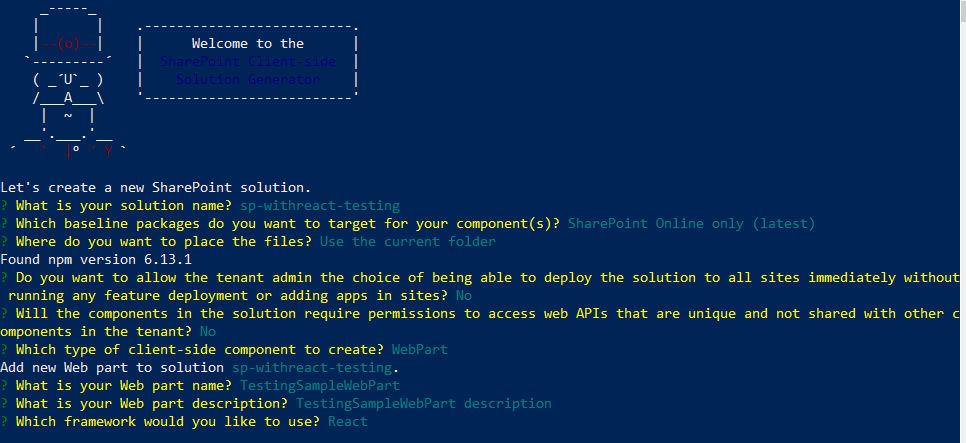
## [Project source at git](https://github.com/Jedikops/ReactHooksAndTestsInSP/tree/master/TestingSample)

## SharePoint project initialization

As I’m SharePoint specialist, I like to keep my React projects within SharePoint Framework (SPFX). Let’s start from quickly creating a SharePoint framework project with React library applied. Remember that YO generator and @MICROSOFT/SHAREPOINT scaffold are necessary in this scenario. Oh, *Gulp* is also required.

*npm install -g yo @microsoft/generator-sharepoint gulp*

The process of creating should look like on a following screenshot.



We will also some additional dependences on the dev side.

*npm i @voitanos/jest-preset-spfx-react16 jest -D*

To simplify this procedure we will use [*@voitanos/jest-preset-spfx-react16*](https://github.com/Voitanos/jest-preset-spfx-react16) package that encapsulates following dependencies:

*@types/enzyme-adapter-react-16*

*@types/enzyme-to-json*

*@types/jest*

*@types/react-test-renderer*

*Enzyme*

*enzyme-adapter-react-16*

*enzyme-to-json*

*identity-obj-proxy*

*raf*

*react-test-renderer*

*ts-jest*

To complete installation process let’s just install jest as fulfillment.

Now we’re all set up to start.

You can test SharePoint application with *gulp* *serve* command.

To test it you will need to run traditionally *npm test* command.

## Context driven application

If we started to build code with common manner, we would quickly end up at a dead end that forfeit ability to change our state from outside involved components. That’s why for the purpose of this article we want to use React context to implement multi component state management. That’s something worth taking in consideration early if we know that application may require some flexibility.

As an example I will build a counter component that might be managed from other components. In the first line I need to build app element that will contains our logic, so in that case I need to add App.tsx to the project with following content.

const App = () => {

return (

<CounterProvider>

<CounterComponent />

<SumComponent />

<MultiplyComponent />

</CounterProvider >

);

};

export default App;

For now our components don’t exist yet, but we will get there. There are four components we need to add. *CounterProvider*– that’s our context provider, it delivers state and actions to our counter components. *CounterComponent –*this one is responsible for displaying count number picked from the state. *SumComponent* - that one is about modifying our state with addition and subtraction functions. *MultiplyComponent –* will modify state with multiply and division functions.

Creating context may be quite tricky in order to build application ready for unit testing. If we create context with *useState,* we would limit ourselves, as in a result our components dispatch functions would have to implement *React.Dispatch<React.SetStateAction<ICounter>> interface. React.SetStateAction* allows to pass function only. Instead we want to apply a reducer to our code which will give us more freedom on dispatched actions.

Let me now introduce you to the interface of our Counter App. We need *ICounterContext* interface to identify what our context will provide. First property defines our state and dispatch is an object associated with actions. It will look like this:

export default interface ICounterContext {

state: ICounter;

dispatch: React.Dispatch<ICounterActionInfo>;

}

We will also need types, that represents our state and actions. In this case let’s name out state *ICounter*– it’s a quite simple structure:

export default interface ICounter {

count: number;

}

*CounterAction* - That’s a type, that identifies our actions, we need to determine input and output types.

export type CounterAction = (state: ICounter, value: number) => ICounter;

Now, we could use these both types to implement our store by using *useState* function and to deliver components a stateful value and function to update it. Because functions’ type must be Dispatch<SetStateAction<S>>, that means we limit ourselves from manipulating state under external conditions. That’s why let me introduce *Reducers* to our project. In order to do that, we have to feed it with some object. *Let it be ICounterActionInfo*.

export default interface ICounterActionInfo {

action: CounterAction;

value: number;

}

Our reducer will use objects implementing this type to modify our state. Thanks to this change, we will give our components possibility to influence on the outcome of stateful values. This may seem hard to understand now, but let me show you the reducer implementation so you may see…

export const DynamicReducer: Reducer<ICounter, ICounterActionInfo> = (state, actionInfo): ICounter=>{

return actionInfo.action(state, actionInfo.value);

}

*DynamicReducer* is going to replace implementation of dispatch functions, when *useState* is applied. It simply accepts *state* and *actionInfo* objects as parameters, and henceforth will apply changes to the returned state object. In our case it will run action function and pass it a state and value. Its result is passed down back to the store.

