|  |
| --- |
| Курсовая работа  **РАЗРАБОТКА И РЕАЛИЗАЦИЯ GUI ИНТЕРФЕЙСА ИНТЕРАКТИВНОГО МАГАЗИНА КОМПЬЮТЕРНОЙ ТЕХНИКИ** |
| Студент гр. ИС-17 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ К.А. Атрощенко  подпись, дата  Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ П.Г. Деркаченко  подпись, дата  Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_ г. с оценкой «\_\_\_\_\_\_\_\_\_\_\_\_\_\_».  Члены комиссии \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_  подпись Фамилия И.О.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_  подпись Фамилия И.О. |

Витебск 2020

|  |
| --- |
|  |

Оглавление

[Введение 2](#_Toc33549412)

[**1** **Теоретическая часть** 4](#_Toc33549413)

[**1.1** **Библиотека React** 4](#_Toc33549414)

[**1.2** **Выбор языка программирования** 6](#_Toc33549415)

[**1.3** **Препроцессор SCSS** 7](#_Toc33549416)

[**1.4** **Обоснование выбора среды разработки** 9](#_Toc33549417)

[**2** **Практическая часть** 12](#_Toc33549418)

[**2.1** **Инициализация проект** 12](#_Toc33549419)

[**2.2** **Создание компонентов** 15](#_Toc33549420)

[**2.3** **Компонент <ContactInformation/>** 17](#_Toc33549421)

[**2.4** **Компонент <Header/>** 21](#_Toc33549422)

[**2.5** **Компонент <Main/>** 22](#_Toc33549423)

[**2.6** **Компонент <OurProduction/>** 24](#_Toc33549424)

[**2.7** **Компонент <DeliveryInfotmation/>** 27](#_Toc33549425)

[**2.8** **Компонент <Partners/>** 30](#_Toc33549426)

[**2.9** **Компонент <Info/>** 33](#_Toc33549427)

[**2.10** **Компонент <Footer/>** 34](#_Toc33549428)

[**2.11** **Рендеринг компонентов приложения** 35](#_Toc33549429)

[**Заключение** 37](#_Toc33549430)

[**Список используемых источников** 38](#_Toc33549431)

[**Содержание электронного носителя** 39](#_Toc33549432)

# Введение

Каждый день миллионы людей активно используют свои мобильные устройства. Они печатают, кликают, листают, перетаскивают и меняют размер изображений на своих сенсорных дисплеях. Естественно, так было не всегда.

В 70-х годах компания Xerox создала первый ***графический пользовательский интерфейс* GUI**. Основная идея пользовательского интерфейса «*что видишь на экране, то и получишь*».

Инженеры в Xerox пытались таким способом упростить подготовку документов к печати.

Вскоре после этого Apple позаимствовала эту идею в обмен на некоторое количество акций своей новой фирмы. В 1984 году свет увидел новый Macintosh – компьютер для пользователей, с помощью которого люди могли выполнять какие-то повседневные задачи.

Идея оказалась настолько успешной, что в очень скором времени ее использовали в компании Microsoft, создав свою собственную платформу Windows.А появление [Windows 95](http://www.iphones.ru/iNotes/30596) без преувеличения сотворило революцию в сфере ПК.

С течением времени появились и другие устройства призванные облегчить жизнь пользователей. Например, музыкальные плееры. На смену CD и MP3 плеерам пришел iPod ознаменовав тем самым доступность и портативность музыки в цифровом формате.

Примерно в это же время начали появляться сотовые телефоны. Первые их модели были похожи на простые стационарные телефоны. С развитием технологий сотовые телефоны начали оснащать ЖК-дисплеями. Вскоре их владельцы смогли отправлять текстовые сообщения по сотовой сети.

Через некоторое время такие разработки как компьютер, сотовый телефон и MP3 стали вполне самодостаточными, состоявшимися продуктами. В Apple воспользовались этим и объединили их в одном устройстве iPhone.

 На фоне этого, iPhone оказался просто-таки революционным продуктом, в своей сфере, с его емкостным сенсорным дисплеем, с поддержкой multitach. Другие телефоны пока еще были оснащены пластиковыми резистивными дисплеями, и их пользователям, чтобы реализовать тот или иной жест, приходилось сильно надавливать на экран. Емкостный же дисплей реагирует на электроимпульсы в человеческом теле, и в сильных нажатиях просто нет необходимости.

Однако отображение контента все еще желало лучшего. Все изменилось с выходом Retina-дисплей. Этот дисплей обладает высокой плотностью пикселей, благодаря чему текст и изображения стали видны намного четче – будто на странице глянцевого журнала.

Основная суть GUI (графический пользовательский интерфейс) использовать вещи которые уже знакомы и понятны пользователям, для того что бы научить их обращаться с новыми вещами.

Возьмем в качестве примера приложение калькулятор. В этом графическом интерфейсе можно уловить схожесть со старым привычным калькулятором.

Интерфейс пользователя и интерфейс аппаратной части гармонично сосуществуют, достигнув симбиоза. Лучшее ПО сливается с прекрасно проработанными аппаратными компонентами, и они вместе создают само устройство. Увидев его, пользователь уже не задумывается над тем, как оно работает. В мыслях его сразу формируются картинки того, как его можно использовать.

# Теоретическая часть

## Библиотека React

React – это open source библиотека компонентов пользовательского интерфейса (UI-компонентов) т.е. данная библиотека занимается отображением UI элементов и ничего больше. Данная библиотека была создана компанией Facebook. UI-компоненты, создаются на языке JavaScript. Этот подход, называется *созданием составного пользовательского интерфейса*. Если просто, то React создан для упрощения создания пользовательских интерфейсов. Он разбивает пользовательский интерфейс на компоненты. Это так же естественно, как разбивать JavaScript код на функции. Функция - это блок кода, который я могу использовать в разных частях приложения. Точно так же и UI компонент – это блок пользовательского интерфейса с каким-то внешним видом, с каким-то поведением которые можно использовать в разных частях приложения. И точно так же как хорошая функция которая должна быть маленькая и независимая, для того что бы ее можно было легко обновлять и тестировать, хороший UI компонент тоже должен быть компактным и независимым от всего остального приложения что бы его можно было легко обновлять и тестировать.

React построен на идее UI компонентов. Он дает инструменты для того, что бы эффективно создавать UI компоненты, и затем строить из них приложение.

UI компоненты используются уже давно. Однако у React есть две сильные стороны, которые отличают его от предыдущих альтернатив.

* React использует язык JSX.

JSX –это язык расширения который позволяет комбинировать JS код с разметкой которая похожа на HTML. Это позволяет более существенно писать UI логику.

* Это алгоритм который React использует внутри себя для того что бы понимать, какие именно части вэб-сайта нужно обновить. К примеру, пользователь кликает на какой-нибудь компонент и этот компонент изменяет свой внешний вид. Или приходят новые данные с сервера и нужно обновить эти данные на UI.

Так вот reconciliation algorithm, который работает внутри React, делает достаточно сложных вещей для того что бы, найти именно те блоки на странице которые требуют обновления и обновить только их, не трогая те блоки которые не изменились. Именно этот подход делает React приложения достаточно быстрыми.

За последние несколько лет библиотека приобрела невероятную популярность и является обязательным инструментом для многих команд и разработчиков, создающих динамические пользовательские интерфейсы.

Появление React в корне изменило обстановку в веб-программировании: сообщество React положило начало многих подходов, понятий и паттернов проектирования.

React — свежий взгляд на разработку компонентов пользовательского интерфейса. Это новое поколение библиотек презентационного уровня. В сочетании с моделью и библиотекой маршрутизации React может заменить Angular, Backbone или Ember в технологическом стеке веб-программирования и мобильной разработки.

Разработчики всегда ищут более эффективные способы разработки пользовательских интерфейсов. Большая часть коммерческой ценности лежит в пользовательских интерфейсах, серверная часть-всего лишь ресурс.

Времена, когда для обычной отладки кода frontend части приходилось набивать вызовами alert() прошло. С развитием интернет технологий развивается и сложность построения сайтов. Сайты превратились в веб-приложения со сложными интерфейсами, бизнес логикой и уровнями данных, которые требуют постоянных изменений и обновлений. Часто в реальном времени.

Для решения проблем, с построением сложных пользовательских интерфейсов было написано много библиотек, паттернов, пакетных менеджеров, которые я буду использовать в курсовом проекте.

Ядро React – это система, автоматически обрабатывающая пользовательский интерфейс, когда изменяются данные в его компонентах. Они вносят изменения, которые помогут перехватить, используя так называемые методы жизненного цикла. Системы React поддерживающие обновление пользовательского интерфейса, значительно упрощают создание надежных модульных компонентов, применяемых приложением.

React в первую очередь библиотека а не framework. Разница в том, что в framework есть масса других аспектов кроме самого UI. Это работа с сервером, валидация, управление глобальным состоянием, Unit test и масса других аспектов. В крупных UI framework-ах вроде Angular, Ember, Aurelia все эти аспекты покрыты из коробки.

React в свою очередь отвечает только за UI компоненты. Но это не страшно т.к. есть масса других библиотек, которые покрывают эти аспекты и которые отлично работают с React. Как раз в этом и заключается гибкость React. Не понравилось, как работает библиотека HTTP запросов. Нет проблем, заменю ее на другую. С другими библиотеками дела обстоят точно так же.

Для интерпретации и компиляции кода с последующим выведением контента в браузер, весь код проходит через виртуально объектную модель документа DOM. DOM - это программный интерфейс, который позволяет JavaScript программам взаимодействовать с различными типами документов (HTML, XML, SVG).

Как работает DOM? Она чем-то схожа с другим миром программного обеспечения-трехмерными играми. В таких играх иногда используется рендеринг, который работает примерно так: получить информацию с игрового сервера, отправить его в игровой мир (визуальное представление которое видит пользователь), установить какие изменения стоит внести в этот мир, и тогда графическая карта определит какой минимум изменений. Однако преимущества этого подхода заключается в том, что нужны только ресурсы для работы с инкрементными изменениями, внести которые, как правило, намного быстрее, чем если бы пришлось обновлять все. Таким способом React пытается наиболее эффективным способом обновлять пользовательский интерфейс.

React создает и поддерживает виртуальную модель DOM в памяти, а средство рендеринга обрабатывает обновление в DOM браузера на основе изменений. React может выполнять интеллектуальные обновления и работать только с измененными частями данных, поскольку способна использовать *эвристическую сверку* для вычисления того, какие части DOM в памяти требуют изменений в фактической модели DOM.

## Выбор языка программирования

Подавляющее большинство веб-сайтов используют JavaScript, и все современные веб-браузеры – для настольных компьютеров, игровых приставок, электронных планшетов и смартфонов – включают интерпретатор JavaScript, что делает JavaScript самым широко применимым языком программирования. JavaScript входит в тройку технологий, которые должен знать любой разработчик: язык разметки HTML, позволяющий определять содержимое веб-страниц, язык стилей CSS, позволяющий определять внешний вид веб страниц, и язык программирования JavaScript, позволяющий определять поведение веб-страниц.

JavaScript является высокоуровневым, динамическим, не типизированным и интерпретируемым языком программирования, который хорошо подходит для программирования в объектно-ориентированном и функциональном стилях.

Ядро языка JavaScript определяет минимальный прикладной интерфейс для работы с текстом, массивами, датами и регулярными выражениями, но в нем отсутствуют операции ввода-вывода. Ввод и вывод (а также более сложные возможности, такие как сетевые взаимодействия, сохранение данных и работа с графикой) перекладываются на «окружающую среду», куда встраивается JavaScript. Обычно роль окружающей среды играет веб-браузер.

С помощью JS можно управлять HTML-элементами, которые определяют содержимое вэб-страниц, а также каскадными таблицами стилей, которые определяют представление содержимого.

JS позволяет не только управлять содержимым и оформлением HTML – документов в браузерах, но и определять поведение этих документов с помощью обработчиков событий. Обработчик событий – это функция JS, которая регистрируется в браузере и вызывается браузером, когда возникает событие определенного типа. Таким событием может быть щелчок мышью или нажатие клавиши, или движение пальцами на экране смартфона.

Прикладной интерфейс, написанный на чистом JS, является достаточно сложным и испытывает проблемы с совместимостью между браузерами. По этим причинам большинство программистов предпочитают использовать клиентские библиотеки или фреймворки, упрощающие программирование. Наиболее популярна из этих библиотек – библиотека React которую я буду использовать в данном курсовом проекте.

## Препроцессор SCSS

В 2007 году появился препроцессор SASS. Разработчиком Sass является француз Hugo Giraudel.

Существует два синтаксиса написания кода SASS и SCSS. В SASS опущены фигурные скобки и точка с запятой после правила. Структура кода строится на основе отступов (ведь фигурных скобок в нем нет) и здесь следует быть внимательным, так как в качестве отступа может выступать, как табулятор (TAB — 4 пробела), так и обычные пробелы (обычно это двойной пробел).

Из-за этих особенностей SASS, многие начинающие разработчики не соблюдают нужное количество отступов, что приводит к ошибкам или банальному не присваиванию свойств, прописанных в селекторе.

Ввиду этого, разработчики Ruby on Rails выпустили препроцессор SCSS. В котором используется CSS подобный синтаксис.

SCSS - это метаязык на основе css, предназначенный для увеличения уровня абстракции CSS кода и упрощения файлов каскадных таблиц стилей.

Сам по себе процесс CSS верстки довольно интересное занятие, но иногда оно становится нудным и кропотливым. Препроцессоры созданы для того, чтобы избавить разработчика от большинства рутинных задач при написании кода, тем самым повысив скорость и качество работы.

Современный CSS -мощь, а в комбинации с препроцессорами – вообще боевая машина для оформления контента на страницах. Вкратце: препроцессор добавляет в возможности CSS логику программирования. Теперь стилизация выполняется не обычным перечислением стилей, а с помощью нескольких простых приемов и подходов: переменных, функций, миксинов, циклов, условий. Вдобавок появилась возможность использовать математику.

Весь код SCSS компилируется в CSS, что бы браузеры могли его понимать и корректно отображать. В настоящее время браузеры не поддерживают работу с SCSS напрямую.

Возможности SCSS

1. Вложенность – SCSS позволяет вкладывать правила CSS друг в друга. Вложенные правила применяются только для элементов, соответствующих внешним селекторам.
2. Переменные – в стандартном CSS тоже есть понятие переменные, но в SCSS с ними можно работать немного по-другому. Например, повторять их через @for. Или генерировать свойства динамически.
3. Математические операции – можно складывать, вычитать, умножать и делить значения CSS.
4. Тригонометрия – SCSS позволяет писать собственные (синусоидальные и косинусоидальные) функции, используя только синтаксис SCSS, подобно тому, как это можно делать в других языках вроде JavaScript.
5. В SCSS есть логические конструкции, которые называются логические директивы: if, for, each, while. В итоге на выходе будет обычный CSS.
6. Миксины (примеси) — можно один раз создать набор CSS-свойств и работать с ними повторно или смешивать с другими значениями. Миксины можно использовать для создания отдельных тем одного макета. Примеси также могут содержать целые CSS-правила или что-либо другое, разрешённое в SCSS-документе. Они даже могут принимать аргументы, что позволяет создавать большое разнообразие стилей при помощи небольшого количества миксинов.
7. Функции***—*** можно создавать определения CSS в виде функций для многократного использования.
8. SCSS может работать с различными типами данных: строки, списки, ассоциативные массивы и даже цвет – это тип данных в SCSS.
9. Переменные в SCSS можно использовать при именовании селекторов и при именовании CSS свойств.
10. Переменные в SCSS могут иметь значения по умолчанию, что очень часто позволяет избегать ошибок при написании кода.
11. Препроцессор SCSS кэширует свои файлы, то есть он сохраняет в виде CSS самые часто используемые куски кода и не компилирует их повторно.

Как видите CSS достаточно эволюционировал что бы порой заменять JavaScript. Это упрощает работу и экономит время.

## Обоснование выбора среды разработки

Visual Studio code – это легкий и мощный редактор исходного кода. При правильной настройке его можно превратить в настоящую IDE. Он поставляется со встроенной поддержкой JavaScript, TypeScript и Node.js и имеет богатую экосистему расширений для других языков таких как С++, С#, Java, Python, PHP, Go и сред выполнения таких как .NET и Unity.

Как и многие другие редакторы кода, VS Code использует общий пользовательский интерфейс. В панели слева показывается древовидная иерархия папок и файлов – это структура проекта. Справа показано содержимое открытых файлов.

Интерфейс VS Code делится на пять основных областей.

* Редактор – основная область для редактирования файлов.
* Боковая панель – содержит различные виды что бы помочь при работе с проектом
* Строка состояния – информация об открытом проекте и файлах
* **Панель активности** - расположена в крайнем левом углу, она позволяет переключаться между представлениями и дает дополнительные индикаторы, зависящие от контекста, например, количество исходящих изменений, когда Git включен.
* **Панели** – Внизу я могу отображать различные панели под областью редактора для вывода или отладки информации, ошибок и предупреждений или встроенного терминала. Панель также можно перемещать вправо для увеличения вертикального пространства.

Для своего проекта я использую встроенный терминал для установки различных зависимостей, библиотек и т.д.

Для настройки VS code я использую следующие плагины:

1. Auto Close Tag - автоматически добавляет закрывающий тег при вводе закрывающей скобки открывающего тега. Область применения языки гипертекстовой разметки HTML/ XML, а так же для PHP, Vue, JavaScript, TypeScript, JSX и т.д.
2. Auto Complete Tag – объединяет функции автоматического закрытия тегов и автоматического переименования тегов.
3. Bracket Pair Colorizer – подсвечивает скобки разными цветами, что прекрасно помогает при навигации, визуализации вложенных функций. Можно настроить список цветов, а также определенный цвет для лишних скобок. Данная функция плагина прекрасно помогает при поиске ошибочно напечатанных скобок.
4. Code Spell Checker – данный плагин проводит базовую проверку орфографии которая хорошо работает с кодом Camel Case.
5. CSS Peek – Данный плагин настраивает навигацию между классами прописанными в HTML, JS, JSX и тд и классами прописанными в каскадной таблице. Для быстрого поиска классы в каскадной таблице нажимаю правой кнопкой мыши на классе прописанном в HTML и выбираю *Перейти к определению*, происходит переход к нужному классу в каскадной таблице стилей.
6. CSSTree validator – плагин проводит проверку CSS в соответствии со спецификациями W3C и реализациями браузера. Расширение подсвечивает неправильные свойства и их значения.
7. EditorConfig for VS Code- данный плагин в фоновом режиме проверяет кодировку каждого открытого файла. Контролирует табуляцию.
8. ESLint – статически анализирует код JavaScript. Находит ошибки. Многие проблемы, обнаруженные этим плагином, могут быть исправлены автоматически. Так же можно написать собственные правила, которые работают со встроенными правилами.
9. Git History Diff- данный плагин позволяет посмотреть детальную историю коммитов. Посмотреть какие изменения были внесены в файл, проект. Работа с ветками проекта и многое другое.
10. HTML CSS Support – этот плагин автоматически дополняет названия id классов на основе определений найденных в рабочей области или внешних файлов.
11. JavaScript (ES6) – это расширение содержит фрагменты кода для JavaScript в синтаксисе ES6.
12. JS Refactor – это инструмент автоматического рефакторинга Javascript для кода Visual Studio, созданный для упрощения и оптимизации процесса разработки. Он предоставляет обширный список автоматизированных действий, включая обычно необходимые: метод извлечения, извлечение переменной, встроенная переменная и псевдоним для переименования встроенного кода VS. JS Refactor также поддерживает множество общих фрагментов и связанных действий для упаковки существующего кода в общие выражения блоков.
13. Prettier - Code formatter - средство форматирования кода. Упорядочивает код.
14. SASS – подсвечивает синтаксис SASS, автоформатирует и заполняет код.
15. W3C Validation - Расширение для включения поддержки проверки W3C библиотекой. Это приложение требует установленной JDK. После установки нужно настроить переменные среды.

# Практическая часть

## Инициализация проект

Для компиляции кода из моего проекта мне понадобится интерпретатор Node. Этот интерпретатор построен на основе механизма V8 JavaScript разработанного компанией Google. Благодаря этому компилятору можно создавать различные серверные приложения, работать с базами данных и т.д.

Для инициализации проекта мне потребовалось установить стандартный пакет React путем ввода, во встроенном терминале VS Code, команды

*npm -i -g create-react-app.*

i-install

-g - global

Данная команда установит библиотеку React глобально на компьютер.

Далее в консоли я перехожу в нужную папку где я буду создавать свой проект.

*cd C:\Users\Konstantine\Documents\GitHub*

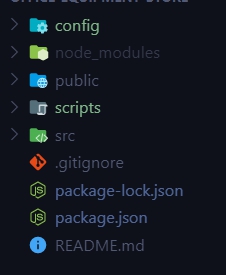
После этого создаю React проект.

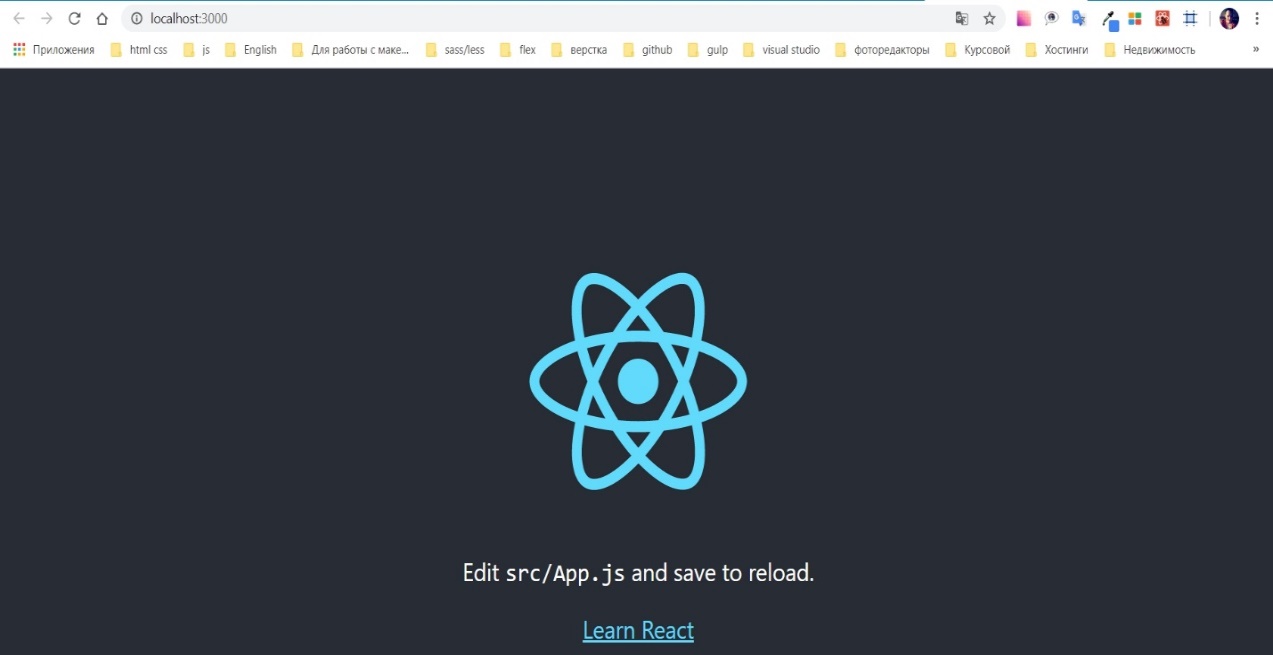
*create-react-app office-equipment-store.*

Где *create-react-app* команда создания, а *office-equipment-store* название проекта над которым я буду работать.

Для того что бы иметь доступ для редактирования файлов конфигурации WebPack, необходимо что бы в проекте появилась папка config. Для этого прописываю команду npm run eject и соглашаюсь с изменениями. В файле webpackDevServer.config.js я могу настраивать режимы разработки под себя.

После этого структура папок меняется.

Если все сработало корректно, то при вводе команды *npm start,* React соберет и запустит то самое маленькое приложение которое идет вместе с пакетом create-react-app. Запускается браузер на localhost:3000. Это именно тот host на котором Node.js интерпретирует код JavaScript. Запускается главная страница React.



Для того что бы мне начать разработку нужно удалить некоторые файлы из двух папок. В папке src лежит исходный код всего того, что относится к приложению. Вторая папка называется public и здесь находятся статические ресурсы.

В папке public удаляю следующие файлы:

**favicon.ico** - это иконка по умолчанию,

**manifest.json** - это файл который необходим для **android девайсов (Этот файл можно удалять по желанию).**

В папке src удаляю файлы которые запускают стандартное приложение. А именно:

**App.css,**

**App.js,**

**App.test.js,**

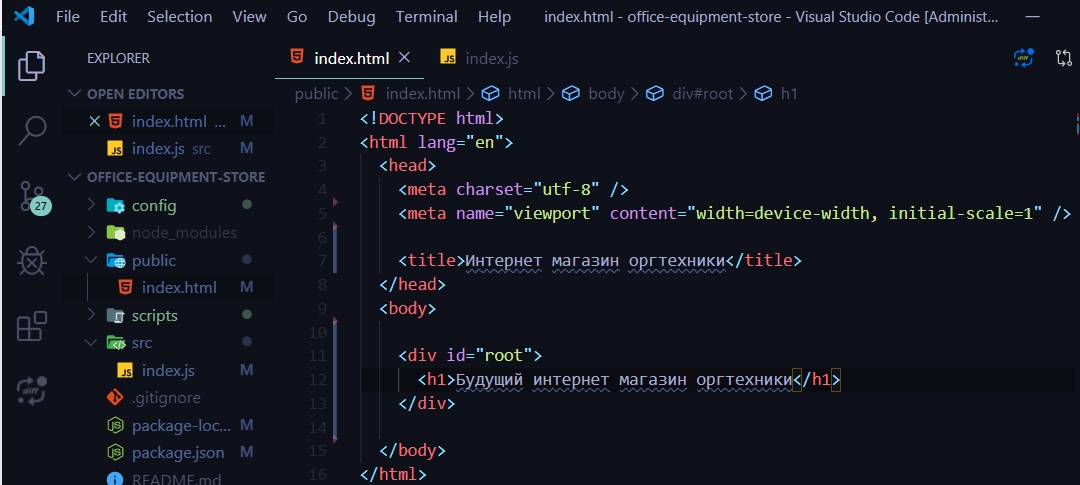
**index.css,**

**logo.svg,**

**registerServiceWorcer.js**.

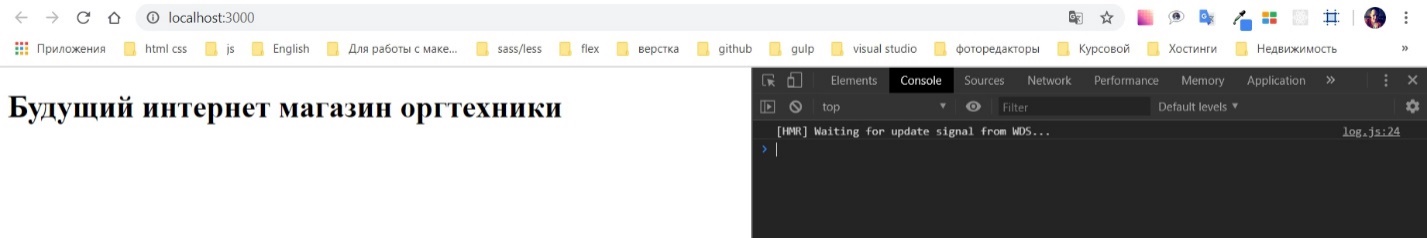
После удаления этих файлов, для того что бы в консоль браузера не выводились ошибки, необходимо почистить некоторые файлы.

В папке public файл index.html привожу к такому виду



В папке src в файле index.js удаляю весь код.

И вот теперь полигон для создания приложения готов. Проверяю нет ли ошибок.



Для создания React компонентов используются три библиотеки: React, React DOM и props-types. Для создания React компонентов используются три библиотеки: React, React DOM и props-types. Для установки необходимых зависимостей прописываю в консоль

* *npm install prop-types react-router-dom redux react-redux*
* Props-types – это библиотека разработки, которая поможет выполнить проверку типов данных, переданных компонентам.
* React DOM – это средство рендеринга для React, которое было отделено от основной библиотеки React, что бы лучше разделить задачи. Оно обрабатывает компоненты рендеринга DOM или строку для рендеринга на стороне сервера. Простыми словами организовывает переход по ссылкам в моем single page application, одностраничном приложении.
* Redux – ядро
* React-redux библиотека которая упрощает интеграцию react приложения с redux

Так же для написания стилей я буду использовать препроцессор scss для установки которого в консоли нужно прописать

* *npm install - -save-dev node-sass sass-loader*

Флаг - -save-dev означает что с данным пакетом я буду работать в режиме разработки т.е. файлы scss не пойдут в production. В production режиме будут скомпилированные css файлы.

Node-sass это движок.

Sass-loader с помощью этого loader происходит компиляция из scss в css. Все эти зависимости устанавливаются в файл package.json.

## Создание компонентов

React не просто использует новый подход к работе с изменением данных с течением времени; она так же фокусируется на компонентах как на парадигме для организации приложения. Компоненты - основополагающая единица React.

В качестве как ментального, так и визуального инструмента они позволяют добиться при проектировании и разработке пользовательских интерфейсов улучшенного, интуитивно более понятного структурирования в приложениях.

Полезно разбивать разные элементы интерфейса на части, которые можно собрать вместе, многократно применить и легко реорганизовать.

Пользовательские интерфейсы часто содержат элементы, которые используются многократно или перепрофилируются в других частях интерфейса. И даже если они не применяются повторно, то по крайней мере различаются. Эти отдельные элементы интерфейса можно рассматривать как компоненты.

Компоненты React хорошо инкапсулированные, многоразовые и составные. Эти характеристики обеспечивают простой и элегантный способ проектирования и создания пользовательских интерфейсов. Приложение состоит из четких сжатых групп, а не из спагетти-кода. Использование React для создания приложения похоже на сборку модели из деталей LEGO. В отличие от конструктора, детали создаются собственноручно и используются многократно.

Хорошо спроектированный компонент React должен быть достаточно автономным. Разбиение пользовательского интерфейса как правило, упрощает работу с разными частями приложения. Разграничение компонентов означает, что функциональность и организация могут быть четко определены, а их автономность - что их можно многократно использовать и легко перемещать.

Компоненты React соединены в древовидные структуры. Как и DOM -элементы они могут быть вложенными и содержать другие компоненты. А еще компоненты могут находится рядом друг с другом на одном и том же уровне.

Поскольку компоненты React самодостаточны их называют составными. Поскольку компоненты составные их можно применять в разных позициях кода приложения. Где бы они не использовались они позволяют сформировать тип отношений – родительский или дочерний. Если один компонент содержит другой, он считается родительским. Тот который находится внутри другого -дочерним(потомком). Все родительские элементы находятся на одном уровне и не связаны между собой. Связка всех компонентов происходит в компоненте App. А все компоненты находятся в одной родительской папке *components.*

Так же компонентный подход я использую для создания двумерных векторных изображений svg.

По правилам все подключаемые шрифты картинки и т.д. выносятся из основной иерархии в папку *assets*. В папке *assets* я создаю еще три папки *fonts* шрифты, папку для картинок *img*, и папку для двумерной векторной графики svg.

В папке svg создаю компоненты двумерной векторной графики. В каждом компоненте создаю три файла: файл с названием компонента и расширением js, файл стилей компонента scss который импортируется в файл с названием компонента и файл index.js в который импортируются предыдущие два файла.

Рассмотрим подробнее на примере компонента карты оплаты Maestro.

В файл maestro.js я импортирую библиотеку React и файл стилей maestro.scss.

Далее создаю функцию Maestro результатом которой возвращаю svg тег в котором содержаться векторные реквизиты картинки. Для того что бы этот файл заработал в него нужно внести некоторые поправки. А именно атрибуты xmls:xlink и атрибут xml:space сделать camel case т.е. каждое последующее имя с верхнего регистра xmlsXline и xmlSpace. А также, из тегов, удаляю атрибут fill который в векторной графике отвечает за присваивание цвета.

Далее в файле стилей maestro.scss, как и в каждом последующем созданном мной компоненте, я создаю переменные в которые помещаю используемые в данном компоненте цвета. Удобство заключается в том, что, если мне придется поменять всю цветовую гамму компонента, мне не придется искать все цвета в десятках, а может и тысячах селекторах scss. Всего лишь присвою новые цвета старым переменным.

Так же и со шрифтами. В каждом родительском компоненте я использую масштабируемые единицы шрифта REM (root em). По умолчанию 1rem = 16px. Что накладывает некоторые неудобства при расчете размера шрифта. Исходя из этого я в каждом родительском компоненте 1rem присваиваю 10px. И точно так же, как с переменными цвета, если я захочу поменять относительный размер шрифта всего компонента, мне нужно сделать это лишь в одном месте.

В селекторе maestro\_\_st-1 задаю ширину и высоту по 80px. Далее в селекторах maestro\_\_st-2, maestro\_\_st-3, maestro\_\_st-4 задаю цвета для картинки. Таким образом я создаю каждый svg компонент.

## Компонент <ContactInformation/>

И так, для того что бы реализовать данный компонент, я импортирую в него зависимости, а именно: библиотеку React, все свойства объекта Link из пакета react – router -dom.

После я импортирую в компонент созданные мной другие компоненты (потомки), и файл каскадной таблицы стилей. Компонент Magnifaer это svg картинка лупы, которую я встраиваю в кнопку поиска.

Далее создаю стрелочную функцию ContactInformation. Результатом данной функции будет возврат JSX разметки. Так как данная функция одна в документе, сразу же, внизу документа, экспортирую ее по умолчанию.

По правилам React возвращаемый элемент должен быть обернут в родительский блок, контейнер div. Каждому родительскому контейнеру присваивается class. Классы в JSX разметке задаются как className. Обусловлено это тем, что при рендеринге компонентов возникают конфликты. Классы нужны для придания стилистики блочно-строчных элементов компонента в каскадной таблице стилей.

Главным селектором компонента **ContactInformation** является селектор ***contact\_\_information***. В теле селектора прописываю следующие правила. Первым делом делаю контейнер гибким **display: flex**. Контейнеры потомки выстаиваю в строку **flex-direction: row**. Распределяю контейнеры равномерно удаленно друг от друга **justify-content : space-around.** Задаю фоновый цвет **background : $background.**

По правилам БЭМ (Блок Элемент Модификатор) т.е. главному блоку присваивается класс. Блоку потомку присваивается класс с названием родительского класса с последующим инкрементированием названия родительского блока через двойное нижнее подчеркивание. Если у блока потомка, имеется, к примеру строчный элемент, которому требуется задать класс то, присваивается класс родителя с добавлением модификатора через тире.

Таким элементом является контейнер рабочего времени **contact\_\_information-worcing-time**. В данный контейнер входит параграф, в котором указывается рабочее время. В параграфе есть тег **span** в котором указаны рабочие дни недели. И так же в блоке данного класса имеется заголовок второго уровня **h2**.

Селектор **contact\_\_information-working-time**. Задаю внешние отступы слева и справа **margin: 0px 10px**. делаю я это для того что бы при адаптивной верстке, когда размер родительского контейнера уменьшается, соседние контейнеры не наплывали друг на друга, а сразу перестраивались в нужном мне порядке.

Селектор **contact\_\_information-working-time p**. Задаю цвет шрифта color: **$contact\_\_information-working-time**. Задаю нижний внешний отступ **margin: 0px 0px 15px 0px**.

Селектор **contact\_\_information-working-time p span**. Размер шрифта **font-size: 1.7rem**. Убираю внешние отступы **margin:0**. Задаю насыщенность шрифта **weight:600**. Цвет **color: $contact\_\_information-working-time**.

Селектор **contact\_\_information-working-time h2**. Цвет шрифта **color:$ contact\_\_information-working-time-h2**. Убираю внешние отступы **margin:0**. Размер **font-size:1.5rem**.

Далее в **contact-information.js** я создаю еще один контейнер div с классом **contact\_\_information-mail.** В данный контейнер я помещаю еще два контейнера: **contact\_\_information-gmail**. Этот контейнер содержит в себе картинку с логотипом и ссылку на почту. И контейнер **contact\_\_information-skype** в котором так же находится логотип и ссылка на контакт skype.

Особенность JSX в том, что некоторые атрибуты и их свойства работают не так как в HTML.Например, если я укажу просто путь к картинке, то картинка не отобразится в пользовательском интерфейсе.

Для того что бы картинка отобразилась в интерфейсе мне необходимо подключить модуль **Node.js require**. Этот модуль похож на директиву **#include в С++**. Данный модуль указывает компилятору на необходимость подключить файл, относительный путь которого указан в круглых скобках.

Селектор **contact\_\_information-mail**. Делаю контейнер **гибким display: flex**. Выстраиваю блоки потомки в колонки **flex-direction: column**. Задаю отступ слева и справа **margin: 0px 10px**.

Селекторы **contact\_\_information-gmail-a, contact\_\_information-skype-a**. Задаю цвет шрифта ссылок **color:$color\_\_link**. Размер шрифта **font-size: 1.6rem**. Насыщенность шрифта **font-weight:600**. Задаю отступ слева что бы картинки логотипов были не так близко к ссылкам **margin-left:10px**.

**contact\_\_information-gmail-a:hover, contact\_\_information-skype-a:hover.** Меняю цвет ссылок **color: $color\_\_link\_\_hover**.

В **contact\_\_information.js** создаю новый контейнер с классом **contact\_\_information-tell** в который помещаю еще два контейнера потомка в которые добавляю логотипы операторов и ссылки контактных телефонов магазина.

Селектор **contact\_\_information-tell**. Делаю контейнер гибким **display: flex**. Контейнеры телефонов выстраиваю в колонки **flex-direction: column**. Задаю отступ слева и справа **margin: 0px 10px**.

Селекторы **contact\_\_information-tell-mts, contact\_\_information-tell-velcome.** Делаю контейнеры гибкими **display: flex**. Выравниваю контейнеры так что бы телефоны и логотипы находились по середине **align-items: center**. Делаю нижний отступ у каждого контейнера **margin-bottom:10px**.

Селектор **contact\_\_information-tell-a**. задаю цвет ссылкам **color:$color\_\_link**. Задаю отступ от логотипов **margin: 0px 0px 0px 10px**. Убираю внутренние отступы у ссылок **padding:0**. Размер шрифта у ссылок **font-size:1.6rem**. Насыщенность шрифта **font-weight:600**.

