**ACADEMIA DE STUDII ECONOMICE A MOLDOVEI**

**FACULTATEA TEHNOLOGII INFORMAȚIONALE ȘI STATISTICĂ ECONOMICĂ**

**DEPARTAMENTUL TEHNOLOGIA INFORMAȚIEI ȘI MANAGEMENTUL INFORMAȚIONAL**

**Piotr PASENCO**

**SISTEM INFORMATIC PENTRU VĂNZARE HRANEI PENTRU CĂINI**

**TEZĂ DE LICENŢĂ**

**Specialitatea Cibernetică și Informatică Economică**

|  |
| --- |
| **Admis la susţinere**  Şef department “Tehnologia  Informației și management  informațional”:  dr., conf. univ.  **Anatolie PRISĂCARU**  ’\_\_’\_\_\_\_\_\_\_ 2022 |
|  |  |

**Autor:**student gr. CIB-151,

**Piotr Pasenсo**

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*(semnătura)*

**Conducător ştiinţific:**

Dr., Conf.univ. **Godonoagă Anatol**

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*(semnătura)*

**Chişinău – 2022****Declaraţia pe propria răspundere**

Subsemnatul (a), **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**absolvent a Facultăţii **Cibernetică, Statistică şi Informatică Economică** al Academiei de Studii Economice din Moldova, specialitatea **Cibernetică, Statistică si Informatică Economică declar pe propria răspundere că teza de licență pe tema \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**a fost elaborată de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau din străinătate, iar exemplarul prezentat şi înregistrat la catedră corespunde integral cu varianta electronică plasată în sistemul Anti-plagiat**.

De asemenea, declar că sursele utilizate în teză, inclusiv cele din Internet, sunt indicate cu respectarea regulilor de evitare a plagiatului:

* fragmentele de text sunt reproduse întocmai şi sunt scrise în ghilimele, deţinând referinţa precisă a sursei;
* redarea/reformularea în cuvinte proprii a textelor altor autori conţine referinţa precisă;
* rezumarea ideilor altor autori conţine referinţa precisă a originalului.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Semnătura

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 4](#_Toc103894032)

[ГЛАВА I. СОВРЕМЕННЫЙ ВЕБ ПРИЛОЖЕНИЯ 5](#_Toc103894033)

[**1.1** Одностраничные веб приложения 5](#_Toc103894034)

[**1.2** Архитектура REST API 7](#_Toc103894035)

[**1.3** Контейнеризация веб приложений 12](#_Toc103894036)

[**1.4** Выводы главы 15](#_Toc103894037)

[ГЛАВА II. ОСОБЕННОСТИ ОДНОСТРАНИЧНЫХ ПРИЛОЖЕНИЙ СТЕКА MEVN 17](#_Toc103894038)

[**2.1** Vue.Js и Vuex 20](#_Toc103894039)

[**2.2** Node.JS и Express.JS 32](#_Toc103894040)

[**2.3** MongoDB 34](#_Toc103894041)

[**2.4** Аутентификация в одностраничных приложения**х** 39](#_Toc103894042)

[**2.5** Выводы главы 45](#_Toc103894043)

[ГЛАВА III. РЕАЛИЗАЦИЯ ОДНОСТРАНИЧНОГО ВЕБ ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ MEVN 47](#_Toc103894044)

[**3.1** Проектирование базы данных MongoDB 47](#_Toc103894045)

[**3.2** Создание REST API сервера 50](#_Toc103894046)

[**3.3** Создание фронтенда с помощью Vue и Vuex 57](#_Toc103894047)

**3.4** Выводы главы……………………………………………………………………........62

[ВЫВОДЫ 67](#_Toc103894048)

[БИБЛИОГРАФИЯ 68](#_Toc103894049)

# ВВЕДЕНИЕ

Деятельность любой организации так или иначе связана с веб-технологиями. Широкое распространение получили веб-порталы различных услуг (в том числе государственных), интернет-магазины, торговые площадки, различные бизнес-приложения, системы дистанционного банковского обслуживания. Невозможно представить себе современную организацию, будь то крупная корпорация или небольшая частная фирма, у которой не было бы своего официального сайта или странички на каком-либо публичном веб-ресурсе.

Типичное веб приложение состоит как минимум из базы данных, веб серверов, операционной системы и среды разработки. Любая компания, разрабатывающая веб-приложения, должна обладать знаниями и ресурсами для реализаций этих функций для доступных на рынке платформ. Для того, чтобы создать функциональное веб приложение, стеки веб разработки предлагают полный набор инструментов и технологий в качестве базовой платформы.

Выбор правильного стека технологий для запуска веб приложения может оказаться сложной задачей, учитывая сложный и динамический характер современной разработки. Важно правильно выбрать технологии, так как от этого зависит успех приложения.

Рост бизнес-процессов, управляемых веб приложениями, привел к появлению различных инструментов, методов и механизмов реализации веб приложений для различных вариантов использования. В результате у каждой организации или разработчика есть свои предпочтения в отношении стека разработки, который решает его задачу.

С исчезновением классической модели, в последнее время были разработаны и внедрены различные стеки для решения распространённых сценариев веб приложений. Набор инструментов для веб разработки развивался более десяти лет. Мобильные устройства стали катализатором значительных изменений и помогли перейти к модульному, легкому и очень гибкому миру веб разработки. В последнее время цепочка инструментов развивается, и ключевая идея заключается в том, что при помощи новых инструментов мы создаем более лучшие приложения. Сложность приложений не уменьшится. Даже если написание кода станет легче, то понимание как части приложения сочетаются друг другом – от элементов frontend до интеграции API – будут иметь решающее значение. Даже если новые технологии будут ломать вам голову и создавать проблемы, о которых вы даже не задумывались, ясно, что в будущем это будет более интересно, с точки зрения веб разработки.

# ГЛАВА I. СОВРЕМЕННЫЙ ВЕБ ПРИЛОЖЕНИЯ

Современные веб приложения имеют более высокие ожидания пользователей и более высокие требования, чем когда-либо раньше. Современные веб приложения должны быть доступны из любой точки мира 24/7 и использоваться с любого устройства. На данный момент ожидается, что веб приложения должны быть клиент ориентированы. Для начала нужно выяснить для кого изначально создается приложение и какую специфичную задачу решает данное приложение. После того, когда появятся ответы на эти два вопроса, можно смело проектировать визуальную часть сайта. Приложения должны быть быстрыми, так как, если сайт медленный, то клиенты с него уходят, а следовательно бизнес теряет прибыль. Раньше разработчики использовали монолитный подход к разработке веб-приложений, встраивая каждую функцию прямо в свое приложение. Основной проблемой при таком подходе является техническое обслуживание. В результате изменение любого модуля может привести к проблемам в других сегментах, что сделает приложения неуправляемыми. Сейчас же используется модульный подход, что позволяет поддерживать и разрабатывать независимые части приложения. [29]

В данной главе, вы рассмотрим современные одностраничные приложения, разберем их специфику, а также архитектуру REST API, для того чтобы узнать, как подружить современный frontend и backend. Кроме того, узнаем для чего нужно создавать контейнеры для веб приложений и что такое Docker.

## **Одностраничные веб приложения**

Давайте сначала разберем как работают обычные веб приложения. Мульти страничные веб приложения работают в привычном нам виде. Каждое изменение, например, отображение данных или отправка данных формирует запрос на сервер, отображая новую страницу в браузере. Эти приложения большие и громоздкие, ими трудно управлять. Им на смену пришли одностраничные веб приложения. Вы скорее всего, вы уже пользовались одностраничным веб приложением, просто вы этого не замечали.[21]

Одностраничное веб приложение – это приложение, которое динамически переписывает страницу новыми данными с сервера, вместо классического способа, когда веб браузер загружает целые страницы. Вот, некоторые примеры одностраничных приложений: PayPal, Gmail, Netflix, Pinterest. Одностраничное приложение – это всего лишь одна страница, которая хранит в себе большое количество данных остается неизменными, и только некоторые части приложения меняются. Одностраничные приложения обрабатывают только то те данные, которые нужно, а браузер отрисовывает эту информацию. В этом основное отличие этих двух подходов. Кроме того, часть логики переносится в браузер, что позволяет обрабатывать данные быстрее, а также позволяет снять часть нагрузки с сервера. Этот подход устраняет прерывание взаимодействия пользователя с последовательными страницами, делая приложение похожим на десктопное. Некоторые части приложения остаются неизменными, такие какие, как шапка, подвал, меню веб приложения, а также могут и другие части, все зависит от макета приложения. Передавая работу по рисованию (рендеринг страницы) с сервера клиенту (вам), страница может быть динамически перезаписана, вместо полной перезагрузки. Это делает все намного быстрее.[40]

У мульти страничных веб приложений есть некоторые положительные стороны:

* Это идеальный подход для пользователей, которые нуждаются визуальная карта, куда идти в приложении. Мало уровневая навигация – это неотъемлемая часть любого мульти страничного приложения
* Это очень хороший и легкий способ для правильного управления SEO. Мульти страничные приложения дают больше шансов ранжироваться по разным ключевым словам, поскольку приложение может быть оптимизировано для одного ключевого слова на странице
* У мульти страничных приложений также есть и минусы:
* Это не подходящий подход, для того чтобы использовать тот же backend с мобильными приложениями
* Фронтенд и бэкэнд тесно связаны между собой
* Разработка становится довольно сложной

У одностраничных приложений можно выявить следующие положительные стороны:

* Единовременная загрузка файлов HTML, CSS и JavaScript. После того, как приложение загрузилось, сервер больше не присылает HTML, оно загрузилось в самом начале.
* Нет лишних запросов к серверу. Поскольку серверу не нужно тратить время и энергию на полную отрисовку, SPA в целом снижают нагрузку на ваши серверы: это означает, что вы можете сэкономить деньги, используя меньше серверов для того же объема трафика.
* Быстрый и отзывчивый интерфейс приложения. Наряду с более быстрой производительностью, описанной выше, SPA также позволяют разработчикам намного быстрее создавать внешний интерфейс. Это связано с несвязанной архитектурой SPA, или разделением фронтенда от бэкэнда.
* Благодаря раздельной настройке разработчики могут создавать, развертывать и экспериментировать с внешним интерфейсом полностью независимо от базовой серверной технологии.

