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-1-19

Практическое занятие 19: Базовые элементы работы с React.js и JSX

В рамках данного занятия рассмотим процесс установки, использования и запуска приложений на React, а также основы JSX.

Подробная теоретическая информация содержится в лекции, а также можно ознакомиться с официальной документацией по React:

https://react.dev/learn

Запуск проектов на React

Перед началом практической работы важно понять базовые принципы React. React — это библиотека для создания пользовательских интерфейсов, которая использует компонентный подход. Каждый компонент представляет собой независимую часть интерфейса со своей логикой и внешним видом.

Основной единицей в React является компонент, который может быть как функцией, так и классом. Мы будем использовать функциональные компоненты, так как это современный и рекомендуемый подход. JSX — это синтаксическое расширение для JavaScript, которое позволяет писать HTML-подобный код внутри JavaScript. Важно помнить, что JSX не является ни строкой, ни HTML, а компилируется в вызовы React.createElement.

Для работы с React потребуется настроенная среда разработки. Мы будем использовать Vite — современный инструмент для быстрой разработки, который предоставляет оптимизированную сборку и сервер разработки.

Для его использования необходимо скачать Node.js со встроенным пакетным менеджером npm, который мы будем использовать далее.

Ссылка для Node.js:

https://nodejs.org/en/download

После установки выполним следующие команды.

1. Проверим версию npm и корректную установку Node.js:

   ```
   npm --v
   ```

   В случае возникновения ошибок необходимо скопировать вывод консоли в поисковик и/или нейросеть для анализа и устранения ошибок.

2. Выполните команду создания проекта:

   ```
   npm create vite@latest
   ```

   Вам будет предложено ввести название проекта (например, test-app) и выбрать шаблон (выберите React, затем JavaScript или TypeScript по желанию). После создания проекта перейдите в папку проекта.

3. Установите зависимости и запустите приложение:

   ```
   cd test-app
   npm install
   npm run dev
   ```

   Просмотрите результат. Именно по данному результату мы с вами и будем выполнять сегодняшние примеры.

Что происходит после создания проекта?

После выполнения команд выше в папке с проектом появится следующая структура:

- `public/` — содержит статические файлы, например `index.html` и иконки. Эти файлы не обрабатываются Vite, но доступны приложению.
- `src/` — основная папка для вашего кода. Здесь находятся компоненты, стили и файлы приложения.
- `package.json` — файл с информацией о проекте и зависимостях. Через него управляются библиотеки и скрипты запуска.
- `vite.config.js` или `vite.config.ts` — конфигурация сборщика Vite.
- `node_modules/` — автоматически созданная папка с установленными зависимостями (не редактируется вручную).

Внутри `src/` вы увидите такие файлы, как:

- `App.jsx` или `App.js` — главный компонент приложения.
- `main.jsx` или `main.js` — точка входа, где приложение подключается к странице.

- Файлы стилей (`App.css`, и др.).

Что увидите при запуске приложения?

После запуска (`npm run dev`) в браузере откроется стартовая страница React/Vite. Обычно она содержит логотип React, приветственный текст и ссылку на документацию. Это означает, что всё установлено корректно, и вы готовы приступать к практике.

Теперь, когда среда готова и структура проекта понятна, переходим к теоретической части.

Теоретическая часть

Пример 1. Первый компонент: приветствие пользователя

Постановка задачи

Начнём с создания самого простого компонента в React. Представим, что нам нужно приветствовать пользователя на странице. Но не просто написать "Привет!", а сделать приветствие динамическим — чтобы оно менялось в зависимости от времени суток: утром показывать "Доброе утро", днём — "Добрый день", вечером — "Добрый вечер".

Ключевая идея: создадим функциональный компонент, который будет содержать логику определения времени суток и возвращать JSX с приветствием.

Что такое компонент в React?

Компонент — это независимая, переиспользуемая часть интерфейса. В React компонент — это обычная JavaScript-функция, которая возвращает JSX (разметку, похожую на HTML).

Простейший пример компонента:

```
function Welcome() {
  return <h1>Привет, мир!</h1>;
}
```

Компоненты можно использовать как HTML-теги: `<Welcome />`.

Что такое JSX?

JSX (JavaScript XML) — это синтаксическое расширение JavaScript, которое позволяет писать HTML-подобную разметку прямо в коде JavaScript. JSX не является ни строкой, ни HTML — это специальный синтаксис, который компилируется в вызовы функций React.

