6. Подходы тестирования Redux

- Обзор
- Тестирование создателей Action
- Тестирование Reducers
- Тестирование Store
- Интеграционное тестирование в Redux
- Тестирование Redux саги
- Итоги

Обзор

Привет и добро пожаловать, друг ✌️👨‍🎓! В этом уроке, наша цель выяснить как мы можем покрыть тестами разные части `Redux` в нашем приложении, выяснить детали и подходы, и как это все сделать наиболее эффективным способом 🤿.

В основном, `Redux` легко поддается тестированию, что является одним из его ключевых преимуществ. Для того чтобы полностью автоматизировать тестирование, мы можем создать `юнит тесты` для всех и каждого из игроков – `reducers`, `action creators`, `middleware` и `store`. И/или совместить их в более обширном понятии `интеграционных тестов` 📄.

Нам доступно большое количество инструментов тестирования, но точная оснастка менее важна так, как большая часть наших приложений Redux будут полагаться на простые функции и объекты JavaScript, без сложных библиотек или асинхронных потоков для тестирования. Хотя мы воплотим в жизнь тест `redux-saga`, используя более продвинутую технику тестирования.

В качестве нашего фреймворка для тестирования мы будем использовать замечательную библиотеку Jest от Facebook, последняя версия которой, уверенно остается лучшей версией для тестирования Redux. Использование других структур и инструментов, таких как `Karma`, `Mocha`, и так далее, будут выглядеть очень похоже на примеры в нашем уроке 👩‍⚕️.

Тестирование создателей действий

Мы уже знаем, как управлять `асинхронными` потоками из `создателей действий` делегируя их `redux-saga`. Такой подход позволяет тестировать `создатели действий` очень просто, по скольку они являются функциями, которые всего лишь возвращают простые объекты JavaScript.

```
// app/actions/magicBook/index.js

import { CHANGE_PAGE } from './types';

export const changePage = (page) => ({
    type:    CHANGE_PAGE,
    payload: page
});
```

Наш `changePage()` `создатель действия` получает единичный параметр и в ответ создает простой объект JavaScript. Так как нет контролируемого логического потока или побочных эффектов, любой вызов данной функции всегда возвращает одно и то же значение делая простым её тестирование.

```
// app/actions/magicBook/index.test.js

import { changePage } from './';

const newPage = '5';

describe('magicBook action creators:', () => {
    test('changePage action creator should produce a correct action', () => {
        expect(changePage(newPage)).toEqual({
            type:    'CHANGE_PAGE',
            payload: newPage
        });
    });
});
```

В примере выше `Jest` функции `expect()` и `toEqual()` используются для проверки того, что `changePage(newPage)` `создатель действия` даёт правильный результат (тот результат, который мы ожидали). Обратите внимание, что метод `toEqual()` специфический для проверки эквивалентности `объектов`, в то время как метод `toBe()` специфический для проверки `примитивов`. Взгляните на API [expect](#) для того, чтобы изучить весь набор утверждений представленных в Jest.

🖐🏼 Совет:

> В процессе разработки в реальном мире вы постоянно будете сталкиваться с ситуациями, в которых вы будете использовать множество повторяющихся данных в ваших тестах. Заметьте, что на `линии кода 5`, где объявлен идентификатор `newPage`. Строка со значением `'5'` привязана к идентификатору `newPage`, таким образом, что он может быть повторно использован в любом месте тестового набора и любое количество раз. Очень легко сделать ошибку, указывая каждый раз `'5'`, когда нам это

> потребуется. Более того, эта проблема возрастает вместе с размером вашего тестового набора. Вот почему извлечение данных в ваших тестах может пригодится 🎅.

Тестирование Reducers

Тестирование `reducers` очень похоже на тестирование `создателей действий`, `reducers` являются `idempotent` по определению (для одних и тех же `state` и `action`, производит один и тот же `new state` каждый раз).

