JS Level 2

SIDE EFFECTS

React

На прошлой лекции мы завершили разработку CRUD-приложения, организовав всё хранение данных в памяти. Но все наши посты автоматически удаляются, стоит лишь нам обновить страницу.

Это не очень хорошо, поэтому давайте рассмотрим более реалистичный сценарий - данные должны храниться не в браузере (хотя и в браузере может храниться часть данных), а на сервере – в базе данных.

Поскольку основы работы с серверами и базами данных мы будем проходить только в Level 3, то в данном курсе мы будем использовать уже готовый сервер.

Web Application

Типичное веб-приложение устроено следующим образом:

Например, если вы открываете страницу www.ya.ru, то браузер загружает и запускает клиентскую часть приложения, которая общается с серверной частью приложения.

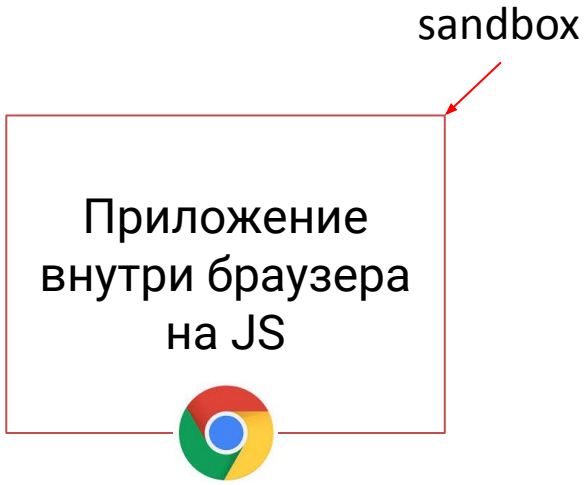
Web Application

Как на самом деле это работает: у вас есть две программы – браузер (на вашем компьютере) и серверное приложение (на сервере). Сервер – это просто компьютер, который в отличие от вашего компьютера постоянно включен и подключен к сети Интернет.

Эти два приложения (браузер и сервер) взаимодействуют между собой, пересылая по сети определённую информацию (представьте, что вы говорите с человеком по телефону - вы тоже пересылаете друг другу информацию посредством сети).

Браузер

При этом браузер помещает приложение в так называемую песочницу (sandbox):



Песочница – это ограничение приложение с точки зрения его возможностей.

Например, приложение, запущенное в браузере не может без вашего разрешения включить видео-камеру или микрофон на вашем ноутбуке.

В первую очередь, это сделано из соображений безопасности – чтобы клиентская часть веб-приложения не могла причинить существенный вред вам или вашему компьютеру.

Сервер

Для серверного приложения, в большинстве случаев, не существует никаких песочниц – ему предоставляется доступ ко всем возможностям ОС*: оно может создавать/удалять файлы, выполнять любые другие действия.

Приложение, работающее на сервере

Примечание*: на самом деле, и для серверных приложений есть ограничения. Существуют они тоже из соображений безопасности: например, приложение не может удалять какие-то критичные для работы файлы операционной системы или мешать работе других приложений (на сервере может быть много приложений).

Протоколы

Для того, чтобы передавать данные между клиентом и сервером (а они могут быть написаны на разных языках программирования), придумали набор соглашений, который позволяет клиенту и серверу понимать друг друга.

Протоколов достаточно много, но самым важным для нас с вами будет протокол HTTP.

HTTP

HTTP (HyperText Transfer Protocol) – это протокол передачи данных. Позволяет передавать практически любой тип контента.

Т.е. когда мы с вами рисуем вот такую картинку, то мы подразумеваем, что все данные передаются по протоколу HTTP:

Протокол

Что такое протокол? Протокол – это правила общения двух сторон. В нашем случае браузера и сервера. Вы можете воспринимать это как язык: например, если один человек говорит на английском, а другой на китайском – они вряд ли друг друга поймут.

Протокол устанавливает правила общения: говорим на таком-то языке, в таком-то формате, разрешено передавать такие-то сообщения.

Как всегда, мы будем достаточно упрощённо всё рассматривать, опуская некоторые детали, но мы вам настоятельно рекомендуем ознакомиться с теми ссылками, которые будут указаны на следующей странице.

Версии протокола

Ключевых версий на данный момент две (хотя уже существуют и более новые версии):

1.1: https://tools.ietf.org/html/rfc2616

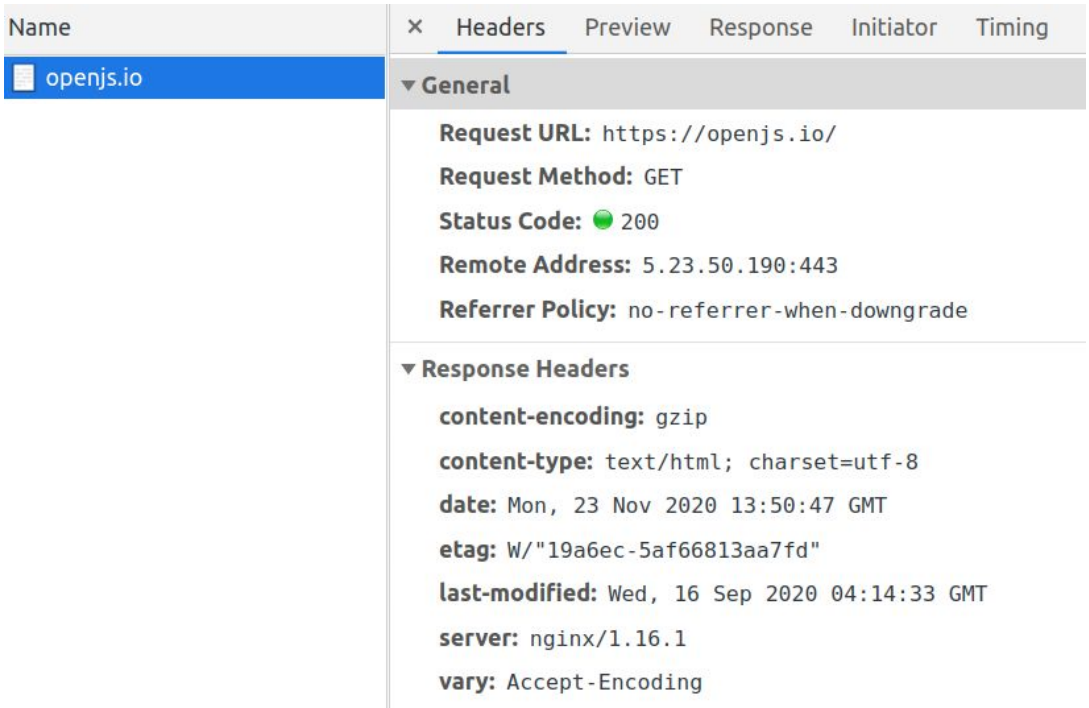
2.0: https://tools.ietf.org/html/rfc7540

Мы будем с вами рассматривать версию 1.1 (поскольку она самая распространённая), но потихоньку все переходят на версию 2.0.