Селектор **contact\_\_information-tell-a:hover**. Меняю цвет у ссылок **color: $color\_\_link\_\_hover**.

И последним контейнером в **contact-information.js** является форма поиска. В форме поиска два тега: **input** – область ввода символов и кнопка поиска **button** которой присвоен компонент svg картинка Magnifier.

Селектор **form**. Делаю контейнер гибким **display: flex**. Вводимые символы вводятся строкой **flex-direction: row**. Ширина формы **width:300px**. Высота формы **height: 34px**.

Селектор **input**. Ширина контейнера - это вся ширина контейнера form **max-width: 100%.** Задаю цветную рамку **border: 1px solid $input\_\_border\_\_color**. Цвет вводимых символов **color: $input\_\_color**. Размер шрифта **font-size:1.6rem**. Насыщенность шрифта **font-weight:700**.

Селектор **button\_\_magnifier**. Ширина **width:40px**. Высота **height:34px**. Убираю рамку **border:none**. Задаю фоновый цвет **background: $color\_\_button\_\_search-svg-background**. Закругляю верхний правый и нижний правый углы **border-radius: 0px 0.5px 0.5px 0**. При наведении курсора мыши на кнопку стрелочку меняю на руку **cursor: pointer**.

**Адаптивная верстка компонента <ContactInformation/>**

Пишу стили для девайсов максимальная ширина экранов которых составляет 1024px.

Селектор **contact\_\_information**. Делаю контейнер гибки **display: flex**. Выстраиваю контейнеры в несколько строк **flex-wrap: wrap**. Позиционирую контейнер по центру относительно ширины контейнера **justify-content: center**. Задаю фоновый цвет **background: $background**. Внутренние отступы **padding: 30px 7px**.

Так как поисковая форма перестаивается на новую строку я увеличиваю ее ширину **width:580px**. Задаю отступы **margin: 30px 0px 0px 10px**.

Пишу стили для девайсов максимальная ширина которых составляет 600px.

Селектор **contact\_\_information**. Делаю блок гибким **display: flex**. Контейнера в колонки **flex-direction: column**. Позиционирую по центру относительно высоты контейнера **align-items: center**. Фоновый цвет **background:$background**. Внутренние отступы **padding:30px 7px**.

Селектор **contact\_\_information-working-time**. Отступ сверху и слева **margin: 20px 0px 0px 130px**.

Селектор **contact\_\_information-tell**. Верхний и нижний отступы **margin: 20px 0px**.

## Компонент <Header/>

Данный компонент я использую как обертку для компонента Nav. Делаю так, потому что роутинг удобнее прописывать в отдельном компоненте.

И так первым делом в компонент Nav я импортирую необходимые зависимости.

Библиотеку React. Далее импортирую из пакета react – router – dom:

Route – это главный строительный блок React Router – а. В нем, через атрибут path который описывает определенный путь и сопоставляет с именем компонента, вторым атрибутом указываю компонент который нужно отрендерить.

Switch – фильтрует и рендерит тот элемент ссылки на который приходится событие клика.

Далее импортирую Link из пакета react – router – dom. В JSX нет тега a как в html, за ссылки в JSX отвечает тег Link. В его атрибут to я передаю точно такой же путь как в Rout и его соответствующем атрибуте path привязанного к определенному компоненту.

В объекте Link атрибут path сопоставляется с URL путем объекта Route, данные пути должны быть идентичны, после чего происходит переход по ссылке. Объект Switch фильтрует данные запросы, тем самым позволяет выполняться лишь одному запросу на переход по ссылке.

И конечно же импортирую все страницы компонентов, на которые будет реализовываться роутинг данной навигации.

Так как функция Navigation одна в данном файле то, экспортирую ее по умолчанию.

Создаю стрелочную функцию Navigation и ее результатом возвращаю JSX разметку. Под навигацией каждому элементу списка прописываю объект Route в котором создаю адрес каждому компоненту, по которому тот будет доступен. Далее оборачиваю элементы навигации и route в объект Switch.

Селектор **headerNav**. Делаю контейнер гибким **display: flex**. Ссылки в строку **flex-direction: row**. Ссылки равномерно распределяются по всей строке **justify-content: space-around**. Ссылки могут размещаться в несколько **строк flex-wrap: wrap**. Ссылки располагаются на всю ширину экрана **max-width:100%.** Ставлю на фон линейный градиент **background-image: line-gradient(-60deg, line\_\_gradient-one 0%, line\_\_gradient-two 100%)**. Размер шрифта **font-size:2rem**. Убираю внешние отступы **margin:0px**.

Селектор **headerNav li a**. прописываю внутренние отступы **padding: 10px 3px**. Задаю цвет ссылок **color: $header\_\_hav**. Насыщенность шрифта **font-weight:700**. Убираю маркеры элементов списка **list-style: none**. Убираю нижнее подчеркивание у ссылок **text-decoration: none**. Прописываю нижнюю пунктирную рамку **border-bottom: 1px dashed $header\_\_nav\_\_border-bottom**.

Селектор **headerNav li a:hover**. Меняю цвет ссылок **color: $header\_\_nav-hover.**

## Компонент <Main/>

Импортирую библиотеку React, объекты Link и Route из пакета react -router – dom, а также файл стилей и компонент страницы learn – more на которую будет переход по ссылке.

Создаю стрелочную функцию Main и ее результатом возвращаю JSX разметку. Так же экспортирую данную функцию по умолчанию (export default Main;).

В контейнере main создаю контейнер main\_\_wrapper и в него помещаю еще один контейнер main\_\_overlay.

Селектор **main\_\_wrapper**. Трансформирую и позиционирую блок **clip-path:circle(1059px at 190px -263px).** На фон ставлю картинку, через url указываю относительный локальный путь к папке с картинкой, запрещаю картинке повторяться и размещаю ее слева относительно блока **background: url(“../../assets/img/main-img/7.jpg”) no-repeat left**. Заполняю картинкой все свободное пространство родительского блока **background-size: cover**.

Селектор **main\_\_overlay**. Задаю высоту контейнера **height: 794px**. Ставлю на фон линейный градиент **background-image: line-gradient(-60deg, $mail\_\_line-gradient-one 0%, $mail\_\_line-gradient-two 100%).** Задаю прозрачность градиенту **opacity:0.9**.

Далее в main.js создаю родительский контейнер main\_\_offset-row в который помещаю контейнер с контентом main\_\_content, контейнер с картинкой принтера main\_\_offset-printer. В контейнер с картинкой помещаю блок с кнопкой button\_\_catalog-btn.

Рассмотрим стили каждого блока по отдельности.

Селектор **main\_\_offset-row**. Делаю контейнер гибким **display: flex**. Верхний отступ **margin-top: -580px**. Делаю левый отступ **margin-left:30px**.

Селектор **main\_\_content**.Делаю контент выше чем слой с градиентом.

Селектор **main\_\_content-span**. Задаю размер шрифта **font-size:3rem**. Насыщенность шрифта **font-weight: 900**. Цвет шрифта **color: $main\_\_content-span.**

Селектор **main\_\_content-a**. размер шрифта **font-size:3.5rem**. Насыщенность **font-weight:900**. Цвет **color:$main\_\_content-a**. Нижнее подчеркивание у ссылок **text-decoration:underline**. Убираю нижнюю рамку **border-bottom: none**.

Селектор **main\_\_content-a:hover**. Меняю цвет у ссылок **color: $main\_\_content-a-hover.**

Селектор **main\_\_content-p**. цвет шрифта **color:$main\_\_content-p**. Размер шрифта **font-size:3rem**. Насыщенность шрифта **font\_\_weight:900**.

Селектор **main\_\_offset-printer-img**. Ширина **width:400px**. Ставлю принтер выше чем градиент **z-index:9**. Позиционирую вправо **margin-left:210px.**

Селектор **button\_\_catalog-btn**. Цвет шрифта **color: $button\_\_catalog**. Символы в верхнем регистре **text-transform: uppercase**. Верхний отступ **margin-top:119px**. Насыщенность шрифта **font-weight:700**. Используемое семейство шрифтов **font-family: Proxima Nova Rg**. Прописываю **рамку border: 1px solid $button\_\_catalog-btn-border**. Закругляю углы **border-radius:9px**. Прописываю отбрасываемую тень **box – shadow: 0px 3px 3px 0px rgba (126, 81, 185, 0.5).** Внутренний отступ **padding: 14px 33px**. Размер шрифта **font-size: 1.5rem**. Указываю время перехода от одного состояния до другого **transition: all 0.3s ease-out**.

Селектор **button\_\_catalog-btn:hover**. Цвет фона **background-color:$button\_\_catalog-btn-hover**. Меняю цвет шрифта **color: $button\_\_catalog-btn-hover-p**. При наведении на кнопку вместо стрелки появляется рука **cursor: pointer**.

Адаптивная верстка компонента Main.

Девайсы с максимальной шириной 1024px.

Селектор **main\_\_wrapper**. Трансформирую и меняю размер контейнера. **clip-path:circle(729px at 190px -263px).** Картинка растягивается и заполняет все свободное пространство **background-size: cover.**

Селектор **main\_\_offset-row**. Делаю блок гибким **display:flex**. Делаю верхний отступ **margin-top:730px**. И левый отступ **margin-left: 30px**.

Селектор **main\_\_offset-printer-img**. Ширина **width:280px**.

Селектор **main\_\_offset-printer**. Отступ слева **margin-left: 106px**. Делаю размер картинки равным родительскому контейнеру **width:280px**.

Селектор **main\_\_content**. Уменьшаю максимальную ширину контейнера с контентом **main\_\_content: max-width:370px**.

Селектор **button\_\_catalog**. Делаю контейнер гибким **display: flex**. Размещаю кнопку по центру относительно высоты **align-items: center**. Размещаю кнопку по центру относительно ширины **justify-content: center**. Задаю цвет шрифта **color: $button\_\_catalog**. Убираю отступ **margin: 0px**.

Мониторы с максимальной шириной 600px.

Селектор **main**. Убираю левый отступ **margin-left:0px**. Делаю контейнер на всю ширину экрана **max-width:100%**. Высота контейнера **height:1150px**. Внутренний нижний отступ **padding-bottom:95px**. И цвет фона.

Селектор **main\_\_wrapper**. Трансформирую и позиционирую контейнер **clip-path:circle(1229px at 210px -263px).**

Задаю фоновую картинку на всю ширину и высоту блока **background-size:cover.**

Селектор **main\_\_offset-row**. Делаю блок гибким **display: flex**. Содержимое отображается колонками **flex-diraction:column**.Размещаю содержимое по центру относительно высоты **align-items: center**.

Селектор **main\_\_offset-printer**. Ширина картинки **width:480px**. Отступ **margin: 400px 0px 100px 0px.**

## Компонент <OurProduction/>

Импортирую библиотеку React. Далее импортирую компонент our-production-nav, т.к. все основное будет содержаться в этом компоненте. Компонент our-production я использую лишь для отображения заголовка h2.

И так создаю стрелочную функцию OurProduction и в ней возвращаю JSX контейнер с классом our\_\_production в котором содержится заголовок h2. Под заголовком подключаю компонент <OurProductionNav/>.

Стили компонента OurProduction.

Селектор **our\_\_production.** Делаю блок гибким **display: flex**. Блоки потомков делаю колонками **flex-direction: column**. Распределяю блоки равномерно друг от друга **justify-content: space-between**. Выравниваю колонки по середине относительно высоты **align-items: center**. На всю ширину экрана **max-width:100%.**

Селектор **our\_\_production h2**. Подключаю шрифт **Proxima Nova Rg**. Размер шрифта **font-size: 3.4rem**. Насыщенность шрифта **font-weight: 700**. Цвет фона **background: $color\_\_production\_\_h2-background**. Цвет шрифта **color: $our\_\_production\_\_h2-color**. Выравниваю заголовок по центру **text-align: center**. Задаю нижний отступ **padding-bottom:56px**. Убираю внешние отступы **margin:0px**.

Для отображения линий слева и справа заголовка использую псевдо-элементы **before** и **after**.

**Our\_\_production h2::before**. На фон полоски ставлю линейный градиент, делаю элемент **inline – block**, задаю высоту в **height :1.6px**, с помощь **vertical-align** выравниваю по базовой линии контейнера, задаю ширину полоски в **200px**, и даю правый отступ от заголовка в **30px**.

**Our\_\_production h2 ::after**. Ставлю на фон линейный градиент, делаю контейнер **inline – block**, высота линии **1.6px**, выравниваю линию по базовой линии блока, ширина полоски **200px**, и делаю левый отступ от заголовка в **30px**.

Адаптивную верстку пишу лишь для экранов у которых максимальная ширина 800px.

При верстке меняю лишь размер шрифта заголовка на 30px, уменьшаю ширину полосок до 50px и уменьшаю ширину отступов у линий от заголовка до 10px.

**Компонент потомок our-production-nav**.

В данный компонент импортирую библиотеку React, объекты Route, Switch, Link из пакета react – router – dom. Далее, импортирую компоненты страниц, на которые будет осуществляться переход по ссылкам.

Импортирую файл стилей.

Создаю стрелочную функцию OurProductionNav результатом которой возвращаю JSX.

В объект Switch оборачиваю элементы навигации, а так же объекты Route, которые определяют набор маршрутов. В объекте Link атрибут path сопоставляется с URL путем объекта Route, данные пути должны быть идентичны, после чего происходит переход по ссылке. Объект Switch фильтрует данные запросы, тем самым позволяет выполняться лишь одному запросу на переход по ссылке.

Далее в каждый элемент списка li добавляю картинку. В атрибуте src каждой картинки указываю параметр require. Данный параметр указывает компилятору на необходимость подключить файл, относительный путь которого указан в круглых скобках.

И так же в элементы li помещаю заголовки h3 которые делаю ссылками.

**Стили our-production – nav.scss.**

Селектор **our\_\_production\_\_nav**: добавляю фоновый цвет **background: $our\_\_production\_\_nav-background.**

Селектор **.our\_\_production\_\_nav ul.** Делаю блок гибким **display:flex**, выстраиваю блоки в строку **flex-direction: row**, относительно высоты блока размещаю по середине **align-items: center**, размещаю контейнеры потомки в родительском контейнере равномерно удаленно друг от друга **justify-content:space-around**, убираю внешние отступы **margin:0**, внутренний нижний отступ **padding-bottom:170px**.

Селектор **.our\_\_production\_\_nav ul li a h3**. Делаю блок гибким **display:flex**, размещаю заголовки по центру относительно вертикальной направляющей **justify-content: center**, задаю цвет шрифта **color:$our\_\_production\_\_nav-h3**, задаю размер шрифта **font-size:2.5rem**, задаю насыщенность шрифта **font-weight:400**, убираю внешние отступы у блока **margin:0px**.

Селектор **.our\_\_production\_\_nav ul li h3:hover**. С помощью псевдо-элемента **hover** меняю цвет ссылок **color:$our\_\_production\_\_nav-h3-hover**.

Селектор **our\_\_production\_\_nav-img**. Контейнера для размещения картинок делаю гибкими **display:flex**, выравниваю картинку по центру относительно ширины блока **justify-content:center**, Выравниваю картинку по центру относительно высоты блока **align-items:center**, если картинка не помещается в размеры блока, то блок не обрезает картинку до своих размеров а позволяет отображаться даже если размеры картинки больше родительского блока **overflow:hidden**, ширина блока **width: 150px**, высота блока **height: 150px.**

Селектор **our\_\_production\_\_nav ul li img**. Делаю картинки круглыми **border-radius:50%**, задаю цвет фона картинок **background-color:$our\_\_production\_\_nav-img-background**, отбрасываемая тень картинками **box-shadow: 0px 9px 12px 12px rgba(2, 66, 164, 0.2)**, ширина и высота картинок **78px**, указываю время перехода от одного состояния до другого. **transition: all 0.3s ease-out**

Селектор **our\_\_production\_\_nav ul li img:hover**. Изменяю размеры картинок ширина и высота равна 100px.

Адаптивная верстка для экранов максимальная ширина которых составляет 800px. В селекторе **our\_\_production\_\_nav ul** прописываю что блоки могут занимать несколько строк.

## Компонент <DeliveryInfotmation/>

Импортирую зависимости: библиотеку React. Объекты Link, Route и Switch из пакета react – roter – dom.

Импортирую компонент страницы DeliveryInformationLearnMore.

Импортирую svg картинки LogisticDelivery, MapSvg, SertificateSvg, OperatorSvg, которые вынесены п папку svg, которая в свою очередь находиться в папке assets.

Так же импортирую файл стилей delivery-information.scss.

Далее создаю стрелочную функцию DeliveryInformation результатом которой возвращаю JSX разметку. Экспортирую данную функцию по умолчанию export default DeliveryInformation.

В HTML и JSX разметку входит родительский контейнер delivery\_\_information, блока потомка delivery\_\_information-items данный контейнер служит для отступов от родительского блока, далее из двух контейнеров delivery\_\_information-column. В первом из них находятся контейнеры delivery\_\_information-item-1 и delivery\_\_information-item-2.

Во втором delivery\_\_information-item-3 и delivery\_\_information-item-4. В которых в свою очередь находятся контейнеры delivery\_\_information-img в которые я помещаю компоненты svg картинок.

Стили delivery-information.scss.

Селектор **.delivery\_\_information**. Задаю максимальную ширину на весь монитор **max-width:100%.** Ставлю на фон линейный градиент.

Селектор **.delivery\_\_information-column**. Делаю блок гибким **display:flex**, распределяю контейнеры потомки равномерно по контейнеру, задаю левый и правый отступы от родительского контейнера **margin: 0 30px**.

Селектор **.delivery\_\_information-items p strong**. Задаю размер шрифта **font-size: 1.6rem**, цвет шрифта **color: $delivery\_\_strong**.

Селектор **.delivery\_\_information-img**. Делаю контейнер гибкими **display:flex**, размещаю контейнера с фоном картинок посреди родительского контейнера относительно ширины **justify-content: center**, размещаю контейнера по центру относительно высоты **align-items : center**, задаю ширину и высоту контейнера в **50px**, делаю контейнера круглыми **border – radius: 50%**, задаю фон **background – color: $delivery\_\_information-img-background**, делаю внешние отступы у контейнеров **margin: 0 8px;**

Далее задаю общие правила для контейнеров **delivery\_\_information-item-(1,2,3,4).**Делаю контейнера гибкими **display: flex**, размещаю контейнера в строку **flex-direction: row**, содержимое блоков т.е. картинки размещаю по центру относительно высоты **align-items: center*.***

Далее задаю правила для картинок svg.

Селектор **.delivery\_\_information-item-2-svg.** Задаю цвет картинке **fill : $delivery\_\_information-svg-fill,** после задаю ширину и высоту картинке 30px.

Селектор **.delivery\_\_information-item-2-svg:hover*.*** Так же задаю ширину и высоту картинке в 40px, прописываю цвет **fill : $delivery\_\_information-svg-fill:hover.**

Селектор **.map\_\_svg**. Задаю ширину и высоту в 25px, прописываю цвет **fill : $delivery\_\_information-svg-fill***.*

Селектор **.map\_\_svg:hover**. Ширина и высота в 33px, цвет **fill :****$delivery\_\_information-svg-fill:hover.**

Селектор **.sertificate\_\_svg**. Ширина и высота в 25px, цвет **fill :****$delivery\_\_information-svg-fill.**

Селектор **.sertificate\_\_svg:hover**. Ширина и высота 35px, цвет цвет **fill : $delivery\_\_information-svg-fill:hover.**

Селектор **operator\_\_svg.** Ширина и высота 30px, цвет **fill : $delivery\_\_information-svg-fill.**

Селектор **operator\_\_svg:hover**. Ширина и высота 36px, цвет **цвет fill : $delivery\_\_information-svg-fill.**

Между контейнерами delivery\_\_information-column есть контейнер delivery\_\_information-learn-more. В данном контейнере находится контейнер delivery\_\_information-learn-more-inner в котором находится ссылка.

