нетология

РАБОТА С ФОРМАМИ

Дмитрий
Федин

Дмитрий Федин

ПЛАН ЗАНЯТИЯ

1. Формы
2. Контролируемые компоненты
3. Неконтролируемые компоненты
4. Lifting State Up
5. Lifting State Down

ФОРМЫ

ФОРМЫ

Мы уже научились обрабатывать события клика, но с формами всё немного сложнее, поскольку внутри элементов формы (полей ввода, списков выбора) хранится значение, введённое пользователем.

ОБРАТНАЯ СВЯЗЬ

Попробуем сделать формочку для обратной связи на сайт.

В форме будут:

- текстовое поле ввода
- чекбокс
- select
- textarea

Вынесем всё в отдельный компонент и назовём его `Feedback`.

Feedback

```
function Feedback(props) {
  return (<form>
    <div>
      <label htmlFor="name">Ваше имя</label>
      <input id="name" name="name" />
    </div>
    <div>
      <label htmlFor="satisfaction">Выберите уровень удовлетворённости</label>
      <select id="satisfaction" name="satisfaction">
        <option value="good">Хорошо</option>
        <option value="bad">Плохо</option>
      </select>
    </div>
    <div>
      <label htmlFor="agreement">
        <input id="agreement" name="agreement" type="checkbox">
        Согласен на передачу перс.данных
        </input>
      </label>
    </div>
    <button type="submit">Отправить</button>
  </form>)
}
```

`htmlFor`

Обратите внимание, вместо атрибута `for` (в HTML), в JSX используется `htmlFor`.

onSubmit

Повесим обработчик `onSubmit` и посмотрим на получившийся результат:

```
const handleSubmit = evt => {
  evt.preventDefault();
  console.log(evt.type);
  console.dir(evt.target);
}

return (
  <form onSubmit={handleSubmit}>
    ...
  </form>
);
```

`onSubmit`

Неплохо, мы перехватили событие, отменили поведение по умолчанию (не забывайте про `preventDefault`) и получили доступ к форме.

Но нам же нужно значение полей, а не сама форма!

Давайте посмотрим, что нам предлагает React.

КОНТРОЛИРУЕМЫЕ КОМПОНЕНТЫ

КОНТРОЛИРУЕМЫЕ КОМПОНЕНТЫ

Для большинства элементов форм React предлагает нам концепцию контролируемых компонентов.

В чём суть: элемент формы не сам отвечает за хранение своего состояния, а полагается на то, что у нас хранится в `state`.

Соответственно, любое изменение элемента формы (события `input` или `change`), ведут к изменению состояния, а изменение состояния ведёт к перерисовке элемента с нужным значением (которое и берётся из состояния).

Посмотрим на примере.

STATE

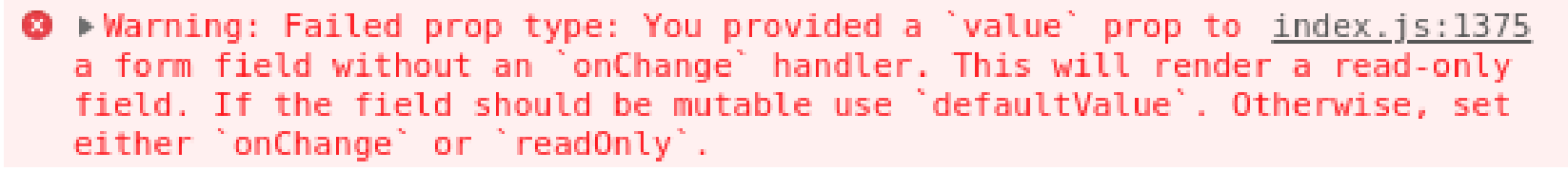
```
const [form, setForm] = useState({
  name: '',
  score: 'good',
  agreement: false
});

...

return (<form onSubmit={handleSubmit}>
<div>
  <label htmlFor="name">Ваше имя</label>
  <input id="name" name="name" value={form.name} />
</div>
...
</form>);
```

ONCHANGE

Но при таком коде значение в поле ввода не вводится, а React выводит нам подсказку:



Т.е. мы должны использовать либо обработчик `onChange` либо установить `defaultValue` (`defaultChecked` у checkbox'ов и radio).