HTTP 1.1

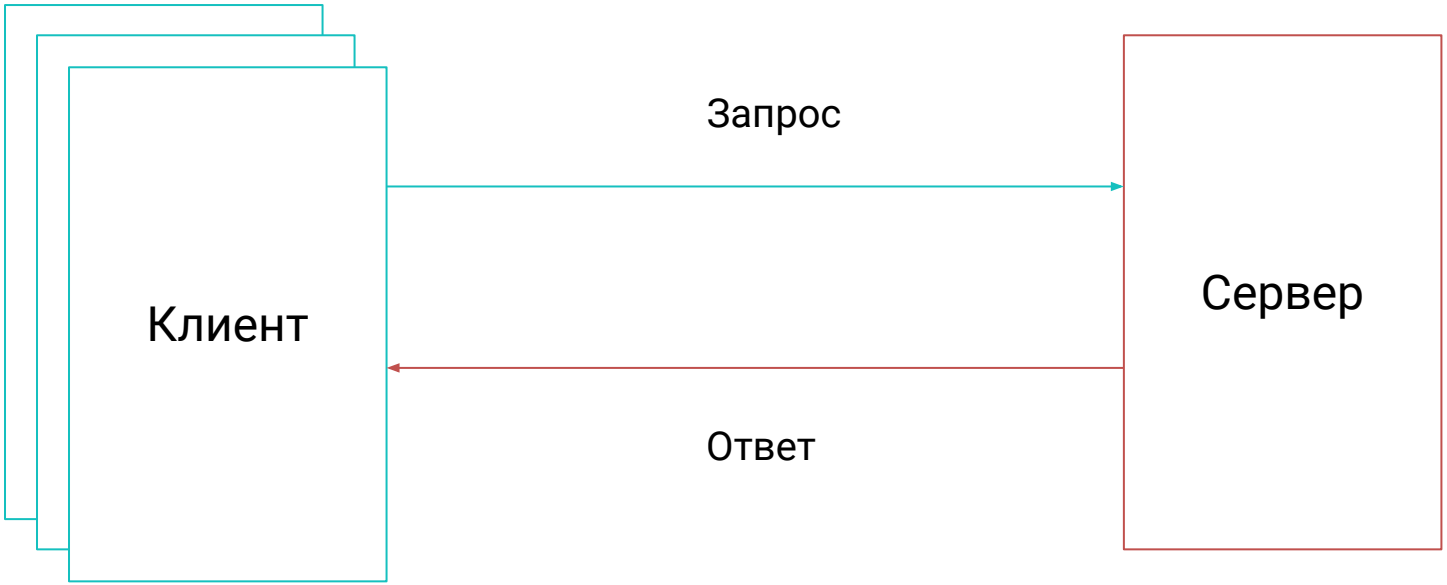
Версия 1.1 является текстовой. Что это значит? Это значит, что все данные, которые передаются, можно представить в виде текста.

Например, вы можете открыть вкладку Network, вбить какой-то адрес, например, https://alif-skills.pro и увидеть, как происходит передача данных:



HTTP 1.1

В рамках этого протокола общение строится в формате запрос-ответ, т.е. клиент должен что-то запросить у сервера, а сервер ему на это должен что-то ответить. Под клиентом в данном случае понимается веб-браузер:



Естественно, клиентов может быть много. Серверов, конечно же, тоже, но мы рассмотрим простейший случай, когда сервер только один.

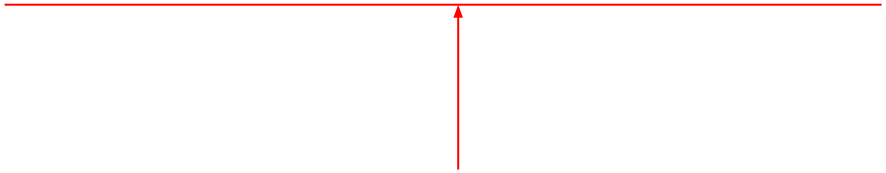
Message

И запрос, и ответ называют message (или сообщение). В рамках спецификации это выглядит вот так:

4.1 Message Types

HTTP messages consist of requests from client to server and responses from server to client.

```
HTTP-message    = Request | Response     ; HTTP/1.1 messages
```



Есть всего два типа сообщений: запрос (request) и ответ (response)

Запрос

Запрос состоит из трёх частей:

5 Request

A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use.

```
Request       = Request-Line              ; Section 5.1
                *(( general-header        ; Section 4.5
                 | request-header         ; Section 5.3
                 | entity-header ) CRLF)  ; Section 7.1
                CRLF
                [ message-body ]          ; Section 4.3
```

1. Request Line (строка запроса) – что мы хотим получить от сервера
2. Headers (заголовки) – мета-данные
3. Message Body (тело) – что мы передаём на сервер (если, например, загружаем файл)

Запрос

Чтобы лучше понимать общую идею, представьте, что изначально всё было придумано так, как будто сервер показывает вам файлы и каталоги находящиеся на удалённом компьютере, а вы, используя HTTP, говорите, содержимое какого каталога или какой файл ему показать.

Ответ

Ответ состоит из трёх частей:

6 Response

After receiving and interpreting a request message, a server responds with an HTTP response message.

```
Response      = Status-Line                ; Section 6.1
                *(( general-header         ; Section 4.5
                 | response-header         ; Section 6.2
                 | entity-header ) CRLF)   ; Section 7.1
                CRLF
                [ message-body ]           ; Section 7.2
```

1. Status Line (строка запроса) – насколько успешно выполнился наш запрос
2. Headers (заголовки) – мета-данные
3. Message Body (тело) – что сервер передаёт нам в ответ (например, содержимое файла index.html)

Данные

В принципе, это почти всё, что нам нужно знать: есть запрос и ответ, у каждого по три части.

Если частей всего три, значит данные мы можем передавать тремя разными способами (или их комбинацией):

1. В Request Line
2. В Headers
3. В Message Body

И получать тоже можем тремя разными способами (или их комбинацией):

1. В Status Line
2. В Headers
3. В Message Body

Инструменты

Остаётся лишь один вопрос: как нам из JS этот самый запрос отправить? Обратите внимание: React и Redux всё равно, поскольку React отвечает за работу с DOM, Redux – за состояние. Поэтому тут нам придётся разбираться самим.

Браузер предоставляет нам целых два инструмента:

1. XMLHttpRequest
2. fetch

Работу с XMLHTTPRequest мы рассматривали в рамках курса Level 1, сейчас же поговорим о работе с fetch.

fetch

Используется fetch следующим образом:

fetch(URL, options);

Где:

- URL - это адрес, на который мы отправляем запрос
- options - это различные опции (возможность указывать метод запроса заголовки, тело)

Server

Ключевой вопрос: на какой URL нам отправлять данные? Мы написали сервер, который запускается по адресу: http://localhost:9999.

Чтобы его запустить, вам нужно скачать файл, положить в каталог с проектом и выполнить следующую команду: node rserver.min.js. В ответ вы увидите:

Сервер вы можете скачать по адресу: https://alif-skills.pro/media/rserver.min.js

Method

По умолчанию, fetch использует запросы с методом GET (если не указывать иное).

Q: что такое метод?

A: в рамках протокола HTTP определён набор "слов", которые называются методами. У этих методов есть смысловая нагрузка, например, метод GET используется для получения данных, а POST – для отправки. Кроме того, в рамках конкретных методов есть и ограничения, например, метод GET должен содержать пустое тело (т.е. вы можете отправить данные только в Request Line или в заголовках).

