

# РАБОТА С ФОРМАМИ



Дмитрий  
Бронских



# ДМИТРИЙ БРОНСКИХ

FullStack-developer в ООО "Телеком-Нейва"



[Дмитрий Бронских](#)



# 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, введенное 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

```
1 function Feedback(props) {
2   return (<form>
3     <div>
4       <label htmlFor="name">Ваше имя</label>
5       <input id="name" name="name" />
6     </div>
7     <div>
8       <label htmlFor="satisfaction">Выберите уровень удовлетворённости</label>
9       <select id="satisfaction" name="satisfaction">
10        <option value="good">Хорошо</option>
11        <option value="bad">Плохо</option>
12      </select>
13    </div>
14    <div>
15      <label htmlFor="agreement">
16        <input id="agreement" name="agreement" type="checkbox">
17          Согласен на передачу перс.данных
18        </input>
19      </label>
20    </div>
21    <button type="submit">Отправить</button>
22  </form>)
23 }
```

# htmlFor

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



# onSubmit

Повесим обработчик `onSubmit` и посмотрим на получившийся результат:

```
1  const handleSubmit = evt => {  
2    evt.preventDefault();  
3    console.log(evt.type);  
4    console.dir(evt.target);  
5  }  
6  
7  return (  
8    <form onSubmit={handleSubmit}>  
9      ...  
10     </form>  
11  );
```

## 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

```
1  const [form, setForm] = useState({
2    name: '',
3    score: 'good',
4    agreement: false
5  });
6
7  ...
8
9  return (<form onSubmit={handleSubmit}>
10    <div>
11      <label htmlFor="name">Ваше имя</label>
12      <input id="name" name="name" value={form.name} />
13    </div>
14    ...
15    </form>);
```

# ONCHANGE

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

```
✖ Warning: Failed prop type: You provided a `value` prop to index.js:1375 a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.
```

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

```
1  const [form, setForm] = useState({
2    name: '',
3    score: 'good',
4    agreement: false
5  });
6
7  const handleNameChange = evt => {
8    setForm(prevForm => ({...prevForm, name: evt.target.value}));
9  }
10
11  return (<form onSubmit={handleSubmit}>
12    <div>
13      <label htmlFor="name">Ваше имя</label>
14      <input id="name" name="name" value={form.name} onChange={handleNameChange}
15    </div>
16    ...
17  </form>);
```



## BOILERPLATE CODE

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

```
1  const [form, setForm] = useState({
2    name: '',
3    score: 'good',
4    agreement: false
5  });
6
7  ... // handleSubmit опущен
8
9  const handleChange = ({target}) => {
10    const name = target.name;
11    const value = target.type === 'checkbox' ? target.checked : target.value;
12    setForm(prevForm => {...prevForm, [name]: value});
13  }
14
15  return (<form onSubmit={handleSubmit}>
16    <div>
17      <label htmlFor="name">Ваше имя</label>
18      <input id="name" name="name" value={form.name} onChange={handleChange} />
19    </div>
20    ...
21  </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`

```
1  return (<form onSubmit={handleSubmit}>
2    <div>
3      <label htmlFor="name">Ваше имя</label>
4      <input id="name" name="name" value={form.name} onChange={handleChange} />
5    </div>
6    <div>
7      <label htmlFor="score">Выберите уровень удовлетворённости</label>
8      <select id="score" name="score" value={form.score} onChange={handleChange}>
9        <option value="good">Good</option>
10       <option value="normal">Normal</option>
11       <option value="bad">Bad</option>
12     </select>
13   </div>
14   <div>
15     <label htmlFor="agreement">
16       <input id="agreement" name="agreement" type="checkbox"
17         checked={form.agreement} onChange={handleChange} />
18       Согласен с политикой обработки персональных данных
19     </label>
20   </div>
21   <button type="submit">Отправить</button>
22 </form>);
```

# STATE

При таком подходе всё работает. Обратите внимание: и для select'a и для checkbox'a (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:

```
1  export class FeedbackClassBased extends Component {
2    state = {
3      name: '',
4      score: 'good',
5      agreement: false
6    };
7    ... // handleSubmit опущен
8    handleChange = evt => {
9      const name = evt.target.name;
10     const value = evt.target.type === 'checkbox' ? evt.target.checked : evt.target
11     this.setState({[name]: value});
12   }
13   render() {
14     const form = this.state;
15     return (<form onSubmit={this.handleSubmit}>
16       <div>
17         <label htmlFor="name">Ваше имя</label>
18         <input id="name" name="name" value={form.name} onChange={this.handleChange}
19       </div>
20       ...
21     </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.

# XYK useRef

```
1  function FileChooser(props) {  
2    const fileRef = useRef();  
3  
4    const handleSubmit = evt => {  
5      evt.preventDefault();  
6      console.dir(fileRef.current.files);  
7    }  
8  
9    return (  
10     <form onSubmit={handleSubmit}>  
11       <input type="file" ref={fileRef} />  
12       <button>Ok</button>  
13     </form>  
14   )  
15 }
```

## 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

```
1  export class FileChooserClassBased extends Component {
2      constructor(props) {
3          super(props);
4          this.fileRef = React.createRef();
5      }
6
7      handleSubmit = evt => {
8          evt.preventDefault();
9          console.dir(this.fileRef.current.files);
10     }
11
12     render() {
13         return (
14             <form onSubmit={this.handleSubmit}>
15                 <input type="file" ref={this.fileRef} />
16                 <button>Ok</button>
17             </form>
18         )
19     }
20 }
```

## 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

```
1 function FileChooserAdv(props) {
2   const [form, setForm] = useState({
3     name: '',
4     files: null
5   });
6
7   // handleSubmit опущен
8
9   const handleChange = evt => {
10     const {name, value} = evt.target;
11     setForm(prevForm => ({...prevForm, [name]: value}));
12   };
13   const handleSelect = evt => {
14     const {name, files} = evt.target;
15     setForm(prevForm => ({...prevForm, [name]: files}));
16   }
17
18   return (<form onSubmit={handleSubmit}>
19     <input name="name" type="text" onChange={handleChange} />
20     <input name="files" type="file" onChange={handleSelect} />
21     <button>Ok</button>
22   </form>)
23 }
```



# 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

```
1 class BookModel {  
2   constructor(id, name, read = false) {  
3     this.id = id;  
4     this.name = name;  
5     this.read = read;  
6   }  
7 }  
8  
9 export default BookModel;
```

# README

```
1 function ReadMe(props) {
2   const [books, setBooks] = useState([]);
3   const [filter, setFilter] = useState('');
4   const [form, setForm] = useState({name: ''});
5
6   return (
7     <input type="search" value={filter} onChange={handleFilter} />
8     <ul>
9       {books.map(o => <li key={o.id}>
10         {o.name}
11         <button>Done!</button>
12         <button>Remove</button>
13       </li>)}
14     </ul>
15     <form onSubmit={handleSubmit}>
16       <input name="name" value={form.name} onChange={handleChange} />
17     </form>
18   )
19 }
```

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

```
1  function ReadMe(props) {
2    const [books, setBooks] = useState([]);
3    const [filter, setFilter] = useState('');
4    const [form, setForm] = useState({name: ''});
5
6    return (<React.Fragment>
7      <input type="search" value={filter} onChange={handleFilter} />
8      <ul>
9        {books.map(o => <li key={o.id}>
10          {o.name}
11          <button>Done!</button>
12          <button>Remove</button>
13          </li>)}
14      </ul>
15      <form onSubmit={handleSubmit}>
16        <input name="name" value={form.name} onChange={handleChange} />
17      </form>
18    </React.Fragment>)
19  }
```

# 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

```
1  function ReadMe(props) {
2    ...
3    const handleFilter = evt => {
4      // TODO: заглушка для фильтрации
5    }
6
7    const handleChange = evt => {
8      const {name, value} = evt.target;
9      setForm(prevForm => ({...prevForm, [name]: value}));
10   }
11
12   const handleSubmit = evt => {
13     evt.preventDefault();
14     const book = new BookModel(nanoid(), form.name);
15     setBooks(prevBooks => [...prevBooks, book]);
16   }
17   ...
18 }
```

# ДОБАВЛЕНИЕ

Добавление работает, но поле ввода не очищается.

