COLLEGIO TECHNOLOGIES INC.

TESTING AND RELEASING YOUR APP

INTRODUCTION: TESTING OUR APP

- Now that we have our app built and working, we want to make sure that it will continue working as we make changes in the future.
  - However, what happens if the change breaks something that you weren't assuming would be broken? You may not always test everything in the app each time you release a new feature.

- By setting up an automated test suite for your app, you can easily verify that the entire application works by running a single command.
  - Repeating tests across the entire system usually takes seconds, instead of the amount of time it would take to test each component manually.

TESTING YOUR APP IS IMPORTANT!

- Sometimes (especially when you're on a tight timeline), it may seem like writing tests for your app is unnecessary overhead, so why not just test manually?

- However, writing tests is not a waste of time!
  - Test cases are only written once, and can be run as many times as needed.
  - Automated tests remove the human-error aspect of manual testing, ensuring consistent tests and output.
  - As the system grows and manual testing becomes more cumbersome, automated tests provide a drastic decrease in overall performance hits.
  - As mentioned, you can validate that any features you add or update are not breaking the app ***system-wide***.
  - You have much more confidence in your app's ability to work as expected.

TEST CASES

- Test Cases are functions that are designed to test specific parts of your app, such as:
  - Components
  - Actions
  - Selectors
  - Reducers
  - Interactions (ie. form submissions)
- Test cases are provided a given set of props, and validate that the output which is rendered/returned by the test is accurate.
- The framework that we'll be using for testing our app is called **Jest**.

WHAT IS JEST?

- Jest is an automated testing framework that is designed to be used with JavaScript, with a focus on simplicity.
  - Developed by Facebook
  - Official site: **https://jestjs.io/**

- In addition to Jest, there are multiple other testing frameworks out there for many different languages, such as Jasmine, Mocha, Karma, PHPUnit, and many others!

- We are using Jest for our framework, because in addition to being a JavaScript testing framework, it is also designed to work very well with React apps.

INSTALLING AND USING JEST

- You can install Jest via Yarn or NPM:
  - **yarn add --dev jest**
  - **npm install --save-dev jest**
- Once installed, Jest can be run from the command line, similar to Webpack or live-server
  - **jest**
  - **jest --watch**
- When run from the command line, Jest will look for any files within your app that contain the ***test.js*** extension.
  - To keep with our folder structure, we will be saving our tests inside of a **tests/** folder.
  - If you install and run jest without creating any test files, you will get a passing test with a 'No tests found' message.

WRITING TESTS

- Once our test file is created, we can test parts of our app by using Jest's built-in **test()** function.
- The test() function accepts two arguments:
  - A label that you can use to identify the test.
  - A function that is used to perform the test.
- Once we have created the test function, we can use the **expect()** function to verify that the returned result from our test is correct.
  - The expect() function has a large variety of functions that can be used to validate output. You can view the documentation for the expect() function at https://jestjs.io/docs/en/expect.
- Let's create a simple test that will multiply two numbers:
  - Create a **multiply.test.js** file inside of the **tests/** folder.
  - Create your multiply function that will accept 2 arguments and return their multiplied result.
  - Return the results of your test using **test()**.

EXAMPLE: SIMPLE MULTIPLICATION TEST

```
const multiply = (a, b) => {
    return a * b;
};

test('trying to multiply 2 numbers', () => {
    const result = multiply(3, 4);

    expect(result).toBe(12);
});
```

TESTING: ACTION GENERATORS

- We will be storing the tests for our Action Generator functions inside of the **/src/tests/actions** folder.
    - Your test names should be the same as your action generator names, except they should end in **.test.js**.
- Inside of your test file, you can import all of the file's actions that you will be testing.
- Similar to how we set up our simple test, we can do something similar for our action generators.
    - Please note, we cannot use ***toBe()*** to compare objects (which most of our action generators will return), because toBe() uses === to compare the values, and this will never equate to true for objects.
    - For objects, you will use the **toEqual()** function with expect().
    - Dynamic values, such as IDs generated by **uuid**, can be tested by using the **.any()** function of expect(). You should pass in the type of value that you are expecting, such as a number or string.
    - If you have default values, you should run tests to ensure that both your default values will work when used, as well as entered values.

