REACT. Начало

Установка

create-react-app

1.1 Установка Node.js

Заходим на сайт https://nodejs.org/en/, устанавливаем последнюю актуальную LTS версию, на момент записи урока это `16.13.0 LTS` Совет. Если у вас уже установлен node.js - рекомендуем проверить его версию командами: **`node -v` и `npm -v`.**

1.2 Установка утилиты create-react-app

Установка утилиты create-react-app с помощью которой мы будем создавать проект для react приложения. https://ru.reactjs.org/docs/create-a-new-react-app.html#create-react-app
В консоли вводим команду:
npm install -g create-react-app

1.3 Создание react приложения

В консоли перемещаемся в директорию внутри которой вы собираетесь создать react приложение (родительскую директорию). Рекомендуется выбрать папку так чтобы в ней или в пути к ней **не было кириллических символов**. Идеально создать папку в корне диска и назвать ее на латинице. Например: **C:\\REACT_PROJECTS**. Переместившись в нее, запускаем команду:
npx create-react-app start-app
Это означает что мы создадим новое приложение в папке **start-app** и наше приложение будет иметь название **start-app**.

1.4 Запуск react приложения

Чтобы запустить react приложение, вначале перемещаемся в консоли в его директорию:
cd start-app
и после запускаем командой
npm start (старт приложения **в режиме разработки**, команда записана в package.json)

Ctrl-C - остановить приложение

Установка node-sass

В новом терминале:
npm i node-sass

Настройка VS Code и Chrome

VS Code. Плагин

Установка плагина Simple React Snippets: https://marketplace.visualstudio.com/items?itemName=burkeholland.simple-react-snippets

VS Code. Настройка Emmet
Параметры - в поиске набрать **emmet**. Ищем Emmet:Include Languages. Добавляем элемент **javascript** со значением **javascriptreact**
Emmet будет работать с react

Плагин для Chrome браузера

Плагин для браузера React Developer Tools:
https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi

Старт работы с новым react приложением

Создаем новое приложение или открываем созданное ранее. Закидываем директорию с приложением в редактор VS Code.
src/ - js сборка. Она подключается в publuc/imdex.html
src/index.js - главный файл в сборке, куда подключаются библиотеки, компоненты, стили, изображения и др.

1. Удаляем все файлы в папке **public** кроме **index.html** и **favicon.ico**. Очищаем файл index.html. Его итоговое наполнение:

```
<!DOCTYPE html>
<html lang="en">
```

```
    <head>
        <meta charset="utf-8" />
        <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>React App</title>
    </head>
    <body>
        <noscript>You need to enable JavaScript to run this app.</noscript>
        <div id="root"></div>
    </body>
</html>
```

2. Удаляем все файлы из папки src кроме файла **index.js**. В index.js оставляем только необходимый код. Для 18-ой версии react это выглядит так:

```
import ReactDOM from "react-dom/client";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
 <div>
  <h1>Hello, world</h1>
 </div>
);
```

root.render - рендер приложения. У него обязательно должен быть **один** родительский тег.

Автообновление корректно работает когда мы правим другие импортируемые *.js файлы в наше приложение, но не index.js. Когда правим index.js - обновлять страницу надо вручную. Иногда достаточно просто обновить страницу, а иногда нужно остановить и перезапустить сборку.

Перенос верстки

Из html файла переносим информацию из head - заголовок, стили. Обратим внимание на тег body - есть ли у него свои стили. Переделаем html комментариии в **{/* */}**

Знакомство с JSX

JSX это смесь html и JavaScript в одной строке. Мы можем писать html разметку смешивая её с JS кодом. JS код необходимо помещать внутрь фигурных скобок **{ ... }** Также есть небольшое различие в написании двух html атрибутов.

В JSX есть особенности в написании двух html **атрибутов**.

- Вместо `**class**` пишем `className` .
- Вместо `**for**` пишем `htmlFor` .
- Вместо `**autocomplete**` пишем `autoComplete` .

Пример:

```
root.render(
 <div>
  <h1 className="green">Hello, world!</h1>
  <label htmlFor="name">Enter your name:</label>
  <input type="text" id="name" />
 </div>
);
```

Также для одиночных тегов (например, img) везде сделать закрытие />

JS код внутри JSX

Подобно шаблонным строкам, в JSX можно использовать JS код. Выводить значение переменных либо результаты работы функций, а также описывать логические блоки.

