**Типизация Redux: повышение надежности приложения и tradeoffs**

Типизация React-приложений однозначно повышает удобство разработки и качество кода. Да, с Typescript кода становится больше, зато результат не заставляет себя ждать: IDE «расцветает» подсказками и подсветкой потенциальных проблем. В этой теме расскажем об особенностях и преимуществах типизации Redux и его составляющих.

**Что принесём в жертву**

Вы уже убедились, что Redux без Redux Toolkit довольно многословен. Конечно, если мы задействуем Typescript в приложениях, которые используют Redux, то неизбежно будем писать больше кода. Но в этом случае принцип «чем меньше кода — тем лучше» не работает. Мы будем использовать Typescript для описания типов экшенов и состояния, а для удобства работы создадим несколько вспомогательных типов и интерфейсов. Не пугайтесь, разработчики Redux позаботились о нас и предоставили удобные конструкции для типизации.

**Что получим**

Может показаться, что добавление ко всем этим экшенам, генераторам экшенов, редьюсерам и прочему ещё и Typesript — не путь самурая. Но к счастью, Typescript многое автоматизирует и щедро вознаграждает разработчиков, уделивших внимание типизации Redux. Кроме того, Redux и Redux Thunk предоставляют интерфейсы, которые расширяются с помощью дженериков. Благодаря этому, при правильной первоначальной настройке типов, разработка приложений с Redux сводится к простым действиям и не отнимает много времени. Как итог — значительное снижение шансов на ошибку: IDE подскажет, если передан какой-то неправильный тип экшена. А при использовании хуков useDispatch и useSelector больше не нужно смотреть в код экшенов и редьюсеров — IDE сделает это за вас и подскажет доступные варианты.

**Структура файлов**

В этой теме мы будем использовать такую структуру файлов, но вы можете выбрать и что-то другое.

Скопировать кодBASH

services/

├── actions/

│ ├── index.ts - экшены

│ └── todo.ts

├── constants/

│ ├── todo.ts - типы экшенов

│ └── index.ts

├── reducers/

│ ├── index.ts

│ └── todo.ts - редьюсеры

├── types/

│ ├── index.ts - типизация вспомогательная

│ └── data.ts - типизация данных (приходящих с сервера, например)

└── store.ts - хранилище

Давайте вместе убедимся, что типизация Redux — не страшнее типизации функции! В следующем уроке расскажем, как типизировать экшены.

**Типизация Actions**

В этом уроке мы покажем, как типизировать экшены в Redux. Но прежде чем начать, давайте обсудим, зачем вообще это нужно.

Когда мы типизируем экшены, мы говорим интерпретатору Typescript, что у нас есть только такие экшены и никаких больше. Так мы защищаем код от опечаток и вызова несуществующих или некорректных экшенов. Для начала добавим Typesript к типам экшенов:

Скопировать кодTSX

// constants/todo.ts

const GET\_TODO\_LIST: 'GET\_TODO\_LIST' = 'GET\_TODO\_LIST';

const GET\_TODO\_LIST\_FAILED: 'GET\_TODO\_LIST\_FAILED' = 'GET\_TODO\_LIST\_FAILED';

const GET\_TODO\_LIST\_SUCCESS: 'GET\_TODO\_LIST\_SUCCESS' = 'GET\_TODO\_LIST\_SUCCESS';

const ADD\_TODO: 'ADD\_TODO' = 'ADD\_TODO';

const ADD\_TODO\_FAILED: 'ADD\_TODO\_FAILED' = 'ADD\_TODO\_FAILED';

const ADD\_TODO\_SUCCESS: 'ADD\_TODO\_SUCCESS' = 'ADD\_TODO\_SUCCESS';

const DELETE\_TODO: 'DELETE\_TODO' = 'DELETE\_TODO';

const DELETE\_TODO\_FAILED: 'DELETE\_TODO\_FAILED' = 'DELETE\_TODO\_FAILED';

const DELETE\_TODO\_SUCCESS: 'DELETE\_TODO\_SUCCESS' = 'DELETE\_TODO\_SUCCESS';

Тут мы воспользовались литеральными типами, так как константы принимают только такие значения. Этот подход выручит нас в будущем.

Далее необходимо описать экшены:

Скопировать кодTSX

*// actions/todo.ts*

import type { TTodoItem } from '../types/data';

import {

GET\_TODO\_LIST,

GET\_TODO\_LIST\_FAILED,

GET\_TODO\_LIST\_SUCCESS,

DELETE\_TODO,

DELETE\_TODO\_FAILED,

DELETE\_TODO\_SUCCESS,

ADD\_TODO,

ADD\_TODO\_FAILED,

ADD\_TODO\_SUCCESS

} from '../constants';

*// Типизация экшенов*

export interface IGetTodoListAction {

readonly type: typeof GET\_TODO\_LIST;

}

export interface IGetTodoListSuccessAction {

readonly type: typeof GET\_TODO\_LIST\_SUCCESS;

readonly list: TTodoItem[];

}

export interface IGetTodoListFailedAction {

readonly type: typeof GET\_TODO\_LIST\_FAILED;

}

export interface IAddTodoAction {

readonly type: typeof ADD\_TODO;

readonly text: string;

}

export interface IAddTodoSuccessAction {

readonly type: typeof ADD\_TODO\_SUCCESS;

readonly list: TTodoItem[];

}

export interface IAddTodoFailedAction {

readonly type: typeof ADD\_TODO\_FAILED;

}

export interface IDeleteTodoAction {

readonly type: typeof DELETE\_TODO;

readonly id: number;

}

export interface IDeleteTodoSuccessAction {

readonly type: typeof DELETE\_TODO\_SUCCESS;

readonly list: TTodoItem[];

}

export interface IDeleteTodoFailedAction {

readonly type: typeof DELETE\_TODO\_FAILED;