Это делает тестирование `reducer` более легким в написании, так как нам всего лишь нужно вызвать `reducer` с разными комбинациями ввода для проверки правильности вывода.

```
// app/reducers/magicBook/index.js

import { CHANGE_PAGE } from '../../actions/types';
import { Map } from 'immutable';

const initialState = Map({
    title:       'Magic and Enchantment',
    totalPages:  898,
    currentPage: '1'
});

export default (state = initialState, action) => {
    switch (action.type) {
        case CHANGE_PAGE:
            return state.set('currentPage', action.payload);

        default:
            return state;
    }
};
```

У нас есть `reducer`, который управляет всего лишь `CHANGE_PAGE` `действием` изменяя `currentPage` значение `state` для нашего теста:

```
// app/reducers/magicBook/index.test.js

import magicBook from './';
import { Map } from 'immutable';

const initialState = Map({
    title:       'Magic and Enchantment',
    totalPages:  898,
    currentPage: '1'
```

```
});

const changePageAction = {
    type:    'CHANGE_PAGE',
    payload: '5'
};

describe('magicBook reducer:', () => {
    test('Should handle \'CHANGE_PAGE\' action correctly', () => {
        expect(magicBook(initialState, changePageAction))
        .toEqual(
            initialState.set('currentPage', changePageAction.payload)
        );
    });
});
```

🔍 Заметка:

> Приглядитесь получше к определению типа `CHANGE_PAGE` – мы используем этот тип как строки вместо того, чтобы импортировать константу, которая у нас уже есть. Это важная деталь в этом тесте – главная цель протестировать `reducer` в изоляции, без каких-либо внешних данных. Используя импортированную `CHANGE_PAGE` было бы нарушением данного правила. Более того, в таком случае адекватность теста будет под вопросом. Поэтому мы определяем тестовые данные буквально как тестовый набор.

В этом тесте, рассматривается вопрос о том, правильно ли значение `currentPage` изменяется в `state` в соответствии с `CHANGE_PAGE` действием преданным в `reducer` в качестве второго аргумента.

Тест `initialState` представляется как `immutable Map`, так же как с оригинальным редюсером. `Reducer` должен вернуть правильно обновленный `state` в ответ, на то что и происходит в нашем случае для данных `state` и `action` о которых он заботится.

Подобный тест можно рассматривать как гипсовую форму `кода`, в некотором смысле. В любое время мы будем уведомлены о том, что реальное обновление логики `reducer` изменяется потому, что старая гипсовая форма больше не подходит под новую функциональность `reducer` 🗿.

Вот почему так полезно писать соответствующие тесты для стольких частей приложения, для скольких возможно. Особенно для таких инфраструктурных решений как `Redux` 🛡️.

Тестирование Store

Основная цель тестирования `store` – убедиться в том, что форма `state` правильная. Во время разработки часто, когда кто-то начнет возиться со структурой редюсера, для того чтобы реорганизовать старый код, добавить новый редюсер, убрать устаревший или расширить функциональность любым другим способом – существует множество причин сделать изменения в директории `reudcers`. Так как `reducers` уже покрыты юнит тестами, так же нам нужно быть уверенным в том, что все эти `reducers` попадут в `store`:

```
// app/store/index.test.js

import { combineReducers, createStore } from 'redux';
import store from './';
import user from '../reducers/user';
import magicBook from '../reducers/magicBook';
import books from '../reducers/books';

const reducer = combineReducers({
    user,
    magicBook,
    books
});

const expectedStore = createStore(reducer);

describe('store:', () => {
    test('The initial store state shape should be correct', () => {
        expect(store.getState()).toEqual(expectedStore.getState())
    })
});
```

Это широко распространённый сценарий, в котором кто-то просто забыл добавить запрашиваемый редюсер в вызов `createStore`. Таким образом, эта дополнительная проверка защищает приложение от еще одного неудачного сценария.

Интеграционное тестирование в Redux

В рамках `Redux`, комплексный пакет `юнит тестов` обеспечит правильную работу всем `reducers` и `action creators`. С интеграционным тестированием `Redux`, мы попытаемся запустить всё вместе в едином тесте, чтобы проверить общесистемное поведение вместе с правильной реакцией `store` на `dispatched` действия.

В качестве примера интеграционного теста, мы проверим, когда `changePage()` `создатель действия` является `запущенным`, `действие` обрабатывается должным образом в `редюсере`, и обновляется `состояние`:

```
// app/store/integration.test.js

import store from './';
import { changePage } from '../actions';

const getCurrentPage = () => store.getState().magicBook.get('currentPage');
const nextPage = '2';

describe('integration test:', () => {
    it('The \'CHANGE_PAGE\' action should update state correctly once
dispatched', () => {
        expect(getCurrentPage()).toBe('1');

        store.dispatch({
            type:    'CHANGE_PAGE',
            payload: nextPage
        });

        expect(getCurrentPage()).toBe(nextPage);
    });
});
```

✋ Совет:

> Попробуйте извлечь повторяющиеся части вашего кода и привяжите их вместе к идентификаторам, для того чтобы их потом повторно использовать. Это поможет вам сохранить ваш код чистым, читабельным и элегантным 💎.

