**Настройка проекта**

Чтобы использовать Redux и TypeScript, нам нужно создать новое приложение React.

Для этого давайте откроем интерфейс командной строки (CLI) и выполним следующую команду:

npx create-react-app my-app --template typescript

Далее давайте структурируем проект следующим образом:

├── src

| ├── components

| | ├── AddArticle.tsx

| | └── Article.tsx

| ├── store

| | ├── actionCreators.ts

| | ├── actionTypes.ts

| | └── reducer.ts

| ├── type.d.ts

| ├── App.test.tsx

| ├── App.tsx

| ├── index.css

| ├── index.tsx

| ├── react-app-env.d.ts

| └── setupTests.ts

├── tsconfig.json

├── package.json

└── yarn.lock

Структура файлов проекта довольно проста. Однако следует отметить две вещи:

* storeПапка, содержащая файлы, связанные с React Redux.
* Файл type.d.ts содержит типы TypeScript, которые теперь можно использовать в других файлах без импорта.

Тем не менее, теперь мы можем установить Redux и создать наш самый первый магазин.

Итак, давайте откроем проект и выполним следующую команду:

yarn add redux react-redux redux-thunk

Или при использовании npm

npm install redux react-redux redux-thunk

Нам также нужно установить их типы в качестве зависимостей для разработки, чтобы TypeScript мог распознавать библиотеки.

Итак, давайте снова выполним эту команду в командной строке.

yarn add -D @types/redux @types/react-redux @types/redux-thunk

Или для npm:

npm install -D @types/redux @types/react-redux @types/redux-thunk

Отлично! Сделав этот шаг вперёд, мы можем приступить к созданию типов TypeScript для проекта в следующем разделе.

**Создайте типы**

Типы TypeScript позволяют задавать типы для переменных, параметров функций и так далее.

* type.d.ts

interface IArticle {

id: number

title: string

body: string

}

type ArticleState = {

articles: IArticle[]

}

type ArticleAction = {

type: string

article: IArticle

}

type DispatchType = (args: ArticleAction) => ArticleAction

Здесь мы начинаем с объявления интерфейса IArticle, который отражает форму данной статьи.

Затем у нас есть ArticleState, ArticleAction, и DispatchType, которые будут служить типами для объекта состояния, создателей действий и функции отправки, предоставляемой Redux.

Итак, теперь у нас есть необходимые типы для начала использования React Redux. Давайте создадим типы действий.

**Создание типов действий**

* store/ActionTypes.ts

export const ADD\_ARTICLE = "ADD\_ARTICLE"

export const REMOVE\_ARTICLE = "REMOVE\_ARTICLE"

Нам нужны два типа действий для хранилища Redux. Одно для добавления статей, а другое для удаления.

**Создайте создателей действия**

* store/ActionCreators.ts

import \* as actionTypes from "./actionTypes"

export function addArticle(article: IArticle) {

const action: ArticleAction = {

type: actionTypes.ADD\_ARTICLE,

article,

}

return simulateHttpRequest(action)

}

export function removeArticle(article: IArticle) {

const action: ArticleAction = {

type: actionTypes.REMOVE\_ARTICLE,

article,

}

return simulateHttpRequest(action)

}

export function simulateHttpRequest(action: ArticleAction) {

return (dispatch: DispatchType) => {

setTimeout(() => {

dispatch(action)

}, 500)

}

}

В этом руководстве я буду имитировать HTTP-запрос, задерживая его на 0,5 секунды. Но вы можете использовать реальный сервер, если хотите.

Здесь функция addArticle отправит запрос на добавление новой статьи, а метод removeArticle сделает противоположное. Поэтому удалите объект, переданный в качестве аргумента.

**Создайте редуктор**

Редуктор — это чистая функция, которая принимает в качестве параметров состояние хранилища и действие, а затем возвращает обновлённое состояние.

* store/reducer.ts

import \* as actionTypes from "./actionTypes"

const initialState: ArticleState = {

articles: [

{

id: 1,

title: "post 1",

body:

"Quisque cursus, metus vitae pharetra Nam libero tempore, cum soluta nobis est eligendi",

},

{

id: 2,

title: "post 2",

body:

"Harum quidem rerum facilis est et expedita distinctio quas molestias excepturi sint",

},

],

}

Как вы можете видеть, здесь мы объявляем начальное состояние, чтобы при загрузке страницы отображались некоторые статьи. Объект состояния должен соответствовать типу ArticleState — в противном случае TypeScript выдаст ошибку.

* store/reducer.ts

const reducer = (

state: ArticleState = initialState,

action: ArticleAction

): ArticleState => {

switch (action.type) {

case actionTypes.ADD\_ARTICLE:

const newArticle: IArticle = {

id: Math.random(), // not really unique

title: action.article.title,

body: action.article.body,

}

return {

...state,

articles: state.articles.concat(newArticle),

}

case actionTypes.REMOVE\_ARTICLE:

const updatedArticles: IArticle[] = state.articles.filter(

article => article.id !== action.article.id

)

return {

...state,

articles: updatedArticles,

}

}

return state

}

export default reducer

Далее у нас есть функция reducer, которая ожидает предыдущее состояние и действие, чтобы обновить хранилище. Здесь у нас есть два действия: одно для добавления, а другое для удаления.

