

Оглавление

Введение.	8
ГЛАВА 1. ОБЗОР СОВРЕМЕННЫХ ТЕХНОЛОГИЙ СОЗДАНИЯ КЛИЕНСКОЙ ЧАСТИ ВЕБ ПРИЛОЖЕНИЯ.	14
1.1 Язык гипертекстовой разметки, таблицы стилей и препроцессоры.	14
1.2 JavaScript как язык разработки современных веб приложений.	18
1.3 Расширение возможностей JavaScript с использованием JS-библиотек	24
1.4 Расширение возможностей JavaScript с использованием JS-фреймворков. .	27
Выводы по главе.	31
ГЛАВА 2. КОМПОНЕНТНЫЙ ПОДХОД ПРОГРАММИРОВАНИЯ ВО FRONT-END.	32
2.1 Обзор компонентного подхода программирования.	32
2.2 Разработка одностраничных приложений.	34
2.3 Компонентная разработка с использованием react.js.	37
2.4 Оптимизация процесса работы с данными, flux архитектура.	40
Выводы по главе.	43
ГЛАВА 3. РАЗРАБОТКА FRONT-END ПРИЛОЖЕНИЯ УПРАВЛЕНИЯ АВТОМАТИЗИРОВАННЫМ ТЕСТИРОВАНИЕМ ПРОЕКТОВ.	44
3.1 Постановка задачи.	44
3.2 Выбор UX фреймворка для react.js приложения.	45
3.3 Обзор структуры разработанного приложения и компонент.	46
3.4 Обзор разработанного программного продукта.	47
Выводы по главе.	51
Заключение.	52
СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ.	54
СПИСОК ТЕРМИНОВ.	55
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.	57
Приложение А. Исходный код некоторых компонент.	66

Введение

На протяжении существования человечества успех той или иной цивилизации в большой степени определялся возможностями сохранять, воспроизводить и передавать накопленные знания. От шумерских табличек, египетских иероглифов или славянской бересты человечеству потребовалось огромное количество времени, чтобы перейти к книгопечатанию, однако переход от книжного формата к электронному произошёл буквально за одно столетие, хотя сам процесс все ещё продолжается.

Однако без совершенствования технологий в области программирования развитие в данном направлении было бы невозможным. Именно процесс совершенствования технологий и подходов проектирования и разработки привело к созданию первой сети обмена информацией между ведущими американскими университетами. Именно этот момент можно назвать отправной точкой развития веб-технологий, а рост вычислительных возможностей и скоростей передачи информации, а так же удешевление производства обусловили высокие темпы развития. Так, если ещё в начале 2000-х среднестатистический горожанин вполне обходился без использования интернета, то в настоящих реалиях абсолютное большинство так или иначе использует веб-технологии в повседневной жизни.

Первой проблемой которая стала перед разработчиками стал поиск приемлемых средств хранения и обработки информации. Выходом стали языки гипертекстовой разметки. Действительно для отображения информации на экране монитора нужно не так уж много. Однако на каком-то этапе расширения всемирной паутины, стало понятно, что отображать информацию более не достаточно. Появилась необходимость в добавлении интерактивности, а так же средств более эффективной стилизации текста. А снижение стоимости персональных компьютеров и увеличение их производительных мощностей позволили переложить большую часть логики на сторону клиента.

Анализ современных тенденций.

Итак не смотря на тот факт, что основополагающими технологиями в области *front-end* разработки по прежнему являются *HTML*, *CSS* и *JavaScript*, данные технологии сами по себе не позволяют обеспечивать все возрастающие запросы рынка. Дабы справляться с необходимыми темпами и уровнем качества разработки, программисту нужны более эффективные средства разработки. Что приводит к созданию не только новых технологий, но и совершенствованию подходов к работе с кодом и тестированием.

В плане организации работы современные реалии заключается в необходимости разработки приложений в рамках команд разбросанных по всему миру. Такая ситуация приводит к необходимости создания приложений с максимально простым для понимания синтаксисом, а также требует совершенствования инструментов разработчика.

И так современные реалии приводят к необходимости организации всего приложения или больших частей в виде универсальных модулей и компонент с максимально понятным синтаксисом. И дальнейшее их использование в большом количестве проектов.

Актуальность темы.

Рассматривая сферы использования веб-приложений, трудно найти какую-либо сферу деятельности абсолютно не использующую веб-технологии. В настоящее время данные технологии широко используются в следующих областях:

1. Электронный бизнес. По результатам исследований, большинство интернет ресурсов так или иначе связаны с коммерческой деятельностью. Интернет широко используется для рекламы и непосредственной продажи товаров и услуг, для маркетинговых исследований, электронных платежей и управления банковскими счетами.

2. Средства массовой информации. По перечню информации интернет-издания не чем не уступают традиционным, а во многих и превосходят – есть новостные сайты, литературные, научно-популярные, детские, женские и т. п. На данный мо-

мент на интернет просторах можно найти практически любую интересующую информацию, за исключением узкоспециализированной. Ещё одним, в равной степени достоинством и недостатком, является динамичность. Так если офлайновые издания выпускаются периодически (раз в день, неделю, месяц), то интернет-издания обновляются по мере появления нового материала.

3. Литература, музыка, кино. Электронные библиотеки, доступные через интернет позволяют получить доступ к большому числу изданий, в том числе редких и труднодоступных в привычной форме.

4. Связь. Электронная почта в настоящее время является одним из наиболее используемых средств связи. В последние годы популярность обрели мессенджеры, передающие сообщения через Интернет, они стали вытеснять из повседневной жизни людей сотовую связь, которая в сравнении с ними чаще всего уступает в функциональности, скорости, а также стоимости.

5. Общение. Интернет является способом массового общения людей, объединённых различными интересами. Для этого используются интернет-форумы, блоги и социальные сети.

Список областей в которых веб-технологии стали привычными или могут найти применение можно расширять до бесконечности. Именно широкое использование обуславливает совершенствование подходов к разработке и появление новых технологий.

Таким образом, актуальность исследования обусловлена широким использованием веб-технологий в различных сферах деятельности, а также высокими темпами развития и совершенствования технологий *front-end* разработки.

За время обучения и при непосредственном участии научного руководителя были определены объект, предмет, цель и задачи исследования.

Цель: проанализировать возможности оптимизации разработки веб-приложений с использованием современных веб-технологий, а также определить эффективность компонентного подхода во *front-end*.

Постановка задачи: для достижения поставленной цели необходимо решить следующие задачи:

1. Провести анализ современных технологий разработки клиентской части веб технологий;
2. Провести анализ расширения возможностей создания динамического содержимого с использованием JS-библиотек и JS-фреймворков;
3. Провести обзор компонентного подхода к созданию современного веб-приложения;
4. Провести обзор компонентной разработки с использованием библиотеки react.js;
5. Провести проверку эффективности использования компонентно-ориентированного подхода, путем создания клиентской части веб-приложения.

Объект исследования: процесс *front-end* разработки веб-приложений и сайтов различной сложности с использованием современных технологий и тенденций программирования (*JS*-фреймворков, и библиотек).

Предмет исследования: подходы и методы оптимизации *front-end* разработки сайтов и веб-приложений, реализованные в конкретной технологии, а также использование ими компонентно-ориентированного подхода.

Теоретической основой явились научные исследования российских (В. Кулямин, И. С. Блинов, В. С. Романчик) и зарубежных (*H. Schildt*, Дж. Рихтер, *A. Goldstein*, *L. Lazaris*, *E. Weyl*, Д. Сидерхолм, Д. Макфарланд, *C. Lindley*, *M. Haverbeke*, *Addy Osmani*, *N. Zakas*, *P. Koch*, Крейн Д., Паскарелло Э., Джеймс Д., *M. Kiessling*, *T. Holowaychuk*, *A. Young*, *R. Jansen*, *R. Murphey*, Б. Бибо, И. Кац, Р. Бенедетти, Р. Крэнли, *B. Bibeault*, *A. Rosa*, *J. Garcia*, *S. Stefano*, *A. Banks*, *E. Porcello* и др.) ученых.

Методологическая основа исследования. В исследовании широко использовались методы сравнения, обобщения, классификации.

Информационная база исследования. В качестве информационной базы исследования использованы инструкции разработчиков, а так же труды отечественных и зарубежных разработчиков.

Научная значимость заключается в обосновании использования компонентного подхода организации программного продукта для эффективной разработки, а

так же создания приложений имеющих достаточно эластичную доля дальнейшего использования структуры.

Теоретическая и прикладная значимость.

Теоретическая значимость заключается в том, что выделены тенденции развития и оптимизации разработки веб-приложений и интерфейсов, а также компонентно-ориентированного подхода; выявленные перспективы использования и развития компонентно-ориентированного подхода в *front-end* разработке.

Практическая значимость заключается в том, что было разработано приложение с использованием современных тенденций компонентно-ориентированного подхода к разработке приложений и библиотеки *react*.

Апробация результатов исследования.

Результаты исследования рассмотрены на следующих конференциях:

1. Доклад на тему: «Создание интернет магазина с использованием CSS- и JS-фреймворков» XXXVII научная конференция студентов I, II ступеней и аспирантов
2. Доклад на тему: «Компонентный подход front-end разработки веб-приложений» в рамках XLIV научной и учебно-методической конференции Университета ИТМО

По результатам исследования опубликованы следующие статьи:

1. Кузменков А. Причины возникновения и развития JS-фреймворков, преимущества, достоинства и недостатки / А. Кузменков // Научный обозреватель. – 2017 – №. 6. – С. 49 – 52.
2. Кузменков А. Компонентный подход разработки приложений во *front-end* / А. Кузменков // Научный обозреватель. – 2018 – №. 5.

Объем и структура работы. Работа состоит из пяти основных разделов: введение, три главы и заключение.

Во введение приведено обоснование актуальности темы, сформированы цель и задачи исследования, а так же определена практическая и теоретическая основа исследования.

В первой главе приведен обзор основополагающих технологий в разработке клиентской части веб-приложения, а так же рассмотрены возможности оптимизации разработки с использованием фреймворков и библиотек.

Вторая глава ориентирована на рассмотрении компонентного подхода к созданию программного обеспечения при создании клиентской части веб приложения, а так же специфика оптимизации обмена данными с сервером.

В третьей главе приведен обзор программного продукта разработанного с применением компонентного подхода программирования. Разработка приложения позволила на практике проверить эффективность метода.

В заключение приводятся выводы по проделанной работе.

ГЛАВА 1. ОБЗОР СОВРЕМЕННЫХ ТЕХНОЛОГИЙ РАЗРАБОТКИ КЛИЕНТСКОЙ ЧАСТИ ВЕБ ПРИЛОЖЕНИЙ

1.1 Язык гипертекстовой разметки, таблицы стилей и препроцессоры

При поверхностном рассмотрении любой веб страницы на просторах современного интернета, можно заключить вывод что любая клиентская часть приложений состоит из трёх основополагающих элементов:

1. *HTML*-документ.
2. Каскадные таблицы стилей.
3. Скрипты динамического поведения.

По существу справедливым будет утверждение о том, что любое современное веб приложение состоит из этих трех элементов. Однако это не совсем так.

С постоянным увеличением количества ресурсов предоставляемых по средством сети интернет, неумолимо вставал вопрос о совершенствовании имеющихся механизмов и технологий. Однако прежде чем переходить к более современным средствам разработки следует разобраться что же из себя представляет основа любого приложения, а именно *html* (*Hyper Text Markup Language*) [1].

Итак, в том или ином виде основой любого веб приложения является *HTML*-документ. Именно на его основе браузер строит *DOM*-модель документа прежде чем отобразить результат на пользовательском мониторе. Однако данная технология является не столько языком, сколько стандартом, необходимым для стандартизации, разрабатываемых продуктов. Как и любой иной стандарт *HTML* пережил большое число изменений, связанных с появлением новых технологий, подходов и методик разработки. Так, если первоначально для создания документа для веб было достаточно одного языка гипертекстовой разметки, современное приложение уже

строится на трех основополагающих элементах, взаимодействие которых и создает клиентское приложение в современном виде. Так, появление технологии каскадных таблиц стилей и постепенное расширение возможностей *CSS*, привело к устареванию большого числа первоначально используемых тегов таких как *center*, *font* и другие. На момент написания этой работы повсеместно используемым стандартом являлся *HTML5*.

Основным нововведением пятого стандарта семантические теги, теги управления аудио и видео содержимым, а также графический контейнер. Итак попробуем кратко разобраться что же привнесли данные элементы в процесс разработки.

Итак, благодаря большому числу семантических тегов, современное веб приложение можно разделить на логические единицы, которые явно указывают о своем назначении [2]. Такое нововведение в большой степени связано развитием систем поиска и анализа данных. Так благодаря правильной комбинации и использованию семантических элементов можно оптимизировать работу поисковых систем, обернув значимую и второстепенную информацию в соответствующие теги. Так же правильное использование нововведений открывает большие возможности для устройств, предназначенных для людей с ограниченными возможностями. Так устройство авто чтения статьи может определить нужный текст и опустить рекламу.

Появление тегов управления медиа содержимым, вероятно связано с расширением доли медиа контента. Появление данных механизмов сделало морально устаревшими более ранние технологии (*Flesh*), что уменьшило количество стороннего программного обеспечения.

Расширение возможностей графических библиотек (таких как *WebGl*), и их использования вероятно послужило необходимостью для введения специализированного тега.

Также на момент написания работы активно развивался шестой стандарт. Ключевой особенностью данного языка является введение пространств имен, а так же предполагает возможность создания одностраничных приложений без использования скриптов [3, 4].

Как было сказано выше многие теги *html* утратили свою необходимость в связи с развитием *CSS*. Данная технология представляет собою язык описания внешнего вида документа, написанного с помощью языка разметки [5]. Язык определяет внешний вид документа (шрифты, цвет текста и фона, отступы), а также позволяет создавать разметку [6, 7].

Уровень 1 (*CSS1*) [8]. Спецификация принята в 1997 году и определяет параметры шрифтов (размер, стиль, гарнитуру), цвета, атрибуты текста, выравнивания, свойства блоков (высота, внутренние и внешние отступы, рамки).

Уровень 2 (*CSS2*) [9]. Спецификация принята в 1998 году и расширила возможности первого уровня. Были добавлены блочная вёрстка, определение типов носителей, звуковые таблицы стилей, расширен механизм селекторов и прочее.

Уровень 2.1 (*CSS2.1*) [5]. Спецификация принята в 2011 году и в основном исправляла ошибки прошлой.

Уровень 3 (*CSS3*) [10]. Данная спецификация расширяет возможности предыдущих, а также добавляет возможности добавления анимации без использования *JavaScript*. Особенностью данной версии является модульная структура, то есть работа ведется по нескольким не связанным направлениям.

Так же активно ведется разработка нового четвертого уровня [11].

Однако использование *CSS* не позволяет использовать при написании таблиц стилей даже базовых возможностей программирования, таких как циклы и функции, механизмы наследования. Для увеличения уровня абстракции и расширения возможностей используются препроцессоры, такие как *SASS* [12] и *LESS* [13]. Главным преимуществом данных технологий является возможность повторного использования кода за счет модульной структуры и примесей, однако при этом перед использованием требуется компиляция кода.

Ещё один недостаток данного подхода заключается в сложной системе блочной-верстки. Из-за сложности системы для решения не тривиальных задач необходим более высокий уровень знаний, а также внимательность разработчика.

С целью совершенствования и упрощения блочной верстки предложены две технологии *CSS Flexbox* [14] и *Grid Layout* [15]. На момент написания данной работы данные технологии широко внедрялись, однако так как ограничения поддержки вынуждали разработчиков использовать две версии стилей.

Основным преимуществом использования технологии *Flexbox* является широкие возможности выравнивания элементов относительно осей координат. Можно легко выравнивать элементы по горизонтали и по вертикали, менять направление и порядок отображение элементов, растягивать блоки на всю высоту родителя или прибивать их к нижнему краю [14]. На Рисунке 1.1 показана ориентация элементов. На данный момент технология поддерживается всеми топовыми версиями браузеров.

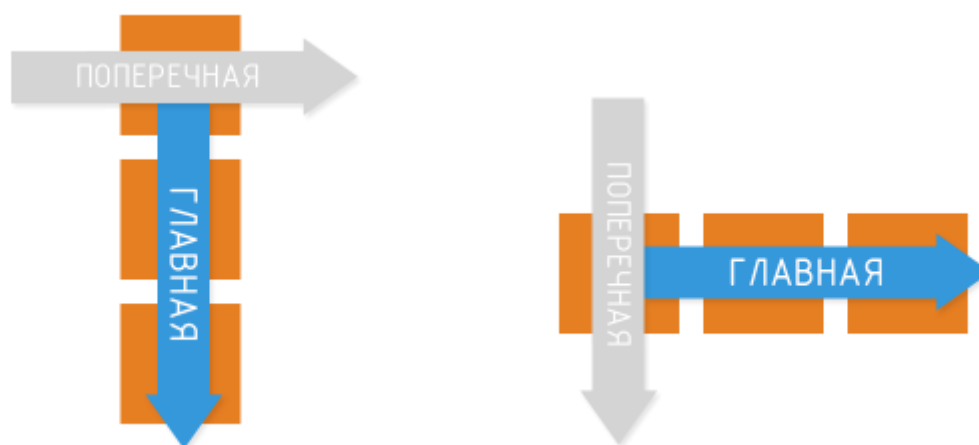


Рисунок 1.1 – Ориентация элементов *flexbox*

Следует отметить что *JS*-реализация *flexbox* нашла свою нишу использования при разработке мобильных приложений. Данная технология широко используется в совокупности с *React Native*.

Grid представляет документ в виде сетки, благодаря чему можно легко манипулировать основными частями приложения. Основным преимуществом данной технологии является отсутствие зависимости отображения элементов вне за-

висимости от их порядка, что позволяет более эффективно разрабатывать интерфейсы для разных типов устройств. На данный момент ещё имеет недостаточную поддержку даже новыми браузерами.

В таблице 1.1 приведены данные по поддержке выше упомянутых технологий. В данной таблице приведены минимальные версии браузеров в которых реализована поддержка технологий [16, 17].

Таблица 1.1 – Поддержка *CSS Flexbox* и *Grid Layout*

Браузер	<i>CSS Flexbox</i>	<i>Grid Layout</i>
<i>Internet Explorer</i>	11 (Не полностью)	11 (Не полностью)
<i>Edge</i>	16	16
<i>Firefox</i>	58	58
<i>Chrome</i>	49	63
<i>Safari</i>	11	11
<i>iOS Safari</i>	10.3	10.3
<i>Opera Mini</i>	Все	–
<i>Chrome for Android</i>	64	64
<i>UC Browser for Android</i>	11.8	11.8
<i>Samsung Internet</i>	4	6.2

1.2 JavaScript как язык разработки современных веб приложений

Не смотря на совершенствование *HTML* и *CSS* данные технологии не позволяют вести полноценную *front-end* разработку, а сфера ответственности ограничивается внешним видом приложения. Поэтому ключевой частью веб-приложения по праву можно назвать *JS*-скрипты. Именно они добавляют интерактивность приложению. И позволяют переносить бизнес логику на сторону клиента.

Итак, что же представляет из себя *JavaScript*. Данный язык программирования представляет собой мультипарадигменный язык программирования [18], направленный на создание клиентской части веб-приложения с использованием наиболее эффективных методик программирования нашего времени. В настоящее время *JS* позволяет использовать все преимущества объектно-ориентированного, императивного и функционального стилей программирования [18], комбинируя подходы для достижения более высоких результатов.

Основным преимуществом сочетания различных стилей программирования, а так же большого числа языковых конструкций является облегченный процесс адаптации разработчиков, при переходе на новую технологию. Так же такое сочетание позволяет вести более эффективную разработку. Это связано с тем фактом что трудоемкость решения различных задач с использованием разных подходов может иметь существенные отличия.

Не смотря на то, что *JavaScript* первоначально являлся функциональным языком программирования, на данный момент имеются широкие возможности в создании объектно-ориентированных клиентских приложений. Однако в связи с большим разнообразием объектная модель имеет целый ряд особенностей по сравнению с другими языками программирования. Так же следует отметить что данный язык активно развивается, а также имеет огромное сообщество разработчиков. Можно сказать, что данный язык постоянно совершенствуется и впитывает наиболее удачные идеи [19 – 21].

Первоначально синтаксис языка схож с синтаксисом С. Однако, с течением времени язык развивается и появляются определенные синтаксические особенности [16]: объекты, функции как объекты первого класса, автоматическое приведение типов, автоматическая сборка мусора, анонимные функции и прочее.

В области *front-end* разработки структуру языка можно представить в виде трех частей [19, 22]:

1. Ядро (*ECMAScript*).
2. Объектная модель браузера (*Browser Object Model* или *BOM*).
3. Объектная модель документа (*Document Object Model* или *DOM*).

Ядро представляет собой основу для построения скриптового языка (*ECMAScript*). Описывает типы данных, инструкции, ключевые слова, операторы, объекты, регулярные выражения, не ограничивая возможности расширения функционала. На момент написания данной работы, активно используемой версией являлась *ES6* или *ES2015*. В таблице 1.2 приведены некоторые данные по развитию стандарта [23].

Таблица 1.2 – Выход версий стандарта *ECMOScript*.

Название	Дата выхода	Редакторы
<i>ES1</i>	Июнь 1997	<i>Guy L. Steele Jr</i>
<i>ES2</i>	Июнь 1998	<i>Mike Cowlishaw</i>
<i>ES3</i>	Декабрь 1999	<i>Mike Cowlishaw</i>
<i>ES4</i>	–	–
<i>ES5</i>	Декабрь 2009	<i>Pratap Lakshman, Allen Wirfs-Brock</i>
<i>ES5.1</i>	Июнь 2011	<i>Pratap Lakshman, Allen Wirfs-Brock</i>
<i>ES6</i>	Июнь 2015	<i>Allen Wirfs-Brock</i>
<i>ES7</i>	Июнь 2016	<i>Brian Terlson</i>
<i>ES8</i>	Июнь 2017	–

Как видно из таблицы темпы развития стандарта только растут. Каждая новая версия приносит что то новое. Так:

ES3 не только улучшил работу но и добавил обработку регулярных выражений и исключительных ситуаций.

ES5 предоставляет строгий режим разработки, призванный улучшить процесс разработки и поиск ошибок связанных с синтаксисом, а так же увеличивает функциональные возможности (к примеру обработка *JSON*) [24].

ES6 добавляет большое количество новых синтаксических конструкций характерных для современных объектно-ориентированных приложений. Существенно измен синтаксис объявления и работы с классами, а так же добавлены коллекции, стрелочные функции. Так же расширены возможности работы с массивами и объектами в функциональном стиле [25].

ES7 продолжает реформу кода в направлении создания изолированных модулей и библиотек.

ES8 предлагает функционал по расширению возможностей работы с потоками, а так же предполагает добавление реализации работы с многопоточностью [26].

Возвращаясь к браузерному *JS* следует разобраться что из себя представляют *DOM* и *BOM*. Итак объектная модель браузера представляет собой прослойку между ядром и объектной моделью документа [27]. При этом основная задача заключается в управлении окнами браузера и обеспечении их взаимодействия. Каждое из окон браузера представляется объектом *window*, центральным объектом *DOM* [28, 29]. *BOM* же обычно обеспечивает возможность управления сущностями браузера такими как [27]: управление фреймами, поддержка задержки в исполнении кода и заикливания с задержкой, системные диалоги, управление адресом открытой страницы, управление информацией о браузере, управление информацией о параметрах монитора, ограниченное управление историей просмотра страниц, поддержка работы с куками.

В настоящее время *JavaScript* широко применяется при реализации:

1. Веб-приложений (клиентской части) – клиент серверные приложения с распределенной бизнес логикой между клиентской частью (браузером) и сервером [30].

2. *AJAX*-запросы – подход организации взаимодействия между сервером и клиентом заключающийся в асинхронном обмене данными, при этом пользователь, при первом обращении, получает клиентское приложение, которое взаимодействует с сервером путем обмена данными в формате *JSON* [31, 32].

3. *COMET* – является широким понятием, описывающим механизмы работы веб приложений на основе постоянных соединений, аналогичных обычным настольным приложениям [33]. Наиболее известной реализацией можно назвать *web-sockets*.

4. Браузерные операционные системы – исходный код некоторых операционных систем более чем на половину состоит из *JS* [18] (*iCloud*, *Glide OS*, *eyeOS*, *myGoya Online Desktop*, *Online Operating System* и др.).

5. Бурмарклеты – используется для создания небольших программ, размещаемых в закладки браузера [18].

6. Пользовательские скрипты в браузере – небольшие пользовательские скрипты, обеспечивающие создание определенной динамики на стороне клиента. Например: валидация в формах, слайтеры, блоки рекламы и прочее [18].

7. Серверные приложения – развитие и популярность языка привели к появлению серверных реализаций, позволяющих вести полноценную разработку серверной части приложения на чистом *JavaScript*. Так же *JS* может исполняться на серверах, использующих *Java* с 6 версии [34]. Наиболее популярные платформы: *Jaxer*, *persevere-framework*, *Helma*, *v8cgi*, *node.js* [35 – 37].

Не смотря на разнообразие синтаксических конструкций, возможности и внутренняя работа по большому счету определяется конкретной реализацией (*JS*-движком). В таблице 1.3 приведен обзор основных *JS*-движков [38 – 45].

Таблица 1.3 – Обзор основных *JS*-движков.

Название	Описание
<i>Rhino</i>	<i>JavaScript</i> -движок с открытым исходным кодом, реализованный на <i>Java</i> и управляемый с помощью <i>Mozilla Foundation</i> в качестве программного обеспечения. Данный движок используется для встраивания <i>JS</i> в <i>Java</i> -приложения.
<i>SpiderMonkey</i>	первый <i>JavaScript</i> -движок с открытым исходным кодом, разработанный Бренданом Эихом для <i>Netscape Communications</i> . В настоящее время движок поддерживается <i>Mozilla Foundation</i> и используется в качестве движка для <i>Mozilla Firefox</i> и его расширений.
V8	<i>JavaScript</i> -движок с открытым исходным кодом, разработанный Ларисом Бак для <i>The Chromium Project</i> . Создателем проекта является Ларс Бак. Данный движок используется в браузерах <i>Google Chrome</i> , а так же имеет

Продолжение таблицы 1.3.

Название	Описание
	серверные реализации для <i>MongoDB</i> , <i>NodeJS</i> . Особенностью данного движка является компиляция в машинный код с дополнительной оптимизацией вместо традиционной интерпретации.
<i>JavaScriptCore</i>	<i>JavaScript</i> -движок для <i>WebKit</i> . Первая реализация движка получена из библиотеки <i>KDE JavaScript engine</i> и библиотеки регулярных выражений. В дальнейшем улучшался с добавлением нового функционала и оптимизацией.
<i>KJS</i>	<i>JavaScript</i> -движок первоначально разрабатываемый для веб-браузера <i>Konqueror</i> . В дальнейшем послужил основой для <i>JavaScriptCore</i> .
<i>Chakra (Jscript9)</i>	Движок разработанный <i>Microsoft</i> для 32-разрядной версии браузера <i>Internet Explorer 9</i> . Первоначально был разработан как программный продукт с закрытым исходным кодом, однако в январе 2016 выпущен с открытым исходным кодом.
<i>Chakra (JavaScript)</i>	Движок разработанный <i>Microsoft</i> для браузера <i>Microsoft Edge</i> . Данный движок основан на движке <i>JScript</i> , использовавшемся в семействе браузеров <i>Internet Explorer</i> . Код основных компонентов движка является открытым как <i>ChakraCore</i> .
<i>Nashorn</i>	Движок с открытым кодом от <i>Oracle</i> , написанный на <i>Java</i> и основанный на машине <i>DaVinci</i> . Распространяется с виртуальной машиной <i>Java</i> и предоставляет возможность использовать <i>JS</i> при разработке на <i>Java</i> . Код машины является открытым и доступен в репозитории <i>OpenJDK</i> .

В выше упомянутой таблице приведены наиболее известные и распространенные *JS*-движки. Так же существует большое число менее известных или закрытых проектов таких как *Juce*, *JerryScript*, *Jsish*, *Charakan* и др. [46].

Однако не смотря на свою универсальность и динамику развития *JavaScript* имеет ряд недостатков. В первую очередь отсутствует строгая типизация, что приводит к неявным ошибкам из-за недочетов программиста. Во вторых новые синтаксические возможности стандарта в различных движках добавляются не равномерно и недостаточно быстро. Так же отсутствует обратная совместимость что особенно остро ощущается при работе с линейкой *Internet Explorer*.

Для решения вышесказанных проблем используется предкомпиляция скриптов в определенную версию стандарта. На момент написания работы наибольшую популярность приобрел *Babel* и его расширения. Данная группа библиотек позволяет разрабатывать приложение с использованием современных синтаксических возможностей и получать на выходе скрипты с желаемым синтаксисом. Так же предкомпиляция позволяет оптимизировать использование сторонних библиотек, взяв в конечный вариант только используемые части.

Так же на процессе предкомпиляции *JS* построены некоторые языки расширяющие возможности стандарта. На момент написания работы наибольшую популярность среди разработчиков приобрел *TypeScript* [47, 48]. Данная технология вводит типизацию, позволяет определить ошибки на этапе компилирования, а также добавляет многие инструменты характерные для объектно-ориентированных технологий (перечисления, коллекции и т. д.).

1.3 Расширение возможностей *JavaScript* с использованием *JS*-библиотек

Не смотря на все преимущества использования *JavaScript*, разработка на чистом *JS* занимает много времени. Для решения многих типовых задач существует большое число расширений (библиотек).

Итак, разберемся что же из себя представляет *JS* библиотека. *JS* библиотека это набор функций для решения типовых задач. Наверно каждому программисту известна стандартная библиотека *Math* присутствующая в любом языке программирования и содержащая основные математические функции.

Возвращаясь в контекст языка *JS* одной из наиболее известных и некогда революционных библиотек *jQuery*. Распространение данной библиотеки обусловлено

в первую очередь историей, так как данная библиотека является одним из первых удачных решений по расширению возможностей *JS* и устранению их недостатков. Во многом её популярность обусловлена несогласованностью, и конфронтацией разработчиков браузеров. Главным образом это приводило к большим проблемам при разработке кроссбраузерных приложений, так как один и тот же скрипт мог не работать без значительных изменений в другом браузере. Взяв на себя решение проблем кроссбраузерности, данная библиотека обеспечила себе место под солнцем. В таблице 1.4 приведена информация по некоторым версиям библиотеки [49].

Таблица 1.4 – История релизов *jQuery*

Версия	Дата релиза	Последнее обновление	Размер (KB)
1.0	26 августа 2006	1.12.4 (20 мая 2016)	95
2.0	18 апреля 2013	2.2.4 (20 мая 2016)	85.6
3.0	9 июня 2016	3.3.1 (20 января 2018)	84.8

Однако в современных реалиях данная библиотека является устаревшей. При этом она все ещё имеет широкое распространение. Во многом это связано с тем фактом, что многие технологии разработки сайтов построены на этой библиотеке. Не секрет что для создания типовых сайтов (блоги, интернет магазины, новостные порталы) используются различные *CMS*, при этом некоторые по умолчанию используют *jQuery* (*Drupal*), а некоторые позволяют установить эту библиотеку в качестве расширения (*Joomla 3*). Наличие большого числа сторонних библиотек позволяют в короткие сроки создать полнофункциональный сайт средней сложности [50].

В своей работе библиотека фокусируется на работе с *DOM*-моделью, то есть обеспечивает взаимодействие *JavaScript* и *HTML*. Также библиотека предоставляет удобный *API* для работы с *AJAX* запросами [51]. Однако новые функциональные возможности *JavaScript* сводят к нулю все преимущества данной библиотеки.

Тем не менее библиотека предоставляет следующие возможности [52 - 54]: движок кросс-браузерных *CSS*-селекторов *Sizzle*, выделившийся в отдельный про-

ект, переход по дереву *DOM*, включая поддержку *XPath* как плагина, события, визуальные эффекты, *AJAX*-дополнения, *JavaScript*-плагины. Модульная структура позволяет в значительной степени расширять базовый функционал за счёт подключения компонент модулей. Библиотека имеет как собственную библиотеку компонент [55] так и сторонние компоненты.

Так же на основе *jQuery* формировались новые технологии в том числе и полноценные *ui*-фреймворки. Наиболее выдающимся является *Bootstrap*. Данная технология представляет собою набор инструментов для создания сайтов и веб приложений. Включает в себя *HTML*- и *CSS*-шаблоны оформления для типографики, веб-форм, кнопок, меток, блоков навигации и прочих компонентов веб-интерфейса, включая *JavaScript*-расширения [56]. Начало фреймворка положено 19 августа 2011 года [57], когда свет увидела первая версия *Bootstrap*. На момент написания работы актуальной версией фреймворка являлась *Bootstrap4*, работа над которой началась 29 октября 2014 года. Альфа версия вышла 19 августа 2015 года [58].

К недостаткам можно отнести бедную цветовую гамму стандартного набора иконок. К преимуществам – хорошую реализацию *grid*-сетки для масштабирования веб-страницы, создания адаптивного дизайна. Основные инструменты *Bootstrap* [59]: сетки, шаблоны, типографика, медиа, таблицы, формы, навигация, алерты и др.

Кроме того существует большое число более узко специализированных библиотек, предназначенных для создания универсального инструмента, решающего одну конкретную задачу. В качестве примера можно привести библиотеку *moment.js* [60]. Данная технология предоставляет инструменты форматирования и обработки дат и времени для всевозможных вариантов локализации.

В заключении стоит отметить что не смотря на преимущества предоставляемые *JS*-библиотеками, они все же не предоставляют полноценных средств разработки, а использование универсальных средств решения узко направленных задач приводит к росту объёма конечных скриптов за счет неиспользуемых частей. Возвращаясь к библиотеке *moment.js* можно отметить что большая часть приложений ограничится двумя локализациями, национальной и международной (английской),

тем не менее код библиотеки будет включен полностью в конечный вариант приложения.

1.4 Расширение возможностей *JavaScript* с использованием *JS*-фреймворков

Следующей ступенью разработки клиентской части веб приложений стали *JS*-фреймворки. Итак, что же из себя представляет фреймворк? В отличии от библиотек фреймворки предлагают оболочку, позволяющую вести разработку полнофункциональных *front-end* приложений, используя только средства самого фреймворка. При этом оптимизировались процессы разработки и сам код приложений. Тем не менее, не смотря на широкие возможности, решения одних и тех же задач для разных фреймворков могут иметь разную степень сложности и эффективности. Так же следует отметить что сам по себе фреймворк обостряет проблему избыточности кода. Далее приведен обзор некоторых фреймворков по отдельности.

Ext JS представляет собой *JS*-фреймворк для построения интерактивных веб-приложений с использованием *AJAX*, *DHTML* и *DOM*. Библиотека включает в себя широкий набор элементов управления для использования в веб-приложениях: текстовые поля, дата и время, числовые поля, списки, элементы выбора, редактор *HTML*, деревья и прочее. Многие из этих элементов позволяют легко настроить взаимодействие с сервером через *AJAX* запросы, что позволяет значительно сократить объем передаваемых по сети данных.

В таблице 1.5 приведен краткий обзор версий фреймворка с описанием нововведений, а так же хронологией выхода[61, 63].

Таблица 1.5 – Обзор версий фреймворка *Ext.js*

Версия	Дата выхода	Нововведение
2.0	Декабрь 2007	Включенные в фреймворк компоненты стилизованы для разработки веб-приложений максимально приближенных к настольным.
3.0	Июль 2009	Расширены возможности фреймворка в области обеспечения коммуникации пользователей, добавлены новые элементы управления флэш-графикой и списками.
4.0	Апрель 2011	Переработана структура классов и пакетов данных, а так же добавлены новые архитектурные возможности для поддержки <i>MVC (Model View Controller)</i> .
5.0	Июнь 2014	Расширены возможности работы с графикой, большими объемами данных, добавлены новые компоненты ориентированные для работы с сенсорными экранами.
6.0	Март 2016	В данную версию добавлен полноценный фреймворк для работы с мобильными приложениями.

Исходя из выше приведенной таблицы к достоинствам работы с данным фреймворком можно отнести [64]:

1. Большой набор виджетов позволяет в короткие темпы разработать высоко функциональный пользовательский интерфейс;
2. Приближенность пользовательского интерфейса к оконному приложению;
3. Оптимизация интерфейса для мобильных устройств;
4. Организация эффективной работы с большими объемами данных;
5. Наличие проработанной документации в том числе и на русском языке.

Недостатки библиотеки заключаются в самой специфике работы фреймворка. Так если нет необходимости выходить за рамки доступного функционала разработку можно вести достаточно быстро и эффективно, то расширение функционала являть очень трудоемким. Ещё одним не маловажным недостатком являются лицензионные ограничения использования разрабатываемых продуктов.

DojoJS представляет собой *JS*-фреймворк, ориентированный на многие потребности крупномасштабной клиентской веб-разработки [65]. Данный фреймворк обеспечивает кросбраузерность, содержит большое число компонентов, предоставляет инструменты оптимизации разработки документации. Так же фреймворк разжился большим числом внешних расширений от сторонних производителей. В организации работы с модулями фреймворк ориентируется на паттерн проектирования *MVVM* (*Model-View-ViewModel*) (Рисунок 1.2) [66].

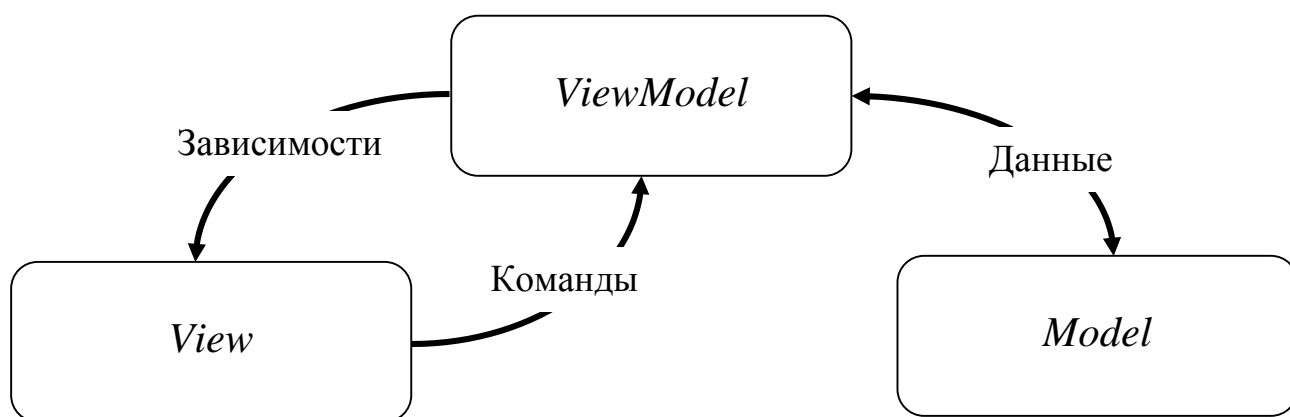


Рисунок 1.2 – *Model-View-ViewModel*

Так же следует отметить что данный фреймворк является продуктом с открытым исходным кодом и имеет большое число дополнений. При этом сам фреймворк состоит из четырех основным частей [65]:

dojo – содержит ядро фреймворка и дополнительные модули.

dijit – библиотека стабильных модулей для создания пользовательского интерфейса.

dojox – содержит модули, которые не обеспечивают достаточно стабильную работу и использование которых не гарантирует стабильную работу программы.

util – дополнительные инструменты для оптимизации, построения документации и другое.

Итак к достоинствам данного фреймворка можно отнести большое количество стандартных и сторонних компонент, а так же поддержку *MVVM* “из коробки”.

При этом данный фреймворк довольно сложный в работе что требует дополнительных знаний для разработчика, а так же имеет достаточно плохую документацию что обостряет проблему. Так же избыточность фреймворка выше чем у прочих.

Angular. При упоминании об данном фреймворке следует произвести разделение на *Angular v1.0* и *Angular v2.0*. По существу данные версии различаются настолько, что могут считаться разными фреймворками, а знание одного не делает следующим в другом.

Итак *Angular v1.0* построен на разделении пользовательского интерфейса и бизнес логики, а так же использовании декларативного и императивного подхода [67]. Фреймворк расширяет и адаптирует традиционный *HTML* для представления динамического контента. Данный фреймворк среди разработчиков обозначается как *AngularJS*. Это связано с использованием *JS*, как рабочего языка. Так же данный фреймворк использует в своей основе *MVC* в качестве архитектуры (Рисунок 1.3)

Последующие версии фреймворка реализованы с использованием *TypeScript*. Также произошел отказ от использования контролеров. Вместо этого используется иерархия компонент (компонентный подход) в качестве основы архитектуры [68]. Так же существуют огромные различия в синтаксисе, присутствует строгая типизация, модульность и др..

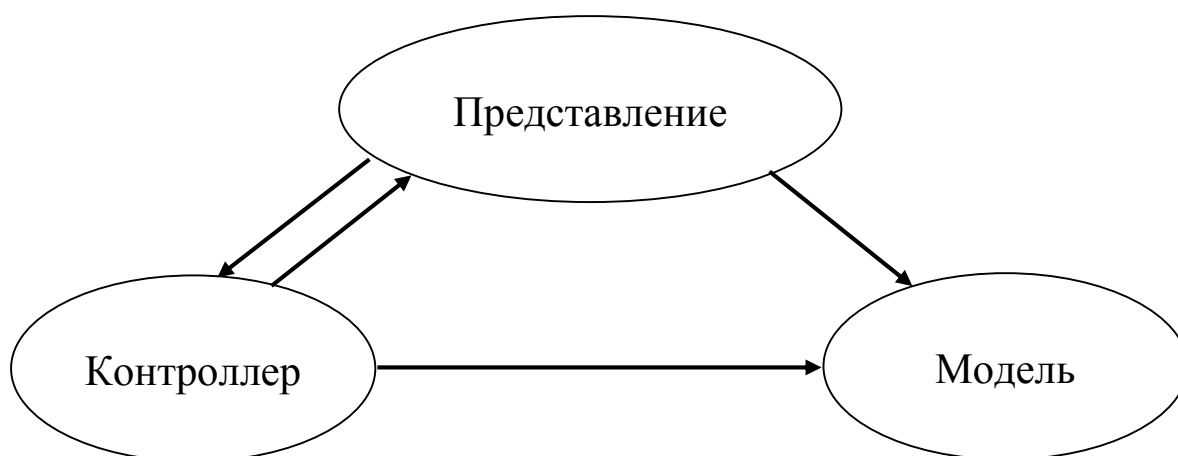


Рисунок 1.3 – *Model-View-Controller*

К достоинствам данного фреймворка можно отнести наличие большого сообщества разработчиков, декларативный подход к разработке, оптимизация времени разработчика за счет использования модульной структуры и готовых решений.

Однако данный фреймворк является трудным в освоении, требует большого уровня знаний для создания по настоящему эффективных приложений, так же эффективность работы приложения сильно зависит от его размеров.

Выводы по главе

В данной главе приведен обзор основополагающих технологий *front-end* разработки, а также возможности их использования в соответствии с современными реалиями и расширениями основного набора технологий. Так же уделено отдельное внимание использованию различных библиотек и фреймворков не только для улучшения производительности самих приложений, но и для оптимизации процесса разработки. Так же рассмотрены достоинства и недостатки использования некоторых технологий.

В заключении можно отметить что за годы развития и использования сети интернет, значительно усовершенствованы подходы к разработке, усовершенствованы основополагающие технологии, а так же появляются и совершенствуются средства оптимизирующие процесс разработки и работы клиентской части приложения.

ГЛАВА 2. КОМПОНЕНТНЫЙ ПОДХОД ПРОГРАММИРОВАНИЯ ВО FRONT-END

2.1 Обзор компонентного подхода программирования

Прежде всего необходимо определить, что подразумевается под компонентным подходом программирования. Итак компонентный подход представляет собой появившуюся в 1987 году [69] парадигму компонентно-ориентированного программирования основанная на понятии компонента.

