**Redux Testkit**

Complete and opinionated testkit for testing Redux projects (reducers, selectors, actions, thunks)

* [Installation](https://github.com/wix/redux-testkit#installation)
* [Recipe - Unit testing reducers](https://github.com/wix/redux-testkit#recipe---unit-testing-reducers)
* [Recipe - Unit testing selectors](https://github.com/wix/redux-testkit#recipe---unit-testing-selectors)
* [Recipe - Unit testing thunks](https://github.com/wix/redux-testkit#recipe---unit-testing-thunks)
* [Recipe - Integration tests for the entire store](https://github.com/wix/redux-testkit#recipe---integration-tests-for-the-entire-store)
* [Building and testing this library](https://github.com/wix/redux-testkit#building-and-testing-this-library)

**What tests are we going to write?**

* *Unit tests* for [reducers](http://redux.js.org/docs/basics/Reducers.html) - test recipe [here](https://github.com/wix/redux-testkit#recipe---unit-testing-reducers)
* *Unit tests* for [selectors](http://redux.js.org/docs/recipes/ComputingDerivedData.html) - test recipe [here](https://github.com/wix/redux-testkit#recipe---unit-testing-selectors)
* *Unit tests* for [action handlers (thunks)](https://github.com/gaearon/redux-thunk) - test recipe [here](https://github.com/wix/redux-testkit#recipe---unit-testing-thunks)
* *Integration tests* for the entire [store](http://redux.js.org/docs/basics/Store.html) - test recipe [here](https://github.com/wix/redux-testkit#recipe---integration-tests-for-the-entire-store)

This library mostly provides syntactic sugar and makes testing Redux fun with less boilerplate. You can naturally test all Redux constructs without this library, but you will miss out on features like automatic immutability checks.

**Installation**

* Install the package from npm

npm install redux-testkit --save-dev

* Make sure you have a test runner installed, we recommend [jest](https://facebook.github.io/jest/docs/getting-started.html)

npm install jest --save-dev

**Recipe - Unit testing reducers**

import { Reducer } from 'redux-testkit';

import uut from '../reducer';

describe('counter reducer', () => {

it('should have initial state', () => {

expect(uut()).toEqual({ counter: 0 });

});

it('should handle INCREMENT action on initial state', () => {

const action = { type: 'INCREMENT' };

const result = { counter: 1 };

Reducer(uut).expect(action).toReturnState(result);

});

it('should handle INCREMENT action on existing state', () => {

const action = { type: 'INCREMENT' };

const state = { counter: 1 };

const result = { counter: 2 };

Reducer(uut).withState(state).expect(action).toReturnState(result);

});

it('should return the same state after accpeting a non existing action', () => {

const action = { type: 'NON\_EXISTING' };

const state = { counter: 1 };

Reducer(uut).withState(state).expect(action).toStayTheSame();

});

});

A redux reducer is a pure function that takes an action object, with a type field, and changes the state. In almost every case the state object itself must remain immutable.

**Reducer(reducer).withState(state).expect(action).toReturnState(result)**

* Runs the reducer on current state providing an action. Calling withState() is optional, if not provided, initial state is used. Makes sure the returned state is result.
* Also verifies immutability - that state did not mutate. [Why is this important? see example bug](https://github.com/wix/redux-testkit/blob/master/BUG-EXAMPLES.md#reducer)

[See some examples of this API](https://github.com/wix/redux-testkit/blob/master/API-EXAMPLES.md#reducerreducerwithstatestateexpectactiontoreturnstateresult)

**Reducer(reducer).withState(state).expect(action).toStayTheSame()**

* Runs the reducer on current state providing an action. Calling withState() is optional, if not provided, initial state is used. Makes sure the returned state is the same as the provided state.

**Reducer(reducer).withState(state).expect(action).toChangeInState(changes)**

* Runs the reducer on current state providing an action. Calling withState() is optional, if not provided, initial state is used. Makes sure the part that changed in the returned state matches changes and the rest of the state hasn't changed. The format of changes is partial state, even a deep internal object - it is compared to returned state after [merging](https://lodash.com/docs/#merge) the changes with the original state (objects are deep merged, arrays are replaced).
* Also verifies immutability of the state.

*The added value of this API compared to toReturnState is when your state object is very large and you prefer to reduce the boilerplate of preparing the entire result by yourself.*

[See some examples of this API](https://github.com/wix/redux-testkit/blob/master/API-EXAMPLES.md#reducerreducerwithstatestateexpectactiontochangeinstatechanges)

**Reducer(reducer).withState(state).execute(action)**

* Runs the reducer on current state providing an action. Calling withState() is optional, if not provided, initial state is used.
* Returns the returned state so you can run expectations manually. It's not recommended to use this API directly because you usually won't verify that parts in the returned state that were not supposed to change, indeed did not change.
* Also verifies immutability of the state.