}

*// Генераторы экшенов*

export const addTodo = (text: string): IAddTodoAction => ({

type: ADD\_TODO,

text

});

export const deleteTodo = (id: number): IDeleteTodoAction => ({

type: DELETE\_TODO,

id

});

Например, так генератор экшена addToDo гарантирует, что его можно вызвать только с аргументом типа «строка»:

Скопировать кодJSX

addToDo({text: 'Купить лодку'}) // Ошибка

addToDo(42) // Ошибка

addToDo('Собрать чемодан') // Ок

Теперь воспользуемся Union-типом, чтобы объединить типы экшенов:

Скопировать кодTSX

*// actions/todo.ts*

import type { TTodoItem } from '../types/data';

import {

GET\_TODO\_LIST,

GET\_TODO\_LIST\_FAILED,

GET\_TODO\_LIST\_SUCCESS,

DELETE\_TODO,

DELETE\_TODO\_FAILED,

DELETE\_TODO\_SUCCESS,

ADD\_TODO,

ADD\_TODO\_FAILED,

ADD\_TODO\_SUCCESS

} from '../constants';

*// Типизация экшенов*

export interface IGetTodoListAction {

readonly type: typeof GET\_TODO\_LIST;

}

export interface IGetTodoListSuccessAction {

readonly type: typeof GET\_TODO\_LIST\_SUCCESS;

readonly list: TTodoItem[];

}

export interface IGetTodoListFailedAction {

readonly type: typeof GET\_TODO\_LIST\_FAILED;

}

export interface IAddTodoAction {

readonly type: typeof ADD\_TODO;

readonly text: string;

}

export interface IAddTodoSuccessAction {

readonly type: typeof ADD\_TODO\_SUCCESS;

readonly list: TTodoItem[];

}

export interface IAddTodoFailedAction {

readonly type: typeof ADD\_TODO\_FAILED;

}

export interface IDeleteTodoAction {

readonly type: typeof DELETE\_TODO;

readonly id: number;

}

export interface IDeleteTodoSuccessAction {

readonly type: typeof DELETE\_TODO\_SUCCESS;

readonly list: TTodoItem[];

}

export interface IDeleteTodoFailedAction {

readonly type: typeof DELETE\_TODO\_FAILED;

}

*// Объединяем в Union*

export type TTodoActions =

| IGetTodoListAction

| IGetTodoListSuccessAction

| IGetTodoListFailedAction

| IDeleteTodoAction

| IDeleteTodoSuccessAction

| IDeleteTodoFailedAction

| IAddTodoAction

| IAddTodoSuccessAction

| IAddTodoFailedAction;

export const addTodo = (text: string): IAddTodoAction => ({

type: ADD\_TODO,

text

});

export const deleteTodo = (id: number): IDeleteTodoAction => ({

type: DELETE\_TODO,

id

});

Такое объединение поможет при типизации соответствующего редьюсера — IDE будет подсказывать допустимые экшены и их поля.

На этом типизация экшенов завершена. Двигаемся дальше — в следующем уроке разберёмся, как типизировать редьюсеры.

# Типизация Reducers

В предыдущем уроке вы узнали, как типизировать экшены. Типизация редьюсеров — ещё один необходимый шаг к TypeScript в Redux приложениях. В этом уроке разберёмся с описанием типов для редьюсеров и подготовим все необходимое для типизации хранилища.

Каждый редьюсер отвечает за свой кусочек данных в общем состоянии приложения, поэтому описание типов для редьюсеров сводится к описанию типов конкретной части состояния хранилища:

Скопировать кодTSX

*// reducers/todo.ts*

import type { TTodoItem } from '../types/data';

type TTodoListState = {

list: ReadonlyArray<TTodoItem>;

listRequest: boolean;

listRequestFailed: boolean;

addRequest: boolean;

addRequestFailed: boolean;

deleteRequest: boolean;

deleteRequestFailed: boolean;

}

Этот тип описывает кусочек состояния, который будет помещён в хранилище. Теперь опишем начальное состояние редьюсера и применим получившийся тип TTodoListState:

Скопировать кодTSX

// reducers/todo.ts

...

// Начальное состояние редьюсера

const todoInitialState: TTodoListState = {

list: [],

listRequest: false,

listRequestFailed: false,

addRequest: false,

addRequestFailed: false,

deleteRequest: false,

deleteRequestFailed: false

}

Помните, [в предыдущем уроке](https://praktikum.yandex.ru/trainer/react/lesson/0946d652-c075-4b6b-904e-a6d341ab8e61) мы сделали union-тип для экшенов списка дел? Самое время им воспользоваться. В этом union-типе описаны все экшены, которые влияют на этот кусочек состояния. Напишем сам редьюсер и применим тип TTodoActions:

Скопировать кодTSX

*// reducers/todo.ts*

import type { TTodoItem } from '../types/data';

*// Допустимые экшены для данного редьюсера*

import type { TTodoActions } from '../actions';

import {

GET\_TODO\_LIST,

GET\_TODO\_LIST\_FAILED,

GET\_TODO\_LIST\_SUCCESS,

DELETE\_TODO,

DELETE\_TODO\_FAILED,

DELETE\_TODO\_SUCCESS,

ADD\_TODO,

ADD\_TODO\_FAILED,

ADD\_TODO\_SUCCESS

} from '../constants';

...

