**ОГЛАВЛЕНИЕ**

|  |  |
| --- | --- |
| Глава 1. Обзор современных технологий создания клиентской части веб приложений | 1 |
| 1.1 Язык гипертекстовой разметки, таблицы стилей и препроцессоры | 2 |
| 1.2 JavaScript как язык разработки современных веб приложений |  |

**ВВЕДЕНИЕ**

bla

**ГЛАВА 1. ОБЗОР СОВРЕМЕННЫХ ТЕХНОЛОГИЙ РАЗРАБОТКИ КЛИЕНТСКОЙ ЧАСТИ ВЕБ ПРИЛОЖЕНИЙ**

**1.1 Язык гипертекстовой разметки*,* таблицы   
стилей и препроцессоры**

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

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

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

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

Итак, в том или ином виде основой любого веб приложения является *HTML*-документ. Именно на его основе браузер строит *DOM*-модель документа прежде чем отобразить результат на пользовательском мониторе. Однако данная технология является не столько языком, сколько стандартом, необходимым для стандартизации, разрабатываемых продуктов. Как и любой иной стандарт *HTML* пережил большое число изменений, связанных с появлением новых технологий, подходов и методик разработки. Так, если первоначально для создания документа для веб было достаточно одного языка гипертекстовой разметки, современное приложение уже строится на трех основополагающих элементах, взаимодействие которых и создает клиентское приложение в современном виде. Так, появление технологии каскадных таблиц стилей и постепенное расширение возможностей *CSS*, привело к устареванию большого числа первоначально используемых тегов таких как center, font и другие. На момент написания этой работы повсеместно используемым стандартом являлся *HTML5*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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



Рисунок 1.1 – Ориентация элементов *flexbox*

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

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

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

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

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

**1.2 JavaScript как язык разработки современных веб приложений**

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

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

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

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

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

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

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

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

Таблица 1.2 – Выход версий стандарта *ECMOScript.*

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

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

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

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

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

*ES7* продолжает реформу кода в направлении создания изолированных модулей и библиотек;

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

Возвращаясь к браузерному *JS* следует разобраться что из себя представляют *DOM* и *BOM*. Итак объектная модель браузера ­представляет собой прослойку между ядром и объектной моделью документа [27]. При этом основная задача заключается в управлении окнами браузера и обеспечении их взаимодействия. Каждое из окон браузера представляется объектом *window*, центральным объектом *DOM* [28, 29]. *BOM* жеобычно обеспечивает возможность управления сущностями браузера такими как [27]: 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 куками.

В настоящее время *JavaScript* широко применяется при реализации:

* веб-приложений (клиентской части) – клиент серверные приложения с распределенной бизнес логикой между клиентской частью (браузером) и сервером [30];
* *AJAX*-запросы – подход организации взаимодействия между сервером и клиентом заключающийся в асинхронном обмене данными, при этом пользователь, при первом обращении, получает клиентское приложение, которое взаимодействует с сервером путем обмена данными в формате *JSON* [31, 32];
* *COMET* – является широким понятием, описывающим механизмы работы веб приложений на основе постоянных соединений, аналогичных обычным настольным приложениям [33]. Наиболее известной реализацией можно назвать *web-sockets*;
* браузерные операционные системы – исходный код некоторых операционных систем более чем на половину состоит из *JS* [18] (*iCloud*, *Glide OS*, *eyeOS*, *myGoya Online Desktop*, *Online Operating System* и др.);
* бурмарклеты – используется для создания небольших программ, размещаемых в закладки браузера [18];
* пользовательские скрипты в браузере – небольшие пользовательские скрипты, обеспечивающие создание определенной динамики на стороне клиента. Например: валидация в формах, слайтеры, блоки рекламы и прочее [18];
* серверные приложения – развитие и популярность языка привели к появлению серверных реализаций, позволяющих вести полноценную разработку серверной части приложения на чистом *JavaScript*. Так же *JS* может исполняться на серверах, использующих *Java* с 6 версии [34]. Наиболее популярные платформы: *Jaxer*, per*severe-framework*, *Helma*, *v8cgi*, *node.js* [35 – 37];
* и др.

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

Таблица 1.3 – Обзор основных JS-движков.

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

Продолжение таблицы 1.3.

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

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

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

Для решения вышесказанных проблем используются средства позволяющие компилировать *JavaScript* в определенную версию стандарта.

С целью исправления данной ситуации появился *TypeScript* [53, 54]. 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.

