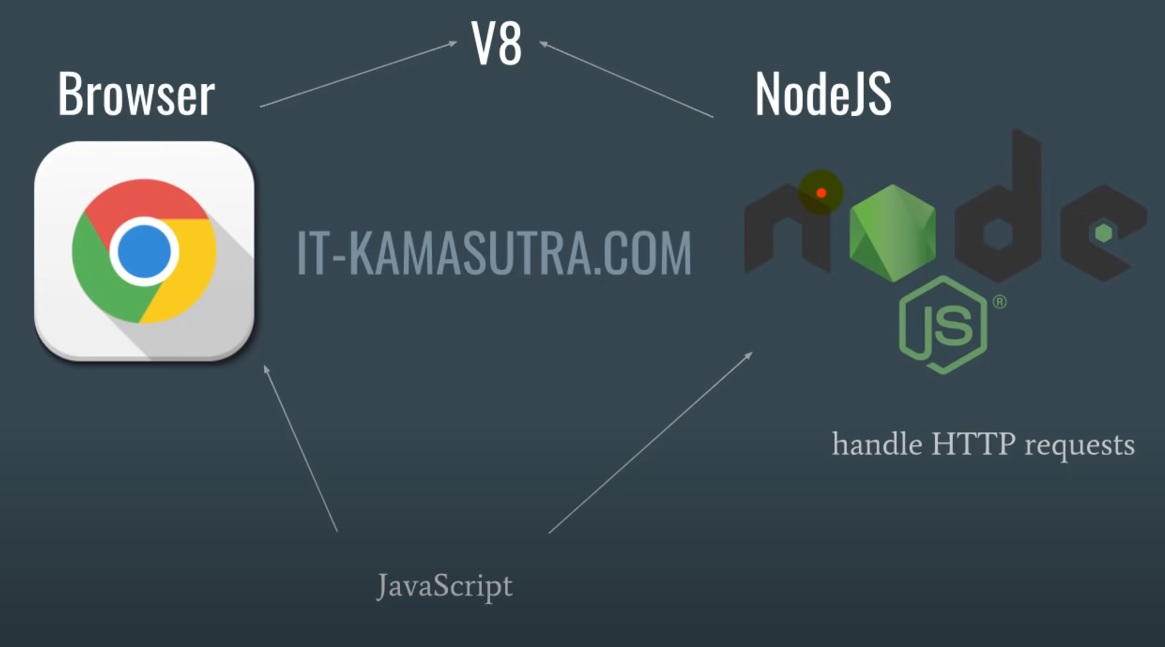
**01. ReactJS - Путь Самурая - Как смотреть данный курс?**

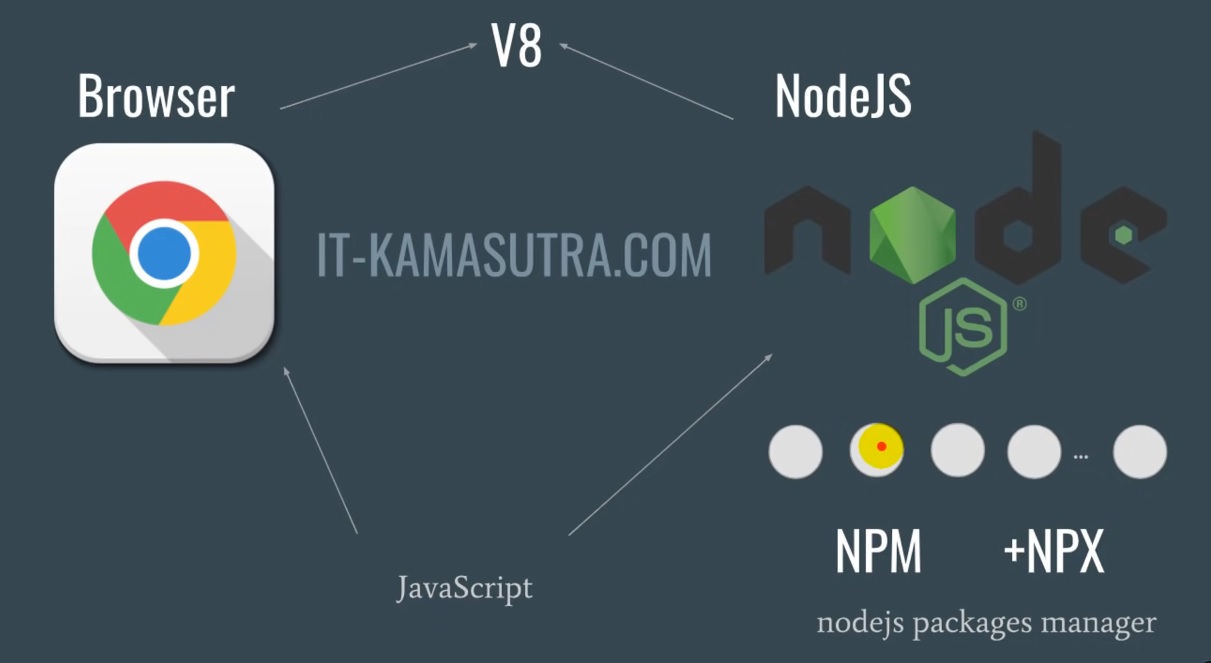
**02. Уроки React JS (Зачем NodeJS на фронте)**