*// Редьюсер*

export const todoReducer = (state = todoInitialState, action: TTodoActions): TTodoListState => {

switch (action.type) {

case GET\_TODO\_LIST: {

return { ...state, listRequest: true, listRequestFailed: false };

}

case GET\_TODO\_LIST\_SUCCESS: {

return { ...state, listRequest: false, list: action.list };

}

case GET\_TODO\_LIST\_FAILED: {

return { ...state, listRequest: false, listRequestFailed: true };

}

case DELETE\_TODO: {

return { ...state, deleteRequest: true, deleteRequestFailed: false };

}

case DELETE\_TODO\_SUCCESS: {

return { ...state, deleteRequest: false, list: action.list };

}

case DELETE\_TODO\_FAILED: {

return { ...state, deleteRequest: false, deleteRequestFailed: true };

}

case ADD\_TODO: {

return { ...state, addRequest: true, deleteRequestFailed: false };

}

case ADD\_TODO\_SUCCESS: {

return { ...state, addRequest: false, list: action.list };

}

case ADD\_TODO\_FAILED: {

return { ...state, addRequest: false, deleteRequestFailed: true };

}

default: {

return state;

}

}

};

Теперь редьюсер польностью типизирован. При попытке вернуть в state какой-то ключ, которого нет в типе TTodoListState, получим ошибку:

Скопировать кодTSX

//...

case GET\_TODO\_LIST: {

return {

...state,

listRequest: true,

listRequestFailed: false,

someNewField: 'Ы' // Ошибка

};

}

//...

Кроме того, написание этого редьюсера — практически автоматическое. Благодаря типизации экшенов IDE заботливо «подсовывает» необработанные условия внутри switch-case:

 Когда IDE заботится о вас, практически как любимая бабуля.

Важно: не забывайте убирать кавычки у каждого условия, чтобы вместо строк использовались соответствующие константы. Такая типизация редьюсеров защищает от некорректных записей в хранилище и не даёт использовать экшены, которые не имеют отношения к данной части состояния.

В следующем уроке поговорим про вспомогательные типы, thunk и типизацию хранилища. А для закрепления знаний из этого урока мы подготовили пару заданий.

**Вспомогательные типы: хранилище, Redux Thunk, хуки**

В предыдущем уроке мы рассказали про типизацию редьюсеров. Теперь разберём типизацию хранилища, Redux-хуков и экшенов, которые являются функциями и служат для выполнения побочных эффектов в Redux-приложении.

**Типизация хранилища**

Вы удивитесь, но типизация хранилища готова. Всё, что было нужно, — это типизировать редьюсеры. А именно это мы и сделали в предыдущем уроке. Когда мы передаем типизированные редьюсеры в функцию combineReducers, rootReducer сам становится типизорованным. Дальше, когда rootReducer передается в createStore, тот в свою очередь тоже типизируется:

Скопировать кодTSX

*// reducers/index.ts*

import { todoReducer } from './todo'; *// Типизированный редьюсер*

export const rootReducer = combineReducers({todo: todoReducer})

Скопировать кодTSX

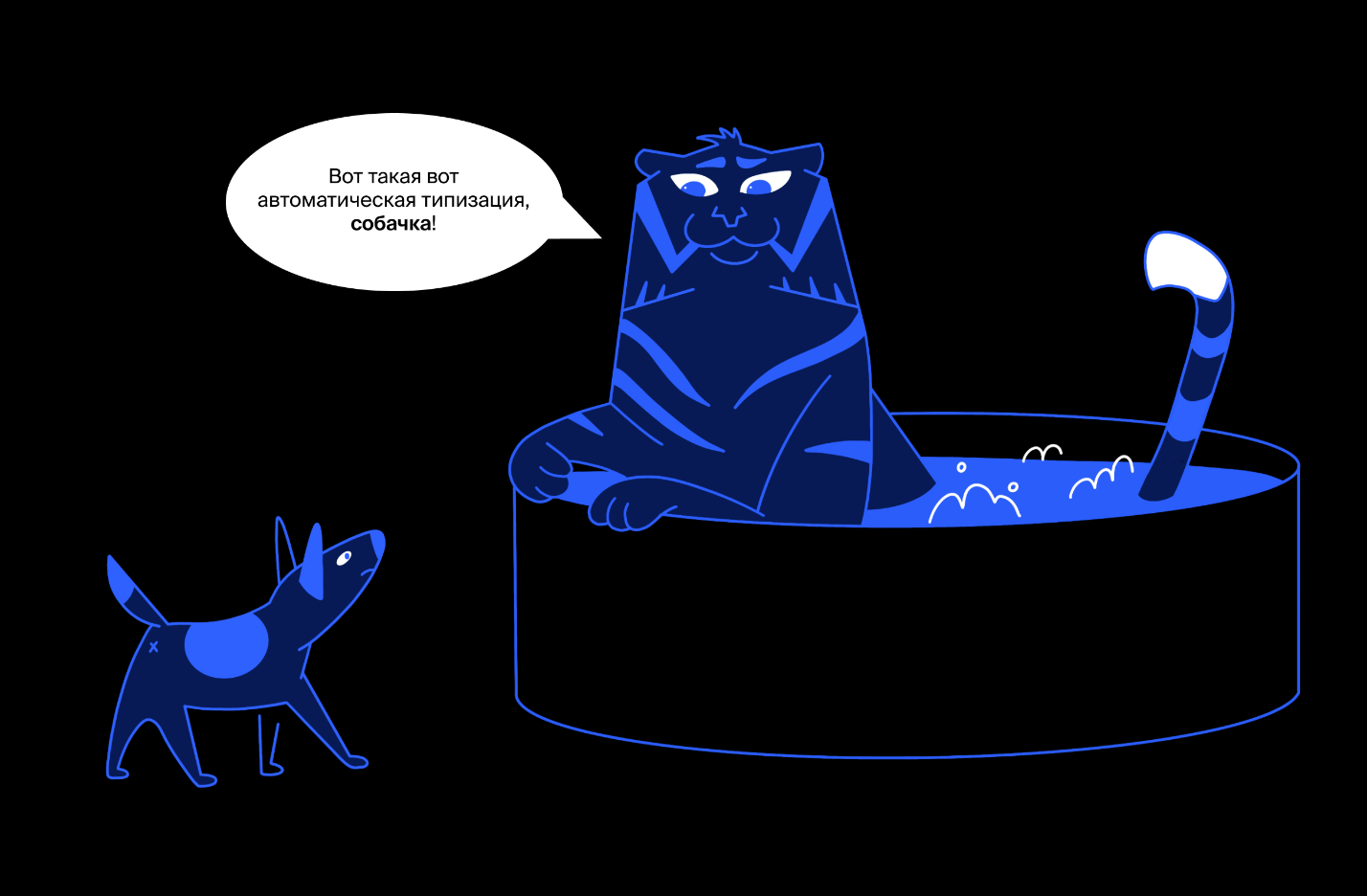
*// store.ts*

import { createStore } from 'redux';

