POC: Unit testing / integration testing

Set up

Requirements

Codewise, some dependencies will be created. These all involve testing and mocking frameworks and are thoroughly tried and true.

1. ***Jest***
   This is the main unit testing, object mocking, stubbing, spying and coverage testing framework.
   It was designed by the team at Facebook that also created React.
   No installation or package is required, Jest is installed and configured out of the box to work with Create-React-App boilerplates. Transpile support for ES6 is also already configured.
2. ***Enzyme***
   This is a React testing framework, made by AirBnb. The main purpose for this framework is to simulate components being rendered and actions on those components being performed.
   It has the ability to render the single component (shallow) or deep render all involved components in the tree (mount).
   Installation / configuration is required. See below.
3. ***redux-mock-store***
   A small framework that is able to mock the Redux store so that integration tests can easily be performed.

Installation

1. ***Installing Jest***
   Already preinstalled as a module of the create-react-app boilerplate framework.
2. ***Installing Enzyme***
   *Install as NPM-package by using **npm install -D enzyme***
   Enzyme also requires an adapter as a middleware between the correct React version and the test framework.
   *Install as NPM-package by using **npm install -D enzyme-adapter-react-16** (the number varies based on the current React version)*
   Finally Enzyme for React 15 also requires the test utilities addon.
   *Install as NPM-package by using **npm install -D react-test-renderer@15**. This step is not required for React 16.*
3. ***Installing redux-mock-store***
   *Install as NPM-package using **npm install -D redux-mock-store***

Configuration

1. ***Configuring Jest***

   Jest is already pre-configured.

   Jest looks for files in the src/ folder containing ".test.js" in its filename.

   There are 2 paths to take:

   - *Create a new folder and name it tests. Recreate the entire folder structure and place test files where normal files should be.*

     This makes the test code isolated and easy to locate.

     But this also makes import reference paths potentially very long
   - *Create a new test file next to each .js or .jsx file.*

     This makes the test code very consistent with the source code and makes reference paths easy and short.

     But it also clutters production code a lot.

2. ***Configuring Enzyme***

   Enzyme looks for a file in the root folder (next to index.js), called ***setupTests.js****.*

   Here, we specify the React adapter that Enzyme should be using.

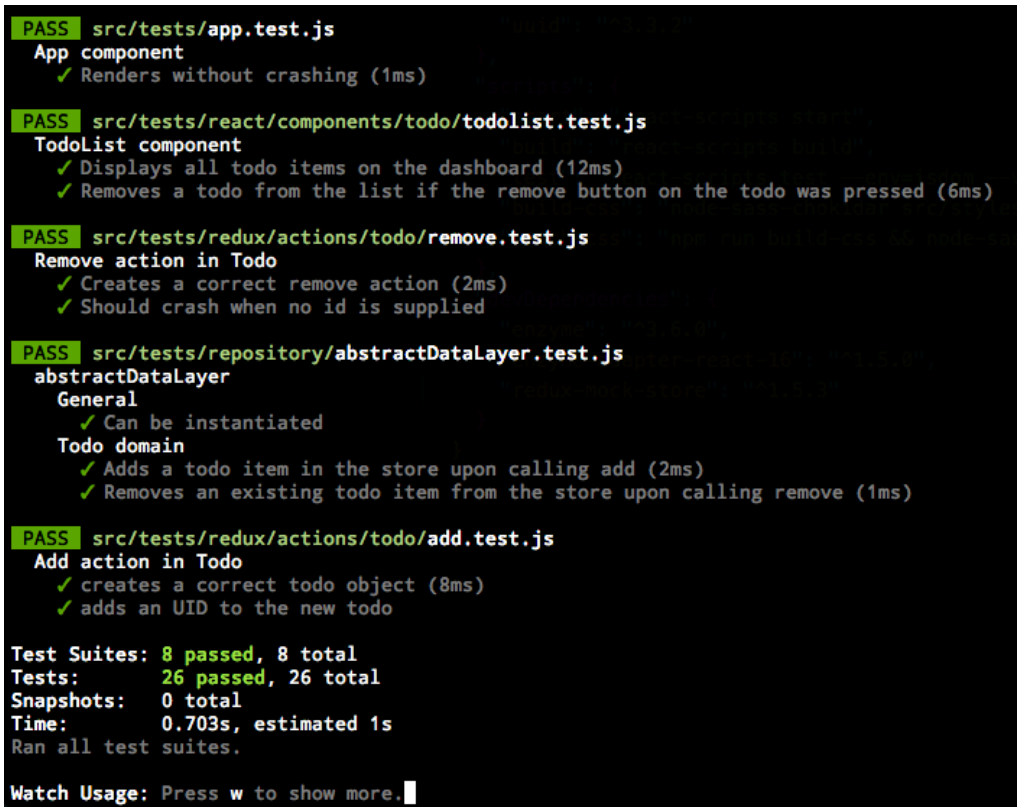
```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

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

Execution

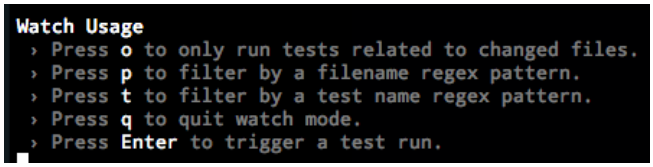
To execute testing, run an NPM command.
This command should be configured in the package.json file:

```
"scripts": {
    "test": "react-scripts test --env=jsdom --verbose",
  },
```

This puts the testing framework into a watch state. It will only rerun tests based on changed code in either the relevant source file or test file.



To rerun a complete set of tests, access the option menu from the terminal.
Press w to show options.
Press a to run all tests.



To execute coverage testing, run previous NPM command together with extra flags.

npm test -- --coverage

This will display a result of unit test coverage in the terminal.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	39.16	15.56	50	50.49	
src	8.06	0	15	14.71	
App.js	100	100	100	100	
diContainer.js	75	100	66.67	75	13
index.js	0	0	0	0	... 13,15,18,23,24
registerServiceWorker.js	0	0	0	0	... 36,137,138,139
setupTests.js	100	100	100	100	
src/react/components	50	100	0	50	
Navigation.jsx	50	100	0	50	5
src/react/components/todo	80	50	83.33	77.78	
TodoItem.jsx	100	100	100	100	
TodoList.jsx	75	50	75	71.43	10,16
src/react/routes	76.92	100	62.5	76.92	
CanvasPage.jsx	50	100	0	50	4
Home.jsx	100	100	100	100	
ResponsivePage.jsx	50	100	0	50	4
ScalablePage.jsx	50	100	0	50	4
src/redux	0	0	0	0	
reducers.js	0	0	0	0	1,3,4,6,8
store.js	0	0	0	0	1,3,5,6
src/redux/actions/todo	100	100	100	100	
add.js	100	100	100	100	
remove.js	100	100	100	100	
src/redux/reducers	60	66.67	66.67	61.54	
todoReducer.js	100	100	100	100	
userReducer.js	0	0	0	0	1,5,6,8,10
src/repository	100	100	100	100	
abstractDataLayer.js	100	100	100	100	
src/tests/mocks	77.78	100	75	77.78	
abstractDataLayerMock.js	77.78	100	75	77.78	16,17

To view more info, Jest will also create a new folder in the root of the application, called coverage.
Open coverage/lcov-report/index.html to view more options about the covering.

All files

39.16% Statements 56/143 **15.56%** Branches 7/45 **50%** Functions 26/52 **50.49%** Lines 52/103

File		Statements		Branches		Functions		Lines	
src		8.06%	5/62	0%	0/25	15%	3/20	14.71%	5/34
src/react/components		50%	1/2	100%	0/0	0%	0/1	50%	1/2
src/react/components/todo		80%	8/10	50%	1/2	83.33%	5/6	77.78%	7/9
src/react/routes		76.92%	10/13	100%	0/0	62.5%	5/8	76.92%	10/13
src/redux		0%	0/16	0%	0/10	0%	0/2	0%	0/9
src/redux/actions/todo		100%	8/8	100%	2/2	100%	2/2	100%	8/8
src/redux/reducers		60%	9/15	66.67%	4/6	66.67%	2/3	61.54%	8/13
src/repository		100%	8/8	100%	0/0	100%	6/6	100%	6/6
src/tests/mocks		77.78%	7/9	100%	0/0	75%	3/4	77.78%	7/9

Clicking on a file also displayed parts that aren't covered by unit tests.

Refactoring / Writing testable code

Dumb, testable components

In order for the application to become testable, some key coding paradigms need to be kept in scope.
One of the most important ones is to keep each code-file (be it a class or a functional component) as dumb as possible.
Dumb component: a component with a single responsibility that should own as little dependencies as possible and should not care where injected dependencies come from. It should work isolated and on its own as much as possible.

