

ДИСЦИПЛИНА	Фронтенд и бэкенд разработка
ИНСТИТУТ	ИПТИП
КАФЕДРА	Индустриального программирования
ВИД УЧЕБНОГО МАТЕРИАЛА	Методические указания к практическим занятиям
ПРЕПОДАВАТЕЛЬ	Астафьев Рустам Уралович
СЕМЕСТР	1 семестр, 2025/2026 уч. год

Ссылка на материал:

<https://github.com/astafiev-rustam/frontend-and-backend-development/tree/practice-1-20>

Практическое занятие 20: Менеджер состояний и компонентов

В рамках данного занятия рассмотрим работу с состоянием компонентов с помощью хука `useState` и управление взаимодействием между компонентами. Подробная информация о нём есть, как в лекциях, так и в ресурсах, например:

<https://purpleschool.ru/knowledge-base/article/usestate-react-js>

Теоретическая часть

Пример 1. Базовое использование `useState`

Постановка задачи

В предыдущей практике мы создавали компоненты, которые отображали статичные данные. Но что, если нам нужно, чтобы интерфейс реагировал на действия пользователя? Например, создадим счётчик, где пользователь может увеличивать, уменьшать и сбрасывать значение. Проблема в том, что обычные переменные JavaScript не заставят React обновить интерфейс при их изменении.

Ключевая идея: используем хук `useState` — специальную функцию React, которая позволяет компоненту "запоминать" данные между рендерами и обновлять интерфейс при их изменении.

Что такое состояние (state) в React?

Состояние (state) — это данные, которые компонент хранит и может изменять. Когда состояние изменяется, React автоматически перерисовывает компонент с новыми данными.

Почему обычные переменные не работают:

```
function Counter() {  
  let count = 0; // обычная переменная
```

```
const increment = () => {  
  count = count + 1; // изменяем переменную  
  console.log(count); // в консоли значение изменится  
  // но интерфейс НЕ обновится!  
};  
  
return <div>{count}</div>;  
}
```

Проблема: React не знает, что переменная изменилась, поэтому не перерисовывает компонент.

Что такое хук `useState`?

Хук — это специальная функция React, которая начинается с `use`. Хуки позволяют использовать возможности React (состояние, эффекты и др.) в функциональных компонентах.

`useState` — это хук для создания состояния. Он возвращает массив из двух элементов:

1. Текущее значение состояния
2. Функцию для обновления этого значения

Синтаксис:

```
const [значение, функцияОбновления] = useState(начальноеЗначение);
```

Пример:

```
const [count, setCount] = useState(0);  
// count — текущее значение (начально 0)  
// setCount — функция для изменения count
```

Создадим компонент счётчика

Шаг 1. Создаём файл компонента

В папке `src` создадим новый файл `Counter.jsx`. Этот компонент будет нашим интерактивным счётчиком.

Шаг 2. Импортируем `useState`

Начнём с импорта хука `useState` из библиотеки React:

```
// src/Counter.jsx  
import { useState } from 'react';  
  
function Counter() {  
  return (  

```

```
    <div className="counter">
      <h2>Счетчик</h2>
    </div>
  );
}

export default Counter;
```

Пояснение:

- `import { useState } from 'react'` — импортируем хук из React
- Фигурные скобки `{}` означают именованный импорт

Шаг 3. Создаём состояние

Добавим состояние для хранения значения счётчика:

```
// src/Counter.jsx
import { useState } from 'react';

function Counter() {
  // создаём состояние с начальным значением 0
  const [count, setCount] = useState(0);

  return (
    <div className="counter">
      <h2>Счетчик: {count}</h2>
      <p>Текущее значение: <strong>{count}</strong></p>
    </div>
  );
}

export default Counter;
```

Пояснение:

- `useState(0)` — создаём состояние с начальным значением 0
- `count` — переменная, хранящая текущее значение
- `setCount` — функция для изменения значения `count`
- `{count}` — отображаем значение в интерфейсе

Шаг 4. Добавляем функции для изменения состояния

Создадим функции для увеличения, уменьшения и сброса счётчика:

```
// src/Counter.jsx
import { useState } from 'react';

function Counter() {
```

```
const [count, setCount] = useState(0);

// функция для увеличения счётчика
const increment = () => {
  setCount(count + 1); // устанавливаем новое значение
};

// функция для уменьшения счётчика
const decrement = () => {
  setCount(count - 1);
};

// функция для сброса счётчика
const reset = () => {
  setCount(0); // возвращаем к начальному значению
};

return (
  <div className="counter">
    <h2>Счетчик: {count}</h2>
    <p>Текущее значение: <strong>{count}</strong></p>
  </div>
);
}

export default Counter;
```

Пояснение:

- `setCount(count + 1)` — вызываем функцию обновления с новым значением
- При вызове `setCount` React автоматически перерисует компонент
- Каждая функция делает одно действие — это хорошая практика

