

JS Level 2



REACT FORMS



React

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

Сегодня наша задача - научиться редактировать элементы списка. 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 (содержимое) будущего поста:

```
JS PostForm.js X
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 - обработчик `onSubmit`:

```
JS PostForm.js X
src > components > PostForm > JS PostForm.js > ...
1  import React from 'react';
2
3  export default function PostForm() {
4    const handleSubmit = (evt) => {
5      evt.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 `evt` - 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 **Ok**? 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 **Ok**.



PostForm

JS PostForm.js X

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

```
1  import React, {useState} from 'react';
2
3  export default function PostForm({onSave}) {
4    const [content, setContent] = useState('');
5
6    const handleSubmit = (evt) => {
7      evt.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

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

Diagram illustrating the flow of data in a controlled component:

- The `handleChange` function is called when the `onChange` event occurs on the `textarea`.
- The function updates the `content` state via `setContent(value)`.
- The `value` prop of the `textarea` is updated to match the new `content` state.

Labels in the diagram:

- "изменяет" (changes) points to the `onChange` event.
- "вызывает" (calls) points to the `handleChange` function.

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.

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



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

```
69   const handlePostSave = (post) => {
70     setPosts((prevState) => [...prevState, {
71       id: Date.now(), // просто генерируем id из даты (потом научимся правильно)
72       author: {
73         avatar: 'https://alif-skills.pro/media/logo_alif.svg',
74         name: 'Alif Skills',
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 edited={edited} onSave={handlePostSave} />
90     <div>
91       {posts.map(o => <Post
92         key={o.id}
93         post={o}
94         onLike={handlePostLike}
95         onRemove={handlePostRemove}
96         onHide={handleTogglePostVisibility}
97         onShow={handleTogglePostVisibility} />)}
98     </div>
99   </>
100 );
101 }
102
103 export default Wall;
```



Тестируем

JS Wall.js x

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

```
69   const handlePostSave = (post) => {
70     setPosts((prevState) => [...prevState, {
71       id: Date.now(), // просто генерируем id из даты (потом научимся правильно)
72       author: {
73         avatar: 'https://alif-skills.pro/media/logo_alif.svg',
74         name: 'Alif Skills',
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:

1. `[...prevState, {xxx}]` - создаёт новый массив на базе старого с помощью оператора spread + мы добавляем в конец массива созданный нами объект;
2. 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`?

JS Wall.js



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

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



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

```
1  import React, {useState} from 'react';
2
3  export default function PostForm({onSave}) {
4    const [post, setPost] = useState({
5      id: Date.now(), // просто генерируем id из даты (потом научимся правильно)
6      author: {
7        avatar: 'https://alif-skills.pro/media/logo_alif.svg',
8        name: 'Alif Skills',
9      },
10     content: '',
11     photo: null,
12     hit: false,
13     likes: 0,
14     likedByMe: false,
15     hidden: false,
16     tags: [],
17     created: Date.now(),
18   });
19
20   const handleSubmit = (evt) => {
21     evt.preventDefault();
22     onSave(post);
23   };
24
25   const handleChange = (evt) => {
26     const {value} = evt.target;
27     setPost((prevState) => ({...prevState, content: value}));
28   };
29
30   return (
31     <form onSubmit={handleSubmit}>
32       <textarea value={post.content} onChange={handleChange}></textarea>
33       <button>Ok</button>
34     </form>
35   )
36   };
```



Добавление

Отдельно нужно остановиться на `setPost`: поскольку в `state` у нас теперь большой объект, 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 = (evt) => {  
26 |     const {value} = evt.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

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



Несколько полей

Обязательно смотрите через инструменты, что у вас получается:

hooks



```
▼ State: {author: {...}, content: "", created: 1604378779038, ...}
  id: 1604378779038
  ▶ author: {avatar: ...}
    content: ""
    photo: null
    hit: false
    likes: 0
    likedByMe: false
    hidden: false
  ▶ tags: ["#homework", "#deadline"]
    created: 1604378779038
    new entry: ""
  State: ""
```



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 = (evt) => {
  const {value} = evt.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 них вызываем метод `join`. Такие цепочки можно строить достаточно большой длины.