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/>

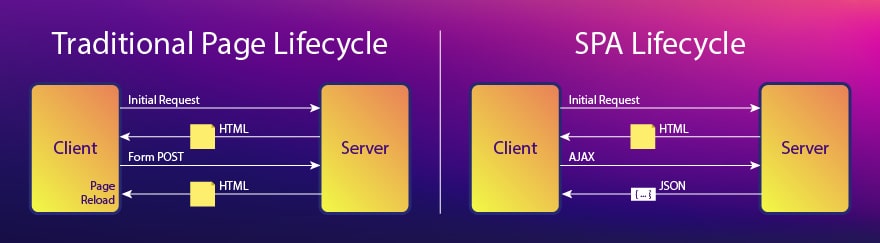
npx create-react-app my-app

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>

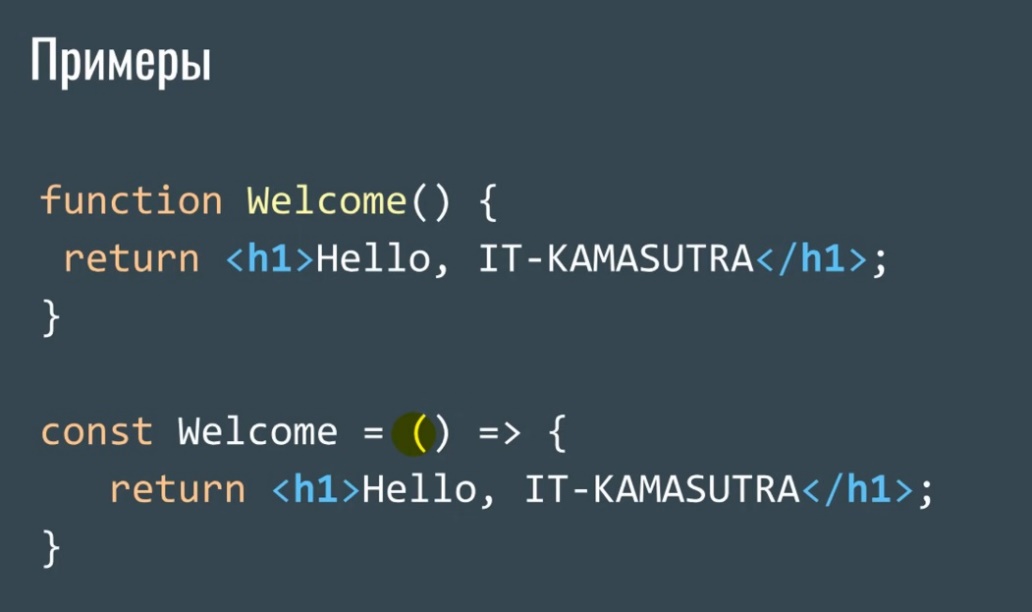
      </div>

    </div>

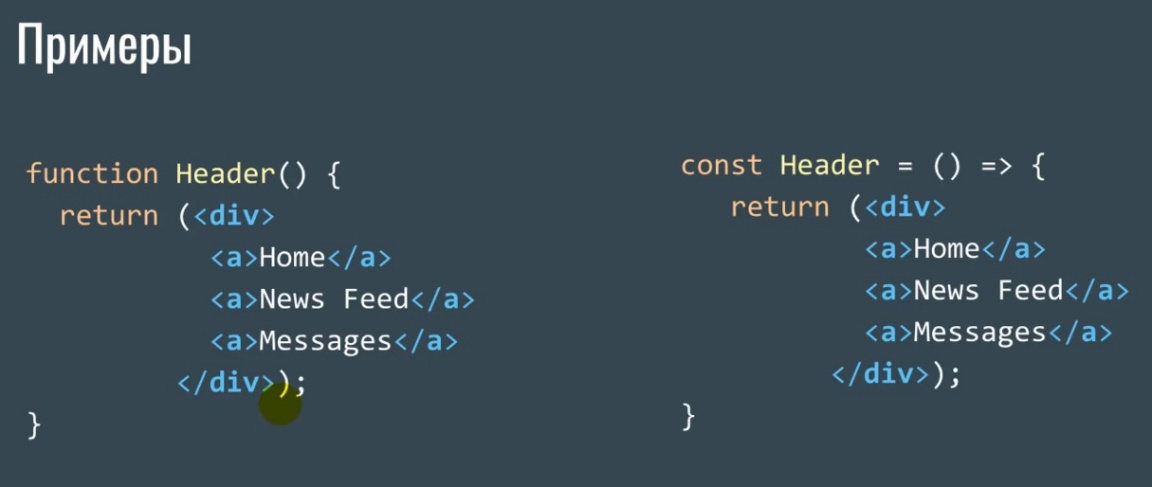
  );

