МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение

высшего образования

«Санкт-Петербургский национальный исследовательский университет

информационных технологий, механики и оптики»

**Кафедра Компьютерного проектирования и дизайна**

Направление (специальность) ⎯ 09.04.02 Информационные системы и технологии

Специализация ⎯ Веб технологии (СОП)

Научно-исследовательская работа

**ТЕМА:** Подходы к оптимизации разработки сайтов с использованием современных веб-технологий.

**4-ИЙ ЭТАП:** Разработка пользовательского интерфейса с использованием библиотеки react.js

ВЫПОЛНИЛ

Студент группы \_s4207c\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.С. Кузменков

№ группы подпись, дата ФИО

ПРОВЕРИЛ \_\_к.т.н., доц.\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ю.А. Сокуренко

ученая степень, должность подпись, дата ФИО

САНКТ-ПЕТЕРБУРГ  
2018 г.

**СОДЕРЖАНИЕ**

|  |  |
| --- | --- |
| Введение...…………………………………………………………………….. | 3 |
| 1. Компонентный подход программирования……………………………… | 7 |
| 2. Обзор технологий *front-end* разработки веб-приложений ……………… | 9 |
| 3. Компонентный подход во *front-end* ……………………………………… | 14 |
| 4. Компонентная разработка с использованием react.js…………………… | 17 |
| 5. Разработка *front-end* приложения управления автоматизированным тестированием проектов……………………………………………………………… | 21 |
| Заключение…………………………………………………………………… | 23 |
| Список использованной литературы……………………………………….. | 25 |
| Приложение А. Исходный код……………………………………………… | 31 |

**ВВЕДЕНИЕ**

На протяжении существования цивилизованного общества одной из ключевых факторов развития являлась возможность сохранения, передачи и использования информации. Совершенствование технологий и подходов привело к созданию первой сети обмена информацией между ведущими американскими университетами. С этого момента начинается развитие веб-технологий для передачи и использования различных видов информации.

С появлением всемирной паутины и ростом её использования становиться понятно что языков разметки (*HTML*, *XML*) не достаточно для удовлетворения пользовательских потребностей. Что привело к развитию более сложных и эффективных средств разработки веб-приложений.

Не смотря на высокие темпы развития технологий и подходов к разработке серверных приложений, становиться очевидным необходимость создания мощных средств разработки на стороне клиента, позволяющего не только переложить часть нагрузки на клиента, но и создавать более эффективные веб-приложения. Наиболее удачным решением стало использование совокупности разметки (*HTML*), таблицы стилей (*CSS*) и динамических скриптов (*JS*).

**Анализ современных тенденций**.

В настоящее время существует несколько

**Актуальность темы.** В настоящее время *JavaScript* и основанные на нём технологии широко используются в следующих областях:

1. Электронный бизнес. По результатам исследований, большинство ресурсов Интернета так или иначе связаны с коммерческой деятельностью. Интернет используется для рекламы и непосредственной продажи товаров и услуг, для маркетинговых исследований, электронных платежей и управления банковскими счетами.
2. Средства массовой информации. По жанрам интернет-издания не отличаются от офлайновых – есть новостные сайты, литературные, научно-популярные, детские, женские и т. п. Однако, если офлайновые издания выпускаются периодически (раз в день, неделю, месяц), то интернет-издания обновляются по мере появления нового материала. Также существует интернет-радио и интернет-телевидение.
3. Литература, музыка, кино. Электронные библиотеки, доступные через Интернет, содержат огромное число произведений. При этом многие книги, доступные в сети, давно стали библиографической редкостью, а некоторые и вообще не издавались.
4. Связь. Электронная почта в настоящее время является одним из наиболее используемых средств связи. Также популярны IP-телефония и использование таких программ, как Skype (бесплатное проприетарное программное обеспечение с закрытым кодом, обеспечивающее шифрованную голосовую связь и видеосвязь через Интернет между компьютерами (VoIP), используя технологии пиринговых сетей, а также платные услуги для звонков на мобильные и стационарные телефоны). 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. Социальные сети стали своего рода Интернет-пристанищем, где каждый может найти техническую и социальную базу для создания своего виртуального «Я». При этом каждый пользователь получил возможность не просто общаться и творить, но и делиться плодами своего творчества с многомиллионной аудиторией той или иной социальной сети.
6. Краудсорсинг. Интернет оказался хорошим средством для решения общественно значимых задач силами множества добровольцев, координирующих при этом свою деятельность.