ONCHANGE VS ONINPUT

Но почему именно `onChange`, а не `onInput`? Ведь из курса JS вы знаете, что `change` для текстовых полей срабатывает только при потере фокуса, а `input` при изменении.

Ответ достаточно простой - в React `onChange` для текстовых полей срабатывает и без потери фокуса. Поэтому везде далее мы будем использовать его.

STATE

```
const [form, setForm] = useState({
  name: '',
  score: 'good',
  agreement: false
});

const handleNameChange = evt => {
  setForm(prevForm => ({...prevForm, name: evt.target.value}));
}

return (<form onSubmit={handleSubmit}>
<div>
  <label htmlFor="name">Ваше имя</label>
  <input id="name" name="name" value={form.name} onChange={handleNameChange}
</div>
...
</form>);
```

BOILERPLATE CODE

Теперь всё работает, но не сложно догадаться, что для select'а придётся писать свой обработчик, для checkbox'а - свой.

И всё, чем они будут отличаться - именем свойства в `state`, в которое мы пишем.

Почему бы не воспользоваться возможностями JS и не использовать вычисляемые имена полей, чтобы написать одну функцию на всё?

```
const [form, setForm] = useState({
  name: '',
  score: 'good',
  agreement: false
});

... // handleSubmit опущен

const handleChange = ({target}) => {
  const name = target.name;
  const value = target.type === 'checkbox' ? target.checked : target.value;
  setForm(prevForm => {...prevForm, [name]: value});
}

return (<form onSubmit={handleSubmit}>
<div>
  <label htmlFor="name">Ваше имя</label>
  <input id="name" name="name" value={form.name} onChange={handleChange} />
</div>
...
</form>);
```

CHECKBOX'Ы

Вы, наверное, заметили, что мы проверяем тип `target`'а, и если он равен `checkbox`, используем свойство `checked`.

На это стоит принять во внимание.

ПОЛНЫЙ КОД `return`

```
return (<form onSubmit={handleSubmit}>
<div>
  <label htmlFor="name">Ваше имя</label>
  <input id="name" name="name" value={form.name} onChange={handleChange} />
</div>
<div>
  <label htmlFor="score">Выберите уровень удовлетворённости</label>
  <select id="score" name="score" value={form.score} onChange={handleChange}>
    <option value="good">Good</option>
    <option value="normal">Normal</option>
    <option value="bad">Bad</option>
  </select>
</div>
<div>
  <label htmlFor="agreement">
    <input id="agreement" name="agreement" type="checkbox"
      checked={form.agreement} onChange={handleChange} />
      Согласен с политикий обработки персональных данных
  </label>
</div>
<button type="submit">Отправить</button>
</form>);
```

STATE

При таком подходе всё работает. Обратите внимание: и для select'а и для checkbox'а (и для других элементов вроде radio, textarea и других).

Да и брать данные нам гораздо удобнее в `onSubmit`: все данные форм уже собраны в `form` (который из `useState`).

CLASS-BASED

Посмотрим на тот же самый код в Class-based реализации:

```
export class FeedbackClassBased extends Component {
  state = {
    name: '',
    score: 'good',
    agreement: false
  };
  ... // handleSubmit опущен
  handleChange = evt => {
    const name = evt.target.name;
    const value = evt.target.type === 'checkbox' ? evt.target.checked : evt.target
    this.setState({[name]: value});
  }
  render() {
    const form = this.state;
    return (<form onSubmit={this.handleSubmit}>
    <div>
      <label htmlFor="name">Ваше имя</label>
      <input id="name" name="name" value={form.name} onChange={this.handleChange}
    </div>
    ...
    </form>);
```

`setState`

Здесь стоит сделать важное замечание про `setState`: `setState` достаточно умный, чтобы уметь только обновлять "кусочек" состояния.

В примере мы передаём только одно поле из `state` - React обновит только его, при этом не затронув остальные.

Если вы используете Class-based компоненты с состоянием, именно такого подхода следует придерживаться.

ОГРАНИЧЕНИЯ КОНТРОЛИРУЕМЫХ КОМПОНЕНТОВ

К сожалению, не всегда мы можем использовать только контролируемые компоненты (т.к. не всегда контролируем значение `value`).

