React JS

Фреймворки и библиотеки

- Язык программирования JavaScript широко используется как во фронтенде, так и в бэкенде. Не удивительно, что в его экосистеме есть множество библиотек, с помощью которых можно легко и быстро создавать самые разные сайты.

- Среда JavaScript стала огромной. Она имеет собственную экосистему библиотек, фреймворков, инструментов, менеджеров пакетов и новых языков, которые компилируются в JavaScript.

- Библиотека JavaScript — это набор готовых функций и / или классов, созданных с использованием языка программирования JS.

- Имея специальную библиотеку JavaScript, разработчик может использовать уже созданные и протестированные сценарии для создания приложения, точно так же, как строители используют кирпичи для строительства дома. Более того, такие готовые скрипты можно использовать как самостоятельные элементы или стать частью более сложных функций.

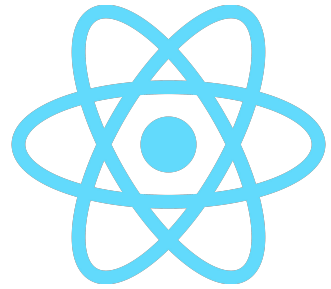
В настоящее время существует множество JS-библиотек, которые можно разделить на:

- масштабные базы данных со всеми распространенными виджетами, аббревиатурами и полифилами;

- узкоспециализированные базы данных для выполнения конкретных задач, таких как создание диаграмм, анимаций, математических функций и т. д.

Самые популярные библиотеки и фреймворки

- React — это декларативная, эффективная и гибкая JavaScript-библиотека для создания пользовательских интерфейсов. Она позволяет вам собирать сложный UI из маленьких изолированных кусочков кода, называемых «компонентами».



- jQuery - это библиотека, которая сделала JavaScript более доступным а DOM-манипуляцией проще, чем раньше.

- AngularJS - одна из самых мощных сред JavaScript. Google использует эту платформу для разработки одностраничного приложения (SPA), предоставляет разработчикам лучшие условия для объединения JavaScript с HTML и CSS.

- Vue.js - это легкий JavaScript фреймворк, который предлагает опыт, похожий на React, с его виртуальными DOM и компонентами повторного использования, которые можно использовать для создания как виджетов, так и целых веб-приложений.

React

React

- React - это библиотека JavaScript, созданная разработчиками Facebook и Instagram. Согласно опросу Stack Overflow Survey 2017, React был признан самой популярной технологией среди разработчиков. React также имеет честь быть самым популярным проектом JavaScript, согласно количеству звезд на GitHub.

- React — это декларативная, эффективная и гибкая JavaScript-библиотека для создания пользовательских интерфейсов. Она позволяет вам собирать сложный UI из маленьких изолированных кусочков кода, называемых «компонентами».

- React использует виртуальную модель DOM, так что вам не нужно беспокоиться о прямом манипулировании с DOM. Другие примечательные особенности React включают однонаправленный поток данных, дополнительный синтаксис JSX и инструмент командной строки для создания проекта React с нуля.

Где используется?

- React используется во множестве веб-приложений:
- **Facebook**
- **Instagram**
- **Netflix**
- **New York Times**
- **WhatsApp**
- **Dropbox**
- **Яндекс**
- **Сбер**
- **Авито**
- **И многие другие...**

Где используется?

- React Native — это кроссплатформенный фреймворк с открытым исходным кодом для разработки нативных мобильных и настольных приложений на JavaScript, его используют:

- **Pinterest**
- **Skype**
- **Discord**
- **Tesla**
- **Instagram**
- **Airbnb**
- **SoundCloud**
- **Microsoft OneDrive**
- **И многие другие...**

Для чего нужен React

- для создания функциональных интерактивных веб-интерфейсов, работая с которыми, не нужно постоянно обновлять страницу;
- быстрой и удобной реализации отдельных компонентов и страниц целиком — элементы в React легко использовать повторно;
- легкой разработки сложных программных структур — их просто описывать, если использовать реализованный в React подход;
- доработки новой функциональности интерфейса с любым изначальным стеком технологий: фреймворк не зависит от остального инструментария и будет хорошо работать, на чем бы ни был написан код;
- разработки одностраничных и многостраничных приложений (SPA и PWA). Это сайты, которые функционируют как программы и веб-сервисы и имеют соответствующий интерфейс;
- работы с серверной частью сайта или разработки интерфейсов мобильных приложений. В таких случаях React используют совместно с инструментами, адаптирующими веб-технологии под другие цели.

React + и -

Преимущества React

- Популярность - Это один из трех самых распространенных фреймворков для фронтенд-разработки. Кроме него, популярны Vue.js и Angular, но первый пока не так распространен, а второй намного сложнее в изучении. Еще одна популярная технология — jQuery, но она постепенно уходит в прошлое.
- Огромное сообщество
- Развитая экосистема
- Простота создания интерфейса
- Реактивность - Фреймворк реагирует на обновление компонента и автоматически отображает его изменения в дереве документа.
- Высокая скорость работы

Недостатки React

- Запутанность синтаксиса - Технология JSX удобная и широко используется, но сначала она может вызвать сложности в изучении.

- Трудности с SEO-оптимизацией (SEO — это поисковая оптимизация, проработка сайта таким образом, чтобы он соответствовал требованиям поисковых систем.)

- Своеобразная архитектура проекта

Особенности React

Декларативность

- Декларативный стиль означает, что разработчику достаточно один раз описать, как будут выглядеть результаты работы кода — элементы в разных состояниях. Ему не нужно фокусироваться на способах достижения результатов: большую часть работы выполнит фреймворк. React будет автоматически обновлять состояние элементов в зависимости от условий, главная задача — грамотно описать эти состояния. Удобный и понятный подход облегчает написание и отладку кода.

Виртуальное DOM-дерево

- Любой веб-интерфейс основан на HTML-документе и CSS-стилях, к которым подключен код на JavaScript. Структура HTML-документа, точнее его модель, называется DOM-деревом (DOM расшифровывается как Document Object Mode, объектная модель документа). Это древовидная модель, в которой в иерархическом виде собраны все используемые на странице элементы.

- Особенность React в том, что он создает и хранит в кэше виртуальное DOM-дерево — копию DOM, которая изменяется быстрее, чем реальная структура. Это нужно для того, чтобы быстро обновлять состояние страницы. Если пользователь выполнит действие или наступит какое-либо событие, DOM должна измениться, так как изменятся объекты на странице. Но реальная объектная модель может быть огромной, ее обновление — медленный процесс. Поэтому React работает не с ней, а с виртуальной копией в кэше, которая весит меньше.

Обновление DOM по частям

- Чтобы улучшить быстродействие, React обновляет DOM не полностью. Он хранит в памяти две облегченных копии: актуальную и предыдущую. Когда что-то обновляется, фреймворк сравнивает версии между собой и изменяет только ту часть дерева, которая действительно поменялась. Это нужно, чтобы не перезагружать DOM целиком и не замедлять работу страницы. Подход кажется сложным, но он важен для оптимизации загрузки.

Возможность повторно использовать компоненты

- React основан на компонентах — отдельных элементах веб-интерфейса. Компоненты инкапсулированы, то есть самостоятельны: в каждом из них размещены все методы и данные, необходимые для работы.

- Инкапсулированные самостоятельные компоненты можно использовать повторно, размещать в другом месте кода, в ином разделе или на другой странице. Данные можно переносить по всему приложению, использовать вне DOM конкретной страницы. Это ускоряет разработку и сокращает количество действий для создания функционирующего интерфейса. Благодаря отсутствию сложных зависимостей инкапсуляция также облегчает отладку.

Нисходящий поток данных

- Компоненты могут передавать свойства и данные друг другу, но только в одном направлении — от «родительских» к дочерним. Это помогает реализовать четкую иерархию, облегчает отладку. Однонаправленный поток данных означает, что программист всегда может понять, откуда именно к элементу поступили данные.

Синтаксис JSX

- JSX расшифровывается как JavaScript XML. Это расширение языка JavaScript, которое помогает описывать HTML-подобные элементы с помощью кода на React. С помощью синтаксиса разработчики на React создают компоненты страницы и гибко управляют ими.

- Несмотря на то что элементы похожи на HTML, это по-прежнему JavaScript с возможностью быстро и легко изменять DOM с помощью кода. И все же JSX воспроизводится как HTML: по сути разработчик описывает нужный компонент на языке разметки, а тот остается JavaScript-объектом с широкой функциональностью. Это удобно, упрощает программирование, но может запутать начинающих разработчиков.

React Hooks

- Сейчас в React есть поддержка хуков — так называются специальные функции-«крючки», которые «цепляются» за состояние элемента или за метод. Изменение состояния или вызов метода «тащит» за собой эти функции, и они автоматически выполняются

React Developer Tools

- Так как React — очень популярная технология, его создатели разработали бесплатные расширения для браузера с инструментами для проверки и отладки. Фронтендеры часто пользуются консолью и панелью разработчика в браузере, чтобы проверить, как работает их код. React Developer Tools облегчают задачу и расширяют возможности.

Начало работы

Пробуем React

- React изначально был спроектирован так, чтобы его можно было внедрять постепенно. Другими словами, вы можете начать с малого и использовать только ту функциональность React, которая необходима вам в данный момент.

- Для внедрения React не надо ничего переписывать. Его можно использовать как для маленькой кнопки, так и для целого приложения. Возможно, вы захотите немного «оживить» вашу страницу. React-компоненты подходят для этого как нельзя лучше.

- Большинство сайтов в Интернете является обычными HTML-страницами. Даже если ваш сайт не относится к одностраничным приложениям, вы можете добавить на него React, написав всего несколько строк кода без каких-либо инструментов сборки. В зависимости от целей, можно постепенно перенести на React весь сайт или переписать всего несколько виджетов.