Широкое распространение технологий, основанных на *JS* во многом обусловлено применением компонентно-ориентированного подхода. Так, на данный момент многие морально устаревшие технологии держаться только лишь благодаря большому количеству готовых компонент.

Развитие технологий привело к появлению библиотеки *react*. Данная библиотека, не смотря на свои недостатки как работа только на уровне представления, по праву представляет собою будущее *front-end* разработки.

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

Так же совместно с научным руководителем были определены объект, предмет, цель и задачи исследования:

**Постановка задачи** (кратко сформулировать основную задачу исследования по теме ВКР)

Постановка следующих задач:

* Исследовать наиболее популярные технологии управления визуализацией информации для интерактивных систем.
* Провести апробацию технологии управления жестами.
* Проанализировать результаты апробации по необходимым критериям распознавания жестов.
* Изучить методы распознавания жестов.
* Исследовать комбинации данных методов для реализации рекомендаций по улучшению критериев распознавания динамических сложных жестов.

**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.

**Теоретическая и прикладная значимость**

Теоретическая значимость работы состоит в том, что все методы и алгоритмы, а так же их комбинирование, будут тщательным образом описаны и предоставлены научному сообществу, в результате чего с их использованием могут быть проведены новые исследования. В открытом доступе находится минимальное количество информации по данному вопросу, что обуславливает ценность данного исследования.

Практическая значимость работы заключается в том, что в результате должно быть предложено такое комбинирование методов распознавания жестов, которое позволит улучшить автоматизацию технологии управления жестами интерфейсом интерактивных систем. Влияющей на такие критерии как скорость отклика системы на жест, распознавание более мелких и сложных жестов, а так же восприятия большего количества людей в области обхвата камеры глубины.

Раздел 1

Раздел 2

Раздел 3 Решение задачи (Описать предложенный метод/технологию и т.д. при помощи которой будет достигнута поставленная цель. Если для решения задачи/метода используются программные продукты, то их так же необходимо описать)

**ГЛАВА 1. ТЕХНОЛОГИИ СОЗДАНИЯ ДИНАМИЧЕСКИХ ВЕБ-ПРИЛОЖЕНИЙ**

* 1. **Язык гипертекстовой разметки *HTML,* таблицы стилей и препроцессоры**

По существу, любая современная страница в веб состоит из *html* страницы, набора стилей *css*, а также динамической составляющей *JavaScript*. Прежде всего разберемся с языком гипертекстовой разметки.

Основой любого приложения является *html* (*Hyper Text Markup Language*) [1] – стандартизированный язык разметки. С момента своего появления данная технология не переживала революционных изменений. С течением времени одни элементы(теги) появлялись другие исчезали, в связи с неэффективностью. Так расширение возможностей *CSS* привело к устареванию тега *font*, необходимого для стилизации текста. На момент написания этой работы пятый стандарт являлся основным, также велась разработка шестого стандарта.

Основным нововведением пятого стандарта являлись семантические контейнеры. Благодаря им структуру веб документа можно разбить на логические единицы по их назначению [2]. Данная возможность является полезной для различных технологий. Так поисковик может определить при анализе документа важные части и второстепенные, это так же касается устройств для людей с ограниченными возможностями и не только. Кроме того, были расширены возможности создания элементов управления.

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 [14]. Язык определяет внешний вид документа (шрифты, цвет текста и фона, отступы), а также позволяет создавать разметку [15, 16].

Уровень 1 (*CSS1*) [17]. 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*) [18]. 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*) [14]. Спецификация принята в 2011 году и в основном исправляла ошибки прошлой.

Уровень 3 (*CSS3*) [19]. 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 [20].

Одним из недостатков является недостаточный уровень обструкции, отсутствие механизмов наследования и вложенности. Для устранения этих недостатков используются препроцессоры, такие как SASS [21] и LESS [22]. Главным преимуществом данных технологий является возможность повторного использования кода за счет модульной структуры и примесей, однако 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 упрощения блочной верстки предложены две технологии *Flexbox* [23] и *Grid* [24]. На момент написания работы технологии не получили широкого распространения (За исключением последних версий браузеров лидирующих разработчиков). Ключевым преимущество данных технологий является, возможность освободить зависимость стилизации от разметки документа.

С помощью Flexbox 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 [23].

*Grid* представляет документ в виде сетки, благодаря чему можно легко манипулировать основными частями приложения.

Однако основой современного веб приложения является *JavaScript*. Данный язык программирования представляет собой мультипарадигменный язык программирования [25], 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 [25]. В своих возможностях язык направлен на реализацию стандарта *ECMA*-262.

*JavaScript* предоставляет возможности в создании объектно-ориентированных клиентских приложений, однако в связи с реализацией других парадигм программирования, объектная модель имеет целый ряд особенностей по сравнению с другими языками программирования. Разнообразие предоставляемых средств предоставляемых средств делает язык гибким и позволяет осуществить быстрый переход с других языков программирования. Так же следует отметить что стандарт постоянно расширяют добавлением новых возможностей. Можно сказать, что данный язык постоянно совершенствуется и впитывает наиболее удачные идеи [26 – 28].

По своему синтаксису язык схож с Си: все идентификаторы регистрозависимы, в названиях переменных можно использовать буквы, подчёркивание, символ доллара, арабские цифры, названия переменных не могут начинаться с цифры, для оформления однострочных комментариев используются //, многострочные и внутристрочные комментарии начинаются с /\* и заканчиваются \*/.

Однако имеет ряд особенностей [25]: объекты с возможностью интроспекции, функции как объекты первого класса, автоматическое приведение типов, автоматическая сборка мусора, анонимные функции.

Структуру языка можно представить в виде трех частей [26, 29]:

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.

Объектная модель браузера ­представляет собой прослойку между ядром и объектной моделью документа [30]. При этом основная задача заключается в управлении окнами браузера и обеспечении их взаимодействия. Каждое из окон браузера представляется объектом *window*, центральным объектом *DOM* [31].

Помимо управления окнами *BOM* обычно обеспечивает следующие сущности [30]: управление фреймами, поддержка задержки в исполнении кода и зацикливания 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 HTTP cookie.

Объектная модель документа представляет собой интерфейс программирования для работы с XML и HTML документами [31]. В общем виде: генерация и добавление узлов, получение узлов, изменение узлов, изменение связей между узлами, удаление узлов.

В настоящее время *JavaScript* широко применяется при реализации:

* веб-приложений (клиентской части) – клиент-серверных программ, в котором клиентом является браузер, а сервером – веб-сервер, имеющих распределённую между сервером и клиентом логику [33];
* *AJAX* – популярном подходе к построению интерактивных пользовательских интерфейсов веб-приложений, заключающемся в «фоновом» асинхронном обмене данными браузера с веб-сервером [34, 35];
* *сomet* – широкое понятие, описывающее механизм работы веб-приложений, использующих постоянные *HTTP*-соединения, что позволяет веб-серверу отправлять данные браузеру без дополнительного запроса со стороны браузера [36];
* браузерные операционные системы – исходный код некоторых операционных систем более чем на половину состоит из *JS* [25];
* бурмарклеты – используется для создания небольших программ, размещаемых в закладки браузера [25];
* пользовательские скрипты в браузере – это программы, написанные на JavaScript, выполняемые в браузере пользователя при загрузке страницы и позволяющие автоматизировать часть работ, связанных с управлением пользовательским интерфейсом [25];
* серверные приложения – *JavaScript* может исполняться на серверах, использующих *Java* с 6 версии [38]. Наиболее популярные платформы: *Jaxer*, per*severe-framework*, *Helma*, *v8cgi*, *node.js* [39 – 41];
* виджеты – вспомогательная мини-программа для добавления дополнительных функций к пользовательскому интерфейсу;
* и др.

Однако не смотря на свою универсальность и динамику развития *JavaScript* имеет ряд недостатков. Во первых отсутствие строгой типизации приводит к неявным ошибкам из-за недочетов программиста. С целью исправления данной ситуации появился *TypeScript* [42, 43]. Данная технология вводит типизацию, позволяет определить ошибки на этапе компилирования (требует предварительной компиляции в *JS*), а также добавляет многие инструменты характерные для объектно-ориентированных технологий.

Следующий недостаток заключается в том, что браузеры не успевают за изменениями, и новые стандарты языка реализуются с запозданием, а в более старых версия браузерах вовсе поддержка отсутствует. С целью решения данной проблемы можно использовать поллифилы, библиотеки, добавляющие в старые браузеры встроенные возможности новых [44]. Либо использовать *JS­­*-фреймворки и библиотеки, такие как jQuery, Ext.js, Dojo и др., создающие прослойку, которая берет обеспечение однотипности работы на себя.

# 3. Компонентный подход во *front-end*

Основное направление в развитии компонентного подхода в разработке *front-end* части веб приложения является организация приложения в виде взаимодействующих компонент (модулей). Обычно модуль включает в себя *html*-каркас, базовые таблицы стилей и скрипты, обеспечивающие взаимодействие. Далее приведен обзор некоторых фреймворков и библиотек.