Также не забываем про отрицательные стороны:

* Сделать SEO-оптимизацию одностраничного приложения очень сложно и непросто.
* Требуется, чтобы JavaScript присутствовал и был включен. Если какой-либо пользователь отключит JavaScript в своем браузере, будет невозможно правильно представить приложение и его действия.

Это является основными моментами разницы и сравнения этих двух подходов. Далее разберем архитектуру REST API.[50]

## **Архитектура REST API**

REST API (также известный как RESTful API) — это интерфейс прикладного программирования (API или веб-API), который соответствует ограничениям архитектурного стиля REST и позволяет взаимодействовать с веб-службами RESTful.[38]

API — это набор определений и протоколов для создания и интеграции прикладного программного обеспечения. Иногда его называют договором между поставщиком информации и пользователем информации, устанавливающим контент, требуемый от потребителя (вызов), и контент, требуемый производителем (ответ). Например, дизайн API для службы погоды может указывать, что пользователь должен предоставить почтовый индекс, а ответ производителя должен состоять из двух частей: первая — высокая температура, а вторая — низкая. Другими словами, если вы хотите взаимодействовать с компьютером или системой для получения информации или выполнения какой-либо функции, API поможет вам передать то, что вы хотите, этой системе, чтобы она могла понять и выполнить запрос. API – это посредник между пользователями или клиентом и ресурсами или веб сервисами. Это также хороший способ для обмена информацией, сохраняя при этом безопасность, контроль и аутентификацию, определяя кто и к чему получает досуп. Еще одно преимущество API заключается в том, что вам не нужно знать особенности кэширования — как извлекается ваш ресурс или откуда он берется. REST — это набор архитектурных ограничений, а не протокол или стандарт. Разработчики API могут реализовать REST различными способами. Когда клиентский запрос выполняется через RESTful API, он передает представление о состоянии ресурса запрашивающей стороне или конечной точке. Эта информация или представление доставляется в одном из нескольких форматов через HTTP: JSON (обозначение объектов Javascript), HTML или обычный текст. JSON является наиболее популярным форматом файлов для использования, потому что, несмотря на свое название, он не зависит от языка, а также удобен для чтения как людьми, так и машинами. Еще кое-что, о чем следует помнить: заголовки и параметры также важны в HTTP-методах HTTP-запроса RESTful API, поскольку они содержат важную информацию идентификатора, касающуюся метаданных запроса, авторизации, универсального идентификатора ресурса (URI), кэширования, файлов cookie и т. д. более. Существуют заголовки запросов и заголовки ответов, каждый из которых имеет собственную информацию о HTTP-соединении и коды состояния. [15]

Чтобы API считался RESTful, он должен соответствовать следующим критериям:

1. Архитектура клиент-сервер состоит из клиентов, серверов и ресурсов, с запросами, управляемыми через HTTP.
2. Связь между клиентом и сервером без состояния, т.е. между запросами на получение не хранится информация о клиенте, и каждый запрос является отдельным и не связан.
3. Единый интерфейс между компонентами для передачи информации в стандартной форме. Для этого необходимо, чтобы:

* запрашиваемые ресурсы идентифицируются и отделяются от представлений, отправляемых клиенту.
* ресурсы могут быть обработаны клиентом через представление, которое они получают, потому что представление содержит достаточно информации для этого.
* сообщения с самоописанием, возвращаемые клиенту, содержат достаточно информации, чтобы описать, как клиент должен их обрабатывать.

1. Многоуровневая система включает извлечение запрашиваемой информации в иерархии, невидимые для клиента.

Несмотря на то, что REST API имеет эти критерии для соответствия, он по-прежнему считается более простым в использовании.

REST — это набор рекомендаций, которые могут быть реализованы по мере необходимости, делая API REST более быстрыми и легкими, с повышенной масштабируемостью - идеально подходит для разработки Интернета вещей и мобильных приложений.[6]

Ключевой абстракцией информации в REST является ресурс. Любая информация, которую мы можем назвать, может быть ресурсом. Например, ресурс REST может быть документом или изображением, коллекцией других ресурсов и т.д. Состояние ресурса в любой конкретный момент времени называется представлением ресурса. API REST состоит из сборки взаимосвязанных ресурсов. Этот набор ресурсов называется моделью ресурсов REST API. [36]

Другая важная вещь, связанная с REST, — это методы ресурсов. Эти методы ресурсов используются для выполнения требуемого перехода между двумя состояниями любого ресурса. Большое число людей ошибочно связывает методы ресурсов с методами HTTP (т.е. GET/PUT/POST/DELETE). Например, если мы решим, что API приложения будут использовать HTTP POST для обновления ресурса - а не большинство людей рекомендуют HTTP PUT - все в порядке. Тем не менее интерфейс приложения будет RESTful. Ожидается, что все ресурсы будут поняты любым клиентом. Переходы могут быть определены (или ограничены) знанием клиентом типов медиа и механизмов связи ресурсов. Проще говоря, в архитектурном стиле REST данные и функциональные возможности считаются ресурсами и доступны с использованием унифицированных идентификаторов ресурсов (URI). Ресурсы обрабатываются с помощью набора простых, четко определенных операций. Кроме того, ресурсы должны быть отделены от представления, чтобы клиенты могли получать доступ к содержимому в различных форматах, таких как HTML, XML, обычный текст, PDF, JPEG, JSON и других.[3]

Когда мы открываем в браузере страницу, за кулисами происходит много вещей. Кратко этот процесс можно представить в следующим виде:



**Рисунок 1.1 – схема работы браузера и сервера**

API REST взаимодействуют через HTTP-запросы для выполнения стандартных функций базы данных, таких как создание, чтение, обновление и удаление записей (также называемых CRUD) в ресурсе. Например, API REST будет использовать запрос GET для извлечения записи, запрос POST для создания записи, запрос PUT для обновления записи и запрос DELETE для удаления записи. В вызовах API могут использоваться все методы HTTP. Хорошо разработанный API REST аналогичен веб-сайту, работающему в веб-браузере со встроенной функциональностью HTTP.[26]

При использовании REST сетевые компоненты являются ресурсом, к которому пользователь запрашивает доступ, подобно черному ящику, детали реализации которого неясны. Все звонки не имеют гражданства; ничто не может быть сохранено службой RESTful между выполнением. [42]

Форматы данных, поддерживаемые API REST, включают:

* application/json
* application/xml
* application/x-www-form-urlencoded
* multipart/form-data

JSON обозначает нотацию объекта JavaScript. JSON - текстовый формат для хранения и переноса данных. JSON — это легко понять, потому что он читабельный. JSON используется для передачи данных между компьютерами. JSON не зависит от языка программирования. Формат JSON синтаксически похож на код для создания объектов JavaScript. Благодаря этому программа JavaScript может легко преобразовывать данные JSON в объекты JavaScript. Поскольку формат является текстовым, данные JSON могут легко передаваться между компьютерами и использоваться любым языком программирования.[25]

XML означает расширяемый язык разметки. Язык разметки — это набор кодов или тегов, описывающих текст в цифровом документе. Наиболее известным языком разметки является язык гипертекстовой разметки (HTML), который используется для форматирования веб-страниц. XML, более гибкий двоюродный брат HTML, позволяет делать более сложные вещи. В то время как HTML сообщает приложению браузера, как должен выглядеть документ, XML описывает, что находится в документе. Другими словами, XML связан с тем, как информация организована, а не с тем, как она отображается.[19]

Файлы и двоичные данные можно отправлять непосредственно на коммуникационный сервер с помощью запроса POST. Данные, отправляемые в запросе POST, должны соответствовать определенным требованиям к форматированию. В запросе POST на коммуникационный сервер можно отправлять только следующие типы контента: application/x-www-form-urlencoded, multipart/form-data. Тип содержимого application/x-www-form-urlencoded описывает данные формы, которые отправляются в одном блоке в теле сообщения HTTP. В отличие от части запроса URL в запросе GET, длина данных не ограничена. Однако коммуникационный сервер отклоняет запросы, размер которых превышает размер, заданный параметром конфигурации MaxFileUploadSize. Этот тип содержимого неэффективен для отправки больших объемов двоичных данных или текста, содержащих символы, отличные от ASCII, и не позволяет загружать файлы. [33]

