**Курс от Владелин Минин.**

**1.Теория. Что такое React.**

**1.1. Что такое React**

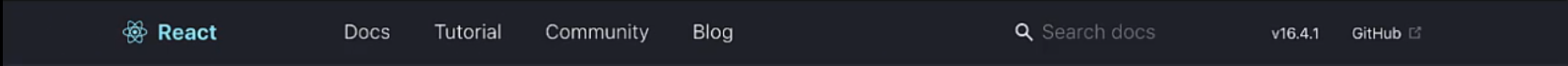
Это отличная технология для построения сложных и динамичных веб-приложений.

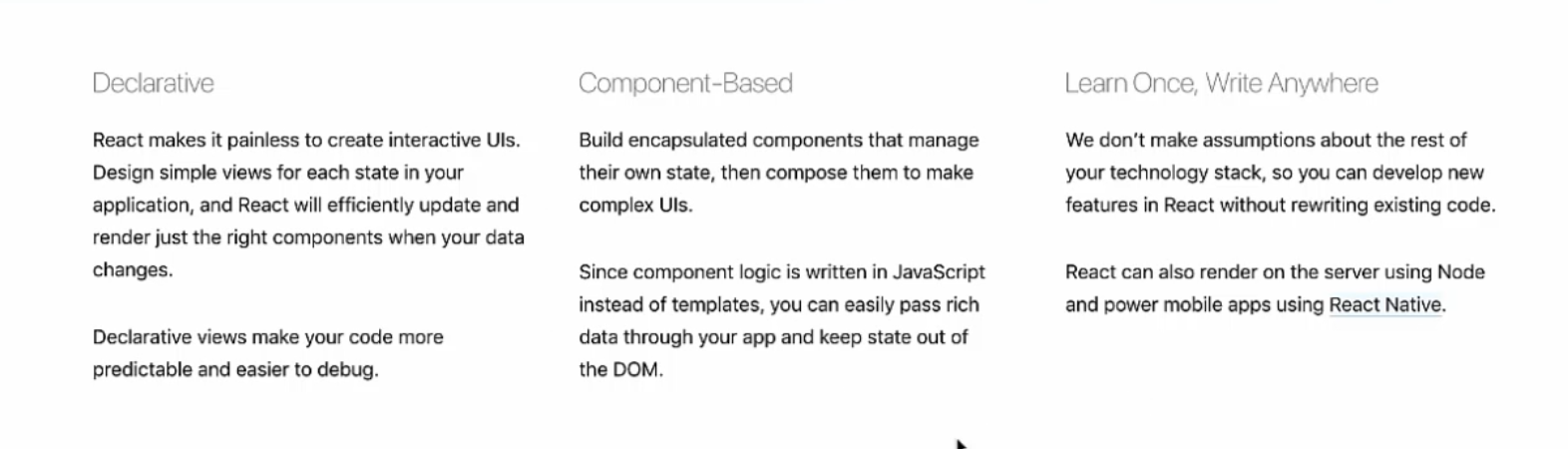
**React –** это JavaScript библиотека, для построения пользовательских интерфейсов. Пользовательский интерфейс другими словами, это интерфейс сайта (веб-приложения). Reactjs позволяет строить сложные и реактивные пользовательские интерфейсы.

Одно из основных свойств react это то, что он основан на компонентах. Другими словами, все элементы, которые есть в реакте делятся на компоненты. Компоненты можно будет пере использовать или по-разному кастомизировать.

Что же такое компонент? Возьмем пример сайта reactjs.org, который сам написан на реакте. Завабно.

К примеру, навигационное меню, это отдельный компонент:





Три этих блока с текстом это один и тот же компонент, только с разным содержанием.

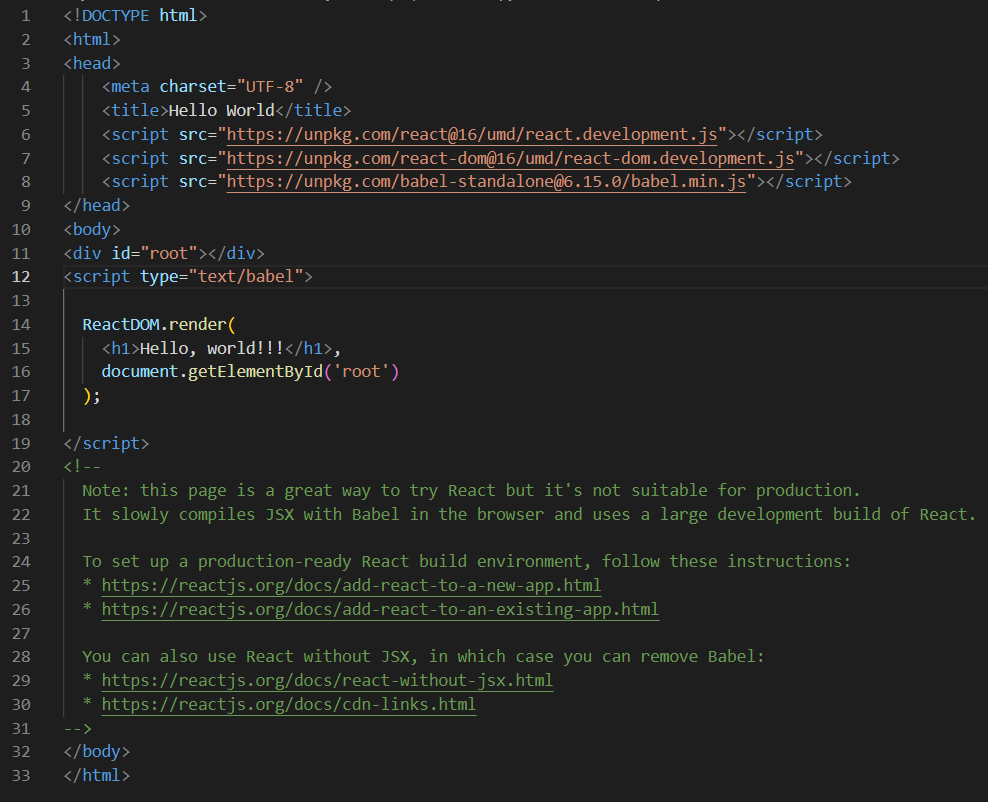
**react** это собственность компании **facebook** это видно даже в ссылке на github, хотя facebook уже давно переименовался в **meta.**



**1.2 Как работает React**

В курсе имеется html файл, на котором есть минимальное, чтобы запустить react в браузере. Но этот не является примером для подражения. **Только для примера**.