Шаг 5. Добавляем кнопки

Теперь добавим кнопки, которые будут вызывать наши функции:

```
// src/Counter.jsx
import { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  const reset = () => {
```

```
    setCount(0);
  };

  return (
    <div className="counter">
      <h2>Счетчик: {count}</h2>
      <div className="counter-buttons">
        { /* при клике вызываем функцию decrement */ }
        <button onClick={decrement}>-1</button>
        <button onClick={reset}>Сбросить</button>
        <button onClick={increment}>+1</button>
      </div>
      <p>Текущее значение: <strong>{count}</strong></p>
    </div>
  );
}

export default Counter;
```

Пояснение:

- `onClick={increment}` — передаём функцию (без скобок!) как обработчик события
- При клике на кнопку React вызовет эту функцию
- Функция обновит состояние, и компонент перерисуеться

Шаг 6. Подключаем компонент в App.jsx

Теперь используем наш счётчик в главном приложении:

```
// src/App.jsx
import './App.css';
import Counter from './Counter';

function App() {
  return (
    <div className="App">
      <Counter />
    </div>
  );
}

export default App;
```

Пояснение:

- Импортируем компонент `Counter`
- Используем его как обычный JSX-элемент
- Каждый раз при клике на кнопки счётчик будет обновляться

Шаг 7. Проверяем результат

Сохраним все файлы и посмотрим на результат в браузере. При клике на кнопки значение счётчика должно изменяться, а интерфейс — обновляться автоматически.

Важные моменты о `useState`

1. **Состояние локально для компонента:** каждый экземпляр компонента имеет своё собственное состояние
2. **Не изменяйте состояние напрямую:** всегда используйте функцию обновления (`setCount`), а не `count = 5`
3. **Обновление асинхронно:** React может группировать несколько обновлений для оптимизации
4. **Начальное значение используется только один раз:** при первом рендере компонента
5. **Хуки вызываются на верхнем уровне:** не используйте `useState` внутри условий или циклов

Частые ошибки и как их избежать

- **Передача функции со скобками:** `onClick={increment()}` вызовет функцию сразу, а не при клике. Правильно: `onClick={increment}`
- **Попытка изменить состояние напрямую:** `count = count + 1` не работает. Используйте `setCount(count + 1)`
- **Забыли импортировать `useState`:** добавьте `import { useState } from 'react'` в начало файла
- **Использование старого значения:** если нужно обновить состояние на основе предыдущего значения несколько раз подряд, используйте функциональную форму: `setCount(prev => prev + 1)`

Полный код примера

Файл `src/Counter.jsx`:

```
// src/Counter.jsx
import { useState } from 'react';

function Counter() {
  // создаём состояние с начальным значением 0
  const [count, setCount] = useState(0);

  // функция для увеличения счётчика
  const increment = () => {
    setCount(count + 1);
  };

  // функция для уменьшения счётчика
  const decrement = () => {
    setCount(count - 1);
  };

  // функция для сброса счётчика
  const reset = () => {
    setCount(0);
  };
}
```

```
};

return (
  <div className="counter">
    <h2>Счетчик: {count}</h2>
    <div className="counter-buttons">
      { /* при клике вызываем соответствующую функцию */ }
      <button onClick={decrement}>-1</button>
      <button onClick={reset}>Сбросить</button>
      <button onClick={increment}>+1</button>
    </div>
    <p>Текущее значение: <strong>{count}</strong></p>
  </div>
);
}

export default Counter;
```

Файл `src/App.jsx`:

```
// src/App.jsx
import './App.css';
import Counter from './Counter';

function App() {
  return (
    <div className="App">
      <Counter />
    </div>
  );
}

export default App;
```

Пример 2. Работа с формой и состоянием

Постановка задачи

В предыдущем примере мы работали с простым числовым значением. Но что, если нужно хранить более сложные данные? Создадим форму регистрации, где пользователь вводит имя, email и пароль. Нам нужно сохранять все эти данные, валидировать их в реальном времени и отправлять при нажатии кнопки.

Ключевая идея: используем `useState` с объектом для хранения всех данных формы в одном состоянии. Это удобнее, чем создавать отдельное состояние для каждого поля.

Работа с формами в React

В обычном HTML формы управляют своими данными сами. Но в React мы хотим, чтобы React контролировал значения полей — это называется **контролируемые компоненты** (controlled components).

Принцип работы:

1. Значение поля хранится в состоянии React
2. При изменении поля вызывается обработчик, который обновляет состояние
3. React перерисовывает поле с новым значением из состояния

Состояние-объект vs множество состояний

Можно создать отдельное состояние для каждого поля:

```
const [name, setName] = useState('');
const [email, setEmail] = useState('');
const [password, setPassword] = useState('');
```

Но удобнее использовать один объект:

```
const [formData, setFormData] = useState({
  name: '',
  email: '',
  password: ''
});
```

Преимущества объекта:

- Все данные формы в одном месте
- Легко передать данные на сервер
- Проще управлять сложными формами

Создадим форму регистрации

Шаг 1. Создаём файл компонента

В папке **src** создадим новый файл **RegistrationForm.jsx**. Этот компонент будет содержать форму регистрации.

