РАСШИРЕННЫЕ ВОПРОСЫ

ЖИЗНЕННЫЙ ЦИКЛ КОМПОНЕНТОВ

FORWARD REF

Компоненты, использующие FancyButton, могут получить ref на кнопку DOM, так как если бы они получали доступ напрямую.

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// Теперь вы можете получить прямую ссылку
// на кнопку в DOM:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

Упаковываем компонент в forwardRef(**component**) – это дает возможность обращаться к DOM-элементам внутри component **снаружи.**

*Во внешнем компоненте есть **ref**: ссылка на DOM элемент, вложенный в дочерний компонент: мы получаем доступ "через голову" **FancyButton** - к "внуку" **OuterComponent***

CHILDREN

- Бывают ситуации, когда необходимо, чтобы компоненты обеспечивали обертывание предоставленных компонентов, а не генерировали их из свойств.
- Дочерние подкомпоненты размещаются между открывающими и закрывающими тегами компонента, как обычные HTML-элементы. Внешний компонент может получать доступ к этим дочерним подкопонентам через this.props.children.

```
let Likes =
  <div>
    <LikesListWrapper>
      <LikeListItem text="turtles"/>
    </LikesListWrapper>
  </div>;

let LikesListWrapper =
  <ul>{ this.props.children }</ul>;
```

CLASSNAME

- Поскольку класс является зарезервированным ключевым словом в JavaScript, для установки класса элемента необходимо использовать имя свойства className.

```
render() {
   return (
     <a className="btn btn-default">I am a button with a class!</a>
  );
}
```

КЛОНИРОВАНИЕ ЭЛЕМЕНТА С ПЕРЕОПРЕДЕЛЕНИЕМ PROPS

- React.cloneElement(
- element,
- [props],
- [...children]
-)

Клонирует и возвращает новый элемент React, используя **element как стартовую точку** (клонируется все дерево).

Результирующий элемент будет иметь оригинальные свойства, **смерженные** с переданными свойствами.

Новые дочерние элементы **заменят** оригинальные.

key и **ref** из оригинального элемента **сохранятся**.

Является аналогом такого вызова:

```
<element
   {...element.props}
   {...props}>
   {children}
</element>
```

ЖИЗНЕННЫЕ ЦИКЛЫ

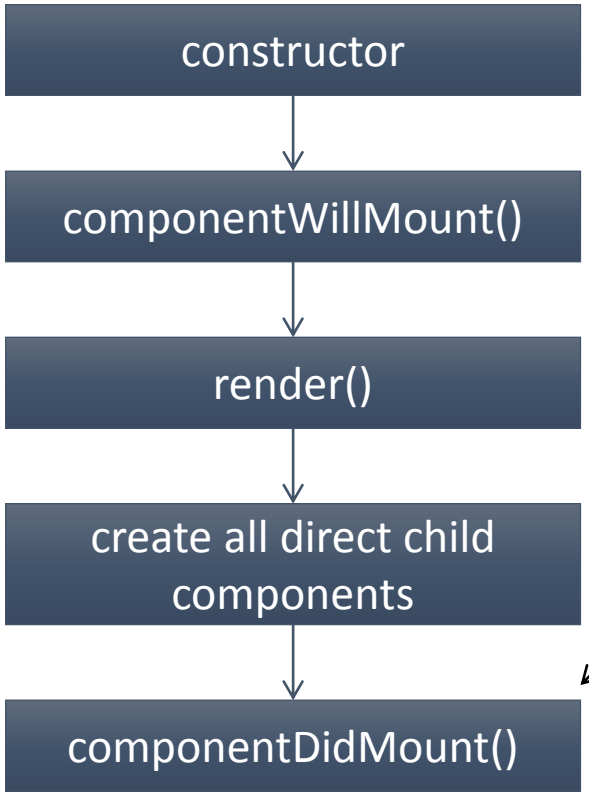
- Метод render() отображения в компоненте React должен быть чистой функцией. Это должен быть метод без сохранения состояния, который не делает запросов Ajax и т. д. Он должен просто получать состояние и свойства, а затем отображать UI:
  render = (state,props)=>JSX