Вот этот код:



Разберем подробнее этот код

Обычный html файл с подключенными 3-мя скриптами. Однако само ядро реакта состоит из 2 библиотек. Кстати, напомню, что такой способ подключения библиотек называется **cdn(**удаленный сервер**).**



В этой строке подключается 16 версия реакта, версия development версия, это версия библиотеки с различными комментариями с не минифицированным кодом (не уменьшенным), который позволит лучше понимать, что конкретно происходит в данном приложении. Это библиотека по факту сам **React.**



Второй скрипт подключает так называемый **react-dom**. Тоже 16 версия. Он позволяет работать с обычным **html** и делать различные манипуляции над ним. Тоже версия для разработки.

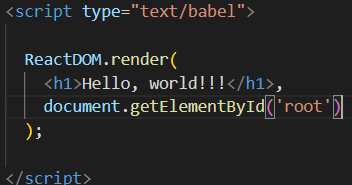
**2 Этих библиотеки и составляют ядро реакта.**

3-й Скрипт **babel-standalone** не относится напрямую к реакту



Подключенный **babel.min.js** нужен чтобы преобразовывать js код прямо налету.

Ниже есть скрипт, который описывает по факту всю страницу.



И если обратить внимание, то скрипт подключен через атрибут **type=”text/babel”.**

Эта строка сообщает, что перед запуском данного скрипта, его нужно будет скомпилировать.

Если пока без подробностей, то **React** поддерживает некоторый синтаксис, который называется **JSX.** Он позволяет описывать некоторый **html** прямо внутри **JS.**  В коде выше, как раз видно **html** тег заголовка с hello, world. Прямо внутри JS кода, в методе .**render** без каких либо скобок, спец. символов или кавычек видно обычный html код.

Этот специальный синтаксис **React** и является его ключевой особенностью от других Фреймворками. О **JSX** подробнее будет позже.

Но что нужно знать, так это то, что такой JS нативно в браузере не запустится и вызовет ошибку.

Поэтому в этом коде и подключается **babel,** который преобразует данный код в то, что будет понимать браузер.

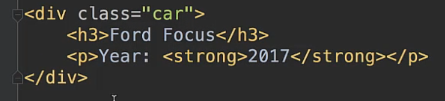
Открыв этот файл в браузере можно увидеть,



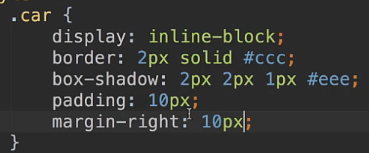
Что внутри тега <**div>**  c id **root,**  появился тег <**h1>** с содержимым **hello world.** Хотя изначально его там не было. Далее более подробно будут рассмотрены **React** компоненты.

**1.3. Что такое компоненты**

Допустим в коде будет такой **div** описывающий авто.



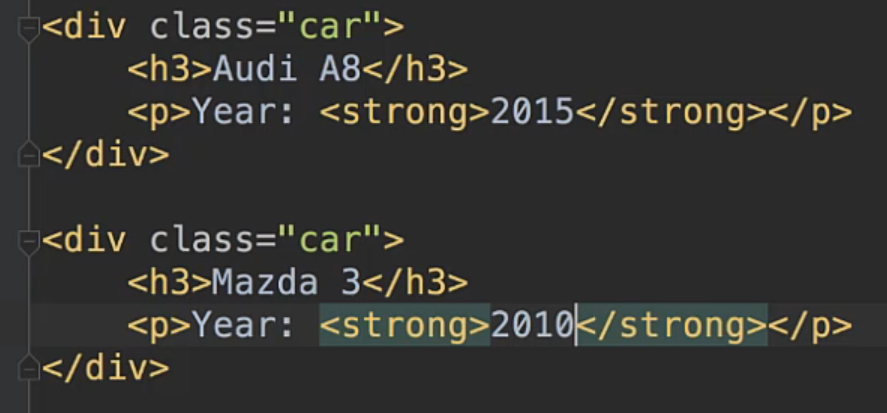
С такими стилями.



Выглядеть этот блок будет так:



Если нужно создать несколько таких блоков, но с другими машинами, нужно создавать новые блоки.



Блоки создадутся, но код на странице заметно увеличится в размерах.

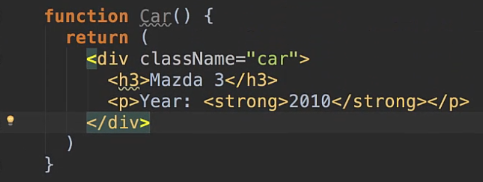


У этих блоков одинаковая структура, но разное содержимое. Эти блоки можно назвать **компонентами.**

Реакт и построен на подобных компонентах. Он позволяет создавать некоторую функцию, которая будет возвращать структура определенного компонента и после этого уже можно будет настраивать каждый из них.

Как тоже самое можно написать уже через Реакт?

Сначала попробуем написать обычную **js** функцию **Car(),** которая будет возвращать такой же компонент.



**Важно обратить внимание,** что содержимое **return** написано внутри **(скобок).**

И второе, вместо **class=’car' и**спользуется **className=’car’. className** необходимо писать из-за того, что в **js** уже есть такое ключевое слово как **class.**

Как можно догадаться, функция **car** В данном случае является простым **react** компонентом.

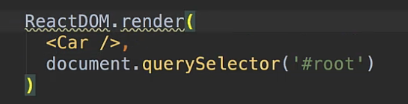
Чтобы внедрить эту функцию в html нужно:

обратится к библиотеке: **ReactDom,** чтобы заносить какие-то данные в html и вызвать метод **render,** с помощью которогои происходит инъекция какого-нибудь компонента реакта в **html.**

Внутри функции нужно указать 2 параметра. **Первый,** это Реакт компонент, который представляет нашу функцию. Но чтобы его положить нужно превратить его в **html** тег. Для этого название функции нужно просто обернуть в скобки и закрыть **<НазваниеФункции />.**

**Второй параметр,** это место куда нужно вставить компонент. Это может быть тег, или **id** например.

**document.getElementById(‘root’)** или проще: **document.querySelector(‘#root’)**



В результате функция отработает и выведет:

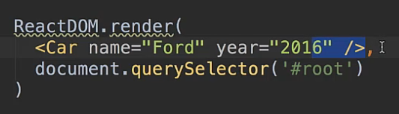


А как сделать так, чтобы в эту функцию можно будет задавать данные другие машины.

Пусть функция в качестве аргумента принимает некоторый **props.** И уже через название этого аргумента, принимает параметры: **props.name** и **props.year**

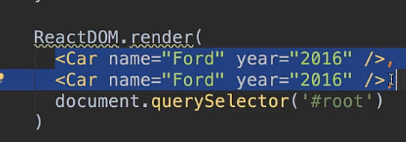


А в методе **render** эти поля вызываются, как обычные html атрибуты.