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 = (evt) => {  
  const {name, value} = evt.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}`.



Типичная ошибка

Здесь стоит отметить, что используя новые конструкции JS, вы всегда должны проверять, а что будет в противном случае (т.е. в `tags` будет `null` или `undefined`). Если мы посмотрим в консоль, то увидим:

```
✖ Warning: A component is changing an uncontrolled input of type undefined index.js:1
  to be controlled. Input elements should not switch from uncontrolled to controlled (or
  vice versa). Decide between using a controlled or uncontrolled input element for the
  lifetime of the component. More info: https://fb.me/react-controlled-components
    in input (at PostForm.js:47)
    in form (at PostForm.js:45)
    in PostForm (at Wall.js:75)
    in Wall (at App.js:7)
    in div (at App.js:6)
    in App (at src/index.js:9)
    in StrictMode (at src/index.js:8)
```

Q: в чём дело?

A: дело в том, что `post.tags?.join(' ')` возвращает `undefined`, если `post.tags` будет `null` или `undefined`. Но значением поля `value` не может быть `undefined`. Поэтому обычно добавляют оператор `||` (или `??` для более современной версии JS), который работает следующим образом: возвращает первый операнд, который `truthy` или последний, если оба `falsy`.



Очищение ввода

Теперь ключевой вопрос: как очистить поля ввода? Ведь после нажатия на кнопке `Ok` у нас остаётся введённым текст.

Всё достаточно просто: нужно в `handleSubmit` после `onSave` выставить в `state` "чистый" объект (см. следующий слайд).



Очищение ввода

```
const handleSubmit = (evt) => {
  evt.preventDefault();
  const parsed = post.tags?.map(o => o.replace('#', '')).filter(o => o.trim() !== '') || [];
  const tags = parsed.length !== 0 ? parsed : null;
  onSave({...post, id: Date.now(), created: Date.now(), tags, photo: post.photo?.url ? {alt: '', ...post.photo} : null});
  setPost({
    id: 0,
    author: {
      avatar: 'https://alif-skills.pro/media/logo_alif.svg',
      name: 'Alif Skills',
    },
    content: '',
    photo: null,
    hit: false,
    likes: 0,
    likedByMe: false,
    hidden: false,
    tags: null,
    created: 0,
  });
};
```

Обратите внимание, мы заменили `id` и `created` на `0` и только в `handleSubmit` выставляем их. Так же, как и с предыдущей ошибкой вы должны были догадаться по сообщениям в консоли, что иначе у вас будут одинаковые `id` для разных элементов. Конечно, после очистки они будут уже не одинаковые, но дата создания всё равно будет неверной.



React

Конечно, глядя на этот код возникает желание вынести чистый объект один раз и не делать одно и то же в `useState` и в `handleSubmit` (см. следующий слайд).



```
const [post, setPost] = useState({
  id: 0,
  author: {
    avatar: 'https://alif-skills.pro/media/logo_alif.svg',
    name: 'Alif Skills',
  },
  content: '',
  photo: null,
  hit: false,
  likes: 0,
  likedByMe: false,
  hidden: false,
  tags: null,
  created: 0,
});
```

```
const handleSubmit = (evt) => {
  evt.preventDefault();
  const parsed = post.tags?.map(o => o.replace('#', '')).filter(o => o.trim() !== '') || [];
  const tags = parsed.length !== 0 ? parsed : null;
  onSave({...post, id: Date.now(), created: Date.now(), tags, photo: post.photo?.url ? {alt: '', ...post.photo} : null});
  setPost({
    id: 0,
    author: {
      avatar: 'https://alif-skills.pro/media/logo_alif.svg',
      name: 'Alif Skills',
    },
    content: '',
    photo: null,
    hit: false,
    likes: 0,
    likedByMe: false,
    hidden: false,
    tags: null,
    created: 0,
  });
};
```