}

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 /> \*/}

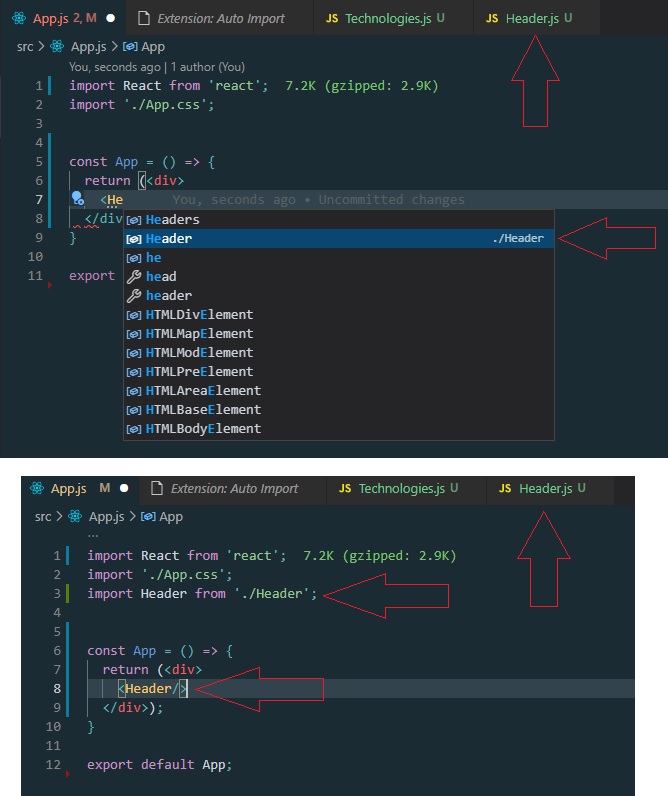
    <Abrakadabra />

  </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.

**16. Уроки React JS (props - важнейшая тема) - react курсы бесплатно**

Компонент является функцией, которую вызывает React. Если функция должна вести себя по разному когда её вызывают, то она должна принимать параметры – что-то, что приходит извне.

Каждый компонент всегда вызывается с параметрами. Самый главный параметр – props, его можно было бы назвать как угодно, но по соглашению принято называть props.

Находясь внутри функции можно манипулировать объектом props, считывать его содержимое и проверять что в нём пришло извне. Например можно изменить разметку, подстроившись под props.

По умолчанию функции в качестве props передаётся ссылка на пустой объект/в нём нет никаких свойств, но props в компонент приходит всегда!

Чтобы передать props и он попал внутрь объекта, который передаётся функции, его нужно прописать в тэга компонента как атрибут.



Компонент – это тэг, а тэг можно настраивать с помощью атрибутов извне. React автоматически создаёт объект у которого будет свойство name, соответствующее имени атрибута – тоже name, и значение, соответствующее присвоенному атрибуту значению (Dima K). Внутрь Header придёт не пустой объект а со свойством name. Если передать два атрибута name и age, то придёт объект с двумя свойствами name и age, таким образом можно отрисовать два разных Header’а.



Порядок передачи props:

- добавляем тэг компонента в разметку

- прописываем атрибуты и присваиваем им значения

- формируется объект, который передаётся функции

- функция получают доступ к объекту внутри props

Если нужно внутри JSX разметки обратиться к JS объекту, ставим {}, внутри которых получаем значение нужного свойства объекта, например {props.name} вернёт строку ‘Dima K’, {props.age} вернёт строку ‘30’, запятая отобразится как запятая, пробел как пробел.

Переходим в MyPosts.jsx и добавляем в тэги <Post /> одинаковый атрибут message, но с разыми значениями:

. . . <div className={s.posts}>

               <Post message='Hi, how are you ?' />

               <Post message="It's my first post" />

            </div> . . .

На данном этапе props приходят внутрь поста, но они нигде не используются. Чтобы отобразить содержимое message вместо текста заглушки post 1, передаём props в качестве аргумента функции Post, и на месте post 1 прописываем {props.message}:

. . . const Post = (props) => {

    return (

        <div className={s.item}>

          <img src="https://vuesax.com/avatars/avatar-3.png" alt="user\_avatar" />

            { props.message }

            <div>

                <span>like</span>

            </div>

        </div>

    );

} . . .

Функция Post вызывается дважды и на странице мы получаем два поста с разными сообщениями.

Чтобы узнать что приходит в props, можно прописать debugger; перед return() в коде выше, тогда выполнение кода остановится в этом месте и в консоли разработчика на странице мы можем навести на props и посмотреть, что там находится. Убеждаемся, что это объект со свойством message:

Object

message: "Hi, how are you ?"

[[Prototype]]: Object

После нажатия на кнопку play функция Post вызывается второй раз и props с:

Object

message: "It's my first post"

[[Prototype]]: Object

Димыч призывает пересмотреть ролик трижды. Пропсы используются в 99% случаев, это важнейшая вещь в React.

Домашнее задание: добавить отображение количества лайков, под первым постом – 15, под вторым – 20. MyPosts.jsx:

. . .<div className={s.posts}>

               <Post message='Hi, how are you ?' likeCounter='15 '/>

               <Post message="It's my first post" likeCounter='20 '/>

     </div> . . .

Post.jsx:

. . . <div>

                <span>like</span> { props.likeCounter }

            </div> . . .

**17. Уроки React JS (остановочка "react за час, часть 1")**

React интерпретирует JSX разметку не как html разметку, а как особые инструкции по созданию virtual DOM(eng. Document Object Model, рус. Объектная Модель Документа) элементов.

Функция/компонент может вернуть только один корневой тэг = одно обёртку.

Чтобы css правила касались только конкретного компонента, к которому они относятся, нужно использовать css классы как модули. При использовании css-модулей, получаем имена классов чем-то похожие на БЭМ. В Angular стили инкапсулируются так же подобным образом.

Кроме npx create-react-app есть и другие способы создания настроенных React приложений.

Точка входа в приложение – файл index.js, именно в нём запускается отрисовка приложения. Единственная страница, которая загружается в браузер – index.html, именно в нём находится <div id="root"></div> , в который index.js помещает тэг <App />:

ReactDOM.render( <App />, document.getElementById('root'));

Компонент App внутри себя рендерит другие компоненты, которые в свою очередь рендерят более мелкие компоненты и т.д.

**18. Уроки React JS (страница Dialogs) - react курсы бесплатно**

Перешли на WebStorm

Компонент – это функция, которая возвращает JSX и принимает параметр props.

Одна из концепций функционального программирования – функция должна работать предсказуемо. Функция должна возвращать что либо основываясь только на входящих данных.

Далее будем делать страничку Messages – тот же Header и Sidebar, но середина другая.

Готовим место для отрисовки будущего компонента, для этого в App.js комментируем <Profile />. Комментировать в WebStorm выделить участок кода, нажать Ctrl+Shift+/

Убеждаемся, что на странице исчез компонент Profile. Вместо тэга <Profile /> прописываем тэг <Dialogs />, такого компонента пока не существует, поэтом импортировать его, вызвав контекстное меню пока невозможно.

Открыть контекстное меню в WebStorm кликнуть по участку кода, нажать Alt+Enter

В папке components создаём новую папку Dialogs с файлами Dialogs.jsx и Dialogs.module.css внутри. Переименовать файл в WebStorm Shift+F6. Временно содержимое Dialogs.jsx:

import s from './Dialogs.module.css'

const Dialogs = (props) => {  
 return (  
 <div>  
 Dialogs  
 </div>  
 );  
}  
export default Dialogs;

Если функция экспортирована по умолчанию, при импорте ей можно дать любое другое имя.

Форматирование кода в WebStorm выделить участку кода, Ctrl+Alt+L

Чтобы не заниматься дублированием кода, а именно css правила *grid-area: c* для каждого компонента, который будет располагаться в середине страницы, т.е. для Profile, Dialogs и т.д., решено создать отдельный <div> внутри компонента App, с классом className='app-wrapper-content' в который будут помещаться компоненты Profile, Dialogs и т.д. и уже ему назначить свойство *grid-area: c*  в файле App.css.

const App = () => {  
 return (<div className='app-wrapper'>  
 <Header/>  
 <Navbar/>  
 <div className='app-wrapper-content'>  
 <Dialogs/>  
 {/\*<Profile />\*/}  
 </div>  
 </div>);  
}

.app-wrapper-content {  
 grid-area: c;  
}

Из файлов компонентов Profile и Dialogs удаляем класс .content и указанное выше css правило.

**19. Уроки React JS (route, browser-router, маршрутизация) - react курсы бесплатно**

Сейчас мы можем увидеть страницы Profile и Dialogs только подменив их вручную в файле App.js, хотим сделать, чтобы можно было кликать по ссылками и соответствующие страницы менялись.

Поиск файла и переход в него в WebStorm Ctrl+Shift+N+вводим имя файла

В компоненте Navbar добавляем адреса первым двум ссылкам:

. . . const Navbar = () => {  
 return (  
 <nav className={s.nav}>  
 <div className={s.item}>  
 <a href="/profile">Profile</a>  
 </div>  
 <div className={`${s.item} ${s.active}`}>  
 <a href="/dialogs">Messages</a> . . .

При клике по этим ссылкам в браузере видим, что страница перезагружается, перенаправление идёт, а т.к. у нас SPA, сервер возвращает одну и ту же index.html, а вместе с ней приходят все имеющиеся компоненты/страницы в виде JS кода. А отображается то, что прописано в index.js, т.е. компонент App, который возвращает другие компоненты. Неважно какая ссылка /profile или /dialogs, если в App страница Dialogs, то отображается она. Меняет такое поведение работа с routing (маршрутизация).

Нужна система, проверяющая адрес/путь, реагирующая его изменение и загружающая нужный компонент. Среди пакетов предоставляемых nodeJS пока нет пакета, позволяющего настроить routing, разработчики CRA не включили этот модуль. Для установки этого пакета <https://reactrouter.com/> можно прописать npm i react-router-dom, тогда пакет появится в node-modules, начиная с npm 5 установленные модули по умолчанию добавляются как зависимости в package.json, при работе с более ранней версией npm, добавляем флаг –save. Модуль установлен.

После установки React router, можно использовать специальный компонент <Route /> - маршрут. Настраиваем <Route /> для каждого компонента согласно документации ( в уроке синтаксис для версии React router ниже 6 ), в атрибуте path указываем url компонента, который прописывали в ссылках в файле Navbar.jsx:

. . . const Navbar = () => {  
 return (  
 <nav className={s.nav}>  
 <div className={s.item}>  
 <a href="/profile">Profile</a>  
 </div>  
 <div className={`${s.item} ${s.active}`}>  
 <a href="/dialogs">Messages</a>  
 </div>  
 <div className={s.item}>  
 <a href="/news">News</a>  
 </div>  
 <div className={s.item}>  
 <a href="/music">Music</a>  
 </div>  
 <div className={s.item}>  
 <a href="/settings">Settings</a>  
 </div>  
 </nav>);  
} . . .

Компонент Route анализирует путь, ищет совпадения url , если совпадение есть, отрисовывается компонент, который мы передаём <Route />в атрибуте element. Тэг нашего компонента помещается в {}:

. . . import {BrowserRouter, Routes, Route} from "react-router-dom";

const App = () => {  
 return (<BrowserRouter>  
 <div className='app-wrapper'>  
 <Header/>  
 <Navbar/>  
 <div className='app-wrapper-content'>  
 <Routes>  
 <Route path='/profile' element={<Profile/>}/>  
 <Route path='/dialogs' element={<Dialogs/>}/>  
 <Route path='/news' element={<News/>}/>  
 <Route path='/music' element={<Music/>}/>  
 <Route path='/settings' element={<Settings/>}/>  
 </Routes>  
 </div>  
 </div>  
 </BrowserRouter>);  
} . . .

Продублировать текущую строку в WebStorm Ctrl+D

Согласно документации, все компоненты <Route /> должны быть помещены в обёртку <Routes>, а функция/компонент App должен быть обёрнут в <BrowserRouter>. Все обозначенные обёртки/компоненты должны быть импортированы из react-router-dom. WebStorm импортирует их автоматически при использовании, но Routes пришлось дописывать вручную.

Домашнее задание: добавить оставшиеся компоненты News, Music, Settings, настроить для них Route. Выполнено выше.

Сейчас при переходе по ссылкам, страница перезагружается полностью. Это можно отследить по консоли разработчика в браузере→Network после загрузки страницы, при клике по ссылкам, видно как список файлов меняется. Так происходит, потому что используются классические ссылки с атрибутом href, стандартное поведение ссылки – перезагрузить страницу при переходе на другой url, что увеличивает расход трафика, SPA так работать не должно.

**20. Уроки React JS (NavLink) - react курсы бесплатно**

Задача – не перезагружая страницу менять адрес в адресной строке браузера.

Route всегда «слушают» изменения в адресной строке и подгружают либо убирают соответствующие компоненты.

Чтобы выполнить задачу, используем специальные тэги. Вместо <a> внутри Navbar.jsx используем компонент/тэг <NavLink>, которые импортируюется из react-router-dom. Вместо атрибута href, <NavLink> использует атрибут to . Атрибуты по итогу превращаются в props.

<https://v5.reactrouter.com/web/api/NavLink>

Содержимое Navbar.jsx:

. . .import {NavLink} from "react-router-dom";  
  
const Navbar = () => {  
 return (  
 <nav className={s.nav}>  
 <div className={s.item}>  
 <NavLink to="/profile">Profile</NavLink>  
 </div>  
 <div className={`${s.item} ${s.active}`}>  
 <NavLink to="/dialogs">Messages</NavLink>  
 </div>  
 <div className={s.item}>  
 <NavLink to="/news">News</NavLink>  
 </div>  
 <div className={s.item}>  
 <NavLink to="/music">Music</NavLink>  
 </div>  
 <div className={s.item}>  
 <NavLink to="/settings">Settings</NavLink>  
 </div>  
 </nav>);  
} . . .

Перейдя на страницу в консоль разработчика→Network убеждаемся, что при переходе по ссылкам больше ничего не приходит. SPA со всем скриптами и компонентами загружен единожды и целиком. Задача выполнена.

Проинспектировав разметку в браузере, видим, что ссылки на все страницы являются обычными ссылками с href. Из JS вспоминаем, что на ссылку можно повесить обработчик события, подменить путь (почитать про history API позволяет делать изменения в адресной строке, появился в html5), предотвратить стандартное поведение, т.е. переход по указанному url/redirect + перезагрузку. Выходит, что компонент/тэг <NavLink> - это обёртка, за которой скрывается функция, выполняющая обозначенные выше действия.

Открыть навигационное меню в WebStorm Alt+F1

Стилизуем ссылки по новому. Хотим, чтобы они были белыми и без подчёркивания, а не становились фиолетовыми после перехода – это их стандартное поведение. Активная ссылка должна быть золотой.

Видим, что при клике по ссылками у активной появляется класс active, его попутно добавляет <NavLink>, но у нас будут свои классы активности. В этой части урока был предложен устаревший синтаксис, для React router v6 порядок действий такой:

Переписываем css правила в Navbar.module.css, чтобы обозначить 2 класса активности ative и incative и убрать подчёркивание у ссылок, вложенных в item:

. . .

.item a{  
 text-decoration: none;  
}  
  
.inactive{  
 color: white;  
}  
  
.active {  
 color: gold;  
}

Возвращаемся в Navbar.jsx, у <NavLink> есть атрибут= props className, который может принять не только строку но и функцию, причём в эту функцию будет передан параметр в виде объекта, у этого объекта есть свойство isActive, которое можно деструктурировать. Свойство isActive проверяет активна ссылка или нет и соответственно хранит значение true/false. Используя тернарный оператор, можно проверить isActive, если в нём true – добавить класс s.active, иначе s.inactive :

. . . const Navbar = () => {  
 return (  
 <nav className={s.nav}>  
 <div className={s.item}>  
 <NavLink to="/profile" className={({isActive}) => isActive ? s.active : s.inactive}>Profile</NavLink>  
 </div>  
 <div className={s.item}>  
 <NavLink to="/dialogs" className={({isActive}) => isActive ? s.active : s.inactive}>Messages</NavLink>  
 </div>  
 <div className={s.item}>  
 <NavLink to="/news" className={({isActive}) => isActive ? s.active : s.inactive}>News</NavLink>  
 </div>  
 <div className={s.item}>  
 <NavLink to="/music" className={({isActive}) => isActive ? s.active : s.inactive}>Music</NavLink>  
 </div>  
 <div className={s.item}>  
 <NavLink to="/settings" className={({isActive}) => isActive ? s.active : s.inactive}>Settings</NavLink>  
 </div>  
 </nav>);  
} . . .

Ссылки отображаются как мы того и хотели.

**21. Уроки React JS (Верстаем страницу Диалогов) - react курсы бесплатно**

Верстаем страницу в две колонки. В одной будут имена, в другой сообщения. Каждое имя – это отдельный <div>. Аналогично, каждое сообщение будет в отдельном <div>. Dialogs.jsx:

. . . const Dialogs = () => {  
 return (  
 <div className={s.dialogs}>  
 <div className={s.dialogsItems}>  
 <div className={s.dialog + ' ' + s.active}>  
 Dimych  
 </div>  
 <div className={s.dialog}>  
 Andrey  
 </div>  
 <div className={s.dialog}>  
 Sveta  
 </div>  
 <div className={s.dialog}>  
 Sasha  
 </div>  
 <div className={s.dialog}>  
 Victor  
 </div>  
 <div className={s.dialog}>  
 Valera  
 </div>  
 </div>  
 <div className={s.messages}>  
 <div className={s.message}>Hi</div>  
 <div className={s.message}>How are you ?</div>  
 <div className={s.message}>Bye</div>  
 </div>  
 </div>  
 );  
} . . .

Выделить все похожие слова одновременно WebStorm выбрать слово + Ctrl+Shift+Alt+j

Вернуться в предыдущее место WebStorm Ctrl+Alt+←

Переключаться между вкладками вверху WebStorm Ctrl+Tab

Делить на две колонки будет используя css grid. Псевдонимы в этот раз не назначаем, а просто задаём ширину колонок для двух вложенных элементов:

.dialogs {  
 display: grid;  
 grid-template-columns: 2fr 10fr;  
}

Для двух основных блоков dialogsItem и messages задаём padding: 10px. Активный элемент Dimych делаем золотым:

.dialogsItems .active {  
 color: gold;  
}

**22. Уроки React JS (Route exact) - react курсы бесплатно**

На странице Dialogs есть список диалогов (имена), нужно, чтобы при клике по диалогу отображались сообщения из соответствующего диалога. url должен меняться по принципу dialogs/3, где 3 – это id диалога/человека. Изменение пути нужно для того, чтобы при отправлении ссылки на диалог другому человеку, его окружение подстраивалось под эту ссылку.

Для реализации используем <NavLink>:

. . . <NavLink to='/dialogs/1'>Dimych</NavLink> . . .   
. . . <NavLink to='/dialogs/2'>Andrey</NavLink> . . .  
. . . <NavLink to='/dialogs/3'>Sveta</NavLink> . . .  
. . . <NavLink to='/dialogs/4'>Sasha</NavLink> . . .  
. . . <NavLink to='/dialogs/5'>Victor</NavLink> . . .  
. . . <NavLink to='/dialogs/6'>Valera</NavLink> . . .