Method

5.1.1 Method

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

```
    Method         = "OPTIONS"                ; Section 9.2
                   | "GET"                    ; Section 9.3
                   | "HEAD"                   ; Section 9.4
                   | "POST"                   ; Section 9.5
                   | "PUT"                    ; Section 9.6
                   | "DELETE"                 ; Section 9.7
                   | "TRACE"                  ; Section 9.8
                   | "CONNECT"                ; Section 9.9
                   | extension-method
    extension-method = token
```

Как определить, как метод нужно использовать и какой адрес?

API

На самом деле, решаете это не вы, а человек, который пишет сервер.

Именно он вам говорит нечто вроде:

- Для получения списка постов отправляй GET-запрос на http://localhost:9999/api/posts
- Для добавления отправляй POST-запрос на http://localhost:9999/api/posts, в теле запроса должен быть JSON с данными (id = 0)
- Для обновления отправляй POST-запрос на http://localhost:9999/api/posts, в теле запроса должен быть JSON с данными (id != 0)
- Для удаления отправляй запрос DELETE на http://localhost:9999/api/posts/{id}, где {id} – это идентификатор поста
- И т.д.

API

Это называется API – определение программного взаимодействия нашего приложения и сервера.

Хорошо, давайте смотреть, выглядеть это должно примерно вот так:

```
fetch('http://localhost:9999/api/posts');
```

Но теперь вопросы:

1. Где это писать?
2. Как получить данные?

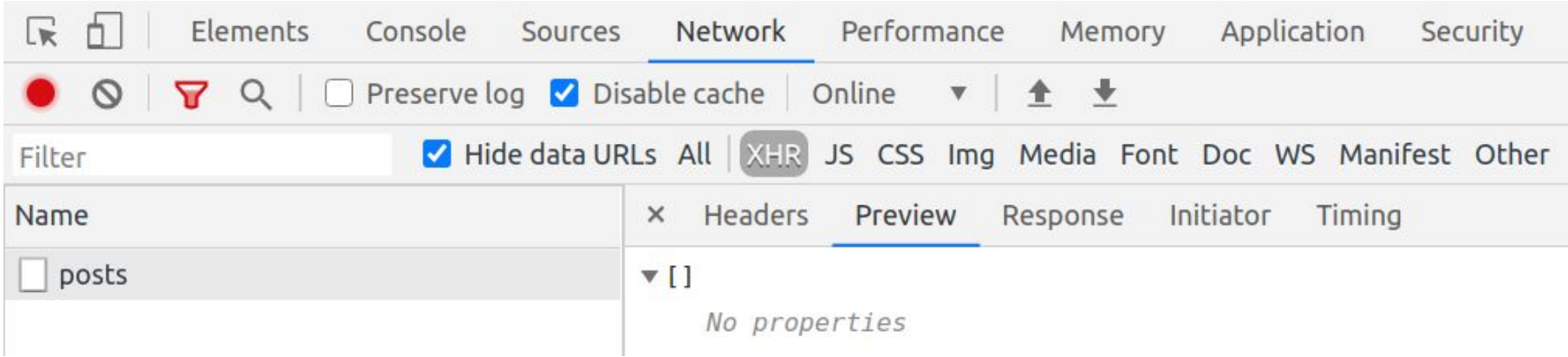
API

Для начала давайте попробуем это написать в index.js, чтобы увидеть, что хотя бы что-то работает:

```
8
9   fetch('http://localhost:9999/api/posts');
10
11  ReactDOM.render(
12    <React.StrictMode>
13      <Provider store={store}>
14        <App />
15      </Provider>
16    </React.StrictMode>,
17    document.getElementById('root')
18  );
```

API

В DevTools браузера необходимо открыть вкладку Network и поставить фильтр XHR (именно туда попадают запросы XMLHttpRequest и fetch):



Во вкладке Preview браузер отображает предпросмотр ответа (если в качестве ответа приходит JSON, то пытается обработать его).

JSON

JSON (JavaScript Object Notation) – это специальный текстовый формат передачи данных, один из самых популярных на сегодняшний день.

В чём суть: программные системы могут быть написаны на разных языках, например, мы можем писать фронтенд на JS, а мобильные приложения пишутся на Kotlin или Swift. Сервер вообще может быть написан на Java. При этом всем этим системам нужно обмениваться данными. JSON – как раз-таки такой формат, который позволяет это делать.

JSON

Нам не с вами пока не нужно знать про внутреннее устройство JSON (хотя мы настоятельно рекомендуем с ним ознакомиться) по двум причинам:

1. fetch позволяет нам "распарсить" ответ и превратить его в JS-объект (в нашем случае - массив постов)
2. есть глобальный объект JSON с методом parse (позволяет превратить JSON-документ в JS объект) и методом stringify (позволяет превратить JS объект в строку)

JSON

В JSON могут присутствовать следующие типы данных:

1. Объекты (имена свойств должны быть в двойных кавычках)
2. Массивы
3. Строки (должны быть в двойных кавычках)
4. Числа
5. Boolean (true и false)
6. Null (null)

Других типов не разрешено. Т.е. undefined нельзя, кроме того, нельзя функции, методы и т.д. JSON служит только для передачи данных. Этого нам должно быть достаточно для работы. Смотрим.

Promise

Осталось понять, как до JSON'а (из ответа) добраться. fetch – это современное API, поэтому большинство вызовов возвращает Promise.

Promise

Promise – это функция-конструктор, которая позволяет создавать специализированные объекты (типа Promise, конечно же).

Эти объекты могут находиться всего в трёх состояниях:

- pending (ожидание)
- fulfilled (выполнено)
- rejected (отклонено)

Изначально объект, созданный с помощью Promise (далее для краткости будем говорить просто Promise) находится в состоянии pending. Из этого состояния он может перейти в fulfilled или rejected и перейти может только один раз (либо не перейти вовсе).

Promise

В асинхронном мире Promise позволяет программировать "будущее" в стиле, вот когда случится это событие, мы будем делать одно, а если произойдёт ошибка, то будем делать другое.

Promise

У самого Promise, есть три метода, в которые можно передавать callback'и:

- then – срабатывает тогда, когда Promise перешёл в состояние fulfilled
- catch – когда Promise перешёл в состояние rejected
- finally – когда Promise изменил своё состояние (перешёл из pending в любое другое)

Поскольку fetch возвращает Promise, попробуем с ним поработать в этом ключе:

```
9  fetch('http://localhost:9999/api/posts')
10   .then(response => {
11     console.log(response);
12   })
13   .catch(error => {
14     console.log(error);
15   });
```

Promise


▼Response {type: "cors", url: "http://localhost:9999/api/posts", redirected: false, status: 200, ok: true, …}
body: (...)
bodyUsed: false
▶headers: Headers {}
ok: true
redirected: false
status: 200
statusText: "OK"
type: "cors"
url: "http://localhost:9999/api/posts"
▶__proto__: Response

Promise

Правильная работа с Promise из fetch для получения "обработанного" тела ответа выглядит следующим образом:

```
8
9   fetch('http://localhost:9999/api/posts')
10    .then(response => {
11      if (!response.ok) {
12        throw new Error('bad http status');
13      }
14
15      return response.json();
16    })
17    .then(body => {
18      console.log(body);
19    })
20    .catch(error => {
21      console.log(error);
22    });
23
```

ok

Метод ok возвращает boolean, отвечающий на вопрос: вернулся ответ с кодом от 200 до 299 или нет.