Ключевые особенности JSX:

- Выражения JavaScript вставляются в фигурных скобках: `{userName}`
- Вместо атрибута `class` используется `className` (так как `class` — зарезервированное слово в JavaScript)
- Компонент должен возвращать один корневой элемент

- Комментарии пишутся внутри `{/* ... */}`

Небольшое отступление: классовые vs функциональные компоненты

Раньше в React часто использовали классовые компоненты. Для полноты картины посмотрим короткий пример:

```
// src/GreetingClass.jsx
import { Component } from 'react';

class GreetingClass extends Component {
  render() {
    const userName = 'Айнура';
    const currentHour = new Date().getHours();
    const timeOfDay = currentHour < 12 ? 'Доброе утро' : currentHour < 18 ?
'Добрый день' : 'Добрый вечер';

    return (
      <div className="greeting">
        <h1>{timeOfDay}, {userName}!</h1>
        <p>Рады видеть вас в нашем приложении.</p>
      </div>
    );
  }
}

export default GreetingClass;
```

Почему мы выбираем функциональные компоненты:

- Проще синтаксис и меньше шаблонного кода
- Хуки (`useState`, `useEffect`) дают доступ к состоянию и эффектам без сложного жизненного цикла
- Легче читать, тестировать и переиспользовать логику (кастомные хуки)
- Рекомендовано командой React; новые возможности приходят в первую очередь в функциональный API

Далее в этой работе мы будем использовать только функциональные компоненты.

Создадим компонент приветствия

Шаг 0. Очищаем проект от лишних файлов (необязательно, но рекомендуется)

После создания проекта с помощью Vite в папке `src` появляются демонстрационные файлы. Для учебных целей и лучшего понимания структуры проекта рекомендуется удалить ненужные файлы и начать с чистого листа.

Что можно удалить из папки `src`:

1. `index.css` — глобальные стили демо-приложения. Можем удалить, если хотим начать со своих стилей.
2. `logo.svg` или `assets/react.svg` — логотипы React, используемые в демо. Не нужны для нашего примера.

Что НЕ удаляем:

1. `main.jsx` (или `main.js`) — точка входа приложения. Этот файл подключает React к странице. **Обязательно оставляем!**
2. `App.jsx` (или `App.js`) — главный компонент приложения. Оставляем, но очистим его содержимое.
3. `App.css` — файл стилей. Оставляем, но очистим и добавим свои стили.

Что делаем с `App.jsx`:

Откроем файл `src/App.jsx` и заменим его содержимое на минимальную версию:

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

function App() {
  return (
    <div className="App">
      <h1>Моё React приложение</h1>
    </div>
  );
}

export default App;
```

Что делаем с `App.css`:

Откроем файл `src/App.css` и заменим его содержимое на базовые стили для наших примеров:

```css
/* src/App.css */

/* базовые стили для приложения */
.App {
  max-width: 1200px;
  margin: 0 auto;
  padding: 20px;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
}

/* стили для заголовка приложения (если используется) */
.App-header {
  background-color: #282c34;
  padding: 20px;
```

```
  color: white;
  text-align: center;
  border-radius: 10px;
  margin-bottom: 20px;
}

.App-logo {
  height: 40px;
  pointer-events: none;
}

.App-link {
  color: #61dafb;
  text-decoration: none;
}

.App-link:hover {
  text-decoration: underline;
}

/* стили для компонента Greeting */
.greeting {
  background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
  color: white;
  padding: 30px;
  border-radius: 10px;
  margin: 20px 0;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}

.greeting h1 {
  margin: 0 0 10px 0;
  font-size: 2em;
}

.greeting p {
  margin: 0;
  font-size: 1.1em;
  opacity: 0.9;
}

/* стили для компонента UserCard */
.user-card {
  display: flex;
  gap: 20px;
  background: white;
  border: 2px solid #e0e0e0;
  border-radius: 10px;
  padding: 20px;
  margin: 20px 0;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  transition: transform 0.2s, box-shadow 0.2s;
}
```

```css
.user-card:hover {
  transform: translateY(-2px);
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.15);
}

.avatar-section {
  text-align: center;
}

.avatar-section img {
  width: 100px;
  height: 100px;
  border-radius: 50%;
  object-fit: cover;
  border: 3px solid #667eea;
}

.avatar-section p {
  margin-top: 10px;
  font-size: 0.9em;
  color: #666;
}

.user-info h3 {
  margin: 0 0 10px 0;
  color: #333;
}

.user-info p {
  margin: 0;
  color: #666;
}

/* стили для компонента TaskList */
.task-list {
  background: white;
  border-radius: 10px;
  padding: 20px;
  margin: 20px 0;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

.task-list h2 {
  margin: 0 0 20px 0;
  color: #333;
}

.task-list ul {
  list-style: none;
  padding: 0;
  margin: 0;
}

.task-list li {
```

```
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 15px;
  margin: 10px 0;
  border-radius: 5px;
  transition: background-color 0.2s;
}

.task-list li.completed {
  background-color: #e8f5e9;
  border-left: 4px solid #4caf50;
}

.task-list li.pending {
  background-color: #fff3e0;
  border-left: 4px solid #ff9800;
}

