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 |
| Глава 1. Технологии создания динамических веб-приложений............... | 7 |
| 1.1 Язык гипертекстовой разметки HTML, таблицы стилей и пре-процессоры............................................................................................................... | 7 |
| 1.2 *JavaScript* особенности и сферы применения....................................... | 9 |
| 1.3 Расширение возможностей JS с использованием фреймворков и библиотек........................................................................................................................ | 12 |
| Глава 2. Компонентный подход программирования во *front-end*............. | 14 |
| 2.1 Обзор компонентного подхода программирования…………………. | 14 |
| 2.2 Компонентная разработка с использованием *react.js*……………….. | 15 |
| 2.3 Оптимизация процесса работы с данными, flux архитектура............. | 18 |
| Глава 3. Разработка *front-*end приложения управления автоматизированным тестированием проектов................................................... | 21 |
| 3.1 Постановка задачи.................................................................................. | 21 |
| 3.2 Обзор структуры разработанного приложения и компонент............. | 21 |
| 3.3 Обзор разработанного программного продукта.................................. | 23 |
| Заключение………………………………………………………………… | 26 |
| Список использованной литературы……………………………………... | 27 |
| Приложение А. Исходный код……………………………………………. | 33 |

**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 основной тенденцией в разработке *front-end* приложений заключается в создании элементов программы в виде повторно используемых модулей или компонент. Что позволяет значительно сократить время затрачиваемое на разработку приложения.

**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:

**Цель НИР:** Разработка пользовательского интерфейса с использованием компонентно-ориентированного подхода и библиотеки *react.js*.

**Постановка задачи:** Для достижение поставленной цели необходимо решить следующие задачи:

* проанализировать подходы к разработке клиентской части веб-приложения на основе основополагающих фреймворков и библиотек;
* проанализировать возможностей использования современных технологий и развитие компонентно-ориентированного подхода на примере библиотеки react.js;
* разработать простое клиентское приложение для выявления преимуществ использования компонентно-ориентированного подхода и библиотеки react.js.

**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 основа исследования.** Виктор Кулямин, И. С. Блинов, В. С. Романчик, *Herbert Schildt*, Дж. Рихтер, *A. Goldstein*, *L. Lazaris*, *E. Weyl*, Ден Сидерхолм, Д. Макфарланд, *Cody Lindley*, Marijn Haverbeke, *Addy Osmani*, *N. Zakas*, *P. Koch*, Крейн Д., Паскарелло Э., *Джеймс Д*., *Manuel Kiessling*, *T. Holowaychuk*, *Alex Young*, *R. Jansen*, *R. Murphey*, Б. Бибо, И. Кац, Райан Бенедетти, Ронан Крэнли, *Bear Bibeault*, *Aurelio De Rosa*, *Jesus Garcia*, *S. Stefanov*, *А. Banks*, *Е. Porcello* и др.

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

**Информационная база исследования.** В качестве информационной базы исследования использованы инструкции разработчиков.

**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*.

По теме НИР опубликована статьи:

Кузменков А. Причины возникновения и развития JS-фреймворков, преимущества, достоинства и недостатки / А. Кузменков // Научный обозреватель. – 2017 – №. 6. – С. 49 – 52.

Кузменков А. Компонентный подход разработки приложений во front-end (в печати).

**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 [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].

Одним из недостатков является недостаточный уровень абстракции, отсутствие механизмов наследования и вложенности. Для устранения этих недостатков 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 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* [14] и *Grid* [15]. На момент написания данной работы данные технологии широко внедрялись, однако так как ограничения поддержки вынуждали разработчиков использовать две версии стилей.

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 [14]. На Рисунке 1.1 показана ориентация элементов. На данный момент технология поддерживается всеми топовыми версиями браузеров.



Рисунок 1.1 – Ориентация элементов *flexbox*

*Grid* представляет документ в виде сетки, благодаря чему можно легко манипулировать основными частями приложения. Основным преимуществом данной технологии является отсутствие зависимости отображения элементов вне зависимости от их порядка, что позволяет более эффективно разрабатывать интерфейсы для разных типов устройств. На данный момент ещё имеет недостаточную поддержку даже новыми браузерами.

**1.2 *JavaScript*, особенности и сферы применения**

Однако основой современного веб приложения является *JavaScript*. Данный язык программирования представляет собой мультипарадигменный язык программирования [16], 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 [16].