EXAMPLE: ACTION GENERATOR TESTS

```
import { setFilterText, setFilterType, sortByName, sortBySkill } from '../../actions/filters';

test('should set filter text', () => {
    const testText = 'some text';
    const result = setFilterText(testText);

    expect(result).toEqual({
        type: 'SET_FILTER_TEXT',
        text: testText
    });
});

test('should set default filter type', () => {
    const result = setFilterType();

    expect(result).toEqual({
        type: 'SET_FILTER_TYPE',
        sport_type: 'all'
    });
});

test('should set specified filter type', () => {
    const testType = 'basketball';
    const result = setFilterType(testType);

    expect(result).toEqual({
        type: 'SET_FILTER_TYPE',
        sport_type: testType
    });
});
```

TESTING: SELECTORS

- Selector tests are typically more complicated than action generator tests, as they tend to involve a range of values, and returning a subset of those while potentially performing operations on the returned result.
- For selector tests, we will manually build a test dataset that will be passed into our selectors.
  - This dataset will be used with all tests, so should not be hard-coded into individual tests.
  - By using this dataset, we can test all of our support selector features (sorting, filtering, etc.) against a consistent input.
  - If you are using this same dataset across multiple tests, it should be stored in it's own file, and then exported into the applicable tests. For our project, we'll be saving any common datasets in the **/tests/fixtures** folder.

EXAMPLE: SELECTOR TESTS

```
import players from '../fixtures/players';
import filters from '../fixtures/playerFilters';
import playerSelector from '../../selectors/players';

test('testing the player selector with no filters', () => {
    const result = playerSelector(players, filters);

    expect(result).toEqual([
        players[2],
        players[3],
        players[4],
        players[1],
        players[0]
    ]);
});

test('testing the player selector with search text set', () => {
    const result = playerSelector(players, {
        ...filters,
        text: 'brees'
    });

    expect(result).toEqual([
        players[2],
        players[4]
    ]);
});
```

TESTING: REDUCERS

- Reducer tests, similar to action generator and selector tests, will be passed in a specific set of values, and a specific result is expected as data.
- In addition to testing basic functionality, we also must test that our reducers are correctly initializing when redux starts.
    - Redux dispatches a special action for this, called **@@INIT**, which returns an object containing a ‘type’ attribute, set to ‘@@INIT’.
- When we are testing reducer functions, we are testing the returned state of each reducer call.
    - If you are testing changing a function to something that’s already a default value, you can set what the initial state is.

EXAMPLE: REDUCER TESTS

```
import filtersReducer from '../../reducers/filters';

test('should setup default filter values', () => {
    const state = filtersReducer(undefined, { type: '@@INIT' });

    expect(state).toEqual({
        text: '',
        sport_type: 'all',
        skill_level: 'all',
        sort_by: 'name'
    });
});
```

```
test('should set filter text', () => {
    const testText = 'some text';
    const action = {
        type: 'SET_FILTER_TEXT',
        text: testText
    }
    const state = filtersReducer(undefined, action);

    expect(state).toEqual({
        text: testText,
        sport_type: 'all',
        skill_level: 'all',
        sort_by: 'name'
    });
});
```

TESTING COMPONENTS

INTRODUCTION: TESTING COMPONENTS

- Testing components is very different from testing functions.
  - For functions, we pass in data, and expect data returned.

- With components, we are concerned with multiple things when testing, such as:
  - What is rendering different situations, with different props.
  - Is the component reacting correctly to user input? (ie. form updates, filters)

- We will be using **Snapshot Testing** to test our components.
  - Snapshot testing involves taking a ‘snapshot’ of the structure of a React Component at a particular point in the test lifecycle, and validate it against an expected result.

TESTING COMPONENTS: SETUP

- Before we begin, we will need to install a few new packages:
- ***React Test Renderer***
  - **yarn add react-test-renderer@16.0.0**
  - Used to virtually render our components.
- ***Enzyme***
  - **https://enzymejs.github.io/enzyme/**
  - Created by AirBnB
  - Uses React Test Renderer to allow us to fully test our components, by allowing features such as input and submission validation
  - **yarn add enzyme@3.0.0 enzyme-adapter-react-16@1.0.0 raf@3.3.2**
- Enzyme to JSON
  - This will be used to help us prevent our snapshots from containing enzyme-specific code
  - **yarn add enzyme-to-json@3.0.1**
- However, once installation is complete, we still need to go through some additional steps.

TESTING COMPONENTS: SETUP

- Create a setup file
  - As per enzyme's docs, we will need to create a setup file in order to initialize our adapter for React.
  - This file will be **/src/tests/setupTests.js**

```
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({
    adapter: new Adapter()
});
```

TESTING COMPONENTS: SETUP

- Create a config file
  - The config file will be located in the root of your project, and will be called **jest.config.json**
  - The config file will be used to load our setup files, as well as set up our serializer.

- Once completed, we will need to update our package.json test script to **jest --config=jest.config.json**

```
{
    "setupFiles": [
        "raf/polyfill",
        "<rootDir>/src/tests/setupTests.js"
    ],
    "snapshotSerializers": [
        "enzyme-to-json/serializer"
    ]
}
```

TESTING COMPONENTS

- There are two types of rendering that can be performed for testing components:
  - **Shallow Rendering -** only renders the top-level component information, without rendering child components. Used for rendering simple components.
  - **Full DOM Rendering -** renders child components, and allows for more full-featured testing of user interactions and lifecycle events.