Способ 1. Добавление на страницу

Добавляем DOM-контейнер в HTML

- Для начала, откройте HTML-файл страницы, которую хотите отредактировать. Добавьте пустой тег в месте, где вы хотите отобразить что-нибудь с помощью React. Например:

```
<!-- ... остальной HTML ... -->

<div id="like_button_container"></div>

<!-- ... остальной HTML ... -->
```

Добавляем script-теги

- Теперь добавьте три <script>-тега перед закрывающим тегом </body>:

```
<!-- ... остальной HTML ... -->

<!-- Загрузим React. -->
<!-- Примечание: при деплое на продакшен замените «development.js» на «production.min.js». -->
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>

<!-- Загрузим наш React-компонент. -->
<script src="like_button.js"></script>
```

Создаём React-компонент

- Создайте файл с именем like_button.js рядом с вашим HTML-файлом.

```
const e = React.createElement;

class LikeButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = { liked: false }; }

  render() {
    if (this.state.liked) {return 'You liked this.'; }

    return e('button',{ onClick: () => this.setState({ liked: true }) },'Like');
  }
}
```

Создаём React-компонент

- Добавьте ещё 3 строки в конец файла like_button.js, после стартового кода:

```
const domContainer = document.querySelector('#like_button_container');
const root = ReactDOM.createRoot(domContainer);
root.render(e(LikeButton));
```

- **Эти три строки кода ищут элемент , который мы добавили в HTML на первом шаге, а затем отображают React-компонент с кнопкой «Like» внутри него.**

Способ 2. React-приложение

Node.js

- Node.js — это программная платформа, которая транслирует JavaScript в машинный код, исполняемый на стороне сервера. Таким образом, JavaScript можно использовать для создания серверной части.
- npm — менеджер (установщик) пакетов, входящий в состав Node.js.
- npx — позволяет запускать пакеты Node.js, не устанавливая его и не добавляя зависимости в ваш проект. Некоторые пакеты разрабатываются специально для того, чтобы быть запущенными с помощью npx.
- Скачиваем на — https://nodejs.org/

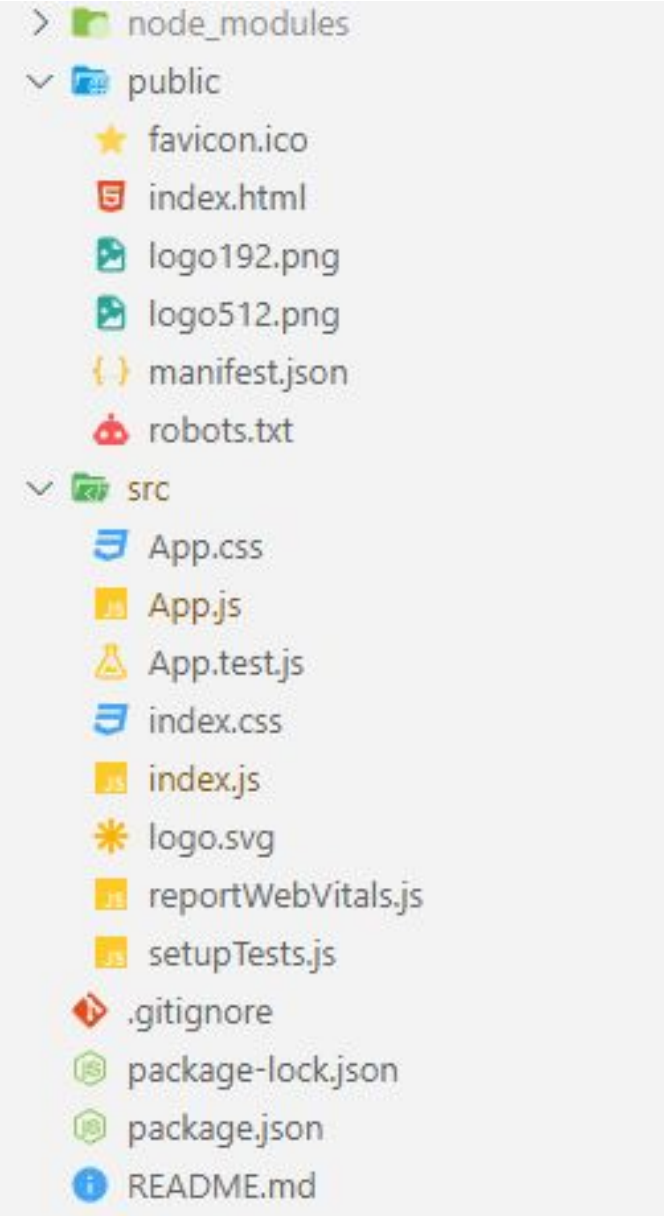
Node.js

- Для установки фреймворка вам понадобятся NodeJS не ниже версии 8.10 и npm не ниже версии 5.6 на вашем компьютере. Для создания проекта в командной строке выполните следующие команды:

  - `npx create-react-app my-app`
  - `cd my-app`
  - `npm start`

- После этого у вас появится папка my-app, содержащая в себе библиотеку. Найдите в этой папке папку src - это будет ваша рабочая папка, в которой вы будете вести разработку вашего проекта.

Структура проекта



node_modules/ содержит все внешние библиотеки JavaScript, используемые приложением. Вам нечасто потребуется использовать его.

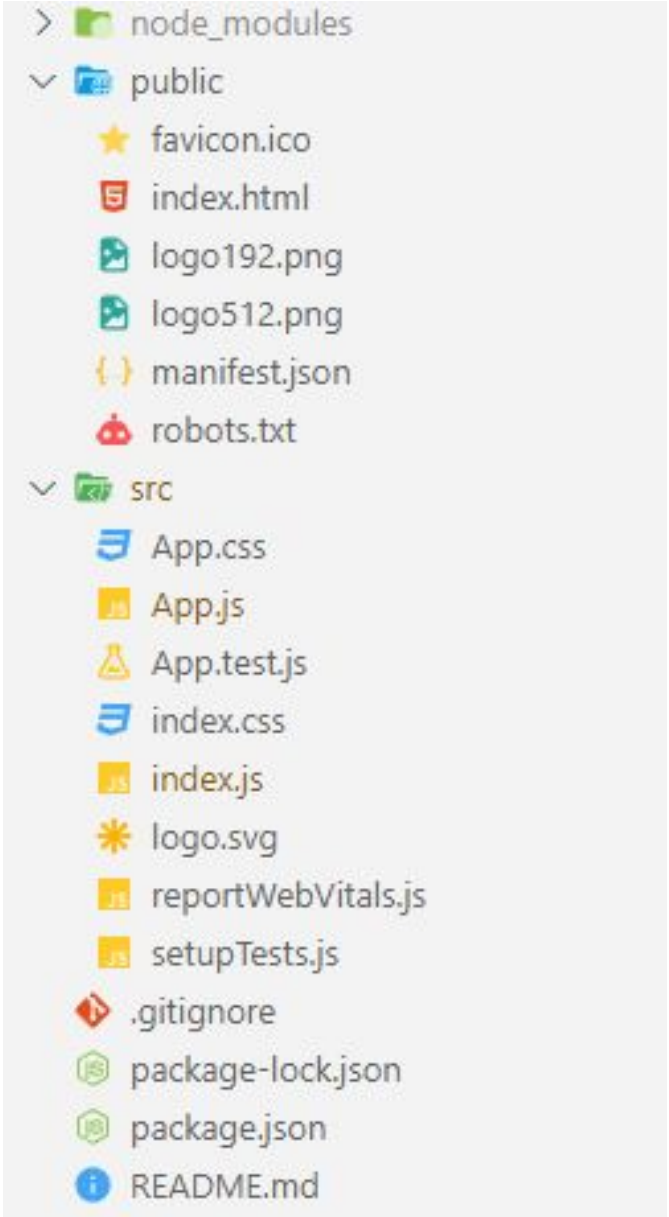
Директория public/ содержит базовые файлы HTML, JSON и изображений. Это корневые ресурсы вашего проекта.

В директории src/ содержится код React JavaScript для вашего проекта. В основном вы будете работать именно с этой директорией.

Файл .gitignore содержит несколько директорий и файлов по умолчанию, которые система контроля исходного кода git будет игнорировать, в том числе директорию node_modules.

README.md — это файл разметки, содержащий много полезной информации о приложении Create React App, в том числе обзор команд и ссылки на расширенные опции конфигурации.

Структура проекта



Последние два файла используются вашим диспетчером пакетов.

При запуске первоначальной команды npx вы создали базовый проект, а также установили дополнительные зависимости. При установке зависимостей вы создали файл package-lock.json. Этот файл используется npm для проверки точного соответствия версий пакетов.

Последний файл — package.json. Он содержит метаданные о вашем проекте, включая заголовок, номер версии и зависимости. Также он содержит скрипты, которые вы можете использовать для запуска проекта.

App.js

- В папке src найдите файл App.js. Ближайшие уроки этот файл будет вашим основным рабочим файлом. В нем вы будете писать код, наблюдая его результаты в окне браузера.

- Откройте этот файл в редакторе и удалите из него все лишнее, приведя его вот к такому виду:

```
import React from 'react';

function App() {
  return <div>
    text
  </div>;
}

export default App;
```

App.js

- После того, как ваш проект установлен, для следующего запуска (например, после перезагрузки компьютера) вам достаточно будет перейти через терминал в папку my-app и выполнить следующую команду:

- `npm start`

Плагин для разработки

- Установите в ваш браузер специальный плагин react-devtools, помогающий вести разработку на React.

Компонентный подход в React

Компонентный подход в React

- Пусть у нас есть сайт. На этом сайте мы можем выделить некоторые блоки: хедер, контент, сайдбар, футер и так далее. Каждый блок можно разделить на более мелкие подблоки. К примеру в хедере обычно можно выделить логотип, менюшку, блок контактов и так далее.

- В React каждый такой блок называется компонентом. Каждый компонент может содержать в себе более мелкие компоненты, те в свою очередь еще более мелкие и так далее.

- Каждому компоненту в React соответствует js модуль, расположенный в папке src. Имя файла с модулем пишется с большой буквы и должно соответствовать функции, которая расположена в коде этого модуля.

- Например, файл с названием App.js должен содержать внутри себя функцию App

Основной компонент

- Один из компонентов должен быть основным - тем, к которому добавляются остальные компоненты. В React по умолчанию таким компонентом будет компонент App.

- К этому компоненту будут подключаться другие компоненты. Как это делается - мы разберем далее.

Макет сайта

- В папке my-app/public в файле index.html расположен макет сайта. Вы можете размещать в нем любой HTML код - и вы увидите результат этого кода в браузере.

- Кроме того, в макете сайта есть специальный див с id, равным root, в который монтируется основной компонент. Мод монтированием понимается то, что в этот див будет выводится результат работы нашего компонента.

Результат работы компонента

- В див с результатом будет выведено то, что возвращает через return функция компонента. В следующем примере это будет див с текстом:

```
function App() {
  return <div>
    text
  </div>;
}
```

- Обратите внимание на то, что див мы пишем без кавычек - просто пишем тег в JavaScript коде! Это основная особенность React.
- На самом деле в React мы пишем не на языке JavaScript, а на языке JSX, который мы будем изучать в следующих уроках.
- Процесс преобразования JSX в итоговый HTML код называется *рендерингом*.

JSX в React

JSX в React

- Язык JSX - это обычный JavaScript, но с некоторыми дополнениями, позволяющими писать теги прямо в коде, без кавычек.

- Теги можно возвращать через return:

```
function App() {
  return <div>
    text
  </div>;
}
```

JSX в React

- Теги можно записывать в переменные или константы:

```
function App() {
  const  elem = <div>text</div>;
  return elem;
}
```

Закрытость тегов

- Все теги в JSX должны быть закрыты, в том числе теги, которые не требуют пары, например, input или br.