*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 [17 – 19].

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.

Структуру языка можно представить в виде трех частей [17, 20]:

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 [21]. При этом основная задача заключается в управлении окнами браузера и обеспечении их взаимодействия. Каждое из окон браузера представляется объектом *window*, центральным объектом *DOM* [22].

Помимо управления окнами *BOM* обычно обеспечивает следующие сущности [21]: 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 [23]. 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 [24];
* *AJAX* – популярном подходе к построению интерактивных пользовательских интерфейсов веб-приложений, заключающемся в «фоновом» асинхронном обмене данными браузера с веб-сервером [25, 26];
* *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 [27];
* браузерные операционные системы – исходный код некоторых операционных систем более чем на половину состоит из *JS* [16];
* бурмарклеты – используется для создания небольших программ, размещаемых в закладки браузера [16];
* 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 [16];
* серверные приложения – *JavaScript* может исполняться на серверах, использующих *Java* с 6 версии [28]. Наиболее популярные платформы: *Jaxer*, per*severe-framework*, *Helma*, *v8cgi*, *node.js* [29 – 31];
* 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* [32, 33]. 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.

**1.3 Расширение возможностей *JS* с использованием фреймворков и библиотек**

Не смотря на все преимущества использования *JS* для разработки динамической составляющей веб-приложений, разработка занимает слишком много времени и является не эффективной. Для увеличения эффективности существует большое число фреймворков и библиотек.

Одной из широко используемых *JS* библиотек 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 [34].

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

Тем не менее библиотека предоставляет следующие возможности [36 - 38]: 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 [39] 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 [40]. Начало фреймворка положено 19 августа 2011 года [41], 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 [42].

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

**ГЛАВА 2. КОМПОНЕНТНЫЙ ПОДХОД ПРОГРАММИРОВАНИЯ ВО FRONT-END**

**2.1 Обзор компонентного подхода программирования**

Прежде всего необходимо определить, что подразумевается под компонентным подходом программирования. Прежде всего это появившаяся в 1987 году [44] парадигма компонентно-ориентированного программирования основанная на понятии компонента.

Компонент [45] — 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 Никлаусам 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 [46].

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 [47-49]:

*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.

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

В рамках платформы .*NET* [50]компонентно-ориентированный поход реализуется в виде *COM(Component Object Model)*. *COM* [8] 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 [51]: *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 компонент. В дальнейшем будут рассматриваться технологии *front-end* разработки.

**2.2 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 [52, 53]. 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.

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* [52, 53]. 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 2.1 приведена схема жизненного цикла *React* компонента [54].



Рисунок 2.1 – Жизненный цикл *React* компонента

Далее приведено описание функций жизненного цикла [55]:

*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* [56] – предоставляет функционал для роутинга приложения. Роутер позволяет определить содержание одностраничного приложения в зависимости от заданного адреса.

*React Motion* [57] – библиотека добавляет анимацию в реакт. Производит интерполяцию значений, используемых в *CSS* для трансформации элементов документа.

*React Addons* [58] – представляет собой пакет опционных функций, которые можно использовать для улучшения производительности приложений реакт.

*Enzyme* [59] – предоставляет функционал для написания автоматизированных тестов для *react* компонент.

*Redux* [60] – библиотека реализующая *flux*-паттерн для работы с данными.

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

**2.3 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 становиться организация обмена данными с сервером (*API*) и изменения состояния компонентов. Особенно остро становиться вопрос в случае одновременного использования данных в нескольких компонентах, при этом изменение данных внутри одного компонента должно повлечь за собою изменение состояния всех компонентов, использующих данный набор данных. Наиболее простым способом решения данной проблемы является организовать работу через общего предка Рисунок 2.2.

Компонент, взаимодействующий с сервером

Компонент, отображения данных

Компонент, отображения данных

Рисунок 2.2 – Схема работы через общего предка

Однако такой подход имеет существенный недостаток. При наличие большого числа связанных компонентов разбросанных в разных частях древа 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 становиться актуальным создание сущности, являющейся общим, для всех связанных компонент, хранилищем данных, а так же организовать систему обработки действий (*action*s) 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 [61, 62]:

*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 2.3 процесс взаимодействия сущностей представлен в виде диаграммы.