React

Давайте попробуем это сделать и посмотрим, к чему это приведёт:

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

export default function PostForm({onSave}) {
  const [post, setPost] = useState(empty);

  const handleSubmit = (evt) => {
    evt.preventDefault();
    const parsed = post.tags?.map(o => o.replace('#', '')).filter(o => o.trim() !== '') || [];
    const tags = parsed.length !== 0 ? parsed : null;
    onSave({...post, id: Date.now(), created: Date.now(), tags, photo: post.photo?.url ? {alt: '', ...post.photo} : null});
    setPost(empty);
  };
};
```



React

Это работает, но нужно быть достаточно аккуратным, поскольку если вы случайно поменяете сам объект `empty` – то ничего хорошего не будет.

Для этого обычно используют создание копии через `{...empty}`, но здесь вас подстерегает опасность: таким образом создаётся поверхностная копия, которая не копирует объекты, хранящиеся в полях. Например, поле `author` у всех скопированных подобным образом объектов будет указывать на один и тот же объект.

Соответственно, некоторые используют либо `JSON.parse(JSON.stringify())` (плохое решение), либо используют собственные/библиотечные функции полного копирования.

Кстати, в новых версиях JS появился метод `structuredClone`, который позволяет удобно создавать копии, но пока его можно встретить не часто.



useRef

С нашей формой добавления вроде всё достаточно неплохо, за одним исключением: после добавления фокус (то, где расположен курсор) остаётся в последнем поле, а неплохо бы переставить его на первое (обязательно следите за такими "мелочами").

Фокус – это функциональность из мира DOM, а не React. Поэтому в React для этого придумали специальный хук, который позволяет в том числе "добираться" до DOM элементов.



useRef

Идея `useRef` достаточно проста – он позволяет хранить ссылку на объект (не обязательно DOM Element), и изменение этой ссылки не приводит к перерисовке компонента (перерендерингу).

Как использовать – см. следующий слайд.



useRef

```
export default function PostForm({onSave}) {
  const [post, setPost] = useState(empty);
  const firstFocusEl = useRef(null); // начальное значение

  const handleSubmit = (evt) => {
    evt.preventDefault();
    const parsed = post.tags?.map(o => o.replace('#', '')).filter(o => o.trim() !== '') || [];
    const tags = parsed.length !== 0 ? parsed : null;
    onSave({...post, id: Date.now(), created: Date.now(), tags, photo: post.photo?.url ? {alt: '', ...post.photo} : null});
    setPost(empty);
    firstFocusEl.current.focus(); // через свойство current получаем доступ и устанавливаем focus
  };

  const handleChange = (evt) => { ...
  };

  return (
    // через ref = устанавливаем ссылку на элемент (React сам установит ссылку)
    <form onSubmit={handleSubmit}>
      <textarea ref={firstFocusEl} name="content" placeholder="content" value={post.content} onChange={handleChange}></textarea>
      <input name="tags" placeholder="tags" value={post.tags?.join(' ') || ''} onChange={handleChange}></input>
      <button>Ok</button>
    </form>
  )
}
```



Читабельность кода

Наш код работает, но вы могли заметить, что строки очень длинные и их тяжело читать. Поэтому не ленитесь и форматируйте свой код так, чтобы его удобно было читать, например:

```
onSave({
  ...post,
  id: Date.now(),
  created: Date.now(),
  tags,
  photo: post.photo?.url ? {alt: '', ...post.photo} : null
});
setPost(empty);
firstFocusEl.current.focus(); // через свойство current получаем доступ и устанавливаем focus
};
```



Читабельность кода

Кроме того, закрывайте пустые теги:

```
return (  
  // через ref = устанавливаем ссылку на элемент (React сам установит ссылку)  
  <form onSubmit={handleSubmit}>  
    <textarea  
      ref={firstFocusEl}  
      name="content"  
      placeholder="content"  
      value={post.content}  
      onChange={handleChange}/>  
    <input  
      name="tags"  
      placeholder="tags"  
      value={post.tags?.join(' ') || ''}  
      onChange={handleChange}  
    />  
    <button>Ok</button>  
  </form>  
)
```



Читабельность кода

Обратите внимание: мы не сразу написали идеальный код (такого не бывает), мы пришли к нему в процессе и навели "красоту" тогда, когда код стал работать.

Старайтесь всегда следовать такому подходу:

1. Сначала добиваемся того, что код работает
2. Затем "вычищаем" его (делаем красивым и аккуратным)



React

Теперь давайте поговорим о редактировании. С редактированием есть два подхода:

1. На редактирование вы делаете отдельную форму
2. Вы выполняете редактирование в той же форме, что и добавление

Мы рассмотрим с вами второй вариант, т.к. первый будет лишь его вариацией.



React

Итак поехали: чтобы отредактировать пост, нам нужно каким-то образом в форму редактирования передать тот пост, который мы собираемся редактировать. Конечно же, пока мы это можем сделать только через `props`. Но до этого нам нужно добавить кнопку "Изменить" в пост и соответствующие функции в `Wall` (см. следующий слайд):

```
return (  
  <article>  
    <header>  
      <img src={author.avatar} className="Post-avatar" width="50" height="50" alt={author.name}/>  
      <h5>{author.name}</h5>  
      <button onClick={handleRemove}>удалить</button>  
      <button onClick={handleHide}>скрыть</button>  
      <button onClick={handleEdit}>изменить</button>  
      <div>{post.created}</div>  
      {post.hit && <span>HIT</span>}  
    </header>  
  </article>  
);  
  
const handleEdit = () => {  
  onEdit(post.id);  
};
```



```

function Wall(props) {
  > const [posts, setPosts] = useState([ ...
    const [edited, setEdited] = useState();

  > const handlePostLike = (id) => { ...
    };

  > const handlePostRemove = (id) => { ...
    };

  > const handleTogglePostVisibility = (id) => { ...
    };

    const handlePostEdit = (id) => {
      const post = posts.find(o => o.id === id);
      if (post === undefined) {
        return;
      }

      setEdited(post);
    };

  > const handlePostSave = (post) => { ...
    };

    return (
      <>
        <PostForm edited={edited} onSave={handlePostSave} />
        <div>
          {posts.map(o => <Post
            key={o.id}
            post={o}
            onLike={handlePostLike}
            onRemove={handlePostRemove}
            onHide={handleTogglePostVisibility}
            onEdit={handlePostEdit}
            onShow={handleTogglePostVisibility} />)}
        </div>
      </>
    );
  }
}

export default Wall;

```



React

JS PostForm.js X

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

```
1  import React, { useState, useRef } from 'react';
2
3  > const empty = { ...
17 };
18
19 export default function PostForm({edited = empty, onSave}) {
20   const [post, setPost] = useState(edited);
21   const firstFocusEl = useRef(null);
22
23 >   const handleSubmit = (evt) => { ...
36   };
37
38 >   const handleChange = (evt) => { ...
53   };
54
55   return (
56     <form onSubmit={handleSubmit}>
57       <textarea
58 >       ref={firstFocusEl} ...
62       onChange={handleChange}/>
63       <input
64 >       name="tags" ...
67       onChange={handleChange}
68       />
69       <button>Ok</button>
70     </form>
71   )
72   };
```



React

Здесь нам потребуются некоторые знания JS. В `Wall`, если мы ничего не устанавливаем в качестве начального значения `useState`, значит значение будет `undefined`.