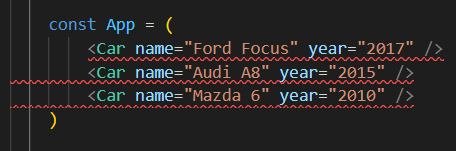
**Но таким образом можно засунуть только один компонент.**

То есть так нельзя:

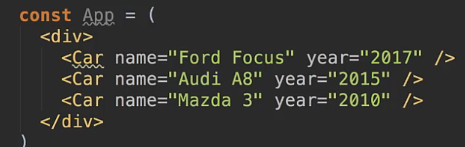


**Суть реакта в том, что любые компоненты, должны быть обернуты в какой-то корневой элемент. В метод рендер можно заносить только ОДИН элемент.**

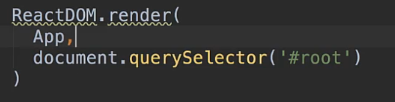
Поэтому можно создать новую переменную, которая будет хранить в себе все эти компоненты с разными параметрами. **Единственное, что нужно отметить, в переменную нельзя просто вставить 3 этих компонента.**



**Необходимо обернуть эти компоненты, в какой-то html. Например в div.**



Так, как переменная **App** уже содержит некоторый **JSX** код, то при вызове функции **render** оборачивать переменную в **</> не надо.**



**Create React App.**

**Create-react-app** – это инструмент, который позволяет создавать приложения основываясь на реакте, без предварительной настройки.

Перед тем как ее установить нужно установить **node.js**

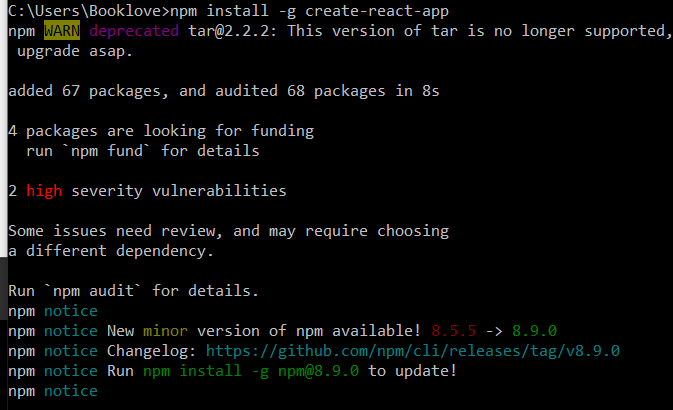
Чтобы проверить установлен ли он можно в **cmd** вбить: **node –v**



При установке **node.js** автоматически устанавливается и **npm,** это менеджер управления пакетами.

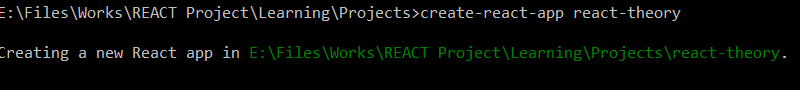


**Если все установлено,** то можно глобально установить **Create-react-app.**

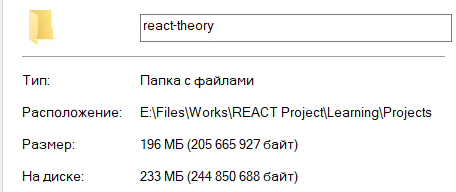


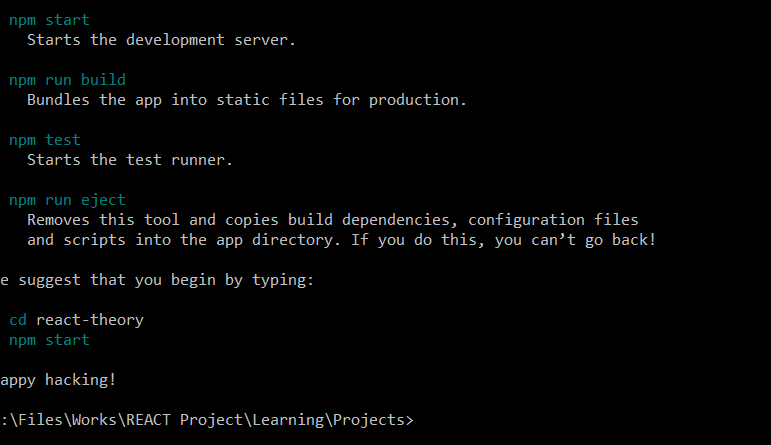
**Теперь с помощью этой технологии можно создать проект.**

Для этого нужно перейти в командной строке в папку, где нужно создать проект. А затем прописать команду: **create-react-app названиеПроекта**

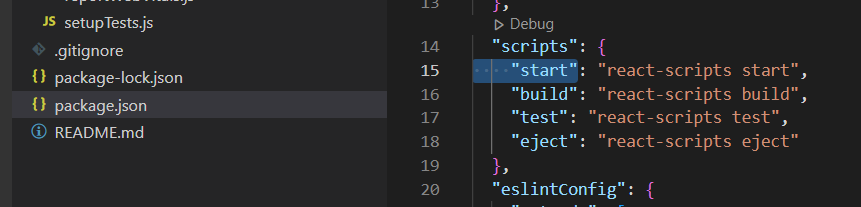


И проект появится в этой папке:





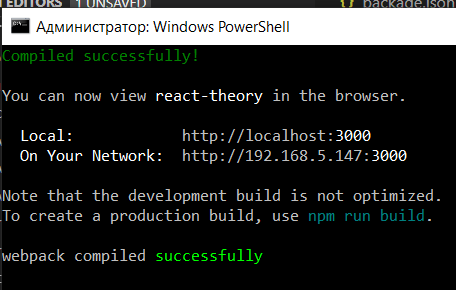
Если открыть проект в текстом редактор и найти там файл **package.json**: Другие команды будут