- Следующий пример кода выдаст ошибку, так как инпут не закрыт:

```
function App() {
  return <div>
    <input>
  </div>;
}
```

- Закроем его с помощью обратного слеша:

```
function App() {
  return <div>
    <input />
  </div>;
}
```

Корректность верстки

- Верстка в JSX должна быть корректной. В частности, не все теги можно вкладывать друг в друга. Например, если в теге ul разместить абзац, это приведет к ошибке.
- Помимо более-менее очевидных недопустимых вложенностей, есть также и неожиданные: таблицы.
- В таблицах мы привыкли сразу в тег table вкладывать теги tr, вот так:

```
function App() {
  return <table>
    <tr>
      <td>1</td>
      <td>2</td>
    </tr>
    <tr>
      <td>3</td>
      <td>4</td>
    </tr>
  </table>;
}
```

Корректность верстки

- Такой код в React приведет к выводу предупреждения в консоли, так как tr должны быть вложены либо в тег tbody, либо в thead, либо в tfoot.

- Давайте исправим проблему, сделав нашу таблицу корректной:

```
function App() {
  return <table>
    <tbody>
      <tr>
        <td>1</td>
        <td>2</td>
      </tr>
      <tr>
        <td>3</td>
        <td>4</td>
      </tr>
    </tbody>
  </table>;
}
```

Атрибуты

- В теги можно добавлять атрибуты:

```
function App() {
  return <div id="elem">
    text
  </div>;
}
```

- Некоторые атрибуты представляют собой исключения: вместо атрибута class следует писать атрибут className, а вместо атрибута for следует писать атрибут htmlFor:

```
function App() {
  return <div className="block">
    <label htmlFor="elem">text</label>
    <input id="elem" />
  </div>;
}
```

Практика

- Функция в вашем основном компоненте сейчас должна выглядеть следующим образом:

```
import React from 'react';

function App() {
  return <div>
    text
  </div>;
}

export default App;
```

№1

- Поменяйте текст внутри дива. Посмотрите на изменения, произошедшие в браузере.
- Добавьте в див несколько абзацев.
- Добавьте в див несколько инпутов, разделенных тегами br.

№2

- Сделайте внутри дива список ul, содержащий в себе 10 тегов li.

№3

- Сделайте внутри дива таблицу с тремя рядами и тремя колонками.

№4

- Сделайте внутри дива три абзаца с различными CSS классами.

Правила возврата функцией тегов JSX

Вложенность тегов

- Внутри тега, который возвращается через return, может быть сколько угодно вложенных тегов:

```
function App() {
  return <div>
    <p>text1</p>
    <p>text2</p>
  </div>;
}
```

Снос тега вниз

- Открывающий тег обязательно должен быть написан на одной строке с командой return.

Например, следующий код работать не будет:

```
function App() {
  return
    <div>
      <p>text1</p>
      <p>text2</p>
    </div>;
}
```

- Для того, чтобы такой снос тега вниз заработал, наш тег необходимо взять в круглые скобки:

```
function App() {
  return (
    <div>
      <p>text1</p>
      <p>text2</p>
    </div>
  );
}
```

Возврат нескольких тегов

- Через return нельзя возвращать сразу несколько тегов. То есть следующий код работать не будет:

```
function App() {
  return (
    <div>
      <p>text1</p>
      <p>text2</p>
    </div>
    <div>
      <p>text3</p>
      <p>text4</p>
    </div>
  );
}
```

Чтобы желаемое заработало, нам придется взять наши теги в какой-нибудь общий тег, например, вот так:

```
function App() {
  return (
    <div>
      <div>
        <p>text1</p>
        <p>text2</p>
      </div>
      <div>
        <p>text3</p>
        <p>text4</p>
      </div>
    </div>
  );
}
```

Возврат нескольких тегов

- Такой прием сработает, однако, он не без недостатков: в результате рендеринга мы получим дополнительный див, который мы в общем не хотели и ввели исключительно для корректности работы React. Этот див, к примеру, может сломать нам часть верстки.

- Для избежания таких проблем в React введен специальный пустой тег <></>, который группирует теги, но в готовую верстку не попадает. Давайте воспользуемся этим тегом:

```
function App() {
  return (
    <>
      <div>
        <p>text1</p>
        <p>text2</p>
      </div>
      <div>
        <p>text3</p>
        <p>text4</p>
      </div>
    </>
  );
}
```

Возврат незакрытого тега

- В качестве результата можно возвращать тег, который не нужно закрывать, например, инпут. Как вы уже знаете, по правилам React в этом случае вместо тега <input> нужно писать тег <input />:

```
function App() {
  return <input />;
}
```

Возврат пустого тега

- Пусть мы хотим вернуть пустой тег:

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

- В этом случае код можно сократить вот так:

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

- При этом в получившемся HTML коде React автоматические преобразует сокращенную форму в нормальную.

№1

- В следующем коде автор кода хочет вернуть тег ul:

```
function App() {
  return
    <ul>
      <li>text1</li>
      <li>text2</li>
      <li>text3</li>
    </ul>;
}
```

- Код, однако, не работает. Исправьте ошибку автора кода.

№2

- Автор следующего кода хочет вернуть сразу два тега ul:

```
function App() {
  return <ul>
    <li>text1</li>
    <li>text2</li>
    <li>text3</li>
  </ul>
  <ul>
    <li>text1</li>
    <li>text2</li>
    <li>text3</li>
  </ul>;
}
```

- Код, однако, не работает. Исправьте ошибку автора кода.

№3

- Автор следующего кода хочет вернуть инпут:

```
function App() {
  return <input>;
}
```

- Код, однако, не работает. Исправьте ошибку автора кода.

№4

- Автор следующего кода хочет вернуть три инпута:

```
function App() {
  return <input><input><input>;
}
```

- Код, однако, не работает. Исправьте ошибку автора кода.

Введение в компоненты JSX

Компоненты React

- Взглянем на любой сайт. Он состоит из набора независимых блоков: хедер, сайдбары, футер, контент. Можно сказать, что эти блоки и есть компоненты в том смысле, в котором подразумевается в React.

- Если посмотреть на тот же хедер, что в нем можно выделить блок с логотипом, блок контактов, блок с меню и так далее. То есть компоненты могут состоять из других подкомпонентов.

- Аналогичным образом дело обстоит в React - сайт строится из набора компонентов, которые в свою очередь могут содержать другие компоненты.

- В React каждый компонент представляет собой отдельный модуль. Обычно разработка начинается с главного компонента App, который содержит в себе остальные.

Компоненты React

- Давайте потренируемся создавать новые компоненты. Их принято создавать в отдельных файлах.

- Пусть для примера нам нужен компонент, выводящий данные продукта. Для этого нам нужно в рабочей папке создать файл Product.js и добавить в него следующий код:

```
function Product() {
    return <p>
        product
    </p>;
}
```

Компоненты React

- Для того, чтобы мы могли использовать компонент в другом файле, нам нужно его экспортировать – разрешить использование в других файлах с помощью команды export

```
function Product() {
    return <p>
        product
    </p>;
}

export default Product;
```

Компоненты React

- Чтобы отобразить наш компонент внутри другого компонента, нам необходимо добавить его – импортировать с помощью команды import

```
import Product from './Product'; // импортируем продукт

function App() {
    return <div>
        text
    </div>;
}

export default App;
```

Компоненты React

- Каждому подключенному компоненту соответствует свой JSX тег. К примеру, у нас есть компонент Product, а значит ему соответствует тег <Product />.
- Название тега компонента обязательно следует писать с большой буквы, чтобы React мог отличить вызов компонента от использования тега HTML.

```
import Product from './Product';

function App() {
    return <div>
        <Product />
    </div>;
}

export default App;
```

Несколько экземляров компонента

- Можно вставить несколько продуктов. Для этого нужно просто написать несколько тегов компонента:

```
import Product from './Product';

function App() {
    return <div>
        <Product />
        <Product />
        <Product />
    </div>;
}

export default App;
```

№1

- Сделайте компонент User, который будет выводить данные юзеров. Пусть сейчас этот компонент просто выводит какой-нибудь текст. Используйте этот компонент в компоненте App.

- Добавьте в компонент App несколько экземпляров компонента User

№2

- Создайте базовую разметку сайта, разбив её на компоненты Header, Main, Footer, компоненты заполните соответствующим контентом. Все компоненты добавьте в компонент App

Вставка значений в JSX

Вставка значений

- Пусть у нас есть следующий код:

```
function App() {
    const str = 'text';

    return <div>
        text
    </div>;
}
```

- Чтобы в текст дива вставилось значение константы str нашу константу нужно написать в фигурных скобках внутри, вот так:

```
function App() {
    const str = 'text';

    return <div>
        {str}
    </div>;
}
```

Нюансы

- 1) Кроме вставки константы в теге может быть еще какой-нибудь текст:

```
function App() {
    const str = 'text';

    return <div>
        eee {str} bbb
    </div>;
}
```

- 2) В один тег можно вставлять сколько угодно констант:

```
function App() {
    const str1 = 'text1';
    const str2 = 'text2';

    return <div>
        {str1} {str2}
    </div>;
}
```

Нюансы

- 3) Вставки констант также могут разделяться каким-либо текстом:

```
function App() {
    const str1 = 'text1';
    const str2 = 'text2';

    return <div>
        {str1} eee {str2}
    </div>;
}
```

Выполнение JavaScript

- Внутри фигурных скобок можно не только вставлять переменные и константы, но и выполнять произвольный JavaScript код. Давайте, к примеру, в момент вставки найдем сумму двух констант:

```
function App() {
    const num1 = 1;
    const num2 = 2;

    return <div>
        {num1 + num2}
    </div>;
}
```

Массивы и объекты

- Можно выполнять вставку не только примитивов, но также массивов и объектов.

Вот пример с массивом:

```
function App() {
    const arr = [1, 2, 3];

    return <div>
        <p>{arr[0]}</p>
        <p>{arr[1]}</p>
        <p>{arr[2]}</p>
    </div>;
}
```

Вот пример с объектом:

```
function App() {
    const obj = {a: 1, b: 2, c: 3};

    return <div>
        <p>{obj.a}</p>
        <p>{obj.b}</p>
        <p>{obj.c}</p>
    </div>;
}
```

№1

- Дан следующий код:

```
function App() {
    const str1 = 'text1';
    const str2 = 'text1';

    return <div>
        <p></p>
        <p></p>
    </div>;
}
```

- Вставьте первую константу в первый абзац, а вторую константу - во второй.

№2

- Дан следующий код:

```
function App() {
    const name = 'user';
    const age  = '30';

    return <div>
        name: ?
        age:  ?
    </div>;
}
```

- Вставьте константу с именем вместо первого знака ?, а константу с возрастом - вместо второго.

