Московский государственный технический университет
имени Н. Э. Баумана

В. И. Кузнецов

Работа с веб-страницами с помощью библиотеки React

Методические указания к выполнению
лабораторной работы №5 по дисциплине
«Языки интернет программирования»

Редакция от 1.09.2025

Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

Кузнецов В.И.

Работа с веб-страницами с помощью библиотеки React: методические указания к выполнению практикума № 5 и лабораторной работы № 5 по дисциплине «Языки интернет-программирования» / В. И. Кузнецов.

Описана теория по React. Приведены примеры кода и способы запуска без дополнительных сборщиков.

Для студентов МГТУ им. Н.Э. Баумана, обучающихся по направлению «Информатика и вычислительная техника».

Теоретическая часть

React.js — это JavaScript-библиотека (иногда называется фреймворком) для удобной разработки интерфейсов, то есть внешней части сайтов и приложений, с которой взаимодействует пользователь.

React-разработка заключается в описании того, что нужно вывести на страницу (а не в составлении инструкций для браузера, посвящённых тому, как это делать). Это, кроме прочего, означает значительное сокращение объёмов шаблонного кода.

Главная фишка React.js — компоненты и состояния.

Компонент — это кусочек кода, который отвечает за внешний вид одного из элементов сайта или приложения. Причём такие кусочки-компоненты могут быть вложенными.

Состояние — это вся информация об элементе, в том числе о его отображении. Например, состояние объекта «термометр» может описываться свойствами current_temperature, min и max.

React имеет несколько разных видов компонентов, но мы начнём с подклассов React.Component:

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Список покупок для {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}
```

ShoppingList — это пример React-компонента, написанного в виде класса. Такой компонент принимает входные параметры — пропсы (от properties, «свойства») — и в методе render() возвращает дерево представлений, которое нужно отобразить.

Метод render формирует описание интерфейса, который должен появиться на экране. React берёт это описание и строит итоговый результат. Точнее говоря, render возвращает React-элемент — «лёгкое» описание того, что следует отрендерить. Чтобы удобнее задавать структуру, большинство разработчиков используют синтаксис JSX. Во время сборки запись вида

 превращается в вызов React.createElement('div'). Иными словами, приведённый выше пример равнозначен следующему:

```
return React.createElement('div', {className: 'shopping-list'},
  React.createElement('h1', /* ... h1 children ... */),
  React.createElement('ul', /* ... ul children ... */)
);
```

JSX — это полноценный по возможностям синтаксис поверх JavaScript. Внутри него можно вставлять любые JS-выражения, заключив их в фигурные скобки. Каждый React-элемент — это обычный объект JavaScript, который можно присвоить переменной, передать в функции и использовать в коде как данные.

В примере выше компонент ShoppingList выводит только стандартные DOM-теги, такие как и . Но вы также можете объявлять свои собственные компоненты и рендерить их. Например, весь список покупок можно подключить одной записью - <ShoppingList/>. Компоненты в React изолированы и переиспользуемы, поэтому из небольших, простых элементов удобно собирать сложные интерфейсы.

Правило именования: с большой буквы (ShoppingList), иначе JSX воспримет как HTML-тег.

Virtual DOM, Diffing и обновление компонентов

Virtual DOM (виртуальный DOM) — это облегчённая копия реального DOM в памяти JavaScript. Это обычный JavaScript-объект, который описывает структуру интерфейса.

Когда вы пишете JSX, React создаёт дерево Virtual DOM объектов:

```
<div className="app">
  <h1>Заголовок</h1>
  <p>Текст</p>
</div>

// Превращается в структуру вида:
{
  type: 'div',
  props: { className: 'app' },
  children: [
    { type: 'h1', props: {}, children: ['Заголовок'] },
    { type: 'p', props: {}, children: ['Текст'] }
  ]
}
```

Зачем нужен Virtual DOM?

Virtual DOM — это один из подходов к обновлению интерфейса со своими преимуществами и недостатками:

Преимущества:

- Мультиплатформенность — один и тот же код может рендериться в разные платформы (веб, мобильные приложения, canvas)
- Батчинг и контроль — React может группировать обновления и управлять их приоритетом
- Предсказуемость — разработчику не нужно думать об оптимальных DOM-операциях

Недостатки:

- Overhead — дополнительная работа на создание и сравнение Virtual DOM
- Фреймворки без Virtual DOM (SolidJS, Svelte) могут быть быстрее

Diffing (сравнение)

Diffing — это процесс сравнения старого и нового Virtual DOM для определения минимального набора изменений. Пример работы:

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

  return (
    <div>
      <h1>Счётчик: {count}</h1>
      <button onClick={() => setCount(count + 1)}>+</button>
    </div>
  );
}
```