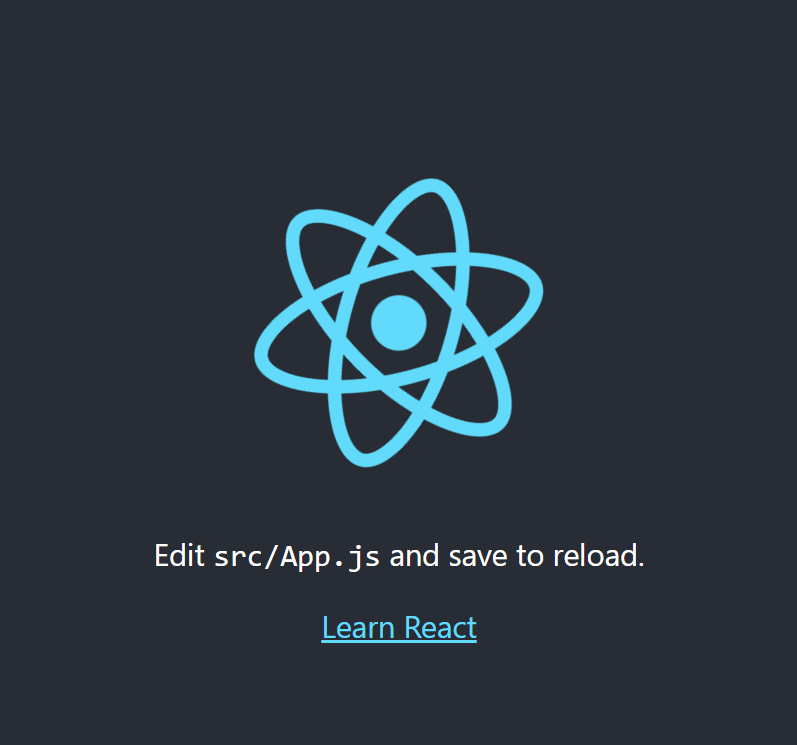
В нем можно понять, как запустить проект (Если start заменить другим словом, то будет работать с другим словом, я проверил).

Переходим в папку проекта в командной строке и пишем команду : **npm run start**

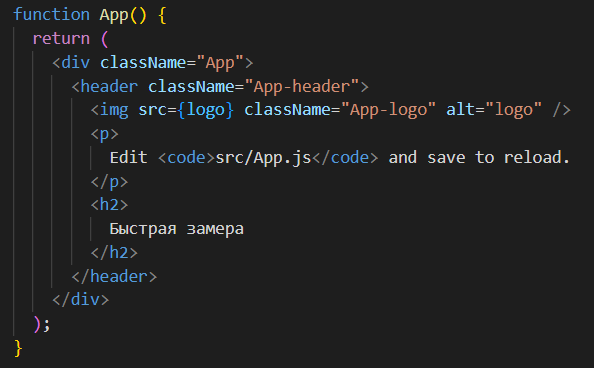
**И проекта запустится.**

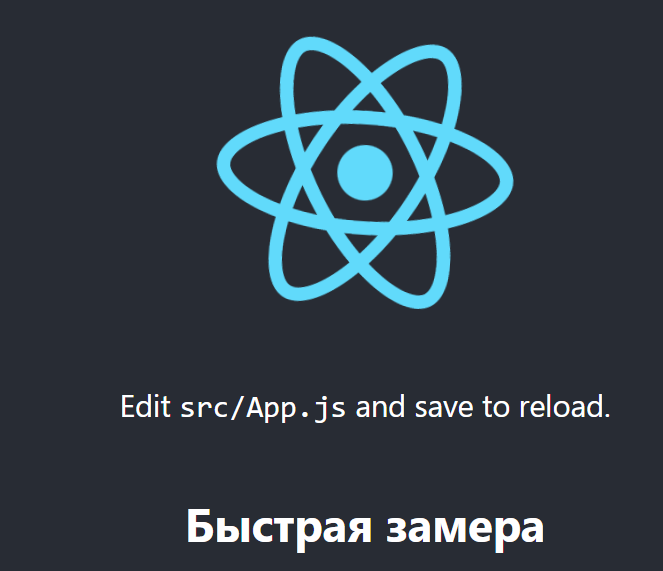


**После этого по этим 2-м IP можно открыть страницу**

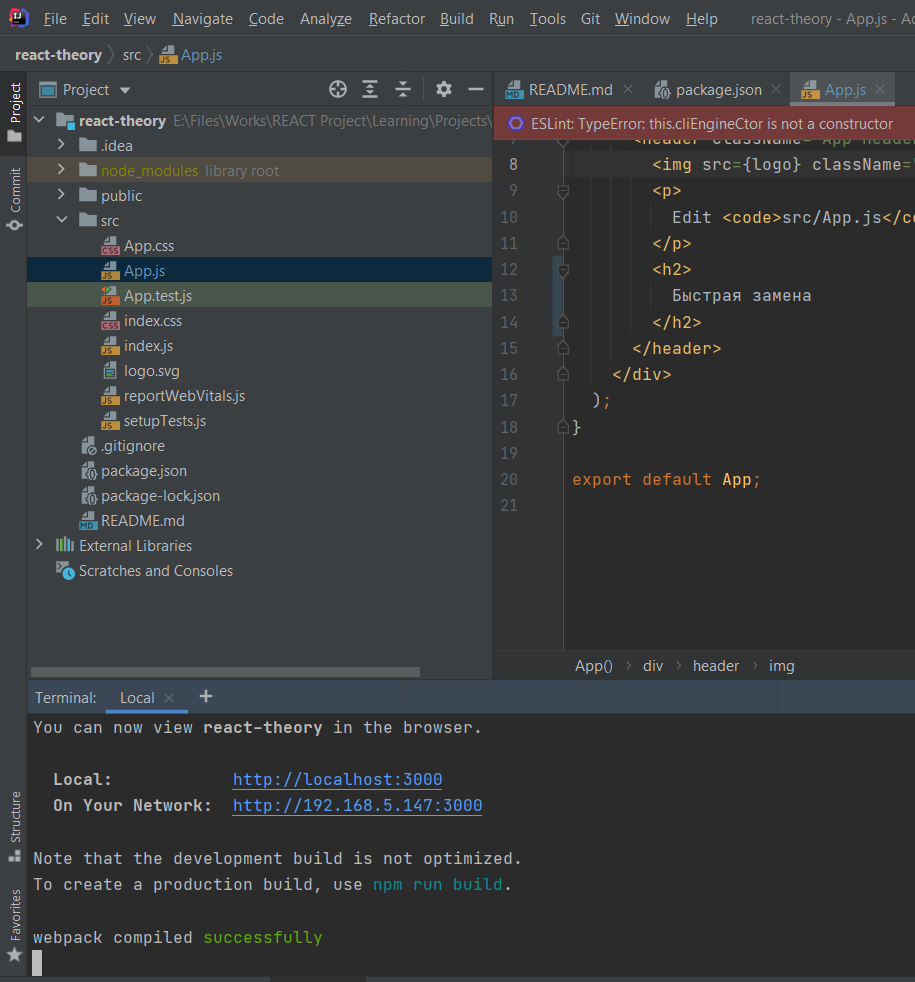


**Изменяя файл App.js** можно сразу видеть результат.





И все же проще использовать редактор со встроенной консолью.



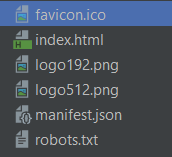
**Обзор приложения.**

**package.json** это автоматически сгенерированный файл. Он содержит название проекта. Его версию. И зависимости подключенных библиотек.