№3

- Напишите код, возвращающий num1 в степени num2:

```
function App() {
    const num1 = 3;
    const num2 = 2;

    return ?
}
```

№4

- Напишите код, возвращающий Имя и Фамилию человека, разделенные нижним подчеркиванием:

```
function App() {
    const name = 'john';
    const surname = 'smit';

    return ?
}
```

№5

- Напишите код, возвращающий корень из числа:

```
function App() {
    const num = 4;

    return ?
}
```

№6

- Сделайте так, чтобы результатом рендеринга был тег ul, в тегах li которого будут стоять элементы массива.

```
function App() {
    const arr = [1, 2, 3, 4, 5];

    return ?

}
```

№7

- Сделайте так, чтобы результатом рендеринга был следующий код:

```
<p>
    name:    <span>john</span>, <br>
    surname: <span>smit</span>,
</p>
```

- Для значений имени и фамилии используйте значения элементов объекта

```
function App() {
    const obj = {name: 'john', surname: 'smit’};

    return ?
}
```

Хранение тегов в JSX

Хранение тегов

- В переменных и константах можно хранить теги, выполняя затем их вставку в нужное место.

```
function App() {
    const str = <p>text</p>;

    return <div>
        {str}
    </div>;
}
```

- В результате рендеринга получится следующий код:

```
<div>
    <p>text</p>
</div>
```

Несколько тегов в константах

- Учтите, что несколько тегов, хранящихся в константе, обязательно нужно обернуть в какой-то общий тег.

```
function App() {
    const str =
<div><p>text1</p><p>text2</p></div>;

    return <main>
        {str}
    </main>;
}
```

- Можно также использовать пустые теги:

```
function App() {
    const str = <><p>text1</p><p>text2</p></>;

    return <main>
        {str}
    </main>;
}
```

Теги не в одну строку

- Теги, записываемые в константы, не обязательно писать на одной строке. Можно сделать и так:

```
function App() {
    const str = <p>
        text
    </p>;
    return <div>
        {str}
    </div>;
}
```

- Константы с тегами можно возвращать через return:

```
function App() {
    const str = <main>
        text
    </main>;

    return str;
}
```

Вставка в атрибуты тегов

- Вставку переменных и констант можно делать не только в тексты тегов, но и в атрибуты. При этом кавычки от атрибутов не ставятся:

```
function App() {
    const str = 'elem';

    return <div id={str}>
        text
    </div>;
}
```

Результатом работы этого кода будет следующее:

```
<div id="elem">
    text
</div>
```

Исключения

- Напоминаю, что вместо атрибута class следует писать атрибут className:

```
function App() {
    const str = 'elem';

    return <div className={str}>
        text
    </div>;
}
```

А вместо атрибута for следует писать атрибут htmlFor:

```
function App() {
    const str = 'elem';

    return <div>
        <label htmlFor={str}>text</label>
    </div>;
}
```

№1

- Дан следующий код:

```
function App() {
    const li1 = <li>text1</li>;
    const li2 = <li>text2</li>;
    const li3 = <li>text3</li>;
}
```

- Сделайте так, чтобы результатом рендеринга было следующее:

```
<ul>
    <li>text1</li>
    <li>text2</li>
    <li>text3</li>
</ul>
```

№2

- Автор следующего кода хотел записать в константу несколько тегов li и вставить их в ul:

```
function App() {
    const items =
<li>text1</li><li>text2</li><li>text3</li>;

    return <ul>
        {items}
    </ul>
}
```

- Код, однако, не работает. Исправьте ошибку автора кода.

№3

- Некий программист написал следующий код:

```
function App() {
    const str1 = 'label';
    const str2 = 'block';
    const str3 = 'elem';

    return <div>
        <label id={str1} for={str2} class={str3}>text</label> <br>
        <input id={str2}>
    </div>;
}
```

- Код, однако, не работает. Исправьте ошибку автора кода.

Применение условий в JSX

Применение условий

- Давайте сделаем так, чтобы в зависимости от содержимого константы show на экран вывелся или один, или другой текст:

```
function App() {
    let text;
    const show = true;

    if (show) {
        text = 'text1';
    } else {
        text = 'text2';
    }

    return <div>
        {text}
    </div>;
}
```

Только показ

- Можно сделать так, чтобы текст показывался, если в show будет true, и не показывался, если в ней будет false:

```
function App() {
    let text;
    const show = true;

    if (show) {
        text = <p>text</p>;
    }

    return <div>
        {text}
    </div>;
}
```

Возврат тегов

- Можно возвращать через return переменную, содержащую тег:

```
function App() {
    let text;
    const show = true;

    if (show) {
        text = <p>text1</p>;
    } else {
        text = <p>text2</p>;
    }

    return text;
}
```

№1

- Пусть в константе isAdult содержится true, если пользователю уже есть 18 лет, и false, если нет:

```
function App() {
    const isAdult = true;


}
```

- Сделайте так, чтобы в зависимости от значения isAdult на экране показался или один абзац с текстом, или другой.

№2

- Пусть в константе isAdmin содержится true, если пользователь админ, и false, если нет:

```
function App() {
    const isAdmin = true;


}
```

- Сделайте так, чтобы, если isAdmin имеет значение true, на экране показался див с абзацами. В противном случае ничего показывать не нужно.

Сокращенные условия в JSX

Применение условий

- Как вы уже знаете, внутри фигурных скобок можно выполнять JavaScript код. На самом деле этот код может быть не любым, а только самым простым.

- В частности, условия if там применять нельзя. Взамен следует пользоваться сокращенными вариантами условий. Давайте разберемся с ними.

Тернарный оператор

- Давайте в зависимости от значения константы show выведем один или другой текст. Используем для этого тернарный оператор:

```
function App() {
    const show = true;

    return <div>
        {show ? 'text1' : 'text2'}
    </div>;
}
```

Тернарный оператор

- Можно работать не только с текстами, но и с тегами:

```
function App() {
    const show = true;

    return <div>
        {show ? <p>text1</p> : <p>text2</p>}
    </div>;
}
```

Оператор &&

- Нам может понадобится выполнить вставку текста, если условие истинно, и ничего не делать, если ложно. Это делается с помощью оператора &&.

- Давайте посмотрим его работу на примере. В следующем коде, если в show будет true, то вставиться абзац с текстом, а если будет false, то ничего не вставится:

```
function App() {
    const show = true;

    return <div>
        {show && <p>text</p>}
    </div>;
}
```

Инвертирование

- Может быть и обратная ситуация: нужно выполнить вставку текста, если условие ложно, и ничего не делать, если истинно. Для этого нужно выполнить инвертирования константы с помощью оператора !.

- В следующем примере, если в hide будет false, то вставиться абзац с текстом:

```
function App() {
    const hide = false;

    return <div>
        {!hide && <p>text</p>}
    </div>;
}
```

№1

- Дан следующий код:

```
function App() {
    const age = 19;

    return <div>

    </div>;
}
```

- Если в age записано больше 18 лет, то в тексте тега div покажите пользователю абзац с одним текстом, а если меньше - то с другим.
- Не использовать if

№2

- Дан следующий код:

```
function App() {
    const isAuth = true;

    return <div>
        <p>вы авторизованы</p>
    </div>;
}
```

- Сделайте так, чтобы приведенный абзац с текстом показывался только если в isAuth записано true.

- Не использовать if

№3

- Дан следующий код:

```
function App() {
    const isAuth = false;

    return <div>
        <p>пожалуйста, авторизуйтесь</p>
    </div>;
}
```

- Сделайте так, чтобы приведенный абзац с текстом показывался только если в isAuth записано false.

- Не использовать if

Использование функций в React

Использование функций

- Внутри основной функции компонента можно делать вспомогательные функции. К примеру, давайте с помощью вспомогательных функций найдем сумму степеней двух чисел и выведем ее в тексте тега:

```
function App() {
    function square(num) {
        return num ** 2;
    }

    function cube(num) {
        return num ** 3;
    }

    const sum = square(3) + cube(4);

    return <div>
        {sum}
    </div>
}
```

Вызов функций внутри тегов

- Функции можно вызывать прямо в фигурных скобках:

```
function App() {
    function square(num) {
        return num ** 2;
    }

    return <div>
        {square(3)}
    </div>
}
```

Вызов функций внутри тегов

- В фигурных скобках можно также делать вызовы нескольких функций:

```
function App() {
    function square(num) {
        return num ** 2;
    }

    function cube(num) {
        return num ** 3;
    }

    return <div>
        {square(3) + cube(4)}
    </div>
}
```

№1

- Сделайте функцию getDigitsSum, которая будет находить сумму цифр переданного числа. С ее помощью выведите на экран сумму цифр числа 123.

- Используя созданную вами функцию getDigitsSum выведите в абзац сумму цифр числа 12345.

События в JSX

Навешивание событий

- Давайте теперь изучим работу с событиями на React. К примеру, сделаем так, чтобы по клику на блок выводился алерт с некоторым текстом.

- Пусть у нас есть функция showMess, которая выводит алерт с сообщением, а в HTML коде есть кнопка, по клику на которую нам и хотелось бы видеть этот алерт:

```
function App() {
    function showMess() {
        alert('hello');
    }

    return <div>
        <button>show</button>
    </div>;
}
```

Навешивание событий

- Давайте привяжем к нашей кнопке событие onclick так, чтобы по клику на этот див срабатывала функция showMess. Для этого нужно добавить атрибут onClick (именно в camelCase, то есть onClick, а не onclick), а в нем указать функцию, которая выполнится по этому событию:

```
function App() {
    function showMess() {
        alert('hello');
    }

    return <div>
        <button onClick={showMess}>show</button>
    </div>;
}
```

Параметры в функции

- Пусть наша функция showMess параметром принимает имя того, с кем мы хотим поздороваться:

```
function showMess(name) {
    alert('hello, ' + name);
}
```

- Можно передать этот параметр при привязывании функции к событию. Для этого вызов нашей функции следует обернуть в стрелочную функцию:

```
function App() {
    function showMess(name) {
        alert('hello, ' + name);
    }

    return <div>
        <button onClick={() => showMess('user')}>show</button>
    </div>;
}
```

Параметры в функции

- Таким образом мы можем привязать одну и ту же функцию к нескольким кнопкам с разными параметрами:

```
function App() {
    function showMess(text) {
        alert(text);
    }

    return <div>
        <button onClick={() => showMess('user1')}>show1</button>
        <button onClick={() => showMess('user2')}>show2</button>
    </div>;
}
```

№1

- Сделайте так, чтобы по клику на первую кнопку срабатывала первая функция, а по клику на вторую кнопку - вторая функция.

```
function App() {
    function show1() {
        alert(1);
    }

    function show2() {
        alert(2);
    }

    return <div>
        <button>act1</button>
        <button>act2</button>
    </div>;
}
```

№2

- Сделайте так, чтобы по клику на первую кнопку алертом выводилось число 1, по клику на вторую кнопку - число 2, а по клику на третью - число 3.

```
function App() {
    return <div>
        <button>act1</button>
        <button>act2</button>
        <button>act3</button>
    </div>;
}
```

Объект Event в React

Объект Event

- Внутри функции, привязанной к обработчику событий, доступен объект Event:

```
function App() {
    function func(event) {
        console.log(event); // объект с событием
    }

    return <div>
        <button onClick={func}>act</button>
    </div>;
}
```

- В переменную event попадает не родной объект Event браузера, а специальная кроссбраузерная обертка над ним со стороны React. Эта обертка называется SyntheticEvent. Эта обертка помогает событиям работать одинаково во всех браузерах.

№1

- Дана кнопка. По клику на нее получите объект Event и выведите его в консоль.

№2

- Дана кнопка. По клику на нее получите выведите в консоль event.target клика.