Изменим `state` в `handleSubmit`:

```
1  function ReadMe(props) {  
2    ...  
3    const handleSubmit = evt => {  
4      evt.preventDefault();  
5      const book = new BookModel(nanoid(), form.name);  
6      setBooks(prevBooks => [...prevBooks, book]);  
7      // начальное значение формы можно вынести в константу  
8      setForm({name: ''});  
9    }  
10   ...  
11 }
```

# УДАЛЕНИЕ

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

```
1  function ReadMe(props) {
2    ...
3    const handleRemove = id => {
4      setBooks(prevBooks => prevBooks.filter(o => o.id !== id));
5    }
6    ...
7
8    return (
9      ...
10     <ul>
11       {books.map(o => <li key={o.id}>
12         {o.name}
13         <button onClick={() => handleDone(o.id)}>Done!</button>
14         <button onClick={() => handleRemove(o.id)}>Remove</button>
15       </li>)}
16     </ul>
17     ...
18   );
19 }
```

# ИЗМЕНЕНИЕ

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

```
1  function ReadMe(props) {
2    ...
3    const handleDone = id => {
4      setBooks(prevBooks => prevBooks.map(
5        o => o.id === id ? new BookModel(o.id, o.name, !o.read) : o
6      ));
7    }
8    ...
9    return (
10     ...
11     <ul>
12       {books.map(o => <li key={o.id}>
13         {o.read && '✓'} {o.name}
14         <button onClick={() => handleDone(o.id)}>Done!</button>
15         <button onClick={() => handleRemove(o.id)}>Remove</button>
16       </li>)}
17     </ul>
18     ...
19   );
20 }
```

# ФИЛЬТРАЦИЯ

```
1  function ReadMe(props) {
2    ...
3    const handleFilter = evt => {
4      const {value} = evt.target;
5      setFilter(value);
6    }
7    ...
8
9    return (
10     ...
11     <ul>
12       {books
13         .filter(o => o.name.toLowerCase().includes(filter.trim().toLowerCase()))
14         .map(o => <li key={o.id}>
15           {o.read && '✓'} {o.name}
16           <button onClick={() => handleDone(o.id)}>Done!</button>
17           <button onClick={() => handleRemove(o.id)}>Remove</button>
18         </li>)}
19     </ul>
20     ...
21   );
22 }
```

# 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:

```
1  function Filter(props) {  
2    const [filter, setFilter] = useState('');  
3  
4    const handleFilter = evt => {  
5      const {value} = evt.target;  
6      setFilter(value);  
7    }  
8  
9    return (  
10     <input type="search" value={filter} onChange={handleFilter} />  
11   );  
12 }
```

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.

```
1  function Filter(props) {
2    const handleFilter = evt => {
3      const {value} = evt.target;
4      props.onFilter(value);
5    }
6
7    return (
8      <input type="search" value={props.filter} onChange={handleFilter} />
9    );
10 }
11 Filter.propTypes = {
12   filter: PropTypes.string.isRequired,
13   onFilter: PropTypes.func.isRequired,
14 }
```

# LIFTING STATE UP

```
1  function ReadMe(props) {  
2    ...  
3    const [filter, setFilter] = useState('');  
4    ...  
5  
6    const handleFilter = value => {  
7      setFilter(value);  
8    }  
9    ...  
10  
11   return (<React.Fragment>  
12     <Filter onFilter={handleFilter} filter={filter} />  
13     ...  
14   </React.Fragment>);  
15 }
```