Есть ли у вас примеры, когда мы не можем контролировать значение `value`?

`input type="file"`

Совершенно верно, это `input type="file"`. Мы, конечно, можем писать туда что-то в `value` (чаще всего пустую строку, чтобы сбросить выбранное пользователем значение), но ничего хорошего из этого не выйдет - браузер не выберет то значение, что мы запишем.

Что же с этим делать?

НЕКОНТРОЛИРУЕМЫЕ КОМПОНЕНТЫ

НЕКОНТРОЛИРУЕМЫЕ КОМПОНЕНТЫ

В чём суть: элемент формы сам отвечает за хранение своего состояния, мы же можем в процессе обработки это его состояние получить.

Соответственно, мы можем подписаться на события `onSubmit` или `onChange` и в них смотреть, что пользователь выбрал.

Посмотрим на примере.

Хук `useRef`

```
function FileChooser(props) {
  const fileRef = useRef();

  const handleSubmit = evt => {
    evt.preventDefault();
    console.dir(fileRef.current.files);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="file" ref={fileRef} />
      <button>Ok</button>
    </form>
  )
}
```

ХУК `useRef`

Хук `useRef` позволяет хранить ссылку на определённый объект (он будет определён свойством `current`) и эта ссылка будет доступна в течение всей жизни компонента (о жизненном цикле компонента поговорим на следующих лекциях).

Т.е. мы можем создать эту ссылку, а затем установить её на определённый элемент с помощью атрибута `ref`.

Таким образом, получаем императивный инструмент доступа к определённому элементу.

Имея подобный доступ, мы легко сможем прочитать выбранные файлы с помощью `FileReader`.

CLASS-BASED

```
export class FileChooserClassBased extends Component {
  constructor(props) {
    super(props);
    this.fileRef = React.createRef();
  }

  handleSubmit = evt => {
    evt.preventDefault();
    console.dir(this.fileRef.current.files);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input type="file" ref={this.fileRef} />
        <button>Ok</button>
      </form>
    )
  }
}
```

`ref`

Не стоит очень часто использовать `ref`, т.к. чаще всего это приводит к императивному подходу (а мы стараемся следовать декларативному).

Но в некоторых случаях без него не обойтись, например:

1. Работа с `input type="file"`
2. Работа с `video/audio`, когда вы хотите программно управлять плеером
3. Работа с фокусом, позицией курсора и т.д.

СОВМЕЩЕНИЕ ПОДХОДОВ

А что, если наша форма содержит совместно и файлы, и поля ввода. Можем ли мы как-то не собирать её по кусочкам?

Да, можем, мы можем хранить объекты типа `File` или `FileList` в `state`.

СОВМЕЩЕНИЕ ПОДХОДОВ

```
function FileChooserAdv(props) {
  const [form, setForm] = useState({
    name: '',
    files: null
  });

  // handleSubmit опущен

  const handleChange = evt => {
    const {name, value} = evt.target;
    setForm(prevForm => ({...prevForm, [name]: value}));
  };
  const handleSelect = evt => {
    const {name, files} = evt.target;
    setForm(prevForm => ({...prevForm, [name]: files}));
  }

  return (<form onSubmit={handleSubmit}>
      <input name="name" type="text" onChange={handleChange} />
      <input name="files" type="file" onChange={handleSelect} />
      <button>Ok</button>
  </form>)
}
```

LIFTING STATE UP

ФИЛЬТР КНИГ

Попробуем решить следующую задачу, используя уже полученные нами знания.

Мы хотим сделать некий каталог книг, которые хотели бы прочитать, с возможностью добавлять туда книги, удалять, отмечать прочитанные и фильтровать по названиям.

МОДЕЛЬ

```
class BookModel {
  constructor(id, name, read = false) {
    this.id = id;
    this.name = name;
    this.read = read;
  }
}

export default BookModel;
```

README

```
function ReadMe(props) {
  const [books, setBooks] = useState([]);
  const [filter, setFilter] = useState('');
  const [form, setForm] = useState({name: ''});

  return (
    <input type="search" value={filter} onChange={handleFilter} />
    <ul>
      {books.map(o => <li key={o.id}>
        {o.name}
        <button>Done!</button>
        <button>Remove</button>
      </li>)}
    </ul>
    <form onSubmit={handleSubmit}>
      <input name="name" value={form.name} onChange={handleChange} />
    </form>
  )
}
```

Почему данный код работать не будет?