Event передача параметров

- Пусть у нас есть некоторая функция func, которую мы хотим использовать в качестве обработчика события. Пусть эта функция принимает некоторый параметр:

```
function func(arg) {
    console.log(arg);
}
```

- Давайте используем эту функцию в качестве обработчика, передав ей параметр:

```
function App() {
    function func(arg) {
        console.log(arg);
    }

    return <div>
        <button onClick={() => func('eee')}>act</button>
    </div>;
}
```

Event передача параметров

- Пусть теперь кроме параметра мы хотим получить в нашей функции объект Event. Для этого нам нужно поступить следующим образом:

```
function App() {
    function func(arg, event) {
        console.log(arg, event);
    }

    return <div>
        <button onClick={event => func('eee', event)}>act</button>
    </div>;
}
```

№3

- Осознайте, как работает приведенный код, запустите у себя.

```
function App() {
    function func(arg, event) {
        console.log(arg, event);
    }

    return <div>
        <button onClick={event => func('eee', event)}>act</button>
    </div>;
}
```

№4

- Переделайте приведенный код так, чтобы функция принимала два параметра.

```
function App() {
    function func(arg, event) {
        console.log(arg, event);
    }

    return <div>
        <button onClick={event => func('eee', event)}>act</button>
    </div>;
}
```

№5

- Модифицируйте предыдущую задачу так, чтобы объект с событием передавался первым параметром функции, а не последним.

№6

- Модифицируйте предыдущую задачу так, чтобы объект с событием передавался вторым параметром функции, находясь между первым и третьим параметрами.

Теги в массивах и циклах JSX

Теги в массивах

- Пусть у нас в массиве хранятся теги:

```
function App() {
  const arr = [<p>1</p>, <p>2</p>, <p>3</p>];
}
```

- Мы можем выполнить вставку содержимого нашей переменной с помощью фигурных скобок:

```
function App() {
  const arr = [<p>1</p>, <p>2</p>, <p>3</p>];

  return <div>
    {arr}
  </div>;
}
```

Формирование в цикле

- Массив с тегами можно создать в цикле:

```
function App() {
  const arr = [];

  for (let i = 0; i <= 9; i++) {
    arr.push(<p>{i}</p>);
  }

  return <div>
    {arr}
  </div>;
}
```

Формирование из массива с данными

- Пусть у нас есть какой-нибудь массив с некими данными, например, вот такой:

```
function App() {
  const arr = [1, 2, 3, 4, 5];
}
```

- Давайте положим каждый элемент этого массива в абзац и выведем эти абзацы в диве. Для этого мы можем воспользоваться любым удобным циклом JavaScript. Например, обычным for-of:

```
function App() {
  const arr = [1, 2, 3, 4, 5];
  const res = [];

  for (const elem of arr) {
    res.push(<p>{elem}</p>);
  }

  return <div>
    {res}
  </div>;
}
```

Формирование из массива с данными

- Однако, в React для таких дел более принято использовать цикл map:

```
function App() {
  const arr = [1, 2, 3, 4, 5];

  const res = arr.map(function(item) {
    return <p>{item}</p>;
  });

  return <div>
    {res}
  </div>;
}
```

Метод «arr.map(callback[, thisArg])» используется для трансформации массива.

Он создаёт новый массив, который будет состоять из результатов вызова callback(item, i, arr) для каждого элемента arr.

№1

- Дан массив:

```
const arr = [
  <li>1</li>,
  <li>2</li>,
  <li>3</li>,
  <li>4</li>,
  <li>5</li>,
];
```

- С помощью этого массива получите результатом рендеринга следующий код:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
```

№2

- С помощью цикла for сформируйте следующий код:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
```

№3

- Дан массив:

```
const arr = ['a', 'b', 'c', 'd', 'e'];
```

- С помощью этого массива получите результатом рендеринга следующий код:

```
<ul>
  <li>a</li>
  <li>b</li>
  <li>c</li>
  <li>d</li>
  <li>e</li>
</ul>
```

Проблема с ключами

- В предыдущем примере мы формировали наши абзацы в цикле, вот так:

```
const res = arr.map(function(item) {
  return <p>{item}</p>;
});
```

- Это будет работать, однако, если заглянуть в консоль - мы увидим ошибку: Warning: Each child in an array or iterator should have a unique "key" prop. В данном случае React требует, чтобы каждому тегу из цикла мы дали уникальный номер, чтобы React было проще с этими тегами работать в дальнейшем.

Проблема с ключами

- Этот номер добавляется с помощью атрибута key. В данном случае в качестве номера мы можем взять номер элемента в массиве. В нашем случае этот номер хранится в переменной index и значит исправленная строка будет выглядеть вот так:

```
function App() {
  const arr = [1, 2, 3, 4, 5];

  const res = arr.map(function(item, index) {
    return <p key={index}>{item}</p>;
  });

  return <div>
    {res}
  </div>;
}
```

Проблема с ключами

- Еще раз: этот атрибут key имеет служебное значение для React, более глубоко вы поймете этот момент в следующих уроках. Пока просто знайте: если вы видите такую ошибку - добавьте атрибут key с уникальным для каждого тега значением и проблема исчезнет.

- Ключ key должен быть уникальным только внутри этого цикла, в другом цикле значения key могут совпадать со значениями из другого цикла.

№4

- Модифицируйте ваше решение предыдущей задачи в соответствии с описанным.

Вывод массива объектов в JSX

Вывод массива объектов

- Пусть у нас есть массив объектов с продуктами:

```
const prods = [
  {name: 'product1', cost: 100},
  {name: 'product2', cost: 200},
  {name: 'product3', cost: 300},
];
```

- Давайте выведем каждый продукт в своем абзаце:

```
function App() {
  const res = prods.map(function(item) {
    return <p>
      <span>{item.name}</span>:
      <span>{item.cost}</span>
    </p>;
  });

  return <div>
    {res}
  </div>;}
```

Атрибут key

- Не забудем добавить атрибут key:

```
function App() {
  const res = prods.map(function(item, index) {
    return <p key={index}>
      <span>{item.name}</span>:
      <span>{item.cost}</span>
    </p>;
  });

  return <div>
    {res}
  </div>;
}
```

№1

- В компоненте App дан следующий массив:

```
const users = [
  {id: 1, name: 'user1', surn: 'surn1', age: 30},
  {id: 2, name: 'user2', surn: 'surn2', age: 31},
  {id: 3, name: 'user3', surn: 'surn3', age: 32},
];
```

- Выведите элементы этого массива в виде списка ul.

Ключи через id

- В приведенном выше коде в атрибут key мы добавляли порядковый номер элемента в массиве. На самом деле такая практика является плохой и ей следует пользоваться лишь в крайнем случае.
- Дело в том, что при сортировке массива у элементов станут другие ключи и React не сможет правильно отслеживать связь между элементами массива и соответствующими тегами.
- Более хорошей практикой будет добавить каждому продукту уникальный идентификатор, который и будет использоваться в качестве ключа.
- Давайте в нашем массиве каждому продукту добавим свойство id с номером нашего продукта:

```
const prods = [
  {id: 1, name: 'product1', cost: 100},
  {id: 2, name: 'product2', cost: 200},
  {id: 3, name: 'product3', cost: 300},
];
```

Ключи через id

- Теперь в качестве ключа используем этот id:

```
function App() {
  const res = prods.map(function(item) {
    return <p key={item.id}>
      <span>{item.name}</span>:
      <span>{item.cost}</span>
    </p>;
  });

  return <div>
    {res}
  </div>;
}
```

№2

- Модифицируйте предыдущую задачу, добавив в массив id и использовав их в качестве значений атрибута key.

Вывод в таблицу

Вывод в таблицу

- Пусть у нас опять дан наш массив с продуктами:

```
const prods = [
  {name: 'product1', cost: 100},
  {name: 'product2', cost: 200},
  {name: 'product3', cost: 300},
];
```

- Давайте выведем элементы нашего массива в виде HTML таблицы:

```
function App() {
  const rows = prods.map(function(item) {
    return <tr key={item.id}>
      <td>{item.name}</td>
      <td>{item.cost}</td>
    </tr>;
  });

  return <table>
    <tbody>
      {rows}
    </tbody>
  </table>;}
```

Вывод в таблицу

- Добавим заголовки колонок нашей таблице:

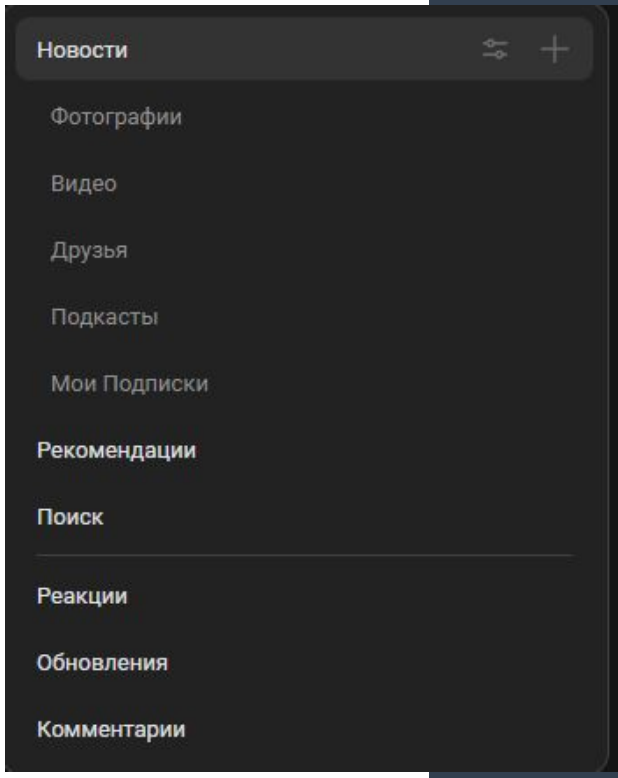
```
function App() {
  const rows = prods.map(function(item) {
    return <tr key={item.id}>
      <td>{item.name}</td>
      <td>{item.cost}</td>
    </tr>;
  });

  return <table>
    <thead>
      <tr>
        <td>название</td>
        <td>стоимость</td>
      </tr>
    </thead>
    <tbody>
      {rows}
    </tbody>
  </table>;
}
```

№1

- Сделайте проект «Новостная лента», это сайт представляющий собой поток информации, как лента в «ВК», нужно сделать его, используя 5 компонент

1. Меню (для ссылок использовать объект как в прошлом дз)
2. Категории (правое меню, для пунктов использовать массив)
3. Основная лента (просто место, куда будут помещаться все посты), содержит в себе только компоненты «Пост»
4. Пост (отдельный компонент, с фоткой и текстом, будет заполнять собой ленту)
5. Подвал (копирайт и год)

Для образца – использовать ленту в ВК



№1

- В компоненте App дан следующий массив:

```
const users = [
  {id: 1, name: 'user1', surn: 'surn1', age: 30},
  {id: 2, name: 'user2', surn: 'surn2', age: 31},
  {id: 3, name: 'user3', surn: 'surn3', age: 32},
];
```

- Выведите элементы этого массива в виде таблицы table так, чтобы каждое поле объекта попало в свой тег td. Сделайте заголовки колонок вашей таблицы.

