JS Level 3

при поддержке

open (js)
school

REACT FORMS

open (js)
school

React

На прошлой лекции мы научились удалять элементы, а также частично модифицировать их (через кнопку like).

Сегодня наша задача - научиться редактировать элементы списка (создавать их). И делать мы это будем не на игрушечном примере с одним полем, а сразу со сложными полями.

Формы

Итак, мы хотим для начала создавать посты. Общая идея достаточно простая: мы создаём форму с полем content, в которое пользователь и вводит контент будущего поста:

JS PostForm.js ×

src > components > PostForm > JS PostForm.js > ...

```
1   import React from 'react';
2
3   export default function PostForm() {
4     return (
5       <form>
6         <textarea></textarea>
7         <button>Ok</button>
8       </form>
9     )
10  };
```

Отлично, теперь два ключевых вопроса:

1. Как обрабатывать данные формы
2. Где их хранить (данные) в компоненте PostForm или в Wall?

События

Обрабатывать достаточно просто: у формы есть событие submit (когда форма отправляется), в React - onSubmit:

PostForm.js

src > components > PostForm > PostForm.js > ...

```
1   import React from 'react';
2
3   export default function PostForm() {
4     const handleSubmit = (ev) => {
5       ev.preventDefault();
6     };
7
8     return (
9       <form onSubmit={handleSubmit}>
10        <textarea></textarea>
11        <button>Ok</button>
12      </form>
13    )
14  };
```

По умолчанию, отправка формы приводит к перезагрузке страницы (см.лекции Level 1), а это нам не нужно. Поэтому мы вызываем метод preventDefault на объекте события, который отменяет поведение по умолчанию.

Несмотря на то, что ev - это не настоящий объект события (React подкладывает нам объект SyntheticEvent), вызов preventDefault приведёт к вызову preventDefault на оригинальном объекте события.

Lift Up & Lift Down

Теперь самый главный вопрос - где и как хранить данные до того момента пока пользователь не нажал на кнопку Ok.

Когда мы рассматривали стену (Wall) и Post, мы пришли к тому, что сам пост - это просто props для компонента Post (т.е. мы "подняли" состояние из компонента Post в компонент Wall). Это называется State Lifting Up - мы поднимаем состояние в родительский компонент тогда, когда хозяином данных становится родитель.

Сейчас же зададим себе вопрос - а нужны ли данные из формы родителю до тех пор, пока пользователь не нажал на кнопку Ок? На самом деле - нет. Поэтому мы можем это состояние "спустить" в сам компонент PostForm. Это называется State Lifting Down - мы опускаем состояние в дочерний компонент, потому что родительскому компоненту эти данные не нужны - ему нужно только уведомление о том, что пользователь нажал на Ок.

PostForm

PostForm.js

src > components > PostForm > PostForm.js > ...

```
1  import React, {useState} from 'react';
2
3  export default function PostForm({onSave}) {
4    const [content , setContent] = useState('');
5
6    const handleSubmit = (ev) => {
7      ev.preventDefault();
8      onSave({
9        content,
10     });
11   };
12
13   return (
14     <form onSubmit={handleSubmit}>
15       <textarea></textarea>
16       <button>Ok</button>
17     </form>
18   )
19 };
```

Давайте разбираться. С тем, что мы вынесли в props onSave - понятно: сюда нам будет родитель присылать callback.

С useState тоже всё понятно, используем состояние для хранения данных.

Controlled Components

А теперь немного теории: поля ввода - достаточно сложные элементы. Они сами хранят своё состояние без всякого React'а. Т.е. вы вводите туда текст и они просто изначально так реализованы, что этот текст никуда не девается и хранится там в поле value.

Но в React'е принято делать немного по-другому: мы на каждое изменение текста в поле будем обрабатывать событие и устанавливать новое состояние. А значение из состояния будем отрисовывать в поле value поля ввода.

Звучит немного сложновато, но давайте посмотрим на практике.

Controlled Components

Теперь все изменения и синхронизация DOM происходят через React и через state и мы полностью контролируем компонент (а не он контролирует нас).

Это как с постом - нажатие на любой кнопке в Post приводит к тому, что запускается функция в Wall и в пост передаются новые props. Здесь то же самое, только вместо поста - textarea.

Обратите внимание, setContent мы вызываем без prevState по одной простой причине, из объекта события нам придёт уже новый текст, а старое состояние нас не интересует.

ev.target - это объект, на котором произошло событие, value - это значение его поля.