Вместо npm можно будет использовать другую технологию под названием yarn. **Поэтому эту технологию желательно тоже установить.**



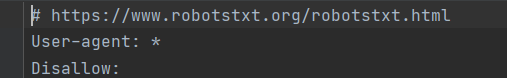
В папке **public хранятся несколько** файлов.



favicon.ico это иконка, которая видна на самой вкладке

Два logo это картинка, которая крутится на странице. Разница в них это разрешение.

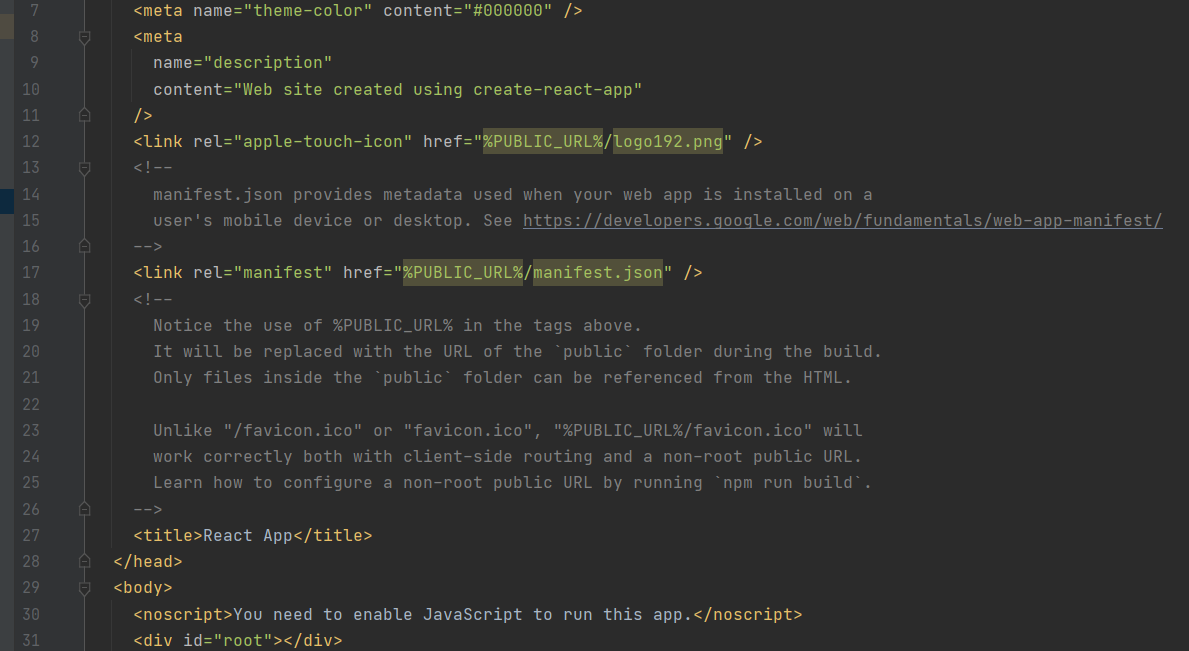
**robots.txt** без понятия



**manifest.json** хранит некоторую информацию, возможно связанную со стилизацией картинок.



**index.html**



Основная страница проекта. Здесь не видно где подключен скрипт. А еще здесь использует уникальная конструкция: **%PUBLIC\_URL%** о ней может быть позже. Но самое главное, на этой странице есть блок с id=“**root**”

С помощью реакта по факту мы разрабатываем **Single page application.** Одностраничное приложение. И индекс html и является такой страницей.

Скрипт **noscript** отрабатывает если выключен **javascript.**

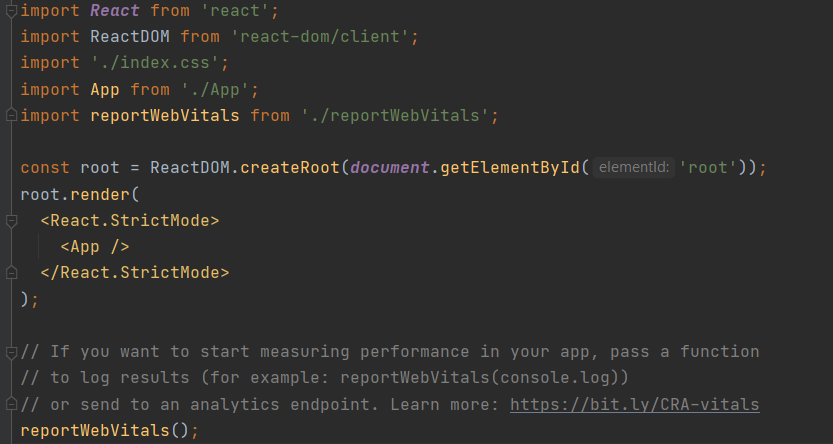
Вернемся к файлу **manifest.json.** Он превращает наше приложение в **progressive application.**

Самая базовая настройка. Об этом не будет подробнее (

**Папка src**

В основном вся работа будет в этой папке.

**index.js**



Здесь уже используется синтаксис ES6. Здесь позволено свободно использовать классы JS, import и стрелочные функции и тд. Реакт все уже настроил.

Первые 2 строки импортируют в проект библиотеку React и React-dom.

Так здесь импортируются стили.



А за ними импортирован первый компонент.

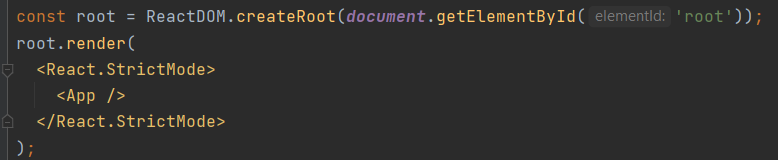


И сюда также импортирована функция **registerServiceWorker.** Сказал я, но у меня импортирован какой-то **reportWebVitals.** Но наверно это одно и тоже.

Затем в курсе продемонстрирована такая строка:



У меня вместо нее



Сложно что-то сказать.

Первая строка создает root методом **.createRoot(id)** Если обратится к этой константе, то внутри будет:



Но это уже после рендера.

Зачем нужны строки

<React.StrictMode>

Тоже хз. Мб вместо div.

Другим способом, который в курсе написать не получилось. Смог только сократить код до такого.



Файл App.js можно сказать и является тем приложением, который виден в браузере.

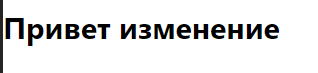
В этом проекте реализован **life reloading.** То есть изменении кода, сервер автоматические обновляет содержимое страницы.

Из папки **src** понял одно. Перемещать можно все файлы, но только в пределах **src,** за **src** обратится не могу. И файл **index.js** лучше не перемещать.