Теперь после клика по диалогу видим пустую страницу, всё из-за prop exact компонента <Route>. В React router v6 этот prop отсутствует, но как бы true по умолчанию, это значит что путь, указанный в <Route> должен совпадать точно. Потому мы видим изменения в url на dialog/1, но страница пуста, потому что <Route> не знает какой компонент загружать, для него путь dialog/1 неизвестен. Нам нужно, чтобы url сопоставлялся дальше, т.к. у нас есть дочерние пути в компоненте Dialogs продолжающие url /dialogs, для этого нужно указать \* в конце пути в <Route>, это индикатор того, что путь должен совпадать более глубоко. Исходя из этого, дописываем путь для Dialogs в App.js:

. . . <Route path='/dialogs/\*' element={<Dialogs/>}/> . . .

Теперь <Route> отобразит Dialogs при клике по ссылке Messages, и продолжит отображать его, дописывая /1-6 в адресной строке, при клике по диалогам (именам). Подпути работают.

К вопросу о колонке с сообщениями компонента Dialogs - не необязательно при смене url, нужно создавать новый <Route> и подгружать другой компонент, просто при клике по диалогам должны показываться другие сообщения, но не другие компоненты. Существуют подроутинги (не для этой ситуации) и подкомпоненты (будут использованы позже).

**23. Уроки React JS (props для Диалогов) - react курсы бесплатно**

Будем разбивать компонент Dialogs на подкомпоненты.

Видим дублирование кода, в блоке с className={s.dialogsItems} :

<div className={s.dialog}>  
 <NavLink to='/dialogs/3'>Sveta</NavLink>  
</div>

И таких 6 штук. Этот <div> можно вынести в отдельный компонент, у которого меняется адрес и меняется заголовок.

Аналогичное дублирование в блоке с className={s.messages} :

<div className={s.message}>Hi</div>

Так же можно вынести в отдельный компонент с приходящими props.

В одном файле допускается создавать несколько подкомпонентов. Выбор выносить их в отдельный файл или нет, должен зависеть от того используются ли это подкомпоненты где-то ещё кроме этого файла или нет, универсален он или нет. Вообще это плохо.

Рефакторим компонент Dialogs в этом же файле, разбивая дублирующийся код на подкомпоненты:

. . . const DialogItem = (props) => {  
 let path = `/dialogs/${props.id} `;  
  
 return (<div className={s.dialog + ' ' + s.active}>  
 <NavLink to={path}>{props.name}</NavLink>  
 </div>);  
}  
  
const Message = (props) => {  
 return <div className={s.message}>{props.message}</div>;  
}  
  
const Dialogs = () => {  
 return (  
 <div className={s.dialogs}>  
 <div className={s.dialogsItems}>  
 <DialogItem name="Dimych" id="1"/>  
 <DialogItem name="Andrey" id="2"/>  
 <DialogItem name="Sveta" id="3"/>  
 <DialogItem name="Sasha" id="4"/>  
 <DialogItem name="Victor" id="5"/>  
 <DialogItem name="Valera" id="6"/>  
 </div>  
 <div className={s.messages}>  
 <Message message="Hi"/>  
 <Message message="How are you ?"/>  
 <Message message="Bye"/>  
 <Message message="Yo"/>  
 <Message message="Yo"/>  
 <Message message="Yo"/>  
 </div>  
 </div>);  
} . . .

Использование props в подкомпонентах позволяет сделать их более гибкими. Путь для всех DialogItem был один и тот же /dialogs/1-6 , отличается лиш id, поэтому мы вынесли этот путь в отдельную переменную path, где он дополняется id, который приходит из props.

Код сократился, стал компактнее и красивее, а детальная логика икапсулирована внутри компонентов DialogItem и Message.

Рефакторинг кода – важна процедура. Если дублируешь/говнокодишь/сложно, выносим это в отдельные подкомпоненты.

Перейти к коду функции/компонента в Webstorm Ctrl+B или Ctrl+клик

Рефакторим компонент Profile, выделяя блок с картинкой, аватаркой и описание в отдельный компонент ProfileInfo, для этого создаём отдельную папку ProfileInfo внутри Profile с файлами ProfileInfo.jsx и ProfileInfo.module.css

. . . const ProfileInfo = () => {  
 return (  
 <div>  
 <div>  
 <img src=". . ." />  
 </div>  
 <div>  
 <img src=". . ." alt="avatar" />  
 <div>Description</div>  
 </div>  
 </div>);  
} . . .

Компонент Profile теперь выглядит как :

const Profile = () => {  
 return (  
 <div>  
 <ProfileInfo/>  
 <MyPosts/>  
 </div>);  
}

Вносим стилистические улучшения. Добавляем небольшие отступы по 10 px для: аватар + описание профиля (ProfileInfo), компонента MyPosts. В MyPosts выносим заголовок в тэг <h3>, <textarea> и кнопку помещаем в отдельные <div>, для <div> с постами задаём margin-top: 10px.

**24. Уроки React JS (UI - BLL, выносим данные в сторону) - react курсы бесплатно**

Есть 2 компонента, которые нужно рефакторить Dialogs и MyPosts. В Dialogs присутствует «хардкод» - т.е. вместо данных с сервера, забиты статические данные. Временно с сервером работать не будем, но и хардкод нужно смягчить, вынося данные в сторону.