.task-list li:hover {
  opacity: 0.8;
}

.task-list span {
  flex: 1;
  color: #333;
}
```

Почему это полезно:

- **Чистая структура:** проект содержит только то, что нам действительно нужно
- **Лучше понимание:** не отвлекаемся на демонстрационный код
- **Проще ориентироваться:** меньше файлов — проще найти нужное
- **Учебная цель:** создаём всё сами, понимая каждый шаг

После очистки структура папки `src` будет выглядеть примерно так:

```
src/
├── main.jsx      (точка входа - не трогаем)
└── App.jsx       (главный компонент - очистили)
```

Теперь можем приступать к созданию своих компонентов!

Шаг 1. Создаём файл компонента

В папке `src` создадим новый файл `Greeting.jsx`. Этот файл будет содержать наш компонент приветствия.

Шаг 2. Пишем простейший компонент

Начнём с самой базовой версии — функции, которая возвращает JSX:

```
// src/Greeting.jsx

function Greeting() {
  return (
    <div className="greeting">
      <h1>Привет!</h1>
      <p>Рады видеть вас в нашем приложении.</p>
    </div>
  );
}

export default Greeting;
```

Пояснение:

- `function Greeting()` — объявляем функцию-компонент
- `return (...)` — возвращаем JSX-разметку
- `<div className="greeting">` — корневой элемент (обязательно должен быть один)
- `export default Greeting` — экспортируем компонент, чтобы его можно было импортировать в других файлах

Шаг 3. Добавляем переменные и динамическое содержимое

Теперь добавим имя пользователя и выведем его в приветствии:

```
// src/Greeting.jsx

function Greeting() {
  // объявляем переменную с именем пользователя
  const userName = 'Айнура'; // можно заменить на своё имя

  return (
    <div className="greeting">
      {/* вставляем переменную в JSX с помощью фигурных скобок */}
      <h1>Привет, {userName}!</h1>
      <p>Рады видеть вас в нашем приложении.</p>
    </div>
  );
}

export default Greeting;
```

Пояснение:

- `const userName = 'Айнура'` — объявляем константу с именем
- `{userName}` — вставляем значение переменной в JSX с помощью фигурных скобок
- Внутри `{}` можно использовать любые JavaScript-выражения

Шаг 4. Добавляем логику определения времени суток

Теперь сделаем приветствие зависимым от текущего времени:

```
// src/Greeting.jsx

function Greeting() {
  const userName = 'Айнура'; // можно заменить на своё имя

  // получаем текущий час (0..23)
  const currentHour = new Date().getHours();

  // определяем приветствие в зависимости от времени
  let timeOfDay; // переменная для хранения приветствия
  if (currentHour < 12) {
    timeOfDay = 'Доброе утро';
  } else if (currentHour < 18) {
    timeOfDay = 'Добрый день';
  } else {
    timeOfDay = 'Добрый вечер';
  }

  return (
    <div className="greeting">
      {/* вставляем динамическое приветствие и имя */}
      <h1>{timeOfDay}, {userName}!</h1>
      <p>Рады видеть вас в нашем приложении.</p>
    </div>
  );
}

export default Greeting;
```

Пояснение:

- `new Date().getHours()` — получаем текущий час (число от 0 до 23)
- Используем условные операторы `if/else` для определения времени суток
- Если час меньше 12 — утро, меньше 18 — день, иначе — вечер
- `{timeOfDay}` и `{userName}` — вставляем обе переменные в приветствие

Шаг 5. Подключаем компонент в App.jsx

Теперь используем наш компонент в главном файле приложения. Откроем `src/App.jsx` и добавим импорт и использование `Greeting`:

```
// src/App.jsx
import './App.css';
import Greeting from './Greeting'; // импортируем наш компонент

function App() {
```

```
  return (
    <div className="App">
      {/* используем компонент как HTML-тег */}
      <Greeting />
    </div>
  );
}

export default App;
```

Пояснение:

- `import Greeting from './Greeting'` — импортируем компонент из файла `Greeting.jsx`
- `<Greeting />` — используем компонент как самозакрывающийся тег
- Компонент отобразится в том месте, где мы его разместили

Шаг 6. Проверяем результат

Сохраним все файлы и посмотрим на результат в браузере. Должно отобразиться приветствие, соответствующее текущему времени суток, например: "Добрый день, Айнура!".

Важные моменты о компонентах и JSX

1. **Компонент — это функция:** любая функция, возвращающая JSX, является компонентом
2. **Имя компонента с большой буквы:** `Greeting`, а не `greeting` — это важное соглашение React
3. **Один корневой элемент:** компонент должен возвращать один корневой элемент (можно использовать `<div>` или фрагмент `<>...</>`)
4. **Фигурные скобки для выражений:** внутри JSX используйте `{}` для вставки JavaScript-выражений
5. **className вместо class:** в JSX используется `className` для CSS-классов

Частые ошибки и как их избежать

- **Забыли `export default`:** без экспорта компонент нельзя импортировать в других файлах. Всегда добавляйте `export default ComponentName` в конце файла.
- **Неверный путь при импорте:** если файл находится в той же папке, используйте `'./Greeting'`, а не `'Greeting'`.
- **Использование `class` вместо `className`:** в JSX это вызовет ошибку. Всегда используйте `className`.
- **Несколько корневых элементов:** компонент должен возвращать один корневой элемент. Оберните содержимое в `<div>` или используйте фрагмент `<>...</>`.
- **Забыли фигурные скобки:** для вставки переменных используйте `{userName}`, а не просто `userName`.

Полный код примера

Файл `src/Greeting.jsx`:

```jsx
// src/Greeting.jsx

function Greeting() {
  const userName = 'Айнура'; // можно заменить на своё имя

  // получаем текущий час (0..23)
  const currentHour = new Date().getHours();

  // определяем приветствие в зависимости от времени
  let timeOfDay; // переменная для хранения приветствия
  if (currentHour < 12) {
    timeOfDay = 'Доброе утро';
  } else if (currentHour < 18) {
    timeOfDay = 'Добрый день';
  } else {
    timeOfDay = 'Добрый вечер';
  }

  return (
    <div className="greeting">
      {/* вставляем динамическое приветствие и имя */}
      <h1>{timeOfDay}, {userName}!</h1>
      <p>Рады видеть вас в нашем приложении.</p>
    </div>
  );
}

export default Greeting;
```

Файл `src/App.jsx`:

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

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

export default App;
```

Пример 2. Компонент со свойствами (props)

Постановка задачи

Представим ситуацию: нам нужно отобразить на странице несколько карточек пользователей. У каждого пользователя есть имя, роль, аватар и статус (онлайн/офлайн). Если создавать отдельный компонент для каждого пользователя, код будет дублироваться. Вместо этого создадим один универсальный компонент `UserCard`, который сможет отображать информацию о любом пользователе.

Ключевая идея: используем механизм **props** (свойства) — это способ передачи данных в компонент извне. Благодаря props компонент становится переиспользуемым и гибким.

Что такое props?

Props — это объект, который React передаёт в компонент при его вызове. Через props родительский компонент может передать дочернему любые данные: строки, числа, булевы значения, массивы, объекты и даже функции.

Пример:

```
<UserCard name="Иван" role="Разработчик" />
```

Здесь `name` и `role` — это props, которые передаются в компонент `UserCard`.

Создадим компонент

Шаг 1. Создаём файл компонента

В папке `src` создадим новый файл `UserCard.jsx`. Этот файл будет содержать наш компонент карточки пользователя.

Шаг 2. Пишем базовую структуру компонента

Начнём с простой функции, которая принимает props и возвращает JSX:

```
// src/UserCard.jsx

// Компонент принимает объект props с нужными свойствами
function UserCard(props) {
  return (
    <div className="user-card">
      <h3>{props.name}</h3>
      <p>{props.role}</p>
    </div>
  );
}

export default UserCard;
```

Пояснение:

- `props` — это объект, содержащий все переданные свойства

- `props.name` и `props.role` — обращаемся к конкретным свойствам через точку
- Фигурные скобки `{}` позволяют вставить JavaScript-выражение в JSX

Шаг 3. Используем деструктуризацию для удобства

Вместо того чтобы каждый раз писать `props.name`, `props.role` и т.д., применим деструктуризацию прямо в параметрах функции:

```
// src/UserCard.jsx

// Деструктурируем props сразу в параметрах функции
function UserCard({ name, role, avatarUrl, isOnline }) {
  return (
    <div className="user-card">
      <h3>{name}</h3>
      <p>{role}</p>
    </div>
  );
}

export default UserCard;
```

Пояснение:

- `{ name, role, avatarUrl, isOnline }` — извлекаем нужные свойства из объекта props
- Теперь можем использовать `name` вместо `props.name` — код становится чище

Шаг 4. Добавляем аватар и статус

Расширим компонент, добавив изображение аватара и отображение статуса пользователя:

```
// src/UserCard.jsx

function UserCard({ name, role, avatarUrl, isOnline }) {
  return (
    <div className="user-card">
      {/* Секция с аватаром */}
      <div className="avatar-section">
        {/* Отображаем изображение аватара */}
        <img src={avatarUrl} alt={`Аватар ${name}`} />

        {/* Условный рендеринг: если isOnline === true, показываем 'online', иначе
'offline' */}
        <p>Статус: {isOnline ? 'online' : 'offline'}</p>
      </div>

      {/* Секция с информацией о пользователе */}
      <div className="user-info">
        <h3>{name}</h3>
        <p>{role}</p>
```

```
      </div>
    </div>
  );
}

export default UserCard;
```