Селектор **.delivery\_\_information-learn-more**. Делаю блок гибким **display: flex**, внутри контейнера размещаю контент по центру контейнера относительно ширины **justify-content: center**, переворачиваю контейнер **transform: rotate(-45deg),**Задаю цвет фона background: **$delivery\_\_information-learn-more-background,** ширина и высота блока 132px,внутренние отступы 5px.

Селектор **.delivery\_\_information-learn-more-inner**. Делаю блок гибким **display:flex**, выравниваю контент по центру относительно ширины **justify-content: center**, выравниваю ссылку по центру **text-align: center,** цвет фона **background: $learn\_\_more\_\_inner-background**, тень от контейнера **box-shadow: 0px 5px 5 px 0px rgba(90, 6, 151, 0.35),** Закругляю углы у контейнера **border-radius: 15px**, ширина и высота 108px, выставляю время перехода от одного состояния в состояние hover **transition:all 0.3s ease-out.**

Селектор **.delivery\_\_information-learn-more-inner-a**. Убираю нижнюю рамку у ссылки **border-bottom:none**, поворачиваю ссылку **transform: rotate(45deg),** задаю размер шрифта **font-size:1.8rem**, задаю насыщенность шрифта **font-weight:700**.

Селектор **.delivery\_\_information-learn-more-inner:hover**. Меняю цвет фона **background-color: $learn\_\_more\_\_inner-background-hover.**

**Адаптивная верстка.**

Мониторы максимальная ширина 1240px.

Селектор **.delivery\_\_information-items**. Делаю контейнер гибким **display: flex**, выравниваю по центру относительно высоты **align- items: center**, выравниваю по середине относительно ширины **justify-content: center**, прописываю внутренние отступы сверху и снизу **padding:30px 0px**.

Селектор **.delivery\_\_information-column**. Делаю контейнер гибким **display:flex**, выстраиваю контейнера потомки в колонки **flex-direction: column**.

Мониторы максимальная ширина 1024px.

Селектор **delivery\_\_information-learn-more**. Увеличиваю внутренний отступ **padding: 10px**.

Мониторы максимальная ширина 800px.

Селектор **.delivery\_\_information-items**. Делаю блок гибким **display:flex**, выравниваю по центру относительно высоты **align-items:center**, Разрешаю контенту располагаться в несколько строк **flex-wrap:wrap**, размещаю контейнеры картинок по левой стороне **justify-content:flex-start**, слева и справа внутренние отступы **padding: 0 30px**;

Селектор **.delivery\_\_information-learn-more**. По сути в данном селекторе остаются такие же правила за исключением отступов. Позиционирую контейнер, отступы слева и снизу **margin: 0px 0px -100px 200px.**

Мониторы с максимальной шириной 600px.

Селектор **.delivery\_\_information-learn-more**. Так же только позиционирую контейнер **margin: -580px 0px 0px 180px**.

Селектор **delivery\_\_information-items**. Позиционирую блок **padding: 150px 0px 40px 0px.**

## Компонент <Partners/>

Импортирую библиотеку React. Импортирую файл стилей partners.scss.

Далее импортирую компоненты svg т.е. логотипы компаний.

После этого создаю стрелочную функцию Partners результатом которой возвращаю JSX. Экспортирую данную функцию по умолчанию export default Partners.

JSX разметка состоит из родительского контейнера partners. В контейнер partners входит два основных контейнера: контейнер partners\_\_logo и контейнер partner\_\_join\_\_us.

Рассмотрим данные контейнеры по подробнее.

Контейнер partners. В данном контейнере содержится контейнер partners\_\_logo-items. Данный контейнер используется как обертка для всех логотипов и используется в основном для позиционирования от родительского контейнера. В контейнере partners\_\_logo-items содержаться десять контейнеров partners\_\_logo-item последние цифры в названии данных контейнеров обозначает порядковый номер контейнера. В данные контейнеры входят компоненты svg логотипов.

Контейнер partner\_\_join\_\_us. Данный контейнер состоит из контейнера partner\_\_join\_\_us-rhombus который я трансформирую в ромб. В данный контейнер входит контейнер partner\_\_join\_\_us-rhombus -background на фон которого ставлю картинку. В данный контейнер входит еще один контейнер с контентом partner\_\_join\_\_us-rhombus -content.

Рассмотрим стили partners.scss.

Селектор **partners**. Делаю блок гибким **display: flex**. Выстраиваю блоки в строку **flex-direction:row**. Равномерно распределяю блоки **justify-content: space-around**. Внутренний нижний отступ **padding-bottom: 60px**. На всю ширину монитора **max-width:100%**. Цвет фона **background: $partners\_\_background.**

Селектор **partners\_\_logo**. Внешние отступы слева и справа **margin: 0px 20px**. Внутренний верхний отступ **padding-top:138px**.

Селектор **partners\_\_logo-items**. Делаю блок гибким **display: flex**. Рарешаю контейнерам размещаться в несколько строк **flex-wrap:wrap**. Равномерно распределяю контейнера **justify-content:space-between**. Задаю максимальную ширину контейнера **max-width: 560px**.

Всем контейнерам **partners\_\_logo-items** задаю нижний **отступ margin: 0px 0px -44px 0px.**

Селектор **partners\_\_join\_\_us**. Делаю блок гибким **display: flex**. Позиционирую блок с ромбом по центру относительно ширины **justify-content:center**. Позиционирую блок с ромбом по центру относительно высоты **align-items:center**. Размещаю слой на нулевом уровне **z-index:0**.

Селектор **partners\_\_join\_\_us-rhombus**. Делаю блок гибким **display:flex**. Размещаю контент внутри ромба по центру относительно ширины **justify\_content:center**. Размещаю контент по центру относительно высоты контейнера **align-items:center**. Внешний верхний отступ **margin-top:138px**. Ширина ромба **width:655px**. Высота ромба **height:500px**. Фон ромба заполняет все свободное пространство **background-size:cover**. Ставлю на фон картинку **background-image: url(../../assets/img/partners/2\_0.png).** Размещаю картинку по центру **background-position: center**. Трансформирую блок в ромб с помощью **clip-path: polygon(50% 0%, 50% 0%, 100% 25%, 100% 25%, 100% 75%, 50% 100%, 0% 75%, 0% 25).**

Селектор **partners\_\_join\_\_us-rhombus-background**. Ширина градиента **width:655px**. Высота градиента **height:500px**. Прозрачность градиента **opacity:0.8**. Задаю градиент **background-image:line-gradient(-60deg, $partners\_\_line-gradient-one 0%, $partners\_\_line-gradient-two 100%).**

Селектор **partners\_\_join\_\_us-rhombus-content**. Делаю блок гибким **display: flex**. Контент размещаю в колонки **flex-direction:column**. Размещаю по центру относительно ширины контейнера **justify-content:center**. Размещаю по центру относительно высоты **align-items:center**. Делаю верхний отступ **margin-top: 60px.** И располагаю слой контента как можно выше **z-index:999**.

Селектор **partners\_\_join\_\_us-rhombus-content h2**. Размер шрифта **font-size:3.4rem**. цвет шрифта **color: $partners\_\_h2**.

Селектор Селектор **partners\_\_join\_\_us-rhombus-content p**. Убираю внешние отступы **margin:0px**. Задаю внутренние отступы **padding: 40px**. Размер шрифта **font-size:2rem**. Цвет шрифта **color: $partners\_\_p**.

Селектор Селектор **partners\_\_join\_\_us-rhombus-content span**. Размер шрифта **font-size:2rem**. Цвет шрифта **color:$partners\_\_span**.

Адаптивная верстка.

Для мониторов с максимальной шириной 1240px.

Селектор **partners\_\_logo**. Задаю левый и правый отступ у блока **margin: 0px 20px**. Задаю верхний внутренний отступ **padding-top: 138px**. Для того что бы svg картинки не расползались задаю минимальную ширину блока **min-width: 360px**.

Для мониторов с максимальной шириной 1024px.

Селектор **partners**. Так же делаю блок гибким **display:flex**. Выстраиваю контейнер с логотипами и контейнер с ромбом в колонки. С помощью reverse меняю порядок колонок **flex-direction: column-reverse**. Выравниваю контент по центру относительно ширины **justify-content: center**. Верхний внутренний отступ **padding-top: 60px**. Максимальная ширина на весь контейнер **max-width:100%**. Выравниваю по центру относительно высоты контейнера **align-items: center.**

Для мониторов с максимальной шириной 600px.

Селектор **partners\_\_join\_\_us-rhombus**. В данном селекторе прописываю все что и ранее за исключением ширины контейнера **width:500px**.

Селектор partners\_\_logo прописываю максимальную ширину экрана **max-width:360px**.

## Компонент <Info/>

Импортирую библиотеку React и файл стилей info.scss.

Создаю стрелочную функцию info в которой возвращаю JSX разметку. Экспортирую данную функцию по умолчанию export default Info.

JSX разметка состоит из родительского контейнера Info и контейнера потомка info\_\_description. В контейнер info\_\_description входит весь остальной контент заголовки, параграфы, элементы списка и т.д.

Рассмотрим стили **info.scss**.