REACT.FRAGMENT

Надеюсь, вы помните, что в JSX корневой элемент должен быть только один.

Но что, если мы не хотим заворачивать нашу разметку в очередной `div`?

Для этих целей есть `React.Fragment` - готовый компонент, который позволяет предоставить нашим элементам единственного родителя и при этом не будет создавать дополнительных элементов при рендеринге в DOM.

README

```
function ReadMe(props) {
  const [books, setBooks] = useState([]);
  const [filter, setFilter] = useState('');
  const [form, setForm] = useState({name: ''});

  return (<React.Fragment>
    <input type="search" value={filter} onChange={handleFilter} />
    <ul>
      {books.map(o => <li key={o.id}>
        {o.name}
        <button>Done!</button>
        <button>Remove</button>
      </li>)}
    </ul>
    <form onSubmit={handleSubmit}>
      <input name="name" value={form.name} onChange={handleChange} />
    </form>
  </React.Fragment>)
}
```

ДОБАВЛЕНИЕ

Напишем обработчик для добавления.

Для этого нам каким-то образом надо генерировать наши id, для этого воспользуемся модулем `nanoid`:

```
npm install nanoid
```

```
yarn add nanoid
```

После чего добавить `import nanoid from 'nanoid';`

ДОБАВЛЕНИЕ

```
function ReadMe(props) {
  ...
  const handleFilter = evt => {
    // TODO: заглушка для фильтрации
  }

  const handleChange = evt => {
    const {name, value} = evt.target;
    setForm(prevForm => ({...prevForm, [name]: value}));
  }

  const handleSubmit = evt => {
    evt.preventDefault();
    const book = new BookModel(nanoid(), form.name);
    setBooks(prevBooks => [...prevBooks, book]);
  }
  ...
}
```

ДОБАВЛЕНИЕ

Добавление работает, но поле ввода не очищается.

Изменим `state` в `handleSubmit`:

```
function ReadMe(props) {
  ...
  const handleSubmit = evt => {
    evt.preventDefault();
    const book = new BookModel(nanoid(), form.name);
    setBooks(prevBooks => [...prevBooks, book]);
    // начальное значение формы можно вынести в константу
    setForm({name: ''});
  }
  ...
}
```

УДАЛЕНИЕ

Для удаления нам понадобится либо сам элемент, либо его id:

```
function ReadMe(props) {
  ...
  const handleRemove = id => {
    setBooks(prevBooks => prevBooks.filter(o => o.id !== id));
  }
  ...

  return (
    ...
    <ul>
      {books.map(o => <li key={o.id}>
        {o.name}
        <button onClick={() => handleDone(o.id)}>Done!</button>
        <button onClick={() => handleRemove(o.id)}>Remove</button>
      </li>)}
    </ul>
    ...
  );
}
```

ИЗМЕНЕНИЕ

Для изменения нам так же понадобится либо сам элемент, либо его id:

```
function ReadMe(props) {
  ...
  const handleDone = id => {
    setBooks(prevBooks => prevBooks.map(
      o => o.id === id ? new BookModel(o.id, o.name, !o.read) : o
    ));
  }
  ...
  return (
    ...
    <ul>
      {books.map(o => <li key={o.id}>
        {o.read && '✓'} {o.name}
        <button onClick={() => handleDone(o.id)}>Done!</button>
        <button onClick={() => handleRemove(o.id)}>Remove</button>
      </li>)}
    </ul>
    ...
  );
}
```

ФИЛЬТРАЦИЯ

```
function ReadMe(props) {
  ...
  const handleFilter = evt => {
    const {value} = evt.target;
    setFilter(value);
  }
  ...

  return (
    ...
    <ul>
      {books
        .filter(o => o.name.toLowerCase().includes(filter.trim().toLowerCase()))
        .map(o => <li key={o.id}>
          {o.read && '✓'} {o.name}
          <button onClick={() => handleDone(o.id)}>Done!</button>
          <button onClick={() => handleRemove(o.id)}>Remove</button>
        </li>)}
    </ul>
    ...
  );
}
```

РАЗБУХАНИЕ КОМПОНЕНТА

Не кажется, ли вам, что наш компонент делает слишком много?