Коды ответа – это специальные числа в нескольких диапазонах, которые информируют клиента о том, как завершился запрос:

- 100-199 – информационные коды
- 200-299 – успешно
- 300-399 – перенаправление (клиенту нужно перенаправить запрос на другой адрес)
- 400-499 – ошибка клиента (клиент сделал неверный запрос)
- 500-599 – ошибка сервера (на сервере произошла ошибка при обработке запроса)

Генерация ошибок

В JS мы можем самостоятельно генерировать ошибки. При генерации ошибки прерывается нормальный ход выполнения программы (т.е. последующие строки не выполняются). Сгенерированная ошибка обрушает нашу программу, если не выполнено одно из следующих условий:

1. Код, в котором генерируется ошибка, находится внутри специально оформленного блока try-catch
2. Код, в котором генерируется ошибка, находится внутри обработчиков then, catch

Важно, что выкидывание ошибки в then, приведёт к срабатыванию следующего catch (а он в нашей цепочке единственный).

then

Если что-то вернуть из then, то оно попадёт в следующий then из цепочки.

Если вернуть Promise, то JS сам вызовет then как раз на этом Promise (если Promise перейдёт в состояние fulfilled). А если Promise перейдёт в состояние rejected, то JS вызовет catch. Поэтому нам достаточно написать один catch на всё и два then. response.json() возвращает Promise, который пытается распарсить тело ответа как JSON документ и превратить его в объект.

Кроме того, если вернуть значение (не Promise), то оно автоматически завернётся в Promise (который сразу перейдёт в состояние fulfilled) и результат мы получим уже в следующем then.

Promise

Это всё хорошо, но как теперь это положить в state? Достаточно просто: у store есть свойство dispatch, через которое мы можем получить тот самый dispatch для отправки action'ов.

Но какой action нам dispatch'ить?

index.js

src > store > actions > index.js > editCancel

```
export const POSTS_REQUEST = 'POSTS_REQUEST';
export const POSTS_SUCCESS = 'POSTS_SUCCESS';
export const POSTS_FAIL = 'POSTS_FAIL';

export const POST_EDIT_SUBMIT = 'POST_EDIT_SUBMIT';
export const POST_EDIT_CANCEL = 'POST_EDIT_CANCEL';
export const POST_EDIT_CHANGE = 'POST_EDIT_CHANGE';
export const POST_LIKE = 'POST_LIKE';
export const POST_REMOVE = 'POST_REMOVE';
export const POST_HIDE = 'POST_HIDE';
export const POST_EDIT = 'POST_EDIT';
export const POST_SHOW = 'POST_SHOW';

export const postsRequest = () => {
  return {
    type: POSTS_REQUEST,
    payload: {},
  };
};

export const postsSuccess = (items) => {
  return {
    type: POSTS_SUCCESS,
    payload: {items},
  };
};

export const postsFail = (error) => {
  return {
    type: POSTS_FAIL,
    payload: {error},
  };
};

```

Почему именно так? Потому что сетевые запросы – не всегда выполняются мгновенно и не всегда завершаются успешно. Поэтому у нас будет три action'а:

- POSTS_REQUEST – начало загрузки
- POSTS_SUCCESS – успешная загрузка
- POSTS_FAIL – ошибка загрузки

state

Ключевое: делая сетевые запросы, всегда предусматривайте две вещи:

1. Длительность (на это время пользователю нужно отображать какой-нибудь loader или анимацию, чтобы он понимал, что данные загружаются и нужно подождать)
2. Ошибку (сервер не смог ответить, у пользователя нет подключения к сети и т.д.)

И вы должны их учитывать и обрабатывать в state:

```
export const initialState = {
  posts: {
    items: [],
    loading: false,
    error: null,
  },
  edited: empty,
};
```

reducer

```
39  export const reducer = (state = initialState, action) => {
40    switch (action.type) {
41      case POSTS_REQUEST:
42        return reducePostsRequest(state, action);
43      case POSTS_SUCCESS:
44        return reducePostsSuccess(state, action);
45      case POSTS_FAIL:
46        return reducePostsFail(state, action);

68  const reducePostsRequest = (state, action) => {
69    return {
70      ...state,
71      posts: {...state.posts, loading: true, error: null},
72    };
73  };
74
75  const reducePostsSuccess = (state, action) => {
76    return {
77      ...state,
78      posts: {items: action.payload.items, loading: false, error: null},
79    };
80  };
81
82  const reducePostsFail = (state, action) => {
83    return {
84      ...state,
85      posts: {...state.posts, loading: false, error: action.payload.error},
86    };
87  };
```

state

Поскольку мы поменяли структуру state, то и в Wall придётся всё поменять (как и в других reducer'ах – но об этом позже):

index.js

src > index.js > ...

```
10  store.dispatch(postsRequest());
11  fetch('http://localhost:9999/api/posts')
12    .then(response => {
13      if (!response.ok) {
14        throw new Error('bad http status');
15      }
16
17      return response.json();
18    })
19    .then(body => {
20      store.dispatch(postsSuccess(body));
21    })
22    .catch(error => {
23      store.dispatch(postsFail(error));
24    });
```

Обратите внимание, как и когда мы dispatch'им action'ы.

Wall

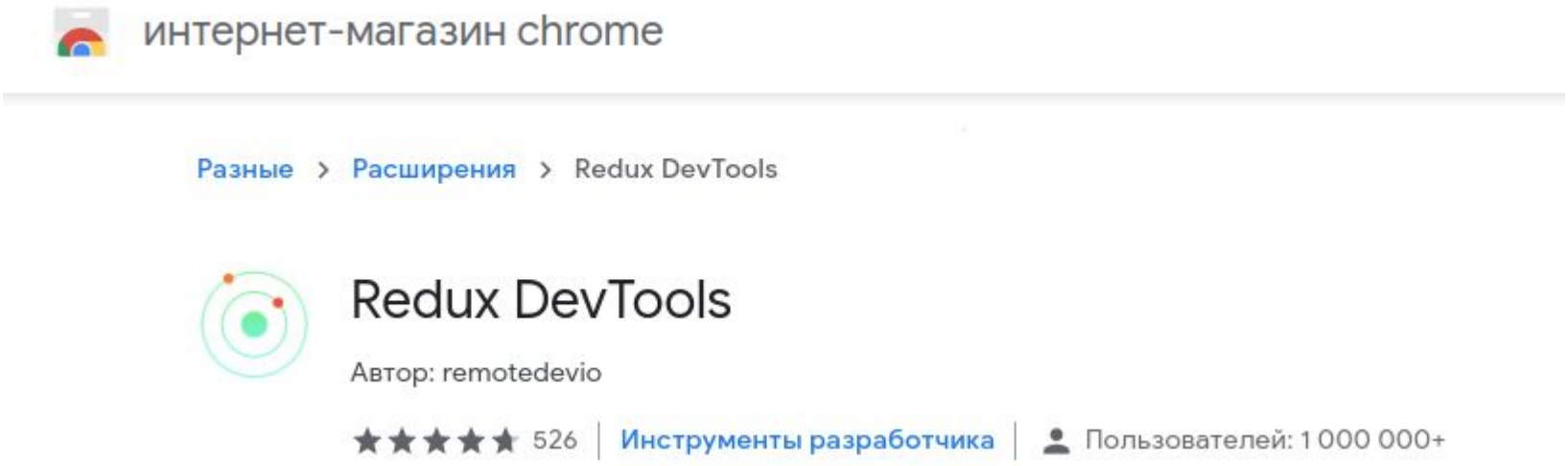
Реализация в Wall достаточно ленивая: мы не отрисовываем список, если у нас идёт загрузка или произошла ошибка (это допустимо для простых приложений):

```
7   function Wall() {
8     const {items, loading, error} = useSelector((state) => state.posts, shallowEqual);
9     const dispatch = useDispatch();
10
11    const handleReload = () => {
12      dispatch(postsRequest());
13    };
14
15    if (loading) {
16      return <>Идёт загрузка данных...</>;
17    }
18
19    if (error) {
20      return <>
21        Произошла ошибка. <button onClick={handleReload}>Повторить запрос?</button>
22      </>;
23    }
24
25    return (
26      <>
27        <PostForm/>
28        <div>
29          {items.map(o => <Post key={o.id} post={o}/>)}
30        </div>
31      </>
32    );
33  }
```

Redux Dev Tools

Несмотря на то, что наше решение работает, смотреть, что реально происходит - совсем не удобно. Потому что не видим, как отправляются action'ы, какие отправляются action'ы и т.д.