So far this might be unclear why do I implement all of these types for. Well this is *typescript*, so implementing interfaces allows for more code readability. Let me show you the result, by wrapping them all together. To do that we need definition of the *React*.C*ontext* that creates store for the state.

export const CounterContext: React.Context<ICounterContext> = createContext({} as ICounterContext);

export const useStore = (): ICounterContext => useContext(CounterContext);

Once store is initialized we can proceed with *CounterProvider* implementation. We have to implement provider to provide our components with update functions that manipulates the state.

const initialState: ICounter = { count: 0 };

export const CounterProvider = ({ children }) => {

const [state, dispatch] = useReducer<Reducer<ICounter, ICounterActionInfo>>(DynamicReducer, initialState);

return (<CounterContext.Provider value={{ state, dispatch: dispatch }}>

{children}

</CounterContext.Provider >);

};

Here we want to initialize *reducer* with *useReducer* function instead of calling *useState*. Reducer in that case will be called instead of *dispatch* function. In *DynamicReducers* definition you might have observed that it’s parameters and its logic replaces *dispatch* function, but there we are able to use it as we like. It is possible to use multiple reducers by switching reducers in *runtime*, but today I’m going to save you from such complexity of that mechanism. I will describe it another time. Therefore, we are almost complete.

What we are missing is components implementations. Do create them we are going use React new approach of component creation, and what I mean by that are *hooks*. Previously to manage state in our components we had to create *classes*. That would troublesome especially when we already had a inline component. Now, we can stick to it. We have three components, one will only use state values to present it on display screen to the user. That component will be *CounterComponent* and its definition looks following:

export const CounterComponent = () => {

    const { state }: ICounterState = useStore();

    return (

        <div>

            <div className="count">{state.count}</div>

        </div>

    );

};

That’s easy. Function *useStore* delivered us *ICounterContext* object with state as *ICounter* property. Two other components are going to manipulate state’s values though. Firstly I will describe *SumComponent*.

export const SumComponent = (props: { component: number}) => {

    const { state, dispatch }: ICounterState = useStore();

    React.useEffect(() => {

        dispatch({ action: CounterActions.ResetTo, value: 0 });

    }, []);

    return (

        <div>

            <div>

                <button className="firstButton" onClick={() => dispatch({ action: CounterActions.Increment, value: props.component })}>+</button>

                <button className="secondButton" onClick={() => dispatch({ action: CounterActions.Decrement, value: props.component })}>-</button>

                <button className="thirdButton" onClick={() => dispatch({ action: CounterActions.ResetTo, value: 0 })}>Reset</button>

            </div>

        </div>

    );

};

This component provides user with buttons to manipulate state. Function *useStore* not only can provide us with the state object but also *dispatch* function, we have to remember that its parameter must implement *ICounterActionInfo* type. So far defined that *ICounterActionInfo* holds *CounterAction* type function and value as a *number* type. We actually never created functions that implements *CounterAction* type so far, and pre-compiler sees that! That’s no problem though. I have prepared a constant object that will provide us with plenty of them. Let’s take a look at it.

export const CounterActions = {

Increment: (state: ICounter, value: number): ICounter => {

return { count: state.count + value };

},

Decrement: (state: ICounter, value: number): ICounter => {

return { count: state.count - value };

},

Multiply: (state: ICounter, value: number): ICounter => {

return { count: state.count \* value };

},

Divide: (state: ICounter, value: number): ICounter => {

return { count: state.count / value };

},

ResetTo: (state: ICounter, value: number): ICounter => {

return { count: value };

}

}

Now, our component is fully operational. We can decide what will happen to the state just from the component level. If we now implement *MultiplyComponent*. Pre-compiler will not cause us problems anymore. Let’s see its implementation.

export const MultiplyComponent = (props: { multiplier: number }) => {

    const { state, dispatch }: ICounterState = useStore();

    React.useEffect(() => {

        dispatch({ action: CounterActions.ResetTo, value: 1 });

    }, []);

    return (

        <div>

            <div>

                <button className="firstButton" onClick={() => dispatch({ action: CounterActions.Multiply, value: props.multiplier })}>\*</button>

                <button className="secondButton" onClick={() => dispatch({ action: CounterActions.Divide, value: props.multiplier })}>/</button>

                <button className="thirdButton" onClick={() => dispatch({ action: CounterActions.ResetTo, value: 1 })}>Reset</button>

            </div>

        </div>

    );

};

The implementation of *MultiplyComponent* is nearly the same as the *SumComponent*. One difference is that instead of sum operations this one will multiply and divide count property in our state. If we didn’t create reducer we might have been able to choose function we want to run, but now we can also provide the number we want to sum/multiply by. That’s why we created *ICounterActionInfo* object and associated it with the dispatch function. That’s actually it, this mechanism will not only provides us flexible approach to manipulate state but we can also create accurate tests for our application.

