child components will be rendered
- And mounted in the DOM

The API is almost the same as with Enzyme shallow rendering

Full rendering a component

```
import { mount } from "enzyme";

describe("The MovieEditor component", () => {
 it('should call onChange when the movie title is changed', () => {
    const fn = jest.fn();
    const component = mount(
      <MovieEditor movie={movie} onChange={fn} />
    );

    component
      .find('input[value="Kill Bill"]')
      .simulate('change', { target: { value: 'Star wars' } });

    expect(fn).toHaveBeenCalledWith({ prop: 'title', value: 'Star wars' });
  });
});
```

Using @testing-library/react

The **@testing-library/react** npm package is a popular alternative to Enzyme

Renders components into **real** DOM elements

- Using JSDOM in Jest

No shallow rendering

- Make sure that props passed **between components** are correct
- Use jest.mock() to prevent large component trees from being rendered

Query for DOM elements based on what **the user would see**

- Text, value etc
- Use **data-testid** only for non visible elements like <form />

Note: Make sure to call **cleanup()** after each test

Testing a form submit

```
import { render, fireEvent, cleanup }
  from '@testing-library/react';

test('should save the movie', () => {
  const onSave = jest.fn();

  const wrapper = render(
    <MovieEditor movie={movie} onSave={onSave} />
  );

  fireEvent.submit(
    wrapper.getByTestId('movie-form'));

  expect(onSave).toHaveBeenCalled();
});
```

Snapshot testing

Jest supports **regression testing** using snapshot testing

- Take a snapshot of a component in a given state and make sure that remains the same

Use the **react-test-renderer** to create a JSON representation of a component

- Use the **toMatchSnapshot()** matcher to verify that the snapshot hasn't changed

The first time the test is run as snapshot is created in the **__snapshots__** folder

- These should be committed to the source repository

Snapshot testing

```
import React from "react";
import reactTestRenderer from "react-test-renderer";
import Greeter from "./Greeter";

describe("The Greeter component", () => {
  it("renders the same tree", () => {
    const tree = reactTestRenderer
      .create(<Greeter name="Maurice" />)
      .toJSON();

    expect(tree).toMatchSnapshot();
  });
});
```

Conclusion

Create React App enables testing by default

- It uses the Jest framework and runner

AirBnB Enzyme is a great extra library for testing

- Using the shallow rendering makes it easy to treat a component as a single unit

Using @testing-library/react results in unit tests that are closer to what a user will do

Snapshot testing help prevent accidental changes

- Use when a component is considered done