Обсуждение id для объектов

id для объектов

- Как вы уже знаете, в массиве объектов должны присутствовать уникальные id. Давайте разберемся, откуда они берутся.
- Массивы объектов могут иметь два происхождения: либо они присланы нам сервером, либо сгенерированы на клиенте (то есть в браузере).
- Данные, присланные нам сервером, как правило хранились там в базе данных. Базы данных (БД) бывают вида SQL (например, mySQL, PostgreSQL) или NoSQL (например, MongoDB).
- SQL базы данных, как правило имеют числовые id, автоматически расставляемые базой данных по возрастанию.
- NoSQL базы данных, как правило, имеют id, представляющие собой случайные уникальные строки. Предполагается, что эти строки не имеют совпадений (коллизий) у двух элементов массива.
- Уникальность id достигается за счет достаточно большой длины случайных строк - настолько больших, что вероятность совпадения двух строк будет близка в нулю.
- При этом, чем больше данных в БД, тем больше вероятность коллизии. Задача программиста состоит в том, чтобы заранее прикинуть объем данных и определить оптимальную длину случайных строк, чтобы вероятность коллизий была минимальна (достаточно мала, чтобы считаться приемлемой).

Проблемы с добавлением данных

- Пусть из SQL базы данных нам пришел следующий массив объектов:

```
const prods = [
  {id: 1, name: 'product1', cost: 100},
  {id: 2, name: 'product2', cost: 200},
  {id: 3, name: 'product3', cost: 300},
];
```

- Как вы видите, id нумеруются по порядку. Однако, нам следует учитывать, что числа могут иметь пропуски, к примеру, после 3-го может сразу идти 5-тый или 6-той. Из-за этого наш клиентский скрипт не может знать, какой будет следующий id (в нашем случае это не обязательно 4).

- Новые id создаются базой данных сервера. Это может вызвать некоторые проблемы при работе на клиенте. Суть в следующем: представим, что с помощью формы мы добавили новый элемент в наш массив. Однако, мы не можем просто взять и добавить данные из формы - ведь мы не знаем какой id будет у нового элемента!

- Нам нужно будет отправить запрос на сервер, чтобы он дал нам следующий по порядку id, и только затем добавить элемент в наш массив объектов. Это вызовет задержку с отображением данных на экране: пока данные придут на сервер, пока сервер пришлет нам их обратно - пройдет некоторое время.

Id в виде строк

- Пусть теперь массив объектов имеет следующий вид:

```
const prods = [
  {id: 'GYi9G_uC4gBF1e2SixDvu', name: 'product1', cost: 100},
  {id: 'IWSpfBPSV3SXgRF87uO74', name: 'product2', cost: 200},
  {id: 'JAmjRlfQT8rLTm5tG2m1L', name: 'product3', cost: 300},
];
```

- Как вы видите, теперь наши id представляют собой случайные уникальные строки. Такой массив мог быть получен с NoSQL базы данных, либо просто сгенерирован на клиенте.

- Опять представим себе ситуацию с добавлением в этот массив данных формы. В этом случае задержки не возникнет - при добавлении данных мы просто сгенерируем случайную строку, которая и будет новым id.

Генерация id

- К счастью, нам не нужно самостоятельно придумывать код, генерирующий случайные id. К нашим услугам есть готовые библиотеки. Например, библиотека nanoid, генерирующая случайные строки, либо библиотека react-uuid, генерирующая UUID.

- Так как используемая библиотека для генерации id может быть разная, далее в уроках я буду использовать следующую функцию-оболочку:

```
function id() {
  // тут генерация id
}
```

- Предполагается, что эта функция генерирует id удобным вам способом.

Использование функции id()

- Мы можем использовать нашу функцию следующим образом при объявлении массива объектов:

```
const prods = [
  {id: id(), name: 'product1', cost: 100},
  {id: id(), name: 'product2', cost: 200},
  {id: id(), name: 'product3', cost: 300},
];
```

Неправильное использование id()

- Неправильно генерировать id прямо в атрибуте, вот так:

```
const res = prods.map(function(prod) {
  return <p key={id()}>
    <span>{prod.name}</span>
    <span>{prod.cost}</span>
  </p>;
});
```

- Причины этого будут вам понятны в следующих уроках. Пока просто имейте ввиду, что так делать неправильно и никогда так не делайте.

№1

- Установите и изучите обе библиотеки (nanoid, react-uuid). Сгенерируйте с их помощью случайные строки.

№2

- Реализуйте функцию id() с помощью библиотеки nanoid.

- Реализуйте функцию id() с помощью библиотеки react-uuid.

№3

- Сделайте массив объектов с юзерами, сгенерировав им id с помощью нашей функции.

Введение в стейты

Введение в стейты

- Следующая концепция, которую мы с вами разберем, называется стейты. Стейты представляют собой реактивные переменные компонентов.

- Реактивность означает, что при изменении стейта изменения произойдут во всех местах, где этот стейт используется. Технически это достигается путем перендерования всего компонента при изменении какого-либо стейта.

Преимущества

Манипулировать DOM (Document Object Model) вручную может быть сложно, трудоемко и чревато ошибками, в то время как использование состояний в React дает ряд преимуществ:

- **Абстракция**: React абстрагируется от сложности манипулирования DOM, предоставляя декларативный синтаксис для создания и обновления компонентов. Это облегчает рассуждения о состоянии приложения и изменениях, которые необходимо внести.
- **Повторное отображение**: Когда состояние компонента меняется, React автоматически перерисовывает компонент и обновляет представление компонента, чтобы отразить новое состояние. Это означает, что нам не нужно беспокоиться о ручном обновлении представления компонента или обработке событий и взаимодействий.
- **Виртуальный DOM**: React использует виртуальное представление DOM, называемое Virtual DOM, которое оптимизирует обновление компонента, сравнивая состояние и параметры компонента до и после, а затем React обновляет только тот компонент, который изменился, тем самым повышая производительность компонента.
- **Производительность**: Алгоритм виртуального DOM в React позволяет оптимизировать обновления и минимизировать количество необходимых манипуляций с DOM, что может помочь улучшить производительность приложения.

Начало работы

- Для использования стейтов для начала необходимо импортировать функцию **useState**:

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

- Функция **useState** параметром принимает начальное значение стейта, а своим результатом возвращает специальный массив из двух элементов. В первом элементе массива будет хранится текущее значение стейта, а во втором - функция для изменения стейта.

- Для корректной работы стейтов их нельзя менять напрямую, а следует пользоваться функцией для их изменения - только тогда будет работать реактивность.

Пример

- Воспользуемся функцией useState для создания стейта, содержащего название продукта:

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

- В результате константа **state** будет представлять собой массив, в первом элементе которого будет хранится название продукта, а во втором - функция для изменения названия:

```
const state   = useState('prod');
const name    = state[0]; // Переменная состояния, сейчас - prod
const setName = state[1]; // Функция для изменения состояния
```

Пример

- Для краткости обычно не используют промежуточную константу для массива, а используют деструктуризацию (да так можно):

```
const [name, setName] = useState('prod');
```

- Давайте теперь выведем наш стейт с именем продукта в какой-нибудь верстке:

```
return <div>
    <span>{name}</span>
</div>;
```

Пример

- Соберем все вместе и получим следующий код:

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

function App() {
    const [name, setName] = useState('prod');
    return <div>
        <span>{name}</span>
    </div>;
}

export default App
```

- Если запустить этот код, то изначально в диве выведется начальное значение стейта, в нашем случае 'prod'. При изменение стейта через функцию setName и верстке автоматически появится новое значение стейта.

Попробуем реактивность

- Сделаем кнопку, по нажатию на которую будет изменяться наш стейт:

```
return <div>
    <span>{name}</span>
    <button onClick={clickHandler}>btn</button>
</div>;
```

- В обработчике клика используем функцию setName, чтобы установить продукту новое название:

```
function clickHandler() {
    setName('xxxx');
}
```

Попробуем реактивность

- Соберем весь наш код вместе. Получится, что после нажатия на кнопку текст в спане мгновенно поменяется на новое значение:

```
function App() {
    let [name, setName] = useState('prod');

    function clickHandler() {
        setName('xxxx');
    }

    return <div>
        <span>{name}</span>
        <button onClick={clickHandler}>btn</button>
    </div>;
}
```

Попробуем реактивность

- Использование отдельных функций-обработчиков не обязательно. Можно использовать анонимную стрелочную функцию:

```
function App() {
    let [name, setName] = useState('prod');

    return <div>
        <span>{name}</span>
        <button onClick={() => setName('xxxx')}>btn</button>
    </div>;
}
```

Сделаем два стейта

- Давайте сделаем один стейт для названия продукта, а второй - для цены и кнопки для изменения наших стейтов:

```
function App() {
    let [name, setName] = useState('prod');
    let [cost, setCost] = useState('1000');

    function clickHandler1() {
        setName('xxxx');
    }
    function clickHandler2() {
        setCost('2000');
    }

    return <div>
        <span>{name}</span>
        <span>{cost}</span>

        <button onClick={clickHandler1}>btn1</button>
        <button onClick={clickHandler2}>btn2</button>
    </div>;
}
```

Сделаем два стейта

- Перепишем на сокращенный вариант:

```
function App() {
    let [name, setName] = useState('prod');
    let [cost, setCost] = useState('1000');

    return <div>
        <span>{name}</span>
        <span>{cost}</span>

        <button onClick={() => setName('xxxx')}>btn1</button>
        <button onClick={() => setCost('2000')}>btn2</button>
    </div>;
}
```

Логическое значение в стейте

- Давайте сделаем стейт **inCart**, который будет показывать, в корзине продукт или нет:

```
function App() {
    let [inCart, setInCart] = useState(false);

    return <div>

    </div>;
}
```

- Пусть значение **false** значит, что продукт не в корзине, а значение **true** - что в корзине. Выведем информацию об этом с помощью тернарного оператора:

```
return <div>
    <span>{inCart ? 'в корзине' : 'не в корзине'}</span>
</div>;
```

Логическое значение в стейте

- А теперь сделаем кнопку, по нажатию на которую продукт добавится в корзину:

```
return <div>
    <span>{inCart ? 'в корзине' : 'не в корзине'}</span>
    <button onClick={() => setInCart(true)}>btn</button>
</div>;
```

- А как сделать так, чтобы первое нажатие добавляло товар в корзину, а второе убирало?

Логическое значение в стейте

- Модифицируем наш код так, чтобы первое нажатие на кнопку добавляло в корзину, а второе - удаляло из нее:

```
function App() {
    let [inCart, setInCart] = useState(false);

    return <div>
        <span>{inCart ? 'в корзине' : 'не в корзине'}</span>
        <button onClick={() => setInCart(!inCart)}>btn</button>
    </div>;
}
```

Счетчик

- Давайте сделаем счетчик кликов по кнопке:

```
function App() {
    let [count, setCount] = useState(0);

    function clickHandler() {
        setCount(count + 1);
    }

    return <div>
        <span>{count}</span>
        <button onClick={clickHandler}>+</button>
    </div>;
}
```

- Как сделать сокращенный вариант?

Счетчик

- Можно избавится от функции-обработчика, заменив ее анонимной стрелочной функцией:

```
function App() {
    let [count, setCount] = useState(0);

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

№1

- 1. Пусть вы хотите отображать на экране данные юзера: его имя, фамилию, возраст. Сделайте для этого соответствующие стейты с начальными значениями.