**! добавить какую нибудь статистику по языкам**

**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* [43]: 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.

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

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

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

1. *Material-UI* [12]: Данный фреймворк содержит широкий набор компонент и инструментов для разработки пользовательских интерфейсов с использованием материал дизайн. Преимущества использования данной технологии является:широкий набор готовых к использованию компонент;гибкая система настройки стилей, позволяющая стилизовать приложения путём модифицирования одного файла настройки стилей; наличие инструментов создания кроссплатформенных приложений. Ключевым недостатком данной технологии является отсутствие должного обеспечения обратной совместимости стабильных версий. Так приложение, при необходимости перехода на новую версию фреймворка должно быть практически полностью переписано.
2. *React Desktop* [13]: Данный фреймворк переназначен для разработки интерфейсов для настольных приложений с использованием технологии *Electron*. Содержит кроссплатформенные компоненты для операционных систем *Windows* 10 и *Mac OS*.
3. *Semantic-UI-react*[14], *Bootstrap-react*[15]: Данные технологии являются клонами одноименных фреймворков, содержат набор инструментов разработки идентичный родительскому, однако используют внутри себя устаревшую библиотеку *jQuery*, что является ключевым недостатком данных технологий.

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

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

**3.2 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**

bla

**СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ**

*AJAX* (*Asynchronous Javascript and XML*) – Асинхронный *JavaScript* и *XML*.

*BOM* (*Browser Object Model*)– Объектная модель браузера.

*CSS* (*Cascading Style Sheets*)– Каскадные таблицы стилей.

*CSS Flexbox* (*Flexible Box Layout Module*) – модуль макета гибкого контейнера.

*DOM* (*Document Object Model*) – Документная объектная модель.

*HTML* (*Hyper Text Markup Language*) – Гипертекстовый язык разметки.

*JS* (*JavaScript*)

*JSON* (*JavaScript Object Notation*) – Описание *JavaScript* объектов.

*LESS* (*Leaner Style Sheets*) – Линейные таблицы стилей.

*SASS* (*Syntactically Awesome Stylesheets*) – Превосходящие статические таблицы стилей.

**СПИСОК ТЕРМИНОВ**

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

*Back-end* – программно-аппаратной части сервиса.

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

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

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

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

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

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

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

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

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

*JSON* – текстовый формат обмена данными, основанный на JavaScript.