- *Но куда же поместить взаимодействие с окружением?*

- Для ответа на этот вопрос рассмотрим методы
  - *жизненного цикла React*

ЖИЗНЕННЫЕ ЦИКЛЫ

создание нового компонента

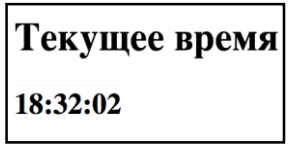


устарела *с React 16.3 – не рекомендуется использовать*

для сайд-эффектов (вызовы AJAX и т.д.)
будет вызвана лишь раз во всем жизненном цикле данного компонента и будет сигнализировать, что компонент и все его дочерние компоненты отрисовались без ошибок.
Не вызывайте setState – вызовет зацикливание!

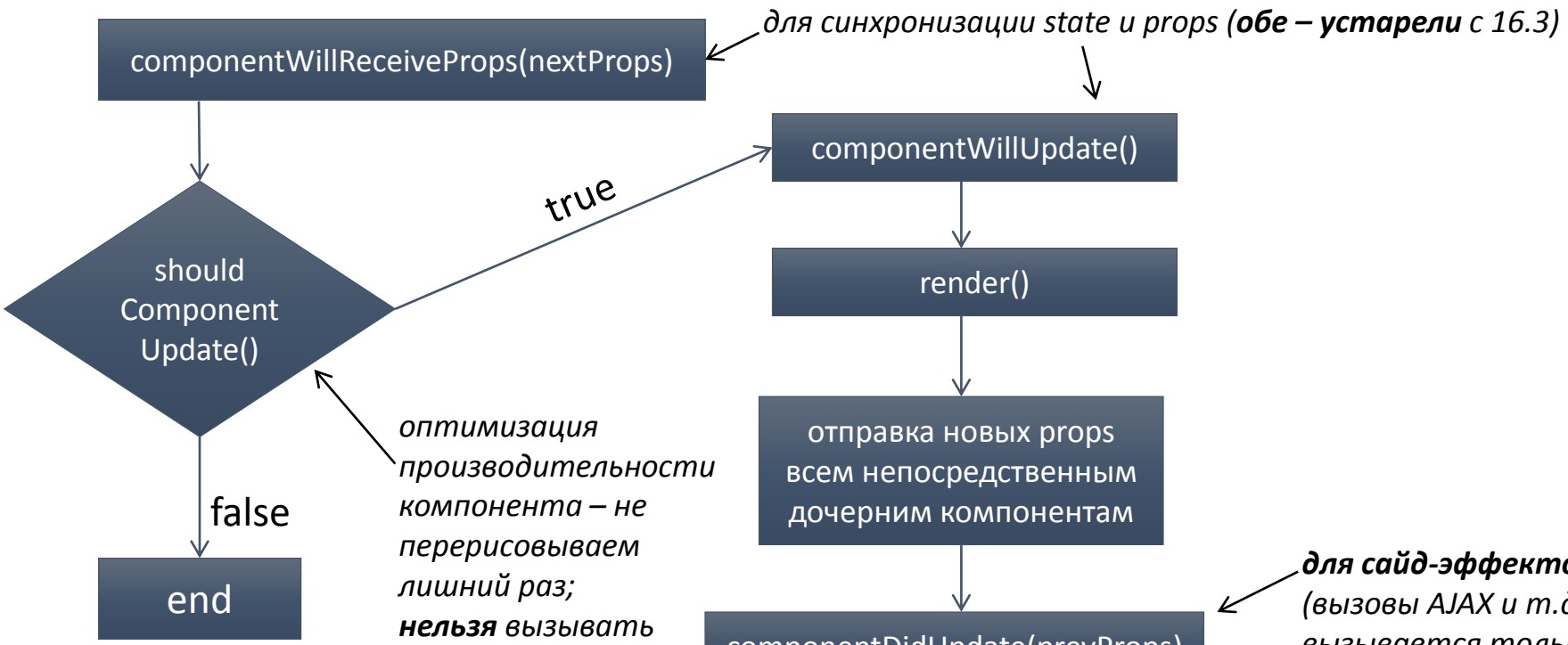
ЖИЗНЕННЫЙ ЦИКЛ: ПРИМЕР ИСПОЛЬЗОВАНИЯ

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }
  render() {
    return (
      <div>
        <h1>Текущее время</h1>
        <h2>
          {this.state.date
            .toLocaleTimeString()}
        </h2>
      </div>
    );
  }
```



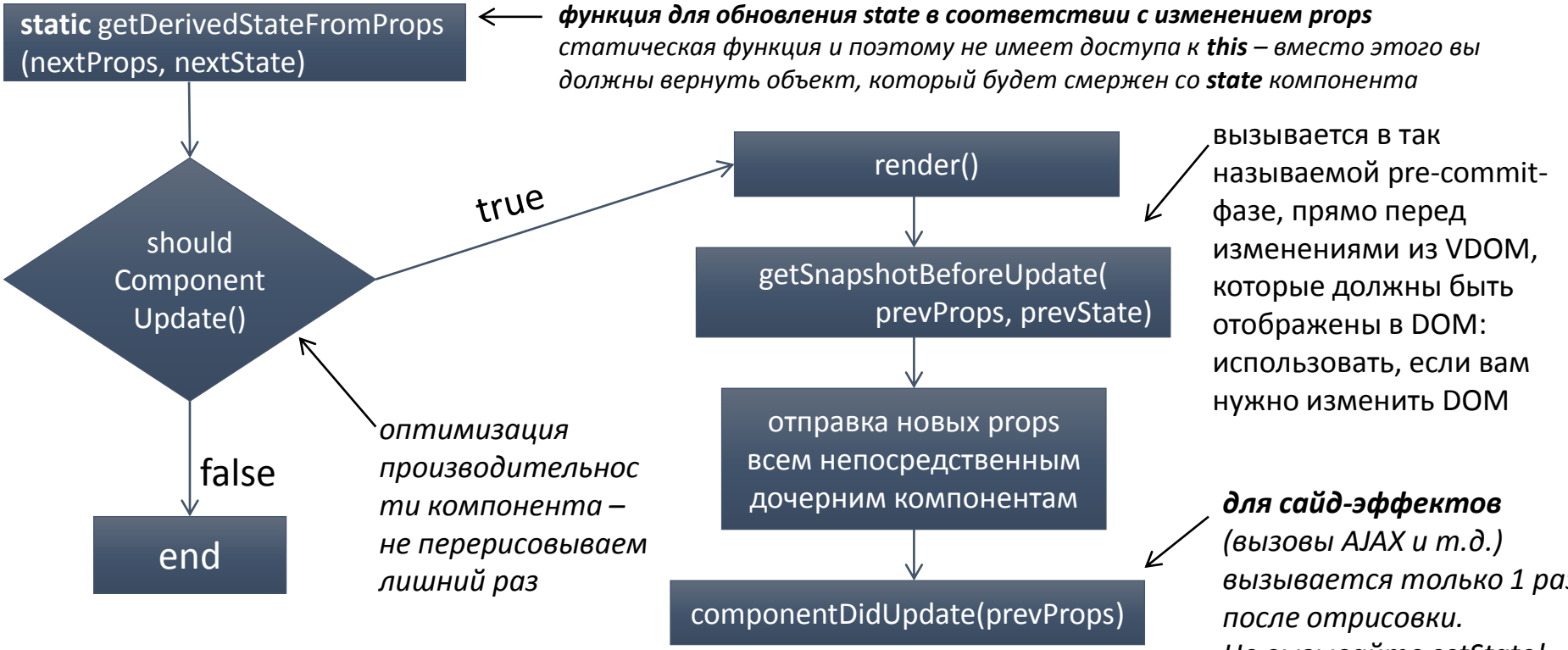
ЖИЗНЕННЫЕ ЦИКЛЫ

ре-рендеринг при изменении props из родительского компонента



ЖИЗНЕННЫЕ ЦИКЛЫ

ре-рендеринг при изменении props из родительского компонента (React 16.3+)



ОБНОВЛЕНИЕ: SHOULDCOMPONENTUPDATE

- Используйте shouldComponentUpdate() как возможность для возврата значения false, когда вы уверены, что переход к новым свойствам и состоянию не требует обновления компонента.