Атрибут enktype определяет способ кодирования данных формы при их отправке на сервер. Multipart/form-data является одним из наиболее используемых типов энктайпа/содержимого.[37]

В нескольких частях каждое из отправляемых полей имеет тип содержимого, имя файла и данные, отделенные границей от другого поля. Кодирование данных не требуется из-за уникальной границы. Двоичные данные отправляются как есть. Сервер считывает до следующей граничной строки. Multipart/form-data содержит границу между отдельными парами имя/значение. Граница действует как маркер каждого фрагмента пар имя/значение, передаваемого при отправке формы. Граница автоматически добавляется к типу содержимого заголовка запроса. Размер границы генерируется автоматически браузером. [51]

Помимо ограничений, связанных с дизайном и архитектурой, отдельные пользователи должны будут решать некоторые проблемы с REST API. Некоторые концепции, которые могут быть сложными, могут включать в себя:

* Согласованность эндпоинтов (адресов url, по которым отправляется запрос для получения данных) - пути конечных точек должны соответствовать общим веб-стандартам, управление которыми может быть затруднено.
* Управление версиями API - URL-адреса конечных точек не должны быть недействительными при использовании внутри или с другими приложениями.
* Длительное время отклика и слишком большое количество данных - количество возвращаемых ресурсов может увеличиться во времени, увеличивая нагрузку и время отклика.
* Запросы и данные -- запросы могут иметь больше данных и метаданных, чем необходимо, или может потребоваться больше запросов для получения всех данных
* Тестирование API - может быть длительным процессом для настройки и запуска. Каждая часть процесса может быть длительной или сложной.
* Определение кодов ошибок и сообщений. При использовании кодов ошибок более распространенной практикой является использование стандартных кодов ошибок HTTP. Они признаются клиентами и разработчиками чаще. Обработка ошибок может не иметь способа различить, является ли ответ успешным или нет, кроме анализа тела или проверки на наличие ошибки.

## **Контейнеризация веб приложений**

Контейнеризация — это упаковка программного кода со всеми необходимыми компонентами, такими как библиотеки, фреймворки и другие зависимости, чтобы они были изолированы в собственном «контейнере». Это означает, что программное обеспечение или приложение внутри контейнера может быть перемещено и последовательно запущено в любой среде и на любой инфраструктуре, независимо от этой среды или операционной системы инфраструктуры. Контейнер действует как своего рода пузырь или вычислительная среда, окружающая приложение и поддерживающая его независимым от окружающей среды. Это полностью функциональная и портативная вычислительная среда. Контейнеры являются альтернативой кодированию на одной платформе или операционной системе, что затрудняет перемещение их приложения, поскольку код может быть не совместим с новой средой. Это может привести к ошибкам, ошибкам и сбоям, которые необходимо исправить (что означает больше времени, меньше производительности и много разочарований). Упаковывая приложение в контейнер, который можно перемещать между платформами и инфраструктурами, это приложение можно использовать везде, где вы его перемещаете, поскольку оно имеет все необходимое для успешного выполнения в нем. [35]

Виртуальная машина (ВМ) — это виртуальная среда, которая функционирует как виртуальная компьютерная система со своим собственным CPU, памятью, сетевым интерфейсом и хранилищем, созданным в физической аппаратной системе (расположенной автономно или локально). Контейнеризация и виртуализация аналогичны тем, что позволяют полностью изолировать приложения, чтобы они могли работать в нескольких средах. Где основные различия заключаются в размере и переносимости. Виртуальные машины — это больше из двух, обычно измеряемых гигабайтами данных и содержащих собственную ОС, что позволяет им выполнять сразу несколько ресурсоемких функций. Увеличенные ресурсы, доступные для виртуальных машин, позволяют им абстрагироваться, разделяться, дублировать и эмулировать целые серверы, операционные системы, настольные компьютеры, базы данных и сети. Контейнеры намного меньше, обычно измеряются мегабайтом и не упаковывают ничего больше, чем приложение и его рабочая среда. [36]

Для того чтобы еще лучше понять, чем отличаются виртуальные машины от контейнеров, сначала нужно разобраться что такое виртуализация. Виртуализация — это процесс, при котором программное обеспечение используется для создания уровня абстракции над компьютерным оборудованием, который позволяет разделять аппаратные элементы одного компьютера на несколько виртуальных компьютеров. [38]

Используемое программное обеспечение называется гипервизором - небольшим уровнем, который позволяет нескольким операционным системам работать рядом друг с другом, совместно используя одни и те же физические вычислительные ресурсы. Когда гипервизор используется на физическом компьютере или сервере (также известном как сервер без металла) в центре обработки данных, он позволяет физическому компьютеру отделять свою операционную систему и приложения от своего оборудования. Затем он может разделить себя на несколько независимых «виртуальных машин».

Виртуальные машины — это технология создания виртуализированных вычислительных сред. Они существуют довольно давно и считаются основой первого поколения облачных вычислений. Проще говоря, виртуальная машина — это эмуляция физического компьютера. Виртуальные машины позволяют командам запускать несколько машин с несколькими операционными системами на одном компьютере.

Контейнеры являются более легким и гибким способом работы с виртуализацией, поскольку они не используют гипервизор, вы можете воспользоваться более быстрым выделением ресурсов и более быстрой доступностью новых приложений. Вместо того чтобы раскручивать всю виртуальную машину, пакеты контейнеризации объединяют все необходимое для запуска одного приложения или микросервиса (вместе с библиотеками времени выполнения, которые они должны запускать). Контейнер включает в себя весь код, его зависимости и даже саму операционную систему. Это позволяет приложениям работать практически в любом месте - настольном компьютере, традиционной ИТ-инфраструктуре или облаке. [37]

Можно выделить следующие плюсы виртуальных машин:

* Полная изоляция безопасности. Виртуальные машины работают изолированно как полностью автономная система. Это означает, что виртуальные машины защищены от любых эксплойтов или помех со стороны других виртуальных машин на общем хосте. Отдельная виртуальная машина все еще может быть захвачена эксплойтом, но эксплуатируемая виртуальная машина будет изолирована и не сможет загрязнять любые другие соседние виртуальные машины.
* Интерактивная разработка. Контейнеры обычно представляют собой статические определения ожидаемых зависимостей и конфигурации, необходимых для запуска контейнера. Виртуальные машины более динамичны и могут разрабатываться в интерактивном режиме. Как только для виртуальной машины задано базовое определение оборудования, виртуальную машину можно рассматривать как компьютер с голыми костями. Программное обеспечение может быть установлено на виртуальную машину вручную, а виртуальная машина может быть сфотографирована для получения текущего состояния конфигурации. Снимки файловой системы виртуальной машины можно использовать для восстановления виртуальной машины на данный момент времени или для ускорения вращения дополнительных виртуальных машин с такой конфигурацией. [35]

А также можно выделить следующие минусы:

* Скорость работы. Создание и регенерация виртуальных машин требует много времени, поскольку они охватывают систему полного стека. Любые изменения моментального снимка виртуальной машины могут занять значительное время для регенерации и проверки их поведения в соответствии с ожиданиями.
* Затраты на размер хранилища. Виртуальные машины могут занимать много места. Они могут быстро вырасти до нескольких гигабайт. Это может привести к нехватке дискового пространства на хост-машине виртуальных машин.

У контейнеров можно выявить следующие положительные стороны:

* Скорость работы. Поскольку контейнеры легковесны и включают только высокоуровневое программное обеспечение, их очень быстро модифицировать и работают они быстрее.
* Надежная экосистема. Большинство систем запуска контейнеров предлагают размещенное публичное хранилище предварительно изготовленных контейнеров. Эти контейнерные репозитории содержат множество популярных программных приложений, таких как базы данных или системы обмена сообщениями, и могут быть мгновенно загружены и выполнены, экономя время для групп разработчиков. [37]

Одним из самых популярных инструментов для контейнеризации веб приложений – является Docker. Docker — это инструмент, который позволяет разработчикам, системным администраторам и т.д. легко развертывать свои приложения в изолированной программной среде (называемой контейнерами) для запуска в операционной системе хоста, т.е. Linux. Ключевое преимущество Docker заключается в том, что он позволяет пользователям упаковывать приложение со всеми его зависимостями в стандартизированное устройство для разработки программного обеспечения. В отличие от виртуальных машин, контейнеры не имеют больших накладных расходов и, следовательно, обеспечивают более эффективное использование базовой системы и ресурсов. [39]

Контейнеры обеспечивают логический механизм упаковки, в котором приложения могут быть абстрагированы от среды, в которой они фактически работают. Это разделение позволяет легко и последовательно развертывать контейнерные приложения независимо от того, является ли целевая среда частным центром обработки данных, общедоступным облаком или даже персональным ноутбуком разработчика. Это дает разработчикам возможность создавать предсказуемые среды, которые изолированы от остальных приложений и могут запускаться в любом месте. [35]

С точки зрения операций, помимо переносимости контейнеры также обеспечивают более детальный контроль над ресурсами, что повышает эффективность инфраструктуры, что может привести к более эффективному использованию вычислительных ресурсов.[39]

Благодаря этим преимуществам контейнеры (& Docker) получили широкое распространение. Такие компании, как Google, Facebook, Netflix и Salesforce, используют контейнеры, чтобы сделать большие инженерные команды более продуктивными и улучшить использование вычислительных ресурсов. Фактически Google зачла контейнеры за устранение необходимости в целом дата центре. [37]