Что происходит при клике:

- Состояние меняется: count становится 1
- React создаёт новый Virtual DOM с обновлённым значением
- Diffing: React сравнивает старый и новый Virtual DOM
- Находит изменение: только текст внутри <h1> изменился
- Обновляет реальный DOM: меняет только текстовый узел, не перерисовывая всё

Когда обновляются компоненты

Компонент перерисовывается (вызывается заново) в трёх случаях:

1. Изменилось состояние (state)

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

  // Каждый вызов setCount перерисует компонент
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}
```

2. Изменились props

```
function Child({ value }) {
  return <div>{value}</div>;
}

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

  // При изменении count, Child получит новый prop и перерисуется
  return <Child value={count} />;
}
```

3. Перерисовался родительский компонент

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

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Увеличить</button>
      <Child /> {/* Перерисуется даже без props! */}
    </div>
  );
}

function Child() {
  console.log('Child отрендерился');
  return <div>Я дочерний компонент</div>;
}
```

Важно понимать: перерисовка компонента не означает изменение реального DOM. React сначала сравнивает Virtual DOM и обновляет реальный DOM только там, где есть реальные изменения.

Примечание: В случае перерисовки дочернего компонента при обновлении родителя можно избежать ненужных ре-рендеров с помощью `React.memo`. Это может улучшить производительность в сложных приложениях, но в большинстве случаев оптимизация не требуется.

Props (Пропсы)

Props — это способ передачи данных от родительского компонента к дочернему. Props доступны как параметр функции-компонента.

```
function Welcome(props) {

  return <h1>Привет, {props.name}!</h1>;
```

```
}


function App() {
  return (
    <div>
      <Welcome name="Анна" />
      <Welcome name="Иван" />
    </div>
  );
}
```

Можно использовать деструктуризацию:

```
function Welcome({ name, age }) {
  return (
    <div>
      <h1>Привет, {name}!</h1>
      <p>Возраст: {age}</p>
    </div>
  );
}
```

Важно: Props доступны только для чтения, их нельзя изменять внутри компонента.

Состояние (State)

Хук useState

Состояние (state) — это данные, которые могут изменяться со временем. Для работы с состоянием используется хук `useState`.

```
import { useState } from 'react';

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

  return (
    <div>
      <p>Счётчик: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Увеличить
      </button>
    </div>
  );
}
```

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

- `count` — текущее значение состояния
- `setCount` — функция для обновления состояния
- `0` — начальное значение

Множественные состояния

Можно использовать несколько `useState` в одном компоненте:

```
function Form() {
  const [name, setName] = useState('');
  const [age, setAge] = useState(0);
  const [email, setEmail] = useState('');

  return (
    <form>
      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Имя"
      />
      <input
        type="number"
        value={age}
        onChange={(e) => setAge(e.target.value)}
        placeholder="Возраст"
      />
      <input
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Email"
      />
    </form>
  );
}
```

Состояние с объектами и массивами

Для сложных состояний используйте объекты:

```
function UserProfile() {
  const [user, setUser] = useState({
    name: '',
    age: 0,
    email: ''
  });

  const updateName = (newName) => {
    setUser({ ...user, name: newName });
  };

  return (
    <input
      value={user.name}
      onChange={(e) => updateName(e.target.value)}
    />
  );
}
```

Для массивов:

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const addTodo = (text) => {
    setTodos([...todos, { id: Date.now(), text }]);
  };

  const removeTodo = (id) => {
    setTodos(todos.filter(todo => todo.id !== id));
  };

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          {todo.text}
          <button onClick={() => removeTodo(todo.id)}>Удалить</button>
        </li>
      ))}
    </ul>
  );
}
```

События

События в React называются так же, как в DOM, но в camelCase: `onClick`, `onChange`, `onSubmit`.