Используйте shouldComponentUpdate(), чтобы позволить React знать, не влияет ли на результат компонента текущее изменение state или props.

Поведение по умолчанию заключается в **повторном рендеринге при каждом изменении состояния**, и в подавляющем большинстве случаев вы должны полагаться на поведение по умолчанию.

Этот метод не вызывается для начальной визуализации или когда используется forceUpdate ().

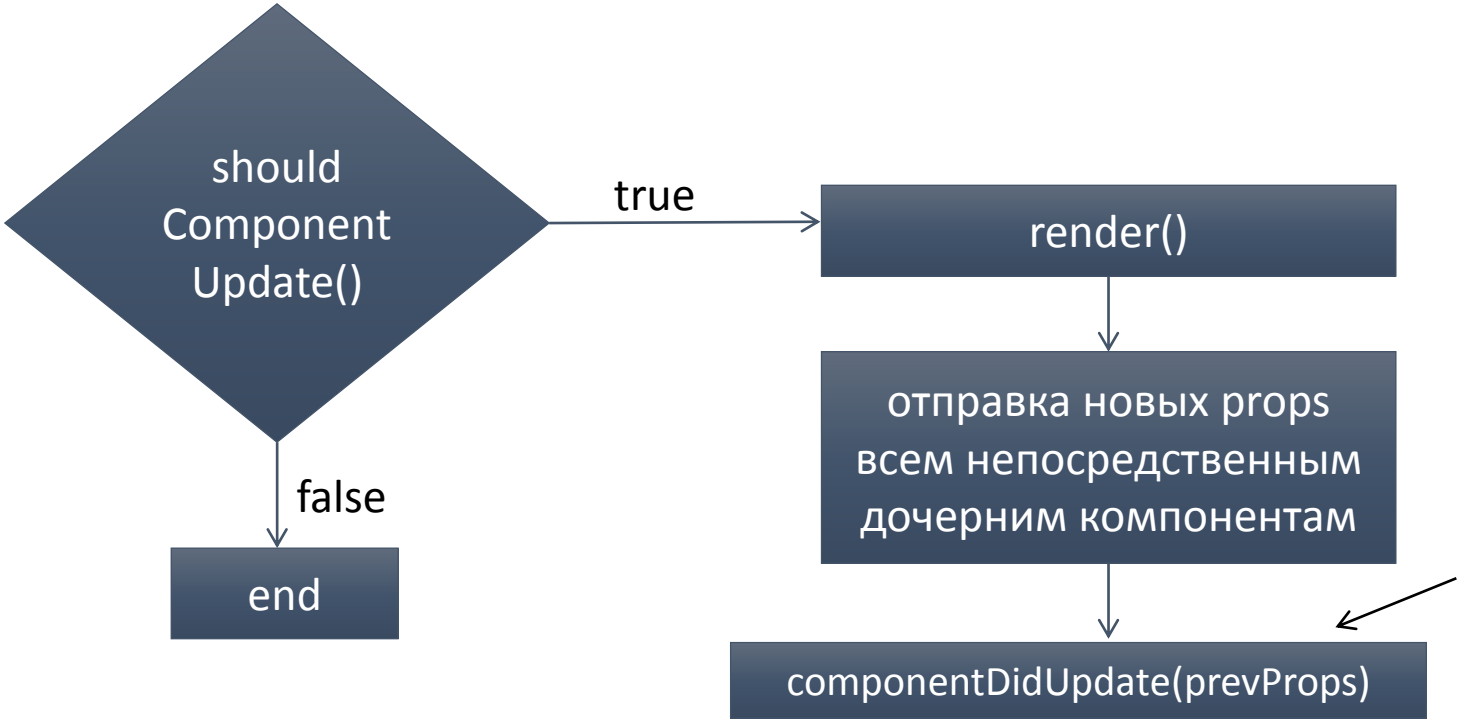
```
class ExampleComponent extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    // поверхностное сравнение state и props
    if (this.props === nextProps &&