**2. Теория. JSX-синтаксис.**

Команда root может принимать не только функцию, которая как бы возвращает компонент. Но и самый обычный html код





**Насчет разницы с курсом. Чтобы использовать метод ReactDOM.render также как в курсе**



**Библиотека ReactDOM должна быть подключена так:**



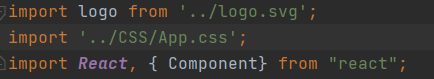
**А не так:**

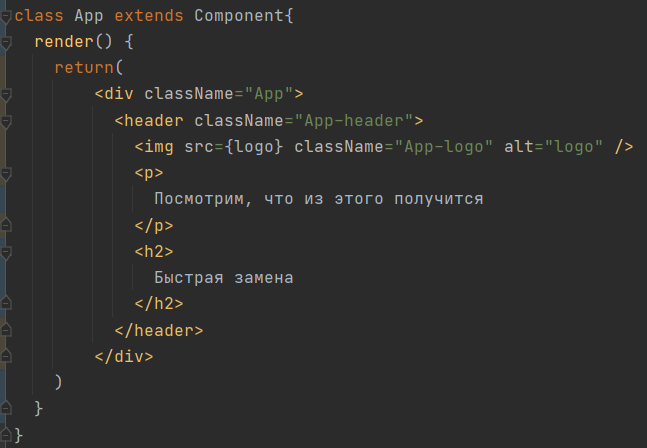


Помимо того, что можно писать внутри функции **html** код, в нем можно превращать некоторые функции или **React компоненты в** компоненты. Ну то есть условно класс **App** превратить в компонент и таким образом его использовать.



Вернемся к файлу **App.js** в курсе код записан следующим образом:

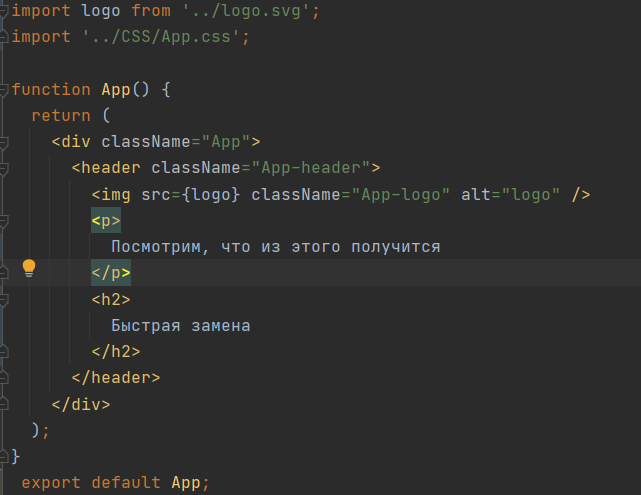




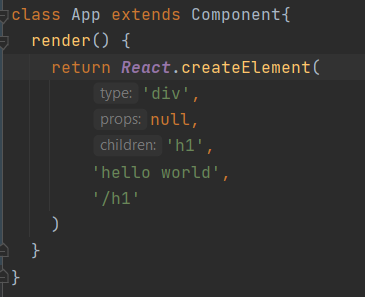


**Причем третий import важен. Без него не заработает.**

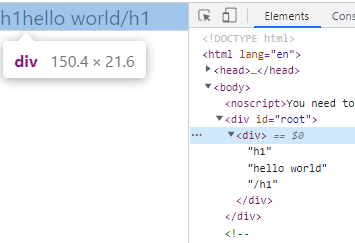
А по умолчанию в файле компонент создавался так:



На самом деле, html код внутри компонента выглядит не так. Попробуем написать в том виде, как он на самом деле прописан.

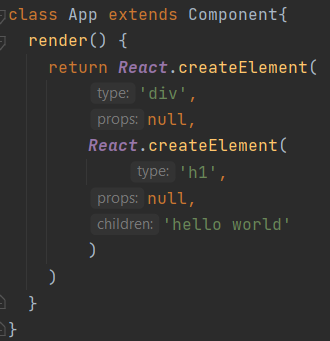


Небольшой пример. Если зайти в браузер. То можно увидеть, что внутри root действительно создался div с содержимым h1, hello world /h1. Но сам тег h1 не стал заголовков.



Как это исправить? Все html элементы создаются через строку **React.createElement().**

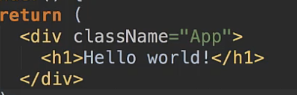
**div** находится первым аргументом в этой функции, поэтому он интерпретировался. С h1 нужно сделать тоже самое. Первый аргумент – **тип html элемента.** Второй аргумент – опции **props.** Третий аргумент - содержимое или контент.



Но как туда добавить классы? Как раз-таки вместо **null** вторым параметром в скобках можно добавить атрибуты. Правда **style** добавить не вышло пока что.



Так вот. То что я вижу и пишу в качестве **JSX** на самом имеет такой вид. То есть после интерпретации это равно **этому:**

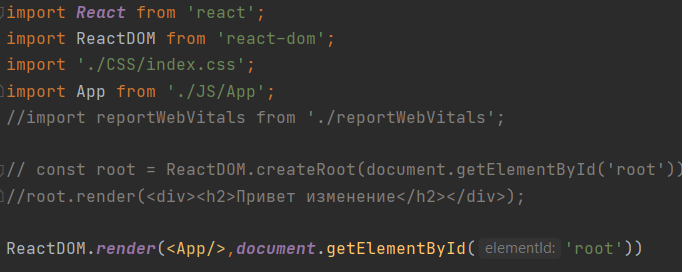


Если где-то в проекте используется синтаксис **JSX.** То, крайне важно, чтобы в этой части проекта был **import React from ‘react’**



**Даже если в коде нет прямого обращения к React.**

Пример страницы **index.js**



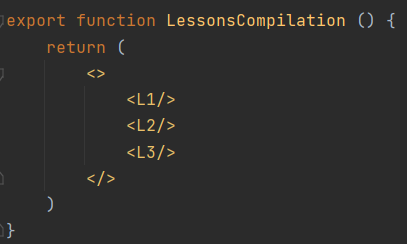
**Правда на моей версии проекта если закомментировать первую строку проект не сломается. Потому что скорее всего где-то еще React импортирован. Я вернулся. И полон сил. Чуть переделал проект. Будет следующая страница с уроками:**

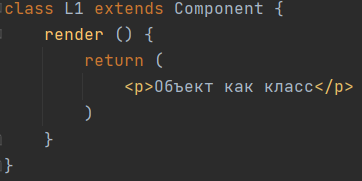
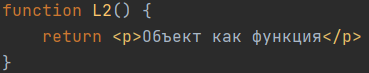
**App куда я буду импортировать компоненты для их изучения.**