В файлах Dockerfile содержатся инструкции по созданию образа. С них, набранных заглавными буквами, начинаются строки этого файла. После инструкций идут их аргументы. Инструкции, при сборке образа, обрабатываются сверху вниз. Слои в итоговом образе создают только инструкции FROM, RUN, COPY, и ADD. Другие инструкции что-то настраивают, описывают метаданные, или сообщают Docker о том, что во время выполнения контейнера нужно что-то сделать, например — открыть какой-то порт или выполнить какую-то команду.[38]

## **Выводы главы**

Веб остаётся самой многогранной средой передачи информации. Мы продолжаем добавлять динамику на наши страницы, используя разнообразные методы и инструменты. Новые уникальные возможности предоставляет JavaScript. Веб разработка стремительно развивается, также как и инструменты. Очень часто появляются новый технологии и инструменты. Тем не менее, перед разработкой нового приложения или перед внедрением нового инструмента в проект, нужно хорошо подумать, какие преимущества те или иные технологии могут преподнести. На текущий момент, одни из самых популярных технологий являются одностраничные веб приложения и архитектура REST API. Они позволяют создавать легковесные, быстрые веб приложения достаточно быстро и без трудностей. Современные веб-технологии предоставляют разработчикам неограниченные возможности для реализации своих идей. Для того, чтобы вы могли использовать весь их потенциал, необходимо знать, как правильно их использовать.

Эта тема такая широкая, что о ней можно издать целую книгу, но рано или поздно разговор стоит завершать. История веба особенно интересна тем, что она не стояла на месте: здесь всегда все активно развивается, что-то бурлит и взрывается, несколько шумно приходит, а потом незаметно исчезает. Учитывая роль веба в мировом бизнесе, нет ничего удивительного в том, что он был, есть и, несомненно, будет в будущем полем битвы больших корпораций за свое влияние на обычных пользователей и разработчиков.

Новые идеи, подходы, технологии, платформы, инструменты нацелены на решение важнейшей задачи веб-разработки — обеспечение более качественного пользовательского опыта. Решения с учетом технологического развития и запроса пользователей.

Мы прекрасно понимаем, что следовать всем трендам веб-разработки очень сложно, они быстро меняются. Но большинство из представленных в этой работе решений появились не сегодня, доказали свою эффективность и с успехом применяются. Так почему же не использовать их на своем сайте или в мобильном приложении?

# ГЛАВА II. ОСОБЕННОСТИ ОДНОСТРАНИЧНЫХ ПРИЛОЖЕНИЙ СТЕКА MEVN

В данной главе мы рассмотрим, что такое технический стек, как его правильно выбрать и почему его нужно выбрать правильно, а также мы рассмотрим стек MEVN. Стек MEVN – это акроним из следующих технологий: Mongo, Express, Vue, Node.JS. В этой главе будет подробно разобран каждый элемент из этого стека. Также будет рассмотрен механизм аутентификации на основе токенов.[22]

Технический стек– это комбинация технологий, которая команда разработчиков использует, чтобы создать и запустить приложение или проект. Иногда технический стек еще называется “solution stack”. Обычно технический стек состоит из языков программирования, фрэймворков, баз данных и других приложений, которые подключены через API. Инвестирование в различные элементы технологичного стека является важным шагом на пути к успеху разработки продукта, поскольку технический стэк даёт команде инструменты для создания и обслуживания продукта, а также обеспечение того, чтобы он продолжал потребности клиентов.[5]

Вот основные категории, а также популярные категории:

* **Операционные системы и язык программирования**. Их выбор зависит от среды, в которой удобно разрабатывать, а также от типа приложения, которое хотим разработать. В результате выбор может пасть на несколько языков программирования, в зависимости от требования серверной части или user experience, а также для каких устройств создается приложение
* **Хранение данных.** Этот слой стэка состоит из реляционных или не реляционных баз данных, хранилищ данных, которые позволяют хранить и обрабатывать данные в режиме реального времени. Эти компоненты являются ключевыми для хранения данных.
* **Фрэймворки для бэкэнда.** Фрэймворк – это набор библиотек и утилит, предназначенных для помощи разработчикам в создании приложений. Фрэймворки часто включают в себя некоторые базовые функции, необходимые для создания приложений и обеспечивают структуру для таких веще, как организация и взаимодействие с базой данных, обработка запросов от пользователей, отправка сообщений электронной почты о регистрации или сбросе пароля. В то время как некоторые предпочитают отказываться от фрэймворков и кодировать все с нуля, они часто получают более сложную и трудоемкую работу.
* **Фрэймворки фронтенда.** Сервисы и платформы, которые будут использоваться для создания пользовательского интерфейса.

Почему так важно выбрать технический стек? То, как вы создаете свой технический стек во многом влияет на дальнейшую разработку продукта. Этот процесс всегда требует компромиссов: одни технологии экономят время, но допускают меньше настроек, другие лучше подходят для определенной аудитории, третьи более масштабируемые, но требуют постоянного обслуживания. Однако можно собрать такой стек, который будет соответствовать требованиям и развиваться по мере развития проекта.[6]

Современный технический стек включает в себя множество компонентов, потому что возрастает количество устройств, с которых пользователи могут использовать приложение и происходит большое количество событий и обработки данных.[20]

Подготовка стека к будущему может быть палкой о двух концах. Если изначально не подумать, как проект будет масштабироваться, им возможно придется добавить дополнительные службы, которые сделают его громоздким и сложным в управлении. С другой стороны, если предвидеть экспоненциальный рост и слишком много инвестируют в дорогостоящие инструменты и услуги, то могут закончиться деньги, до того, как приложение добьется успеха на рынке – если вообще добьется. Одна из лучших стратегий – создавать приложения, с использованием инструментов с открытым исходным кодом для проверки концепций, прежде чем инвестировать в них, и искать инструменты, которые обеспечивают гибкость. Как правило разработчики отдают предпочтения языкам, которые они уже знают, но создание наилучшего технического стека иногда стоит сделать шаг назад и позволить цели проекта выбирать стек. Например, будет ли доступно для мобильных устройств или на компьютере? Если на компьютере, то какие браузеры будут поддерживать. Если это медиасайт, который будет принимать миллионы пользователей в день или приложение для мобильного банкинга, то насколько оно должно быть безопасным. Для каждой из этих целей есть свой технический стек.[29]

Каждый хочет сделать успешное приложение, которым будут пользоваться большое количество людей. Если выбрать платформы и базы данных, которые хорошо масштабируются, то на этапе планирования вы будете избавлены от проблем по мере роста пользовательской базы данных. Кроме того, правильный выбор технического стека может ускорить процесс разработки, так как всегда можно нанять разработчиков, которые являются экспертами по отдельным технологиям. Также можно выбрать конкретный стек технологий, чтобы использовать опыт команды. Кроме того, разработчикам становится проще общаться о функциональности приложения.

Заблаговременное планирования стека технологий означает, что вы уже знаете требования к ресурсам, инфраструктуре, оборудованию и другим требованиям проекта, что позволяет сэкономить и время, и деньги. [9]

Наконец, поскольку вы знаете цель проекта, которого вы планируете создать, можно принять решение о сложности стека и избежать использование технологий, которые не требуются для текущего проекта, и избежать громоздких ресурсов.

В данной работе мы будем использовать технический стек MEVN. Акроним MEVN исходит их следующих технологий: MongoDB, Express.Js, Vue.Js, Node.Js. Стек MEVN – это программный стек JavaScript с открытым исходным кодом, который появился, как новый и развивающийся способ создания мощных и динамических веб приложений. Его программные компоненты можно использовать для эффективного проектирования внутреннего и внешнего интерфейса и улучшения функциональности вашего веб приложения.

MongoDB – документно-ориентированная база данных, которая хранит данные приложения.

Express.Js – фреймворк, наложенный поверх Node.Js, используемый для создания серверной части сайта с использованием функций и структур Node.Js. Поскольку Node.Js создавался для запуска JavaScript на компьютере, а не для создания визуальной части веб приложений.

Vue.Js – фреймворк для создания визуальной части приложений, активно используется во фронтенд разработке.

Node.Js – среда выполнения JavaScript. Он используется для запуска JavaScript на компьютере, а не в браузере.