Шаг 2. Создаём состояние для данных формы

Начнём с создания состояния-объекта для хранения всех полей формы:

```
// src/RegistrationForm.jsx
import { useState } from 'react';

function RegistrationForm() {
```

```
// создаём состояние-объект для всех полей формы
const [formData, setFormData] = useState({
  name: '',
  email: '',
  password: ''
});

return (
  <form className="registration-form">
    <h2>Регистрация</h2>
  </form>
);
}

export default RegistrationForm;
```

Пояснение:

- `useState({...})` — передаём объект как начальное значение
- Все поля изначально пустые строки
- `formData` — объект с тремя свойствами

Шаг 3. Добавляем поля ввода

Создадим контролируемые поля ввода, связанные с состоянием:

```
// src/RegistrationForm.jsx
import { useState } from 'react';

function RegistrationForm() {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    password: ''
  });

  return (
    <form className="registration-form">
      <h2>Регистрация</h2>

      <div className="form-group">
        <label>Имя:</label>
        <input
          type="text"
          name="name"
          value={formData.name} // значение из состояния
          required
        />
      </div>

      <div className="form-group">
        <label>Email:</label>
```

```
        <input
          type="email"
          name="email"
          value={formData.email}
          required
        />
      </div>

      <div className="form-group">
        <label>Пароль:</label>
        <input
          type="password"
          name="password"
          value={formData.password}
          required
        />
      </div>

      <button type="submit">Зарегистрироваться</button>
    </form>
  );
}

export default RegistrationForm;
```

Пояснение:

- `value={formData.name}` — значение поля берётся из состояния
- `name="name"` — атрибут `name` соответствует ключу в объекте `formData`
- Пока поля только для чтения, так как нет обработчика изменений

Шаг 4. Создаём обработчик изменений

Добавим функцию, которая будет обновлять состояние при вводе текста:

```
// src/RegistrationForm.jsx
import { useState } from 'react';

function RegistrationForm() {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    password: ''
  });

  // обработчик изменения любого поля
  const handleChange = (e) => {
    // получаем name и value из события
    const { name, value } = e.target;

    // обновляем состояние
    setFormData(prev => ({
```

```
    ...prev,          // копируем все существующие поля
    [name]: value      // обновляем конкретное поле
  }));
};

return (
  <form className="registration-form">
    <h2>Регистрация</h2>

    <div className="form-group">
      <label>Имя:</label>
      <input
        type="text"
        name="name"
        value={formData.name}
        onChange={handleChange} // добавляем обработчик
        required
      />
    </div>

    <div className="form-group">
      <label>Email:</label>
      <input
        type="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
        required
      />
    </div>

    <div className="form-group">
      <label>Пароль:</label>
      <input
        type="password"
        name="password"
        value={formData.password}
        onChange={handleChange}
        required
      />
    </div>

    <button type="submit">Зарегистрироваться</button>
  </form>
);
}

export default RegistrationForm;
```

Пояснение:

- `e.target` — элемент, который вызвал событие (поле ввода)

- `{ name, value }` — деструктурируем атрибуты элемента
- `...prev` — оператор spread, копирует все поля из предыдущего состояния
- `[name]: value` — вычисляемое свойство, обновляет нужное поле
- Один обработчик работает для всех полей благодаря атрибуту `name`

Шаг 5. Добавляем валидацию

Создадим состояние для ошибок и добавим простую валидацию email:

```
// src/RegistrationForm.jsx
import { useState } from 'react';

function RegistrationForm() {
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    password: ''
  });

  // состояние для хранения ошибок валидации
  const [errors, setErrors] = useState({});

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData(prev => ({
      ...prev,
      [name]: value
    }));

    // валидация email в реальном времени
    if (name === 'email' && value && !value.includes('@')) {
      // если email не содержит @, показываем ошибку
      setErrors(prev => ({ ...prev, email: 'Некорректный email' }));
    } else if (name === 'email') {
      // если email корректен, убираем ошибку
      setErrors(prev => ({ ...prev, email: '' }));
    }
  };

  return (
    <form className="registration-form">
      <h2>Регистрация</h2>

      <div className="form-group">
        <label>Имя:</label>
        <input
          type="text"
          name="name"
          value={formData.name}
          onChange={handleChange}
          required
        />
      </div>
    </form>
  );
}
```

```

    </div>

    <div className="form-group">
      <label>Email:</label>
      <input
        type="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
        required
      />
      {/* показываем ошибку, если она есть */}
      {errors.email && <span className="error">{errors.email}</span>}
    </div>

    <div className="form-group">
      <label>Пароль:</label>
      <input
        type="password"
        name="password"
        value={formData.password}
        onChange={handleChange}
        required
      />
    </div>

    <button type="submit">Зарегистрироваться</button>
  </form>
);
}

export default RegistrationForm;