Поэтому есть специальное расширение, которое позволяет анализировать, как и что меняется:



Redux DevTools

Чтобы оно заработало, необходимо соответствующим образом инициализировать store:

index.js

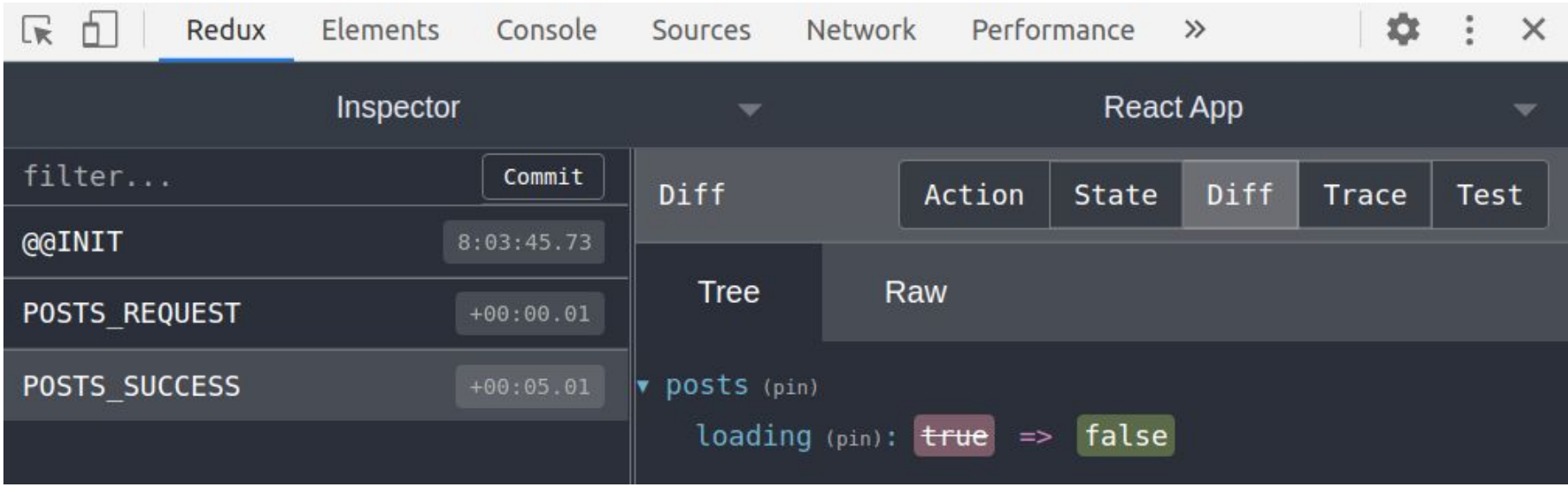
src > store > index.js > ...

```
1  import { createStore } from 'redux';
2  import { reducer } from './reducers';
3
4  const store = createStore(
5    reducer,
6    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),
7  );
8  export default store;
```

Если вкладка Redux не появилась в DevTools, перезапустите полностью браузер.

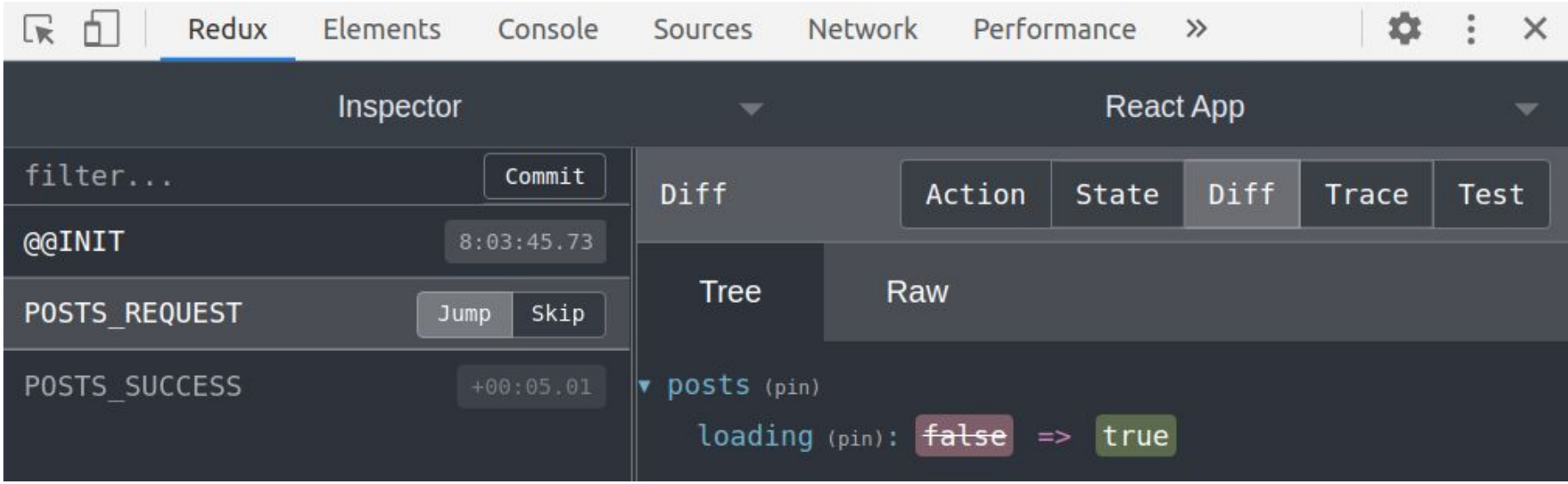
Redux DevTools

Это расширение позволяет вам посмотреть, как менялся state:



Redux DevTools

И не только посмотреть, но и "вернуть" приложение на определённый state в прошлом:



Возможностей у этого инструмента действительно много и мы настоятельно рекомендуем вам ознакомиться с ним поближе.

state

Теперь давайте подумаем: ведь с сохранением будет то же самое? Когда пользователь будет жать на кнопку сохранить, то данные должны обновиться не только в его браузере, но и на сервере (т.е. должен быть запрос на сервер):

```
30  export const initialState = {
31    posts: {
32      items: [],
33      loading: false,
34      error: null,
35    },
36    edited: {
37      item: empty,
38      loading: false,
39      error: null,
40    },
41  };
```

Но, где и как этот запрос делать? Понятно, что нам нужно что-то сохранять только когда мы нажимаем на кнопку Ok. Но где это делать?

state

Возможные варианты:

1. В самом компоненте
2. ~~В reducer'е~~
3. В action creator'е

Нужно запомнить: не стоит делать в reducer'е.