Через JSX можно выводить:

- переменные (строки, числа)
- массивы (массивы интерполируются в строки)
- свойства объектов (если в них записаны строки, числа, массивы)
- запускать функции и выводить результат их работы
- выполнять логические и другие конструкции языка
- объекты в виде значений для атрибутов (атрибут style). Значения свойств - **camelCase!**

- **теги** внутри переменных в JSX **трансполируются**.

Через JSX нельзя выводить:

- объекты в разметку
- булевые значения true / false. Не выводятся, но они могут быть использованы в условиях

Примеры использования JSX в разметке:

```
const name = "Yurij";
const dayNumber = 25;
const month = "April";
const boolean = true;
const array = [1, 2, 3, 4, 5];

const person = {
 name: "Peter",
 age: 30,
};

const headingStyle = {
 color: "coral",
 fontSize: "32px",
 fontFamily: "sans-serif",
};

const siteUrl = "https://webcademy.ru";
const script = "<script>alert('Error!')</script>";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
 <div>
  <h1 style={headingStyle}>Hello, {name}!</h1>  // Hello, Yurij!
  <label htmlFor="name">Enter your name: </label>
  <input type="text" id="name" />

  <p>Today is {dayNumber} of {month}{" "} </p>  // Today is 25 of April
  <p>One value from array: {array[0]}</p>  // One value from array: 1
  <p>Array to string: {array}</p>   // Array to string: 12345

  <p>{person}</p>   // Objects are not valid as a React child - error
  <p>Person name: {person.name}</p>   // Person name: Peter

  <p>Function result: {Math.random()}</p>   // Function result: 0.47873483948592876
  <p>Logical expressions: {boolean ? "Thath true!" : "Oh, it's false"}</p>  // Logical expressions: Thath true!
  <p><a href={siteUrl}>Go to WebCademy</a>{" "} </p>   // Go to WebCademy - ссылка
  <p>{script}</p>   // <script>alert('Error!')</script> - скрипт не работает
 </div>
);
```

Компоненты

Разобьем наше приложение на отдельные **компоненты**. Вначале мы познакомимся с функциональными компонентами (компоненты созданные с помощью функций), позже в уроках мы затронем и работу с компонентами созданными на основе классов. В своей базовой структуре функциональный компонент выглядит следующим образом (ffc-tab):

```
function ComponentName() {
 return (  );
}
export default ComponentName;
```

Или (sfc-tab):

```
const ComponentName = () => {
 return (  );
}
export default ComponentName;
```

Если хотим вернуть только одну строку, то скобки можно опустить. Компонент должен возвращать **один** тег-обертку. Названия компонентов пишутся с **заглавной буквы**. Можно все компоненты вложить в один - **App**

В компонентах можно использовать **не только вывод разметки**, но там может быть и логика и др.

Чтобы использовать компонент в рендере или внутри других компонентов его надо вызвать подобно тегу:
<ComponentName />

Пример:
```
const Header = () => {
 return <h1>Список дел</h1>;
};

const List = () => {
 return (
  <ul>
   <li>Проснуться пораньше</li>
   <li>Сделать зарядку</li>
   <li>Выпить кофе</li>
   {/*<li>Написать React приложение</li>*/} - комментарии
  </ul>
 );
};

const Footer = () => {
 return (
  <footer>
   <input type="text" placeholder="Новая задача" />
   <input type="submit" value="Добавить" />
  </footer>
 );
};

const App = () => {
 return (
  <div>
   <Header />
   <List />
   <Footer />
  </div>
 );
};

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);
```

Для создания компонента можно воспользоваться **сниппетом ffc** tab (функциональный компонент из Simple React Snippets):
```
function () {
 return (  );
}

export default ;
```

Элементы

Элементы, это кусочки JSX разметки, самые маленькие структурные блоки приложения. Создадим несколько JSX элементов, запишем их в переменные. Необязательно расписывать всю разметку вплоть до элементов записывая их в перменные. Обычно достаточно ограничиться выводом разметки внутри return компонента. Но иногда необходимо предварительно создать элемент, и уже после использовать его в рендере.

```
const List = () => {
    const item1 = <li>Проснуться пораньше</li>;  - без кавычек!
    const item2 = <li>Сделать зарядку</li>;
    const allItems = (
        <ul>
            {item1}
            {item2}
```

```
            <li>Выпить кофе</li>
            <li>Написать React приложение</li>
        </ul>
    )
    return allItems;
};
```

или

```
const List = () => {
 const items = ["Проснуться пораньше", "Сделать зарядку", "Выпить кофе", "Написать React приложение"];
 const render = items.map((el) => <li>{el}</li>);
 return <ul>{render}</ul>;
};
```

Вывод списка делается. Есть один момент, однако сейчас в консоли мы увидим предупреждение:
Each child in a list should have a unique "key" prop.
Эта ошибка связана с Reconciliation алгоритмом внутри React.