Селектор **info**. Делаю блок гибким **display: flex**. Контент внутри этого блока выравниваю по центру относительно высоты контейнера **align-items: center.** Выравниваю по центру относительно ширины контейнера **justify-content:center**. Выстраиваю содержимое в колонки **flex-direction:column**. Максимальная ширина экрана на всю ширину монитора **max-width:100%.** Делаю отступы сверху и снизу **padding:100px 0px**.

Селектор **info\_\_description**. Задаю максимальную ширину блока, в котором размещаю контент **max-width:1200px**.

Селектор **info\_\_description h2**. Задаю размер шрифта **font-size:3.4rem**. Насыщенность шрифта **font-weight :700**. Цвет шрифта **color: $info\_\_description-h2**. Выравниваю текст по центру **text-align: center**.

Селектор **info\_\_description p**. Размер шрифта **font-size.1.6rem**. Межстрочный интервал **line-height: 1.2**. Цвет шрифта **color: $info\_\_description-p.**

Селектор **info\_\_description ul li**. Выставляю маркеры у элементов списка **list-style:disc**. Размер шрифта **font-size: 1.6rem**. Цвет шрифта **color: $info\_\_description-li**. Убираю внутренние отступы **padding:0px**.

Адаптивная верстка.

Максимальная ширина дисплея 1240px и для дисплеев максимальная ширина которых 1024px.

Селектор **info**. Внутренние отступы **padding:100px 50px**.

## Компонент <Footer/>

Импортирую библиотеку React. Объект Link из пакета react – router – dom.

Импортирую svg Visa, Maestro, MasterCard. Импортирую объект навигации FooterNav и объект Информация FooterInformation. И импортирую файл стилей footer.scss.

Далее создаю стрелочную функцию Footer результатом которой возвращаю JSX разметку.

В JSX разметку данного компонента входит родительский тег footer в который в свою очередь входит четыре родительские секции: footer\_\_info – это контейнер с юридическими реквизитами магазина и контейнером footer\_\_info-carts в котором находятся svg картинки карт оплаты. Секция footer\_\_menu содержит компонент <FooterNav/>. Секция footer\_\_menu-info содержит компонент <FooterInformation/>. И последняя секция данного компонента footer\_\_contact содержит контактную информацию.

Стили **footer.scss**.

Селектор **footer**. Делаю контейнер гибким **display: flex**. Распределяю контейнеры потомки равномерно **justify-content: space-around**. Выравниваю контейнеры по центру относительно высоты родительского контейнера **align-items: center**. Выстраиваю контейнера в строку **flex-direction: row**. Прописываю цвет фона **background: $background\_\_color**. Ширина родительского контейнера равна ширине дисплея **max-width: 100%.** Внутренние отступы **padding: 100px 50px**.

Селектор **footer h4**. Размер заголовков **font-size: 2.4rem**.

Селектор **footer\_\_info**. Делаю отступ слева и справа **margin-top: 0px 10px.**

Селектор **footer\_\_info p**. Размер шрифта **font-size:2.2rem**. Межстрочный интервал **line-height: 1.6.** Цвет шрифта **color: $footer\_\_p**.

Селектор **footer\_\_info-carts**. Делаю блок гибким **display: flex**. Размещаю контейнер по левой стороне **justify-content: flex-start**. Размещаю картинки внутри контейнера по центру относительно высоты **align-items: center**. Размещаю картинки в строку **flex-direction: row**.

Селектор **visa , maestro, mastercard**. Ширина и высота по 80px.

Селектор **footer\_\_menu**. Отступы слева и справа margin: 0 10px.

Селектор **footer\_\_contacts**. Делаю блок гибким **display:flex.** Содержимое контейнера контактов выстраиваю в колонки. И делаю левый и правый отступы **margin: 0px 10px**.

Селектор **footer\_\_contacts a**. Размер шрифта **font-size:2.2rem**. Внутренний отступ **padding:5px**. Цвет шрифта **color:$footer\_\_contacts\_\_a\_\_color**.

Селектор **footer\_\_contacts a:hover**. Прописываю нижнее подчеркивание у ссылок **text-decoration: underline**.

Селектор **footer\_\_contacts p**. Отступы **margin: 10px**. Размер шрифта **font-size: 2.2rem**. Цвет шрифта **color: $footer\_\_p**.

Адаптивная верстка.

Стили для дисплеев 1024px.

Селектор **Footer**. Делаю блок гибким **display:flex**. Выравниваю контент по центру относительно ширины контейнера **justify-content: center**. Выравниваю по центру относительно высоты контейнера **align-items: center**. Все содержимое в строку **flex-direction: row**. Разрешаю контенту выстраиваться в несколько строк **flex-wrap: wrap**. Цвет фона **background: $background\_\_color**. Контейнер на всю ширину дисплея **max-width: 100%.** Внутренние отступы у контейнера **padding: 100px 50px**.

Для селекторов **footer\_\_info, footer\_\_menu, footer\_\_menu-info, footer\_\_contacts** прописываю общие правила. Делаю блоки **гибкими display: flex**. Выстраиваю в колонки **flex-direction: column**. Прижимаю к левому краю **align-items: flex-start**. Ширина блоков **width:300px**. Отступы **margin: 20px 30px.**

## Рендеринг компонентов приложения

Для того что бы компоненты были отрендерены и выведены в браузере мне необходимо импортировать все главные компоненты приложения в компонент App.

После чего я создаю функцию App. В тело функции создаю JSX теги в которых указываю имена функций родительских компонентов. Данные теги - это блоки приложения, которые я могу конфигурировать, расставлять в том соотношении в котором мне угодно.

После итерации и возврата результата, функция App импортируется в index.js в котором происходит рендеринг данного компонента.

Первым параметром функции **ReactDOM.render** передается тот компонент который нужно отрендерит **<App/>,** а вторым параметром я указываю куда нужно поместить отрендеренный компонент **document.getElementById(‘root’).**

После обнаружения данного id в файле index.html, который в свою очередь находится в папке public, происходит отображение всего GUI интерфейса приложения в браузере.

# 

# Заключение

Разработанное GUI приложение реализовано при помощи JavaScript библиотеки React в среде программирования Visual Studio Code.

Данное приложение, «Интерактивного магазина компьютерной техники», может найти свое применение в интернет магазинах компьютерной техники, техники для офисов, а так же их комплектующих. Оно будет полезно как коммерческим организациям, которые пользуются узкоспециализированной техникой и их комплектующими, так и обычному пользователю.

Данное одностраничное приложение имеет перспективы развития так как пока реализованы не все возможности такие как переход по ссылкам. За это отвечает библиотека Redux. Нет покупательской корзины в которой пользователь может совершать покупки.

При написании данного курсового проекта мною были получены практические навыки программирования реального UI приложения с помощью языка программирования JavaScript а так же библиотеки React. Так же были изучены и применены на практике следующие технологии:

* Язык программирования JavaScript а именно ECMAScript ES6.
* JSX – расширение языка JavaScript.
* Препроцессор SСSS – мета язык на основе CSS предназначенный для увеличения уровня абстракции CSS кода и упрощения файлов каскадных таблиц стилей.
* SVG – шрифт используемый для создания двухмерных векторных изображений.
* REM (root em) – масштабируемые единицы шрифта.

# Список используемых источников

1. Бен Фрейн. Разработка сайтов для любых браузеров и устройств - СПБ Питер, 2014г -304 стр.
2. Сэмми Пьюривал. Основы разработки вэб-приложений – СПБ Питер, 2015г – 272стр (Серия «Бестселлеры O`Reilly»).
3. Азат Мардан. React быстро. Веб-приложения на React, JSX, Redux и GraphQL. — СПб.: Питер, 2019. — 560 с.: ил. — (Серия «Библиотека программиста»). ISBN 978-5-4461-0952-4.
4. Томас Марк Тиленс. React в действии. — СПб.: Питер, 2019. — 368 с.: ил. —(Серия «Для профессио­налов»). ISBN 978-5-4461-0999-9.
5. Дженифер Роббинс. HTML5: карманный справочник, 5-е издание. : Пер. с англ. - М. :ООО "И.Д. Вильяме·: 2015. - 192 с.: ил. - Парал. тит. анrл.ISBN 978-5-8459-1937-3 (рус.).
6. Фленаган Д. JavaScript. Подробное руководство, 6\_е издание. – Пер. с англ. – СПб: Символ\_Плюс, 2012. – 1080 с., ил.ISBN 978\_5\_93286\_215\_5.
7. Шмитт К. Симпсон К. HTML5. Рецепты программирования. — СПб.: Питер, 2012. — 288 с.: ил.ISBN 978-5-459-01265-1.
8. Пассинг К. Яндер Й. Программирование без дураков. — СПб.: Питер, 2017. — 416 с.: ил.ISBN 978-5-496-02023-7.
9. Веру Л. Секреты CSS. Идеальные решения ежедневных задач. — СПб.: Питер, 2016. — 336 с.: ил. — (Серия «Бестселлеры O’Reilly»). ISBN 978-5-496-02082-4.
10. Б**.** Хоган**,** К**.** Уоррен**,** М**.** Уэбер**,** К**.** Джонсон**,** А**.** Годин
11. Книга веб-программиста: секреты профессиональной разработки веб-сайтов. — СПб.: Питер, 2013. — 288 с.: ил.
12. Юрий Б. React + Redux – Профессиональная разработка. Практическая разработка коммерческих React приложений - 2019г [Электронный ресурс] – Режим доступа: <https://www.udemy.com/course/pro-react-redux/>
13. Владилен М. React JS с Нуля до профессионала – 2019г [Электронный ресурс] – Режим доступа: <https://www.udemy.com/course/react-2020-complete-guide/>
14. Александр К. Разработка интернет магазина на React / Redux – 2019г [Электронный ресурс] – Режим доступа: <https://www.udemy.com/course/ecommerce-shop-react-redux/>

# Содержание электронного носителя