- We can use Enzyme to perform both types of rendering in our tests.

SHALLOW RENDER TESTING

- We use Enzyme's **shallow()** function to render our components.
  - The **shallow()** function accepts JSX as an argument, and returns a wrapper component that we can then run tests against.
  - Using enzyme, we can make assertions on several different things, such as the number of tags, or value of components.
- In addition to shallow(), some other useful functions that we will be using for our tests:
  - **.find():** we will use this to iterate through our components to find particular elements that we can test against
  - **.at():** used in conjunction with .find(), this function allows you to select a particular element if multiple are found
  - **.toMatchSnapshot():** used to create and test our current code against established snapshots of our component (the first time it is run, it creates the snapshot, and tests against the established snapshot with each subsequent test).

EXAMPLE: TESTING THE HEADER COMPONENT

```
import React from 'react';
import { shallow } from 'enzyme';
import Header from '../../components/Header';

test ('should render the header', () => {

    const wrapper = shallow(<Header>Test Header</Header>);

    expect(wrapper.find('h1').length).toBe(1);
    expect(wrapper).toMatchSnapshot();
});
```

TESTING SNAPSHOTS WITH DYNAMIC COMPONENTS

- In order to test components with dynamic values, we will have to provide the props for these components ourselves.

- We will do this by passing props directly into the components when they are initialized in the **shallow()** function.
    - If our component is using redux, we will need to make sure that, in addition to exporting the default component using connect(), we are also exporting the named component.

EXAMPLE: TESTING DYNAMIC COMPONENTS

```
import React from 'react';
import { shallow } from 'enzyme';
import { PlayersList } from '../../components/PlayersList';
import players from '../fixtures/players';

test('should render the players list with players', () => {
    const wrapper = shallow(<PlayersList players={players} />);
    expect(wrapper).toMatchSnapshot();
});

test('should render the players list no players', () => {
    const wrapper = shallow(<PlayersList players={[]} />);
    expect(wrapper).toMatchSnapshot();
});
```

MOCKING LIBRARIES WITH JEST

- When testing some components, we may be using 3rd party libraries that we have little control over, such as moment.js.
    - For instance, testing an Add Player function that sets the create date to the current date will always throw an incorrect test, since it will always be testing against a time in the past.
- In order to properly test our apps, we may need to mock these functions in order to have consistent tests.
- When mocking 3rd party libraries, we place these in the **__mocks__** folder.
    - For instance, with moment.js, we would place the mock inside of **/src/tests/__mocks__/moment.js**
- In order to prevent infinite looping inside of our mocked test, we will need to import the actual version of the library via **require.requireActual()**

```
const moment = require.requireActual('moment');

export default (timestamp = 0) => {
    return moment(timestamp);
}
```

TESTING INPUTS AND INTERACTIONS

- In order to test inputs, we will actually need a way to test these user interactions.
- We use the **.simulate()** Enzyme function to simulate user interactions, such as clicks, submits, changes, etc.
- **simulate()** accepts two arguments:
  - The type of interaction we're simulating
  - The argument that is passed to the event (called 'e' in most of our functions).
- We can apply the **.state()** function to our wrapper in order to determine the value of the state after simulating interactions. By passing the name to **.state()**, we can look at a particular piece of state.

EXAMPLE: TESTING INPUT AND INTERACTIONS

```
test('should set name on input change', () => {
    const value = 'Sidney Crosby';
    const wrapper = shallow(<PlayerForm />);
    expect(wrapper).toMatchSnapshot();
    wrapper.find('input').simulate('change', {
        target: { value }
    });
    expect(wrapper.state('name')).toBe(value);
    expect(wrapper).toMatchSnapshot();
});

test('should set sport type on select change', () => {
    const value = 'baseball';
    const wrapper = shallow(<PlayerForm />);
    expect(wrapper).toMatchSnapshot();
    wrapper.find('select').at(0).simulate('change', {
        target: { value }
    });
    expect(wrapper.state('sport_type')).toBe(value);
    expect(wrapper).toMatchSnapshot();
});
```

TEST SPIES

- Test Spies, also known as ***mock functions***, are used to test functions that are passed into your app.
- Test spies are initialized using **jest.fn()**
  - Once a test spy is declared, you can then test several different assertions, such as whether the function was declared, and what values the function was called with.

EXAMPLE: TEST SPIES

```
test('should trigger valid form submission', () => {
    const onSubmitSpy = jest.fn();
    const wrapper = shallow(<PlayerForm player={players[0]} onSubmit={onSubmitSpy}/>);

    wrapper.find('form').simulate('submit', {
        preventDefault: () => {}
    });

    expect(wrapper.state('error')).toBe('');
    expect(onSubmitSpy).toHaveBeenLastCalledWith({
        name: players[0].name,
        sport_type: players[0].sport_type,
        gender: players[0].gender,
        skill_level: players[0].skill_level,
        message: players[0].message
    });
});
```

TESTING FUNCTIONS USING DISPATCH()

- If we want to test any of our functions that are dispatching actions, we will need a way to extrapolate out the dispatch() command.
  - We can do this with **mapDispatchToProps**.