Q: почему?

A: потому что reducer – это чистая функция, которая никак не взаимодействует с внешним миром. Её задача работать быстро и только на основании state и action генерировать новый state.

state

Можно это (side effects) делать в компоненте, но для этого сначала нужно создать соответствующие action'ы:

```
5  export const POST_SAVE_REQUEST = 'POST_SAVE_REQUEST';
6  export const POST_SAVE_SUCCESS = 'POST_SAVE_SUCCESS';
7  export const POST_SAVE_FAIL = 'POST_SAVE_FAIL';
```

```
40 export const postSaveRequest = () => {
41   return {
42     type: POST_SAVE_REQUEST,
43     payload: {},
44   };
45 };
46
47 export const postSaveSuccess = (item) => {
48   return {
49     type: POST_SAVE_SUCCESS,
50     payload: {item},
51   };
52 };
53
54 export const postSaveFail = (error) => {
55   return {
56     type: POST_SAVE_FAIL,
57     payload: {error},
58   };
59 };
```

reducers

```
43  export const reducer = (state = initialState, action) => {
44    switch (action.type) {
45      case POSTS_REQUEST:
46        return reducePostsRequest(state, action);
47      case POSTS_SUCCESS:
48        return reducePostsSuccess(state, action);
49      case POSTS_FAIL:
50        return reducePostsFail(state, action);
51      case POST_SAVE_REQUEST:
52        return reducePostSaveRequest(state, action);
53      case POST_SAVE_SUCCESS:
54        return reducePostSaveSuccess(state, action);
55      case POST_SAVE_FAIL:
56        return reducePostSaveFail(state, action);
```

reducers

```
 99  const reducePostSaveRequest = (state, action) => {
100    return {
101      ...state,
102      edited: {...state.edited, loading: true, error: null},
103    };
104  };
105
106  const reducePostSaveSuccess = (state, action) => {
107    return {
108      ...state,
109      edited: {item: empty, loading: false, error: null},
110    };
111  };
112
113  const reducePostSaveFail = (state, action) => {
114    return {
115      ...state,
116      edited: {...state.edited, loading: false, error: action.payload.error},
117    };
118  };
```

reducers

```
160  const reduceChange = (state, action) => {
161    const {item} = state.edited;
162    const {payload: {name, value}} = action;
163    if (name === 'tags') {
164      const parsed = value.split(' ');
165      return {
166        ...state,
167        edited: {...state.edited, item: {...item, [name]: parsed}},
168      }
169    }
170
171    if (name === 'photo' || name === 'alt') {
172      const prop = name === 'photo' ? 'url' : name;
173      return {
174        ...state,
175        edited: {...state.edited, item: {...item, photo: {...item.photo, [prop]: value}}},
176      }
177    }
178
179    return {
180      ...state,
181      edited: {...state.edited, item: {...item, [name]: value}},
182    }
183  };
```

```
5   export default function PostForm() {
6     const dispatch = useDispatch();
7     const {item, loading, error} = useSelector((state) => state.edited, shallowEqual);
8     const firstFocusEl = useRef(null);
9
10    const handleSubmit = (evt) => {
11      evt.preventDefault();
12      dispatch(postSaveRequest());
13
14      fetch('http://localhost:9999/api/posts', {
15        method: 'POST',
16        headers: {
17          'Content-Type': 'application/json',
18        },
19        body: JSON.stringify(item),
20      })
21        .then(response => {
22          if (!response.ok) {
23            throw new Error('bad http status');
24          }
25
26          return response.json();
27        })
28        .then(body => {
29          dispatch(postSaveSuccess(body));
30          // а вот тут неплохо бы запросить все посты
31        })
32        .catch(error => {
33          dispatch(postSaveFail(error));
34        });
35
36      firstFocusEl.current.focus();
37    };
```

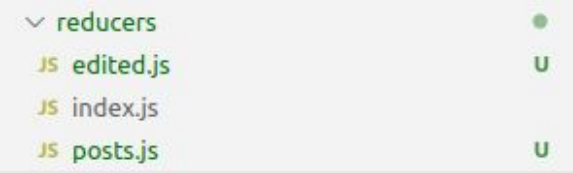
Проблемы

У такого решения сразу две проблемы:

1. Как запросить после сохранения посты с сервера? Сервер, конечно, нам присылает пост после сохранения с выставленным id и мы можем dispatch'ить action, который добавит в начало списка постов. Но что если мы захотим снова запросить все посты? Не дублировать же логику.
2. Слишком уж сложной стала работа со state'ом в reducer'ах.

Проблемы

Давайте начнём решать проблемы с конца. Разберёмся с тем, как упростить reducer'ы. Мы на прошлой лекции говорили о том, что можно разделить весь state (не всегда это удаётся) на кусочки и отдать на управление отдельным reducer'ам:



edited.js

src > store > reducers > edited.js > ...

```
import {
  POST_EDIT_CANCEL, POST_EDIT_CHANGE,
  POST_SAVE_FAIL,
  POST_SAVE_REQUEST,
  POST_SAVE_SUCCESS,
} from '../actions';

const empty = {
  id: 0,
  author: {
    id: 1,
    avatar: 'https://alif-skills.pro/logo_js.svg',
    name: 'Alif Skills',
  },
  content: '',
  photo: null,
  hit: false,
  likes: 0,
  likedByMe: false,
  hidden: false,
  tags: null,
  created: 0,
};

export const initialState = {
  item: empty,
  loading: false,
  error: null,
};

export const editedReducer = (state = initialState, action) => {
  switch (action.type) {
    case POST_SAVE_REQUEST:
      return reducePostSaveRequest(state, action);
    case POST_SAVE_SUCCESS:
      return reducePostSaveSuccess(state, action);
    case POST_SAVE_FAIL:
      return reducePostSaveFail(state, action);
    case POST_EDIT_CANCEL:
      return reduceCancel(state, action);
    case POST_EDIT_CHANGE:
      return reduceChange(state, action);
    default:
      return state;
  }
};
```

edited.js

src > store > reducers > edited.js > ...

```
46
47  const reducePostSaveRequest = (state, action) => {
48    return {
49      ...state, loading: true, error: null,
50    };
51  };
52
53  const reducePostSaveSuccess = (state, action) => {
54    return {
55      ...state, item: empty, loading: false, error: null,
56    };
57  };
58
59  const reducePostSaveFail = (state, action) => {
60    return {
61      ...state, loading: false, error: action.payload.error,
62    };
63  };
64
65  const reduceCancel = (state, action) => {
66    return {
67      ...state, item: empty, loading: false, error: null,
68    }
69  };
70
71  const reduceChange = (state, action) => {
72    const {item} = state;
73    const {payload: {name, value}} = action;
74    if (name === 'tags') {
75      const parsed = value.split(' ');
76      return {
77        ...state, item: {...item, [name]: parsed},
78      }
79    }
80
81    if (name === 'photo' || name === 'alt') {
82      const prop = name === 'photo' ? 'url' : name;
83      return {
84        ...state, item: {...item, photo: {...item.photo, [prop]: value}},
85      }
86    }
87
88    return {
89      ...state, item: {...item, [name]: value},
90    }
91  };
```