Теперь, когда всё готово, мы можем управлять состоянием с помощью редьюсера. Давайте создадим хранилище для проекта.

**Создайте магазин**

Хранилище Redux - это место, где находится состояние вашего приложения.

* index.tsx

import \* as React from "react"

import { render } from "react-dom"

import { createStore, applyMiddleware, Store } from "redux"

import { Provider } from "react-redux"

import thunk from "redux-thunk"

import App from "./App"

import reducer from "./store/reducer"

const store: Store<ArticleState, ArticleAction> & {

dispatch: DispatchType

} = createStore(reducer, applyMiddleware(thunk))

const rootElement = document.getElementById("root")

render(

<Provider store={store}>

<App />

</Provider>,

rootElement

)

Как видите, мы импортируем функцию-редуктор, а затем передаём её в качестве аргумента методу createStore для создания нового хранилища Redux. Промежуточное ПО redux-thunk также необходимо передать в качестве второго параметра методу, чтобы можно было обрабатывать асинхронный код.

Далее мы подключаем React к Redux, предоставляя объект store в качестве реквизита компоненту Provider.

Теперь мы можем использовать Redux в этом проекте и получить доступ к хранилищу. Итак, давайте создадим компоненты для получения данных и управления ими.

**Создание компонентов**

* components/AddArticle.tsx

import \* as React from "react"

type Props = {

saveArticle: (article: IArticle | any) => void

}

export const AddArticle: React.FC<Props> = ({ saveArticle }) => {

const [article, setArticle] = React.useState<IArticle | {}>()

const handleArticleData = (e: React.FormEvent<HTMLInputElement>) => {

setArticle({

...article,

[e.currentTarget.id]: e.currentTarget.value,

})

}

const addNewArticle = (e: React.FormEvent) => {

e.preventDefault()

saveArticle(article)

}

return (

<form onSubmit={addNewArticle} className="Add-article">

<input

type="text"

id="title"

placeholder="Title"

onChange={handleArticleData}

/>

<input

type="text"

id="body"

placeholder="Description"

onChange={handleArticleData}

/>

<button disabled={article === undefined ? true : false}>

Add article

</button>

</form>

)

}

Чтобы добавить новую статью, мы будем использовать этот компонент формы. Он принимает функцию saveArticle в качестве параметра, которая позволяет добавить новую статью в магазин.

Объект article должен соответствовать типу IArticle, чтобы сделать TypeScript счастливым.

* components/Article.tsx

import \* as React from "react"

import { Dispatch } from "redux"

import { useDispatch } from "react-redux"

type Props = {

article: IArticle

removeArticle: (article: IArticle) => void

}

export const Article: React.FC<Props> = ({ article, removeArticle }) => {

const dispatch: Dispatch<any> = useDispatch()

const deleteArticle = React.useCallback(

(article: IArticle) => dispatch(removeArticle(article)),

[dispatch, removeArticle]

)

return (

<div className="Article">

<div>

<h1>{article.title}</h1>

<p>{article.body}</p>

</div>

<button onClick={() => deleteArticle(article)}>Delete</button>

</div>

)

}

Компонент Article показывает объект article.

Функция removeArticle должна быть отправлена для доступа к хранилищу и, следовательно, для удаления конкретной статьи. Именно поэтому мы используем здесь хук useDispatch, который позволяет Redux завершить действие по удалению.

Кроме того, использование useCallback помогает избежать ненужного повторного рендеринга за счёт запоминания значений в качестве зависимостей.

Наконец-то у нас есть компоненты, необходимые для добавления и отображения статей. Теперь давайте добавим последний элемент головоломки, используя их в файле App.tsx

* App.tsx

import \* as React from "react"

import { useSelector, shallowEqual, useDispatch } from "react-redux"

import "./styles.css"

import { Article } from "./components/Article"

import { AddArticle } from "./components/AddArticle"

import { addArticle, removeArticle } from "./store/actionCreators"

import { Dispatch } from "redux"

const App: React.FC = () => {

const articles: readonly IArticle[] = useSelector(

(state: ArticleState) => state.articles,

shallowEqual

)

const dispatch: Dispatch<any> = useDispatch()

const saveArticle = React.useCallback(

(article: IArticle) => dispatch(addArticle(article)),

[dispatch]

)

return (

<main>

<h1>My Articles</h1>

<AddArticle saveArticle={saveArticle} />

{articles.map((article: IArticle) => (

<Article

key={article.id}

article={article}

removeArticle={removeArticle}

/>

))}

</main>

)

}

export default App

useSelector позволяет получить доступ к состоянию хранилища. Здесь мы передаём shallowEqual в качестве второго аргумента методу, чтобы сообщить Redux, что при проверке изменений следует использовать поверхностное равенство.

Далее мы используем useDispatch для отправки запроса на добавление статей в магазин. Наконец, мы проходимся по массиву статей и передаём каждую из них компоненту Article для отображения.

Теперь мы можем перейти в корень проекта и выполнить эту команду:

yarn start

Или для npm:

npm start

Если вы откроете http://localhost:3000/ в браузере, вы должны увидеть это:

