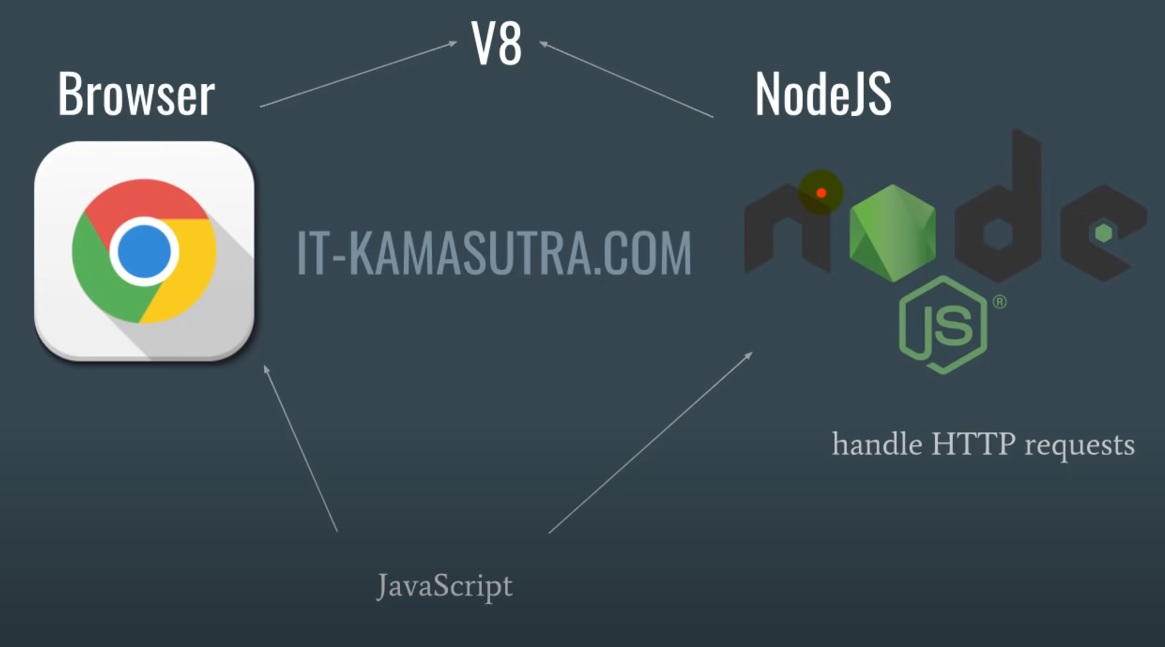
**01. ReactJS - Путь Самурая - Как смотреть данный курс?**

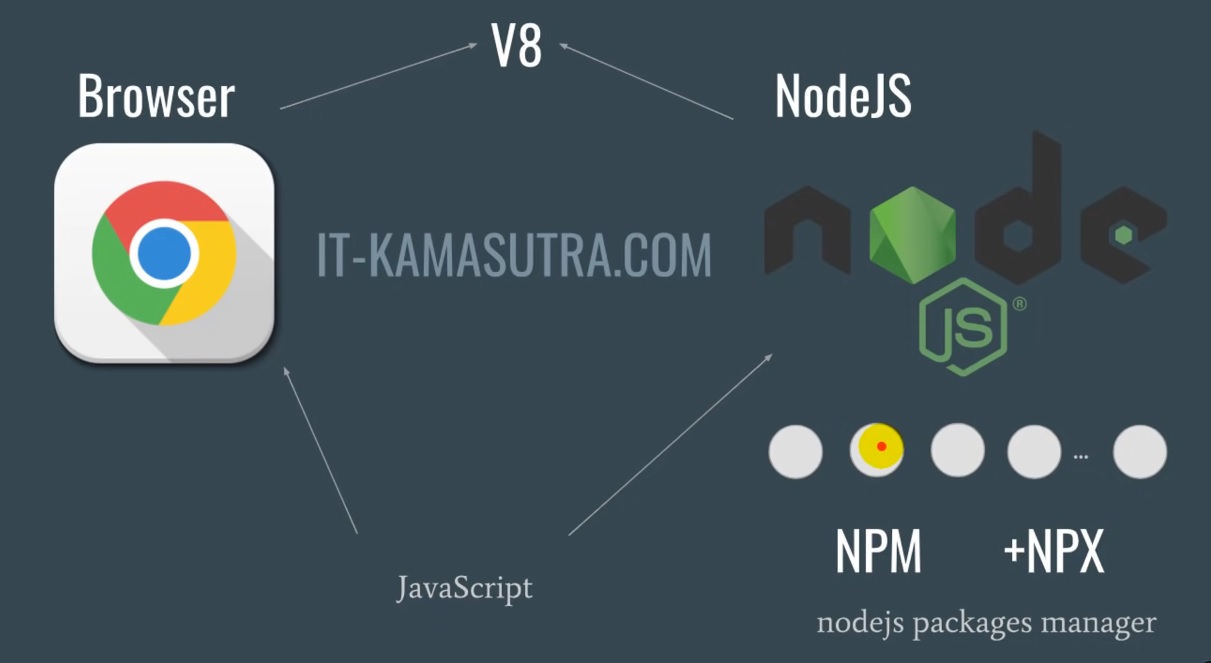
**02. Уроки React JS (Зачем NodeJS на фронте)**

Изначально JS разработан для того, чтобы выполняться в браузере. С развитием JS, он начал работать с базами данных, файловыми системами и т.д., для этого была создана альтернативная браузеру платформа NodeJS.