```

Пояснение:

- `errors` — объект для хранения сообщений об ошибках
- `{errors.email && ...}` — условный рендеринг: показываем ошибку только если она есть
- Валидация происходит при каждом изменении поля

Шаг 6. Добавляем обработчик отправки формы

Создадим функцию для обработки отправки формы:

```

// src/RegistrationForm.jsx
import { useState } from 'react';

function RegistrationForm() {
  const [formData, setFormData] = useState({
    name: '',
    email: '',

```

```
    password: ''
  });

  const [errors, setErrors] = useState({});

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData(prev => ({
      ...prev,
      [name]: value
    }));

    if (name === 'email' && value && !value.includes('@')) {
      setErrors(prev => ({ ...prev, email: 'Некорректный email' }));
    } else if (name === 'email') {
      setErrors(prev => ({ ...prev, email: '' }));
    }
  };

  // обработчик отправки формы
  const handleSubmit = (e) => {
    e.preventDefault(); // предотвращаем перезагрузку страницы
    console.log('Данные формы:', formData);
    alert(`Добро пожаловать, ${formData.name}!`);
  };

  return (
    <form onSubmit={handleSubmit} className="registration-form">
      <h2>Регистрация</h2>

      <div className="form-group">
        <label>Имя:</label>
        <input
          type="text"
          name="name"
          value={formData.name}
          onChange={handleChange}
          required
        />
      </div>

      <div className="form-group">
        <label>Email:</label>
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
          required
        />
        {errors.email && <span className="error">{errors.email}</span>}
      </div>

      <div className="form-group">
```

```
    <label>Пароль:</label>
    <input
      type="password"
      name="password"
      value={formData.password}
      onChange={handleChange}
      required
    />
  </div>

  <button type="submit">Зарегистрироваться</button>
</form>
);
}

export default RegistrationForm;
```

Пояснение:

- `e.preventDefault()` — отменяем стандартное поведение формы (перезагрузку страницы)
- `onSubmit={handleSubmit}` — привязываем обработчик к форме
- В реальном приложении здесь был бы запрос на сервер

Шаг 7. Подключаем компонент в App.jsx

Добавим форму в главное приложение:

```
// src/App.jsx
import './App.css';
import Counter from './Counter';
import RegistrationForm from './RegistrationForm';

function App() {
  return (
    <div className="App">
      <Counter />
      <RegistrationForm />
    </div>
  );
}

export default App;
```

Шаг 8. Проверяем результат

Сохраним файлы и проверим в браузере. При вводе текста в поля значения должны обновляться, при вводе некорректного email должна появляться ошибка, а при отправке формы — всплывающее окно с приветствием.

Важные моменты о работе с формами

1. **Контролируемые компоненты:** значение поля всегда синхронизировано с состоянием
2. **Не мутируйте объекты:** всегда создавайте новый объект с помощью spread-оператора
3. **Функциональное обновление:** используйте `prev =>` когда новое значение зависит от предыдущего
4. **Один обработчик для всех полей:** используйте атрибут `name` для идентификации поля
5. **Предотвращайте перезагрузку:** всегда вызывайте `e.preventDefault()` в `onSubmit`

Частые ошибки и как их избежать

- **Мутация объекта:** `formData.name = value` не работает. Используйте `setFormData({...formData, name: value})`
- **Забыли spread-оператор:** без `...prev` потеряете все остальные поля объекта
- **Неправильный атрибут name:** убедитесь, что `name` в input соответствует ключу в объекте состояния
- **Забыли preventDefault:** форма перезагрузит страницу при отправке

Полный код примера

Файл `src/RegistrationForm.jsx`:

```
// src/RegistrationForm.jsx
import { useState } from 'react';

function RegistrationForm() {
  // состояние для данных формы
  const [formData, setFormData] = useState({
    name: '',
    email: '',
    password: ''
  });

  // состояние для ошибок валидации
  const [errors, setErrors] = useState({});

  // обработчик изменения любого поля
  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData(prev => ({
      ...prev,
      [name]: value
    }));

    // валидация email в реальном времени
    if (name === 'email' && value && !value.includes('@')) {
      setErrors(prev => ({ ...prev, email: 'Некорректный email' }));
    } else if (name === 'email') {
      setErrors(prev => ({ ...prev, email: '' }));
    }
  };
}
```

```
// обработчик отправки формы
const handleSubmit = (e) => {
  e.preventDefault();
  console.log('Данные формы:', formData);
  alert(`Добро пожаловать, ${formData.name}!`);
};

return (
  <form onSubmit={handleSubmit} className="registration-form">
    <h2>Регистрация</h2>

    <div className="form-group">
      <label>Имя:</label>
      <input
        type="text"
        name="name"
        value={formData.name}
        onChange={handleChange}
        required
      />
    </div>

    <div className="form-group">
      <label>Email:</label>
      <input
        type="email"
        name="email"
        value={formData.email}
        onChange={handleChange}
        required
      />
      {errors.email && <span className="error">{errors.email}</span>}
    </div>

    <div className="form-group">
      <label>Пароль:</label>
      <input
        type="password"
        name="password"
        value={formData.password}
        onChange={handleChange}
        required
      />
    </div>

    <button type="submit">Зарегистрироваться</button>
  </form>
);
}

export default RegistrationForm;
```

```
// src/App.jsx
import './App.css';
import Counter from './Counter';
import RegistrationForm from './RegistrationForm';

function App() {
  return (
    <div className="App">
      <Counter />
      <RegistrationForm />
    </div>
  );
}

export default App;
```

Пример 3. Подъем состояния (Lifting State Up)

Постановка задачи

Представим ситуацию: у нас есть два компонента — один отображает выбранный цвет, а другой содержит кнопки для выбора цвета. Проблема в том, что оба компонента должны знать, какой цвет выбран. Если каждый компонент будет хранить своё состояние, они не смогут синхронизироваться.

Ключевая идея: "поднимаем" состояние в общий родительский компонент. Родитель хранит данные и передаёт их дочерним компонентам через props. Это называется **Lifting State Up** (подъём состояния).

Что такое подъём состояния?

Подъём состояния — это паттерн в React, когда состояние перемещается из дочерних компонентов в их общего родителя. Это позволяет нескольким компонентам использовать одни и те же данные.

Принцип работы:

1. Состояние хранится в родительском компоненте
2. Родитель передаёт данные дочерним компонентам через props
3. Родитель передаёт функции для изменения состояния через props
4. Дочерние компоненты вызывают эти функции для обновления данных

Когда нужен подъём состояния?

Используйте подъём состояния, когда:

- Несколько компонентов должны отображать одни и те же данные
- Изменение в одном компоненте должно отражаться в другом
- Компоненты должны быть синхронизированы

Пример: корзина покупок и счётчик товаров в шапке сайта должны показывать одинаковое количество товаров.

Создадим компонент выбора цвета

Шаг 1. Создаём файл и дочерние компоненты

В папке `src` создадим новый файл `ColorPicker.jsx`. Начнём с создания двух дочерних компонентов:

```
// src/ColorPicker.jsx
import { useState } from 'react';

// компонент для отображения выбранного цвета
function ColorDisplay({ color }) {
  return (
    <div
      className="color-display"
      style={{
        backgroundColor: color,
        width: '200px',
        height: '100px',
        margin: '10px 0'
      }}
    >
      <p>Выбранный цвет: {color}</p>
    </div>
  );
}

// компонент с кнопками для выбора цвета
function ColorControls({ color, onColorChange }) {
  const colors = ['#ff0000', '#00ff00', '#0000ff', '#ffff00', '#ff00ff'];

  return (
    <div className="color-controls">
      <h3>Выберите цвет:</h3>
      <div className="color-buttons">
        {colors.map((col) => (
          <button
            key={col}
            style={{ backgroundColor: col }}
            onClick={() => onColorChange(col)}
            className={color === col ? 'active' : ''}
          >
            {col}
          </button>
        ))}
      </div>
    </div>
  );
}

export default ColorPicker;
```

Пояснение:

- `ColorDisplay` — просто отображает цвет, получает его через props
- `ColorControls` — отображает кнопки выбора цвета
- Оба компонента пока не имеют своего состояния

Шаг 2. Анализируем проблему

Если бы каждый компонент хранил своё состояние:

```
// ✗ Неправильно - состояния не синхронизированы
function ColorDisplay() {
  const [color, setColor] = useState('#ff0000');
  // этот компонент не знает о выборе в ColorControls
}

function ColorControls() {
  const [color, setColor] = useState('#ff0000');
  // этот компонент не может обновить ColorDisplay
}
```

Проблема: компоненты изолированы, изменения в одном не влияют на другой.

Шаг 3. Создаём родительский компонент с состоянием

Создадим родительский компонент `ColorPicker`, который будет хранить состояние:

```
// src/ColorPicker.jsx
import { useState } from 'react';

function ColorDisplay({ color }) {
  return (
    <div
      className="color-display"
      style={{
        backgroundColor: color,
        width: '200px',
        height: '100px',
        margin: '10px 0'
      }}
    >
    <p>Выбранный цвет: {color}</p>
    </div>
  );
}

function ColorControls({ color, onColorChange }) {
  const colors = ['#ff0000', '#00ff00', '#0000ff', '#ffff00', '#ff00ff'];

  return (
    <div className="color-controls">
```

```

    <h3>Выберите цвет:</h3>
    <div className="color-buttons">
      {colors.map((col) => (
        <button
          key={col}
          style={{ backgroundColor: col }}
          onClick={() => onColorChange(col)}
          className={color === col ? 'active' : ''}
        >
          {col}
        </button>
      ))}
    </div>
  </div>
);
}

// родительский компонент - хранит состояние
function ColorPicker() {
  // состояние находится здесь, в родителе
  const [selectedColor, setSelectedColor] = useState('#ff0000');

  return (
    <div className="color-picker">
      <h2>Выбор цвета</h2>
      {/* передаём текущий цвет в ColorDisplay */}
      <ColorDisplay color={selectedColor} />
      {/* передаём и цвет, и функцию для его изменения */}
      <ColorControls
        color={selectedColor}
        onColorChange={setSelectedColor}
      />
    </div>
  );
}

export default ColorPicker;