## Testing

A well-known practice of testing React components is by using *Snapshots*. Although this technique may be appealing I strongly advice against using it. By relaying on snapshots, we are under risk of missing potential bugs that will get smuggled along with snapshot creation. Let me enclose this to You.

When a change is committed to the code, and that even involves applying for example simple *onClick* event to a component, whole snapshot is required to be recreated. When that happens, either you are obligated to manually verify newly created literal of a snapshot by yourself or to believe that nothing but *onClick* event has changed inside. But what if there’s a logic manipulating *onClick* event already? You will not notice the change, so forth your code will mostly likely become corrupted.

There’s also *shallow* test, which may be worth trying, by in long term will inevitably will lead you to a crash. By following Kent C. Dodds [article](https://kentcdodds.com/blog/why-i-never-use-shallow-rendering) you will identify risks that come along with that technique. Shallow rendering doesn’t render subcomponents, but only mocks the subcomponent with applied props onto them, and it may actually hide the actual problem hidden inside. Kent C. Dodds states that testing components introduces new dependencies that might trigger an error while the unit test might still pass. And that’s a not a way to do it.

If you wish to write code which will be testable you may want to keep reading. Let’s say we want to test *SumComponent*. We need to create SumComponent.spec.tsx file with following implementation.

let component = 1;

let wrapper: ReactWrapper;

beforeEach(() => {

    wrapper = mount(<CounterProvider>

        <CounterComponent />

        <SumComponent component={component} />

    </CounterProvider>);

});

afterEach(() => {

    wrapper.unmount();

});

test('initial test to verify inital count of counter', () => {

    expect(wrapper).toMatchSnapshot();

});

test('first button restult check', () => {

    expect(wrapper.find('button.firstButton')).not.toBeNull();

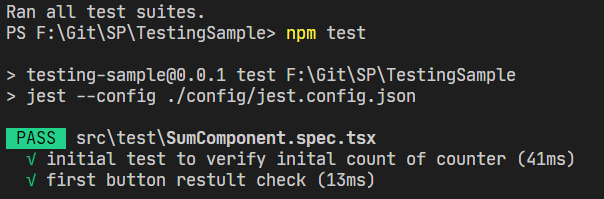
    let button = wrapper.find('button.firstButton').first();

    button.simulate('click');

    expect(wrapper).toMatchSnapshot();

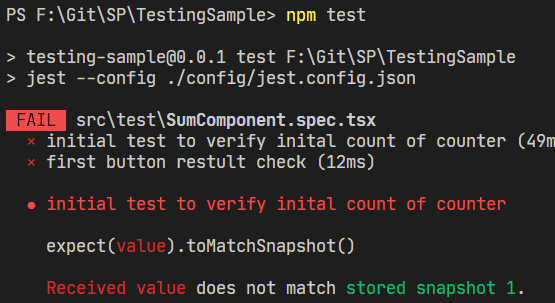
});

Now as we run our test for the first time, *jest* will create a literal object that is going to be used next time test is run. And so when we do that, test passes.



Although, when we try to change the component value in our test or change implementation in our component, test inevitably will fail.

let component = 2;



Now let’s try to test *MultiplyComponent*. Implementation is fairly simple but fully covers the component.

let initialValue = 1;

let component = 2;

let expectedMultiply = initialValue \* component;

let expectedDivide = initialValue / component;

let wrapper: ReactWrapper;

beforeEach(() => {

    wrapper = mount(<CounterProvider>

        <CounterComponent />

        <MultiplyComponent multiplier={component} />

    </CounterProvider>);

});

afterEach(() => {

    wrapper.unmount();

});

test('initial test to verify inital count of counter', () => {

    //  expect(wrapper).toMatchSnapshot();

    expect(wrapper.find('div.count').text()).toBe((initialValue).toString());

});

test('multiply result check', () => {

    expect(wrapper.find('button.firstButton')).not.toBeNull();

    let button = wrapper.find('button.firstButton').first();

    button.simulate('click');

    expect(wrapper.find('div.count').text()).toBe((expected).toString());

});

test('divide result check', () => {

    expect(wrapper.find('button.secondButton')).not.toBeNull();

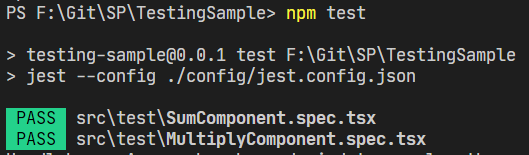
    let button = wrapper.find('button.firstButton').first();

    button.simulate('click');

    expect(wrapper.find('div.count').text()).toBe((expected).toString());

});

During these tests we check initial value and further on observe the result after multiplying/dividing operation. In this scenario it really doesn’t matter if we change the multiplier value. The test shall be still successful!



I hope this article have provided you with a constructive approach on creating react components with context included. I also hope that you enjoyed it😊

End