Вывод списков и Reconciliation алгоритм

При выводе списков и других итерируемых элементов, к каждому такому элементу, необходимо добавить постоянное уникальное значение для атрибута **key**.

```
const List = () => {
 const items = [
  { id: 0, title: "Проснуться пораньше" },
  { id: 1, title: "Сделать зарядку" },
  { id: 2, title: "Выпить кофе" },
  { id: 3, title: "Написать React приложение" },
 ];
 const render = items.map((el) => <li key={el.id}>{el.title}</li>);
 return <ul>{render}</ul>;
};
```

key - без кавычек, в разметке не выводится

Обновление страницы - достаточно дорогостоящаа по ресурсам опреация. Поэтому реакт старается обновлять только те элементы на странице которые изменились, и не трогать те которые остались неизменными. Процесс поиска изменений называется **recancallation алгоритмом**.

Чтобы реакт понимал какие именно элементы были обновлены, для них стоит добавлять атрибут key с уникальным значением для каждого элемента. Таким образом react сможет сравнить старые и новые элементы. И в результате будет добавлен только один элемент, и больше ничего перерисовано не будет.

По умолчанию, когда свойство key отсутствует, react сам добавляет ключи по порядку. При изменениях меняются тогда все элементы. По той же причине не стоит использовать индексы элементов массива как значения для свойства key, так как индексы могут поменяться при, н-р, вставке элемента в начало массива.

Логика и функционал внутри компонентов

Пример1:

```
const UserLogin = () => {
 const isLoggidIn = true;
 const loggedIn = <p>Hello, User!</p>;
 const notLoggedIn = <a href="#!">Log in</a>;
 return isLoggidIn ? loggedIn : notLoggedIn;
};

const App = () => {
 return (
  <div>
   <UserLogin />
   <Header />
   <List />
   <Footer />
  </div>
 );
};
```

Пример2:

```
const UserGreeting = () => {
  return <p>Hello, User!</p>;
};

const UserLogin = () => {
  return <a href="#!">Log in</a>;
};

const App = () => {
  const isLoggidIn = false;
  return (
    <div>
      {isLoggidIn ? <UserGreeting /> : <UserLogin />}
      <Header />
      <List />
      <Footer />
    </div>
  );
};
```

Передача свойств (props) внутрь компонентов

Компоненты при вызове могут принимать свойства, как их принято называть - **props** и использовать полученные из них значения в своей работе. Имя для props мы придумываем самостоятельно. Значение также может быть любым: строки, числа, массивы, объекты, функции...

```
<UserGreeting name="Yurij" />
```

Чтобы использовать props внутри функционального компонента, необходимо в параметрах функции объявить объект props и после обращаться к имени переданного свойства как к свойству данного объекта.

```
const UserGreeting = (props) => {
        return <p>Hello, {props.name}!</p>;
}
```

Отдельные файлы для React компонентов

Пришло время вынести код каждого компонента в отдельный файл. К тому же, при изменении index.js не идеально работает автообновление. А вот при правке других импортируемых в него файлов, автообновление будет работать лучше.

Создаем в **src** папку **components**, в ней будут находится файлы компонентов. Для каждого файла создать файл с таким же названием в папке components - Header.js и др. Перенесем туда код компонента и сделаем экспорт:

```
const Header = () => {
  return <h1>Список дел</h1>;
};

export default Header;
```

Для App.js сделаем импорты:

```
import Header from "./Header";
import UserGreeting from "./UserGreeting";
import UserLogin from "./UserLogin";
import List from "./List";
import Footer from "./Footer";

const App = () => {
  const isLoggidIn = false;
  return (
    <div>
      {isLoggidIn ? <UserGreeting name="Yurij" /> : <UserLogin />}
      <Header />
      <List />
      <Footer />
    </div>
  );
```

```
};

export default App;
```

index.js:

```
import ReactDOM from "react-dom/client";
import App from "./components/App";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);
```

CSS стили в react приложении

Допустим мы хотим стилизовать div в котором находится наше приложение. Даем ему CSS класс **app**:

```
const App = () => {
    return (
        <div className="app">
            ...
        </div>
    );
};
```

1 вариант. Далее описываем данный CSS класс. Либо в теге style в index.html:

```
<style>
        .app {
            background-color: rgb(229, 242, 248);
            padding: 30px;
            font-family: sans-serif;
            line-height: 1.5;
        }
</style>
```

2 вариант. Либо в отдельном CSS файле public/main.css который подключен к index.html

index.html

```
<link rel="stylesheet" href="./main.css">
```

main.css

```
.app {
  background-color: rgb(229, 242, 248);
  padding: 30px;
  font-family: sans-serif;
  line-height: 1.5;
}
```

3 вариант. Базовый и самый распространённый способ добавлять CSS оформление - это создавать .css файл **для каждого компонента** и импортировать его в компонент.