posts.js

src > store > reducers > posts.js > ...

```
import {
  POST_LIKE, POST_REMOVE,
  POSTS_FAIL,
  POSTS_REQUEST,
  POSTS_SUCCESS
} from '../actions';

export const initialState = {
    items: [],
    loading: false,
    error: null,
};

export const postsReducer = (state = initialState, action) => {
  switch (action.type) {
    case POSTS_REQUEST:
      return reducePostsRequest(state, action);
    case POSTS_SUCCESS:
      return reducePostsSuccess(state, action);
    case POSTS_FAIL:
      return reducePostsFail(state, action);
    case POST_LIKE:
      return reduceLike(state, action);
    case POST_REMOVE:
      return reduceRemove(state, action);
    default:
      return state;
  }
};
```

posts.js

src > store > reducers > posts.js > ...

```
30
31  const reducePostsRequest = (state, action) => {
32    return {
33      ...state, loading: true, error: null,
34    };
35  };
36
37  const reducePostsSuccess = (state, action) => {
38    return {
39      ...state, items: action.payload.items, loading: false, error: null,
40    };
41  };
42
43  const reducePostsFail = (state, action) => {
44    return {
45      ...state, loading: false, error: action.payload.error,
46    };
47  };
48
49  const reduceLike = (state, action) => {
50    const {items} = state;
51    return {
52      ...state,
53      posts: items.map((o) => {
54        if (o.id !== action.payload.id) {
55          return o;
56        }
57
58        return {...o, likes: o.likedByMe ? o.likes - 1 : o.likes + 1, likedByMe: !o.likedByMe};
59      })
60    }
61  };
62
63  const reduceRemove = (state, action) => {
64    const {items} = state;
65    return {
66      ...state,
67      posts: items.filter((o) => o.id !== action.payload.id),
68    }
69  };
```

index.js

src > store > reducers > index.js > ...

```
import {
  POST_EDIT,
} from '../actions';
import { combineReducers } from 'redux';
import { postsReducer } from './posts';
import { editedReducer } from './edited';

const appReducer = combineReducers({
  posts: postsReducer,
  edited: editedReducer,
})

export const rootReducer = (state, action) => {
  switch (action.type) {
    case POST_EDIT:
      return reduceEdit(state, action);
    default:
      return appReducer(state, action);
  }
};

const reduceEdit = (state, action) => {
  const {posts} = state;
  const post = posts.items.find(o => o.id === action.payload.id);
  if (post === undefined) {
    return state;
  }

  return {
    ...state,
    edited: post,
  }
};
```

Store

index.js

src > store > index.js > ...

```
1  import { createStore } from 'redux';
2  import { rootReducer } from './reducers';
3
4  const store = createStore(
5    rootReducer,
6    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),
7  );
8  export default store;
```

combineReducers

combineReducers – удобная функция, которая позволяет "разделить" reducer'ы, передавая им свой "кусочек" state. Но при этом любой action проходит через все reducer'ы.

Важно: обычно это редко показывают, но специально выбрали такой пример: у нас есть action, при обработке которого нужен весь state целиком, а не кусочки.

Такой action можно обработать в отдельном reducer'е (rootReducer), который имеет доступ ко всему state. Он же прокидывает action дальше, если это не тот action, который должен обработать только он (это упрощённая версия).

Side Effects

Реализованный нами вариант работает, но давайте обсудим, чем он (этот вариант) плох и попробуем сделать его лучше.

useEffect

Для начала вынесем код получения всех постов из index.js в useEffect компонента Wall:

```
useEffect(() => {
  dispatch(postsRequest());
  fetch('http://localhost:9999/api/posts')
    .then(response => {
      if (!response.ok) {
        throw new Error('bad http status');
      }

      return response.json();
    })
    .then(body => {
      dispatch(postsSuccess(body));
    })
    .catch(error => {
      dispatch(postsFail(error));
    });
}, [dispatch]);
```

useEffect

Теперь немного преобразуем его, используя async/await:

```
15    useEffect(() => {
16      const effect = async () => {
17        try {
18          dispatch(postsRequest());
19          const response = await fetch('http://localhost:9999/api/posts')
20          if (!response.ok) {
21            throw new Error('bad http status');
22          }
23
24          const body = await response.json();
25          dispatch(postsSuccess(body));
26        } catch (e) {
27          dispatch(postsFail(e));
28        }
29      };
30      effect();
31    }, [dispatch]);
```

Ключевое здесь следующее: в useEffect нельзя передавать асинхронную функцию, поэтому мы передаём функцию, внутри которой находится вызов асинхронной функции.

useEffect

async/await позволяют нам в более простом виде работать с Promise: await дожидается состояния fulfilled Promise, если Promise перейдёт в состояние rejected, то будет выброшена ошибка (которую мы и перехватим в catch).

Проблема

Теперь вернёмся к нашей проблеме: после сохранения поста в PostForm, мы не можем запросить полный список постов, поскольку тогда нам придётся дублировать логику, которую мы только что написали в Wall. То же самое при возникновении ошибки – кнопка "Повторить запрос" – ничего не грузит.

Конечно же можно, используя уже изученные нами техники сделать так, чтобы PostForm как-то уведомлял Wall, после чего тот перезапускал эффект для получения данных и т.д.

Но есть способ проще – давайте просто вынесем это всё в action creator'ы.

action creator'ы

Но, как вы видите, этим action creator'ам нужен dispatch, который приходится передавать в виде аргумента:

```
118  export const loadPosts = async (dispatch) => {
119    try {
120      dispatch(postsRequest());
121      const response = await fetch('http://localhost:9999/api/posts')
122      if (!response.ok) {
123        throw new Error('bad http status');
124      }
125
126      const body = await response.json();
127      dispatch(postsSuccess(body));
128    } catch (e) {
129      dispatch(postsFail(e));
130    }
131  };
132
133  export const savePost = async (dispatch, item) => {…
135  };
```

action creator'ы

Поэтому вызов этих action creator'ов выглядит "наоборот":

```
7  function Wall() {
8    const {items, loading, error} = useSelector((state) => state.posts, shallowEqual);
9    const dispatch = useDispatch();
10
11   const handleReload = () => {
12     loadPosts(dispatch);
13   };
14
15   useEffect(() => {
16     loadPosts(dispatch);
17   }, [dispatch]);
18
```

Мы не в dispatch передаём вызов action creator'а, а в action creator передаём dispatch.

action creator'ы

При этом мы можем удобно вызывать одни action creator'ы из других:

```
133 export const savePost = async (dispatch, item) => {
134   try {
135     dispatch(postSaveRequest());
136
137     const response = await fetch('http://localhost:9999/api/posts', {
138       method: 'POST',
139       headers: {
140         'Content-Type': 'application/json',
141       },
142       body: JSON.stringify(item),
143     })
144
145     if (!response.ok) {
146       throw new Error('bad http status');
147     }
148
149     const body = await response.json();
150     dispatch(postSaveSuccess(body));
151
152     loadPosts(dispatch);
153   } catch (e) {
154     dispatch(postSaveFail(e));
155   }
156 };
```

action creator'ы

И вызов в PostForm сократится:

```
10  const handleSubmit = (evt) => {
11    evt.preventDefault();
12    savePost(dispatch, item);
13    firstFocusEl.current.focus();
14  };
```

action creator'ы

Несмотря на то, что это рабочая схема, она не очень красивая, поэтому неплохо бы под рукой иметь какой-то инструмент, который решает эту проблему.

middleware

Для этого применяется концепция middleware – фактически, это промежуточная функция, которая может быть исполнена до того, как action попадёт в store или после.