Браузер работает с html и модифицирует его при помощи JS. NodeJS работает с файлами, базам данных и т.д. при помощи JS. И браузер и NodeJS внутри используют движок V8, только каждая из этих платформ расширяет его по своему.

NodeJS называют серверным JS, потому, что одна из программ, которую можно написать под него сможет принимать http запросы от браузера, обрабатывать и возвращать ответы – програмный http сервер.



Современная разработка – это сложный процесс, поэтому Frontend разработчик использует NodeJS для получения и работы с модулями, компилирующими, минифицирующими, склеивающими JS, запускающими сервер и т.д. Модули разработаны под NodeJS и призваны упростить разработку. Node Package Manager(npm) – менеджер пакетов, устанавливается вместе с NodeJS, занимается, в основном, каталогизацией, установкой пакетов/модулей, поддержанием актуальных версий. npx позволяет не только устанавливать пакеты, но и запускать/исполнять.

**03. Уроки React JS (Установка NodeJS)**

NodeJS как и браузер является мультиплатформенным. Аналогично сайту, запущенному в браузере и работающему на любой ОС и платформе, программы NodeJS будут работать внутри NodeJS независимо от ОС и платформы.

После установки NodeJS запускаем не Node.js, а Node.js command prompt.

**04. Уроки React JS (create-react-app)**

Создаём новое React приложение согласно документации:

<https://github.com/facebook/create-react-app>

<https://create-react-app.dev/>

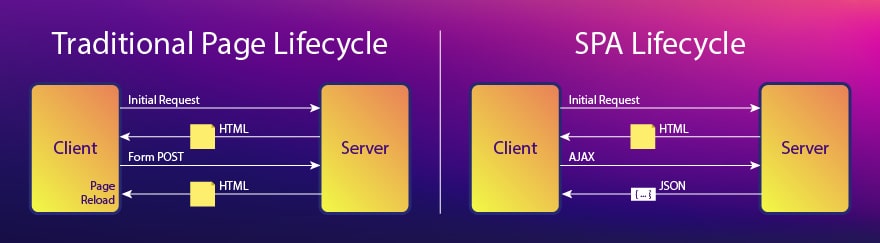
npx create-react-app my-app

cd my-app

npm start

Рекомендуется называть папку в стиле kebab-case, например 01-first-project. Проект/пакет/модуль, в том числе и React приложение считается таковым при наличии файла package.json, в котором хранится информация об этом пакете.

**05. Уроки React JS (SPA: single page application)**

****

В случае традиционной страницы сервер генерирует множество html и возвращает его в браузер. Следом за html приходит малое количество JS. Браузер/клиент отображает этот html, а чтобы проверить наличие изменений на сервере необходимо обновить страницу, клиент посылает на сервер новый запрос, сервер возвращает html страницу целиком. Если изменений не произошло, новая html страница аналогична предыдущей. При таком подходе, для каждой страницы на многостраничном сайте существует отдельный адрес, после запроса на которой загружается отдельная html страница.

SPA – маленькая полупустая html страница, в которую помимо html приходит много JS генерирующего динамическую разметку. SPA страницы/приложения загружают страницу единожды. Html приходит мало, страница практически пустая, контент и готовая разметка не приходит, но приходит больше количество JS. JS посылает AJAX запрос на сервер, сервер возвращает JSON файл с данными, на основании которых JS генерирует html. Страница не перезагружается, кроме того JS периодически отправляет на сервер AJAX запросы проверяя наличие изменений. Если изменения есть, генерируется небольшой html соответствующий этим изменениям. Все операции происходят с такой скоростью, что пользователь этого не замечает. В SPA всегда одна страница (не landing page!), при переходе на другую страницу многостраничного сайта, URL адрес меняется, но клиент не отправляет на сервер запрос, чтобы получить новую html страницу, вместо этого JS перехватывает изменение URL и делает AJAX запрос на сервер, на основе полученного в ответ JSON файл, JS стирает ненужное содержимое предыдущей страницы и рендерит новый контент, страница не перезагружается и не перерисовывается целиком.

Плюсы SPA – экономия трафика, уменьшение нагрузки на сервер, увеличение скорости работы страницы. Серверу проще отдать 100 человекам пустую html страницу + JS, а затем легковесный JSON практически не требующий генерации, нежели взять данные, сгенерировать html страницу с этими данными и вернуть клиенту. Сервер выступает в роли API (программный интерфейс приложения, application programming interface), и становится универсальным, он единожды отдаёт html и далее работает с AJAX запросами. При разработке мобильных приложений, html не используется, но нужны JSON данные, для их получения достаточно сделать запрос на тот же URL, что и браузер, но далее данные обрабатываются и отрисовываются по-другому. Технически сервер пригоден для работы с мобильным приложением, это существенная экономия средств.

React, Angular и другие фреймворки работают по принципу SPA.

**06. Уроки React JS (index.js, App, JSX)**

Внутренняя структура папки приложения, созданной npx create-react-app my-app:

* node\_modules Папку в ручную никогда не меняем, при необходимости можно «покопаться» в поисках нужных файлов;
* public в папке находится та самая единственна страница index.html SPA приложения;
* src в папке находится файл App.js содержащий ту самую разметку, которая отображается на странице.

Facebook разработал отдельный язык JSX позволяющий писать html внутри JS. Компилятор превращающий JSX в JS следит за тем, чтобы JSX был валидным, иначе на странице приложения отображается ошибка. Если бы это был html файл, браузер бы всё равно как-то его отрисовал.

В JSX вместо html атрибута class, пишется className, т.к. слово class зарезервировано для других целей.

Внешний вид приложения задаётся с помощью импорта стилей:

import './App.css';

Стили прописанные в App.css применятся к странице. Ранее, в случае традиционной страницы, скрипты подключались с помощью тэга <script>, а стили с помощью <link>.

Основная работа происходит не в html файле, а в js файлах и соответствующих им css файлах. Основная задача функций типа App и классов в React – возвращать JSX разметку, которая будет рендериться на страницу. Разметка обязательно должна быть обёрнута в один родитель <div> (корневой html узел).

Отформатировать код в VS Code Ctrl+K Ctrl+F

Открыть палитру команд в VS Code (верхняя строка) Ctrl + Shift + P

**07. Уроки React JS (что такое Компонента)**

Компонент – это функция, возвращающая разметку JSX. При этом может выполнять множество операций, но через return возвращать разметку. К примеру функция App из предыдущего урока ничего не делает, кроме возврата разметки:

const App = () => {

  return (

    <div>

      <div>one div</div>

      <div className="App">

        <ul>

          <li>css</li>

          <li>html</li>

          <li>js</li>

          <li>react</li>

        </ul>

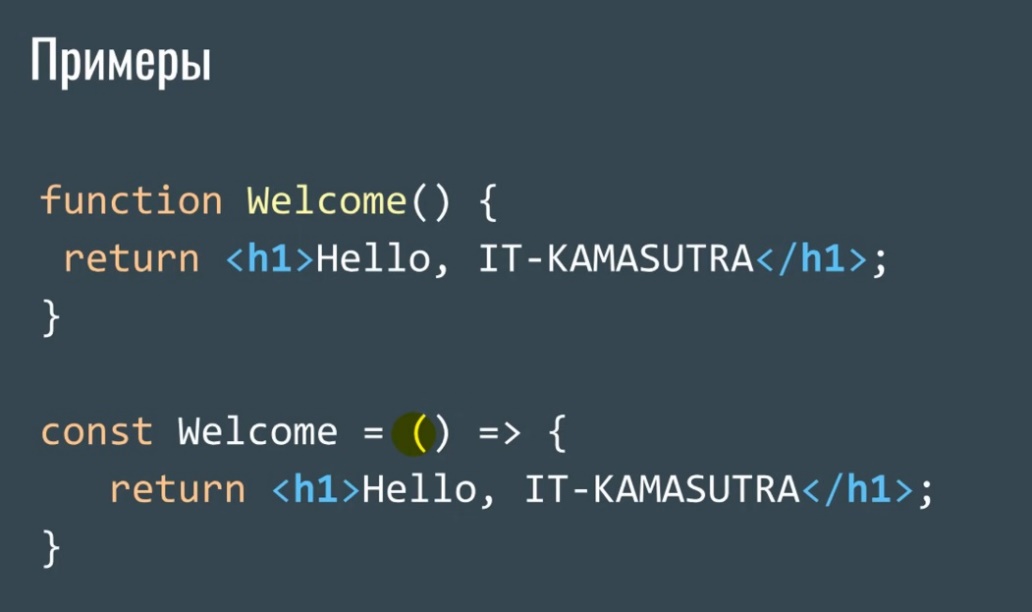
      </div>

    </div>

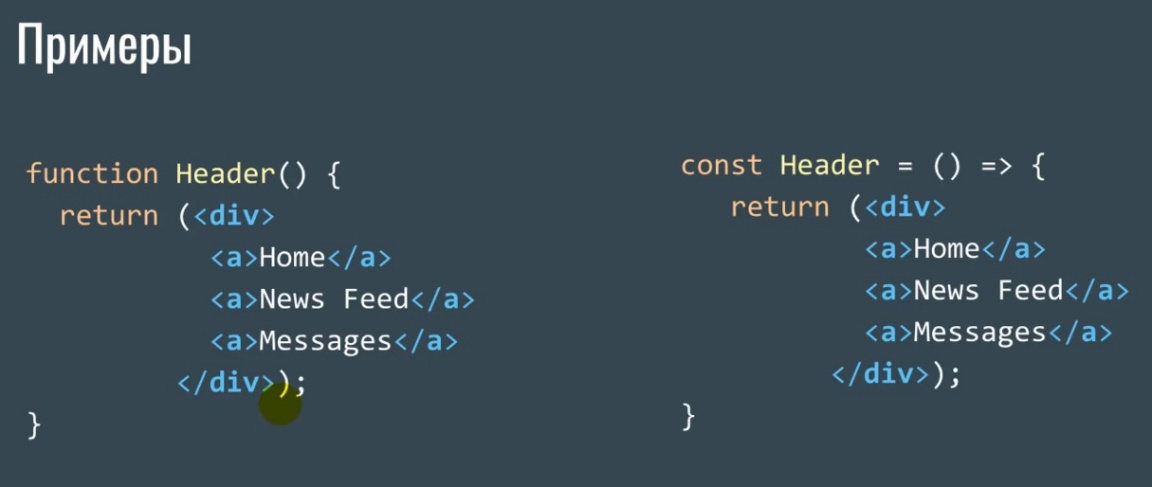
  );

}

Компоненты – «кирпичики» из которых строится React приложение.



В первом случае компонент написан с помощью объявления функции, во втором – с помощью функционального выражения с использованием стрелочной функции. return может использоваться без скобок, если возвращаемый им код написан в одну строку.



Компонент – это, в том числе, и часть какого-то итогового html.

Как использовать компонент ?

При каждом объявлении функции-компонента, в системе появляется новый условный тэг, например <App />, который далее можно использовать для отображения на странице. Компонент никогда не вызывается вручную App(); !!! Вызовом этой функции занимается React.

В папке public/index.html есть <div id="root"></div>, JS находит его по id и отрисовывает своё содержимое именно внутри этого тэга.

Компонент можно вкладывать в другой компонент, аналогично тэгам в html. Основная идея для создания новых компонентов – семантическое разделение страницы на header, sidebar, menu, footer и т.д.

Компонент App чаще всего это компонент, внутри которого лежат другие компоненты:

const App = () => {

  return (<div>

    <Header />

    <Technologies />

  </div>);

}

При рендере на страницу, React вызывает компоненты, вместо тэгов компонентов вставляется возвращаемая ими JSX разметка, которая перед отображением на странице компилируется в JS.

Приложение может дробиться сколь угодно большое количество компонентов. Даже поисковая строка может состоять из нескольких компонентов: сама строка – один компонент, копки поиска – другой. Компоненты всегда должны иметь понятные названия, это обеспечивает красиво свёрстанную семантически осмысленную структуру приложения.

**08. Уроки React JS (import\export - теория)**

Компоненты принято разносить по разным файлам.

Пример. Есть 4 файла:

1) yandex.js

let openYandex = () => {

    // connect to Yandex

    // show map

}

2) google.js

let openGoogle = () => {

    // connect to Google

    // show map

}

3) maps.js

let openAllMaps = () => {

    openYandex();

    openGoogle();

}

4) main.js

openAllMaps();

В нативном JS не приветствуется плодить глобальные переменные – те, что видны для всех файлов, т.к. от этого код становится запутанным. Функцию приходится искать во всех файлах, а если её название дублируется, то даже среда разработки не сможет помочь.

Глобальный поиск в VS Code Ctrl+Shift+F

Чтобы найти откуда пришла функция в VS Code зажать Ctrl, навести и кликнуть по ней

React использует модульный подход. Почитать modules AMD require.js. На данный момент код из примера в каждом конкретном файле виден только в этом файле. Чтобы использовать функции из других файлов, их нужно явно импортировать, с указанием что импортируется и откуда=пусть к файлу откуда импорт относительно текущего файла. Относительно maps.js:

import openGoogle from './google.js'

import openYandex from './yandex.js'

На практике тип файла .js не ставится. Импортируем openAllMaps в main.js:

import openAllMaps from './maps.js'

Для того, чтобы что-то импортировать в файл, так же нужно это экспортировать из исходного файла. Экспортируем openGoogle из google.js:

export default openGoogle;

default – один из типов экспорта, когда из файла экспортируется что-то одно. Существует несколько вариантов экспортов/импортов, они будут рассмотрены позже.

Аналогично экспортируем openYandex из yandex.js, а так же openAllMaps из maps.js.

Возвращаясь к React проекту.

import './App.css'; - импорт целого css файла, а не чего либо из JS файла.

import React from 'react'; - импорт React не из файла react.js (его нет в папке и пути к нему, соответственно, тоже), а из модуля, который лежит в папке node\_modules.

import App from './App'; - импорт компонента App из файла App.js.

export default App; - экспорт компонента App из файла App.js, где он объявляется.

**09. Уроки React JS (import export наших компонент)**

Перемещаем код компонентов в отдельные файлы Technologies.js и Header.js, и сразу же экспортируем их по умолчанию.

Сделать копию файла на месте в VS Code выбрать файл, Ctrl+C Ctrl+V

Чтобы использовать тэги перемещённых компонентов, эти компоненты нужно импортировать в App.js:

import Header from './Header';

import Technologies from './Technologies';

У меня всё ОК, но после запуска у Димыча в браузере выскакивает ошибка ‘React’ must be in scope when using JSX, т.е. т.к. в Technologies.js и Header.js, а так же в App.js используется синтаксис JSX, React должен быть доступен=импортирован. При создании моего проекта, React был импортирован только в index.js, значит в моей версии импортировать React в каждый файл с JSX уже не нужно (по крайней мере в рамках проекта). Но я прописал импорты как в уроке.

UPD. вышел React 17 и можно использовать JSX без импорта React в другие файлы, в react компоненты добавили компонент jsx-runtime, он как раз и позволяет использовать JSX.

https://habr.com/ru/post/521930/

Импортируем React в Technologies.js и Header.js (и в App.js). Приложение работает исправно, как и раньше.

**10. Уроки React JS (default опасность плагин auto import)**

Если в файле App.js закомментировать импорт Header и тэг <Header /> в функции App, то на странице будет отрисован только компонент Technologies. Но если импортировать Technologies из Header.js import Technologies from './Header'; то ошибки не будет, а на странице будет отрисован компонент Header. Причина такого поведения в export default, с которым всё равно придётся работать, поэтому при импорте нужно внимательно следить за тем, что и откуда импортируется.

export default Header;

не обозначает привязку к имени Header, а значит что что-то экспортируется по умолчанию. Мы осознаём, что экспортируем функцию Header, но в контексте экспорта у неё фактически нет имени. При импорте можно дать этому чему-то любое имя, и приложение продолжит работать:

import Abrakadabra from './Header';

const App = () => {

  return (<div>

    {/\* <Header /> \*/}

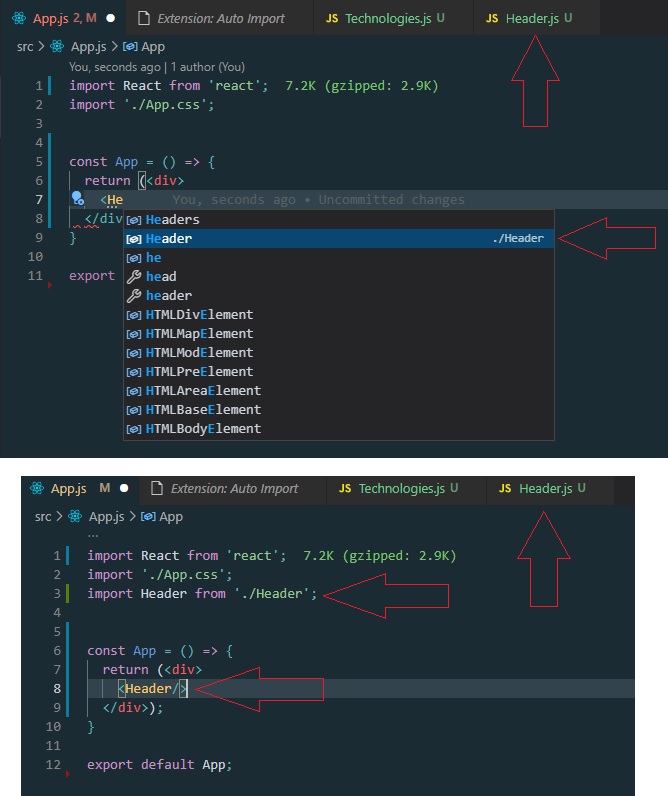
    <Abrakadabra />

  </div>);

}

НО! Как правило название при импорте соответствует тому, что мы импортируем один к одному.

Устанавливаем расширение Auto Import для VS Code. Оно позволяет автоматически правильно импортировать компоненты. В процессе написания тэга компонента, расширение автоматически импортирует его, при условии что файл импортируемого компонента открыт и в нём прописан экспорт (возможно расширению нужно считать файл и убедиться в наличии экспорта).



Посмотреть полный текст подсказки в VS Code при выборе из выпадающего списка Ctrl+Space

Создаём новый файл Footer.js, чтобы импортировать функцию из него в App.js с помощью нового расширения.

const Footer = () => {return <div>it-kamasutra.com footer</div>;}

Если компоненты на странице дублируются, то нужно перепроверить не ошибся ли Auto Import.

**11. Уроки React JS (соц. сеть, шаг 1, вёрстка, grid)**

Название компонентов обязательно нужно писать с большой буквы.

Делаем простую страницу социальной сети по макету:

<https://marvelapp.com/prototype/39ef033/screen/50996553>?

Абстрактно разбиваем страницу проекта на высокоуровневые блоки - header сверку, navblock/sidebar слева, и content справа от sidebar, под header. Приступаем к вёрстке макета, используя grid. Разобраться что такое css grid. Всю вёрстку поместим внутрь компонента App, учебные файлы Header.js, Technologies.js и Footer.js удаляем. Соответственно удаляем импорты этих компонентов в App.js.

Задача – обозначить названные выше три контейнера и уже потом заниматься деталями. Навигационные ссылки для навигации можно сделать с помощью <ul> и вложенных <li>, но Димыч предпочитает <div>.

const App = () => {

  return (<div className='app-wrapper'>

    <header className='header'>

      <img src=" https://www.pinclipart.com/naruto.png " alt="logo" />

    </header>

    <nav className='nav'>

      <div>Profile</div>

      <div>Messages</div>

      <div>News</div>

      <div>Music</div>

      <div>Settings</div>

    </nav>

    <div className='content'>

      Main Content

    </div>

  </div>);

}

Родительскому блоку, в котором находятся остальные контейнеры назначается className='app-wrapper'. Вложенным контейнерам задаются className header, nav и content соответственно.

.app-wrapper {

    display: grid;

    grid-template-areas: "h h" "n c";

    grid-template-rows: 60px 1fr;

    grid-template-columns: 2fr 10fr;

    gap: 10px;

}

.header {

    grid-area: h;

    background-color: #12cf67;

}

.header img {

    width: 60px;

}

.nav {

    grid-area: n;

    background-color: #12cf67;

}

.content {

    grid-area: c;

}

В стилях классу app-wrapper задаётся dispay:grid это значит, что все прямые потомки этого элемента станут элементами сетки. С помощью css свойства grid-area задаём им имена/псевдонимы по первой букве их классов, на которые можно будет сослаться при создании шаблона сетки. Далее с помощью свойства grid-template-areas определяем шаблон сетки ссылаясь на псевдонимы, а с помощью grid-template-rows/columns задаём размеры строк и колонок. С помощью gap задаём отступы между строками и колонками.

Когда общая структура проекта выглядит как нужно, переходим к детализации. Верстаем наброски секции nav, затем content. Использование гридов в начале, не обязывает их использовать повсеместно (<https://htmlacademy.ru/blog/boost/frontend/flexbox-vs-grid>).

**12. Уроки React JS (Разбивка на компоненты)**

Вёрстка всего в одном компоненте становится плохо читаемой, лучше разделять её на подкомпоненты. Создаём для них папку components, в которой создаём файл Header.js, в котором объявляем компонент Header, внутрь return() вставляем JSX из App отвечающий за отображение шапки приложения, обязательно экспортируем Header.

const Header = () => {

    return (

        <header className='header'>

            <img src="https://www.pinclipart.com/naruto.png" alt="logo" />

        </header>

    );

}

export default Header;

Затем тэг <Header /> помещается в компонент App, внутрь return(), на то же место, где была соответствующая вёрстка. В процессе написания тэга, Auto Import импортирует Header внутрь App.js. Аналогично для nav → Navbar.jsx. Для content ситуация другая – заранее предполагаем, что в социальной сети будет много разных страниц, у которых Header и Navbar останутся прежними, а внутренность будет меняться. На данный момент содержимое content соответствует странице профиля пользователя, поэтому называем компонент Profile.jsx:

const Profile = () => {

    return (

        <div className='content'>

            <div>

                <img src="https://cdn.pixabay.com/photo/2022/01/08/19/13/animal-6924664\_960\_720.jpg" alt="background\_image" />

            </div>

            <div>

                <img src="https://cdn.iconscout.com/icon/free/png-256/avatar-370-456322.png" alt="avatar" />

                <div>Description</div>

            </div>

            <div>

                My posts

                <div>

                    New post

                </div>

                <div>

                    <div>

                        post 1

                    </div>

                    <div>

                        post 2

                    </div>

                </div>

            </div>

        </div>

    );

}

export default Profile;

Теперь компонент App выглядит гораздо более читаемо. Новая задача – доработать каждый из компонентов, чтобы они выглядели как на макете.

Для добавления в портфолио крайне желательно изменить стилизацию приложения и красиво сверстать. <https://themeforest.net/search/social%20network> – примеры оформления.

Каждый компонент в дальнейшем может быть разбит на более мелкие компоненты, лишнего дробления на компоненты не бывает.

**13. Уроки React JS (git-репозиторий)**

<https://github.com/it-kamasutra/react-way-of-samurai>

Разобраться что такое SSH

Клонируем репозиторий. Устанавливаем node\_modules командой npm i. Все необходимые модули прописаны в package.json→ dependencies, существуют так же devDependencies используемые только в процессе разработки. Устанавливается множество зависимостей, то что мы видим мало зависимостей в package.json связано с тем, что разработчики Facebook упаковали/инкапсулировали детали для удобства пользования, можно всё распаковать командой npm eject, всё распакуется, но мы не сможем получать обновления React, только ручная поддержка.

При подключении новых модулей в проект важно прописывать флаг –save, тогда зависимость будет записана в package.json.

В клонированной папке находится последняя версия приложения, чтобы откатываться к предыдущим версиям, нужно перейти в клонированную папке, ПКМ→Git Bash Here→gitk --all&, откроется графический интерфейс. При запуске npm start на последних версиях выскакивает ошибка, исходя из комментариев под видео, исправляется заменой версии тайпскрипта npm i typescript@3.8 –save

<https://www.youtube.com/watch?v=6pAkynDPXEc&t=541s>

Если внесены изменения (после манипуляций выше они будут внесены), чтобы откатиться на другой коммит, нужно откатить изменения – удаляем packages-lock.json, проверяем git status какие файлы были модифицированы, после прописываем git checkout имя\_файла для этих файлов. Порядок перемещения на другой коммит – в GUI выбираем нужный коммит, копируем его ID командой ctrl + insert, прописываем git checkout и вставляем ID командой shift + insert, проект откатится у указанному коммиту.

В идеале, это можно делать не останавливая выполнение проекта, работая в двух консолях.

**14. Уроки React JS (css-модули, css-modules)**

Почитать эволюция CSS

Знать концепцию БЭМ методологии

Чтобы повысить удобство работы и читабельность кода, как ранее дробили html на компоненты, так же разбиваем и стили. Рядом с Header.jsx, Navbar.jsx, Profile.jsx создаём Header.css, Navbar.css, Profile.css. Вырезаем из App.css правила стилизации соответствующих блоков и помещаем их в созданные css файлы. Файлы со стилями импортируем в .jsx файлы. Проверяем, что ничего не сломалось, приложение выглядит как и прежде.

Пунктам меню в Navbar задаём className='item' и хотим сделать текст внутри них белым, для этого прописываем для item правило color: white; Далее постам в Profile так же хотим задать класс className='item' и стилизовать их. Видим, что для постов так же применилось правило color: white; не смотря на то, что в Navbar.jsx и Profile.jsx импортированы разные css файлы. Это вызывает сложности, и необходимость использовать методологию БЭМ или вложенности. Почему так происходит ? Используя на странице devTool→Network видим, что никаких css файлов не приходит, т.к. css помещён внутрь JS, после чего JS добавляет его прямо head html документа, в тэг style. Эти стили являются глобальными и общими для всей страницы, и если у содержимого одного из компонентов есть класс item, то стили для этого класса будут влиять на все элементы приложения с таким же классом = инкапсуляция стилей отсутствует.

Все JSX файлы в итоге компилируются в bundle.js

Чтобы инкапсулировать стили, разработчики React внедрили css-модули. Объявить, что файл является модулем можно просто добавив .module к его названию Navbar.module.css Такой файл ведёт себя иначе и взаимодействовать с ним нужно иначе. Это не обычный сss файл, а css-модуль = css-объект. При импорте этому объекту нужно давать условное название и указывать откуда он импортируется:

import s from'./Navbar.module.css';

После подключения такого модуля стили компонента Navbar не отображаются на странице, а в инспекторе кода появились новые классы .Navbar\_nav\_\_Q704K и .Navbar\_item\_\_TFAWD с рандомными постфиксами. Т.е. наши классы уже не называются .nav и .item, именно по этой причине компонент Navbar не стилизован. Если добавить новые классы, элементы будут стилизованы как и прежде

<nav className='Navbar\_nav\_\_Q704K'>

Имена классов были намеренно искажены, чтобы стать уникальными. Разработчики могут использовать любые имена css классов, в том числе и одинаковые для стилизации разных компонентов и они не будут переопределять друг друга.

Как правильно добавлять такие css классы в JSX ? Все эти названия классов находятся внутри импортируемого объекта s. s – это объект ключами которого являются названия изначальных классов, а значениями являются сгенерированные классы:

let s = {

  'nav': 'Navbar\_nav\_\_Q704K',

  'item': 'Navbar\_item\_\_TFAWD'

}

Чтобы добавить такой класс, нужно начать с {} – это говорит о том, что внутрь JSX-html разметки будет вставлен JS код. Т.к. мы не знаем имени сгенерированного класса, то можно передать значение нужного свойства объекта s:

<nav className={s.nav}>

Компилятор перейдёт в объект s, из его свойств nav получит строку с именем класса и подставит её вместо всего выражения {s.nav}.

Аналогично преобразовываем Profile.css в css-модуль Profile.module.css и Header.css в Header.module.css. Вложеность .header img внутри Header.module.css будет работать как и прежде, т.к. изменилось только название класса и картинка останется вложенной в новый класс header. Медиа стили, так же продолжат работать исправно при использовании css-модулей.

Создание общих стилей будет рассмотрено далее по курсу.

App.css оставляем без изменений, т.к. разрешается использовать обычные глобальные css классы наряду с классами css-модулей.

Договорённость! В названии свойств объекта импортируемого из css-модуля не используем дефис – {s.header-jk} т.к. в JS свойствам запрещено давать имя с дефисом, если использован дефис, то к свойству нужно обращаться при помощи другой нотации {s[“header-js”]}, что неудобно. Но можно использовать подчёркивание либо camelCase header\_jk, headerJk.