Одним из широко используемых *JS* фрейморков является *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. 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 этот фреймворк в качестве расширения (*Joomla* 3). Наличие большого числа сторонних библиотек позволяют в короткие сроки создать полнофункциональный сайт средней сложности [45].

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 [46]. Однако новые функциональные возможности *JavaScript* сводят к нулю все преимущества данной библиотеки.

Тем не менее библиотека предоставляет следующие возможности [47 - 49]: движок кросс-браузерных *CSS*-селекторов *Sizzle*, выделившийся в отдельный проект, 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 [50] так и сторонние компоненты.

Так же на основе jQuery формировались новые технологии в том числе и полноценные 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 [51]. Начало фреймворка положено 19 августа 2011 года [52], 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 [53].

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* [51]: сетки, шаблоны, типографика, медиа, таблицы, формы, навигация, алерты и др.

Однако наиболее серьёзную реализацию компонентного подхода можно видеть в фреймворке *Ext JS*. Данная технология 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.

Достоинства [54]:

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 заключены в ограниченных возможностях стилевого оформления приложения, а также в больших размерах самой библиотеки, что делает её не эффективной при создании малых проектов. Так же не мало важным недостатком является лицензия, она позволяет использовать библиотеку только на не коммерческих проектах.

Подытожив можно сказать что на данный момент существует большое разнообразие технологий так или иначе использующих компонентный подход в создании *front-end* части приложения таких как *jQuery*, *Ext.js*, *Bootstrap*, *Dojo* и другие. Не смотря на то что многие из этих решений устарели, они занимают свою долю рынку разработки как раз из-за наличия огромного количества библиотек компонентов.

# 4. Компонентная разработка с использованием *react.js*

Одной из любопытных технологий в *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 [55, 56]. Основным преимуществом данной библиотеки являлось возможность создания веб-интерфейсов быстро, с использованием читаемого и понятного кода. Одной из ключевых особенностей языка является возможность написания динамического создания *html*-узлов в приближенном к *HTML* виде с использованием так называемого *JSX* (*JavaScript Syntax eXtension*), расширения стандартного *JS* для динамического создания элементов. На Рисунке 4.1 приведен пример синтаксиса c использованием *JSX*, а на рисунке 4.2 без [57]. Как видно из представленных скриптов *JSX* делает код более понятным и читаемым для разработчика, однако требуется предварительная компиляция перед использованием.

const Com = props => {

return(

<form style="width: 100%">

<input defaultValue="Name"/>

<input defaultValue="E-mail"/>

</form> );};

Рисунок 4.1 – Синтаксис описания простого компонента с использованием *JSX*

"use strict";

var Com = function Com(props) {

return React.createElement("form",

{ style: "width: 100%" },

React.createElement("input", { defaultValue: "Name" }),

React.createElement("input", { defaultValue: "E-mail" }));};

Рисунок 4.2 – Синтаксис описания простого компонента без использованием *JSX*

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* [55, 56]. При этом перестройка происходит при изменение данных *state* (внутреннее состояние компонента) и *props* (внешнее состояние). Вовремя своей работы компоненты проходит через ряд этапов жизненного цикла. На каждом этапе вызывается определенная функция, переопределив которую можно задать собственное поведение. На Рисунке 4.3 приведена схема жизненного цикла *React* компонента [58].



Рисунок 4.3 – Жизненный цикл *React* компонента

Далее приведено описание функций жизненного цикла [58]:

*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*;

*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 рендорингом (­*true* если компонент нужно обновить). Таким образом можно запретить рендаринг 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* [59] – предоставляет функционал для роутинга приложения. Роутер позволяет определить содержание одностраничного приложения в зависимости от заданного адреса.

*React Motion* [60] – библиотека добавляет анимацию в реакт. Производит интерполяцию значений, используемых в *CSS* для трансформации элементов документа.

*React Addons* [61] – представляет собой пакет опционных функций, которые можно использовать для улучшения производительности приложений реакт.

*Enzyme* [62] – предоставляет функционал для написания автоматизированных тестов для *react* компонент.

*Redux* [63] – библиотека реализующая *flux*-паттерн для работы с данными.

Так же существует большой набор компонентов доступных для скачивания и использования в системе *npm*.