- The mapDispatchToProps argument, like mapStateToProps, is used as an argument in redux's **connect()** function to map dispatch calls to their own props.
  - Once dispatch statements are converted to props, they can then be tested by passing spies in place of the dispatched functions.

- Exercise: Let’s see how we can use mapDispatchToProps by converting our AddPlayersPage component.

EXAMPLE: MAPPING DISPATCH TO PROPS

```
export class AddPlayerPage extends React.Component {

    constructor(props) {
        super(props);
        this.onSubmit = this.onSubmit.bind(this);
    }

    onSubmit(player) {
        this.props.addPlayer(player);
        this.props.history.push('/');
    }

    render() {
        return (
            <div className="container">
                <h1>Add a Player</h1>
                <PlayerForm
                    onSubmit={this.onSubmit}
                />
            </div>
        );
    }
}

const mapDispatchToProps = (dispatch) => ({
    addPlayer: (player) => dispatch(addPlayer(player))
});

export default connect(undefined, mapDispatchToProps)(AddPlayerPage);
```

CONCLUSION: TESTING

- Congratulations, you now have the knowledge to be able to test your entire React app! You can be confident in the knowledge that your app works, and will continue to work as you make changes to it.
- One last thing we will be going through in the course is how we can optimize our app build, and create a simple node server to view our optimized production build.

RELEASING YOUR APP

SETTING UP WEBPACK FOR PRODUCTION

- When working in development mode, the build for our projects tend to be very large in size (sometimes >5MB).
  - This isn't something that we want our users to have to load when they're using our app.
- We can reduce the size of our builds by optimizing webpack for production.
  - When calling webpack, we will be specifying the **-p** flag to trigger the production build. This will slightly reduce the size of our file.
  - However, we will really see a change in our file size once we extrapolate out the source maps from our build. This can be done by specifying a different source map while in production.

CONDITIONALLY LOADING SOURCE MAPS BASED ON PRODUCTION VS. DEVELOPMENT

- We will need to make some slight modifications to our **webpack.config.js** file to determine if we are in production:
  - Our module.exports value will need to be converted from an object, to a function that returns the object. This will allow us to access environment variables, as well as conditionally set our source map.
  - Once we can determine if we are in production, we will load the 'source-map' module instead of 'cheap-module-eval-source-map'. The source-map module will create an external source map that is only loaded when a user opens the console.

- In addition, we will need to specify the environment variable in our webpack command:
  webpack -p --env production

CONFIGURE CSS TO LOAD SEPARATELY FROM SCRIPTS

- In order to further optimize our build, we are going to remove the CSS from our build, and load it inside of it's own .css file.

- In order to do this, we will need the **Mini CSS Extract Plugin**.
- **yarn add mini-css-extract-plugin**
- Once installed, we add the plugin to our webpack.config.js file, and use it within our rules to extract any found CSS to our specified file.

CONCLUSION: REACT

- That's it! We've covered the entirety of the React Lesson Materials! Unfortunately, we can only cover so much over the time allotted, and there are some things that need to be left out.

- Happy Coding!

RESOURCES

- **Lesson Video:** https://www.youtube.com/watch?v=v9LYUtzobNk
- **Lesson Code:** https://github.com/collegio/intro_react_testing_deployment
- **Completed Lesson Code:** https://github.com/collegio/intro_react_testing_deployment_completed
- **Jest:**
  - **Official Site -** https://jestjs.io/
  - **Code Documentation -** https://jestjs.io/docs/en/getting-started
  - **Expect() Documentation -** https://jestjs.io/docs/en/expect
  - **Mock functions with Jest -** https://jestjs.io/docs/en/mock-functions.html
- **React Test Renderer:**
  - **Official Docs:** https://reactjs.org/docs/test-renderer.html
- **Enzyme:**
  - **Official Docs -** https://enzymejs.github.io/enzyme/
  - **Enzyme CheatSheet -** https://devhints.io/enzyme
- **Production Webpack:**
  - **Official Production Webpack Guide -** https://webpack.js.org/guides/production/
  - **Devtool Docs -** https://webpack.js.org/configuration/devtool/
  - **Mini CSS Extract Plugin -** https://github.com/webpack-contrib/mini-css-extract-plugin

DISCLAIMER

- Copyright 2020 Collegio Technologies Inc.
- All content within this presentation has been compiled by Collegio Technologies Inc. All material has been sourced through open source resources and source code and as such conforms to Canadian copyright laws and regulations. All rights reserved.