*// Автоматически типизированный корневой редьюсер*

import { rootReducer } from './reducers';

const store = createStore(rootReducer);



Чуть позже, для типизациии thunk'ов, нам потребуется отдельный тип, который содержит описание нашего хранилища. Сделать его можно двумя способами:

1. Описать тип, повторяющий структуру хранилища, а в качестве значений установить типизацию кусочков состояния:

Скопировать кодTSX

*// types/index.ts*

import { TTodoListState } from '../reducers';

export type RootState = { todo: TTodoListState };

Такой подход имеет один недостаток: при добавлении новых ветвей состояния, нужно описывать их все вручную. Второй способ позволяет автоматизировать эту задачу.

1. Воспользоваться вспомогательным типом ReturnType из стандартной библиотеки и конструкцией typeof:

Скопировать кодTSX

*// types/index.ts*

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

export type RootState = ReturnType<typeof store.getState>;

С помощью такой конструкции можно описать все ветви состояния (сколько бы их ни было) разом.

Всё готово! Хранилище типизировано!

**Типизация Redux Thunk**

Для приготовления Redux Thunk под соусом Typesript нам потребуются:

1. Тип ThunkAction из пакета redux-thunk
2. Интерфейсы Action и ActionCreator из пакета redux
3. Union-тип TTodoActions и RootState из нашего приложения.

Начнём с уже известных частей и опишем типизацию всех экшенов приложения:

Скопировать кодTSX

*// types/index.ts*

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

import { TTodoActions } from './actions';

export type RootState = ReturnType<typeof store.getState>;

*// Типизация всех экшенов приложения*

type TApplicationActions = TTodoActions;

Сейчас тип TApplicationActions не очень «содержательный», но такая запись позволит без труда расширять его с помощью union-конструкции, если в приложение добавится дополнительный функционал.

Прежде чем описывать типизацию thunk-экшена, вспомним, как он выглядит:

Скопировать кодJSX

export const getTodoListThunk = () => dispatch => {

dispatch(getTodoListAction());

getTodoListRequest().then(res => {

if (res && res.success) {

dispatch(getTodoListSuccessAction(res.data));

} else {

dispatch(getTodoListFailedAction());

}

});

};

Можно сказать, что thunk — это функция которая возвращает другую функцию, в замыкании которой есть метод dispatch и которая может вернуть (а может и не вернуть) какой-то результат. 🤯

Да, мы знаем, как это звучит, но нам сегодня везёт — разработчики библиотек позаботились об удобных обертках для типизации. Попробуем описать наш thunk с помощью Typescript:

Скопировать кодTSX

*// types/index.ts*

import { ThunkAction } from 'redux-thunk';

import { Action, ActionCreator } from 'redux';

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

import { TTodoActions } from './actions';

export type RootState = ReturnType<typeof store.getState>;

*// Типизация всех экшенов приложения*

type TApplicationActions = TTodoActions;

*// Типизация thunk'ов в нашем приложении*

export type AppThunk<TReturn = void> = ActionCreator<

ThunkAction<TReturn, Action, RootState, TApplicationActions>

>;

Осталось чуть-чуть! Как и с типизацией хранилища, типизация метода dispatch необходима для полного покрытия типами наших thunk'ов. Для решения этой задачи воспользуемся typeof:

Скопировать кодTSX

*// types/index.ts*

import { ThunkAction } from 'redux-thunk';

import { Action, ActionCreator } from 'redux';

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

import { TTodoActions } from './actions';

export type RootState = ReturnType<typeof store.getState>;

*// Типизация всех экшенов приложения*

type TApplicationActions = TTodoActions;

*// Типизация thunk'ов в нашем приложении*

export type AppThunk<TReturn = void> = ActionCreator<

ThunkAction<TReturn, Action, RootState, TApplicationActions>

>;

*// Типизация метода dispatch для проверки на валидность отправляемого экшена*

export type AppDispatch = typeof store.dispatch;

Есть и альтернативный способ типизации метода dispatch — с помощью типа Dispatch из пакета redux. Он принимает типизацию всех экшенов как дженерик:

Скопировать кодTSX

*// types/index.ts*

import { Dispatch } from 'redux';

import { TTodoActions } from './actions';

*// Типизация всех экшенов приложения*

type TApplicationActions = TTodoActions;

*// Типизация метода dispatch для проверки на валидность отправляемого экшена*

export type AppDispatch = Dispatch<TApplicationActions>;

С точки зрения валидации типов разницы между этими двумя способами нет.

Типизация thunk-экшенов готова! Теперь типизированный thunk выглядит так:

Скопировать кодTSX

export const getTodoListThunk: AppThunk = () => (dispatch: AppDispatch) => {

dispatch(getTodoListAction());

getTodoListRequest().then(res => {

if (res && res.success) {

dispatch(getTodoListSuccessAction(res.data));

} else {

dispatch(getTodoListFailedAction());

}

});

};

Ну и пока вошли в раж, типизируем ещё и redux-хуки.

**Типизация useDispatch и useSelector**

Экшены и редьюсеры мы типизировали, thunk'и, dispatch и хранилище — тоже. Чтобы повысить надёжность и качество кода, а ещё минимизировать шансы на опечатки в хуках, нам осталось типизировать хуки.