- 2. Сделайте кнопки для изменения имени и фамилии.

- 3. Добавьте еще один стейт, который будет показывать, забанен пользователь или нет. Выведите информацию об этом в каком-нибудь теге.

- 4. Сделайте кнопку, нажатие на которую будет банить пользователя и кнопку, нажатие на которую будет разбанивать пользователя.

№2

- Модифицируйте предыдущую задачу так, чтобы из двух кнопок всегда была видна только соответствующая. То есть, если пользователь забанен, то видна кнопка для разбанивания, а если не забанен - для забанивания.

№3

- Сделайте еще две кнопки. Пусть первая кнопка увеличивает возраст на единицу, а вторая - уменьшает его.

Работа с инпутами

Работа с инпутами

- Работа с инпутами в React происходит с помощью стейтов. Каждому инпуту назначается свой стейт, содержащий в себе value инпута.

- Давайте посмотрим на примере. Пусть у нас есть инпут и стейт к нему, пока не связанные:

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

    return <div>
        <input />
    </div>;
}
```

Работа с инпутами

- Давайте к атрибуту value инпута привяжем нас стейт:

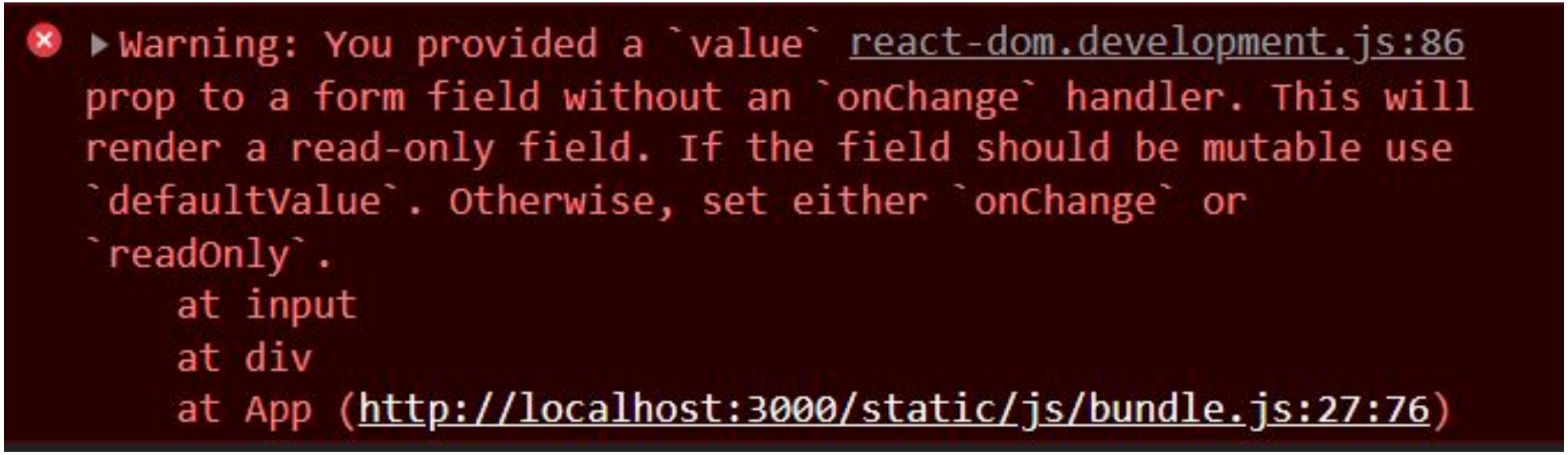
```
function App() {
    const [value, setValue] = useState('text');

    return <div>
        <input value={value} />
    </div>;
}
```

- В таком случае получится, что при изменении стейта, реактивно поменяется и текст инпута.

Работа с инпутами

- Это, однако, дает интересный побочный эффект: теперь при запуске кода в браузере в инпуте невозможно поменять текст! Почему: потому что при вводе текста в инпут не меняется стейт value, соответственно, и текст в инпуте не должен меняться.

- Откройте консоль браузера - вы увидите в ней предупреждение React. Это предупреждение указывает нам, что мы привязали стейт к инпуту, но тем самым заблокировали инпут.



Работа с инпутами

- Давайте сделаем так, чтобы в наш инпут можно было вводить текст. Для этого нужно сделать так, чтобы при вводе изменялся наш стейт на текущее значение инпута.

- Для начала для этого нужно навесить на инпут событие **onChange**:

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

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

- Данное событие в React ведет себя по-другому по сравнению с чистым JS. В React оно *срабатывает сразу же по изменению инпута*. То есть при вводе или удалении символа.

Работа с инпутами

- Давайте теперь добавим обработчик нашего события. В этом обработчике мы должны прочитать текущий текст инпута и установить его в стейт с помощью функции **setValue**.

- Проблема в том, что this данной функции не будет указывать на наш инпут - такова особенность React. Чтобы получить элемент, в котором случилось событие, нам необходимо использовать **event.target**:

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

    function handleChange(event) {
        console.log(event.target); // ссылка на DOM элемент инпута
    }

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

Работа с инпутами

- А теперь запишем текст инпута в наш стейт вытащив его из **event.target.value**:

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

    function handleChange(event) {
        setValue(event.target.value);
    }

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

Работа с инпутами

- Теперь мы сможем вводить текст в инпут. При этом стейт value всегда будет содержать текущий текст инпута.

- Мы можем легко убедится в этом. Выведем содержимое нашего текста в абзац. В этом случае при вводе текста в инпут введенный текст будет автоматически появляться в абзаце:

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

    function handleChange(event) {
        setValue(event.target.value);
    }

    return <div>
        <input value={value} onChange={handleChange} />
        <p>text: {value}</p>
    </div>;
}
```

Сокращаем

- Можем переписать на более компактный вариант с анонимной стрелочной функцией:

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

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

Изменение данных при выводе

- Пусть мы будем вводить в инпут число. Давайте сделаем так, чтобы по мере ввода числа в инпут в абзац выводился квадрат вводимого числа:

```
function App() {
    const [value, setValue] = useState(0);

    function handleChange(event) {
        setValue(event.target.value);
    }

    return <div>
        <input value={value} onChange={handleChange} />
        <p>{value ** 2}</p>
    </div>;
}
```

Используем функцию

- Не обязательно совершать некие операции над стейтом прямо на выводе. Можно воспользоваться функцией:

```
function square(num) {
    return num ** 2;
}

function App() {
    const [value, setValue] = useState(0);

    function handleChange(event) {
        setValue(event.target.value);
    }

    return <div>
        <input value={value} onChange={handleChange} />
        <p>{square(value)}</p>
    </div>;
}
```

Несколько инпутов

- Пусть у нас есть два инпута, в которые будут вводится числа. Давайте сделаем так, чтобы по мере ввода в абзац выводилась сумма двух инпутов:

```
function App() {
    const [value1, setValue1] = useState(0);
    const [value2, setValue2] = useState(0);

    function handleChange1(event) {
        setValue1(+event.target.value);
    }

    function handleChange2(event) {
        setValue2(+event.target.value);
    }

    return <div>
        <input value={value1} onChange={handleChange1} />
        <input value={value2} onChange={handleChange2} />
        <p>result: {value1 + value2}</p>
    </div>
}
```

Выводы

- Таким образом, для работы любого инпута нам нужно следующее: создать стейт для этого инпута, привязать стейт к атрибуту **value** инпута, навесить событие **onChange** на инпут, в обработчике события менять стейт инпута на его текст.

- Данные операции нужно будет проводить с каждым инпутом. То есть, если у вас два инпута, то у вас будет два стейта и две функции-обработчика события **onChange**.

№1

- Сделайте два инпута. Пусть текст первого инпута выводится в первый абзац, а текст второго инпута - во второй абзац.

№2

- Дан инпут. Дан абзац. Сделайте так, чтобы при вводе текста в инпут, в абзаце выводилось количество введенных в инпут символов.

№3

- Дан инпут и абзац. В инпут вводится возраст пользователя. Сделайте так, чтобы при наборе текста, в абзаце автоматически появлялся год рождения пользователя. (использовать функцию)

№4

- Дан инпут и абзац. В инпут вводятся градусы Фаренгейта. Сделайте так, чтобы при наборе текста, в абзаце автоматически выполнялась конвертация в градусы Цельсия. (использовать функцию)

№5

- Даны 5 инпутов. Сделайте так, чтобы при вводе чисел в наши инпуты в абзац выводилось среднее арифметическое введенных чисел.

Обработка данных формы

Обработка данных формы

- В предыдущем уроке мы делали так, чтобы при вводе символа в инпут в абзаце мгновенно появлялся результат. Это, конечно же, смотрится красиво, но имеет недостаток.

- Представим себе, что нам нужно сделать некоторую "тяжелую", ресурсоемкую операцию. Не очень оптимально делать ее на каждый ввод символа - лучше дождаться окончательного ввода данных и потом выполнить нужную операцию один раз и результат вывести в абзац.

- Для этого нам нужно ввести кнопку, по нажатию на которую будет совершаться наша ресурсоемкая операция. В таком случае у нас опять каждому инпуту будет соответствовать свой стейт, плюс еще один стейт нам нужен для записи результата операции и отображения его на экран.

Обработка данных формы

- Посмотрим на примере. Пусть у нас есть два инпута и кнопка. По нажатию на кнопку давайте найдем сумму чисел, введенных в инпуты.

- Реализуем сразу в сокращенном виде:

```
function App() {
    const [value1, setValue1] = useState(0);
    const [value2, setValue2] = useState(0);
    const [result, setResult] = useState(0);

    return <div>
        <input value={value1} onChange={event => setValue1(event.target.value)} />
        <input value={value2} onChange={event => setValue2(event.target.value)} />

        <button onClick={() => setResult(Number(value1) + Number(value2))}>btn</button>
        <p>result: {result}</p>
    </div>;
}
```

№1

- Даны два инпута, две кнопки и абзац. Пусть в инпуты вводятся числа. По нажатию на первую кнопку найдите сумму чисел, а по нажатию на вторую кнопку - произведение. Результат выводите в абзац.

№2

- Даны два инпута, кнопка и абзац. Пусть в инпуты вводятся даты в формате *31-12-2025*. По нажатию на кнопку найдите разницу между датами в днях и результат выведите в абзац.

