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

Проблема. Необходимо создать компонент счетчика, где пользователь может увеличивать, уменьшать и сбрасывать значение. Без состояния React не может запоминать изменения между рендерами.

Подход к решению. Используем хук useState для создания переменной состояния и функции для её обновления.

Исходный код в файле `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">
        <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;
```

Пояснения. Хук useState возвращает массив из двух элементов - текущего значения и функции для его обновления. При вызове функции обновления компонент перерендеривается с новым значением.

Разместим компонент внутри `App.js`:

```
import logo from './logo.svg';
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

Проблема. Нужно создать форму регистрации, где данные полей ввода сохраняются и валидируются в реальном времени.

Подход к решению. Для каждого поля ввода создаем отдельное состояние, обрабатываем изменения и добавляем базовую валидацию.

Исходный код в файле `RegistrationForm.jsx`:

```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
    }));

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

  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;
```

Пояснения. Используем объект для хранения всех данных формы. При обновлении состояния обязательно создаем новый объект, а не мутируем существующий.

Добавим полученный компонент в `App.js`:

```
import logo from './logo.svg';
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)

Проблема. Несколько компонентов должны работать с одними и теми же данными и синхронизировать свои состояния.

Подход к решению. Поднимаем состояние до общего родительского компонента и передаем данные и функции обратного вызова через props.

Исходный код в файле `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>
  );
}

function ColorPicker() {
  const [selectedColor, setSelectedColor] = useState('#ff0000');

  return (
    <div className="color-picker">
      <h2>Выбор цвета</h2>
      <ColorDisplay color={selectedColor} />
```

```
        <ColorControls
          color={selectedColor}
          onColorChange={setSelectedColor}
        />
      </div>
    );
}

export default ColorPicker;
```

Пояснения. Состояние хранится в родительском компоненте ColorPicker, а дочерние компоненты получают данные и функции для их изменения через props.

Реализуем в `App.js` добавив соответствующий компонент:

```
import logo from './logo.svg';
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;
```

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"

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.