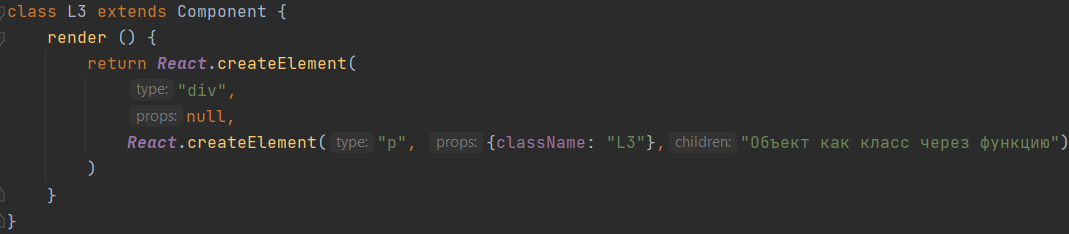


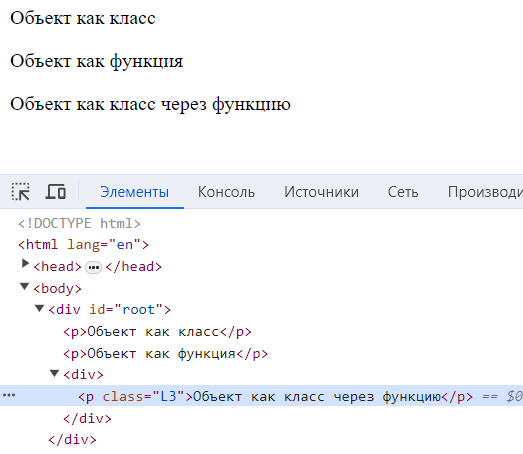
**LessonsCompilation включает в себя все тестовые компоненты, которые я пробую создавать.**

**На данный момент их 3**





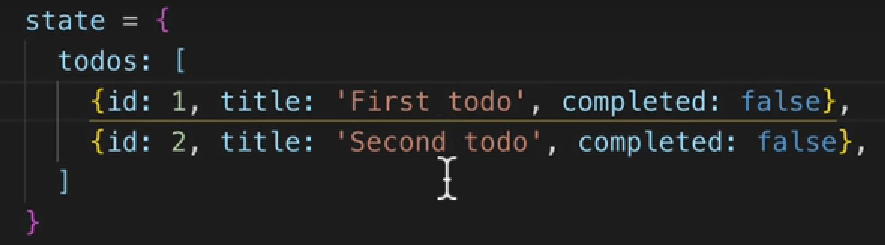


**Внезапное отступление. На Ютубе есть быстрый плейлист по React Хукам от него. Быстро по нему пройдусь.**

[**https://www.youtube.com/watch?v=hQRPsKzpo34&list=PLqKQF2ojwm3n6YO3BDSQIg35GGKn\_ImFD**](https://www.youtube.com/watch?v=hQRPsKzpo34&list=PLqKQF2ojwm3n6YO3BDSQIg35GGKn_ImFD)

**Я уже в курсе много из этого, но так правильно.**

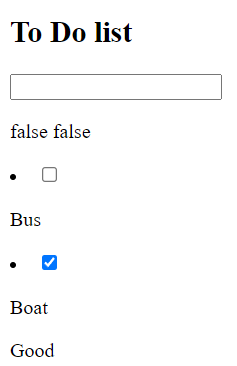
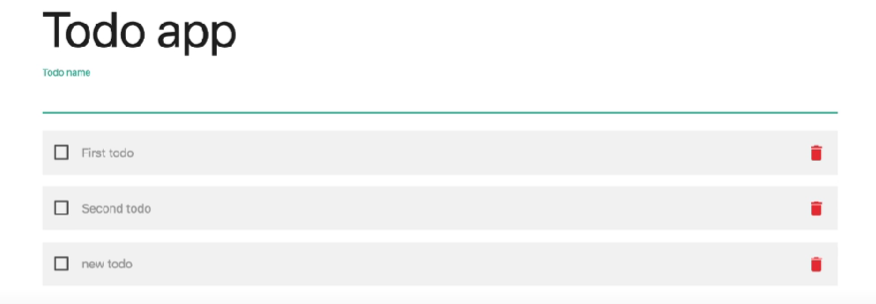
**Вот как раньше в классовом компоненте, нужно было задать и передать state.**





**Теперь для этого просто используют useState.**

**Задание себе. Быстро написать аналог вот такого компонента.**



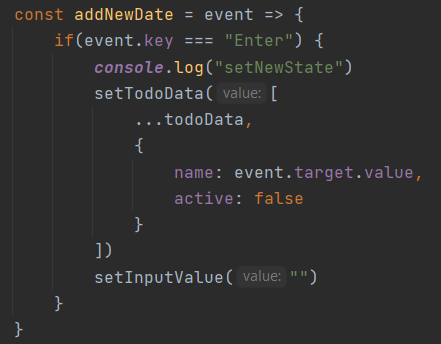
**Я сделал быстренько уродливый вариант.**

**Эти false, false. Это состояния списка в родителе. Good добавляется в дочернем компоненте при нажатом чекбоксе.**

**В варианте из видео у каждого чекбокса свое состояние и независимо его изменения будут влиять только на состояние этого компонента чекбокса. То есть общему родителю не известно о изменении состояния.**

**Но есть интересный код с обработки нажатия Enter, чтобы добавить новый элемент в список**





**UseEffect. Я уверен, что нарушаю кучу правил прописывая там setState.**

**Так вот. Начнем с того, что удалим стандартное значение по умолчанию у компонента.**

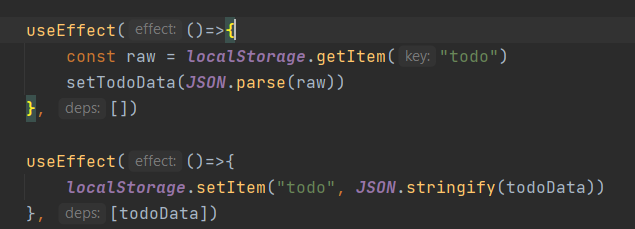


**Теперь при старте страницы список будет пустым. Его можно заполнить, но если снова обновить страницу, он опять будет пустым. С помощью useEffect можно это исправить… без сарказма, рил?**

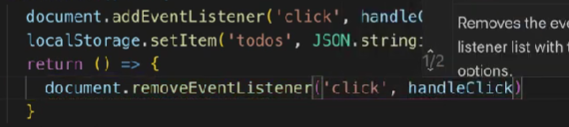
**Благодаря нему можно реализовывать react lifecycle hooks в компонентах. Первым параметром принимается Callback, который будет выполнятся. А вторым список зависимостей, на которые будет откликаться useEffect.**

**Если в useEffect оставить [ ] пустой список зависимостей, то будет производится эмуляция хука componentDidMount. Который вызывается тогда, когда html шаблон готов для работы. А если поместить туда, какое-то состояние, то callback будет вызывать при каждом изменении этого state.**