Besides this, we should also work towards having as much pure functions as possible.
Pure function: Receives input and gives output. The output is purely defined on its input (always given the same input should always give the same output).
A pure function should have a single responsibility and should not mutate other parts of the component or the data (no side effects).

To avoid a component looking for other functionality or third-party-code (= dependencies), we should implement a programming paradigm, called Dependency Injection.
In typed languages (such as java and c#), this is very often paired with a second paradigm, called Inversion Of Control.
Both of these paradigms are based on the SOLID principles that every OOP object should follow.
Because Javascript is not a typed language, it cannot easily implement the Inversion Of Control paradigm: due to its missing interfaces, it cannot be enforced and would therefor create bad code.

Many of the SOLID principles SHOULD be followed in order to both write clean, easy to read code AND to create a testable codebase.

S: SRP (single responsibility principle):
Make your components dumb and give them one job and not a list of tasks.

O: OCP (open for extension, closed for modification principle):
Components should not be modified but should be able to be extended from. (inheritance and encapsulation)

L: LSP (Lisskov substation principle):
Objects should be able to be easily replaced by other objects and the code should still work the same. This is a requirement for dependency injection and mocking in unit tests.

I: ISP (Interface segregation principle):
This is tightly coupled with IOC and cannot easily be enforced in Javascript.

D: DIP (Dependency inversion principle):
The component should be dependent on abstractions and not implementations. Instead of creating dependencies in components, abstract away these dependencies and inject these abstractions in the object.
This is also tightly-coupled with LSP.

Creating an abstraction with dependency injection

The first step is to create an abstraction between certain layers. For instance, between the redux store and the react component. When sending an action from the React component all the way to the store, we are making React completely dependent on Redux. It is now tightly coupled, which also means that React KNOWS Redux exists, making it a smart component.
Not only React components access the store: global function logic also makes use of the Redux functionality, meaning there is also a coupling on other components. Replacing or excluding Redux from the codebase will become impossible if there is no form of abstraction present.

The next code block is an example of an abstraction layer. In this example an abstraction for Redux is created. It is now a generic data layer. The data layer itself SHOULD know Redux, because it is its single responsibility to communicate with the store.
However, the store object itself is pre-configured and injected in the constructor so there is no bottom-to-top dependency.