Store

Controller View

Dispatcher

Actions

API

Рисунок 2.3 – 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 рамка 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 [63, 64].

Таблица 2.1 – Статистические данные по использованию *Flux* архитектуры

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

**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.

**3.2 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;
* *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 отоброжение информации об проектах, навыках и др.;
* *Counter* – компонент выводит информацию об количестве зависимых сущносей;
* *Header, Footer* – подписи хедера и футера.

**Компонент**

**Результат**

**Данные**

Рисунок 3.1 – Схема работы простых компонентов

1. 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.

**Компонент**

**Результат**

**Данные**

**Внутренние**

**компоненты**

**Контент**

Рисунок 3.2 – Схема компонентов обёрток

1. Роутер. Данный компонент служит для контроля отображения содержимого в зависимости от содержимого адресной строки браузера. Данный функционал позволяет создать аллюзию работы с многостраничным приложением при этом работа ведется на одной странице.
2. Умные компоненты. Данные компоненты работают производят предварительную обработку данных изменения, способны динамически изменять свое состояние и осуществлять обмен данными с сервером.

**3.3 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 5.1 продемонстрирован внешний вид веб-приложения.

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*.

**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.

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* 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]. – 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-TRICS / 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. Википедия / JavaScript [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/JavaScript (дата обращения: 24.03.2018).
17. Lindley, C. JavaScript Enlightenment // Cody Lindley – O’Reilly Media, 2012. ­– 166 p.
18. Haverbeke, M. Eloquent JavaScript // Marijn Haverbeke – No Starch Press, 2011. ­– 224 p.
19. Osmani, A. Developing Backbone.js Application // Addy Osmani –O’Reilly Media, 2013. ­– 373 p.
20. Zakas N. 1. What is JavaScript? // Professional JavaScript for Web Developers. — 2nd ed. — USA, Canada: Wiley Publishing, Inc., 2009. — P. 3.
21. Koch P.-P. Chapter 6. BOM // ppk on JavaScript. — 1st ed. — New Riders Press, 2006. — 528 p.
22. MDN / Mozila Developer Network [Электронный ресурс]. – 2018. – Режим доступа: https://developer.mozilla.org/ru/ (дата обращения: 24.03.2018).
23. Zakas N. The Document Object Model // Professional JavaScript for Web Developers. — 2nd ed. — USA, Canada: Wiley Publishing, Inc., 2009. — P. 261 — 317.
24. Википедия / Веб-приложение [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Веб-приложение (дата обращения: 24.03.2018).
25. Крейн, Д. Ajax в действии // Крейн Д., Паскарелло Э., Джеймс Д. – Вильямс, 2006. ­– 640 с.
26. Википедия / AJAX [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/AJAX (дата обращения: 24.03.2018).
27. Википедия / Comet [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Comet\_(программирование) (дата обращения: 24.03.2018).
28. Википедия / WebOS [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/WebOS (дата обращения: 24.03.2018).
29. Oracle / webnotes [Электронный ресурс]. – 2018. – Режим доступа: http://www.oracle.com/technetwork/java/javase/webnotes-136672.html (дата обращения: 24.03.2018).
30. Kiessling, M. Node Beginner / Manuel Kiessling // The Node Beginner Book[Электронный ресурс]. − 2018. − Режим доступа: http://www.  
    nodebeginner.org/ (дата обращения: 24.03.2018).
31. 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).
32. Young, A. Building A JavaScript Framework / Alex Young // DailyJS - JavaScript Blog [Электронный ресурс]. − 2018. − Режим доступа: http://dailyjs.com (дата обращения: 24.03.2018).
33. TypeScript / Documentation [Электронный ресурс]. – 2018. – Режим доступа: https://www.typescriptlang.org/docs/home.html (дата обращения: 24.03.2018).
34. Jansen, R. TypeScript: Modern JavaScript Development // R. Jansen – Packt, 2012. ­– 1087 p. Murphey, R. jQuery Fundamentals / Rebecca Murphey // DailyJS - jQuery Fundamentals [Электронный ресурс]. − 2018. − Режим доступа: http://jqfundamentals.com (дата обращения: 24.03.2018).
35. Бибо, Б. jQuery. Подробное руководство по продвинутому JavaScript, 2-е издание // Б. Бибо, И. Кац – Спб.: Символ-плюс, 2011. ­– 623 с.
36. Бенедетти, Р. Изучаем работу с jQuery // Райан Бенедетти, Ронан Крэнли. – Спб.: Питер , 2012. ­– 508 с.
37. Bibeault, В. jQuery in Action, Third Edition // Bear Bibeault, Yehuda Katz and Aurelio De Rosa – Manning, 2015. ­– 504 с.
38. Бибо, Б. jQuery. Подробное руководство по продвинутому JavaScript // Бер Бибо, Иегуда Кац — Спб.: Символ-Плюс, 2009. — 384 с.
39. jQuery user interface / API Documentation [Электронный ресурс]. – 2017. – Режим доступа: http://api.jqueryui.com/ (дата обращения: 24.03.2018).
40. Википедия / Bootstrap (фреймворк) [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Bootstrap\_(фреймворк) (дата обращения: 24.03.2018).
41. blog.twitter.com / Bootstrap from Twitter [Электронный ресурс]. – 2018. – Режим доступа: https://blog.twitter.com/2011/bootstrap-from-twitter (дата обращения: 24.03.2018).
42. 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).
43. Stefanov, S. React: Up & Running // S. Stefanov. – O’Reilly Media, 2016. ­– 222 p.
44. 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).
45. Википедия / Компонентно-ориентированное программирование [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/  
    wiki/Компонентно-ориентированное программирование (дата обращения: 24.03.2018).
46. Кулямин, В. Технология программирования. Компонентный подход // Виктор Кулямин – М.: ИСП РАН, 2006. ­– 315 c.
47. Кулямин, В. Компонентный подход в программировании // Виктор Кулямин – М.: НОУ «Интуит», 2016. ­– 590 c.
48. Блинов, И. С. Методы программирования, второе издание // И. С. Блинов, В. С. Романчик – Минск: издательство «Четыре четверти», 2013. ­– 896 c.
49. Schildt, H. Java: The Complete Reference. Tenth Edition // Herbert Schildt – McGraw-Hill Education, 2017. ­– 1923 p.
50. 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.
51. 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 (дата обращения: 27.03.2018).
52. Википедия / Component Object Model [Электронный ресурс]­. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Component\_Object\_Model (дата обращения: 27.03.2018).
53. Banks, A. Learning React // А. Banks, Е. Porcello. – O’Reilly Media, 2017. ­– 350 p.
54. Babel / Try it out [Электронный ресурс]. – 2018. – Режим доступа: https://babeljs.io/repl/ (дата обращения: 24.03.2018).
55. Metanit / Жизненный цикл компонента [Электронный ресурс]. – 2018. – Режим доступа: https://metanit.com/web/react/2.6.php (дата обращения: 24.03.2018).
56. React training / React router [Электронный ресурс]. – 2018. – Режим доступа: https://reacttraining.com/react-router/ (дата обращения: 24.03.2018).
57. GitHub / React motion [Электронный ресурс]. – 2018. – Режим доступа: https://github.com/chenglou/react-motion (дата обращения: 24.03.2018).
58. React / Add-Ons [Электронный ресурс]. – 2018. – Режим доступа: https://reactjs.org/docs/addons.html (дата обращения: 24.03.2018).
59. Airbnb / Enzyme [Электронный ресурс]. – 2018. – Режим доступа: http://airbnb.io/enzyme/ (дата обращения: 24.03.2018).
60. Redux / Read Me [Электронный ресурс]. – 2018. – Режим доступа: https://redux.js.org/ (дата обращения: 24.03.2018).
61. Boduch, A. Flux Architecture // А. Boduch. – Packt Publishing, 2017.   
    ­– 316 p.
62. Хабрахабр / Разбираемся с Flux, реактивной архитектурой от facebook [Электронный ресурс]. – 2018. – Режим доступа: https://habrahabr.ru/post/246959/ (дата обращения: 27.03.2018).
63. GitHub / Flux [Электронный ресурс]. – 2018. – Режим доступа: https://github.com/facebook/flux?ref=stackshare (дата обращения: 27.03.2018).
64. GitHub / Redux [Электронный ресурс]. – 2018. – Режим доступа: https://github.com/reactjs/redux?ref=stackshare (дата обращения: 27.03.2018).

**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>

<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;