Наш тест включает в себя три шага:

- Проверить значение начального свойства `currentPage` в `state`;
- `Dispatch` `действия` созданного `changePage()` `создателем действия`;
- Проверить, что наш ключ `currentPage` `хранилища` содержит новое значение.

Чтобы быть уверенным в том, что `reducer` обновляет `state` правильно, для начала нужно проверить, что начальное состояние осталось без изменений, так как нам нужно затем убедится что оно изменилось правильно после того как `changePage()` `создатель действия` был `запущен`.

Тестирование Redux саги

Сага – это функция генератор, которая производит описанные `effects` для обработки в сага `middleware`.

Относительно сложный характер функций генераторов может немного усложнить понимание тестового набора. В частности, возможности `bitwise communication` функций генераторов 📲.

🔍 Заметка:

> Как вы наверное уже знаете, функция генератор создает экземпляр `итератора` инициализированный единожды. Созданный `итератор` имеет свойство `value`, которое определяет завершения статуса тела функции генератора и метод `next()`, который используется для вызова следующего `тика` функции генератора. `Bitwise communication` функции генератора проявляется в возможности передавать аргумент вызову `next()`. Поступая таки образом, передача значения отправляется обратно в поток выполнения функции генератора. Обратите внимание на документацию MDN для получения более детализированной информации.

Основная мотивация использования `redux-saga` для управления `асинхронной` логикой– является неповторимая сила `effects`, которая расширяется с помощью `redux-saga` API, и имеет хороший показатель тестируемости. Функция генератор последовательна и поэтому мы можем тестировать каждый элемент кода пошагово 🔬.

Обновление саги, которую мы собираемся протестировать:

```
// app/sagas/index.js

import { call, put, takeLatest } from 'redux-saga/effects';
import {
    START_FETCHING_BOOKS,
    END_FETCHING_BOOKS,
    GET_FAVORITE_BOOKS,
    GET_FAVORITE_BOOKS_SUCCESS
} from '../actions/profile/types';

export function* getFavoriteBooksWorker () {
    yield put({ type: START_FETCHING_BOOKS });

    const result = yield call(
        fetch,
        'https://data.book-reader.io/users/Oscar/favorite-books'
    );

    const favoriteBooks = yield call([result, result.json]);

    yield put({
        type:    GET_FAVORITE_BOOKS_SUCCESS,
        payload: favoriteBooks
    });
```

```
    yield put({ type: END_FETCHING_BOOKS });
}

export function* watchGetFavoriteBooks () {
    yield takeLatest(GET_FAVORITE_BOOKS, getFavoriteBooksWorker);
}
```

И фактический тест:

```
// app/sagas/index.test.js

import { getFavoriteBooksWorker } from './';
import { call, put, takeLatest } from 'redux-saga/effects';

const saga = getFavoriteBooksWorker();
const responseData = [
    {
        title: 'The mystery of sunlight spectrum.'
    }
];
const fetchResponse = {
    status: 200,
    json:   () => Promise.resolve(responseData)
};

global.fetch = jest.fn(() => Promise.resolve(fetchResponse));

describe('getFavroiteBooks saga:', () => {
    test(`Should dispatch 'START_FETCHING_BOOKS' action`, () => {
        expect(saga.next().value).toEqual(
            put({
                type: 'START_FETCHING_BOOKS'
            })
        );
    });

    test(`Should call a 'fetch' request`, () => {
        expect(saga.next().value).toEqual(
            call(
                fetch,
                'https://data.book-reader.io/users/Oscar/favorite-books'
            )
        );
    });
```

```
    test(`The 'fetch' request result should be handled correctly`, () => {
        expect(saga.next(fetchResponse).value).toEqual(
            call([fetchResponse, fetchResponse.json])
        );
    });

    test(`Should dispatch a 'GET_FAVORITE_BOOKS_SUCCESS' action`, () => {
        expect(saga.next(responseData).value).toEqual(
            put({
                type:    'GET_FAVORITE_BOOKS_SUCCESS',
                payload: responseData
            })
        );
    });

    test(`Should dispatch a 'END_FETCHING_BOOKS' action`, () => {
        expect(saga.next().value).toEqual(
            put({
                type: 'END_FETCHING_BOOKS'
            })
        );
    });
});
```