**Хотя, не. Я не супер сильно удивлен. Хотя он разделил useEffect на два разных. В одном он устанавливал стартовое состояние и вызывается он только 1 раз, при первой загрузке. А в другом уже брал состояние из локалки.**



**Однако, если на втором useEffect добавить обработчик кликов например. То чем чаще будет вызван этот useEffect, тем больше будет утечка памяти, т.к. на каждый вызов будет добавляться новый обработчик событий. Утечки памяти можно будет избежать, добавив в return удаление созданного обработчика, когда компонент будет закрываться.**

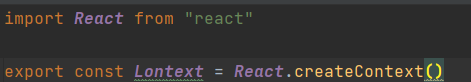


**Теперь о хуке, который я не использовал. Буду рад ему научится.**

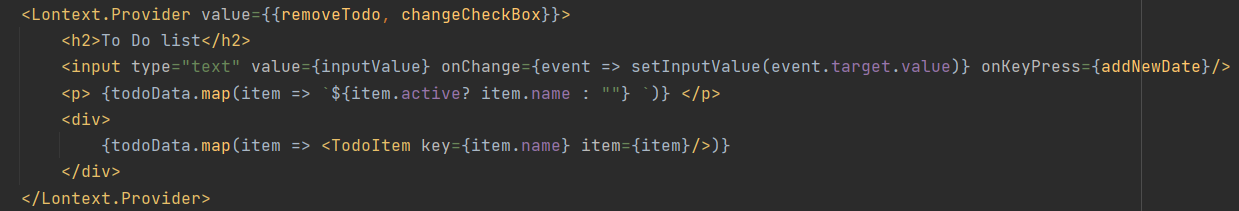
**Вот допустим мы отмечаем чекбокс в компоненте todoItem, но после перезагрузки состояние сбрасывается. Я уже писал выше почему.**

**Можно пробросить его в основной компонент. Но есть другой вариант, использовать контекст.**

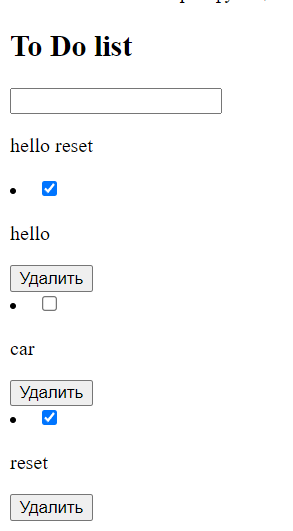
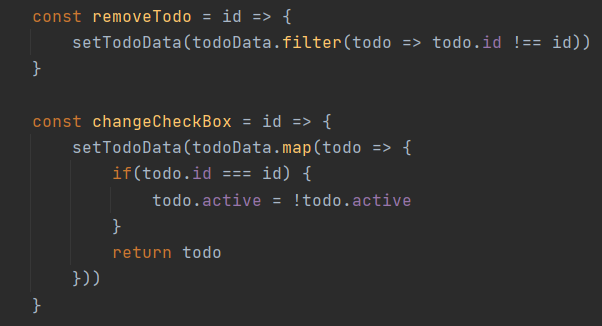
**Реально классная вещь. Для начала его нужно инициализировать. Я специально перепутал первую букву, чтобы показать, как эта переменная используется.**



**Затем обернуть родителя, контекст которого нужно будет передать. В обертку нужно в значение value поместить функции, которые будут переданы дочернему элементу.**



**Вот эти функции.**



**А теперь дочерний элемент просто должен их себе вызвать через useContext и константу.**

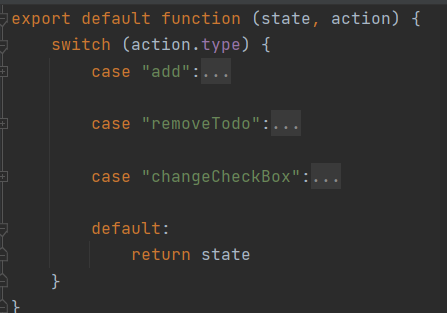


**И все, дочерний элемент может изменять состояние родителя. Классно. И все сохраняется после перезагрузок страниц.**

**И последний хук, из этого короткого пака видео, UseReducer. За 50 вопросов упоминался, но я не уловил.**

**Упоминается первая проблема текущего приложения. Все методы, которые меняют state, хранятся прямо в компоненте. Если их станет еще больше, то поддерживать такой компонент станет сложнее. Но с помощью хуку useReducer, который точно также работает в Redux, можно будет вынести эту логику из компонента в отдельный файл**

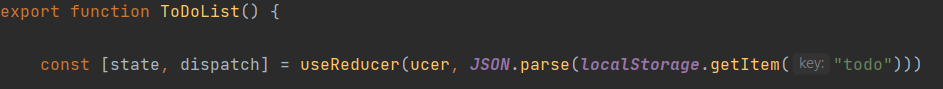
**Вникнуть в это было сложнее всего. Reducer нужен, чтобы поддерживать какое-то сложное состояние. Но мне бы больше примеров хотелось бы глянуть, где подключаются еще. Ну во первых вот так в отдельном файле прописывают эту функцию. И ей даже не нужно название прописывать?**



**Ну точнее на данный момент это обычная функция обработчик. В этом файле даже React не прописан. По умолчанию, функция будет возвращать состояние. Обработчики позже посмотрим.**

**Теперь из этого файла можно подключить на основной. Причем т.к. тут дефолт функция по умолчанию, то и назвать ее можно как угодно. Я такого никогда не видел.**



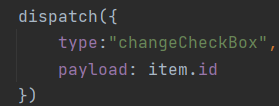


**Первый useEffect в котором была инициализация можно убрать. Вообще лучше вместо рандома и дефолта. Лучше их назвать, чтобы между ними можно было переключаться. Но пока он один, видимо так работает. В будущем узнаю.**

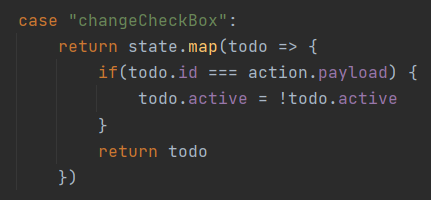
**Но теперь на странице есть этот state. Везде где он у меня использовался ранее, нужно теперь заменить его на state. В Контекст вместо функций теперь нужно передать dispatch**



**А теперь, как dispatch использовать. Формально нужно в него помещать 2 объекта. В type поместить название функции, которую нужно будет прописать в switch case и передаваемый параметр.**



**А потом просто прописать логику работы в файле.**



**Если что payload, action, state. Это все заменяемые слова.**

**Ну здесь можно прописывать эту логику и так передавать. Все.**

**Далее курс с моей флешки.**