Начнём с useSelector. Как всегда, для решения этой задачи уже есть подготовленный инструментарий — интерфейс TypedUseSelectorHook. Он необходим для типизации хука useSelector и принимает в качестве дженерика типизацию хранилища:

Скопировать кодTSX

*// hooks.ts*

import {

TypedUseSelectorHook,

useSelector as selectorHook

} from 'react-redux';

import { RootState } from './types';

*// Теперь этот хук «знает» структуру хранилища*

export const useSelector: TypedUseSelectorHook<RootState> = selectorHook;

Для хука useDispatch не требуется дополнительных обёрток, он принимает в качестве дженерика типизацию метода dispatch. Так как мы используем библиотеку Redux Thunk, нам необходимо с помощью union-типа передать в дженерик хука типизацию thunk-экшенов:

Скопировать кодTSX

*// hooks.ts*

import {

TypedUseSelectorHook,

useDispatch as dispatchHook,

useSelector as selectorHook

} from 'react-redux';

import { AppDispatch, AppThunk, RootState } from './types';

*// Теперь этот хук «знает» структуру хранилища*

export const useSelector: TypedUseSelectorHook<RootState> = selectorHook;

*// Хук не даст отправить экшен, который ему не знаком*

export const useDispatch = () => dispatchHook<AppDispatch | AppThunk>();

Теперь вместо хуков из пакета react-redux мы будем пользоваться собственными типизированными версиями:

Скопировать кодTSX

import { useSelector, useDispatch } from '../services/hooks';

export const TodoList = () => {

const dispatch = useDispatch();

const { list, listRequest } = useSelector(state => state.todo);

useEffect(() => {

dispatch({ type: GET\_TODO\_LIST });

}, []);

if (listRequest) {

return 'Loading';

}

return (

<>

{list.map((item, index) => (

<p key={index}>{item}</p>

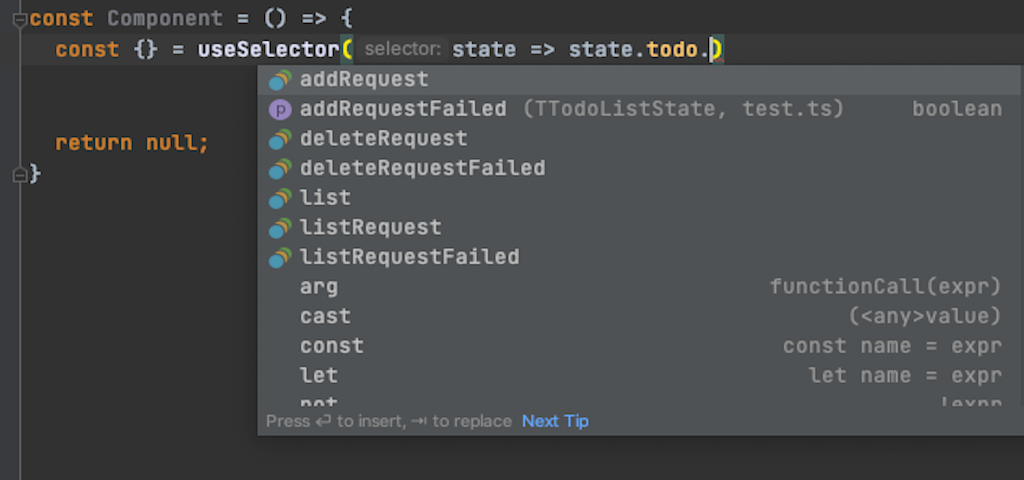
))}

</>

);

};

А IDE любезно подсказывает доступные ключи состояния приложения:

*Заботливая бабуля IDE снова в деле, разве что блины не печёт*

Теперь вы знаете, как типизировать приложение, которое использует Redux и Redux Thunk. Для закрепления материала из этого урока мы подготовили несколько задач. Перейдём к практике.