        this.state === nextState) return false;
    else return true;
  }
}
```

PureComponent уже имеет реализованный **shouldComponentUpdate(),** который выполняет поверхностное сравнение state и props.

Обратите внимание, что возвращение **false** не предотвращает повторный рендеринг дочерних компонентов при изменении их состояния.

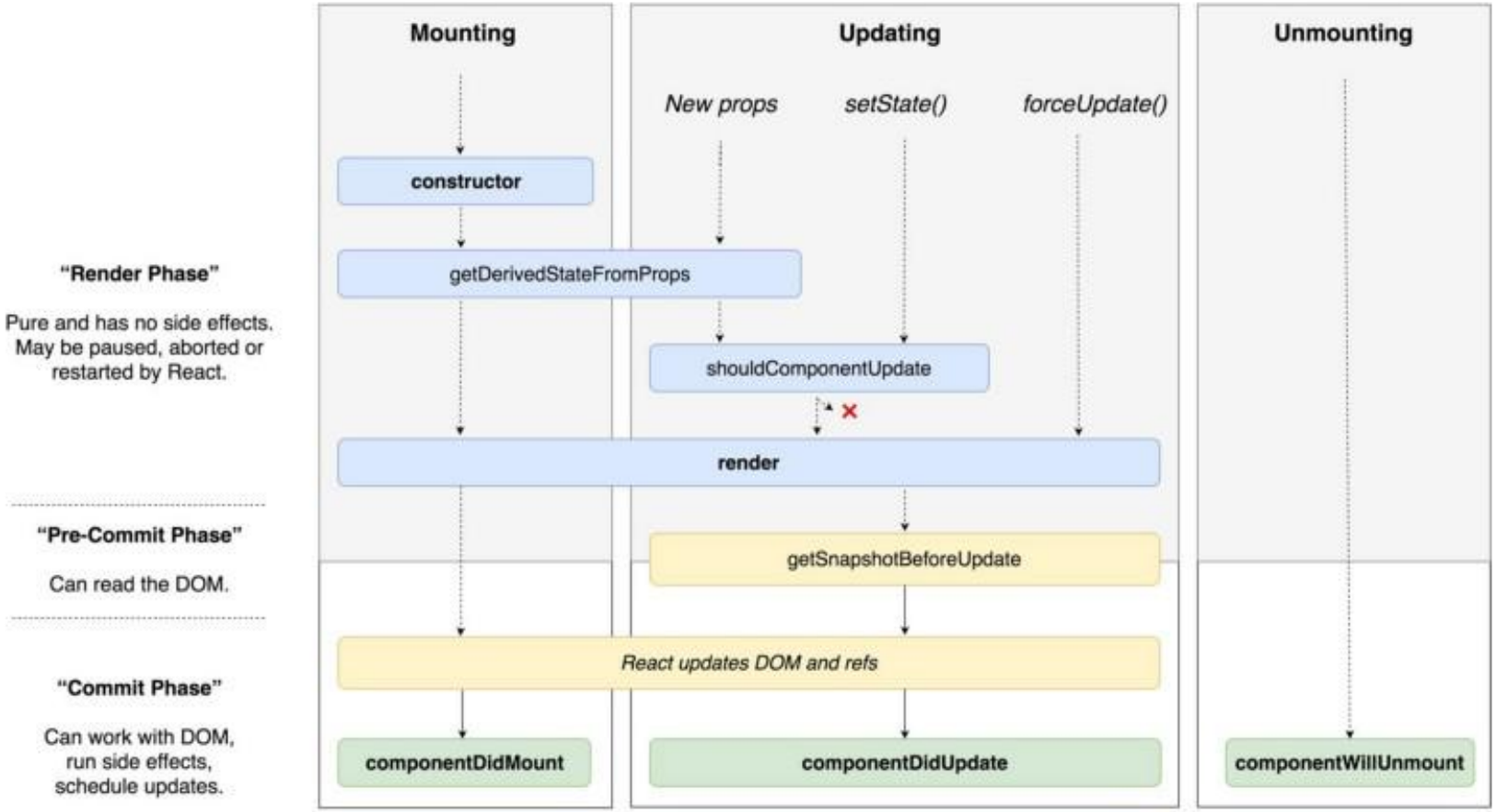
ЖИЗНЕННЫЕ ЦИКЛЫ

ре-рендеринг при вызове setState() в 16.3+



для сайд-эффектов *(вызовы AJAX и т.д.) вызывается только 1 раз после отрисовки.*
Не вызывайте setState!

ЖИЗНЕННЫЕ ЦИКЛЫ: СВОДНАЯ СХЕМА (BY DAN ABRAMOV)



АНТИ-ПАТТЕРН: БЕЗУСЛОВНОЕ КОПИРОВАНИЕ PROPS В STATE