**ГЛАВА 2. ...**

**2.1 ...**

**2... Краткие выводы**

**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 разработке *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 компонентный подход разработки клиентской части веб-приложений;
2. Для разработки стилей выбран CSS препроцессор LESS, позволяющий использовать возможности программирования для разработки таблиц стилей;
3. Так как приложение одностраничное, приложение должно эффективно переключаться между окнами. Для этого был выбран модуль, построенный на библиотеке react.js, react-router;
4. Для оптимизации работы, и обмена данными с серверной частью выбрана реализация flax паттерна проектирования redux.

**3.1 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 (Рис. 3.1). В разработанном программном продукте присутствуют следующие компоненты такого типа:

* *HomePage* – компонент генерирует содержимое первоначальной страницы приложения;

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 5.1 продемонстрирован внешний вид веб-приложения.

**3.3 Структура разработанного программного продукта**

**3.2 Управление содержимым программного продукта**

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



Рисунок 5.1 – Пользовательский интерфейс веб-приложения

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

* стадия проекта, говорит о том какой этап разработки веб-приложения покрывает тестирование (возможны четыре состояния: *int*, *qa*, *production* и *staging*),
* название проекта,
* *url* адрес проекта,
* статус (хранит статус последнего тестирования),
* версия,
* дата последнего тестирования,
* результаты последнего тестирования,
* длительность тестирования.

После получения данных формируется пользовательский интерфейс состоящий из элементов соответствующих одному тестируемому проекту и его версиям. На Рисунке 5.2 показано интерфейс для отдельного элемента.



Рисунок 5.2 – Интерфейс отдельного элемента

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

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

**Компоненты**

**Роутинг  
Оптимизация с использование redux**

**3.4 Краткие выводы**

**ЗАКЛЮЧЕНИЕ**

По результатам проведенных исследований можно сделать вывод о широком распространении *JavaScript* в различных областях разработки *front-end* и *back-end* частей современных веб-приложений. Развитие стандарта *ECMOScript* позволяет в значительной степени улучшить процесс разработки, а так же использовать современные подходы к написанию кода различной сложности. Так стандарт *ES6* не только улучшил процесс работы с дом моделью, но и ввел синтаксические новшества позволяющие использовать объектно-ориентированный подход в более привычном для back-end разработке виде. Что не мало важно в развитии технологий back-end и мобильной разработки.

Так же следует отметить что из исследованя

Так же JavaScript нашол свою нисшу для разработки мобильныйх приложений (*React Native*, *JQuery mobile* и др.). Постоянное развитие и совершенствование языка позволяет не только решить проблемы вызванные несогласованностью разработчиков браузеров, но и в значительной степени расширить его возможности. Разработчики стандартов *ECMOScript* стараются учитывать при расширении функциональных возможностей языка тенденции в развитии современного программирования. Это позволяет использовать преимущества нескольких подходов и опустить недостатки, а также обеспечивает легкий переход программистов с других языков.

Развитие современных фреймворков и библиотек позволяет в значительной степени ускорить разработку за счет использования готовых решений типовых задач. Во многом это достигается в использовании готовых компонент, с заложенным для выполнения конкретной задачи функционалом.

Однако при этом высокие темпы развития приводят к высоким темпам устаревания решений. Так такие технологии как *jQuery*, *Bootstrap*, *ExtJS* на данный момент морально устарели, хотя и имеют широкую сферу применения за счет обширных библиотек компонентов и использования в связки с другими технологиями. Таким образом использование компонентного подхода обеспечивает существование данных технологий и их использование для решения узкого спектра задач.

Однако наиболее широкого применения данный подход достиг в библиотеке *react.js*. Концепция данной технологии с одной стороны не нова, с другой очень проста в использовании. На момент своего появления технологии получила достаточно противоречивые отзывы, однако она представляет собой будущее *front-end* разработки. Неожиданным результатом оказалось удобство подхода для использования в мобильной разработке.

**СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ**

1. Википедия / HTML [Электронный ресурс]. – 2017. – Режим доступа: https://ru.wikipedia.org/wiki/HTML (дата обращения: 23.12.2017).
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]. – 2017. – Режим доступа: https://itvdn.com/ru/blog/article/looking-into-html6 (дата обращения: 24.12.2017).
4. Helix / HTML6 – когда необходима семантика [Электронный ресурс]. – 2017. – Режим доступа: http://helix.su/html6-kogda-neobhodima-semantika.html (дата обращения: 24.12.2017).

**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>

<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 propsTypes = {

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 } envirment 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>

{ 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="horisontal"><a href="/">Home</a></li>

{

(name)

? <li className="horisontal">{ 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;