Почему бы не разбить его на несколько компонентов, каждый из которых выполняет собственную задачу. Начнём с фильтра:

```
function Filter(props) {
  const [filter, setFilter] = useState('');

  const handleFilter = evt => {
    const {value} = evt.target;
    setFilter(value);
  }

  return (
    <input type="search" value={filter} onChange={handleFilter} />
  );
}
```

Но теперь вопрос: а кто является владельцем состояния фильтра? Сам ли фильтр или его родительский компонент, который затем на изменение этого состояния будет реагировать и фильтровать список?

LIFTING STATE UP

Lifting State Up - термин, используемый для описания ситуации, при которой состояние компонента выносится в ближайшего общего предка.

Перенесём состояние (условно) из компонента `Filter` (сделав его stateless) в родительский компонент.

```
function Filter(props) {
  const handleFilter = evt => {
    const {value} = evt.target;
    props.onFilter(value);
  }

  return (
    <input type="search" value={props.filter} onChange={handleFilter} />
  );
}
Filter.propTypes = {
  filter: PropTypes.string.isRequired,
  onFilter: PropTypes.func.isRequired,
}
```

LIFTING STATE UP

```
function ReadMe(props) {
  ...
  const [filter, setFilter] = useState('');
  ...

  const handleFilter = value => {
    setFilter(value);
  }
  ...

  return (<React.Fragment>
    <Filter onFilter={handleFilter} filter={filter} />
    ...
  </React.Fragment>);
}
```

LIFTING STATE UP: СПИСОК

```
function ReadMe(props) {
  ...
  const [books, setBooks] = useState([]);
  const handleRemove = id => {
    setBooks(books.filter(o => o.id !== id));
  }
  const handleDone = id => {
    setBooks(prevBooks => prevBooks.map(o => {
      if (o.id === id) {
        return new BookModel(o.id, o.name, !o.read)
      }
      return o;
    }));
  }
  ...
  const filtered = books.filter(
    o => o.name.toLowerCase().includes(filter.trim().toLowerCase())
  );
  return (<React.Fragment>
    <Filter onFilter={handleFilter} filter={filter} />
    <BookList books={filtered} onRemove={handleRemove} onDone={handleDone} />
    ...
  </React.Fragment>);
}
```

LIFTING STATE UP: BOOKLIST

```
function BookList (props) {
  const {books, onRemove: handleRemove, onDone: handleDone} = props;
  return (
    <ul>
      {books.map(o => <BookItem key={o.id} book={o}
            onRemove={handleRemove} onDone={handleDone} />)}
    </ul>
  )
}

BookList.propTypes = {
  books: PropTypes.arrayOf(PropTypes.instanceOf(BookModel)).isRequired,
  onRemove: PropTypes.func.isRequired,
  onDone: PropTypes.func.isRequired,
}
```

LIFTING STATE UP: BOOKITEM

```
function BookItem(props) {
  const {book, onRemove: handleRemove, onDone: handleDone} = props;
  return (
    <li key={book.id}>
      {book.read && '✓'} {book.name}
      <button onClick={() => handleDone(book.id)}>Done!</button>
      <button onClick={() => handleRemove(book.id)}>Remove</button>
    </li>
  )
}

BookItem.propTypes = {
  book: PropTypes.instanceOf(BookModel).isRequired,
  onRemove: PropTypes.func.isRequired,
  onDone: PropTypes.func.isRequired,
}
```

КОМПОЗИЦИЯ

С одной стороны всё хорошо - мы декомпозировали сложный компонент на несколько простых (более подробно о композиции - на следующей лекции).

Но с другой стороны, чем глубже будет становиться уровень вложенности компонентов, тем глубже нам придётся "пробрасывать" `props`.

Через какое-то время это начнём приводить к проблемам, особенно, если мы хотим объявить какое-то глобальное состояние, которое должно быть доступно многим компонентам (например, статус аутентификации пользователя - залогинен или нет).