Давайте разделим это тестирование пошагово:

- `На линии кода 6`: мы инициализируем сагу, которую собираемся протестировать;
- `На линии кода 7`: необработанные данные, которые мы ожидаем получить от вызова `fetch`. Данные обратного значения `fetch` используются в нашей реализации, которую мы собираемся изучить;
- `На линии кода 12`: Возвращаемое значение из `fetch`. То, что мы собираемся получить в ответ на удачный вызов `fetch`;
- `На линии кода 17`: мы вызываем `jest.fn()` для того, чтобы зарегистрировать заглушку для функции `fetch`;
- `На линии кода 21`: в первом случае мы проверяем `value` первого `tick` функции генератора. Обратите внимание на `эффект` `put()` – в нашем тесте, нам нужно подражать каждому `эффекту`, который мы использовали в фактической реализации саги. В этом `tick` `START_FETCHING_BOOKS` `действие` должно быть `запущено`;
- `На линии кода 29`: эта проверка гарантирует, что `fetch` был вызван с помощью правильного URL;
- `На линии кода 38`: здесь мы проверяем, что возвращаемое значение `fetch` трансформируется в необработанные данные правильно. Обратите внимание на аргумент, который мы передали вызову `next()`. Поступая таким образом, мы

используем фишки функции генератора `bitwise communication` для того, чтобы передать заглушку `fetch` соответствующего объекта в поток функции генератора;
- `На линии кода 43`: все `асинхронные` запросы на этот момент завершены. Теперь нам нужно только проверить, если надлежащие `действия` `запущены` с правильным `payload` в соответствии с удачным вызовом `fetch`. В нашем случае это `действие` `GET_FAVORITE_BOOKS_SUCCESS`;
- `На линии кода 53`: наконец, последняя проверка того, что `END_FETCHING_BOOKS` `действие` `запущено`.

Возможность управления `асинхронной` функцией, которое предоставляет `redux-saga` впечатляющее. Преимущества большой мощности взамен компенсируются дополнительной сложностью. Однако, преимущества намного более значительны, чем все недостатки в этой технологии. Попробуйте сделать некоторые элементарные операции `асинхронными` самостоятельно, написать несколько тестов, и использовать дополнительные формы `эффектов` пакета `redux-saga`. Чем больше вы практикуетесь, тем более опытным экспертом вы становитесь. Таким образом, как только вы `ощутите` `redux-saga` – вам будет сложно выбрать любой другой инструмент для управления `асинхронной` логикой в вашем `Redux` приложении 👨🏽‍🎨.

Итоги

Следуя принципу четкого разделения обязанностей в `Redux`, придерживаясь использовани обычных объектов JavaScript, `идемпотентных` функций, то большинство `юнит тестов` (и `интеграционных тестов`) будут короткими и простыми в написании ✍🏽.

В случае идиоматических `создателей действий` и `редюсеров` с `юнит тестами` ясно то, что нам нужно создать гипсовую форму кода, чтобы при любой реализации изменений тестируемых сущностей – мы были немедленно уведомлены об этом 🚨.

Это означает, что мы как разработчики можем минимизировать время, которое мы тратим на написание `тестов` и все еще имеем всеобъемлющий тестовый набор 📝.

Написание `интеграционных тестов` в `Redux` обычно довольно-таки просто. Так как всё управляется при помощи `действий`, в большинстве случаев, наш `интеграционный тест` будет следовать по трем шагам, изложенным ниже: мы проверяем `начальное состояние` системы, `диспатч` `действия`, и проверяем, что `состояние` изменилось. В некоторых типичных случаях, существует так же не обязательный шаг — заглушать любые внешние зависимости 🗄.

Использование саги для обработки побочных эффектов делает код более легким для проверки и более организованным, потому что мы не смешиваем `создтели действий` с побочными эффектами, а используем мощный набор `эффектов` предложенных `redux-saga` в сочетании с могущественными функциями генераторами JavaScript 💪🏽.

Спасибо, что остаетесь с нами, и да прибудет с вами идеальный код ❤️!

> Если у вас есть идеи как улучшить этот урок, пожалуйста, поделитесь своими идеями с нами `hello@lectrum.io`. Ваш отзыв очень важен для нас!