Давайте попробуем сами написать подобную функцию, а затем воспользуемся готовым инструментом.

logger middleware

Самое простое middleware, которое мы можем написать – это logger, который просто будет печатать в консоль информацию о том, как action к нам пришёл:

middleware.js

src > store > middleware.js > ...

```
1  export const logger = (store) => (next) => (action) => {
2    console.log(action);
3    return next(action);
4  };
```

Давайте разбираться:

1. Первая функция на вход принимает store (именно для этого мы в своё время напрямую работали в ним)
2. Вторая принимает next – это следующий middleware (если их несколько или dispatch, если наш middleware всего один)
3. И третья – это функция, в которую попадает уже action.

logger middleware

Самая вложенная функция и будет выполняться при каждом dispatch'е, а благодаря концепции замыканий мы будем видеть в этой функции и store, и next, и action.

middleware.js

src > store > middleware.js > ...

```
1  export const logger = (store) => (next) => (action) => {
2    console.log(action);
3    return next(action);
4  };
```

На самом деле, там будет не совсем store, а объект, состоящий из getState и dispatch.

Настройка middleware

Для применения middleware есть специальная функция applyMiddleware, а также функция compose, которая позволит нам скомпоновать middleware с Redux DevTools:

index.js

src > store > index.js > ...

```
1  import { applyMiddleware, compose, createStore } from 'redux';
2  import { rootReducer } from './reducers';
3  import { logger } from './middleware';
4
5  const store = createStore(
6    rootReducer,
7    compose(
8      applyMiddleware(logger),
9      window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),
10   ),
11 );
12 export default store;
```

middleware

Теперь при применении мы увидим следующее:

```
[HMR] Waiting for update signal from WDS...                    log.js:24
▶ {type: "POSTS_REQUEST", payload: {…}}                        middleware.js:2
▶ {type: "POSTS_SUCCESS", payload: {…}}                        middleware.js:2
>
```

middleware

Что всё это нам даёт? Это даёт достаточно интересные возможности: что, если в качестве action закидывать функцию, а не объект? Тогда middleware может сделать следующее:

middleware.js

src > store > middleware.js > ...

```
1   export const logger = (store) => (next) => (action) => {
2     console.log(action);
3     return next(action);
4   };
5
6   export const fn = ({dispatch, getState}) => (next) => (action) => {
7     if (typeof action === 'function') {
8       return action(dispatch);
9     }
10
11    return next(action);
12  };
```

store

index.js

src > store > index.js > ...

```
1  import { applyMiddleware, compose, createStore } from 'redux';
2  import { rootReducer } from './reducers';
3  import { logger, fn } from './middleware';
4
5  const store = createStore(
6    rootReducer,
7    compose(
8      applyMiddleware(logger, fn),
9      window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),
10   ),
11 );
12 export default store;
```

Wall

```
11  const handleReload = () => {
12    // loadPosts(dispatch);
13    dispatch(loadPosts);
14  };
15
16  useEffect(() => {
17    // loadPosts(dispatch);
18    dispatch(loadPosts);
19  }, [dispatch]);
```

Уже достаточно неплохо, но всё равно отличается от того, как мы обычно вызываем dispatch (ведь мы кладём туда вызов функции):

```
dispatch(postSaveRequest());
```

Это не проблема, мы можем просто переделать наши action creator'ы так, чтобы они возвращали функцию:

```
118 export const loadPosts = () => async (dispatch) => {
119   try {
120     dispatch(postsRequest());
121     const response = await fetch('http://localhost:9999/api/posts')
122     if (!response.ok) {
123       throw new Error('bad http status');
124     }
125
126     const body = await response.json();
127     dispatch(postsSuccess(body));
128   } catch (e) {
129     dispatch(postsFail(e));
130   }
131 };
132
133 export const savePost = (item) => async (dispatch) => {
134   try {
135     dispatch(postSaveRequest());
136
137     const response = await fetch('http://localhost:9999/api/posts', {
138       method: 'POST',
139       headers: {
140         'Content-Type': 'application/json',
141       },
142       body: JSON.stringify(item),
143     })
144
145     if (!response.ok) {
146       throw new Error('bad http status');
147     }
148
149     const body = await response.json();
150     dispatch(postSaveSuccess(body));
151
152     dispatch(loadPosts());
153   } catch (e) {
154     dispatch(postSaveFail(e));
155   }
156 };
```

Wall/PostForm

И вызывать их так же, как и остальные action creator'ы:

```
11  const handleReload = () => {
12    // loadPosts(dispatch);
13    dispatch(loadPosts());
14  };
15
16  useEffect(() => {
17    // loadPosts(dispatch);
18    dispatch(loadPosts());
19  }, [dispatch]);
```

```
10  const handleSubmit = (evt) => {
11    evt.preventDefault();
12    //savePost(dispatch, item);
13    dispatch(savePost(item));
14    firstFocusEl.current.focus();
15  };
```

Redux Thunk

На самом деле, не нужно писать подобный middleware каждый раз, поскольку есть уже готовый – Redux Thunk:

npm install redux-thunk

index.js

src > store > index.js > ...

```
1  import { applyMiddleware, compose, createStore } from 'redux';
2  import { rootReducer } from './reducers';
3  import { logger } from './middleware';
4  import thunk from 'redux-thunk';
5
6  const store = createStore(
7    rootReducer,
8    compose(
9      applyMiddleware(logger, thunk),
10     window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),
11   ),
12 );
13 export default store;
```

Больше ничего не изменится.

Redux Thunk

Поскольку в middleware также присылается getState, мы можем немного упростить наш код и в PostForm не посылать никаких item (ведь их можно напрямую достать из store):

```
10  const handleSubmit = (evt) => {
11    evt.preventDefault();
12    //savePost(dispatch, item);
13    //dispatch(savePost(item));
14    dispatch(savePost());
15    firstFocusEl.current.focus();
16  };
```

```
133 export const savePost = () => async (dispatch, getState) => {
134   try {
135     const {item} = getState().edited;
136
137     dispatch(postSaveRequest());
```

Таким же образом мы можем избавиться от reduceEdit, создав action creator, который сам ходит в state, по id находит там нужный item и диспатчит его уже в editedReducer (попробуйте это сделать самостоятельно).

Advanced

Мы с вами написали практически полнофункциональное приложение, которому не хватает пары вещей:

- навигации (React Router)
- механизмов безопасности:
  - идентификации
  - аутентификации
  - авторизации

Эти темы мы разберём на следующем курсе, когда научимся писать сервера на Node.js.

ИТОГИ

Итоги

Сегодня мы рассмотрели вопросы работы с Side Effects с Redux Thunk.

ДОМАШНЕЕ ЗАДАНИЕ

Орг.моменты

Домашнего задания к сегодняшней лекции не предусмотрено – то, что вы пройдёте все шаги и добьётесь работоспособности приложения и будет признаком того, что вы освоили курс.

Помните, что вы учитесь не для нас и не для бота, а для себя.

P.S.: а если вы не добьётесь работоспособности приложения, а просто пролистаете лекцию, то на следующем курсе ничего не поймёте (особенно когда мы будем связывать сервер с React-приложением 😊).

Спасибо за внимание

alif skills

2023г.