Компонент [70] – независимый модуль программы, предназначенный для повторного использования и развертывания, реализованный в виде множества языковых конструкций, объединённых по схожим признакам.

Началом компонентно-ориентированного подхода принято считать предложенный в 1987 году Никлаусом Виртом паттерна написания блоков для языка Оберон. Через два года Бертон Мейер предложил идею взаимодействия, вызывающего и вызываемого компонентов. Идея воплотилась в решениях *CORBA*, *COM*, *SOAP*, а позднее в компонентном паскале.

На данный момент компонентно-ориентированный подход в том или ином виде может использоваться в различных языках программирования и технологиях.

Существуют языки программирования, реализующие компонентно-ориентированное программирование на конструктивном уровне: *Oberon*, *Component Pascal*, *Active Oberon*.

В рамках платформы *Java 2 Enterprise Edition* компонентный подход реализуется в четырех видах [71, 72]:

Enterprise JavaBeans (EJB). Данные компоненты предназначены для организации бизнес логики приложения и работы с данными. Отличительной чертой данных компонентов является работа в компонентной среде (*EJB-контейнер*).

Веб-компоненты (*Web components*). Данные компоненты необходимы для предоставления интерфейсов работающих поверх протоколов сети Интернет. К компонентам данной категории относятся фильтры, обработчики веб-событий, сервлеты и серверные страницы (*JavaServer Pages, JSP*).

Обычные приложения. Могут также использовать веб-компоненты, так как J2EE является расширением J2SE.

Апплеты. Представляют собою небольшие компоненты с графическим интерфейсом и предназначенные для работы внутри браузера.

В рамках платформы *.NET* [73] компонентно-ориентированный подход реализуется в виде *COM(Component Object Model)*. *COM* [74] представляет собою собой стандарт, предназначенный для создания программного обеспечения на основе взаимодействующих компонентов, каждый из которых может использоваться во многих программах одновременно. На данном стандарте основаны многие технологии такие как [75]: *DCOM, COM+, OPC, OLE, DComLab*.

Так же большое распространение компонентный подход обрёл в сфере *front-end* разработки. Вместе с появлением новых технологий разработки веб интерфейсов появлялись и библиотеки готовых компонент, позволяющие создавать приложения как набор взаимодействующих элементов (Рисунок 2.1).

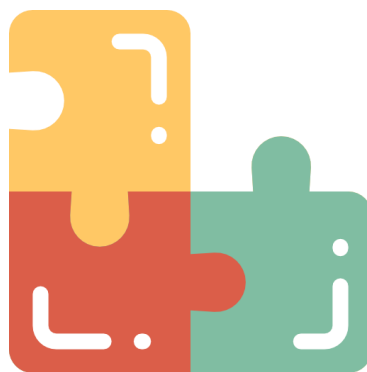


Рисунок 2.1 – Набор взаимодействующих элементов

При этом использование компонентно-ориентированного программирования позволяет создавать универсальные строительные элементы современного сайта.

Основным преимуществом подхода является легкость модернизации приложения путем замены одного компонента другим. Такой эффект достигается благодаря использованию черного ящика (Рисунок 2.2), то есть элемент программы проектируется и реализуется таким образом, что бы иметь определенный типовой интерфейс при этом совершенно не важна внутренняя реализация.

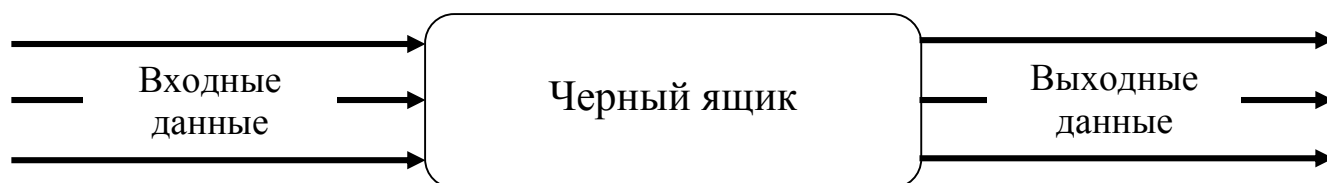


Рисунок 2.2 – Схема работы организации работы программного компонента «Черный ящик»

Для веб технологий так же характерно разделение компонентов на логические и стилизующие, что в значительной степени облегчает дальнейшую работу с кодом программы, так как позволяет беспрепятственно подменять один компонент другим.

2.2 Разработка одностраничных приложений

Как уже говорилось выше в последнее время наблюдается тенденция переключения части бизнес логики на сторону клиента. Во многом это связано с ростом производительности персональных компьютеров и нагрузки на серверную часть приложения. Таким образом все имеющиеся на просторах интернета веб приложения можно разделить на два типа: многостраничные и одностраничные приложения.

При этом многостраничное приложение представляет собой классическое веб-приложение при котором клиент запрашивает страницу. На стороне сервера формируется веб-документ и возвращается клиенту. При этом процессе тоже существует определенная оптимизация в виде кэширования скриптов, что предотвращает их повторную загрузку.

Одностраничное же приложение представляет собой, базовую станицу и набор скриптов динамически формирующих содержимое документа в зависимости от действий пользователя. При этом приложение состоит из *HTML*-шаблона, таблиц стилей, *JS*-скриптов и прочего содержимого (изображений и др.). При этом взаимодействие с сервером осуществляется по средствам *AJAX* запросов. В результате которых приходят чистые данные, а клиент обрабатывает данные, определяя отображение. При этом серверную часть принято называть *API*. Схема работы веб приложения приведена на Рисунке 2.3.

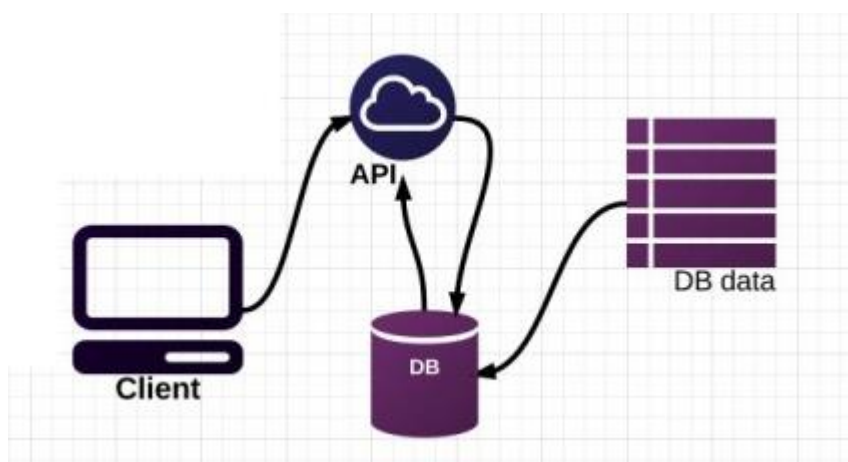


Рисунок 2.3 – Схема работы одностраничного приложения

Существует несколько подходов к разработке API сервера. Один из наиболее распространенных является Rest API [76]. Ключевой особенностью является зависимость обработки запроса в зависимости от типа и переданных данных.

GET: smth/ – возвращает совокупность данных какого либо типа. Обычно данные возвращаются в сокращенном виде и используются для организации навигации по приложению.

GET: smth/id – возвращает конкретную запись, при этом данные максимально полные по содержанию.

POST: smth/ – используется для сохранения данных.

PUT: smth/id – используется для изменения уже существующих данных.

DELETE: smth/id – используется для удаления.

В зависимости от конкретной реализации запросы *PUT* и *DELETE* могут быть приведены к *POST*. В таких случаях тип действия определяется переданными данными.

Данный подход весьма эффективен при разработке небольших приложений, так как разделяет клиентскую и серверную часть. При этом *API* может использоваться для нескольких приложений в неизменном виде. Однако при создании более сложных систем, как, к примеру социальная сеть встает вопрос с ростом количества обработчиков при получении различных вариантов одной и той же информации.

Для решения данной проблемы предложена концепция *GraphQL* [77]. По сути при использовании данного подхода вместо большого числа специализированных роутов имеется один умный, который определяет, что необходимо вернуть в зависимости от переданных данных.

Так при использовании *REST API* для получения одного типа данных и нескольких вариаций с зависимостями, необходимо либо делать для каждой вариации отдельный роут, либо делать несколько запросов. Однако, не смотря на тот факт, что *GraphQL* справляется с данной проблемой (Рисунок 2.4), использование на малых проектах является не целесообразным. Это связано с тем, что каждый уровень абстракции делает приложение более сложным, а так же требует ресурсов на дополнительную обработку.

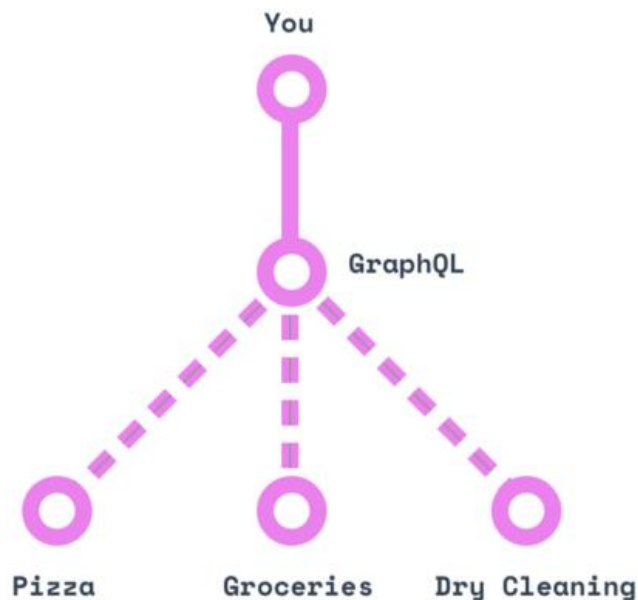


Рисунок 2.4 – Схема работы *GraphQL*

2.3 Компонентная разработка с использованием *react.js*

Возвращаясь к клиентской части веб приложения можно отметить что наиболее эффективная организация работы с любой *API* достигается благодаря компонентному подходу.

Одной из реализаций компонентного подхода во *front-end* разработке приложений является разработанная *Facebook* библиотека *react.js*. Тем не менее на момент своего появления в 2013 году предложенная концепция была принята весьма скептически так как имела уникальный подход [78, 79] и отбрасывала некоторые общепринятые концепции. По существу разработчики откинули опыт сообщества и создали библиотеку, позволяющую получить максимум от использования модульной структуры приложения.

Основным преимуществом данной библиотеки являлось возможность создания веб-интерфейсов быстро, с использованием читаемого и понятного кода, что во многом обеспечило большой приток свежей крови. А так же серьёзного расширения возможностей стандартной библиотеки путём использования сторонних библиотек, построенных на *react.js*. Если осуществить поиск на *github* или с помощью *npm*, можно найти большое количество решений типовых задач.

Следующей ключевой особенностью языка является возможность использования языковых конструкций при динамическом создании *html*-узлов в приближенном к *HTML* виде с использованием так называемого *JSX* (*JavaScript Syntax eXtension*), расширения стандартного *JS* для динамического создания элементов. Что значительно сокращает и упрощает понимание синтаксиса компонента, увеличивая тем самым эффективность создания и поддержки кода.

Следующим преимуществом является высокая скорость перестроения *DOM*, благодаря внутренней оптимизации. Это достигается путем создания так называемой виртуальной *DOM*-модели, которая накапливает изменения и, в дальнейшем, вносит сразу все накопленные правки. Как известно работа с *DOM* является одной из наиболее затратных процедур, и оптимизация позволяет в значительной степени ускорить работу *front-end* [78, 79]. При этом перестройка происходит при изменении данных *state* (внутреннее состояние компонента) и *props* (внешнее состояние). Вовремя своей работы компоненты проходят через ряд этапов жизненного цикла. На каждом этапе вызывается определенная функция, переопределив которую можно задать собственное поведение. При этом функционал можно разделить на три основных части: создание, изменение и удаление.

Далее приведено описание функций жизненного цикла [80]:

constructor(props) – конструктор, в котором происходит начальная инициализация компонента, при этом *props* внешние параметры компонента.

componentWillMount() – вызывается непосредственно перед рендерингом компонента.

render() – рендеринг компонента.

componentDidMount() – вызывается после рендеринга компонента. Здесь можно выполнять запросы к удаленным ресурсам.

componentWillUnmount() – вызывается перед удалением компонента из *DOM*.

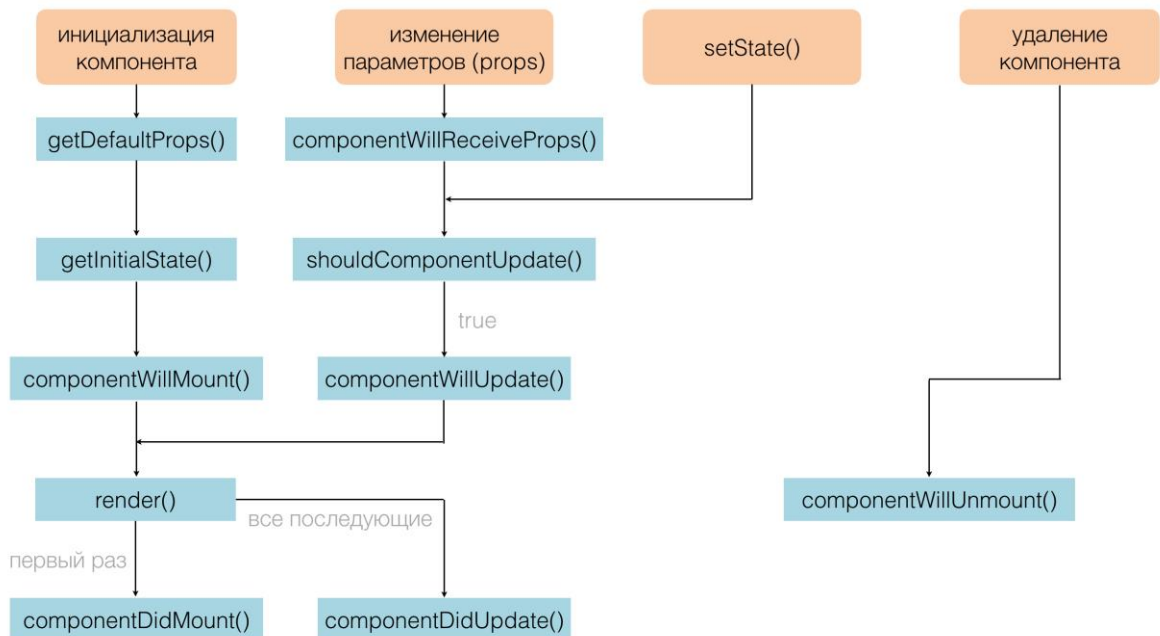


Рисунок 2.5 – Жизненный цикл *React* компонента

shouldComponentUpdate(nextProps, nextState) – вызывается каждый раз при обновлении объекта. В качестве параметров передаются новый *props* и *state*. В зависимости от возвращаемого функцией значения можно управлять рендерингом (*true* если компонент нужно обновить). Таким образом можно запретить рендеринг компонента если заведомо ничего не изменилось.

componentWillUpdate(nextProps, nextState) – вызывается перед обновлением компонента.

componentDidUpdate(prevProps, prevState) – вызывается сразу после обновления компонента.

componentWillReceiveProps(nextProps) – вызывается при обновлении объекта *props*. Новые значения этого объекта передаются в функции в качестве параметра.

Как правило, в этой функции устанавливаются те свойства компонента, в том числе из *this.state*, которые зависят от значений из *props*.

Большое количество разработанных сторонних компонентов позволяет создавать высоко функциональные интерфейсы при минимальных затратах времени и сил. Ниже приведено описание некоторых основных компонентов:

React Router [81] – предоставляет функционал для роутинга приложения. Роутер позволяет определить содержание одностраничного приложения в зависимости от заданного адреса.

React Motion [82] – библиотека добавляет анимацию в реакт. Производит интерполяцию значений, используемых в *CSS* для трансформации элементов документа.

React Addons [83] – представляет собой пакет опционных функций, которые можно использовать для улучшения производительности приложений реакт.

Enzyme [84] – предоставляет функционал для написания автоматизированных тестов для *react* компонент.

Redux [85] – библиотека реализующая *flux*-паттерн для работы с данными. Так же существует большой набор компонентов доступных для скачивания и использования в системе *npm*.

2.4 Оптимизация процесса работы с данными, *flux* архитектура

При разработке веб приложений с большой долей обработки бизнес логики на стороне клиента одним из основных вопросов встающих перед разработчиком становится организация обмена данными с сервером (*API*) и изменения состояния компонентов. Особенно остро становится вопрос в случае одновременного использования данных в нескольких компонентах, при этом изменение данных

внутри одного компонента должно повлечь за собой изменение состояния всех компонентов, использующих данный набор данных. Наиболее простым способом решения данной проблемы является организовать работу через общего предка Рисунок 2.6.

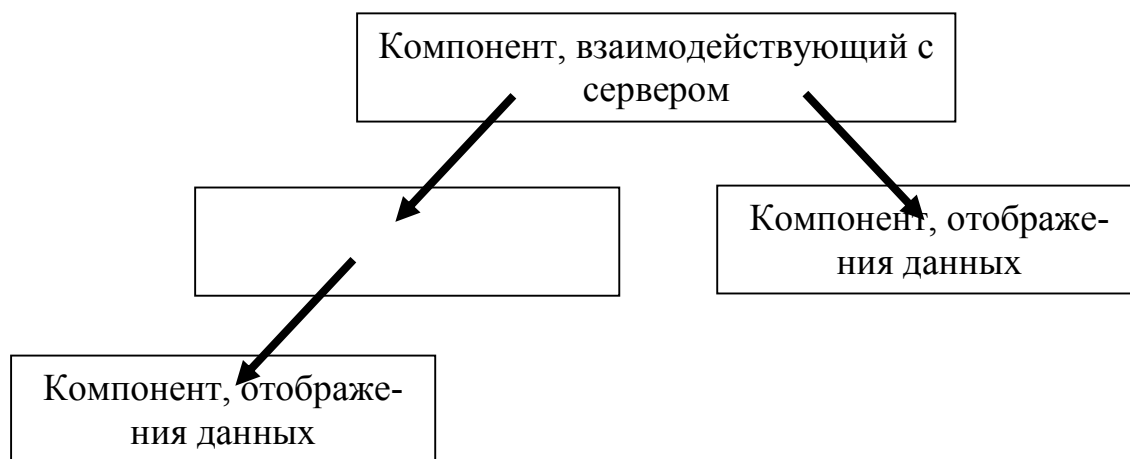


Рисунок 2.6 – Схема работы через общего предка

Однако такой подход имеет существенный недостаток. При наличие большого числа связанных компонентов разбросанных в разных частях дерева компонент, встает необходимость пробрасывать функции обработчики и данные через большое число родительских компонентов. Такой процесс значительно ухудшает не только эффективность работы приложения, но так же влечет к неумолимому ухудшению качества кода.

Таким образом становится актуальным создание сущности, являющейся общим, для всех связанных компонент, хранилищем данных, а так же организовать систему обработки действий (*actions*) таким образом, что бы при вызове действия из одного компонента, произошло обновление состояния всех связанных компонент.

Для решения данной задачи команда *Facebook* предложила свой архитектурный подход (*Flux*-архитектура). Данная архитектура предлагает реализовать следующий набор взаимосвязанных сущностей [86, 87]:

Controller Views или представления. Представляют собой компоненты собирающие состояния хранилищ и передающих их дочерним компонентам.

Stores или хранилища. Представляют собой контейнеры хранящие состояния, и бизнес логику для компонент.

Dispatcher или диспетчер. Принимает действия и рассылает нагрузку зарегистрированным обработчикам.

Actions или действия. Упрощают передачу данных диспетчеру.

На Рисунке 2.7 процесс взаимодействия сущностей представлен в виде диаграммы.

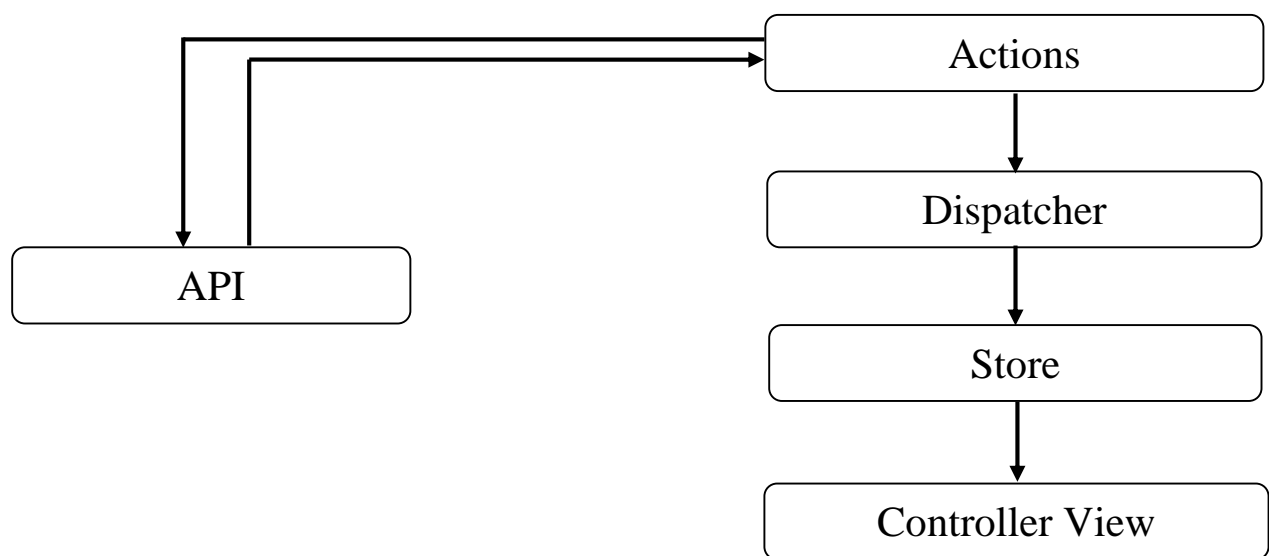


Рисунок 2.7 – Диаграмма работы *Flux* архитектуры

Таки образом когда в компоненте происходит действие, происходит обращение к *API*. После получения ответа от сервера данные предаются в диспетчер, который вызывает изменение общего для группы компонент хранилища, что в свою очередь влечет за собою изменение состояния всех компонент использующих данное хранилище.

На данный момент широко используются две реализации *Flux*-архитектуры: одноименная библиотека *Flux* и *Redux*. Отличие данных реализаций заключается в организации хранилища. Так если *flux* подразумевает существование нескольких

отдельных хранилищ в рамках приложения, работающих с разными группами компонент, то *redux* предлагает работать с общим для всего приложения хранилищем. Так же библиотека *redux* вводит понятие *reducer*, таким образом разделяя бизнес логику и хранение данных.

В Таблице 2.1 представлены некоторые данные по использованию данных технологий в реальных проектах [88, 89].

Таблица 2.1 – Статистические данные по использованию *Flux* архитектуры

Название технологии	Flux	Redux
Наблюдателей	703	1435
Звезд	14987	39380
Копий	3631	9394

Выводы по главе

В данной главе сфокусировано внимание на компонентно-ориентированном подходе в разработке *front-end* приложений. Проанализированы преимущества использования, сферы применения.

В заключении можно отметить что данный подход позволяет вести разработку понятных, легко модернизируемых приложений, а так же открывает большие перспективы в оптимизации работы современных веб-приложений.

ГЛАВА 3. РАЗРАБОТКА FRONT-END ПРИЛОЖЕНИЯ УПРАВЛЕНИЯ АВТОМАТИЗИРОВАННЫМ ТЕСТИРОВАНИЕМ ПРОЕКТОВ

3.1 Постановка задачи

В первую очередь необходимо определить, поставленную для разработки задачу и перечень используемых технологий. Итак, задача заключается в практической проверке эффективности процесса разработки с использованием компонентно-ориентированного подхода, путём создания *front-end* части приложения управления автоматизированным тестированием проектов.

Для решения поставленной задачи были выбраны следующие технологии:

1. Для динамической части приложения выбрана библиотека *react.js*, позволяющая использовать в полной мере компонентно-ориентированный подход при разработке клиентской части веб-приложений, вести разработку используя наиболее прозрачные языковые конструкции присущее современным стандартам *ECMAScript*.

2. Для разработки стилей выбран *CSS* препроцессор *LESS*, позволяющий использовать возможности программирования для разработки таблиц стилей, что существенно увеличивает возможности повторного использования стилевых конструкций.

3. Для оптимизации размера передаваемых данных необходимо вести разработку одностраничного приложения. Так как приложение должно эффективно переключаться между различными типами используемой информации, необходимо. Для этого был выбран модуль, построенный на библиотеке *react.js*, *react-router*.

4. Для оптимизации работы, и обмена данными с серверной частью выбрана реализация *flux* паттерна проектирования *redux*.

Таким образом выбранные технологии максимально соответствуют специфике выбранного направления разработки, а так же обеспечивают все необходимый уровень быстродействия при обработке бизнес логики на стороне клиента.

3.2 Выбор *UX* фреймворка для *react.js* приложения

В связи с наличием большого числа технологий, разрабатываемых как крупными компаниями (*Facebook*, *Google*), так и отдельными группами энтузиастов, существует возможность существенно ускорить разработку приложения. Одной из таких возможностей является использование *UX* фреймворков.

Прежде всего стоит разобраться что же из себя представляет *UX* фреймворк. Прежде всего это совокупность подходов и инструментов разработки пользовательских современных кроссплатформенных приложений. Далее приведен краткий анализ возможностей основных, на момент написания работы *UX* фреймворков:

1. *Material-UI* [90]: Данный фреймворк содержит широкий набор компонент и инструментов для разработки пользовательских интерфейсов с использованием материал дизайн. Преимущества использования данной технологии является: широкий набор готовых к использованию компонент; гибкая система настройки стилей, позволяющая стилизовать приложения путём модифицирования одного файла настройки стилей; наличие инструментов создания кроссплатформенных приложений. Ключевым недостатком данной технологии является отсутствие должного обеспечения обратной совместимости стабильных версий. Так приложение, при необходимости перехода на новую версию фреймворка должно быть практически полностью переписано.

2. *React Desktop* [91]: Данный фреймворк переназначен для разработки интерфейсов для настольных приложений с использованием технологии *Electron*. Содержит кроссплатформенные компоненты для операционных систем *Windows 10* и *Mac OS*.

3. *Semantic-UI-react* [92], *Bootstrap-react* [93]: Данные технологии являются клонами одноименных фреймворков, содержат набор инструментов разработки идентичный родительскому, однако используют внутри себя устаревшую библиотеку *jQuery*, что является ключевым недостатком данных технологий.

Так же альтернативу при рассмотрении могут представлять и другие менее известные фреймворки (*Ant-design*, *Blueprint*, *React-Toolbox* и др.).

Не смотря на широкие возможности предоставляемые вышеупомянутыми UX фреймворков, преимущества не перевешивают недостатки при разработке простых пользовательских интерфейсов. В связи с этим можно сделать вывод что использование UX фреймворка для конкретной задачи является нецелесообразным.

3.3 Обзор структуры разработанного приложения и компонент

При разработке приложений были использованы как готовые компоненты, предоставляемые различными библиотеками (*react-router*, *react-redux* и др.). В общем компоненты можно разделить на следующие группы:

1. Простые компоненты. Данные компоненты служат для вывода информации а так же стилизации типовых элементов, таких как заголовки (*title*) параграфы(*p*) и др. В общем виде работу таких компонентов можно представить, как черный ящик получающий данные на входе, генерирующий *dom*-дерево внутри, и

возвращающий корневой элемент (Рисунок 3.1). В разработанном программном продукте присутствуют следующие компоненты такого типа:

HomePage – компонент генерирует содержимое первоначальной страницы приложения.

ErrorPage – компонент генерирует сообщение об ошибках, в случае не правильно заданного адреса.

ErrorMessage – сообщение об ошибках.

Views (*EducationView*, *EmploymentView*, *ProjectView*, *SkillView*) – компоненты генерируют и контролируют отображение информации об проектах, навыках и др.

Counter – компонент выводит информацию об количестве зависимых сущностей.

Header, *Footer* – подписи хедера и футера.



Рисунок 3.1 – Схема работы простых компонентов

2. Компоненты обертки. Данные компоненты представляют собой своего рода обёртки над содержимым, содержащие общие компоненты для нескольких вариантов содержимого рисунок 3.2.

Main – контролирует вывод содержимого в зависимости от процесса загрузки. Пока данные не получены выводит изображение процесса загрузки, после получение данных содержимое.

ModalWindow – компонент добавляет функционал и элементы присущие модальному окну для любого вложенного содержимого.

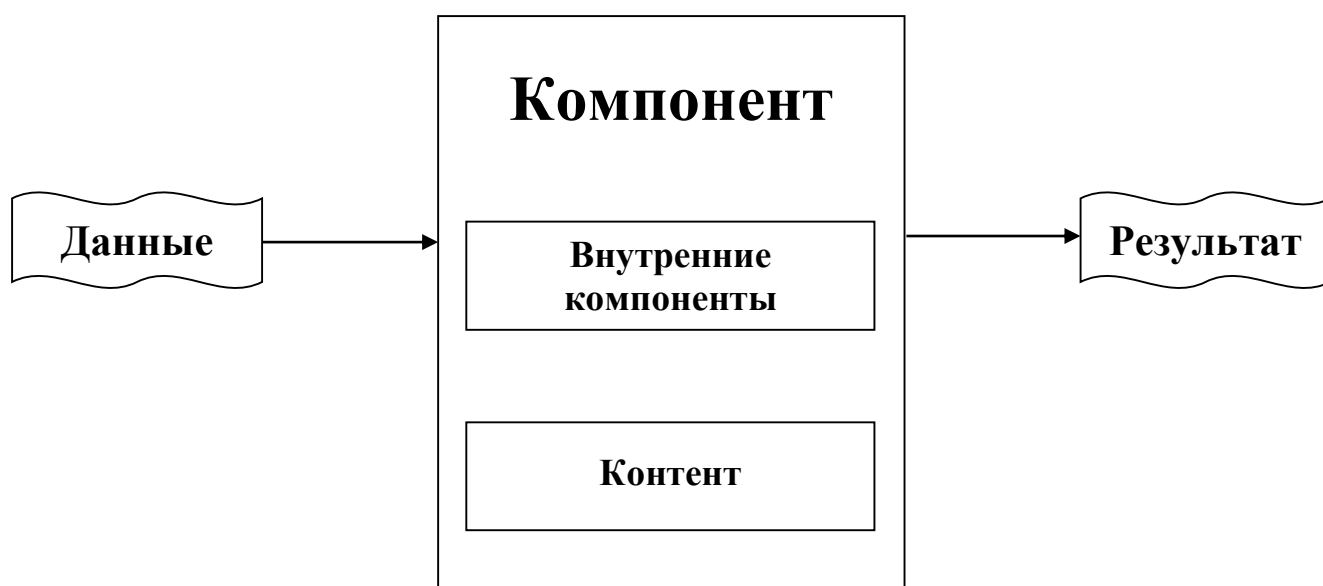


Рисунок 3.2 – Схема компонентов обёрток

3. Роутер. Данный компонент служит для контроля отображения содержимого в зависимости от содержимого адресной строки браузера. Данный функционал позволяет создать иллюзию работы с многостраничным приложением при этом работа ведется на одной странице.

4. Умные компоненты. Данные компоненты работают производят предварительную обработку данных изменения, способны динамически изменять свое состояние и осуществлять обмен данными с сервером. При этом данный вид компонент является наиболее значимым и сложным в разработке. Однако это так же один из наиболее менее представленных типов компонент в программе.

Совокупность разработанных компонент образует полнофункциональное, легко модернизируемое приложение. При этом основными преимуществами являются: чистый код (легкий и понятный в понимании); модульная структура позволяет модернизировать приложение с минимальным количеством временных затрат.

3.4 Обзор разработанного программного продукта

Для решения поставленной задачи нужно реализовать пользовательский интерфейс с использованием современных технологий. Для реализации выбрана библиотека `react.js`, без использования построенных на библиотеке фреймворков. Для работы со стилями используется `css` препроцессор `less`. Не мало важной частью работы является оптимизация работы с данными

В первую очередь необходимо определить, поставленную для разработки задачу. Необходимо разработать *front-end* часть приложения, которая имела возможность предоставлять управление автоматизированным тестированием по средствам взаимодействия с отдаленным сервером, а также предоставляла удобный интуитивно понятный пользовательский интерфейс. На рисунке 3.3 продемонстрирован внешний вид веб-приложения.













В первую очередь необходимо определить, что же представляет из себя разработанный программный продукт.

При запросе, сервер формирует *JSON*-файл, хранящий массив состояний проектов, хранящий актуальную о добавленных проектах. Информация об отдельном проекте предоставляется в следующем виде:

1. Стадия проекта, говорит о том какой этап разработки веб-приложения покрывает тестирование (возможны четыре состояния: *int*, *qa*, *production* и *staging*).
2. Название проекта.
3. *URL* адрес проекта.
4. Статус (хранит статус последнего тестирования).
5. Версия.
6. Дата последнего тестирования.
7. Результаты последнего тестирования.
8. Длительность тестирования.

Shared Services / Component (3)

Add Component

Name	INT	QA	Staging	Production
REACT PRO	⚙️ MISSING AURA.JSON	v1.1.1 FAILED 55 failed, 35 passed, 0 skipped, 90 total 3 months ago  	v1.1.1 FAILED 55 failed, 35 passed, 0 skipped, 90 total 3 months ago  	
JQUERY PRO	⚙️	v1.1.1 FAILED 135 failed, 15 passed, 0 skipped, 150 total 3 months ago  		v1.1.1 FAILED 135 failed, 15 passed, 0 skipped, 150 total 3 months ago  
WEB PRO	⚙️	v1.1.1 FAILED 55 failed, 35 passed, 0 skipped, 90 total 3 months ago  	v1.1.1 FAILED 55 failed, 35 passed, 0 skipped, 90 total 3 months ago  	

© 2018, Kuzmiankou Anatoli

Рисунок 3.3 – Пользовательский интерфейс веб-приложения

После получения данных формируется пользовательский интерфейс состоящий из элементов соответствующих одному тестируемому проекту и его версиям. На рисунке 3.4 показано интерфейс для отдельного элемента.

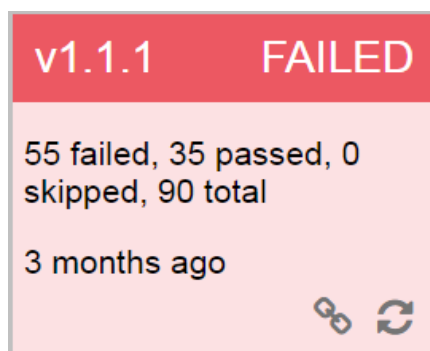


Рисунок 3.4 – Интерфейс отдельного элемента

Интерфейс отображает информацию об тестировании (версию, статус, время последнего тестирования), ссылку для отображения подробной информации и кнопку обновить (для перезапуска проекта).

Выводы по главе

Результатом третьей, и заключительной части работы, является разработанное приложение, использующее все преимущества компонентно-ориентированного программирования. Таким образом на практике проверены возможности и преимущества данного подхода с практической точки зрения.

Подробную информацию, а так же исходный код программы доступен по адресу *git@gitlab.com:kuzmiankou_anatoli/kuzmiankou_anatoli.git*.

Заключение

В ходе работы проведен анализ использования современных технологий в области разработки клиентской части веб-приложений. Из сказанного выше можно сделать вывод что как и раньше, так и в настоящее время разработка базируется на трех основополагающих технологиях (*HTML*, *CSS*, *JavaScript*). Тем не менее за годы использования технологии и подходы к разработке существенно изменились. Так изменилась специфика работы с *HTML*, к примеру существенно увеличены возможности интерпретации документов роботами за счет внедрения новых тегов. Существенно расширены возможности каскадных таблиц стилей. Так же основательно доработан *JavaScript*, а именно расширены возможности, добавлены новые языковые конструкции, а так же обеспечена однообразная работа скриптов в основных браузерах. Кроме того язык нашёл свои пути к использованию в *back-end* и мобильной разработке.

Однако, как было сказано в работе, современные реалии требуют увеличивать производительность при разработке, а так же снижать стоимость конечного продукта. Кроме того нужно учитывать необходимость максимально открытой и чистой разработки, так как коммуникация между разработчиками затруднена из-за языковых различий и разбросу по большому пространству. При этом использование библиотек и фреймворков позволяет значительно увеличить производительность команд.

Так же в работе фокусируется внимание на возможностях и специфике компонентно-ориентированного программирования во *front-end*. Ключевым преимуществом данного подхода является создание легко модернизируемых приложений. Так приложение можно представить, как набор взаимодействующих модулей, каждый из которых может быть изменен или заменен на более эффективный без вреда для всей системы в целом.

Для практической проверки возможностей использования компонентного подхода, разработано клиентское приложение по управлению автоматизированным тестированием проектов. Данный программный продукт использует всю мощностъ подхода, а так же несколько типов компонент. В третьей главе произведен подробный разбор разработанного программного продукта.

Таким образом в исследовательской работе произведен обзор некоторых технологий веб-разаботке, проанализированы подходы, преимущества и недостатки. Так же проанализирован компонентный подход и проверено на практике его полезность при создании современных приложений.

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

API – Application Programming Interface

AJAX – Asynchronous Javascript and XML

BOM – Browser Object Model

CMS – Content Management System

CSS – Cascading Style Sheets

CSS Flexbox – Flexible Box Layout Module

DOM – Document Object Model

HTML – Hyper Text Markup Language

JS – JavaScript

JSON – JavaScript Object Notation

LESS – Leaner Style Sheets

MVC – Model-View-Controller

MVVM – Model-View-ViewModel

SASS – Syntactically Awesome Stylesheets

СПИСОК ТЕРМИНОВ

AJAX: подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером.

Back-end: программно-аппаратной части сервиса.

Content Management System: информационная система или компьютерная программа, используемая для обеспечения и организации совместного процесса создания, редактирования и управления содержимым.

Cascading Style Sheets (CSS): формальный язык описания внешнего вида документа, написанного с использованием языка разметки.

COMET: любая модель работы веб-приложения, при которой постоянное HTTP-соединение позволяет веб-серверу отправлять данные браузеру без дополнительного запроса со стороны браузера.

CSS Flexbox: технология расположения блоков на странице, основанная на осях координат.

DOM-модель: объектная модель, используемая для XML/HTML-документов.

ECMAScript: встраиваемый расширяемый не имеющий средств ввода-вывода язык программирования, используемый в качестве основы для построения других скриптовых языков.

Front-end: клиентская сторона пользовательского интерфейса к программно-аппаратной части сервиса.

CSS Grid layout: технология двумерного макетирования в Веб, с возможностью размещения элементов в строках и столбцах.

Hyper Text Markup Language (HTML): стандартизированный язык разметки документов во Всемирной паутине.

JavaScript: мультипарадигменный язык программирования, поддерживающий объектно-ориентированный императивный и функциональный стили программирования и реализующий стандарт ECMO-262.

JSON: текстовый формат обмена данными, основанный на JavaScript.

Leaner Style Sheets (LESS): это динамический язык стилей, разработанный на основе технологии SASS.

Model-View-Controller: схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер.

Model-View-ViewModel: это шаблон архитектуры клиентских приложений, который был предложен Джоном Госсманом как альтернатива шаблонам MVC и MVP при использовании технологии связывания данных.

Syntactically Awesome Stylesheets (SASS): это метаязык на основе CSS, предназначенный для увеличения уровня абстракции CSS кода и упрощения файлов каскадных таблиц стилей.

UI-фреймворк: программная платформа, определяющая структуру программной системы. В отличие от обычного фреймворка, UI-фреймворк фокусируется на создании пользовательских интерфейсов.

Web-based Graphics Library (WebGL): программная библиотека для языка программирования JavaScript, позволяющая создавать интерактивную 3D-графику, функционирующую в широком спектре совместимых с ней веб-браузеров.

WebSocket: протокол полнодуплексной связи (может передавать и принимать одновременно) поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени.

React Native: это JS-фреймворк для создания нативно отображаемых iOS- и Android-приложений.

Rest API: согласованный набор ограничений, учитываемых при проектировании распределенной системы.

СПИСОК ИСПОЛЬЗОВАННОЙ ИСТОЧНИКОВ

1. Википедия / HTML [Электронный ресурс]. – 2018. – Режим доступа: <https://ru.wikipedia.org/wiki/HTML> (дата обращения: 24.03.2018).
2. Goldstein, A. HTML5 & CSS3 For The Real World, 2nd Edition // A. Goldstein, L. Lazaris, E. Weyl – SitePoint, 2015. – 350 p.
3. ITVDN / Взгляд в HTML6 [Электронный ресурс]. – 2018. – Режим доступа: <https://itvdn.com/ru/blog/article/looking-into-html6> (дата обращения: 24.03.2018).
4. Helix / HTML6 – когда необходима семантика [Электронный ресурс]. – 2017. – Режим доступа: <http://helix.su/html6-kogda-neobhodima-semantika.html> (дата обращения: 24.03.2018).
5. Википедия / CSS [Электронный ресурс]. – 2018. – Режим доступа: <https://ru.wikipedia.org/wiki/CSS> (дата обращения: 24.03.2018).
6. Сидерхолм, Д. CSS3 для веб-дизайнеров // Ден Сидерхолм – Манн, Иванов и Фербер, 2012. – 125 с.
7. Макфарланд, Д. Большая книга CSS3 // Д. Макфарланд – Питер, 2014. – 608 с.
8. W3C / Cascading Style Sheets, level 1 [Электронный ресурс]. – 2018. – Режим доступа: <https://www.w3.org/TR/CSS1/> (дата обращения: 24.03.2018).
9. W3C / Cascading Style Sheets, level 2 CSS Specification [Электронный ресурс]. – 2017. – Режим доступа: <https://www.w3.org/TR/2008/REC-CSS2-20080411/> (дата обращения: 24.03.2018).
10. W3C / Текущая работа CSS и как принять участие [Электронный ресурс]. – 2018. – Режим доступа: <https://www.w3.org/Style/CSS/current-work> (дата обращения: 24.03.2018).

11. WIRED / Discover What's New in CSS 4 [Электронный ресурс]. – 2018. – Режим доступа: <https://www.wired.com/2011/10/discover-whats-new-in-css-4/> (дата обращения: 24.03.2018).
12. SASS / Docs [Электронный ресурс]. – 2018. – Режим доступа: http://sass-lang.com/documentation/file.SASS_REFERENCE.html (дата обращения: 24.03.2018).
13. LESS / Docs [Электронный ресурс]. – 2018. – Режим доступа: <https://less-lang.info/> (дата обращения: 24.03.2018).
14. Про CSS / Flexbox [Электронный ресурс]. – 2018. – Режим доступа: <http://css.yoksel.ru/flexbox/> (дата обращения: 24.03.2018).
15. CSS-TRICKS / A Complete Guide to Grid [Электронный ресурс]. – 2018. – Режим доступа: <https://css-tricks.com/snippets/css/complete-guide-grid/> (дата обращения: 24.03.2018).
16. Can I Use / Flexbox [Электронный ресурс]. – 2018. – Режим доступа: <https://caniuse.com/#search=flexbox> (дата обращения: 12.04.2018).
17. Can I Use / Grid Layout [Электронный ресурс]. – 2018. – Режим доступа: <https://caniuse.com/#search=Grid%20layout> (дата обращения: 12.04.2018).
18. Википедия / JavaScript [Электронный ресурс]. – 2018. – Режим доступа: <https://ru.wikipedia.org/wiki/JavaScript> (дата обращения: 24.03.2018).
19. Lindley, C. JavaScript Enlightenment // Cody Lindley – O'Reilly Media, 2012. – 166 p.
20. Haverbeke, M. Eloquent JavaScript // Marijn Haverbeke – No Starch Press, 2011. – 224 p.
21. Osmani, A. Developing Backbone.js Application // Addy Osmani – O'Reilly Media, 2013. – 373 p.
22. Zakas N. 1. What is JavaScript? // Professional JavaScript for Web Developers. — 2nd ed. — USA, Canada: Wiley Publishing, Inc., 2009. — P. 3.
23. Википедия / ECMAScript [Электронный ресурс]. – 2018. – Режим доступа: <https://en.wikipedia.org/wiki/ECMAScript#Versions> (дата обращения: 18.04.2018).