```

Пояснение:

- `selectedColor` — состояние хранится в родителе `ColorPicker`
- `ColorDisplay` получает `color` через props — только для чтения
- `ColorControls` получает `color` и `onColorChange` — может изменять состояние
- `onColorChange={setSelectedColor}` — передаём функцию обновления состояния

Шаг 4. Понимаем поток данных

Поток данных в этом примере:

1. **Состояние в родителе:** `ColorPicker` хранит `selectedColor`
2. **Данные вниз:** `ColorPicker` передаёт `selectedColor` обоим дочерним компонентам

3. **События вверх:** когда пользователь кликает на кнопку в `ColorControls`, вызывается `onColorChange`
4. **Обновление состояния:** `onColorChange` это на самом деле `setSelectedColor`, которая обновляет состояние
5. **Перерисовка:** React перерисовывает `ColorPicker` и все его дочерние компоненты с новым цветом

Схема:

```
ColorPicker (состояние: selectedColor)
  ↓ передаёт color
  └─ ColorDisplay (отображает цвет)
  └─ ColorControls (изменяет цвет через onColorChange)
      ↑ вызывает onColorChange
      └─ обновляет selectedColor в родителе
```

Шаг 5. Детали реализации

Обратите внимание на ключевые моменты:

```
// в ColorControls
onClick={() => onColorChange(col)}
```

- Стрелочная функция нужна, чтобы передать аргумент `col`
- Без стрелочной функции `onColorChange` вызовется сразу

```
// в ColorPicker
onColorChange={setSelectedColor}
```

- Передаём саму функцию, а не результат её вызова
- Дочерний компонент вызовет её с нужным аргументом

Шаг 6. Подключаем компонент в App.jsx

Добавим компонент выбора цвета в главное приложение:

```
// src/App.jsx
import './App.css';
import Counter from './Counter';
import RegistrationForm from './RegistrationForm';
import ColorPicker from './ColorPicker';

function App() {
  return (
    <div className="App">
```

```
        <Counter />
        <RegistrationForm />
        <ColorPicker />
      </div>
    );
  }

  export default App;
```

Шаг 7. Проверяем результат

Сохраним файлы и проверим в браузере. При клике на кнопки цвета должен меняться как цвет фона в `ColorDisplay`, так и активная кнопка в `ColorControls`.

Важные моменты о подъёме состояния

1. **Единственный источник истины:** состояние хранится в одном месте — в родительском компоненте
2. **Однонаправленный поток данных:** данные передаются сверху вниз через props
3. **События поднимаются вверх:** дочерние компоненты вызывают функции родителя для изменения состояния
4. **Синхронизация автоматическая:** все компоненты всегда видят актуальные данные
5. **Дочерние компоненты проще:** они не управляют состоянием, только отображают данные

Частые ошибки и как их избежать

- **Дублирование состояния:** не создавайте копии состояния в дочерних компонентах. Используйте props.
- **Забыли передать функцию:** если дочерний компонент должен изменять данные, передайте ему функцию через props.
- **Неправильная передача функции:** передавайте `onColorChange={setColor}`, а не `onColorChange={setColor()}`.
- **Мутация props:** никогда не изменяйте props напрямую. Вызывайте переданную функцию.

Полный код примера

Файл `src/ColorPicker.jsx`:

```
// src/ColorPicker.jsx
import { useState } from 'react';

// компонент для отображения выбранного цвета
function ColorDisplay({ color }) {
  return (
    <div
      className="color-display"
      style={{
        backgroundColor: color,
        width: '200px',
```

```

        height: '100px',
        margin: '10px 0',
        border: '2px solid #333',
        borderRadius: '8px',
        display: 'flex',
        alignItems: 'center',
        justifyContent: 'center'
      }}
    >
    <p style={{ color: '#fff', textShadow: '1px 1px 2px #000' }}>
      Выбранный цвет: {color}
    </p>
  </div>
);
}

// компонент с кнопками для выбора цвета
function ColorControls({ color, onColorChange }) {
  const colors = ['#ff0000', '#00ff00', '#0000ff', '#ffff00', '#ff00ff'];

  return (
    <div className="color-controls">
      <h3>Выберите цвет:</h3>
      <div className="color-buttons">
        {colors.map((col) => (
          <button
            key={col}
            style={{
              backgroundColor: col,
              border: color === col ? '3px solid #000' : '1px solid #ccc',
              padding: '10px 20px',
              margin: '5px',
              cursor: 'pointer',
              borderRadius: '5px'
            }}
            onClick={() => onColorChange(col)}
          >
            {col}
          </button>
        ))}
      </div>
    </div>
  );
}

// родительский компонент - хранит состояние
function ColorPicker() {
  // состояние находится здесь, в родителе
  const [selectedColor, setSelectedColor] = useState('#ff0000');

  return (
    <div className="color-picker" style={{ margin: '20px 0', padding: '20px',
      border: '1px solid #ddd', borderRadius: '8px' }}>
      <h2>Выбор цвета</h2>