```
function Button() {
  const handleClick = () => {
    alert('Кнопка нажата!');
  };

  return <button onClick={handleClick}>Нажми меня</button>;
}
```

Передача параметров в обработчик:

```
function ItemList() {
  const handleClick = (id) => {
    console.log(`Элемент ${id} выбран`);
  };

  return (
    <div>
      <button onClick={() => handleClick(1)}>Элемент 1</button>
      <button onClick={() => handleClick(2)}>Элемент 2</button>
    </div>
  );
}
```

Условный рендеринг

Показывайте разный контент в зависимости от условий:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
```

```
    return <h1>Добро пожаловать!</h1>;
  }
  return <h1>Пожалуйста, войдите</h1>;
}
```

С помощью тернарного оператора:

```
function Status({ isOnline }) {
  return (
    <div>
      Статус: {isOnline ? 'В сети' : 'Не в сети'}
    </div>
  );
}
```

С помощью `&&`:

```
function Notification({ hasNewMessages, count }) {
  return (
    <div>
      {hasNewMessages && <p>У вас {count} новых сообщений</p>}
    </div>
  );
}
```

Списки и ключи

Для отображения списков используйте метод `.map()`:

```
function UserList() {
  const users = [
    { id: 1, name: 'Анна' },
    { id: 2, name: 'Иван' },
    { id: 3, name: 'Мария' }
  ];

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Зачем нужны ключи

Ключи (keys) — это специальный атрибут, который помогает React идентифицировать элементы списка и эффективно обновлять интерфейс.

Как React использует ключи:

Когда список изменяется (добавляется, удаляется или переупорядочивается элемент), React использует ключи, чтобы понять:

- Какие элементы остались теми же

- Какие элементы новые
- Какие элементы удалены

Это позволяет React **переиспользовать существующие DOM-узлы** вместо полного пересоздания списка.

Контролируемые компоненты

Контролируемый компонент — это элемент ввода (input, textarea, select), значение которого управляется состоянием React.

Принцип работы

В обычном HTML элементы формы хранят своё состояние внутри себя:

```
<!-- Браузер сам управляет значением -->
<input type="text" />
```

В React мы делаем состояние **единым источником правды**:

```
function Input() {
  const [value, setValue] = useState('');

  return (
    <input
      value={value} // React контролирует значение
      onChange={(e) => setValue(e.target.value)} // Обновляем состояние
    />
  );
}
```

Поток данных:

1. Пользователь вводит символ
2. Срабатывает `onChange`
3. Вызывается `setValue()` с новым значением
4. Компонент перерисовывается
5. Input получает новое значение из состояния

Зачем нужны контролируемые компоненты

Валидация на лету:

```
function PhoneInput() {
  const [phone, setPhone] = useState('');

  const handleChange = (e) => {
    const value = e.target.value;
    // Разрешаем только цифры
    if (/^\d*$/.test(value)) {
      setPhone(value);
    }
  };
```

```
  return <input value={phone} onChange={handleChange} />;
}
```

Форматирование:

```
function UpperCaseInput() {
  const [text, setText] = useState('');

  return (
    <input
      value={text}
      onChange={(e) => setText(e.target.value.toUpperCase())}
    />
  );
}
```

Синхронизация нескольких элементов:

```
function SyncedInputs() {
  const [value, setValue] = useState('');

  return (
    <div>
      <input value={value} onChange={(e) => setValue(e.target.value)} />
      <input value={value} onChange={(e) => setValue(e.target.value)} />
      <p>Текущее значение: {value}</p>
    </div>
  );
}
```

Контролируемые компоненты вне форм

Контролируемые компоненты полезны не только для форм с отправкой данных:

```
function SearchFilter() {
  const [search, setSearch] = useState('');
  const [items] = useState(['Яблоко', 'Банан', 'Апельсин']);

  const filtered = items.filter(item =>
    item.toLowerCase().includes(search.toLowerCase())
  );

  return (
    <div>
      <input
        value={search}
        onChange={(e) => setSearch(e.target.value)}
        placeholder="Поиск..."
      />
      <ul>
        {filtered.map(item => <li key={item}>{item}</li>)}
      </ul>
    </div>
  );
}
```

Работа с формами

Для форм с отправкой данных контролируемые компоненты позволяют легко собрать все значения:

```
function LoginForm() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault(); // Предотвращаем перезагрузку страницы
    console.log('Данные:', { username, password });
    // Здесь можно отправить данные на сервер
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
        placeholder="Логин"
      />
      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        placeholder="Пароль"
      />
      <button type="submit">Войти</button>
    </form>
  );
}
```

Важно: Элемент `<form>` здесь нужен для:

- Семантики HTML
- Отправки по Enter
- Доступности (accessibility)
- Встроенной HTML5 валидации

Но все данные всё равно управляются через состояние React, а не через встроенное поведение формы.

Работа с разными типами полей