24. ECMA international / ECMAScript 2015 Language Specification [Электронный ресурс]. – 2018. – Режим доступа: <http://www.ecma-international.org/ecma-262/6.0/index.html> (дата обращения: 18.04.2018).
25. ECMA international / ECMAScript 2016 Language Specification [Электронный ресурс]. – 2018. – Режим доступа: <http://www.ecma-international.org/ecma-262/7.0/index.html> (дата обращения: 18.04.2018).
26. ECMA international / ECMAScript 2017 Language Specification (ECMA-262, 8th edition, June 2017) [Электронный ресурс]. – 2018. – Режим доступа: <http://www.ecma-international.org/ecma-262/8.0/index.html> (дата обращения: 18.04.2018).
27. Koch P.-P. Chapter 6. BOM // ppk on JavaScript. — 1st ed. — New Riders Press, 2006. — 528 p.
28. MDN / Mozilla Developer Network [Электронный ресурс]. – 2018. – Режим доступа: <https://developer.mozilla.org/ru/> (дата обращения: 24.03.2018).
29. Zakas N. The Document Object Model // Professional JavaScript for Web Developers. — 2nd ed. — USA, Canada: Wiley Publishing, Inc., 2009. — P. 261 — 317.
30. Википедия / Веб-приложение [Электронный ресурс]. – 2018. – Режим доступа: <https://ru.wikipedia.org/wiki/Веб-приложение> (дата обращения: 24.03.2018).
31. Крейн, Д. Аяx в действии // Крейн Д., Паскарелло Э., Джеймс Д. – Вильямс, 2006. – 640 с.
32. Википедия / AJAX [Электронный ресурс]. – 2018. – Режим доступа: <https://ru.wikipedia.org/wiki/AJAX> (дата обращения: 24.03.2018).
33. Википедия / Comet [Электронный ресурс]. – 2018. – Режим доступа: [https://ru.wikipedia.org/wiki/Comet_\(программирование\)](https://ru.wikipedia.org/wiki/Comet_(программирование)) (дата обращения: 24.03.2018).
34. Википедия / WebOS [Электронный ресурс]. – 2018. – Режим доступа: <https://ru.wikipedia.org/wiki/WebOS> (дата обращения: 24.03.2018).

35. Oracle / webnotes [Электронный ресурс]. – 2018. – Режим доступа: <http://www.oracle.com/technetwork/java/javase/webnotes-136672.html> (дата обращения: 24.03.2018).
36. Kiessling, M. Node Beginner / Manuel Kiessling // The Node Beginner Book [Электронный ресурс]. – 2018. – Режим доступа: <http://www.nodebeginner.org/> (дата обращения: 24.03.2018).
37. Holowaychuk, T. Mastering NodeJS / TJ Holowaychuk // Mastering NodeJS Open Source Node eBook [Электронный ресурс]. – 2018. – Режим доступа: <http://visionmedia.github.io/masteringnode/> (дата обращения: 24.03.2018).
38. MDN web docs / Rhino history [Электронный ресурс]. – 2018. – Режим доступа: <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/History> (дата обращения: 18.04.2018).
39. Mozilla wiki / Jaeger Monkey [Электронный ресурс]. – 2018. – Режим доступа: <https://wiki.mozilla.org/JaegerMonkey> (дата обращения: 18.04.2018).
40. Chrome V8 / Introduction [Электронный ресурс]. – 2018. – Режим доступа: <https://developers.google.com/v8/?hl=ru> (дата обращения: 18.04.2018).
41. WebKit / The WebKit Open Source Project [Электронный ресурс]. – 2018. – Режим доступа: <https://webkit.org/project/> (дата обращения: 18.04.2018).
42. KDE / Release of KDE Frameworks 5.45.0 [Электронный ресурс]. – 2018. – Режим доступа: <https://www.kde.org/announcements/kde-frameworks-5.45.0.php> (дата обращения: 18.04.2018).
43. Microsoft Blogs / An Early Look At IE9 for Developers [Электронный ресурс]. – 2018. – Режим доступа: <https://blogs.msdn.microsoft.com/ie/2009/11/18/an-early-look-at-ie9-for-developers/> (дата обращения: 18.04.2018).
44. Microsoft Blogs / Microsoft Edge's JavaScript engine to go open source [Электронный ресурс]. – 2018. – Режим доступа: <https://blogs.windows.com/msedgedev/2015/12/05/open-source-chakra-core/> (дата обращения: 18.04.2018).
45. OpenJDK / JDK8 [Электронный ресурс]. – 2018. – Режим доступа: <http://openjdk.java.net/projects/jdk8/> (дата обращения: 18.04.2018).

46. Википедия / JavaScript engine [Электронный ресурс]. – 2018. – Режим доступа: https://en.wikipedia.org/wiki/JavaScript_engine (дата обращения: 18.04.2018).
47. TypeScript / Documentation [Электронный ресурс]. – 2018. – Режим доступа: <https://www.typescriptlang.org/docs/home.html> (дата обращения: 24.03.2018).
48. Jansen, R. TypeScript: Modern JavaScript Development // R. Jansen – Packt, 2012. – 1087 p.
49. Murphey, R. jQuery Fundamentals / Rebecca Murphey // DailyJS - jQuery Fundamentals [Электронный ресурс]. – 2018. – Режим доступа: <http://jqfundamentals.com> (дата обращения: 24.03.2018).
50. Бибо, Б. jQuery. Подробное руководство по продвинутому JavaScript, 2-е издание // Б. Бибо, И. Кац – Спб.: Символ-плюс, 2011. – 623 с.
51. Бенедетти, Р. Изучаем работу с jQuery // Райан Бенедетти, Ронан Крэнли. – Спб.: Питер, 2012. – 508 с.
52. Bibeault, B. jQuery in Action, Third Edition // Bear Bibeault, Yehuda Katz and Aurelio De Rosa – Manning, 2015. – 504 с.
53. Бибо, Б. jQuery. Подробное руководство по продвинутому JavaScript // Бер Бибо, Иегуда Кац — Спб.: Символ-Плюс, 2009. — 384 с.
54. jQuery user interface / API Documentation [Электронный ресурс]. – 2018. – Режим доступа: <http://api.jqueryui.com/> (дата обращения: 24.03.2018).
55. Википедия / Bootstrap (фреймворк) [Электронный ресурс]. – 2018. – Режим доступа: [https://ru.wikipedia.org/wiki/Bootstrap_\(фреймворк\)](https://ru.wikipedia.org/wiki/Bootstrap_(фреймворк)) (дата обращения: 24.03.2018).
56. blog.twitter.com / Bootstrap from Twitter [Электронный ресурс]. – 2018. – Режим доступа: <https://blog.twitter.com/2011/bootstrap-from-twitter> (дата обращения: 24.03.2018).
57. The Bootstrap Blog / Bootstrap 4 alpha [Электронный ресурс]. – 2018. – Режим доступа: <http://blog.getbootstrap.com/2015/08/19/bootstrap-4-alpha/> (дата обращения: 24.03.2018).

58. Stefanov, S. React: Up & Running // S. Stefanov. – O'Reilly Media, 2016. – 222 p.
59. INTUIT / Лекция: Компонентное программирование в .Net – Введение в теорию программирования [Электронный ресурс]. – 2018. – Режим доступа: <http://www.intuit.ru/department/se/tppobj/17/> (дата обращения: 24.03.2018).
60. Википедия / Компонентно-ориентированное программирование [Электронный ресурс]. – 2018. – Режим доступа: [https://ru.wikipedia.org/wiki/Компонентно-ориентированное программирование](https://ru.wikipedia.org/wiki/Компонентно-ориентированное_программирование) (дата обращения: 24.03.2018).
61. Ext JS Documentation / Blog [Электронный ресурс]. – 2018. – Режим доступа: <http://docs.sencha.com/blog> (дата обращения: 27.04.2018).
62. eWeek / Sencha Ext JS 5 Unifies Mobile, Desktop App Dev [Электронный ресурс]. – 2018. – Режим доступа: <http://www.eweek.com/blogs/first-read/sencha-ext-js-5-unifies-mobile-desktop-app-dev> (дата обращения: 27.04.2018).
63. Dr.Doob's / Sencha Ext JS 5 Streamlines and Unifies [Электронный ресурс]. – 2018. – Режим доступа: <http://www.drdoobbs.com/web-development/sencha-ext-js-5-streamlines-and-unifies/240168397> (дата обращения: 27.04.2018).
64. Garcia, J. Ext JS in Action // Jesus Garcia. – Manning Publication, 2011. – 495 p.
65. Dojo / Dojo documentation [Электронный ресурс]. – 2018. – Режим доступа: <https://dojotoolkit.org/documentation/> (дата обращения: 06.05.2018).
66. Metanit / Определение паттерна MVVM [Электронный ресурс]. – 2018. – Режим доступа: <https://metanit.com/sharp/wpf/22.1.php> (дата обращения: 06.05.2018).
67. AngularJS / Tutorial [Электронный ресурс]. – 2018. – Режим доступа: <https://docs.angularjs.org/tutorial> (дата обращения: 06.05.2018).

68. Angular / Documentation [Электронный ресурс]. – 2018. – Режим доступа: <https://angular.io/docs> (дата обращения: 06.05.2018).
69. Кулямин, В. Технология программирования. Компонентный подход // Виктор Кулямин – М.: ИСП РАН, 2006. – 315 с.
70. Кулямин, В. Компонентный подход в программировании // Виктор Кулямин – М.: НОУ «Интуит», 2016. – 590 с.
71. Блинов, И. С. Методы программирования, второе издание // И. С. Блинов, В. С. Романчик – Минск: издательство «Четыре четверти», 2013. – 896 с.
72. Schildt, H. Java: The Complete Reference. Tenth Edition // Herbert Schildt – McGraw-Hill Education, 2017. – 1923 p.
73. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C#. 4-е издание. // Дж. Рихтер – Питер, 2017. – 896 с.
74. Windows Dev Center / The Component Object Model [Электронный ресурс]. – 2018. – Режим доступа: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms694363\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms694363(v=vs.85).aspx) (дата обращения: 27.03.2018).
75. Википедия / Component Object Model [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Component_Object_Model (дата обращения: 27.03.2018).
76. Mulesoft / What is rest api design? [Электронный ресурс]. – 2018. – Режим доступа: <https://www.mulesoft.com/resources/api/what-is-rest-api-design> (дата обращения: 08.05.2018).
77. GraphQL / learn [Электронный ресурс]. – 2018. – Режим доступа: <https://graphql.org/learn/> (дата обращения: 08.05.2018)
78. Banks, A. Learning React // A. Banks, E. Porcello. – O'Reilly Media, 2017. – 350 p.
79. Babel / Try it out [Электронный ресурс]. – 2018. – Режим доступа: <https://babeljs.io/repl/> (дата обращения: 24.03.2018).

80. Metanit / Жизненный цикл компонента [Электронный ресурс]. – 2018. – Режим доступа: <https://metanit.com/web/react/2.6.php> (дата обращения: 24.03.2018).
81. React training / React router [Электронный ресурс]. – 2018. – Режим доступа: <https://reacttraining.com/react-router/> (дата обращения: 24.03.2018).
82. GitHub / React motion [Электронный ресурс]. – 2018. – Режим доступа: <https://github.com/chenglou/react-motion> (дата обращения: 24.03.2018).
83. React / Add-Ons [Электронный ресурс]. – 2018. – Режим доступа: <https://reactjs.org/docs/addons.html> (дата обращения: 24.03.2018).
84. Airbnb / Enzyme [Электронный ресурс]. – 2018. – Режим доступа: <http://airbnb.io/enzyme/> (дата обращения: 24.03.2018).
85. Redux / Read Me [Электронный ресурс]. – 2018. – Режим доступа: <https://redux.js.org/> (дата обращения: 24.03.2018).
86. Boduch, A. Flux Architecture // A. Boduch. – Packt Publishing, 2017. – 316 p.
87. Хабрахабр / Разбираемся с Flux, реактивной архитектурой от facebook [Электронный ресурс]. – 2018. – Режим доступа: <https://habrahabr.ru/post/246959/> (дата обращения: 27.03.2018).
88. GitHub / Flux [Электронный ресурс]. – 2018. – Режим доступа: <https://github.com/facebook/flux?ref=stackshare> (дата обращения: 27.03.2018).
89. GitHub / Redux [Электронный ресурс]. – 2018. – Режим доступа: <https://github.com/reactjs/redux?ref=stackshare> (дата обращения: 27.03.2018).
90. Material-UI / React components that implement Google's Material Design [Электронный ресурс]. – 2018. – Режим доступа: <https://material-ui-next.com> (дата обращения: 02.04.2018).
91. React Desktop / Documentation. [Электронный ресурс]. – 2018. – Режим доступа: <http://reactdesktop.js.org> (дата обращения: 02.04.2018).

92. GitHub / Semantic-UI-React. [Электронный ресурс]. – 2018. – Режим доступа: <https://github.com/Semantic-Org/Semantic-UI-React> (дата обращения: 02.04.2018).
93. React Bootstrap / Documentation. [Электронный ресурс]. – 2018. – Режим доступа: <https://react-bootstrap.github.io/getting-started/introduction> (дата обращения: 02.04.2018).