Схему взаимодействия каждых компонентов стека можно увидеть на рисунке:

**Рисунок 2.1 - схема взаимодействия компонентов стека MEVN**

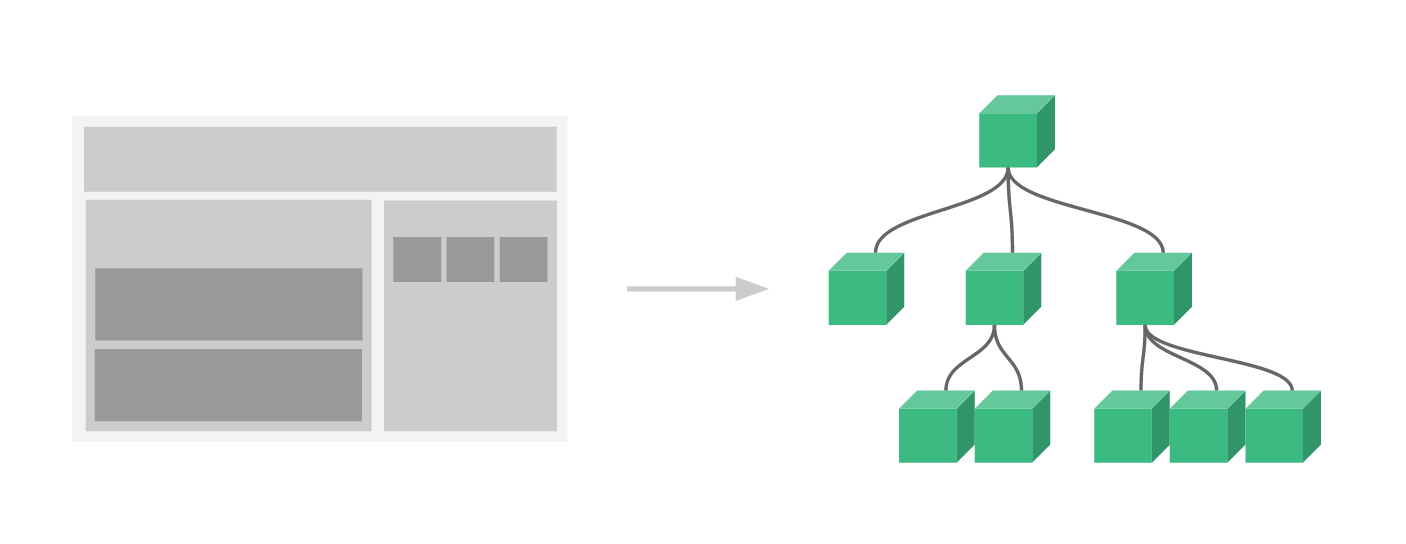
## **Vue.Js и Vuex**

Vue.Js – это прогрессивный JavaScript фрэймворк для разработки веб приложений и он не загружается в браузере, нам нужен отдельный механизм чтобы загрузить index.html (один файл html), куда соберется готовый проект со всеми зависимостями CSS и JavaScript. В данном случае мы будем использовать node.js как веб сервер, который будет подгружать все зависимости и делать запросы на REST API. [32]

За последние 2 года Vue прошел долгий путь от малоизвестного фреймворка до очень известного фреймворка. Причины, по которым Vue стал таким популярным кроятся в его характеристиках:

* Доступность. Если вы знакомы с HTML, CCS, JavaScript, то изучить этот фрэйворк не будет большой проблемой
* Универсальный. Vue хорошо масштабируется и идеально адаптируется к потребностям. Вы можете использовать его для создания небольшого интерактивного компонента, который хорошо сочетается с остальной частью вашего сайта или веб приложения, или же для полноценного гигантского веб приложения. Что еще важно, что Vue работает с другим кодом лучше, чем другой фрэймворк.
* Мощный и эффективный. Сжатая версия данной библиотеки занимает всего 20 килобайт, что делает его идеально подходящим для веб приложений, которые ориентированы на производительность или в качестве замены библиотекам, которые ориентированы на манипуляции с DOM деревом.

Vue также является фреймворком, ориентированным на простоту достижения целей. Многие люди говорят, что у Vue лучшая документация в отрасли, что делает его еще проще. Vue – это один из фрэймворков, который позволяет строить веб приложения на основе компонентной архитектуры. Это означает, что мы можем разделять веб страницы на более маленькие автономные и повторно используемые компоненты. В идеале ни один из компонентов не должен зависеть от других компонентов. В проекте Vue каждая часть приложения – это отдельный компонент, что упрощает дальнейшую разработку приложения. Давайте рассмотрим следующую схему:



**Рисунок 2.2 – Разделение веб страницы на компоненты**

С левой стороны мы можем увидеть типичное веб приложение, разделенное на секции. Каждая секция может быть представлена, как отдельный компонент. Этот едва заметный светло серый контейнер, который охватывает все остальные фигуры – называется корневым компонентом. Корневой компонент – это всего лишь контейнер для нашего приложения. Каждый компонент внутри этого контейнера называется ребёнком, компоненты внутри ребенка, тоже могут быть разделены на более мелкие компоненты, и тоже могут быть детьми этого компонента. С данного рисунка мы можем увидеть, что у нашего корневого компонента трое детей: заголовок, левая часть контента и правая часть контента. Некоторые из них имеют и своих детей, такие как один пост. С правой стороны мы можем увидеть дерево, представляющее связь родитель ребенок внутри приложения. Каждое приложение можно представить в виде такого дерева. [32]

Компонентная архитектура фокусируется на декомпозиции дизайна на индивидуальные, функциональные и автономные компоненты. Это даёт нам высокий уровень абстракции и даёт нам возможность разделить большую проблему на несколько маленьких. Основная цель компонентной архитектуры – это возможность повторного использования компонентов. Компонент инкапсулирует функциональность и поведение. У компонентного подхода есть несколько преимуществ [12]:

* Сокращение времени разработки за счет пере использования компонентов
* Повышенная надежность за счет использования существующих компонентов

Что такое компонент? Компонент – это модульный, переносимый, заменяемый и повторно используемый набор четко определённого функционала, который инкапсулирует его реализацию и экспортирует его как интерфейс более высокого уровня. Компонент – это программный объект, предназначенный для взаимодействия с другими компонентами, инкапсулирующий определённую функциональность или набор функций. Он имеет четко определенный интерфейс и соответствует его поведению, общему для всех компонентов в архитектуре. Компонент может быть определен как единица композиции только в определенном контракте с интерфейсом и явными контекстными зависимостями. То есть программный компонент может быть развернут независимо и может успешно быть использован в других проектах. Можно выявить следующие характеристики компонента [18]:

* **Возможность повторного использование.** Компоненты обычно предназначены для повторного использования в различных ситуациях и в разных приложениях. Однако некоторые компоненты могут быть предназначены для какой-то конкретной задачи.
* **Заменяемые.** Компоненты могут быть успешно заменены другими аналогичными компонентами.
* **Не зависят от контекста.** Компоненты предназначены для работы в любых средах и любых контекстах.
* **Расширяемые.** Компонент может быть расширен из существующих компонентов, чтобы обеспечить новое поведение.
* **Инкапсулированные.** Например, компонент А отображает интерфейсы, которые позволяют вызывающей стороне использовать его функциональные возможности и не раскрывать подробности внутренних процессов или каких-то внутренних переменных или состояний.
* **Независимые.** Компоненты разработаны, чтобы у них была минимальная зависимость от других компонентов.

Компонентный дизайн может быть представлен с использованием некоторого промежуточного состояния (например, с использованием графического, табличного или текстового), которое можно преобразовать в исходный код. Проектирование структур данных, интерфейсов, алгоритмов и компонентов должно соответствовать общепринятым правилам, чтобы избежать ошибок [26]:

* Программное обеспечение разбивается на многократно используемые, связанные и инкапсулированные компоненты
* Каждый компонент имеют свой собственный интерфейс и каждый компонент скрывает свою детальную реализацию
* Компонент должен быть расширен без изменения внутреннего кода или изменения дизайна в существующей части компонента
* Компонент должен быть зависим от абстракций и не зависеть от других конкретных компонентов, которые увеличат сложность использования
* Взаимодействие компонентов может принимать форму вызова методов, асинхронных вызовов, широковещательной рассылки, взаимодействий, управляемых сообщениями, обмена потоков данных
* В интерфейсе должны быть только те методы, которые актуальны для текущего компонента.

Существуют определенные соглашения о наименовании компонентов, которые указаны как часть архитектурной модели:

* Имя компонента должно быть из предметной области и должно гарантировать, что оно понятно всем заинтересованным сторонам, которые просматривают эту архитектурную модель
* Извлекает сущности бизнес-процессов, которые могут существовать независимо от других сущностей
* Использовать такие имена, которые отражают их конкретное значение для реализации
* Моделировать любые зависимости компонентов как интерфейсы, а не представлять их как прямую зависимость между компонентами.

Можно выявить плюсы у компонентного архитектурного подхода [10]:

* **Простота развертывания**. По мере появления новых совместимых версий проще заменить существующие версии, не влияя на другие компоненты или систему в целом
* **Снижение стоимости**. Использование сторонних компонентов, позволяет распределить стоимость разработки и обслуживания
* **Простота разработки**. Компоненты реализуют известные функции и свойства, что позволяет разрабатывать, не затрагивая другие части системы
* **Многоразовое использование.** Использование повторно используемых компонентов, означает, что их можно использовать для распределение затрат на разработку и обслуживания между несколькими приложениями или системами
* **Модификация технической сложности.** Компонент изменяет сложность за счет использования контейнера и абстракции
* **Надежность.** Общая надёжность системы увеличивается, поскольку надежность каждого одного компонента повышает надежность всей системы за счет повторного использования.
* **Обслуживание и развитие системы.** Легко обновить и изменить реализацию, не затрагивая остальную часть системы.
* **Независимость.** Независимость и гибкое подключение компонентов. Независимая разработка компонентов разными группами разработчиков параллельно. А также это увеличивает производительность разработки.