Пояснение:

- `src={avatarUrl}` — передаём URL изображения в атрибут src
- `alt={Аватар ${name}}` — используем шаблонную строку для динамического текста
- `{isOnline ? 'online' : 'offline'}` — тернарный оператор для условного отображения
- Комментарии в JSX пишутся внутри `{/* ... */}`

Шаг 5. Подключаем компонент в App.jsx

Теперь используем наш компонент в главном файле приложения. Откроем `src/App.jsx` и добавим импорт и использование `UserCard`:

```
// src/App.jsx
import './App.css';
import Greeting from './Greeting'; // компонент из Примера 1
import UserCard from './UserCard'; // наш новый компонент

function App() {
  return (
    <div className="App">
      {/* Компонент из предыдущего примера */}
      <Greeting />

      {/* Используем UserCard и передаём ему props */}
      <UserCard
        name="Иван Иванов"
        role="Администратор"
        avatarUrl="https://encrypted-tbn0.gstatic.com/images?
q=tbn:ANd9GcRfVMhpKmVy_-iwfRLAiNiaDslMa-2oEz7KTw&s"
        isOnline={true}
      />
    </div>
  );
}

export default App;
```

Пояснение:

- Импортируем `UserCard` из файла `./UserCard`
- При использовании компонента передаём все необходимые props как атрибуты
- Строковые значения (`name`, `role`, `avatarUrl`) передаём в кавычках
- Булевы значения и другие типы данных передаём в фигурных скобках: `{true}`

Шаг 6. Проверяем результат

Сохраним все файлы и посмотрим на результат в браузере. Должна отобразиться карточка пользователя с аватаром, именем, ролью и статусом.

Важные моменты о props

1. **Props передаются сверху вниз:** родительский компонент передаёт данные дочернему
2. **Props доступны только для чтения:** внутри компонента нельзя изменить значения props
3. **Props могут быть любого типа:** строки, числа, массивы, объекты, функции
4. **Можно задать значения по умолчанию:** если props не передан, можно использовать значение по умолчанию

Частые ошибки и как их избежать

- **Забыли передать обязательный prop:** компонент может не отобразиться корректно. Всегда проверяйте, что передали все необходимые данные.
- **Неправильный синтаксис передачи props:** строки передаём в кавычках `name="Иван"`, остальные типы — в фигурных скобках `isOnline={true}`.
- **Попытка изменить props внутри компонента:** props неизменяемы. Для изменяемых данных используйте state (изучим позже).

Полный код примера

Файл `src/UserCard.jsx`:

```
// src/UserCard.jsx

function UserCard({ name, role, avatarUrl, isOnline }) {
  return (
    <div className="user-card">
      {/* Секция с аватаром */}
      <div className="avatar-section">
        {/* Отображаем изображение аватара */}
        <img src={avatarUrl} alt={`Аватар ${name}`} />

        {/* Условный рендеринг: если isOnline === true, показываем 'online', иначе
'offline' */}
        <p>Статус: {isOnline ? 'online' : 'offline'}</p>
      </div>

      {/* Секция с информацией о пользователе */}
      <div className="user-info">
        <h3>{name}</h3>
        <p>{role}</p>
      </div>
    </div>
  );
}
```

```
export default UserCard;
```

Файл `src/App.jsx`:

```
// src/App.jsx
import './App.css';
import Greeting from './Greeting';
import UserCard from './UserCard';

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

      <UserCard
        name="Иван Иванов"
        role="Администратор"
        avatarUrl="https://encrypted-tbn0.gstatic.com/images?
q=tbn:ANd9GcRfVMhpKmVy_-iwfRLAiNiaDslMa-2oEz7KTw&s"
        isOnline={true}
      />
    </div>
  );
}

export default App;
```

Пример 3. Работа со списками и ключами

Постановка задачи

Представим, что нам нужно отобразить список задач на странице. Задач может быть много, и каждая имеет свои данные: название, статус выполнения. Создавать отдельный элемент для каждой задачи вручную неудобно и непрактично. Вместо этого создадим массив с данными задач и автоматически преобразуем его в список JSX-элементов.

Ключевая идея: используем метод массива `map()` для преобразования данных в JSX-элементы. При этом каждому элементу списка назначим уникальный ключ `key`, чтобы React мог эффективно отслеживать изменения.

Что такое метод map() и зачем нужны ключи?

Метод map() — это метод массива в JavaScript, который создаёт новый массив, применяя функцию к каждому элементу исходного массива.