JS Wall.js ×

src > components > Wall > JS Wall.js > ...

```
69    const handleSave = (post) => {
70      setPosts((prevState) => [...prevState, {
71        id: Date.now(), // просто генерируем id из даты (потом научимся правильно)
72        author: {
73          avatar: 'https://lms.openjs.io/logo_js.svg',
74          name: 'OpenJS',
75        },
76        content: post.content,
77        photo: null,
78        hit: false,
79        likes: 0,
80        likedByMe: false,
81        hidden: false,
82        tags: [],
83        created: 1603774800,
84      }])
85    };
86
87    return (
88      <>
89        <PostForm onSave={handleSave} />
90        <div>
91          {posts.map(o => <Post
92                                key={o.id}
93                                post={o}
94                                onLike={handlePostLike}
95                                onRemove={handlePostRemove}
96                                onHide={handleToggleVisibility}
97                                onShow={handleToggleVisibility} />)}
98        </div>
99      </>
100   );
101 }
102
103 export default Wall;
```

Тестируем

Wall.js

src > components > Wall > Wall.js > ...

```
69    const handleSave = (post) => {
70      setPosts((prevState) => [...prevState, {
71        id: Date.now(), // просто генерируем id из даты (потом научимся правильно)
72        author: {
73          avatar: 'https://lms.openjs.io/logo_js.svg',
74          name: 'OpenJS',
75        },
76        content: post.content,
77        photo: null,
78        hit: false,
79        likes: 0,
80        likedByMe: false,
81        hidden: false,
82        tags: [],
83        created: 1603774800,
84      }])
85    };
```

Ключевых момента два: [...prevState, {xxx}] - создаёт новый массив на базе старого с помощью оператора spread + мы добавляем в конец массива созданный нами объект. Объект мы заполняем данными по умолчанию + прописываем тот контент, что ввёл пользователь.

Тестируем

Здесь важно заметить, что посты, конечно, обычно добавляются наверх, и имя автора и аватар мы должны будем откуда-то брать, а не хардкодить (когда вы авторизуетесь в социальной сети, то ваши данные приходят оттуда, с сервера).

Добавление

Следующий момент достаточно интересный - собирать такой большой объект - достаточно тяжело. Почему бы сразу просто не "закинуть" его в PostForm?

Wall.js

src > components > Wall > Wall.js > ...

```
69    const handleSave = (post) => {
70      setPosts((prevState) => [...prevState, {...post}])
71    };
72
73    return (
74      <>
75        <PostForm onSave={handleSave} />
76        <div>
77          {posts.map(o => <Post
78                                key={o.id}
79                                post={o}
80                                onLike={handlePostLike}
81                                onRemove={handlePostRemove}
82                                onHide={handleToggleVisibility}
83                                onShow={handleToggleVisibility} />)}
84        </div>
85      </>
86    );
87  }
88
89  export default Wall;
```

PostForm.js

src > components > PostForm > PostForm.js > ...

```jsx
import React, {useState} from 'react';

export default function PostForm({onSave}) {
  const [post , setPost] = useState({
      id: Date.now(), // просто генерируем id из даты (потом научимся правильно)
      author: {
        avatar: 'https://lms.openjs.io/logo_js.svg',
        name: 'OpenJS',
      },
      content: '',
      photo: null,
      hit: false,
      likes: 0,
      likedByMe: false,
      hidden: false,
      tags: [],
      created: Date.now(),
  });

  const handleSubmit = (ev) => {
    ev.preventDefault();
    onSave(post);
  };

  const handleChange = (ev) => {
    const {value} = ev.target;
    setPost((prevState) => ({...prevState, content: value}));
  };

  return (
    <form onSubmit={handleSubmit}>
      <textarea value={post.content} onChange={handleChange}></textarea>
      <button>Ok</button>
    </form>
  )
};
```

Добавление

Отдельно нужно остановиться на setPost: поскольку в state у нас теперь большой объекты, мы обязаны использовать prevState и копировать объект. Чтобы стрелочная функция не путала возвращаемый объект {} с телом функции (которое тоже пишется в {}), мы заключаем объект ещё в ():

```
25  const handleChange = (ev) => {
26    const {value} = ev.target;
27    setPost((prevState) => ({...prevState, content: value}));
28  };
```

Это просто синтаксис JS, ничего связанного с React здесь нет.

React

И чтобы добавлять в начало, а не в конец, нам достаточно переставить местами аргументы:

```
const handleSave = (post) => {
  setPosts((prevState) => [{...post}, ...prevState])
};
```

onChange vs onInput

Если вы проходили курс Level 1, то помните, что мы там использовали событие input, а не change. Т.к. change срабатывает только тогда, когда поле ввода теряет фокус.

В React всё немного не так: onChange срабатывает и на событие input, поэтому мы используем onChange.

Несколько полей

Ок, задавать одно поле - неплохо, но что будет, если мы захотим задавать несколько? Например, задавать теги? Тут нужно решить, как мы их будем задавать - возьмём самый простой сценарий, когда пользователь просто вводит их через пробел: #deadline #homework.