*The added value of this API compared to the others is that it allows you to run your own custom expectations (which isn't recommended).*

[See some examples of this API](https://github.com/wix/redux-testkit/blob/master/API-EXAMPLES.md#reducerreducerwithstatestateexecuteaction)

**Recipe - Unit testing selectors**

import { Selector } from 'redux-testkit';

import \* as uut from '../reducer';

describe('numbers selectors', () => {

it('should select integers from numbers state', () => {

const state = { numbers: [1, 2.2, 3.14, 4, 5.75, 6] };

const result = [1, 4, 6];

Selector(uut.getIntegers).expect(state).toReturn(result);

});

});

A redux selector is a pure function that takes the state and computes some derivation from it. This operation is read-only and the state object itself must not change.

**Selector(selector).expect(state, ...args).toReturn(result)**

* Runs the selector function on a given state. If the selector takes more arguments, provide them at ...args (the state is always assumed to be the first argument of a selector). Makes sure the returned result is result.
* Also verifies that state did not mutate. [Why is this important? see example bug](https://github.com/wix/redux-testkit/blob/master/BUG-EXAMPLES.md#selector)

[See some examples of this API](https://github.com/wix/redux-testkit/blob/master/API-EXAMPLES.md#selectorselectorexpectstate-argstoreturnresult)

**Selector(selector).execute(state, ...args)**

* Runs the selector function on a given state. If the selector takes more arguments, provide them at ...args (the state is always assumed to be the first argument of a selector).
* Returns the returned state so you can run expectations manually.
* Also verifies that state did not mutate.

*The added value of this API compared to the others is that it allows you to run your own custom expectations.*

[See some examples of this API](https://github.com/wix/redux-testkit/blob/master/API-EXAMPLES.md#selectorselectorexecutestate-args)

**Recipe - Unit testing thunks**

import { Thunk } from 'redux-testkit';

import \* as uut from '../actions';

import redditService from '../../../services/reddit';

jest.mock('../../../services/reddit');

describe('posts actions', () => {

beforeEach(() => {

jest.resetAllMocks();

});

it('should clear all posts', () => {

const dispatches = Thunk(uut.clearPosts).execute();

expect(dispatches.length).toBe(1);

expect(dispatches[0].getAction()).toEqual({ type: 'POSTS\_UPDATED', posts: [] });

});

it('should fetch posts from server', async () => {

redditService.getPostsBySubreddit.mockReturnValueOnce(['post1', 'post2']);

const dispatches = await Thunk(uut.fetchPosts).execute();

expect(dispatches.length).toBe(3);

expect(dispatches[0].getAction()).toEqual({ type: 'POSTS\_LOADING', loading: true });

expect(dispatches[1].getAction()).toEqual({ type: 'POSTS\_UPDATED', posts: ['post1', 'post2'] });

expect(dispatches[2].getAction()).toEqual({ type: 'POSTS\_LOADING', loading: false });

});

it('should filter posts', () => {

const state = { loading: false, posts: ['funny1', 'scary2', 'funny3'] };

const dispatches = Thunk(uut.filterPosts).withState(state).execute('funny');

expect(dispatches.length).toBe(1);

expect(dispatches[0].getAction()).toEqual({ type: 'POSTS\_UPDATED', posts: ['funny1', 'funny3'] });

});

});

A redux thunk wraps a synchronous or asynchronous function that performs an action. It can dispatch other actions (either plain objects or other thunks). It can also perform side effects like accessing servers.

**Thunk(thunk).withState(state).execute(...args)**

* Runs the thunk thunk on current state given optional arguments ...args. Calling withState() is optional, no need to provide it if the internal thunk implementation doesn't call getState().
* Returns an array of dispatches performed by the thunk (shallow, these dispatches are not executed). You can run expectations over them manually. If the tested thunk is asynchronous, await on the result to get the actual dispatches array. If it's synchronous, you can use the return value directly without await.
* Also verifies that state did not mutate. [Why is this important? see example bug](https://github.com/wix/redux-testkit/blob/master/BUG-EXAMPLES.md#thunk)

[See some examples of this API](https://github.com/wix/redux-testkit/blob/master/API-EXAMPLES.md#thunkthunkwithstatestateexecuteargs)