Любой компонент в Vue состоит из HTML, CSS, JS. Такой подход называют Single File Components (SFC). Эта функция позволяет поместить весь код, который относится к одному компоненту в один файл с расширением .vue. Давайте рассмотрим базовую структуру компонента:



**Рисунок 2.3 – структура компонента**

Каждый SFC – разделен на три секции:

* Шаблон – обычно это HTML разметка
* Скрипт – сущность, где описана логика компонента
* Стили – стили описанные на CSS, или на любом другом препроцессоре, например, как SCSS

Как вы заметили, это отличный способ хранить код, относящийся к конкретному компоненту в одном месте. Кроме того, такие компоненты могут быть разделены и использованы намного проще, чем если бы все три части были бы разделены. Это одна из причин почему очень легко поддерживать даже очень сложные приложения, которые разработаны на Vue. Все три части связаны между собой и могут влиять друг на друга. Стили из раздела стилей напрямую применяются к шаблону, там же можно использовать свойства данных и методы из раздела скриптов. Давайте рассмотрим приведенный выше код, чтобы понять, что там происходит: в разделе шаблон мы отображаем параграф, внутри этого параграфа, используя интерполяцию, чтобы отобразить данные из скрипта. Внутри фигурных скобок мы может отображать JavaScript. Как вы могли догадаться, что результат в параграфе, это значение переменной message из секции скриптов – “Hello World”. Ниже у нас есть кнопка, которая при нажатии вызывает метод sayHello. В Vue привязка событий к элементам чуть отличается от обычных JavaScript приложений, тут используется директива v-bind:click, ее также можно упростить до :click и передать ей callback функцию, в то время как в обычном JavaScript проекте, чтобы привязать событие нужно добавить такой код: button.addEventListener(‘click’, sayHello). Согласитесь, что во Vue это делается намного проще. Ниже шаблоны в экспортируем сущность компонента, которая состоит из данных и методов. Зная всего лишь эти два свойства, вы можете выполнять почти все основные задачи, используя Vue. Вы можете получить доступ к свойствам и методам внутри экземпляра используя ключевое слово this, в то время как в компоненте, чтобы отобразить эти данные ключевое слово this не используется [4].

Так же очень важно, осветить одну концепцию Vue – это реактивность. В Vue есть так называемая система реактивности. Если коротко говорить, то данная система реактивности отслеживает изменения в переменных и меняет их значения везде, где вы его использовали, поэтому вызывая метод sayBye, мы меняет значение это переменной не только в свойстве данных экземпляра Vue, но также и в шаблоне, что придает динамичности нашему приложению и позволяет динамически отображать данные, которые приходят нам от сервера. Так же стоит отметить, что компоненты общаются между собой по принципу “Props In, Events Out”. То есть, от родительского элемента в дочерний информация передается через пропсы, а обратно вызываются события. За маршрутизацию в приложении отвечает определенный пакет – vue-router. Он поддерживает вложенные маршруты к компонентам [7].

У компонентов есть свой жизненный цикл. Каждая сущность Vue через серию шагов инициализации. От создания настроек за наблюдением данных до компиляции шаблона, до монтирования в DOM дерева, и далее до изменения DOM дерева в процессе изменения данных. Этот процесс известен как жизненный цикл компонентов Vue, и у компонентов есть некоторые функции, которые запускаются по умолчанию, когда они проходят этот процесс создания и обновления DOM. Есть всего 8 функций жизненного цикла [16]:

* Before create
* Created
* Before mount
* Mounted
* Before update
* Updated
* Before destroy
* Destroyed

Before create – это функция жизненного цикла, вызывается сразу как только компонент был инициализирован. Стоит обратить внимание, что данная функция жизненного цикла, срабатывает до того, как компонент был загружен. Эта функция жизненного цикла, до того, как движок Vue создает компонент. На этом этапе события, данные не доступны [16].

Created – данная функция жизненного цикла, как вы могли догадаться, создается сразу после before create хука. На этой стадии, Vue уже инициализирован и активирует такие вещи, как данные события и другое. То есть уже можно манипулировать данными событиями, вычисляемыми свойствами. Когда мы запускаем приложение, то мы можем заметить, что наши данные уже отображаются в шаблоне. Это невозможно сделать на этапе before create, так как активация компонента еще не произошла. Однако на этом этапе компонент еще не монтируется в DOM дерево, так что пока еще нельзя манипулировать элементами DOM дерева [16].

Before mount – это следующий жизненный цикл, который вызывается сразу после created хука. Этот метод непосредственно вызывается перед монтированием компонента в DOM дерева. На данном этапе компилируется шаблоны и стили компилируются, но еще нельзя управлять элементами в DOM дереве. Вот где должны происходить инициализации, которые вы не хотите включать в компонент [16].

Mounted – это следующий хук жизненного цикла, который будет вызван сразу после хука Before Mount. Он вызывается сразу после монтирования экземпляра. Здесь компонент приложения или любой другой компонент приложения в проекте становится функциональным и теперь его можно использовать. Данные помещаются в шаблон, элемент DOM заменяется элементом, заполненным данными и теперь ими можно манипулировать [16].

Before update – этот элемент жизненного цикла срабатывает сразу после хука mounted. Каждый раз, когда происходят изменения в данных, которые требуют изменения в DOM дереве данный элемент жизненного цикла срабатывает. Эта стадия идеально подходит для любой логики до изменения данных, такая как удаление событий. Первоначально – это приветственное обращение к DOM дереву на этапе монтирования (где можно манипулировать DOM деревом), данные в компоненте меняются, и перед их изменением срабатывает данный метод [16].

Updated – данный метод жизненного цикла вызывается сразу после того как произошло изменение в DOM дереве. Это означает, что он вызывается сразу же после вызова метода before update. На этом этапе доступны операции с DOM деревом, а также не рекомендуется изменять состояние приложения внутри этого метода жизненного цикла [16].

Before update – этот метод жизненного цикла вызывается сразу когда компонент разрушается. На этом этапе сам компонент еще существует и весь функционал работает. Этот этап подходит для управления ресурсами, удаление переменных и очистку компонента [16].

Destroyed – это финальная стадия жизненного цикла компонента и всех его детей. Такие вещи как данные, события на этом этапе не привязаны. Этот метод срабатывает тогда, когда компонент разрушается и удаляется из DOM дерева [16].

Когда мы работаем с Vue, общение между компонентами и передача состояния наверх или вниз между компонентами происходит достаточно легко. Когда мы работаем над маленькими приложениями, общение между компонентами не доставляет нам проблем, но, когда в нашем приложении становится очень много компонентов, у которых есть вложенность на два или три уровня, а то и более – поддерживание состояния приложения становится достаточно сложной задачей. На этом этапе стоит рассмотреть такой инструмент, как Vuex. Хотя Vuex помогает справиться с общим управлением состояния приложения, это также требует дополнительных концепций и шаблонов. Это компромисс между краткосрочной и долгосрочной продуктивностью [17].

Главной целью Vuex является обеспечение предсказуемого состояния приложения с централизированным хранилищем для всех компонентов вашего приложения. Данная библиотека обеспечивает использование некоторых правил для управления состоянием, что помогает обеспечить лучшую структуру и удобство сопровождения кода в дальнейшем. [15]

Давайте пройдемся по ключевым концепциям, чтобы лучше понять, как работает Vuex. Компоненты могут использовать глобальное хранилище, а также использовать своё внутреннее состояние. Любая информация, которая будет нужна в нескольких компонентах, должна находиться в глобальном хранилище. Чтобы у компонентов был доступ к глобальному хранилищу, глобальное хранилище должно быть подключено в корневом компоненте. Vuex использует единое дерево состояний, то есть всего единственный объект содержит все состояние вашего приложения и служит единым источником истины. Единое дерево состояний позволяет легко найти конкретную часть состояния и легко делать снимки текущего состояния приложения. Единое дерево состояний не противоречит модульности. Данные, которые мы храним в хранилище подчиняются таким же правилам, что и данные в экземпляре Vue, то есть объект состояния должен быть простым. Использование Vuex, не означает мы должны помещать все данные состояния в хранилище. Добавление большого количества состояний делает изменение состояния более ясным, иногда это может сделать код более многословным и косвенным. Если часть состояния принадлежит одному компоненту, то было бы неплохо оставить его как локальное состояние компонента. Мы должны взвесить все компромиссы и принять решения, которые соответствуют требованиям нашего приложения. [31]