Идея достаточно простая: мы можем создать отдельный обработчик на изменение тегов, поскольку теги нужно разрезать по пробелу и удалять символ #:

```
const handleTagsChange = (ev) => {
  const {value} = ev.target;
  setTags(value);
};

return (
  <form onSubmit={handleSubmit}>
    <textarea value={post.content} onChange={handleChange}></textarea>
    <input value={tags} onChange={handleTagsChange}></input>
    <button>Ok</button>
  </form>
)
```

Несколько полей

Пока этот код ничего не удаляет и не разрезает, он просто хранит её так, как ввёл пользователь.

```
const handleTagsChange = (ev) => {
  const {value} = ev.target;
  setTags(value);
};

return (
  <form onSubmit={handleSubmit}>
    <textarea value={post.content} onChange={handleChange}></textarea>
    <input value={tags} onChange={handleTagsChange}></input>
    <button>Ok</button>
  </form>
)
```

И мы можем при каждом вводе просто обновлять state поста:

```
const handleTagsChange = (ev) => {
  const {value} = ev.target;
  setTags(value);
  const parsed = value.split(' ');
  setPost((prevState) => ({...prevState, tags: parsed}));
};
```

Несколько полей

Обязательно смотрите через инструменты, что у вас получается:

Несколько полей

А можем и не при каждом, а только при нажатии на кнопку сохранить (но обычно, конечно, стараются при каждом). Другой вопрос - а нужно ли нам вообще тогда tags?

Ведь можно сделать вот так:

```
const handleTagsChange = (ev) => {
  const {value} = ev.target;
  const parsed = value.split(' ');
  setPost((prevState) => ({...prevState, tags: parsed}));
};

return (
  <form onSubmit={handleSubmit}>
    <textarea value={post.content} onChange={handleChange}></textarea>
    <input value={post.tags?.join(' ')} onChange={handleTagsChange}></input>
    <button>Ok</button>
  </form>
)
```

Что это за tags? - это optional chaining, возможность, которая появилась в новом JS. Если вдруг в tags будет null или undefined, то не произойдёт ошибки из-за того, что мы на null вызываем метод join.

Объединяем

Теперь, если посмотреть, то изменение контента и тегов отличается только логикой того, что теги надо парсить. Можно ли как-то объединить обработчики?

```
const handleChange = (ev) => {
  const {name, value} = ev.target;
  if (name === 'tags') {
    const parsed = value.split(' ');
    setPost((prevState) => ({...prevState, [name]: parsed}));
    return;
  }

  setPost((prevState) => ({...prevState, [name]: value}));
};

return (
  <form onSubmit={handleSubmit}>
    <textarea name="content" value={post.content} onChange={handleChange}></textarea>
    <input name="tags" value={post.tags?.join(' ')} onChange={handleChange}></input>
    <button>Ok</button>
  </form>
)
```

Мы ввели дополнительный атрибут name, по которому и определяем, это "специальный" случай (когда надо что-то обрабатывать) или обычный, когда достаточно поставить value.

Объединяем

Синтаксис {[name]: value} называется вычисляемые поля. Т.е. если в переменной name будет значение 'tags', то это превратится в {tags: value}.

ИТОГИ

Итоги

Сегодня мы рассмотрели вопросы добавления на примере сложных объектов, поля которых могут быть null.

ДОМАШНЕЕ ЗАДАНИЕ

ДЗ: Теги

Реализация тегов, описанная в лекции приводит к тому, что #, вводимые пользователем, не удаляются:

ДЗ: Теги

А если поле пустое (пользователь всё стёр), то в теги попадает пустая строка (а должен быть null):

ДЗ: Теги

А ещё можно добавлять "пустые" теги, если ставить несколько пробелов:

ДЗ: Теги

Всё это вроде мелочи, но этим мелочи определяют качество вашей работы и профессионализм.

Поэтому ваша задача заключается в том, чтобы устранить эти недочёты.

ДЗ: Фото

С текстом, конечно, здорово, но хотелось бы научиться добавлять и фото.

Напоминаем, фото у нас или null, если картинки нет:

```
photo: null,
```

Или объект, если есть картинка:

```
photo: {
  url: 'https://lms.openjs.io/openjs.jpg',
  alt: 'openjs logo',
},
```

Обратите внимание, что alt может быть пустым, но не наоборот: т.е. нет смысла в alt, если url пустой - тогда всё фото надо делать null (подумайте, где это лучше сделать).