**Available expectations over a dispatch**

// when a plain object action was dispatched

expect(dispatches[0].isPlainObject()).toBe(true);

expect(dispatches[0].getType()).toEqual('LOADING\_CHANGED');

expect(dispatches[0].getAction()).toEqual({ type: 'LOADING\_CHANGED', loading: true });

// when another thunk was dispatched

expect(dispatches[0].isFunction()).toBe(true);

expect(dispatches[0].getName()).toEqual('refreshSession'); // the function name, see note below

**Being able to expect dispatched thunk function names**

This is relevant when the tested thunk dispatches another thunk. In order to be able to test the name of the thunk that was dispatched, you will have to provide an explicit name to the internal anonymous function in the thunk implementation. For example:

export function refreshSession() {

return async function refreshSession(dispatch, getState) {

// ...

};

}

**Limitations when testing thunks that dispatch other thunks**

* If the tested thunk dispatches another thunk, the other thunk is not executed. Different thunks should be considered as different units. Executing another unit should be part of an [integration test](https://github.com/wix/redux-testkit#recipe---integration-tests-for-the-entire-store), not a unit test.
* If the tested thunk dispatches another thunk, you cannot set expectations on the arguments given to the other thunk. Different thunks should be considered as different units. Testing the interfaces between them should be part of an [integration test](https://github.com/wix/redux-testkit#recipe---integration-tests-for-the-entire-store), not a unit test.
* If the tested thunk dispatches another thunk, you cannot mock the return value of the other thunk. Relying in your implementation on the return value of another thunk is considered bad practice. If you must test that, you should probably be changing your implementation.

These limitations may seem annoying, but they stem from best practices. If they disrupt your test, it's usually a sign of a code smell in your implementation. Fix the implementation, don't fight to test a bad practice.

**Recipe - Integration tests for the entire store**

import { createStore, applyMiddleware, combineReducers } from 'redux';

import thunk from 'redux-thunk';

import { FlushThunks } from 'redux-testkit';

import \* as reducers from '../reducers';

import \* as uut from '../posts/actions';

import \* as postsSelectors from '../posts/reducer';

import redditService from '../../services/reddit';

jest.mock('../../services/reddit');

describe('posts store integration', () => {

let flushThunks, store;

beforeEach(() => {

jest.resetAllMocks();

// create a redux store with flushThunks added as the first middleware

flushThunks = FlushThunks.createMiddleware();

store = createStore(combineReducers(reducers), applyMiddleware(flushThunks, thunk));

});

it('should select posts', () => {

expect(postsSelectors.getSelectedPost(store.getState())).toEqual([]);

store.dispatch(uut.selectPost('post1'));

expect(postsSelectors.getSelectedPost(store.getState())).toEqual(['post1']);

store.dispatch(uut.selectPost('post2'));

expect(postsSelectors.getSelectedPost(store.getState())).toEqual(['post1', 'post2']);

});

it('should fetch posts from server', async () => {

redditService.getPostsBySubreddit.mockReturnValueOnce(['post1', 'post2']);

expect(postsSelectors.getPostsLoading(store.getState())).toBe(false);

expect(postsSelectors.getPosts(store.getState())).toEqual([]);

await store.dispatch(uut.fetchPosts());

expect(postsSelectors.getPostsLoading(store.getState())).toBe(false);

expect(postsSelectors.getPosts(store.getState())).toEqual(['post1', 'post2']);

});

it('should test a thunk that dispatches another thunk', async () => {

expect(postsSelectors.isForeground(store.getState())).toBe(false);

await store.dispatch(uut.initApp()); // this dispathces thunk appOnForeground

await flushThunks.flush(); // wait until all async thunks resolve

expect(postsSelectors.isForeground(store.getState())).toBe(true);

});

});

Integration test for the entire store creates a real redux store with an extra flushThunks middleware. Test starts by dispatching an action / thunk. Expectations are set over the final state using selectors.

**flushThunks = FlushThunks.createMiddleware()**

* Creates flushThunks middleware which should be applied to the store on creation. This middleware is useful for the case where one thunk dispatches another thunk. It allows to wait until all of the thunk promises have been resolved.
* Returns a flushThunks instance which has the following methods:

**flushThunks.flush()**

* Flushes all asynchronous thunks. Run await on this method to wait until all dispatched thunk promises are resolved.

**flushThunks.reset()**

* Call this method to reset the list of thunk promises observed by flushThunks.

**Building and testing this library**

This section is relevant only if you want to contribute to this library or build it locally.

* Install and build

npm install

npm run build

* Run lint and tests

npm run test