Приложение А. Исходный код некоторых компонент

App.js:

```
import React, { Component } from 'react';
import { Router, Switch, Route } from 'react-router'
import createHistory from 'history/createBrowserHistory';
import Header from './components/Header';
import Footer from './components/Footer';
import HomePage from './pages/HomePage';
import AboutMePage from './pages/AboutMePage';
import ItemPage from './pages/ItemPage';
import ErrorPage from './pages/ErrorPage';
import LogoPage from './pages/LogoPage';
import TaskManagerPage from './pages/TaskManagerPage';
import * as PropTypes from 'prop-types';

class App extends Component {
  static PAGE_LINKS = [
    { "link": "/task-manager", "title": "Task manager" },
    { "link": "/about", "title": "About me" }
  ];

  getChildContext() {
    return {
      pageLinks: App.PAGE_LINKS
    }
  }

  render() {
    return (
      <div className="app">
        <Header text="Front-end EPAM training" />
        <Router history={ createHistory() }>
          <Switch>
            <Route exact path="/" component={ HomePage } />
            <Route path="/about" component={ AboutMePage } />
            <Route path="/task-manager" component={ TaskManagerPage } />
            <Route path="/item/:name/:environment" component={ ItemPage } />
            <Route path="/logo.txt" component={ LogoPage } />
            <Route path="/*" component={ ErrorPage } />
          </Switch>
        </Router>
      </div>
    );
  }
}
```

```

    </Router>
    <Footer date="2017" title="Kuzmiankou Anatoli" />
  </div>
);
}

static childContextTypes = {
  pageLinks: PropTypes.arrayOf(PropTypes.shape({
    link: PropTypes.string,
    title: PropTypes.string
  }))
}
}

export default App;

```

TaskManagerPage.js:

```

import React, { Component } from 'react';
import AddComponentWindow from '../components/AddComponentWindow';
import TableBox from '../components/TableBox';
import TaskComponentManager from '../components/TaskComponentManager';
import Counter from '../components/Counter';
import Main from '../components/Main';
import * as PropTypes from 'prop-types';
import { connect } from 'react-redux'
import { loadTasks, loadTasksSuccess } from '../redux/action/TaskAction';

class TaskManagerPage extends Component {
  componentDidMount() {
    this.props.loadingData();
  }

  createTaskManager(data, number) {
    const { name, environments } = data;
    return (
      <TaskComponentManager taskName={ name } key={ name } tasks={ environments } />
    );
  }

  static ENVIRONMENTS = [
    { name: "int", title: "INT" },

```

```

    { name: "qa", title: "QA" },
    { name: "staging", title: "Staging" },
    { name: "production", title: "Production" }
  ];

```

```

getChildContext() {
  return {
    environments: TaskManagerPage.ENVIRONMENTS
  };
}

```

```

render() {
  const { loading, tasks } = this.props;
  return (
    <Main name="Task manager" loading={ loading }>
      <Counter title="Shared Services / Component" name="components-count" count={ tasks.length } />
      <AddComponentWindow />
      <TableBox>
        { tasks.map( (item, number) => this.createTaskManager(item, number) ) }
      </TableBox>
    </Main>
  );
}

```

```

static childContextTypes = {
  environments: PropTypes.arrayOf(PropTypes.shape({
    name: PropTypes.string,
    title: PropTypes.string
  }))
}

```

```

const mapStateToProps = state => {
  const { TaskReducer } = state;
  return { ...TaskReducer };
}

```

```

const mapDispatchToProps = dispatch => ({
  loadingData: () => {
    dispatch(loadTasks());
    fetch('http://localhost:9999/data/projects')
      .then(res => res.json())

```

```

        .then(json => dispatch(loadTasksSuccess(json)));
    }
});

export default connect(mapStateToProps, mapDispatchToProps)(TaskManagerPage);

```

Task.js:

```

import * as BodyFactory from "../lib/TaskBodyFactory";
import * as StatusFactory from "../lib/TaskStatusFactory";
import TaskStatus from "../lib/TaskStatus";
import * as PropTypes from 'prop-types';
import { connect } from 'react-redux';
import { refreshTask, refreshTaskSuccess } from '../../redux/action/TaskAction';
import autoBind from 'react-autobind';

class Task extends Component {
  constructor(props) {
    super(props);
    autoBind(this);
  }

  refreshAction() {
    const { name, env } = this.props;
    this.props.refreshAction({ name, env });
  }

  shouldComponentUpdate(nextProps, nextState) {
    for(let index in this.props) {
      if(this.props[index] !== nextProps[index]) {
        return true;
      }
    }
    return false;
  }

  render() {
    const { env, link, data } = this.props;
    const { status, version } = this.props.data;
    return(
      <td className={ `task task-${ env } task-${ TaskStatus[status].toLowerCase() }` }>
        <div className="header">

```

```

    <div className="version">{ version }</div>
    { StatusFactory.createStatus(status) }
  </div>
  { BodyFactory.createBodyElem(data) }
  <div className="control-panel">
    <a href={ link } className="link" ><span className="icon-link" /></a>
    <button className="refresh icon-refresh" onClick={ this.refreshAction } />
  </div>
</td>
);
}

```

```

static defaultProps = {
  name: "PROJECT",
  env: "int",
  data: {
    version: "v-.-.-",
    status: TaskStatus.MESSING
  }
}

```

```

static propTypes = {
  name: PropTypes.string,
  env: PropTypes.string,
  data: PropTypes.shape({
    version: PropTypes.string,
    status: PropTypes.number,
    timestamps: PropTypes.string,
    testResult: PropTypes.shape({
      total: PropTypes.number,
      failed: PropTypes.number,
      passed: PropTypes.number,
      skipped: PropTypes.number
    }),
    logo: PropTypes.string
  }),
  link: PropTypes.string
};
}

```

```

const mapDispatchToProps = dispatch => ({
  refreshAction: params => {

```



```

    dispatch(refreshTask());
    fetch('http://localhost:9999/data/refresh', {
      method: 'post',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(params)
    }).then(res => res.json())
      .then(data => dispatch(refreshTaskSuccess(params, data)));
  }
});

export default connect(null, mapDispatchToProps)(Task);

```

TaskBodyFactory.js:

```

import React from 'react';
import Status from './TaskStatus';
import moment from 'moment';

const viewQueue = status => (
  <div className="inqueue" key="inqueue">In queue</div>
)

const viewTests = tests => (
  <div key="tests">
    { `${ tests.failed } failed, ${ tests.passed } passed, ${ tests.skipped } skipped, ${ tests.total } total` }
  </div>
);

const viewLogo = link => (
  <div key="link">See <a href={ `${ link }` } target="_blank">{ link }</a></div>
);

const viewTime = time => (
  <div key="date">{ moment(time).fromNow() }</div>
);

```

```

export const createBodyElem = data => (
  <div className="body">
    { data.status === Status.QUEUE && viewQueue(data.status) }
    { data.testResult && viewTests(data.testResult) }
    { data.logo && viewLogo(data.logo) }
    { data.timestamps && viewTime(data.timestamps) }
  </div>
);

```

TaskStatus.js:

```

const TaskStatus = {
  QUEUE: 'QUEUE',
  MISSING: 'MISSING',
  RUNNING: 'RUNNING',
  FAILED: 'FAILED',
  SUCCESS: 'SUCCESS',
  W_O_FAILED: 'W_O_FAILED'
}

```

```

export default TaskStatus;

```

TaskStatusFactory.js:

```

import React from 'react';
import Status from './TaskStatus';

const titles = {
  [Status.QUEUE]: "IN QUEUE",
  [Status.MISSING]: "MISSING AURA.JSON",
  [Status.RUNNING]: "RUNNING",
  [Status.FAILED]: "FAILED",
  [Status.W_O_FAILED]: "FAILED"
}

const icons = {
  [Status.SUCCESS]: "icon-ok",
  [Status.W_O_FAILED]: "icon-warning"
}

```

```

}

export const createStatus = status => (
  <div className={ `status ${ icons[status] }` }>{ titles[status] }</div>
);

```

AddComponentWindow.js:

```

import React, { Component } from 'react';
import TextForm from './TextForm';
import ModalWindow from './ModalWindow';
import * as PropTypes from 'prop-types';
import { addProjectTasks, addProjectTasksSuccess } from '../redux/action/TaskAction';
import { connect } from 'react-redux';
import autoBind from 'react-autobind';

```

```

class AddComponentWindow extends Component {
  constructor(props) {
    super(props);
    autoBind(this);
  }

  getChildContext() {
    const { environments } = this.context;
    const fields = [{ name: "name", labelValue: "Item name" }];
    environments && environments.forEach( item =>
      fields.push({
        name: item.name,
        labelValue: `${ item.title } environment url`
      })
    );
    return { fields };
  }

```

```

  formAction(data) {
    this.hideWindow();
    this.props.addComponent(data);
    this.form.resetAction();
  }

```

```

  hideWindow() {
    this.modalWindow.hide();
  }

```

```

}

showWindow() {
  this.modalWindow.show();
}

render() {
  return (
    <figure className="add-component" name="add-component">
      <button name="add-component" onClick={ this.showWindow }
        className="btn-add-component">Add Component</button>
      <ModalWindow ref={ window => this.modalWindow = window } title="Add component item">
        <TextForm ref={ form => this.form = form } submitName="Add" submitAction={ this.formAction }/>
      </ModalWindow>
    </figure>
  );
}

static contextTypes = {
  environments: PropTypes.arrayOf(PropTypes.shape({
    name: PropTypes.string,
    title: PropTypes.string
  })),
};

static childContextTypes = {
  fields: PropTypes.arrayOf(PropTypes.shape({
    name: PropTypes.string,
    labelValue: PropTypes.string
  }))
};

const mapDispatchToProps = dispatch => ({
  addComponent: params => {
    dispatch(addProjectTasks(params.name));
    fetch('http://localhost:9999/data/add-project', {
      method: 'post',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
    },
  },

```

```

        body: JSON.stringify(params)
    }).then(res => res.json())
    .then(json => dispatch(addProjectTasksSuccess(json)));
}
});

export default connect(null, mapDispatchToProps)(AddComponentWindow);

```

ModalWindow.js:

```

import React, { Component } from "react";
import * as PropTypes from 'prop-types';
import autoBind from 'react-autobind';

class ModalWindow extends Component {
  constructor(props) {
    super(props);
    autoBind(this);
  }

  componentDidMount() {
    if(this.props.startVisible) {
      this.show();
    }
    else {
      this.hide();
    }
  }

  hide() {
    this.window.style.display = "none";
  }

  show() {
    this.window.style.display = "block";
  }

  render() {
    const { id, name, title, children } = this.props;
    return (
      <figure
        id={ id }

```

```

    name={ name }
    className={ "modal modal-" + name }
    ref={ window => this.window = window }
  >
  <div className="modal-content">
    <h2>{ title }</h2>
    <button className="btn-close" onClick={ this.hide }>x</button>
    { children }
  </div>
</figure>
);
}

static defaultProps = {
  name: "window",
  title: "Modal window",
  startVisible: false
};

static propTypes = {
  id: PropTypes.string,
  name: PropTypes.string,
  title: PropTypes.string,
  startVisible: PropTypes.bool
};
}

export default ModalWindow;

```

TaskComponentManager.js:

```

import React, { Component } from 'react';
import Task from './task/Task';
import * as PropTypes from 'prop-types';
import { connect } from 'react-redux';
import { removeProjectTasks, removeProjectTasksSuccess } from '../redux/action/TaskAction';
import autoBind from 'react-autobind';

class TaskComponentManager extends Component {
  constructor(props) {
    super(props);
    autoBind(this);
  }

```

```

}

createTask(key, task) {
  const name = this.props.taskName;
  return (
    <Task
      name={ name }
      key={ key }
      env={ key }
      data={ task }
      link={ `/item/${ name }/${ key }` }
    />
  )
}

createEmptyTask(key) {
  return <td key={ key }><div className="task-empty" /></td>
}

createTasks(tasks) {
  return this.context.environments.map(item => {
    const name = item.name;
    const task = tasks[name];
    return task ? this.createTask(name, task) : this.createEmptyTask(name);
  });
}

removeComponent() {
  this.props.removeAction({ name: this.props.taskName });
}

render() {
  const { name, taskName, tasks } = this.props;
  return (
    <tbody className={ `task-component-manager manager-${ name }` }>
      <tr>
        <td className="task-manager">
          <h2 className="title">{ taskName }</h2>
          <ul className="settings icon-setting">
            <li onClick={ this.removeComponent }>Remove</li>
          </ul>
        </td>
      </tr>
    </tbody>
  )
}

```

```

        { this.createTasks(tasks) }
      </tr>
    </tbody>
  );
}

static defaultProps = {
  name: "task-manager"
};

static contextTypes = {
  environments: PropTypes.arrayOf(PropTypes.shape({
    name: PropTypes.string,
    title: PropTypes.string
  })),
};

static propTypes = {
  name: PropTypes.string,
  taskName: PropTypes.string,
  tasks: PropTypes.objectOf(PropTypes.shape({
    version: PropTypes.string,
    status: PropTypes.string,
    testResult: PropTypes.shape({
      total: PropTypes.number,
      failed: PropTypes.number,
      passed: PropTypes.number,
      skipped: PropTypes.number
    }),
    timestatmps: PropTypes.string,
    logo: PropTypes.string
  }))
};
}

const mapDispatchToProps = dispatch => ({
  removeAction: params => {
    dispatch(removeProjectTasks());
    fetch('http://localhost:9999/data/remove-tasks', {
      method: 'post',
      headers: {
        'Accept': 'application/json',

```



```

    'Content-Type': 'application/json'
  },
  body: JSON.stringify(params)
}).then(res => res.json())
  .then(data => dispatch(removeProjectTasksSuccess(params.name, data.result)));
}
});

export default connect(null, mapDispatchToProps)(TaskComponentManager);

```

NavBar.js:

```

import React from 'react';
import * as PropTypes from 'prop-types';

const NavBar = (props, context) => {
  const { name } = context
  return (
    <nav className="nav-bar">
      <ul className="nav-bar-menu">
        <li className="horizontal"><a href="/">Home</a></li>
        {
          (name)
            ? <li className="horizontal">{ name }</li>
            : undefined
        }
      </ul>
    </nav>
  );
}

NavBar.contextTypes = {
  name: PropTypes.string
};

export default NavBar;

```

TableBox.js:

```

import React, { Component } from 'react';
import * as PropTypes from 'prop-types';

```

```

class TableBox extends Component {

  createTH(name) {
    return (
      <th className="column-tittle" key={ name }>{ name }</th>
    );
  }

  render() {
    const { environments } = this.context;
    return (
      <table className="table-box">
        <thead>
          <tr>
            { this.createTH("Name") }
            { environments && environments.map( item => this.createTH(item.title) ) }
          </tr>
        </thead>
        { this.props.children }
      </table>
    )
  }

  static contextTypes = {
    environments: PropTypes.arrayOf(PropTypes.shape({
      title: PropTypes.string
    })),
  };
}

export default TableBox;

```

Main.js:

```

import React, { Component } from 'react';
import * as PropTypes from 'prop-types';
import NavBar from './NavBar';
import Loading from './Loading';

class Main extends Component {
  getChildContext() {
    const { name } = this.props;

```

```

return {
  name
}
}

render() {
  const { loading, children } = this.props;
  return (
    <main>
      <NavBar />
      {
        (loading)
          ? <Loading />
          : <div className="main-content">{ children }</div>
      }
    </main>
  );
}

static childContextTypes = {
  name: PropTypes.string,
  loading: PropTypes.bool
};
}

export default Main;

```