```

```
    { /* передаём текущий цвет в ColorDisplay */}  
    <ColorDisplay color={selectedColor} />  
    { /* передаём и цвет, и функцию для его изменения */}  
    <ColorControls  
      color={selectedColor}  
      onColorChange={setSelectedColor}  
    />  
  </div>  
);  
}  
  
export default ColorPicker;
```

Файл `src/App.jsx`:

```
// src/App.jsx  
import './App.css';  
import Counter from './Counter';  
import RegistrationForm from './RegistrationForm';  
import ColorPicker from './ColorPicker';  
  
function App() {  
  return (  
    <div className="App">  
      <Counter />  
      <RegistrationForm />  
      <ColorPicker />  
    </div>  
  );  
}  
  
export default App;
```

Публикация React Vite приложения на GitHub Pages

После завершения разработки приложения, его необходимо опубликовать на GitHub Pages. Данный процесс будет отличаться от публикации сайта в предыдущих практических занятиях. Обратите внимание! Это пошаговая инструкция по публикации React-приложения на Vite. Для публикации приложения create-react-app (не рекомендуется) требуется отдельная инструкция (можно найти в интернете).

Предварительные условия

У вас уже должно быть:

- Работющее React-приложение на Vite
- Репозиторий на GitHub с веткой `main`
- Код загружен в репозиторий (`git push`)

Шаг 1. Установка пакета gh-pages

Откроем терминал в папке проекта и установим пакет для автоматической публикации:

```
npm install --save-dev gh-pages
```

Шаг 2. Настройка vite.config.js

Откроем файл `vite.config.js` в корне проекта и добавим поле `base`:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  base: '/название-репозитория/' // добавляем эту строку
})
```

⚠ Важно:

- Замените `название-репозитория` на реальное название вашего репозитория на GitHub
- Обязательно добавьте слешы в начале и в конце: `/repo-name/`

Пример: Если ваш репозиторий называется `technology-tracker`, то:

```
base: '/technology-tracker/'
```

Шаг 3. Добавление скриптов в package.json

Откроем `package.json` и добавим два новых скрипта в раздел `"scripts"`:

```
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview",
  "predeploy": "npm run build",
  "deploy": "gh-pages -d dist"
}
```

Что делают эти скрипты:

- `predeploy` — автоматически создаст оптимизированную сборку перед публикацией

- **deploy** — опубликует содержимое папки **dist** на GitHub Pages

Шаг 4. Публикация на GitHub Pages

Теперь опубликуем приложение одной командой:

```
npm run deploy
```

Что произойдёт:

1. Автоматически создастся оптимизированная сборка проекта (папка **dist**)
2. Создастся специальная ветка **gh-pages** в вашем репозитории
3. Содержимое папки **dist** загрузится в эту ветку
4. GitHub Pages автоматически опубликует сайт

Процесс займёт 1-2 минуты. В конце увидите сообщение:

```
Published
```

Шаг 5. Проверка результата


Включаем GitHub Pages (если требуется)

1. Перейдите в ваш репозиторий на GitHub
2. Откройте **Settings** → **Pages** (в левом меню)
3. Убедитесь, что в разделе **Source** выбрана ветка **gh-pages**
4. Если нужно, нажмите **Save**

Открываем приложение

Через 1-2 минуты ваше приложение будет доступно по адресу:

```
https://ваш-username.github.io/название-репозитория/
```

 Теперь данную ссылку можно прикрепить в СДО в раздел данной практической работы.

Обновление приложения

Когда внесёте изменения в код:

```
git add .  
git commit -m "Описание изменений"
```

```
git push
npm run deploy
```

Через 1-2 минуты изменения появятся на опубликованном сайте.

Возможные проблемы и решения

Проблема 1: Страница показывает пустой экран

Причина: неправильно указан `base` в `vite.config.js`.

Решение:

1. Проверьте, что `base` указан правильно: `/название-репозитория/`
2. Убедитесь, что слеша есть и в начале, и в конце
3. Исправьте, сохраните и выполните `npm run deploy` снова

Проблема 2: Ошибка "Failed to get remote.origin.url"

Причина: не подключён удалённый репозиторий.

Решение:

```
git remote add origin https://github.com/ваш-username/название-репозитория.git
git push -u origin main
```

Проблема 3: Ресурсы не загружаются (ошибки 404)

Причина: используются абсолютные пути вместо относительных.

Решение:

1. Убедитесь, что в коде используются относительные пути для изображений и файлов
2. Проверьте консоль браузера (F12) на наличие ошибок 404
3. Используйте импорты для изображений: `import logo from './logo.svg'`

Шпаргалка по публикации (основные моменты)

Убедитесь, что выполнили все шаги:

- ☐ Установлен пакет `gh-pages`
 - ☐ Добавлено поле `base` в `vite.config.js` (с слешами: `/repo-name/`)
 - ☐ Добавлены скрипты `predeploy` и `deploy` в `package.json`
 - ☐ В скрипте `deploy` указано: `gh-pages -d dist`
 - ☐ Выполнена команда `npm run deploy`
 - ☐ Сайт открывается по адресу `https://ваш-username.github.io/название-репозитория/`
-

Источники

- **Официальная документация GitHub Pages:** <https://pages.github.com/>
- **Документация Vite о публикации:** <https://vitejs.dev/guide/static-deploy.html#github-pages>
- **Документация пакета gh-pages:** <https://www.npmjs.com/package/gh-pages>

Задание на практику

Практическая часть

Добавление состояния в карточки технологий

Шаг 1: Обновление компонента TechnologyCard Модифицируйте существующий компонент `TechnologyCard`, чтобы он мог изменять свой статус по клику. Добавьте обработчик клика, который циклически переключает статусы: "not-started" → "in-progress" → "completed" → "not-started".

Шаг 2: Создание состояния для дорожной карты В компоненте App создайте состояние для хранения массива технологий. Изначально используйте тестовые данные, но теперь с возможностью их обновления.

Пример начального состояния:

```
const [technologies, setTechnologies] = useState([
  {
    id: 1,
    title: 'React Components',
    description: 'Изучение базовых компонентов',
    status: 'not-started'
  },
  {
    id: 2,
    title: 'JSX Syntax',
    description: 'Освоение синтаксиса JSX',
    status: 'not-started'
  },
  // ... остальные технологии
]);
```

Шаг 3: Реализация функции изменения статуса Создайте функцию в компоненте App, которая будет обновлять статус конкретной технологии по id. Передайте эту функцию в каждый компонент `TechnologyCard`.

Шаг 4: Добавление интерактивности В компоненте `TechnologyCard` добавьте обработчик клика, который вызывает переданную функцию для изменения статуса. Убедитесь, что внешний вид карточки меняется в зависимости от статуса.

Улучшение пользовательского интерфейса

Шаг 5: Визуальная обратная связь Добавьте анимации или переходы при изменении статуса карточки. Можно использовать CSS-transition для плавного изменения цвета фона или границы.

Шаг 6: Статистика в реальном времени Модифицируйте компонент `ProgressHeader` (если он уже создан) или создайте новый компонент `Statistics`, который показывает:

- Количество технологий в каждом статусе
- Процент завершения
- Самую популярную категорию (если добавите категории)

Самостоятельная работа

Задание 1: Создайте компонент "Быстрые действия" (QuickActions) с кнопками:

- "Отметить все как выполненные"
- "Сбросить все статусы"
- "Случайный выбор следующей технологии".
 - Выбор технологии должен происходить только среди тех технологий, которые имеют статус "Не начато" (not-started). Технологии, имеющие статус, отличный от "Не начато" (not-started) не должны выбираться при нажатии на кнопку случайного выбора.
 - Если все технологии уже имеют статус "В процессе" или "Завершено", система должна отобразить сообщение о невозможности выбора новой технологии для изучения.

Задание 2: Реализуйте систему фильтрации технологий по статусу. Добавьте кнопки/вкладки для отображения:

- Всех технологий
- Только не начатых
- Только в процессе
- Только выполненных

Рекомендации по выполнению:

- Для фильтрации используйте метод `filter` массива `technologies`
- Создайте состояние для активного фильтра
- Не забывайте про ключи при рендере отфильтрованного списка

Что проверить перед завершением:

- Карточки меняют статус по клику
- Прогресс-бар обновляется в реальном времени
- Фильтры корректно работают
- Все изменения состояния происходят без мутаций

Дополнительные CSS-стили для интерактивности:

```
.technology-card {  
  transition: all 0.3s ease;  
  cursor: pointer;  
}
```

```
.technology-card:hover {  
  transform: translateY(-2px);  
  box-shadow: 0 4px 8px rgba(0,0,0,0.1);  
}  
  
.status-not-started { border-left-color: #ff6b6b; }  
.status-in-progress { border-left-color: #4ecdc4; }  
.status-completed { border-left-color: #45b7d1; }
```

По завершении этой работы ваше приложение станет полностью интерактивным, а пользователи смогут отслеживать свой прогресс в реальном времени.