```
function FormExample() {
  const [formData, setFormData] = useState({
    text: '',
    textarea: '',
    select: 'option1',
    checkbox: false,
    radio: 'option1'
  });

  const handleChange = (e) => {
    const { name, value, type, checked } = e.target;
    setFormData({
      ...formData,
      [name]: type === 'checkbox' ? checked : value
```

```
    });
  };

  return (
    <form>
      <input
        type="text"
        name="text"
        value={formData.text}
        onChange={handleChange}
      />

      <textarea
        name="textarea"
        value={formData.textarea}
        onChange={handleChange}
      />

      <select
        name="select"
        value={formData.select}
        onChange={handleChange}
      >
        <option value="option1">Вариант 1</option>
        <option value="option2">Вариант 2</option>
      </select>

      <label>
        <input
          type="checkbox"
          name="checkbox"
          checked={formData.checkbox}
          onChange={handleChange}
        />
        Согласен
      </label>

      <label>
        <input
          type="radio"
          name="radio"
          value="option1"
          checked={formData.radio === 'option1'}
          onChange={handleChange}
        />
        Опция 1
      </label>
    </form>
  );
}
```

Неконтролируемые компоненты

Иногда можно использовать **неконтролируемые компоненты** — элементы, которые управляют своим состоянием сами, а не через React-состояние.

Основное отличие от контролируемых компонентов:

- **Контролируемые**: значение хранится в состоянии React, React — единственный источник правды

- **Неконтролируемые**: значение хранится в самом DOM-элементе, доступ к нему получаем через ref

Хук useEffect

useEffect используется для выполнения побочных эффектов: запросов к API, подписок, таймеров, работы с DOM.

Важное предупреждение: `useEffect` — это мощный, но опасный инструмент. Он является источником многих багов и проблем с производительностью. Используйте его очень аккуратно и только когда это действительно необходимо. Во многих случаях можно обойтись без него, используя обработчики событий или вычисляемые значения.

```
import { useState, useEffect } from 'react';

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    // Этот код выполнится после первого полного рендера компонента
    const interval = setInterval(() => {
      setSeconds(s => s + 1);
    }, 1000);

    // Функция очистки
    return () => clearInterval(interval);
  }, []); // Пустой массив = выполнится один раз

  return <div>Прошло секунд: {seconds}</div>;
}
```

Важно понимать: `useEffect` **первый раз исполняется после полного первого рендера** компонента, а не во время рендера. То есть сначала компонент отрисовывается на экране, а затем запускается эффект.

- **Используйте для побочных эффектов**: API, подписки, таймеры, работа с DOM
- **Всегда очищайте ресурсы**: таймеры, слушатели, соединения
- **Правильно указывайте зависимости**: все значения из компонента, которые используются в эффекте
- **Разделяйте логику**: используйте несколько useEffect для разных задач
- **Функция очистки вызывается**:
    - Перед каждым следующим запуском эффекта
    - При размонтировании компонента

useMemo

useMemo кэширует результат вычислений и пересчитывает его только при изменении зависимостей.

Зачем нужен: Оптимизация производительности — избегает тяжёлых вычислений при каждом рендере.

```
function ProductList({ products, filter }) {
  // Без useMemo фильтрация выполнялась бы при каждом рендере
  const filteredProducts = useMemo(() => {
    console.log('Фильтрация выполнена');
    return products.filter(p => p.category === filter);
  }, [products, filter]); // Пересчитывается только при изменении products
или filter