# Введение в WebSocket

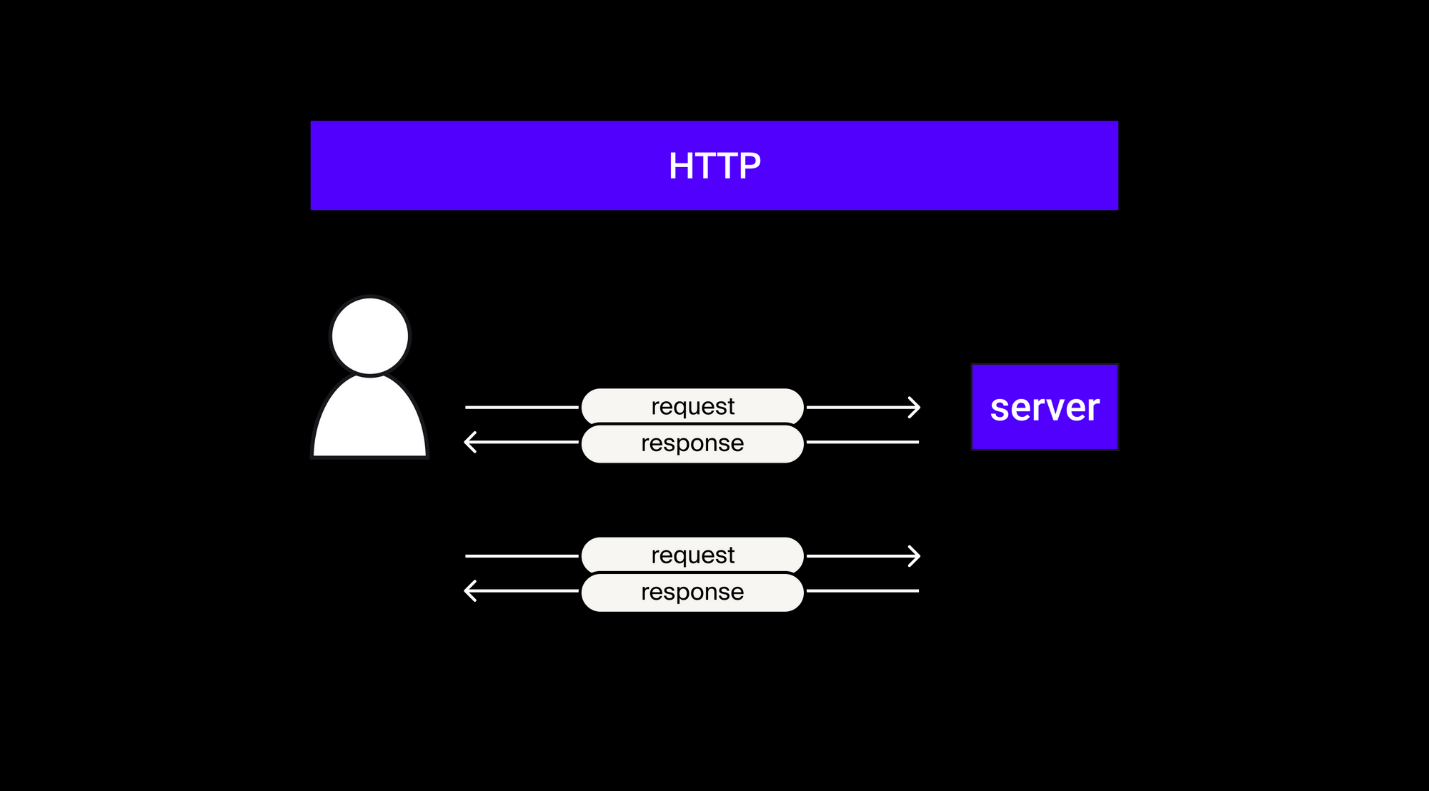
В этой теме вам предстоит познакомиться с новым протоколом передачи данных — WebSocket. Вы наверняка с ним уже сталкивались на просторах интернета, просто не задумывались об этом. Протокол WebSocket обычно применяют для реализации чатов, финансовых приложений (рынков ценных бумаг, облигаций), игровых платформ или других инструментов для совместной работы.

Протокол WebSocket создаёт постоянное соединение, с помощью которого можно непрерывно обмениваться данными без необходимости создавать новые запросы.

## Обмен данными по HTTP и WebSocket

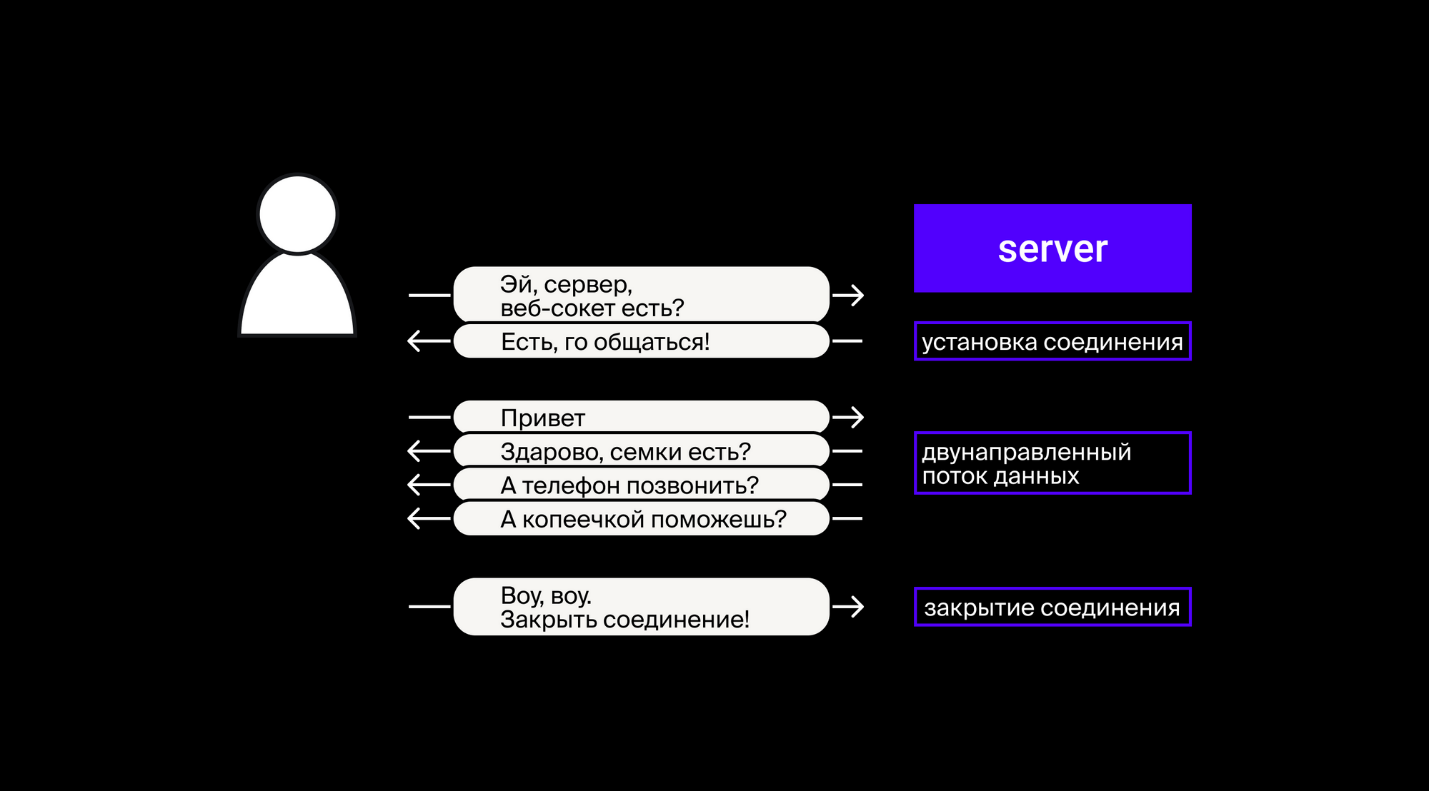
Сравним, как происходит обмен данными по протоколу WebSocket с уже хорошо знакомым нам HTTP.