```
class EmailInput extends Component {
  state = { email: this.props.email };

  render() {
    return <input
             onChange={this.handleChange}
             value={this.state.email} />;
  }

  handleChange = event => {
    this.setState({ email: event.target.value });
  };

  componentWillReceiveProps(nextProps) {
    this.setState({ email: nextProps.email });
  }
}
```

Распространенным заблуждением является то, что **getDerivedStateFromProps** и **componentWillReceiveProps** вызываются только в том случае, если props «изменяются». НЕТ!
Они вызываются в любое время родительскими компонентами, независимо от того, изменились ли props.
Поэтому нельзя просто копировать состояние из state – это приведет к потере обновлений!

Это уничтожит все локальные обновления! Не делайте так.

РЕШЕНИЕ 1: ИСПОЛЬЗУЕМ УПРАВЛЯЕМЫЙ КОМПОНЕНТ

```
class App extends Component {
  state = {  draftEmail: this.props.user.email };

  handleEmailChange = event => {
    this.setState({ draftEmail: event.target.value });
  };

  resetForm = () => {
    this.setState({
      draftEmail: this.props.user.email
    });
  };

  render() {
    return  <div>
          <ControlledEmailInput
            email={this.state.draftEmail}
            handleChange={this.handleEmailChange}
          />
          <button onClick={this.resetForm}>Reset</button>
         </div>;
  }
```

```
const ControlledEmailInput =
      ({email,handleChange})=
            <label>Email: <input
              value={email}
              onChange={handleChange} />
            </label>;
```

Здесь состояние EmailInput полностью контролируется App, поэтому проблем не возникает.

РЕШЕНИЕ 2: ИСПОЛЬЗУЕМ НЕУПРАВЛЯЕМЫЙ КОМПОНЕНТ С КЛЮЧОМ

```
class EmailInput extends Component {
  state = { email: this.props.defaultEmail };

  handleChange = event => {
    this.setState({ email: event.target.value });
  };

  render() {
    return <input onChange={this.handleChange} value={this.state.email} />;
  }
}

<EmailInput
  defaultEmail={this.props.user.email}
  key={this.props.user.id}
/>
```

*Компонент будет **пересоздан** при изменении **key,** а значит, state будет сброшен и email будет установлен в defaultEmail.*

*Возможно, было бы разумнее поставить **ключ на всю форму**. Каждый раз, когда ключ изменяется, все компоненты в форме будут пересозданы с начальным состоянием.*

ОБРАБОТКА ОШИБОК В КОМПОНЕНТЕ

- componentDidCatch(errorString, errorInfo)
- Дополнение в React 16 – этот метод жизненного цикла является особым, т.к. он позволяет реагировать на события, происходящие в дочернем компоненте, а конкретно на любые неперехваченные ошибки в любом из дочерних компонентов.
- С помощью этого дополнения вы можете сделать ваш родительский элемент обработчиком ошибок. Например, писать информацию об ошибке в состояние компонента, возвращать соответствующее сообщение в рендер или делать логирование ошибки.

```
componentDidCatch(errorString, errorInfo) {
   this.setState({
      error: errorString
   });
   ErrorLoggingTool.log(errorInfo);
}
render() {
   if(this.state.error)
      return <ShowErrorMessage error={this.state.error} />
   return (
      // render normal component output
   );
}
```

Блок 2.
Задание 1.
Работа с вложенными элементами React.