№3

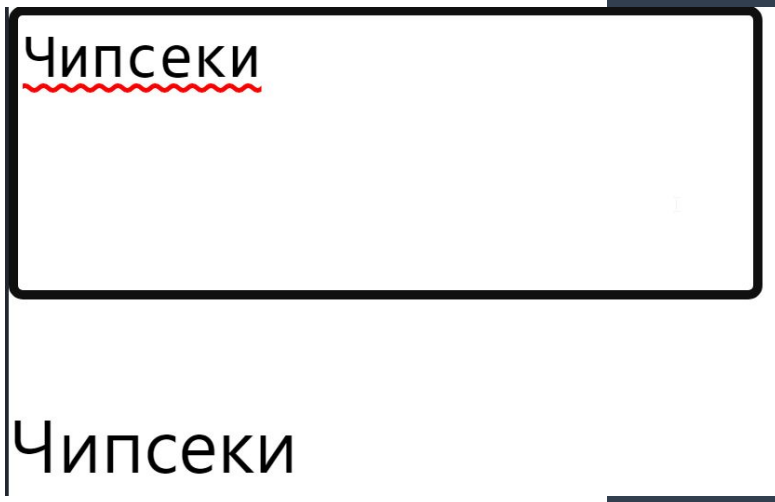
- Модифицируйте предыдущую задачу так, чтобы по умолчанию в инпутах стояла текущая дата.

Работа с textarea

Работа с textarea

- Давайте теперь научимся работать с многострочным полем ввода textarea. В React для удобства работа с ним сделана похожа на работу с текстовым инпутом. В отличие от чистого JS, в React в textarea не нужен закрывающий тег, а его текст следует размещать в атрибуте value.

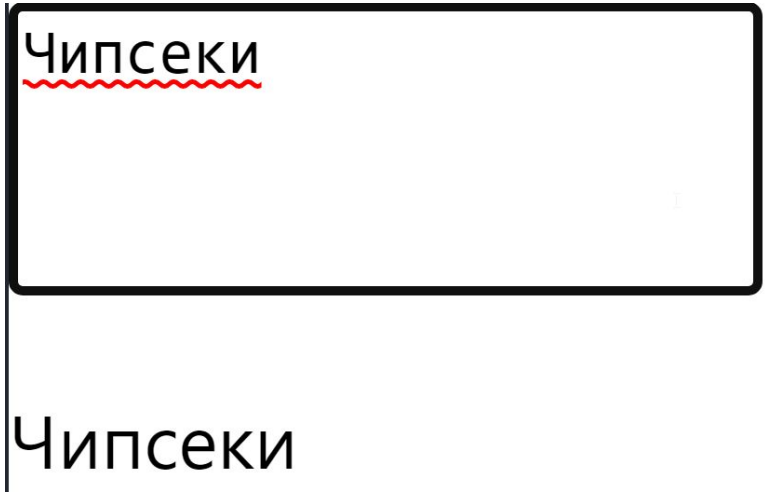
```
function App() {
    const [value, setValue] = useState('');

    function handleChange(event) {
        setValue(event.target.value);
    }

    return <div>
        <textarea value={value} onChange={handleChange} />
        <p>{value}</p>
    </div>;
}
```



Работа с textarea

- В сокращенной форме:

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

    return <div>
        <textarea value={value} onChange={event => setValue(event.target.value)} />
        <p>{value}</p>
    </div>;
}
```



Работа с чекбоксами

Работа с чекбоксами

- Работа с чекбоксами также осуществляется по схожему принципу, только вместо атрибута **value** мы указываем атрибут **checked**. Если в этот атрибут передать **true** - то чекбокс будет отмечен, а если **false** - не будет отмечен:

```
function App() {
    return <div>
        <input type="checkbox" checked={true}  /> отмечен
        <input type="checkbox" checked={false} /> не отмечен
    </div>;
}
```

Работа с чекбоксами

- Обычно в атрибут checked передается стейт, содержащий логическое значение:

```
function App() {
    const [checked, setChecked] = useState(true);

    return <div>
        <input type="checkbox" checked={checked} />
    </div>;
}
```

Работа с чекбоксами

- Так же, как и при работе с инпутами, если жестко задать значение атрибута checked - состояние чекбокса нельзя будет сменить. Для корректной работы будем по изменению чекбокса менять его стейт на противоположное ему значение:

```
function App() {
    const [checked, setChecked] = useState(true);

    return <div>
        <input type="checkbox" checked={checked} onChange={() => setChecked(!checked)} />
    </div>;
}
```

Работа с чекбоксами

❑Давайте выведем состояние чекбокса в абзац, используя тернарный оператор:

```
function App() {
    const [checked, setChecked] = useState(true);

    return <div>
        <input type="checkbox" checked={checked} onChange={() => setChecked(!checked)} />
        <p>состояние: {checked ? 'отмечен' : 'не отмечен'}</p>
    </div>;
}
```

Чекбоксы и условный рендеринг

- Давайте сделаем так, чтобы в зависимости от отметки чекбокса, на экран выводился либо один кусочек кода, либо другой. Используем для этого условный рендеринг:

```
function App() {
    const [checked, setChecked] = useState(true);

    let message;
    if (checked) {
        message = <p>сообщение 1</p>;
    } else {
        message = <p>сообщение 2</p>;
    }

    return <div>
        <input type="checkbox" checked={checked} onChange={() => setChecked(!checked)} />
        <div>{message}</div>
    </div>;
}
```

№1

- Дан чекбокс, кнопка и абзац. По клику на кнопку, если чекбокс отмечен, выведите в абзац текст приветствия с пользователем, а если не отмечен - текст прощания.

№2

- С помощью трех чекбоксов попросите пользователя выбрать языки, которые он знает: html, css и js. Результат выбора по каждому языку выводите в отдельные абзацы.

№3

- Дан чекбокс. С помощью чекбокса спросите у пользователя, если ли ему уже 18 лет.

  Если чекбокс отмечен, покажите пользователю следующий блок кода:

```
<div>
    <h2>Ура, вам уже есть 18</h2>
    <p>
        здесь расположен контент только для взрослых
    </p>
</div>
```

- А если чекбокс не отмечен - то следующий:

```
<div>
    <p>
        увы, вам еще нет 18 лет:(
    </p>
</div>
```

№4

- Дан чекбокс и абзац. Если чекбокс отмечен, пусть абзац будет видимым на экране, а если не отмечен - спрячьте его.

Работа с селектами

Работа с селектами

- Давайте теперь займемся выпадающими списками select. Работа с ними также практически не отличается от работы с инпутами и чекбоксами.

- Пусть у нас есть вот такой селект:

```
function App() {
    return <div>
        <select>
            <option>text1</option>
            <option>text2</option>
            <option>text3</option>
            <option>text4</option>
        </select>
    </div>;
}
```

Работа с селектами

- Давайте обеспечим его работу средствами React:

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

    function handleChange(event) {
        setValue(event.target.value);
    }

    return <div>
        <select value={value} onChange={handleChange}>
            <option>text1</option>
            <option>text2</option>
            <option>text3</option>
            <option>text4</option>
        </select>
        <p>
            ваш выбор: {value}
        </p>
    </div>;
}
```

Атрибуты value

- Пусть теперь у нас в тегах option есть атрибуты value:

```
function App() {
    return <div>
        <select>
            <option value="1">text1</option>
            <option value="2">text2</option>
            <option value="3">text3</option>
        </select>
    </div>;
}
```

Атрибуты value

- В таком случае из-за наличия атрибутов **value** в стейт будут попадать именно их значения, а не тексты тегов **option**. Можно убедится в этом, выведя результат выбора в абзац:

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

    return <div>
        <select value={value} onChange={(event) =>
setValue(event.target.value)}>
            <option value="1">text1</option>
            <option value="2">text2</option>
            <option value="3">text3</option>
        </select>
        <p>
            ваш выбор: {value}
        </p>
    </div>;
}
```

Атрибуты value

- Разделить текст **option** и его значение может быть удобно: текст тега мы можем менять как нам угодно, при этом в нашем скрипте результат выбора будет обрабатываться по значению атрибута **value**, которые останется неизменным.

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

    return <div>
        <select value={value} onChange={event => setValue(event.target.value)}>
            <option value="1">text1</option>
            <option value="2">text2</option>
            <option value="3">text3</option>
        </select>
        <p>
            {value === '1' && 'вы выбрали первый пункт'}
            {value === '2' && 'вы выбрали второй пункт'}
        </p>
    </div>;}
```

- Теперь, если мы изменим тексты **option**, то работа скрипта не нарушится - ведь она привязана к значению атрибута value.

№1

- Сделайте выпадающий список городов. Сделайте также абзац, в который будет выводиться выбор пользователя.

№2

- С помощью выпадающего списка предложите пользователю выбрать к какой возрастной группе он относится: **от 0 до 12 лет, от 13 до 17, от 18 до 25, либо старше 25 лет**.
- В зависимости от выбранной группы вывести на страницу: **Младшая группа, Средняя группа, Взрослая группа, Пожилая группа.**

Работа с radio

Работа с radio

- Работа с радиокнопочками radio несколько отличается, к примеру, от тех же чекбоксов. Проблема в том, что у нескольких радио будет один и тот же стейт, но разные value.

- Поэтому работа происходит следующем образом: каждой радиокнопке в атрибут value записывают свое значение, а в атрибут checked - специальное условие, которое проверяет, равен ли стейт определенному значению. Если равен - радиокнопочка станет отмеченной, а если не равен - будет не отмеченной.

Работа с radio

- Вот реализация описанного:

```
function App() {
    const [value, setValue] = useState(1);

    function changeHandler(event) {
        setValue(event.target.value);
    }

    return <div>
        <input type="radio" name="radio" value="1"
            checked={value === '1' ? true : false}
            onChange={changeHandler}
        />
        <input type="radio" name="radio" value="2"
            checked={value === '2' ? true : false}
            onChange={changeHandler}
        />
        <input type="radio" name="radio" value="3"
            checked={value === '3' ? true : false}
            onChange={changeHandler}
        />
    </div>
}
```

№1

- Даны 3 радиокнопки. Дан абзац. Сделайте так, чтобы значение выбранной радиокнопки выводилось в этот абзац.

№2

- С помощью радиокнопок спросите у пользователя его любимый язык программирования. Выведите его выбор в абзац. Если выбран язык JavaScript, похвалите пользователя.

Практика со стейтами

Практика со стейтами

- Создайте простой компонент переключателя, который отображает кнопку и сообщение. При нажатии на кнопку сообщение должно измениться с "включено" на "выключено" и наоборот. Сообщение должно выводиться разными цветами. Используйте state для отслеживания текущего состояния.

Практика со стейтами

- Создайте простой компонент смены темы, который позволяет пользователям включать и выключать светлую тему. Используйте state для отслеживания текущего состояния темы.

Практика со стейтами

- Создайте простой компонент светофора, который отображает текущее состояние светофора с помощью state и переключается по кнопке.

Практика со стейтами

- Создайте простой компонент калькулятора, который позволяет пользователям выполнять основные арифметические операции с использованием состояния.

Практика со стейтами

- Создайте викторину: Создайте компонент викторины, который отображает серию вопросов и отслеживает оценку пользователя. Компонент должен использовать состояние для отслеживания текущего вопроса, ответов пользователя и итоговой оценки.