```
import { add } from '../redux/actions/todo/add';
import { remove } from '../redux/actions/todo/remove';

//implementation is hidden outside of this file
const todosAdd = (store) => (todo) => { //CURRYING
    store.dispatch(add(todo));
}

const todosRemove = (store) => (id) => { //CURRYING
    store.dispatch(remove(id));
}

//only this class is exposed
class abstractDataLayer {
    constructor(store) {
        this._store = store;
    }

    get todo() {
        return ({
            add: todosAdd(this._store),
            remove: todosRemove(this._store)
        });
    }
}

export default abstractDataLayer;
```

To the abstractDataLayer object it should not matter how store exists or how it is created. Store is injected by an orchestrating component that has the single job of configuring and setting up the application (see later in this document).

By also injecting the store object into the 2 functions (add and remove) and, by currying, not exposing it to the outside, now a dumb component can simply call the abstractDataLayer class to add a Todo item by: abstractDataLayer.todo.add(todoObject);

Store and Redux are completely extracted from the knowledge of the dumb component. It should only know there is a datalayer that it has access to.

Bootstrapping the abstracted components

There is one last element missing to make the exercise complete:
In order to be compliant with the Lisskov Substition Principle and further avoid dependencies, we will need to be able to access a general component, who's only job is to pass through the required abstract component.

In the example above, there is still a dependency from the child component to a higher component: the component that is calling the abstractDataLayer class, still needs to refer to it and therefor creates a dependency.

With a bootstrapper object, the only dependency that will be created, is the one towards the bootstrapper.
If we make sure it is a static class, then it can get configured only once and sends back already living components.
This means that there is no need to reïnstantiate abstractDataLayer class (and the requirement to know and pass along its dependencies) outside of the configuration part.

The code below is an example of a bootstrapper class:

```
class diContainer {
    static _store = null;
    static _dataLayer = null;

    static bootstrap(store, datalayer) {
        this._store = store;
        this._dataLayer = datalayer;
    }

    static get store() {
        return this._store;
    }

    static get dataLayer() {
        return this._dataLayer;
    }
}

export default diContainer;
```

The configuration of the bootstrapper class happens inside of one of the higher components (such as index.js). This higher component is orchestrating its children objects and disallows bottom-to-top dependencies.

```
diContainer.bootstrap(store, new abstractDataLayer(store));
```

As seen in this line of code, abstractDataLayer itself is also injected with store. This means that the data layer class is also loosely-coupled and not dependent on the location of store.

In a dumb component, now the right way to access the datalayer is through following code:

```
    onAddTodo = () => {
        diContainer.dataLayer.todo.add({
            title: this.state.newTitle,
            description: this.state.newDescription
        });
```

Unit testing

Using Jest

There are no requirements or dependencies in order to use Jest. The framework works on a separate Node.js execution and has access to the filesystem. This means that it can interpret the files and filenames. In order to make the tests visible for Jest, we name them <filetotest>.test.js.
Eg. abstractDataLayer.test.js.

In order to create a test-suite, use the describe-function. In order to create a unit test in the test-suite, use the it-function. In order to skip a test, use the xit-function. In order to execute initialize or cleanup code before and after each test, use beforeEach-and afterEach-functions.

For unit testing, we make use of the triple-A flow.
***Arrange:** set up everything in order to start the test. This includes creating objects, filling data or setting up test variables.*
***Act:** this part is the actual executing of the object to test. Testing always starts with an action (executing a function, mounting a component, instantiating a class, …)*
***Assert:** the last part is gathering the test data and asserting them to match our expected results.*

Jest comes with a lot of assertion methods out of the box.
Start an assertion with the Expect function.

View the Jest API documentation here: https://jestjs.io/docs/en/api

Eg:

```
expect(todoList).toBeDefined();
expect(isClicked).toBeTruthy();
expect(homeState).toHaveProperty('newTitle');
expect(homeState.newTitle).toBe('test input');
expect(reducer).toEqual(expectedState);
```

Mocking functions (stubs) and spies

Whenever we want to assert if a function got called, but not execute the function itself, we can use a built-in function mock from Jest.
We replace the existing functionality (because Javascript is untyped, it can execute without repercussion).
The mocked function can be asserted.

```
//arrange
const mock = jest.fn();
home.onAddTodo = mock;

//act
button.simulate('click');

//assert
expect(mock).toHaveBeenCalled();
expect(mock).toHaveBeenCalledTimes(1)
```

Mocking components

Now that we have a dumb component, a dumb abstract data layer, dependency injection and bootstrapping in place, it is easy to test them separately (unit testing). It is also easy to test them working together (integration testing) because of the Lisskov Substitution Principle: we can just inject a mock instead of the store into the abstractDataLayer class through configuration and we can as easily inject a mock of abstractDataLayer into the bootstrapper.

Create and implement the abstract mocking layer

In the below example code, we have created a mock of the data layer object. It should still feature the accessible endpoints (todo.add() and todo.remove()).
The implementation of these endpoint functions is completely replaced by spying code that reveals whether the class was correctly accessed.

```
class abstractDataLayerMock {
    constructor() {
        this._todosAddWasCalled = false;
        this._todosAddArguments = undefined;

        this._todosRemoveWasCalled = false;
        this._todosRemoveArguments = false;
    }

    _todosAdd = (todo) => {
        this._todosAddArguments = todo;
        this._todosAddWasCalled = true;
    }

    _todosRemove = (id) => {
        this._todosRemoveArguments = id;
        this._todosRemoveWasCalled = true;
    }

    get todo() {
        return({
            add: this._todosAdd,
            remove: this._todosRemove,

            info: {
                todosAdd: {
                    wasCalled: this._todosAddWasCalled,
                    arguments: this._todosAddArguments
                },
                todosRemove: {
                    wasCalled: this._todosRemoveWasCalled,
                    arguments: this._todosRemoveArguments
                }
            }
        });
    }
}

export default abstractDataLayerMock;
```

In our test code, we can now inject the mock as a bootstrapper configuration.

```
//Dependency injection - bootstrapping
diContainer.bootstrap(null, new abstractDataLayerMock());
```

Testing the abstract layer itself

Because the data layer is now a dumb component as well, it should be tested.

There is one injected dependency (store) which now can also be mocked.

Redux store mocking already exists in a library called redux-mock-store, so we can make use of it in our test code.

```
import configureStore from 'redux-mock-store';
import abstractDataLayer from '../../repository/abstractDataLayer';

describe('abstractDataLayer', () => {

    let mockStore;
    beforeEach(() => {
        const initialState = { todos: [] };
        const store = configureStore();
        mockStore = store(initialState);
    });

    describe('Todo domain', () => {
        it('Adds a todo item in the store upon calling add', () => {
            //arrange
            const dl = new abstractDataLayer(mockStore); //dependency injection
            const newTodo = {
                id: 'test id',
                title: 'test title',
                description: 'test description'
            };
            //act
            dl.todo.add(newTodo);

            //assert
            const actions = mockStore.getActions(); //mockStore does not write to store
            expect(actions.length).toBe(1);

            const addedAction = actions[0];
            expect(addedAction.type).toBe('domain/todo/TODO_ADD');

            const addedTodo = addedAction.payload;
            expect(addedTodo.id).toBe('test id')
            expect(addedTodo.title).toBe('test title');
            expect(addedTodo.description).toBe('test description');
        });
    });
});
```

Testing with Enzyme

Enzyme is used to test the React components themselves. It cannot function on its own, but it works perfectly when mixed with Jest.

To start unit testing React, import either shallow or mount from the enzyme framework.
Also import React to be able to create React jsx components.

```
import React from 'react';
import { shallow } from 'enzyme';
import { mount } from 'enzyme';
```

Use shallow when only rendering the component itself (easy components).
Use mount when actually simulating a mount of the React component and render all its children.