В Dialogs данные парные - name="Dimych" id="1", т.е. одно «данное» - это name + id, это выглядит как объект (пары ключ: значение). Если посмотреть со стороны, то это не один объект а совокупность объектов с однотипными данными внутри. Совокупность объектов, постов, треков, сообщений – это массив данных, значит нам не нужна куча объектов, а нужен один массив, а обращаться к объектам будем по индексатору.

В идеале нужно разграничивать код на 2 уровня/мира:

1) мир данных – BLL business logic layer – истина

2) мир UI user interface - компоненты

Суть – имея истину, перерисовываем UI. Компонент получает данные через props и отрисовывает.

Выносим данные для Dialogs в массив внутри компонента Dialogs:

. . . let dialogsData = [  
 {id: 1, name: 'Dimych'},  
 {id: 2, name: 'Andrey'},  
 {id: 3, name: 'Sveta'},  
 {id: 4, name: 'Sasha'},  
 {id: 5, name: 'Victor'},  
 {id: 6, name: 'Valera'}

]; . . .

Пока данные остаются внутри UI, будут вынесены позже, договариваемся, что они как бы пришли из props. Использование данных из такого массива объектов выглядит как:

. . .<div className={s.dialogsItems}>  
 <DialogItem name={dialogsData[0].name} id={dialogsData[0].id}/>  
 <DialogItem name={dialogsData[1].name} id={dialogsData[1].id}/>  
 <DialogItem name={dialogsData[2].name} id={dialogsData[2].id}/>  
 <DialogItem name={dialogsData[3].name} id={dialogsData[3].id}/>  
 <DialogItem name={dialogsData[4].name} id={dialogsData[4].id}/>  
 <DialogItem name={dialogsData[5].name} id={dialogsData[5].id}/>  
 </div> . . .

Технически из массива объектов/данных мы должны получить массив JSX элементов, т.е. нужно сделать преобразование одного массива в другой массив. Такое преобразование достигается с помощью функции map, преобразование будет сделано в следующем уроке.

Продолжаем подготовку к преобразованию, переработав сообщения, для этого создадим массив с набором сообщений. Сообщения, диалог, пользователи и другая информация – это сущности = бизнес сущности = записи в базе данных, у каждой такой записи есть id. id используется для взаимодействия с сущностью – выбрать/пометить/удалить/идентифицировать среди других и т.д. Поэтому у каждого сообщения должен быть id. В наших сообщениях мы временно не будем использовать id.

Можно было бы сделать для сообщений массив строк, но если копнуть глубже, то сообщение – это не просто текст, это ещё и дата отправки, отправитель, id и т.д. Т.е. сообщение – это объект. Массив сообщений:

. . . let messagesData = [  
 {id: 1, message: 'Hi'},  
 {id: 2, message: 'How are you ?'},  
 {id: 3, message: 'Bye'},  
 {id: 4, message: 'Yo'},  
 {id: 5, message: 'Yo'},  
 {id: 6, message: 'Yo'}  
]; . . .

id могут совпадать с массивом для dialogs, т.к. это разные сущности, они хранятся в базе данных отдельно друг от друга и вообще, в реальности приходят с сервера.

Двигаться не по символам а по словам зажать Ctrl + стрелки

Переработанные сообщения:

. . . <div className={s.messages}>  
 <Message message={messagesData[0].message}/> . . .

Аналогично рефакторим MyPosts. Аналогично с сообщениями, посты – это другая таблица в базе данных, поэтому id могут пересекаться. Учитываем, что кроме сообщения и id, каждый пост характеризуется количеством лайков:

. . . let postsData = [  
 {id: 1, message: "Hi, how are you ?", likeCounter: 12},  
 {id: 2, message: "It's my first post", likeCounter: 20},  
]; . . .

Каждый объект характеризует какую то сущность в БД = бизнес объект в виртуальном мире = диалог = сообщение = новость = трек и т.д.

Посты, получающие данные из массива postsData:

. . . <div className={s.posts}>  
 <Post message={postsData[0].message} likeCounter={postsData[0].likeCounter}/>  
 <Post message={postsData[1].message} likeCounter={postsData[1].likeCounter}/>  
</div> . . .

Таким образом близимся к соблюдению принципа single responsibility – принцип единственной ответственности, самый главный принцип в программировании по мнению Димыча. Суть – не нагружать «исполнители»/функции/объекты чрезмерным количеством дел. Лучше пусть хорошо делают что-то одно. Из этой концепции и родился BLL + UI. Далее узнаем, что в нашем случае BLL’ом является Redux, а UI’ем – React.

Почитать про AJAX запросы на сервер.

В этом уроке мы сделали первый шаг – выделили данные в объекты и вынесли в сторону и передаём компонентам в props данные из объектов. Минус в том, что количество отображаемых компонентов/диалогов/постов мы пока захардкодили. В реальности мы не будет знать для какого количества компонентов придут данные, поэтому нужно делать автоматическое преобразование из массива данных в массив тэгов/компонентов с помощью метода массива .map().