Схема обмена данными по HTTP выглядит так — клиент отправляет запрос на сервер и получает от него ответ:



С веб-сокетом дела обстоят иначе:

1. Сначала необходимо установить соединение. Клиент отправляет HTTP-запрос, чтобы узнать, поддерживает ли сервер WebSocket.
2. Если сервер поддерживает сокет, то устанавливается постоянное соединение, в котором происходит обмен данными. При этом сервер может присылать клиенту сообщения без запроса (например, пуш-уведомления от приложений).
3. В любой момент времени любая из сторон может закрыть соединение.



При подключении по WebSocket используется один из протоколов: открытый (незащищённый) и зашифрованный.

## Открытый (ws://) и зашифрованный (wss://) протоколы

Протоколы в WebSocket соответствуют открытому http:// и зашифрованному https:// и записываются так: ws:// и wss://.

Вид протокола, по которому будет открываться WebSocket, в первую очередь зависит от сервера. Это значит, что если на стороне клиента мы выбираем зашифрованный протокол, а сервер его не поддерживает, сокет не будет работать. Поэтому на стороне клиента нужно всегда выбирать тот вид протокола, который поддерживается сервером.

Скопировать кодJSX

// открытый протокол

const ws = new WebSocket("ws://norma.nomoreparties.space/");

// зашифрованный протокол

const ws = new WebSocket("wss://norma.nomoreparties.space/");

Если в браузере открыть DevTools и посмотреть на запрос во вкладке Network, то среди заголовков запроса вы найдете те, которые сигнализируют серверу, что клиент хочет использовать веб-сокет как протокол передачи данных:

* Connection: Upgrade — указывает, что клиент хочет изменить протокол;
* Upgrade: websocket — указывается запрашиваемый протокол;
* Sec-WebSocket-Key: блаблабла== — случайный ключ для обеспечения безопастности;
* Sec-WebSocket-Version: 13 — версия протокола WebSocket.

Если сервер поддерживает WebSocket, клиент получит ответ со статусом 101 Switching Protocols и такими заголовками:

* Sec-WebSocket-Accept: newблаблабла== — перекодированный Sec-WebSocket-Key. Этот заголовок нужен, чтобы клиент (браузер) мог убедиться, что ответ соответствует запросу.
* Connection: Upgrade;
* Upgrade: websocket.

## Альтернативы WebSocket

WebSocket — не единственный в своём роде. Его альтернативой считают протокол server-sent events (SSE). Для работы с ним используется класс EventSource.

В целом его поведение очень похоже на поведение WebSocket за несколькими исключениями:

* это однонаправленный канал — данные посылает только сервер;
* отправляться может только текст;
* протокол — HTTP.

Протокол server-sent events на практике применяют гораздо реже, чем WebSocket. Но о нём полезно знать для расширения кругозора. С более подробной информацией о SSE вы можете ознакомиться [по ссылке](https://developer.mozilla.org/en-US/docs/Web/API/EventSource), а вот о WebSocket — уже в следующем уроке!

# Открытие и закрытие соединения

В этом уроке вы узнаете, как открывать и закрывать соединение. Кроме этого, научитесь определять тип состояния подключения.

## Открытие соединения

Для открытия соединения требуется создать объект WebSocket. Это можно сделать так:

Скопировать кодJSX

const ws = new WebSocket("ws://norma.nomoreparties.space/")

При создании объекта класса WebSocket не требуется никаких дополнительных параметров, кроме адреса API WebSocket с типом подключения ws или wss. О них мы говорили [в предыдущем уроке](https://praktikum.yandex.ru/trainer/react/lesson/d35a0419-85ee-4011-961e-dede83f55715).

## Состояние соединения

Объект ws содержит свойство readyState:

Скопировать кодJSX

const ws = new WebSocket("ws://norma.nomoreparties.space/")

console.log(ws.readyState)

А ещё у свойства readyState объекта ws есть четыре значения:

* 0 (CONNECTING): соединение ещё не установлено. WebSocket переходит в это состояние после создания объекта класса, но до начала обмена данными.
* 1 (OPEN): обмен данными. Свойство readyState принимает это состояние после того, как соединение установлено.
* 2 (CLOSING): соединение закрывается. Свойство readyState переходит в это состояние после начала закрытия соединения.
* 3 (CLOSED): соединение закрыто. Значение readyState принимает последнее состояние после закрытия соединения.

## Закрытие соединения

Право на закрытие соединения имеют как браузер, так и сервер. Это происходит с помощью вызова метода close:

Скопировать кодJSX

const ws = new WebSocket("ws://norma.nomoreparties.space/")

ws.close('code', 'reason')

У метода есть два необязательных параметра: code и reason.

code — специальный код закрытие. Это может быть любое число, но есть принятая конвенция кодов. Вот несколько примеров ([обо всех типах можно прочитать по ссылке](https://datatracker.ietf.org/doc/html/rfc6455#section-7.4.1)):

| **Close code (uint16)** | **Codename** | **Description** |
| --- | --- | --- |
| 0 - 999 |  | Обычно не используется. |
| 1000 | CLOSE\_NORMAL | Нормальное закрытие без ошибок. |
| 1001 | CLOSE\_GOING\_AWAY | Страница с открытым сокетом была закрыта клиентом. |
| 1002 | CLOSE\_PROTOCOL\_ERROR | Ошибка внутри протокола фрейма. |
| 1003 | CLOSE\_UNSUPPORTED | Неправильный тип данных. |
| 1004 |  | Зарезервированный тип ошибки. |
| 1005 | CLOSED\_NO\_STATUS | Закрыто без определённого статуса. |

reason — строка, которая описывает причину закрытия соединения.