```
const [edited, setEdited] = useState();
```

В `props PostForm`, если значение `edited = undefined`, будет использоваться значение по умолчанию, которое выставлено как `empty`:

```
export default function PostForm({edited = empty, onSave}) {
```

Таким образом, мы реализуем логику:

- если в `edited` в `Wall` равно `undefined`, то используем `empty` (добавление)
- если в `edited` в `Wall` какой-то пост, то используем его (редактирование)

Конечно же, логику с `empty` можно сразу разместить в `Wall`, но это лишь один из вариантов.



React

Несмотря на то, что можно увидеть в React Dev Tools как `props` меняется, отрисовываться в форме редактируемый пост не будет.

Q: почему?

A: дело в том, что `useState` лишь один раз устанавливает начальное значение. Изменение `props` (в нашем случае) не ведёт к установке начального значения.

Поэтому мы можем использовать немного другую механику: вместо того, чтобы пытаться установить начальное значение, мы можем на изменение `props` устанавливать новое значение `state`.



useEffect

Хук `useEffect` используется для того, чтобы реализовывать различные побочные действия (в том числе установку `state`) в ответ на изменение зависимостей. Что подразумевается под зависимостью? Давайте смотреть:

```
export default function PostForm({edited = empty, onSave}) {  
  const [post, setPost] = useState(edited);  
  const firstFocusEl = useRef(null);  
  useEffect(() => {  
    |   setPost(edited);  
  }, [edited]);  
}
```

Каждый раз, когда `edited` будет меняться, мы будем менять `state`.



useEffect

А теперь попробуйте отредактировать самый первый пост, у которого `content = null`:

```
✖ Warning: `value` prop on `textarea` should not be null. Consider using an empty string to clear the component or `undefined` for uncontrolled components.  
    in textarea (at PostForm.js:60)  
    in form (at PostForm.js:59)  
    in PostForm (at Wall.js:85)  
    in Wall (at App.js:7)  
    in div (at App.js:6)  
    in App (at src/index.js:9)  
    in StrictMode (at src/index.js:8)
```

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



"Чистота данных"

Почему мы так много внимания обращаем на "неудобные" данные? Ведь можно было сразу договориться в `content` хранить не `null`, а пустую строку. В `tags` – пустой массив, а в `photo` – объект с пустыми значениями `alt` и `url`.

Всё зависит от того, кто и как разрабатывает серверную часть вашего приложения. Если вы сами или тот, с кем вы можете согласовать удобный для себя формат данных – тогда нет проблем, вы используете его (и большая часть обсуждаемых нами вопросов снимается).

Но не всегда это возможно, именно поэтому мы вам показываем эти проблемы, чтобы вы знали, как их решать.



Редактирование

Хорошо, редактирование работает, но достаточно странно. Если отредактировать пост, то он вместо обновления существующего, добавит новый.


Один из вариантов решения этой проблемы: прямо в [Wall](#) проверять, это обновление или создание нового:

```
const handlePostSave = (post) => {  
  if (edited !== undefined) {  
    setPosts((prevState) => prevState.map((o) => {  
      if (o.id !== post.id) {  
        return o;  
      }  
  
      return {...post};  
    })))  
    setEdited(undefined);  
    return;  
  }  
  setPosts((prevState) => [{...post}, ...prevState]);  
};
```



Редактирование

Но так тоже не работает. Почему? Вы можете посмотреть в отладчике (поставить инструкцию `debugger`) и увидеть, что поста с таким `id` просто нет, даже если мы редактируем. Почему? Потому, что в `PostForm` мы каждый раз при сохранении меняем `id`:



```
onSave({
  ...post,
  id: Date.now(),
  created: Date.now(),
  tags,
  photo: post.photo?.url ? {alt: '', ...post.photo} : null
});
setPost(empty);
firstFocusEl.current.focus();
```



Редактирование

Соответственно, мы можем просто поменять логику, чтобы `id` не назначался, если он уже назначен (помните, мы в `empty` выставили `0`? мы это сделали не случайно):