  return (
    <ul>
      {filteredProducts.map(p => (
        <li key={p.id}>{p.name}</li>
      ))}
    </ul>
  );
}
```

Когда использовать:

- Тяжёлые вычисления (сортировка, фильтрация больших массивов)
- Предотвращение лишних ре-рендеров дочерних компонентов, обёрнутых в React.memo

Когда НЕ использовать:

- Простые вычисления (сложение, конкатенация строк)
- Преждевременная оптимизация

Правила React

React имеет несколько строгих правил, которые необходимо соблюдать. Нарушение этих правил приводит к багам и непредсказуемому поведению.

Правила компонентов

1. Компоненты должны быть чистыми функциями

Компонент должен возвращать один и тот же результат при одних и тех же входных данных (props, state, context). Не должен изменять объекты или переменные, существовавшие до рендера.

Примечание: Это правило регулярно нарушается опытными разработчиками для обхода Virtual DOM и достижения более высокой производительности в специфических случаях. Но для начинающих важно соблюдать это правило.

2. Не вызывайте побочные эффекты во время рендера

Побочные эффекты (запросы к API, изменение DOM, установка таймеров) должны быть в обработчиках событий или в `useEffect`, но не в теле компонента.

Правила хуков

1. Вызывайте хуки только на верхнем уровне

Нельзя вызывать хуки внутри условий, циклов или вложенных функций. Хуки должны быть на верхнем уровне функции компонента.

Почему это важно: React запоминает порядок вызова хуков. Если порядок меняется между рендерами, состояние перепутается.

2. Вызывайте хуки только из React-функций

Хуки можно вызывать только в:

- Функциональных компонентах React
- Пользовательских хуках (функции, имя которых начинается с use)

Правила состояния

1. Считайте состояние неизменяемым (immutable)

Никогда не изменяйте объекты и массивы в состоянии напрямую. Всегда создавайте новые копии.

2. Обновления состояния асинхронны

React может группировать несколько обновлений состояния. Если новое значение зависит от предыдущего, используйте функциональное обновление.

Полезные ресурсы

- Официальная документация: https://react.dev
- React Beta Docs (новая документация): https://react.dev/learn

Заключение

Это базовые концепции React, которых достаточно для создания несложных интерактивных сайтов. Практикуйтесь, создавайте небольшие проекты, и постепенно переходите к более сложным темам: Context API, custom hooks, React Router.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Скачать шаблонный файл запуска React без сборщика (ФАЙЛ) и выполнить следующие задания.

Задание 1. Счётчик «кликер» со смарт-логикой

Цель: отработать useState, обработчики событий и условное отображение.

Что сделать:

- Кнопки +1, −1, Reset для значения count.
- −1 не должен опускать счётчик ниже 0 (кнопка блокируется при count === 0).
- Отображать «Молодец!» когда count кратен 10.
- (Дополнительно) Поддержать горячие клавиши: ArrowUp = +1, ArrowDown = −1.

Критерии проверки:

- Состояние меняется мгновенно, кнопки корректно блокируются, сообщение о кратности показывается только при count % 10 === 0 и count > 0.

Подсказки: useState, обработчики onClick, disabled={...}, тернарный оператор для условного текста.

Задание 2. Todo-список с фильтрами и локальным хранением

Цель: контролируемые поля ввода, списки, ключи, эффекты, localStorage.

Что сделать:

- Поле ввода «Новая задача» и кнопка «Добавить». Enter тоже добавляет.
- Список задач: чекбокс «выполнено», текст, кнопка удаления.
- Фильтры: **Все / Активные / Выполненные**.
- Счётчик: «Всего: N | Выполнено: M».
- Данные должны сохраняться между перезагрузками через localStorage.

Критерии проверки:

- После перезагрузки задачи сохраняются, фильтры работают, элементы списка имеют стабильные ключи.

Подсказки: useEffect для синхронизации с localStorage, JSON.parse/stringify, event.key === 'Enter'.

Задание 3. Мини-квиз (тест) с результатом

Цель: многосоставное состояние, вычисляемые значения, условный рендеринг.

Что сделать:

- Набор из 5 вопросов (варианты — радиокнопки). Можно взять вопросы из примера ниже или придумать свои.
- Кнопка **Далее** активна только если выбран ответ.
- Прогресс: «Вопрос X / 5», прогресс-бар заполняется.
- Финальный экран: «Ваш результат: Y/5» и кнопка «Сыграть ещё раз».
- (Дополнительно) Таймер на каждый вопрос (например, 20 сек) — при истечении времени автоматически переходить далее.

Критерии проверки:

- Нельзя «перепрыгнуть» без ответа, подсчёт баллов корректный, перезапуск сбрасывает состояние.

Подсказки: массив вопросов, индексация текущего, выбранный вариант, вычисление результата в конце.

Контрольные вопросы

1. Что такое компонент, состояние и пропс?
2. Что такое JSX?
3. Как REACT разворачивает функцию `render()`?
4. В каких случаях обновляются компоненты?
5. Какие хуки были использованы вами в лабораторной работе?
6. Назовите основные правила работы с REACT.