```
it('Displays passed-through todo item correctly', () => {
    //arrange
    //nothing to arrange

    //act
    const component = shallow(<TodoItem todo={testTodo} />);

    //assert
    const title = component.find('.todoItem h3');
    expect(title.html()).toBe('<h3>testTitle</h3>');

    const description = component.find('.todoItem p');
    expect(description.html()).toBe('<p>testDescription</p>');

    //cleanup
    component.unmount();
});
```

Enzyme creates a ReactWrapper object when shallow or mount functions are called.
This object is not made for testing, but has some useful features.
Make use of html() to render to parsable html, find() to query the dom (this also works with React components) and instance() to get access to the state, props and deeper code.
Use simulate() in order to execute html events.

View the entire Enzyme API documentation here: https://airbnb.io/enzyme/docs/api/

Simulate click

```
it('Calls the remove function upon clicking on the Remove button', () => {
        //arrange
        let isClicked = false;

        const component = shallow(<TodoItem todo={testTodo} onRemove={() => { isClicked =
true; }} />);
        const button = component.find('button');

        //act
        button.simulate('click');

        //assert
        expect(isClicked).toBeTruthy();

        //cleanup
        component.unmount();
    });
```

Simulate onChange

```
it('Updates the state when entering a title', () => {
    //arrange
    const component = createMount(mockedStore);

    //act
    const inputTitle = component.find('.todo-addnew input').first();
    inputTitle.simulate('change', { target: { value: 'test input' } });

    //assert
     const home = component.find('Home').instance();
     const homeState = home.state;
     expect(homeState.newTitle).toBe('test input');

     //cleanup
     component.unmount();
});
```

Enzyme caveats

In order to mock functions on React components by using Jest, the component needs to be updated after the mock.
If the component is not the highest component, that one needs to be updated as well.

```
//arrange
const component = mount(
    <Provider store={store}>
        <TodoList />
    </Provider>
);

const list = component.find('TodoList').instance();
let mockFn = jest.fn();
list.onRemoveClick = mockFn;

list.forceUpdate();
component.update();
```

To set and read the state of a component, a reference to the instance is required. This can be obtained by using the component.instance()-function.
However, after changing state, which causes a rerender of component, a new fetch of instance() is required.
The previous instance is out of sync.

```
const createMount = (mockedStore) => {
    return mount(
        <Provider store={mockedStore}>
            <Home />
        </Provider>
    );
}

//global arrange
const initialState = {
    todos: [
        {
            id: 'testId1',
            title: 'testTitle1',
            description: 'testDescription1'
        },
        {
            id: 'testId2',
            title: 'testTitle2',
            description: 'testDescription2'
        }
    ]
};
let mockedStore;

beforeEach(() => {
    mockedStore = configureStore()(initialState)
});

//Dependency injection – bootstrapping
diContainer.bootstrap(null, new abstractDataLayerMock());
```

See continued example below.

```
it('Resets the state to a default state when clicking the add button', () => {
    //arrange
    const component = createMount(mockedStore);
    const home = component.find('Home');
    home.instance().state = {
        newTitle: 'test title',
        newDescription: 'test description'
    };

    //act
    const button = home.find('.todo-addnew button');
    button.simulate('click');

    //assert
    const finState = home.instance().state;
    expect(finState).toHaveProperty('newTitle');
    expect(finState).toHaveProperty('newDescription');
    expect(finState.newTitle).toBe('');
    expect(finState.newDescription).toBe('');

    //cleanup
    component.unmount();
});



it('Calls the datalayer add function', () => {
    //arrange
    const component = createMount(mockedStore);

    //act
    const button = component.find('.todo-addnew button');
    button.simulate('click');

    //assert
    const funcInfo = diContainer.dataLayer.todo.info;
    expect(funcInfo.todosAdd.wasCalled).toBeTruthy();

    //cleanup
    component.unmount();
});


it('Supplies the datalayer with the correct arguments from the state', () => {
    //arrange
    const component = createMount(mockedStore);
    const home = component.find('Home');
    home.instance().state = {
        newTitle: 'test title',
        newDescription: 'test description'
    };

    //act
    const button = home.find('.todo-addnew button');
    button.simulate('click');

    //assert
    const funcInfo = diContainer.dataLayer.todo.info;
    const args = funcInfo.todosAdd.arguments;
    expect(args.title).toBe('test title');
    expect(args.description).toBe('test description');

    //cleanup
    component.unmount();
});
```