Пример:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2); // [2, 4, 6]
```

В React мы используем `map()` для преобразования массива данных в массив JSX-элементов.

Атрибут key — это специальный атрибут, который помогает React идентифицировать элементы списка. Когда список изменяется (добавляются, удаляются или переупорядочиваются элементы), React использует ключи, чтобы понять, что именно изменилось, и обновить только необходимые части интерфейса.

Как работает key и почему он важен?

Представим ситуацию: у нас есть список из трёх задач. Пользователь добавляет новую задачу в начало списка. Как React поймёт, что это новая задача, а не изменение существующих?

Без key: React сравнивает элементы по позиции в массиве. Он думает, что первый элемент изменился, второй изменился, третий изменился, и появился четвёртый. Результат — React перерисует все элементы, даже те, что не изменились. Это неэффективно и может привести к потере состояния компонентов.

С key: React сравнивает элементы по их уникальным ключам. Он видит, что элементы с ключами `1`, `2`, `3` остались на месте, а элемент с ключом `4` — новый. Результат — React добавит только один новый элемент, не трогая остальные. Это быстро и правильно.

Какие значения использовать для key?

Для key правильно использовать уникальный ID из данных:

```
const tasks = [
  { id: 1, title: 'Задача 1' },
  { id: 2, title: 'Задача 2' },
  { id: 3, title: 'Задача 3' }
];

// правильно: используем id из объекта
{tasks.map(task => (
  <li key={task.id}>{task.title}</li>
))}
```

Почему это правильно:

- `id` уникален для каждого элемента
- `id` стабилен — не меняется при сортировке или фильтрации
- React всегда сможет правильно идентифицировать элемент

❌ **Неправильно — использовать индекс массива:**

```
// плохая практика: использование индекса
{tasks.map((task, index) => (
  <li key={index}>{task.title}</li>
))}
```

Почему это плохо:

- Индекс меняется при изменении порядка элементов
- При удалении элемента индексы всех последующих элементов сдвигаются
- React может перепутать элементы и неправильно обновить интерфейс

Пример проблемы с индексом:

```
// исходный список (key = индекс)
[
  { key: 0, text: 'Яблоко' },
  { key: 1, text: 'Банан' },
  { key: 2, text: 'Апельсин' }
]

// удаляем "Банан"
[
  { key: 0, text: 'Яблоко' },    // key не изменился
  { key: 1, text: 'Апельсин' }   // key изменился с 2 на 1!
]

// React думает, что "Банан" превратился в "Апельсин"
// и удалился последний элемент — это неправильно!
```

Как правильно выбрать key?

Вариант 1. Использовать ID из базы данных:

```
// данные с сервера обычно имеют уникальный id
const users = [
  { id: 'user_123', name: 'Иван' },
  { id: 'user_456', name: 'Мария' }
];

{users.map(user => (
  <div key={user.id}>{user.name}</div>
))}
```

Вариант 2. Сгенерировать уникальный ID при создании данных:

```
// при создании задачи генерируем уникальный id
const newTask = {
  id: Date.now(), // простой способ: временная метка
  // или используйте библиотеку uuid для более надёжных ID
  title: 'Новая задача'
};
```

❌ Что НЕЛЬЗЯ использовать для key:

- Случайные числа: `key={Math.random()}` — ключ будет меняться при каждом рендере
- Одинаковые значения: `key="item"` — все элементы получат одинаковый ключ
- Данные, которые могут повторяться: `key={task.title}` — две задачи могут иметь одинаковое название

Создадим компонент со списком

Шаг 1. Создаём файл компонента

В папке `src` создадим новый файл `TaskList.jsx`. Этот компонент будет отображать список задач.

Шаг 2. Создаём массив данных

Начнём с создания массива задач внутри компонента. Каждая задача — это объект с полями `id`, `title` и `completed`:

```
// src/TaskList.jsx

function TaskList() {
  // массив задач - каждая задача это объект с уникальным id
  const tasks = [
    { id: 1, title: 'Изучить JSX', completed: true },
    { id: 2, title: 'Разобраться с компонентами', completed: false },
    { id: 3, title: 'Освоить работу с props', completed: false },
    { id: 4, title: 'Отклеить этикетки от бананов', completed: false }
  ];

  return (
    <div className="task-list">
      <h2>Список задач</h2>
      {/* здесь будем отображать список */}
    </div>
  );
}

export default TaskList;
```

Пояснение:

- Создали массив `tasks` с четырьмя задачами

- Каждая задача имеет уникальный `id` — это важно для атрибута `key`
- `completed` — булево значение, показывающее, выполнена ли задача

Шаг 3. Используем map() для создания списка

Теперь преобразуем массив задач в список JSX-элементов с помощью метода `map()`:

```
// src/TaskList.jsx

function TaskList() {
  const tasks = [
    { id: 1, title: 'Изучить JSX', completed: true },
    { id: 2, title: 'Разобраться с компонентами', completed: false },
    { id: 3, title: 'Освоить работу с props', completed: false },
    { id: 4, title: 'Отклеить этикетки от бананов', completed: false }
  ];

  return (
    <div className="task-list">
      <h2>Список задач</h2>
      <ul>
        {/* используем map() для преобразования массива в JSX */}
        {tasks.map(task => (
          <li key={task.id}>
            <span>{task.title}</span>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TaskList;
```

Пояснение:

- `tasks.map(task => ...)` — для каждого элемента массива `tasks` создаём JSX-элемент `<li>`
- `task` — это текущий элемент массива (объект задачи)
- `key={task.id}` — **обязательный** атрибут для элементов списка, должен быть уникальным
- Результат `map()` — массив JSX-элементов, который React отобразит

Шаг 4. Добавляем условный рендеринг

Сделаем так, чтобы выполненные и невыполненные задачи отображались по-разному. Добавим разные CSS-классы и иконки:

```
// src/TaskList.jsx

function TaskList() {
  const tasks = [
```

```
    { id: 1, title: 'Изучить JSX', completed: true },
    { id: 2, title: 'Разобраться с компонентами', completed: false },
    { id: 3, title: 'Освоить работу с props', completed: false },
    { id: 4, title: 'Отклеить этикетки от бананов', completed: false }
  ];

  return (
    <div className="task-list">
      <h2>Список задач</h2>
      <ul>
        {tasks.map(task => (
          // добавляем key и условный className
          <li
            key={task.id}
            className={task.completed ? 'completed' : 'pending'}
          >
            <span>{task.title}</span>
            {/* условно отображаем иконку в зависимости от статуса */}
            {task.completed ? '✅' : '⏳'}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TaskList;
```

Пояснение:

- `className={task.completed ? 'completed' : 'pending'}` — динамически назначаем CSS-класс
- Если `task.completed === true`, класс будет `'completed'`, иначе `'pending'`
- `{task.completed ? '✅' : '⏳'}` — отображаем галочку для выполненных задач, песочные часы для невыполненных
- Тернарный оператор `? :` позволяет выбрать одно из двух значений

Шаг 5. Подключаем компонент в App.jsx

Теперь добавим наш компонент списка задач в главное приложение:

```
// src/App.jsx
import './App.css';
import Greeting from './Greeting'; // компонент из Примера 1
import UserCard from './UserCard'; // компонент из Примера 2
import TaskList from './TaskList'; // наш новый компонент

function App() {
  return (
    <div className="App">
      {/* компоненты из предыдущих примеров */}
```

```
        <Greeting />

        <UserCard
          name="Иван Иванов"
          role="Администратор"
          avatarUrl="https://encrypted-tbn0.gstatic.com/images?
q=tbn:ANd9GcRfVMhpKmVy_-iwfRLAiNiaDslMa-2oEz7KTw&s"
          isOnline={true}
        />

        {/* добавляем компонент списка задач */}
        <TaskList />
      </div>
    );
  }

  export default App;
```

Пояснение:

- Импортируем `TaskList` из файла `./TaskList`
- Добавляем компонент `<TaskList />` в разметку
- Компонент отобразит весь список задач автоматически

Шаг 6. Проверяем результат

Сохраним все файлы и посмотрим на результат в браузере. Должен отобразиться список из четырёх задач, где первая задача помечена галочкой (выполнена), а остальные — песочными часами (не выполнены).

Важные моменты о работе со списками

1. **Атрибут key обязателен:** каждый элемент списка должен иметь уникальный `key`
2. **key должен быть стабильным:** используйте `id` из данных, а не индекс массива (индекс может меняться при сортировке)
3. **key не передаётся в props:** это служебный атрибут React, компонент его не получит
4. **map() возвращает новый массив:** исходный массив не изменяется
5. **Можно использовать любые данные:** массив объектов, строк, чисел — `map()` работает со всеми

Частые ошибки и как их избежать

- **Забыли добавить key:** React выдаст предупреждение в консоли. Всегда добавляйте уникальный `key` для элементов списка.
- **Использование индекса как key:** `key={index}` работает, но может вызвать проблемы при изменении порядка элементов. Используйте уникальный `id` из данных.
- **key не уникален:** если несколько элементов имеют одинаковый `key`, React не сможет правильно отследить изменения.
- **Забыли вернуть JSX из map():** убедитесь, что функция в `map()` возвращает JSX-элемент.

Полный код примера

Файл src/TaskList.jsx:

```
// src/TaskList.jsx

function TaskList() {
  // массив задач - каждая задача это объект с уникальным id
  const tasks = [
    { id: 1, title: 'Изучить JSX', completed: true },
    { id: 2, title: 'Разобраться с компонентами', completed: false },
    { id: 3, title: 'Освоить работу с props', completed: false },
    { id: 4, title: 'Отклеить этикетки от бананов', completed: false }
  ];

  return (
    <div className="task-list">
      <h2>Список задач</h2>
      <ul>
        {/* используем map() для преобразования массива в JSX */}
        {tasks.map(task => (
          // добавляем key и условный className
          <li
            key={task.id}
            className={task.completed ? 'completed' : 'pending'}
          >
            <span>{task.title}</span>
            {/* условно отображаем иконку в зависимости от статуса */}
            {task.completed ? '✅' : '⏳'}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TaskList;
```

Файл src/App.jsx:

```
// src/App.jsx
import './App.css';
import Greeting from './Greeting';
import UserCard from './UserCard';
import TaskList from './TaskList';

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

```
      <UserCard
        name="Иван Иванов"
        role="Администратор"
        avatarUrl="https://encrypted-tbn0.gstatic.com/images?
q=tbn:ANd9GcRfVMhpKmVy_-iwfRLAiNiaDslMa-2oEz7KTw&s"
        isOnline={true}
      />

      <TaskList />
    </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:

- Работающее 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 слеши 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 слеши 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

Перед выполнением практической части стоит ознакомиться с содержимым файла `task.pdf` в этом репозитории.

Теперь приступим к созданию нашего основного проекта - трекера изучения технологий. На этом занятии мы заложим фундамент приложения, создав базовые компоненты для дальнейшей разработки.

Начало работы

Шаг 1: Создание проекта Откройте терминал и выполните команду:

```
npm create vite@latest
```

Вам будет предложено ввести название проекта (например, technology-tracker) и выбрать шаблон (выберите React, затем JavaScript или TypeScript по желанию). После создания проекта перейдите в папку проекта.

Шаг 2: Организация структуры проекта Создайте в папке `src` следующую структуру:

```
src/
├── components/
│   ├── TechnologyCard.js
│   └── TechnologyCard.css
├── App.js
├── App.css
└── index.js
```

Такая организация поможет поддерживать порядок в коде по мере роста приложения.

Создание компонента карточки технологии

Шаг 3: Разработка компонента TechnologyCard В файле `TechnologyCard.jsx` создайте компонент, который будет отображать отдельный пункт дорожной карты. Компонент должен принимать следующие свойства:

- `title` - название технологии
- `description` - описание для изучения
- `status` - статус изучения

Используйте подход из второго теоретического примера, адаптировав его под нашу задачу. Не забудьте добавить условное отображение для разных статусов.

Шаг 4: Стилизация карточки В файле `TechnologyCard.css` создайте базовые стили. Важно предусмотреть визуальное различие для различных статусов изучения:

- Используйте разные цвета границ или фона
- Добавьте иконки или индикаторы прогресса
- Обеспечьте четкое визуальное разделение карточек

Сборка основного приложения

Шаг 5: Создание тестовых данных В компоненте `App.js` создайте массив с тестовыми данными для проверки работы. Пример структуры данных:

```
const technologies = [
  { id: 1, title: 'React Components', description: 'Изучение базовых компонентов',
status: 'completed' },
  { id: 2, title: 'JSX Syntax', description: 'Освоение синтаксиса JSX', status:
'in-progress' },
  { id: 3, title: 'State Management', description: 'Работа с состоянием
компонентов', status: 'not-started' }
];
```

Шаг 6: Отображение списка технологий Используя подход из третьего теоретического примера, отобразите массив технологий с помощью компонента `TechnologyCard`. Не забудьте передавать все необходимые свойства и назначить уникальные ключи для каждого элемента списка.

Доработка интерфейса

Шаг 7: Базовая стилизация приложения В файле `App.css` добавьте стили для основного контейнера приложения:

- Задайте максимальную ширину контейнера
- Добавьте отступы для лучшего восприятия
- Определите основные шрифты и цвета

Шаг 8: Проверка работы Убедитесь, что приложение корректно отображает все созданные карточки с разными статусами. Проверьте:

- Отображаются ли все переданные данные
- Работает ли условное отображение статусов
- Корректно ли применяются стили

Самостоятельная работа

Задание для самостоятельного выполнения: Создайте компонент `ProgressHeader`, который будет отображать общую статистику по дорожной карте. Компонент должен показывать:

- Общее количество технологий
- Количество изученных технологий
- Процент выполнения в виде прогресс-бара

Рекомендации по выполнению:

- Используйте полученные знания о работе с props
- Примените подход условного рендеринга для разных состояний прогресса
- Рассчитайте процент выполнения на основе количества технологий со статусом 'completed'
- Добавьте визуальный прогресс-бар с помощью CSS

Что проверить перед завершением:

- Все компоненты импортируются и экспортируются корректно
- Приложение запускается без ошибок
- Карточки отображаются с правильными данными и статусами

- Стили применяются ко всем элементам

По завершении этой работы у вас будет основа для трекера изучения технологий, который мы будем развивать на следующих занятиях. Не стремитесь к идеальному дизайну - сейчас важнее работоспособность и правильная архитектура компонентов.