Создадим css файл components/App.css

```
.app {
  background-color: rgb(229, 242, 248);
  padding: 30px;
  font-family: sans-serif;
  line-height: 1.5;
}
```

Импортируем App.css в App.js

```
import "./App.css";
```

Импорт изображений

Добавим файл с svg иконкой в папку components/logo.svg.
Далее импортируем его в App.js

```
import icon from "./logo.svg";
```

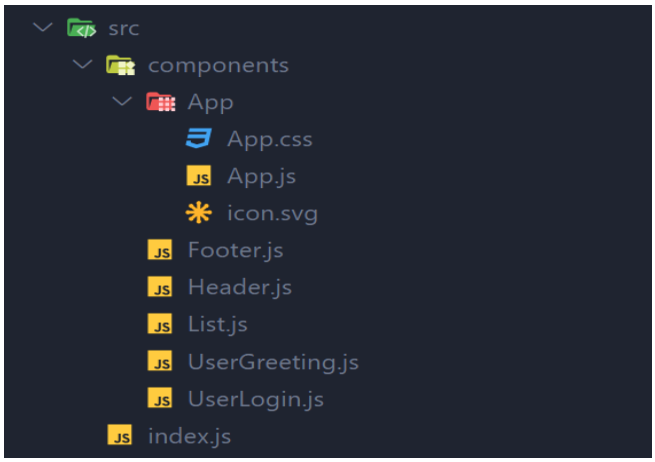
И выведем на страницу используя тег <img> и используя имя, под которым импортировали:

```
<div className="app">
  <img src={icon} alt="" />
  ...
</div>
```

Отдельные директории для компонентов

По мере роста приложения, у каждого компонента могут появиться дополнительные JS, CSS и другие файлы. Поэтому удобно будет создавать отдельные папки для каждого компонента. При изменении структуры файлов и директорий будьте внимательны чтобы пути в импортах оставались корректными.

На текущий момент создадим папку App внутри components и разместим в неё все файлы которые относятся к компоненту App. Структура папки src будет выглядеть следующим образом:



index.js:

```
import App from './components/App/App';
```

Пути для импорта других компонентов в файле **App.js**

```
import Header from './../Header';
import UserLogin from './../UserLogin';
import UserGreeting from './../UserGreeting';
import List from './../List';
import Footer from './../Footer';
```

Перенос верстки в случае с sass и отдельными папками под блоки (корзина с товарами)

Удаляем из **src** все лишнее. Правим index.js и App.js.

Создадим src/components/App. Перенесем и переименуем расширение для App.js - src/components/App/App.jsx. Перенесем src/components/App/_base.scss, _reset.scss, _vars.scss, _section-cart.scss Подключим их в **App.jsx**

```
import "./_vars.scss"
import "./_reset.scss"
import "./_base.scss"
import "./_section-cart.scss";
```

Переносим в App.jsx всю верстку из index.html. Меняем **class** на **className.** Закроем все одиночные теги img и input

Создадим папку для заголовка - src/components/Tiitle. Создадим в ней Title.jsx. В нем scf-tab:

```
const Title = () => {
  return <h1 className="title-1">Корзина товаров</h1>;
};
export default Title;
```

Подключим его к App.jsx. В Title.jsx подключим для него стиль _titles.scss и переименуем его _style.scss. Можем также переименовать Title.jsx в index.jsx Тогда в импорте в App.jsx можно указать только папку:

```
import Title from "../Title";
```

Создадим src/components/Cart/Cart.jsx. Все подключим.

Создадим src/components/CartHeader/index.jsx и style.scss. Все подключим.

Создадим src/components/Ptoduct/index.jsx и style.scss. Все подключим.

Создадим src/components/CartFooter/index.jsx и style.scss. Все подключим.

Подключим файл с переменными внутри стилей для некоторых компонентов, добавим в стилях style.scss сверху строку:

```
@import "../App/vars";
```

Создадим для счетчика src/components/Count/index.jsx и style.scss. Все подключим.

Создадим для кнопки удаления src/components/ButtonDelete/index.jsx. Все подключим.