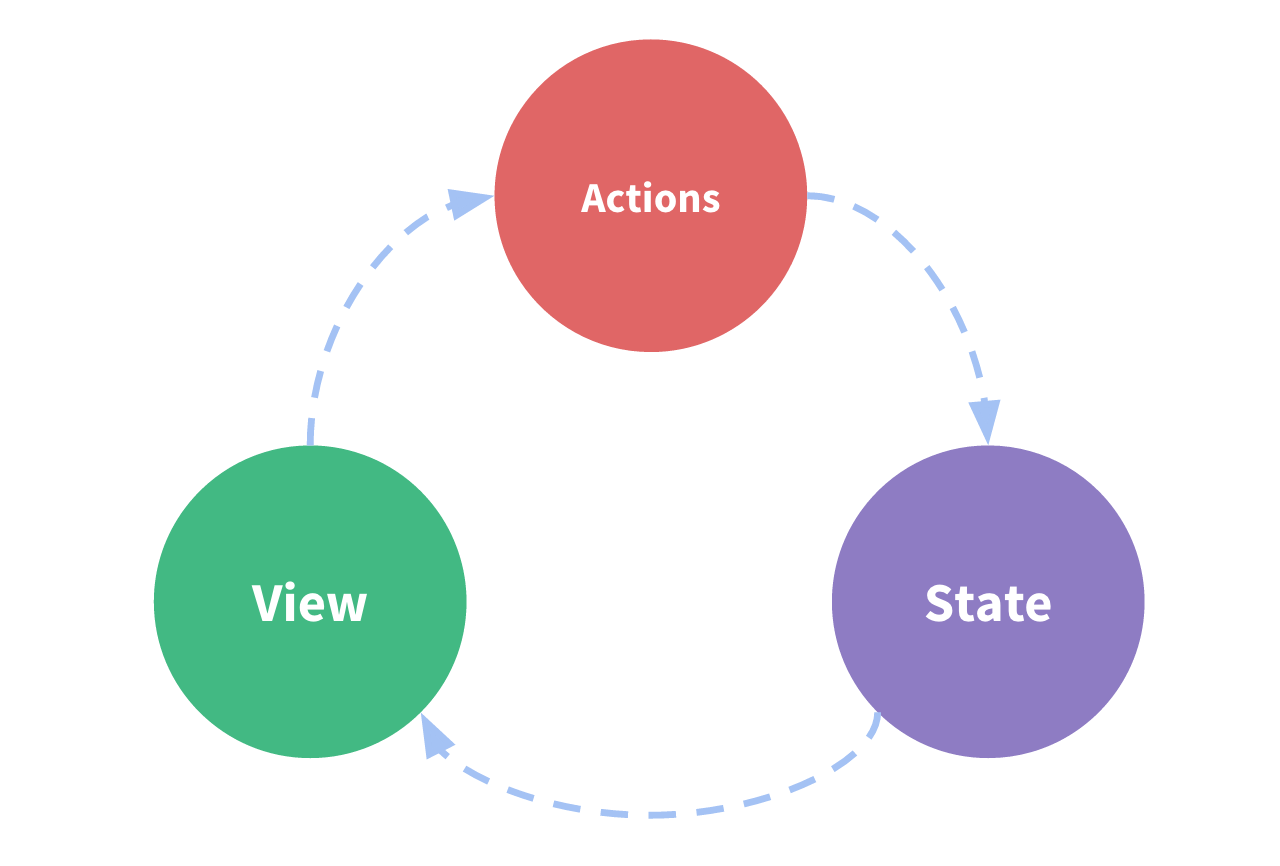
Сначала нужно познакомиться с основными концепциями Vuex.У Vuex очень впечатляющая реализация, и требует глубокое понимание следующих концепций.

Состояние (Store, State) – это просто данные приложения, с которыми взаимодействуют компоненты. Объект состояния – это единый объект, который содержит все состояние вашего приложения и служит единым источником правды и единым источником данных. Его основное местонахождение – это состояние.

Vuex позволяют нам определять геттеры в хранилище. Вы можете думать о них как динамические свойства хранилища. Результаты геттеров кэшируются и будут переоцениваться, и будут переоцениваться только тогда, когда одна из зависимостей изменилась. [30]

Один из способов изменения состояния – это мутации. Мутации похожи на события, в том смысле что у них есть обработчик для совершения мутаций. Мутации синхронны. Это означает что может быть выполнена только одна мутация за одно действие. [21]

Действия аналогичны мутациям с той разницей, действия вызывают мутации, а действия могут содержать произвольные асинхронные операции. Действия запускаются с помощью метода хранилища – dispatch. Схему работы Vuex можно увидеть на рисунке

**Рисунок 2.4 – схема работа Vuex**

Создание крупномасштабного приложения может привести к большому объему данных в хранилище. На помощь приходят модули, которые служат контейнером для хранилища.

Для начала мы должны инициализировать свойства состояния нашего приложения, когда мы создаем хранилище. Изменения в хранилище можно достичь всего лишь двумя способами:

Совершить мутацию. Мутации – это операция над состоянием приложения. Мы не может выполнять мутации напрямую из компонентов. Чтобы выполнить мутацию, мы должны использовать метод commit из хранилища, который получает имя мутации и может получить объект, который может содержать дополнительные данные. Мутации синхронны. [2]

Вызов действий. Хранилище так же может содержать действия. Для того чтобы выполнить действия из компонентов, мы должны вызвать метод из хранилища – dispatch. Этот метод также получает имя действия и объект с дополнительными данными. Действия в конечном итоге совершают мутации, они не изменяют состояние напрямую. Действия могут быть асинхронными. [11]

Основная причина существование действий и мутаций заключается в разделении проблем между синхронными и асинхронными операциями. Когда состояние хранилища изменяет какой-то компонент, то компонент используя часть состояния реагирует и изменяет состояние. Для закрепления информации о работе внутренних механизмов Vuex, предлагаю рассмотреть следующий рисунок:

**Рисунок 2.5 – схема работы внутренних механизмов Vuex**

Vuex – это замечательный и мощный инструмент. Однако с большой силой, приходит ответственность. Vuex позволяет нам контролировать данные приложения более организованным способом. С его помощью мы определяем состояние, а также выполнять действия и мутации. Компоненты также могут беспрепятственно передавать друг другу информацию, по сравнению с традиционным методом обновления состояния. Жизненный цикл управления состоянием работаем безупречно и поддерживает эффективный способ передачи состояние между компонентами. Использование Vuex действительно означает добавления большого количества шаблонного кода. Однако окончательный результат оправдывает это, и когда приложение вырастет в размерах, вы скажете спасибо. Как говорил Дэн Абрамов: «Библиотеки для управления менеджментом состояния приложений похожи на очки: вы будете точно знать, когда они вам понадобиться.» [17]

## **Node.JS и Express.JS**

Node.JS – это кроссплатформенная среда выполнения JavaScript с открытым исходным кодом. Это популярный инструмент практически для любого проекта. Node.JS использует движок V8, ядро Google Chrome все браузера. Это позволяет ему быть очень производительным.

Приложения Node.JS выполняется в одном процессе, без создания нового потока для каждого запроса. Node.JS предоставляет набор функций для асинхронного ввода-вывода в своей стандартной библиотеки, которые предотвращают блокировку кода JavaScript, и как правило библиотеки в Node.JS написаны с использованием неблокирующих парадигм, что делает поведение блокировки исключением. Когда Node.JS выполняет операцию ввода-вывода, например чтение из сети, доступ к базе данных или файловой системе вместо того, чтобы блокировать поток или тратить ресурсы CPU на ожидание, Node.JS возобновит операцию, когда возвращается ответ. Это позволяет Node.JS обрабатывать тысячи одновременных подключений к одному серверу, не утруждая себе параллельными потоками, что может быть значительным источником ошибок. Node.JS имеет огромное преимущество, потому что миллионы frontend разработчиков, который пишут JavaScript для браузеров, теперь могут писать код и для сервера без необходимости изучения совершенного нового языка программирования. В Node.JS новые стандарты языка можно использовать без проблем, так как не нужно ждать пока все ваши пользователи обновят свои браузеры – вы сами решаете какую версию стандарта языка использовать, просто изменив версию Node.JS, а также можно включить определенные экспериментальные функции, запустив Node.JS с определенными флагами.

NPM – это пакетный менеджер для Node.JS. NPM имеет простую структуру, которая помогла экосистеме Node.JS развиваться. В реестре NPM размещено более чем 1000000 пакетов с открытым исходным кодом, которые можно свободно использовать.

И браузер, и Node.JS используют один и тот же язык программирования. Создание приложений, которые запускаются в браузере – это совершенно другое, чем создание приложений на Node.JS. Несмотря на то, что везде JavaScript, все же имеются ключевые отличия. С точки зрения фронтенд разработчика, который широко использует JavaScript, приложения на Node.JS дают огромное преимущество – удобство программирование чего угодно и фронтенда и бэкэнда на одном языке. У разработчика есть огромные возможности, потому что это очень сложно полностью и глубоко изучить язык программирования, а используя один язык программирования, разработчик находится в уникальной и выгодной позиции [13].

В браузере большую часть времени, чем занят разработчик – это взаимодействие с DOM деревом или другими вэб элементами, такие как cookie или localstorage. Этого, конечно, не существует в Node.JS, тут нет таких глобальных объектов, как window или document, которые представлены браузером. В тоже время у браузера нет, такого функционала как предоставляет Node.JS через его модули, как доступ к файловой системе. Другое большое различие, в том, что в Node.JS можно контролировать среду. Если мы создаем открытое приложение, которое можно развернуть где угодно, уже заранее известно какая версия Node.JS будет использована. По сравнению со средой браузера, где мы не может себе позволить себе роскошь выбирать, какой браузер будут использовать наши посетители. [19]

Express.JS – это веб-фрэймворк, который основан на Node.JS. Используя этот фрэймворк, разработчики вольны сами выбирать любые библиотеки, которые им нужны для конкретного проекта, что обеспечивает гибкость в разработке и настройке. Если писать код на чистом Node.JS, то часто пишешь один и тот же код для решения определенных задач, таких как:

* Разбор тела запроса HTTP
* Разбор cookie
* Оперирование сессиями
* Организация маршрутов для запросов
* Определение правильных заголовков ответа на основе типов данных
* Express.JS решает эти проблемы и много других проблем. У Express.JS есть основной файл входа в приложение. В этом файле мы может сделать следующие шаги:
* Подключать сторонние библиотеки, свои модуля, контроллеры, вспомогательные функции и модели
* Настраивать приложение Express (изменять шаблонизатор или расширения файлов)
* Определять маршруты запросов
* Подключаться к базам данных, таким как MongoDB, MySQL и др.
* Запускать приложение

Когда приложение Express запущено, оно слушает запросы. Каждый входящий запрос проходит маршрутизацию. Этот аспект очень важен, потому что это позволяет управлять потоком выполнения [16].