```
onSave({  
  ...post,  
  id: post.id || Date.now(),  
  created: post.created || Date.now(),  
  tags,  
  photo: post.photo?.url ? {alt: '', ...post.photo} : null  
});  
setPost(empty);  
firstFocusEl.current.focus();
```

Теперь всё работает, как нужно.



useEffect

Единственный вопрос, а нужно ли нам теперь в `handleSubmit` делать `setPost(empty)`? Ведь мы после редактирования делаем `setEdited(undefined)` в `Wall`?

На самом деле, нужно, потому что иначе после добавления не будет вычищаться форма (именно после добавления), либо можно вызывать `setEdited(undefined)` в `Wall` после добавления:

```
const handlePostSave = (post) => {  
  if (edited !== undefined) {  
    setPosts((prevState) => prevState.map((o) => {  
      if (o.id !== post.id) {  
        return o;  
      }  
  
      return {...post};  
    })))  
    setEdited(undefined);  
    return;  
  }  
  setPosts((prevState) => [...post, ...prevState]);  
  setEdited(undefined);  
};
```



Важно

Мы специально провели вас полностью через этап создания приложения и показали, как на самом деле происходит работа, когда вы делаете что-то впервые (никто не делает сразу идеально, если до этого уже раз 5 не делал того же самого).

Поэтому будьте готовы редактировать свой код, проверять его на различные сценарии и т.д.



ИТОГИ



Итоги

Сегодня мы рассмотрели вопросы добавления и редактирования на примере сложных объектов, поля которых могут быть `null` или `undefined`.



ДОМАШНЕЕ ЗАДАНИЕ



Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе).

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.

Если не успеете сдать в срок домашнее задание, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в [Телеграм канале](#).



Важный момент

Чтобы вам было интереснее, бот будет требовать, чтобы у автора так же был `id`.

Выставляйте его равным `1`.



ДЗ: Теги

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

hooks



```
▼ State: {author: {...}, content: "", created: 1604378779038, ...}
  id: 1604378779038
  ▶ author: {avatar: ...}
    content: ""
    photo: null
    hit: false
    likes: 0
    likedByMe: false
    hidden: false
  ▶ tags: ["#homework", "#deadline"]
    created: 1604378779038
    new entry: ""
  State: ""
```



ДЗ: Теги

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

hooks

```
▼ State: {author: {...}, content: "", created: 1604378779038, ...}  
  id: 1604378779038  
  ▶ author: {avatar: ...}  
    content: ""  
    photo: null  
    hit: false  
    likes: 0  
    likedByMe: false  
    hidden: false  
  ▶ tags: [""]  
    created: 1604378779038  
    new entry: ""
```



ДЗ: Теги

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

hooks



```
▼ State: {author: {...}, content: "", created: 1604378779038, ...}  
  id: 1604378779038  
  ▶ author: {avatar: ...}  
    content: ""  
    photo: null  
    hit: false  
    likes: 0  
    likedByMe: false  
    hidden: false  
  ▶ tags: ["#homework", "", "", "", "#deadline"]  
    created: 1604378779038  
    new entry: ""
```



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://alif-skills.pro/media/meme.jpg',  
  alt: 'Мем про дедлайн',  
},
```

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>  
)
```



ДЗ: Отмена

Несмотря на то, что кажется, что наше приложение работает, на самом деле оно не работает. Почему? Просто потому, что в нём нельзя отменить редактирование. Т.е. если мы нажали на каком-то посту "изменить", мы не можем "отменить" это (перезагрузка страницы не считается).

Что вам нужно сделать? Сделайте кнопку "Отменить" на форме, которая:

1. Появляется только тогда, когда пользователь редактирует существующий пост, а не создаёт новый
2. Отменяет редактирование (т.е. выставляет `state` в `empty`)
3. И не забудьте, что ещё стоит как-то `Wall` сообщить о том, что пользователь нажал на отмену (сделайте с помощью props `onCancel`)
4. **Важно:** устанавливайте обработчик именно на кнопку (`onClick`), а не на форму (`onReset`)



Спасибо за внимание

alif skills

2023г.