При закрытии соединения срабатывает функция onclose — подробнее разберём её в следующем уроке.

# События Websocket

В предыдущем уроке мы говорили о том, как отрывать и закрывать соединение. Чтобы работать с WebSocket, нужно разобраться в его событиях, на которые можно «подписаться».

Все события, которые происходят с WebSocket, доступны по соответствующему ключу из из объекта WebSocket. К примеру так выглядят события onopen и onmessage:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen

ws.onmessage

Чтобы добавить метод, который будет отвечать за соответствующее событие, нужно приравнять необходимый метод этому событию, например:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen = (event: Event) => {

console.log("Соединение установлено")

}

Мы добавили метод для обработки события onopen, который будет выводить сообщение в консоль в случае успешного открытия WebSocket. Вне зависимости от того, на какое событие подписан метод, его параметр — всегда event.

У WebSocket есть четыре типа события — разберём каждый из них подробнее.

## Событие onopen

Начнём с события open. Оно вызывается в случае успешного открытия подключения по WebSocket. Подписаться на него можно так:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen = (event: Event) => {

console.log("Соединение установлено");

}

Метод — слушатель события onopen — получает event в качестве параметра. Параметр метода содержит много дополнительной информации (метаданные), описание всех событий и методов, подписанных на эти события. Подробно о типе данных параметра event можно почитать [по ссылке.](https://dom.spec.whatwg.org/#interface-event)

Например, по ключу type параметр event возвратит значение 'open':

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onopen = (event: Event) => {

console.log("Соединение установлено");

console.log(event.type); *// open*

}

## Событие onmessage

Это событие вызывается каждый раз, когда приходят данные по протоколу. Иначе говоря, когда сервер отправляет сообщение, в браузере вызывается метод, определённый в качестве слушателя метода onmessage объекта класса WebSocket:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onmessage = (event: MessageEvent) => {

console.log(`Получены данные: ${event.data}`)

}

Рассмотрим onmessage на примере чата. Вы уже знаете, что событие обрабатывается в том случае, если сервер отправляет сообщение с данными. Эти данные хранятся в объекте event, их можно получить по ключу data. Подробнее о других полях event можно почитать в описании его типа данных [MessageEvent](https://html.spec.whatwg.org/multipage/comms.html#messageevent).

Например, пользователи в чате обмениваются сообщениями. Когда арендодатель отправляет сообщение — сервер присылает событие 'сообщение', которое будет содержать объект с такими данными:

Скопировать кодJSON

{

"id": 1, // идентификатор пользователя

"name": 'Иванов Василий' // Имя пользователя

"message": 'Привет, дома? Пришли показатели счётчиков, срочно!' // Текст сообщения

}

Чтобы получить доступ к этому объекту — достаточно обратиться к полю data объекта event, который мы передали как параметр в обработчик события:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onmessage = (event: MessageEvent) => {

const userMessage = event.data;

console.log(userMessage); *// {id: 1, name: 'Иванов Василий', message: 'Привет, дома? Пришли показатели счётчиков, срочно!'}*

}

## Событие onclose

Событие onclose вызывается в случае закрытия соединения вне зависимости от того, было оно закрыто сервером или браузером. Например, вы можете общаться по одному небезызвестному видеочату и собеседник закроет окно. Или же множество пользователей участвовали в онлайн-аукционе по продаже утопленных авто и торги закончились. В обоих случаях сработает событие onclose:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onclose = (event: CloseEvent) => {

if (event.wasClean){

console.log('Соединение закрыто корректно');

console.log(`Код закрытия - ${event.code}`);

console.log(`Причина закрытия - ${event.reason}`)

} else {

console.log('Соединение закрыто с кодом - '

}

}

Для onclose параметр event будет типа данных [CloseEvent](https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent). Одно из полей event указывает на закрытие соединения и называется wasClean. Если сокет был закрыт корректно, wasClean возвращает true, а если некорректно, к примеру по причине отсутствия сети, отображается false.

Конвенция кодов ошибок события onclose аналогична тем, о которых мы говорили [в предыдущем уроке](https://practicum.yandex.ru/trainer/react/lesson/17fa740a-b7eb-4fcf-9e27-de0ef1d8b573).

## Событие onerror

Это событие происходит, если во время соединения возникла ошибка:

Скопировать кодTSX

const ws = new WebSocket("ws://norma.nomoreparties.space/");

ws.onerror = (event: Event) => {

console.log(`Ошибка ${event.message}`)

}

Причины события onerror могут быть разные, например, если в процессе подключения прервалось интернет-соединение, или после подключения были отправлены данные, которые вызвали ошибку подключения.

Но не все ошибки вызовут событие onerror — некоторые из них передадутся с событием onclose. Разберёмся, почему так происходит.

Событие onerror сработает только при ошибке: пропало подключение, возникли проблемы с сервером и другие возможные причины, к примеру пользователь отправляет некорректные данные.

Событие onclose сработает при закрытии соединения: зашёл ли пользователь в свой бункер, где отсутствует сеть, или сам прекратил соединение при закрытии чата.

Поэтому onclose в любом случае отработает при закрытии соединения и ошибки он не ловит, он ловит именно закрытие соединение. Тем не менее можно обратиться к полям объекта event и получить информацию об ошибках. Получается, что onclose и onerror — не взаимозаменяемые слушатели.

Это всё, что нужно знать о типах событий WebSocket. В следующем уроке пойдём дальше и разберёмся, как получать и отправлять данные. А пока перейдём к практике.