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

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

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

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

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

**СПИСОК ИСПОЛЬЗОВАННОЙ ИСТОЧНИКОВ**

1. Википедия / HTML [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/HTML (дата обращения: 24.03.2018).
2. Goldstein, A. HTML5 & CSS3 For The Real World, 2nd Edition // A. Goldstein, L. Lazaris, E. Weyl – SitePoint, 2015. ­– 350 p.
3. ITVDN / Взгляд в HTML6 [Электронный ресурс]. – 2018. – Режим доступа: https://itvdn.com/ru/blog/article/looking-into-html6 (дата обращения: 24.03.2018).
4. Helix / HTML6 – когда необходима семантика [Электронный ресурс]. – 2017. – Режим доступа: http://helix.su/html6-kogda-neobhodima-semantika.html (дата обращения: 24.03.2018).
5. Википедия / CSS [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/CSS (дата обращения: 24.03.2018).
6. Сидерхолм, Д. CSS3 для веб-дизайнеров // Ден Сидерхолм – Манн, Иванов и Фербер, 2012. ­– 125 с.
7. Макфарланд, Д. Большая книга CSS3 // Д. Макфарланд – Питер, 2014. ­– 608 с.
8. W3C / Cascading Style Sheets, level 1 [Электронный ресурс]. – 2018. – Режим доступа: https://www.w3.org/TR/CSS1/ (дата обращения: 24.03.2018).
9. W3C / Cascading Style Sheets, level 2 CSS Specification [Электронный ресурс]. – 2017. – Режим доступа: https://www.w3.org/TR/2008/REC-CSS2-20080411/ (дата обращения: 24.03.2018).
10. W3C / Текущая работа CSS и как принять участие [Электронный ресурс]. – 2018. – Режим доступа: https://www.w3.org/Style/CSS/current-work (дата обращения: 24.03.2018).
11. WIRED / Discover What’s New in CSS 4 [Электронный ресурс]. – 2018. – Режим доступа: https://www.wired.com/2011/10/discover-whats-new-in-css-4/ (дата обращения: 24.03.2018).
12. SASS / Docs [Электронный ресурс]. – 2018. – Режим доступа: http://sass-lang.com/documentation/file.SASS\_REFERENCE.html (дата обращения: 24.03.2018).
13. LESS / Docs [Электронный ресурс]. – 2018. – Режим доступа: https://less-lang.info/ (дата обращения: 24.03.2018).
14. Про CSS / Flexbox [Электронный ресурс]. – 2018. – Режим доступа: http://css.yoksel.ru/flexbox/ (дата обращения: 24.03.2018).
15. CSS-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. Can I Use / Flexbox [Электронный ресурс]. – 2018. – Режим доступа: https://caniuse.com/#search=flexbox (дата обращения: 12.04.2018).
17. Can I Use / Grid Layout [Электронный ресурс]. – 2018. – Режим доступа: https://caniuse.com/#search=Grid%20layout (дата обращения: 12.04.2018).
18. Википедия / JavaScript [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/JavaScript (дата обращения: 24.03.2018).
19. Lindley, C. JavaScript Enlightenment // Cody Lindley – O’Reilly Media, 2012. ­– 166 p.
20. Haverbeke, M. Eloquent JavaScript // Marijn Haverbeke – No Starch Press, 2011. ­– 224 p.
21. Osmani, A. Developing Backbone.js Application // Addy Osmani –O’Reilly Media, 2013. ­– 373 p.
22. Zakas N. 1. What is JavaScript? // Professional JavaScript for Web Developers. — 2nd ed. — USA, Canada: Wiley Publishing, Inc., 2009. — P. 3.
23. Википедия / ECMAScript [Электронный ресурс]. – 2018. – Режим доступа: https://en.wikipedia.org/wiki/ECMAScript#Versions (дата обращения: 18.04.2018).
24. ECMA international / ECMAScript 2015 Language Specification [Электронный ресурс]. – 2018. – Режим доступа: http://www.ecma-international.org/ecma-262/6.0/index.html (дата обращения: 18.04.2018).
25. ECMA international / ECMAScript 2016 Language Specification [Электронный ресурс]. – 2018. – Режим доступа: http://www.ecma-international.org/ecma-262/7.0/index.html(дата обращения: 18.04.2018).
26. ECMA international / ECMAScript 2017 Language Specification (ECMA-262, 8th edition, June 2017) [Электронный ресурс]. – 2018. – Режим доступа: http://www.ecma-international.org/ecma-262/8.0/index.html (дата обращения: 18.04.2018).
27. Koch P.-P. Chapter 6. BOM // ppk on JavaScript. — 1st ed. — New Riders Press, 2006. — 528 p.
28. MDN / Mozila Developer Network [Электронный ресурс]. – 2018. – Режим доступа: https://developer.mozilla.org/ru/ (дата обращения: 24.03.2018).
29. Zakas N. The Document Object Model // Professional JavaScript for Web Developers. — 2nd ed. — USA, Canada: Wiley Publishing, Inc., 2009. — P. 261 — 317.
30. Википедия / Веб-приложение [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Веб-приложение (дата обращения: 24.03.2018).
31. Крейн, Д. Ajax в действии // Крейн Д., Паскарелло Э., Джеймс Д. – Вильямс, 2006. ­– 640 с.
32. Википедия / AJAX [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/AJAX (дата обращения: 24.03.2018).
33. Википедия / Comet [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Comet\_(программирование) (дата обращения: 24.03.2018).
34. Википедия / WebOS [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/WebOS (дата обращения: 24.03.2018).
35. Oracle / webnotes [Электронный ресурс]. – 2018. – Режим доступа: http://www.oracle.com/technetwork/java/javase/webnotes-136672.html (дата обращения: 24.03.2018).
36. Kiessling, M. Node Beginner / Manuel Kiessling // The Node Beginner Book[Электронный ресурс]. − 2018. − Режим доступа: http://www.  
    nodebeginner.org/ (дата обращения: 24.03.2018).
37. Holowaychuk, T. Mastering NodeJS / TJ Holowaychuk // Mastering NodeJS Open Source Node eBook[Электронный ресурс]. − 2018. − Режим доступа: http://visionmedia.github.io/masteringnode/ (дата обращения: 24.03.2018).
38. MDN web docs / Rhino history [Электронный ресурс]. – 2018. – Режим доступа: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/History (дата обращения: 18.04.2018).
39. Mozilla wiki / Jaeger Monkey [Электронный ресурс]. – 2018. – Режим доступа: https://wiki.mozilla.org/JaegerMonkey (дата обращения: 18.04.2018).
40. Chrome V8 / Introduction [Электронный ресурс]. – 2018. – Режим доступа: https://developers.google.com/v8/?hl=ru (дата обращения: 18.04.2018).
41. WebKit / The WebKit Open Source Project [Электронный ресурс]. – 2018. – Режим доступа: https://webkit.org/project/ (дата обращения: 18.04.2018).