Если элементу нужно добавить другие стили/классы, при использовании обычного css, они бы указывались через пробел:

<div className="item active"><a>Messages</a></div>

Но при использовании css-модулей, сгенерированные имена этих классов находятся внутри импортируемого объекта со стилями, значит чтобы указать новые классы нужно через пробел передать значения соответствующих свойств этого объекта. До ES6 это выглядело как конкатенация значений со строкой, внутри которой пробел:

<div className={s.item + " " + s.active}><a>Messages</a></div>

После ES6 и появления шаблонных строк = интерполяции, синтаксис упростился до:

<div className={`${s.item} ${s.active}`}><a>Messages</a></div>

Пробел воспринимается как строка, а вместо переменных `${s.item}` и `${s.activ}` компилятор подставит строки с именами классов.

Вывод в консоль объекта со стилями s даёт:

Object

active: "Navbar\_active\_\_v8nEw"

item: "Navbar\_item\_\_TFAWD"

nav: "Navbar\_nav\_\_Q704K"

**15. Уроки React JS (структура папок, новые компоненты, улучшаем css)**

При разработке проекта, хороший подход разрабатывать структуру не углубляясь в вёрстку, но по итогу сделать её как надо.

Компонентов будет очень много и если оставить их в папке components как есть, будет легко запутаться. Чтобы избежать этого, договариваемся, что если с компонентом лежит ещё и css файл, то создаём для них отдельную папку Header, Navbar, Profile. После разнесения по папка может возникнуть ошибка, связанная с изменением вложенности файлов, импорты перестали работать. WebStorm и VS Code автоматически модифицирует импорты, в VS Code для этого нужно остановить компиляцию проекта, затем перемещать файлы.

Далее будем работать с компонентом Profile, блоком posts. Раздробим Profile на MyPosts и Post, содержащий одну запись на стене. Для начала создаём внутри папки Profile папку MyPosts, куда помещаем файл MyPosts.jsx с одноимённым компонентом, внутрь которого вырезаем и вставляем из Profile.jsx часть JSX разметки, отвечающую за отображение блока с постами:

import s from './MyPosts.module.css';

const MyPosts = () => {

    return (

        <div>

            My posts

            <div>

                New post

            </div>

            <div className={s.posts}>

                <div className={s.item}>

                    post 1

                </div>

                <div className={s.item}>

                    post 2

                </div>

            </div>

        </div>

    );

}

export default MyPosts;

Не забываем об импорте стилей и экспорте компонента MyPosts. Рядом с MyPosts.jsx создаём MyPosts.module.css, куда вставляем вырезанные из Profile.module.css правила стилизации постов:

.item {

    color: plum;

}

В Profile.jsx, вместо блока постов прописываем тэг компонента <MyPosts />, auto import автоматически импортирует этот компонент сверху:

. . . import MyPosts from './MyPosts/MyPosts';

const Profile = () => {

    return (

        <div className={s.content}>

            <div>

                <img src=". . ." alt="background\_image" />

            </div>

            <div>

                <img src=". . ." alt="avatar" />

                <div>Description</div>

            </div>

            <MyPosts />

        </div>

    );

} . . .

MyPosts тоже хотим раздробить на компоненты. Постов может быть много и разных, чтобы не верстать и стилизовать каждый из них по отдельности, создаём для поста отдельный тэг/компонент. Все новые посты будут создаваться с помощью одного компонента Post. Внутри папки MyPosts создаём папку Post, копируем в неё файлы MyPosts.jsx и MyPosts.module.css переименовываем из в Post.jsx и Post.module.css. Переименовываем компонент в Post, удаляя из него лишнюю JSX/html вёрстку не относящуюся к посту, исправляем импорт и экспорт:

import s from './Post.module.css';

const Post = () => {

    return (

        <div className={s.item}>

            <img src=". . ." alt="user\_avatar" />

            post 1

        </div>

    );

}

export default Post;

Удаляем стили из MyPosts.module.css т.к. они относятся к Post и уже прописаны в Post.module.css.

В MyPosts.jsx удаляем разметку постов, вместо них прописываем тэг <Post />, auto import импортирует этот компонент автоматически:

. . . import Post from './Post/Post';

const MyPosts = () => {

    return (

        <div>

            My posts

            <div>

                <textarea></textarea>

                <button type='submit'>Add post</button>

            </div>

            <div className={s.posts}>

               <Post />

               <Post />

               <Post />

               <Post />

               <Post />

               <Post />

            </div>

        </div>

    );

} . . .

Теперь управлять стилями всех постов можно из Post.module.css, а если нужно изменить вёрстку постов, то всё это можно сделать в одном месте - в Post.jsx.

Проблема – наполнение всех постов одинаковое и статично меняется для всех постов одновременно. Хотелось бы настраивать каждый <Post /> отдельно, чтобы он мог принимать данные и менять своё содержимое.