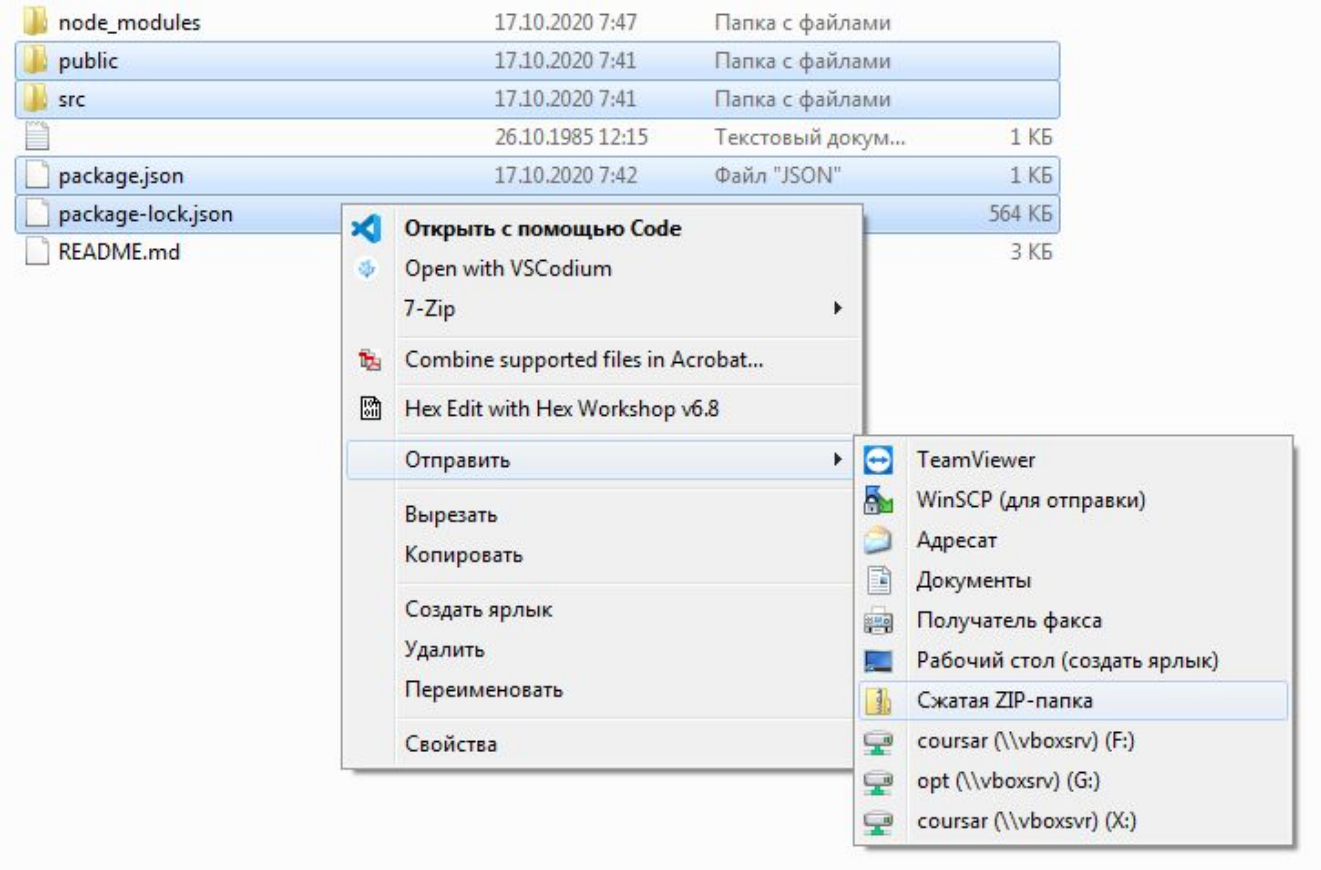
ДЗ: Фото

Что мы хотим: добавьте 2 поля: фото и описание, чтобы пользователь мог подставлять ссылку с фото и описание:

```
return (
  <form onSubmit={handleSubmit}>
    <textarea name="content" placeholder="content" value={post.content} onChange={handleChange}></textarea>
    <input name="tags" placeholder="tags" value={post.tags?.join(' ')} onChange={handleChange}></input>
    <input name="photo" placeholder="photo"></input>
    <input name="alt" placeholder="alt"></input>
    <button>Ok</button>
  </form>
)
```

Как сдавать ДЗ

Вам нужно запаковать в zip-архив ваш проект те файлы и каталоги, которые указаны на скриншоте ниже. Для этого выберите их, нажмите правую кнопку мыши и выберите Отправить –> Сжатая ZIP-папка:



Как сдавать ДЗ

Полученный архив загружаете в личном кабинете пользователя.

Важно: учитывается только последняя отправленная попытка.

Спасибо за внимание

alif academy совместно с aims

2020г.

open (js)
school