# LIFTING STATE UP: СПИСОК

```
1 function ReadMe(props) {
2   ...
3   const [books, setBooks] = useState([]);
4   const handleRemove = id => {
5     setBooks(books.filter(o => o.id !== id));
6   }
7   const handleDone = id => {
8     setBooks(prevBooks => prevBooks.map(o => {
9       if (o.id === id) {
10         return new BookModel(o.id, o.name, !o.read)
11       }
12       return o;
13     }));
14   }
15   ...
16   const filtered = books.filter(
17     o => o.name.toLowerCase().includes(filter.trim().toLowerCase())
18   );
19   return (<React.Fragment>
20     <Filter onFilter={handleFilter} filter={filter} />
21     <BookList books={filtered} onRemove={handleRemove} onDone={handleDone} />
22     ...
23   </React.Fragment>);
24 }
```

# LIFTING STATE UP: BOOKLIST

```
1 function BookList (props) {  
2   const {books, onRemove: handleRemove, onDone: handleDone} = props;  
3   return (  
4     <ul>  
5       {books.map(o => <BookItem key={o.id} book={o}  
6         onRemove={handleRemove} onDone={handleDone} />)}  
7     </ul>  
8   )  
9 }  
10  
11 BookList.propTypes = {  
12   books: PropTypes.arrayOf(PropTypes.instanceOf(BookModel)).isRequired,  
13   onRemove: PropTypes.func.isRequired,  
14   onDone: PropTypes.func.isRequired,  
15 }
```

# LIFTING STATE UP: BOOKITEM

```
1 function BookItem(props) {
2   const {book, onRemove: handleRemove, onDone: handleDone} = props;
3   return (
4     <li key={book.id}>
5       {book.read && '✓'} {book.name}
6       <button onClick={() => handleDone(book.id)}>Done!</button>
7       <button onClick={() => handleRemove(book.id)}>Remove</button>
8     </li>
9   )
10 }
11
12 BookItem.propTypes = {
13   book: PropTypes.instanceOf(BookModel).isRequired,
14   onRemove: PropTypes.func.isRequired,
15   onDone: PropTypes.func.isRequired,
16 }
```

# КОМПОЗИЦИЯ

С одной стороны всё хорошо - мы декомпозовали 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 начнёт 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

```
1  function BookAddForm(props) {
2    const {onAdd} = props;
3    const [form, setForm] = useState({name: ''});
4    const handleChange = evt => {
5      const {name, value} = evt.target;
6      setForm(prevForm => ({...prevForm, [name]: value}));
7    }
8    const handleSubmit = evt => {
9      evt.preventDefault();
10     const book = new BookModel(nanoid(), form.name);
11     onAdd(book);
12     setForm({name: ''});
13   }
14   return (<form onSubmit={handleSubmit}>
15     <input name="name" value={form.name} onChange={handleChange} />
16   </form>);
17 }
18
19 BookAddForm.propTypes = {
20   onAdd: PropTypes.func.isRequired,
21 }
```



# README (ИТОГОВАЯ ВЕРСИЯ)

```
1 function ReadMe(props) {
2   const [books, setBooks] = useState([]);
3   const [filter, setFilter] = useState('');
4   const handleFilter = value => { setFilter(value); }
5   const handleAdd = book => { setBooks(prevBooks => [...prevBooks, book]); }
6   const handleRemove = id => {
7     setBooks(prevBooks => prevBooks.filter(o => o.id !== id));
8   }
9   const handleDone = id => {
10    setBooks(prevBooks => prevBooks.map(o => {
11      if (o.id === id) { return new BookModel(o.id, o.name, !o.read) }
12      return o;
13    }));
14  }
15  const filtered = books.filter(
16    o => o.name.toLowerCase().includes(filter.trim().toLowerCase())
17  );
18  return (<React.Fragment>
19    <Filter onFilter={handleFilter} filter={filter} />
20    <BookList books={filtered} onRemove={handleRemove} onDone={handleDone} />
21    <BookAddForm onAdd={handleAdd} />
22  </React.Fragment>);
23 }
```



**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).





**Спасибо за внимание! Время задавать вопросы**

**ДМИТРИЙ БРОНСКИХ**

 Дмитрий Бронских