42. KDE / Release of KDE Frameworks 5.45.0 [Электронный ресурс]. – 2018. – Режим доступа: https://www.kde.org/announcements/kde-frameworks-5.45.0.php (дата обращения: 18.04.2018).
43. Microsoft Blogs / An Early Look At IE9 for Developers [Электронный ресурс]. – 2018. – Режим доступа: https://blogs.msdn.microsoft.com/ie/2009/11  
    /18/an-early-look-at-ie9-for-developers/ (дата обращения: 18.04.2018).
44. Microsoft Blogs / Microsoft Edge’s JavaScript engine to go open source [Электронный ресурс]. – 2018. – Режим доступа: https://blogs.windows.com  
    /msedgedev/2015/12/05/open-source-chakra-core/ (дата обращения: 18.04.2018).
45. OpenJDK / JDK8 [Электронный ресурс]. – 2018. – Режим доступа: http://openjdk.java.net/projects/jdk8/ (дата обращения: 18.04.2018).
46. Википедия / JavaScript engine [Электронный ресурс]. – 2018. – Режим доступа: https://en.wikipedia.org/wiki/JavaScript\_engine (дата обращения: 18.04.2018).
47. Young, A. Building A JavaScript Framework / Alex Young // DailyJS - JavaScript Blog [Электронный ресурс]. − 2018. − Режим доступа: http://dailyjs.com (дата обращения: 24.03.2018).
48. TypeScript / Documentation [Электронный ресурс]. – 2018. – Режим доступа: https://www.typescriptlang.org/docs/home.html (дата обращения: 24.03.2018).
49. 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).
50. Бибо, Б. jQuery. Подробное руководство по продвинутому JavaScript, 2-е издание // Б. Бибо, И. Кац – Спб.: Символ-плюс, 2011. ­– 623 с.
51. Бенедетти, Р. Изучаем работу с jQuery // Райан Бенедетти, Ронан Крэнли. – Спб.: Питер , 2012. ­– 508 с.
52. Bibeault, В. jQuery in Action, Third Edition // Bear Bibeault, Yehuda Katz and Aurelio De Rosa – Manning, 2015. ­– 504 с.
53. Бибо, Б. jQuery. Подробное руководство по продвинутому JavaScript // Бер Бибо, Иегуда Кац — Спб.: Символ-Плюс, 2009. — 384 с.
54. jQuery user interface / API Documentation [Электронный ресурс]. – 2017. – Режим доступа: http://api.jqueryui.com/ (дата обращения: 24.03.2018).
55. Википедия / Bootstrap (фреймворк) [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Bootstrap\_(фреймворк) (дата обращения: 24.03.2018).
56. blog.twitter.com / Bootstrap from Twitter [Электронный ресурс]. – 2018. – Режим доступа: https://blog.twitter.com/2011/bootstrap-from-twitter (дата обращения: 24.03.2018).
57. The Bootstrap Blog / Bootstrap 4 alpha [Электронный ресурс]. – 2018. – Режим доступа: http://blog.getbootstrap.com/2015/08/19/bootstrap-4-alpha/ (дата обращения: 24.03.2018).
58. Stefanov, S. React: Up & Running // S. Stefanov. – O’Reilly Media, 2016. ­– 222 p.
59. INTUIT / Лекция: Компонентное программирование в .Net ­– Введение в теорию программирования [Электронный ресурс]. – 2018. – Режим доступа: http://www.intuit.ru/department/se/tppobj/17/ (дата обращения: 24.03.2018).
60. Википедия / Компонентно-ориентированное программирование [Электронный ресурс]. – 2018. – Режим доступа: https://ru.wikipedia.org/  
    wiki/Компонентно-ориентированное программирование (дата обращения: 24.03.2018).
61. Кулямин, В. Технология программирования. Компонентный подход // Виктор Кулямин – М.: ИСП РАН, 2006. ­– 315 c.
62. Кулямин, В. Компонентный подход в программировании // Виктор Кулямин – М.: НОУ «Интуит», 2016. ­– 590 c.
63. Блинов, И. С. Методы программирования, второе издание // И. С. Блинов, В. С. Романчик – Минск: издательство «Четыре четверти», 2013. ­– 896 c.
64. Schildt, H. Java: The Complete Reference. Tenth Edition // Herbert Schildt – McGraw-Hill Education, 2017. ­– 1923 p.
65. 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.
66. 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).
67. Википедия / Component Object Model [Электронный ресурс]­. – 2018. – Режим доступа: https://ru.wikipedia.org/wiki/Component\_Object\_Model (дата обращения: 27.03.2018).
68. Banks, A. Learning React // А. Banks, Е. Porcello. – O’Reilly Media, 2017. ­– 350 p.
69. Babel / Try it out [Электронный ресурс]. – 2018. – Режим доступа: https://babeljs.io/repl/ (дата обращения: 24.03.2018).
70. Metanit / Жизненный цикл компонента [Электронный ресурс]. – 2018. – Режим доступа: https://metanit.com/web/react/2.6.php (дата обращения: 24.03.2018).
71. React training / React router [Электронный ресурс]. – 2018. – Режим доступа: https://reacttraining.com/react-router/ (дата обращения: 24.03.2018).
72. GitHub / React motion [Электронный ресурс]. – 2018. – Режим доступа: https://github.com/chenglou/react-motion (дата обращения: 24.03.2018).
73. React / Add-Ons [Электронный ресурс]. – 2018. – Режим доступа: https://reactjs.org/docs/addons.html (дата обращения: 24.03.2018).
74. Airbnb / Enzyme [Электронный ресурс]. – 2018. – Режим доступа: http://airbnb.io/enzyme/ (дата обращения: 24.03.2018).
75. Redux / Read Me [Электронный ресурс]. – 2018. – Режим доступа: https://redux.js.org/ (дата обращения: 24.03.2018).
76. Boduch, A. Flux Architecture // А. Boduch. – Packt Publishing, 2017.   
    ­– 316 p.
77. Хабрахабр / Разбираемся с Flux, реактивной архитектурой от facebook [Электронный ресурс]. – 2018. – Режим доступа: https://habrahabr.ru/post/246959/ (дата обращения: 27.03.2018).
78. GitHub / Flux [Электронный ресурс]. – 2018. – Режим доступа: https://github.com/facebook/flux?ref=stackshare (дата обращения: 27.03.2018).
79. GitHub / Redux [Электронный ресурс]. – 2018. – Режим доступа: https://github.com/reactjs/redux?ref=stackshare (дата обращения: 27.03.2018).
80. Material-UI / React components that implement Google’s Material Design [Электронный ресурс]. – 2018. – Режим доступа: https://material-ui-next.com (дата обращения: 02.04.2018).
81. React Desktop / Documentation. [Электронный ресурс]. – 2018. – Режим доступа: http://reactdesktop.js.org (дата обращения: 02.04.2018).
82. GitHub / Semantic-UI-React. [Электронный ресурс]. – 2018. – Режим доступа: https://github.com/Semantic-Org/Semantic-UI-React (дата обращения: 02.04.2018).
83. React Bootstrap / Documentation. [Электронный ресурс]. – 2018. – Режим доступа: https://react-bootstrap.github.io/getting-started/introduction (дата обращения: 02.04.2018).