Такие состояния придётся выносить на самый верх (до компонента App) и пробрасывать вниз до компонента отображения кнопок login/logout.

Как решать эту проблему, мы поговорим при изучении Context API и Redux.

ФОРМА ДОБАВЛЕНИЯ

Остался вопрос, связанный с формой добавления.

Но здесь есть нюанс - если мы внимательно посмотрим на саму форму добавления, то никого (имеется в виду компоненты) не интересует текущее состояние формы, их интересует только момент добавления (обработки `onSubmit`).

LIFTING STATE DOWN

LIFTING STATE DOWN

Lifting State Down - термин, используемый для описания ситуации, при которой состояние из родительского компонента переносится в дочерний и хранится там локально.

Происходит это потому, что отсутствуют общие данные, связанные с этим состоянием.

В нашем случае - мы перенесём состояние из компонента `ReadMe` в компонент `BookAddForm`.

BOOKADDFORM

```
function BookAddForm(props) {
  const {onAdd} = props;
  const [form, setForm] = useState({name: ''});
  const handleChange = evt => {
    const {name, value} = evt.target;
    setForm(prevForm => ({...prevForm, [name]: value}));
  }
  const handleSubmit = evt => {
    evt.preventDefault();
    const book = new BookModel(nanoid(), form.name);
    onAdd(book);
    setForm({name: ''});
  }
  return (<form onSubmit={handleSubmit}>
    <input name="name" value={form.name} onChange={handleChange} />
  </form>);
}

BookAddForm.propTypes = {
  onAdd: PropTypes.func.isRequired,
}
```

README (ИТОГОВАЯ ВЕРСИЯ)

```
function ReadMe(props) {
  const [books, setBooks] = useState([]);
  const [filter, setFilter] = useState('');
  const handleFilter = value => { setFilter(value); }
  const handleAdd = book => { setBooks(prevBooks => [...prevBooks, book]); }
  const handleRemove = id => {
    setBooks(prevBooks => prevBooks.filter(o => o.id !== id));
  }
  const handleDone = id => {
    setBooks(prevBooks => prevBooks.map(o => {
      if (o.id === id) { return new BookModel(o.id, o.name, !o.read) }
      return o;
    }));
  }
  const filtered = books.filter(
    o => o.name.toLowerCase().includes(filter.trim().toLowerCase())
  );
  return (<React.Fragment>
    <Filter onFilter={handleFilter} filter={filter} />
    <BookList books={filtered} onRemove={handleRemove} onDone={handleDone} />
    <BookAddForm onAdd={handleAdd} />
  </React.Fragment>);
}
```

CSS

CSS

В рамках лекций мы не уделяем достаточного времени CSS, поскольку вы его уже проходили. Но пару простейших вариантов использования CSS в React обозначить должны.

1. Вариант 1. Все стили в `src/index.css` - обычный глобальный файл стилей, следуете любой конвенции, которая вам нравится
2. Вариант 2. Отдельный css-файл для каждого компонента.

С первым вариантом всё понятно, давайте детальнее рассмотрим второй.

CSS

При этом подходе рекомендуется каждый компонент помещать в отдельный каталог, а внутри каталога уже хранить сам компонент и css-файл к нему.

Общую идею вы можете посмотреть на примере компонента `App` - создаётся три файла:

1. App.js - файл компонента
2. App.test.js - файл с тестами для компонента
3. App.css - файл с css-стилями

Поскольку мы работаем в Webpack с уже настроенными плагинами, достаточно импортировать файлы стилей в компоненте:

```
import './App.css';
```

При этом имена классов принято предварять именем компонента, например `App-logo`.

И ключевое: не забывать для корневого элемента в компоненте задать `className="App"`

STYLED COMPONENTS

Есть ещё целая библиотека Styled Components, которая позволяет создавать стилизованные компоненты, но её изучение находится за рамками нашего курса.

ИТОГИ

ИТОГИ

- используйте по возможности контролируемые компоненты
- используйте неконтролируемые компоненты при работе с файлами/медиа/фокусом и т.д.
- декомпозируйте сложные компоненты на простые (желательно stateless).

нетология

Задавайте вопросы и пишите отзыв о лекции!

Дмитрий Федин