**Приложение А. Исходный код некоторых компонент**

App.js:

import React, { Component } from 'react';

import { Router, Switch, Route } from 'react-router'

import createHistory from 'history/createBrowserHistory';

import Header from './components/Header';

import Footer from './components/Footer';

import HomePage from './pages/HomePage';

import AboutMePage from './pages/AboutMePage';

import ItemPage from './pages/ItemPage';

import ErrorPage from './pages/ErrorPage';

import LogoPage from './pages/LogoPage';

import TaskManagerPage from './pages/TaskManagerPage';

import \* as PropTypes from 'prop-types';

class App extends Component {

static PAGE\_LINKS = [

{ "link": "/task-manager", "title": "Task manager" },

{ "link": "/about", "title": "About me" }

];

getChildContext() {

return {

pageLinks: App.PAGE\_LINKS

}

}

render() {

return (

<div className="app">

<Header text="Front-end EPAM training" />

<Router history={ createHistory() }>

<Switch>

<Route exact path="/" component={ HomePage } />

<Route path="/about" component={ AboutMePage } />

<Route path="/task-manager" component={ TaskManagerPage } />

<Route path="/item/:name/:environment" component={ ItemPage } />

<Route path="/logo.txt" component={ LogoPage }/>

<Route path="/\*" component={ ErrorPage } />

</Switch>

</Router>

<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;