Express – позиционирует себя, как минималистичный и гибкий фрэймворк для веб приложений, который предоставляет надёжный функционал для построения одностраничных, мульти страничных и гибридных приложений. Давайте разберем плюсы данного фрэймворка:

**Минимализм** – это один из главных аспектов Express. Философия фреэймворка заключается в том, чтобы предоставить минимальный слой между мозгом и сервером.

**Гибкость** – это другой ключевой аспект, который позволяет расширять Express. Express предоставляет минималистичный фрэймворк, и мы можем добавлять в разные части фрэймворка функционал, который нам нужен, заменяя то, что нам не нужно. Так много фрэмворков дают нам большое количество функционала, оставляя нас с загадочным и сложным проектом, еще до того, как мы написали первую строчку кода.

**Веб приложения.** Что такое веб приложение? Означает ли это, что мы не можем построить веб сайт или веб странички, используя Express? Нет, веб сайт – это веб приложение, и веб страничка – это тоже веб приложение. Вообще, веб приложение используется для того, чтобы обозначить какой-то функционал, это не только статичная коллекция контента. На текущий момент граница между веб сайтом и веб приложением постепенно размывается.

**Одностраничные приложения –** это относительно новая идея. Вместо веб сайта, который требует сетевой запрос каждый раз, когда пользователь переходит на другую страницу, одностраничные веб приложения загружает весь сайт на клиентский браузер. После первоначальной загрузки навигация выполняется очень быстро, потому что общение с сервером сводиться до минимума. [15]

**Многостраничные приложения –** это более традиционный подход для разработки веб приложений. Каждая страница в веб приложении представлена серверу отдельным запросом. Существует несколько вариантов создания веб приложений, используя данный фрэймворк. Также существует гибридный способ – это такой способ, который совмещает в себе подход одностраничного и многостраничного приложения. [15]

## **MongoDB**

MongoDB — это кроссплатформенная документо-ориентированная система управления базами данных, которая работает на принципах коллекций и документов. Данная база данных дает высокую производительность, доступность и автоматическое масштабирование.

Вы спросите, зачем нам еще одна база данных, если уже и так существуют множество других? На этот вопрос, есть простой ответ. Все современные приложения нуждаются в быстрой обработке большого количества данных, в быстром создании функционала и быстрого и в быстром развертывании. Старые базы данных в этих вопросах некомпетентны, поэтому MongoDB была создана [43].

В случае MongoDB, база данных — это контейнер для хранения коллекций. Коллекция — это группа документов MongoDB, это эквивалент таблицы в реляционных базах данных. Коллекция существует только в одной базе данных. Документы в коллекции могут иметь разные поля. Как правило, все документы в коллекции имеют сходное или родственное назначение. Документ представляет собой набор ключ-значение. Документам присуще динамическая схема. Динамическая схема подразумевает, что документы в одной коллекции могут не иметь одинаковые поля и одну и ту же структуру, и общие поля в коллекции могут иметь разные типы данных [40].

Документ — это сердце базы данных MongoDB. Представление документа варьируется от языка программирования, но у каждого языка программирование есть свой способ его представить, таки как hash map, hash или словарь. В JavaScript, например, документ можно представить как объект: {"greeting»: "Hello, world!"}. Большинство документов на самом деле являются более сложной структурой. Все ключи в документе являются строкой, а также все ключи должны быть уникальными. Тем не менее, порядок ключей не важен, и мы не должны разрабатывать схему, чтобы зависеть от определенного порядка ключей, так как MongoDB может изменить их порядок [49].

Коллекции — это группа документов. Если документ — это аналог ряда в реляционной базе данных, то коллекция — это аналог таблицы. Коллекциям присуще динамическая схема, это означает, что документы в коллекции, могут иметь разное количество форм. Например, оба следующих вариант могут быть сохранены в одной коллекции:

{"greeting»: "Hello, world!"} {"foo»: 5}. Обратите внимание, что эти документы имеют не только разные типы данных, а также разные ключи. Так как любой документ может быть добавлен в любую коллекцию, напрашивается вопрос: “Почему мы должны разделить коллекции?”. Вот ответ: из-за отсутствия нужды отдельных схем для разных видов документов, почему мы должны использовать больше, чем одну коллекцию? Вот ряд причин почему [47]:

1. Работая с разными типами документов в одной коллекции, становится сложно с ней работать и это становится кошмаром.Разработчики должны убедиться, что каждый запрос возвращаем документы только определенного типа. Если мы запрашиваем посты в блоге, то трудно выбрать информации об авторе.
2. Быстрее всего получить список коллекций, чем получить список типов всех коллекций. Например, у нас есть поле “type” в каждом документе со следующими значениями: “skim,” “whole,” or “chunky monkey”. Так было бы намного медленнее найти все эти три значения в одной коллекции, чем имея три разные коллекции.
3. Группировка документов одного типа, вместе в одной коллекции позволяет локализовать данные. Получить несколько постов из коллекции, содержащей только посты затратит меньше ресурсов диска, чем поиск этих постов по автору.

В дополнение к группированию документов в коллекциях, MongoDB группирует коллекции в базы данных. Одна сущность базы данных может размещать несколько баз данных, каждая из которых группирует от нуля до нескольких коллекций. У базы данных есть свои разрешения, и каждая база данных хранится в отдельном файле на диске. Хорошее эмпирическое правило, хранить все данные для одного приложения в той же базе данных. Разделять базы данных полезно, когда хранишь данные для нескольких приложений или нескольких пользователей на одном сервере [44].

В данной работе для работы с MongoDB мы будет использовать специальную библиотеку для NodeJS - mongoose. Mongoose служит для моделирования объектов в базе данных MongoDB. С помощью этой библиотеки мы можем моделировать наши данные. Например, мы управляем магазином. У магазинов есть товары, а у каждого товара есть свои свойства,такие как: имя, идентификатор, цена и так далее. При помощи данной библиотеки мы можем смоделировать наши товары и так делать менеджмент наших моделей в базе данных, не думаю о деталях. Mongoose предоставляет абстракцию на уровне модели [50].

Для того чтобы подключиться к базе данных через Mongoose, нам нужно установить специальный пакет mongoose, используя пакетный менеджер npm Node.JS и вызвать функцию connect с нужными параметрами. Пример подключения к базе данных будет выглядеть следующим образом:

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/database\_name');

Давайте представим, что мы управляем интернет-магазином и нам нужно написать приложение с использованием Node.JS и Mongoose для управления товарами книжного магазина. Самая простая сущность для базы данных — это книга. В данном случае мы должны сделать модель книги. Самая простая схема книги будет выглядеть следующим образом:

const BookSchema = mongoose.Schema({

  name: String,

  price: Number,

  quantity: Number

});

Далее, чтобы использовать эту схему в приложении мы должны создать объект из этой схемы, выглядит это примерно так:

|  |
| --- |
| const book1 = new Book ({name: 'Introduction to Mongoose', price: 10, quantity: 25});  После вызова данной конструкции, создается документ в коллекции с данными из конструктора этого объекта [45].  Если сравнивать MySQL и MongoDB, то можно сказать следующее, что эти две базы данных одни из самых востребованных и конкурентоспособных сервисов баз данных для веб-приложений. Обе базы данных являются системами управления базами данных, которые позволяют извлекать данные, но оформлены они по-разному. MySQL — это система с табличной структурой, а MongoDB — это система, основанная на документах [51].  Что такое MySQL? SQL — это акроним от Structured Query Language (Структурированный Язык Запросов). MySQL разработана в 1995 году и стала структурой баз данных по умолчанию и получила широкое распространение. На протяжении двух десятилетий, этот язык запросов был основной моделей моделирования для разработки систем управления реляционными базами данных для поддержки данных и их хранения. Язык построен таким образом, что позволяет вводить, классифицировать, искать и извлекать данные любого типа. От корпоративных базы данных с большими данными, до небольшого сайта, такого как веб-сайта небольшого бизнеса [41].  База данных MongoDB была создана в 2007 году. Это был новый подход к дизайну баз данных. По мере роста цифровых сервисов и количества данных, появилась востребованность масштабирования такого количества данных, а также возможность более гибко управлять такими базами данных. MongoDB является ответом на потребность в гибкой, производительной базы данных. MongoDB используют для сайта электронной коммерции и обслуживания контента для сайтов, которые выигрывают своей гибкостью и масштабированию. Бизнесы используют базу данных MongoDB, как высокопроизводительное решение для быстрого обновления данных в системе [42].  Чтобы понять уникальные различия, сначала нужно разобраться со схожестью, а затем понять, чем отличаются из схемы базы данных [48].  Эти базы данных очень похоже между собой. По своей сути эти две базы данных представляют собой систему управления базами данных, которые являются фундаментом для любого цифрового пространства. Они хранят данные в компьютерной системе, как основа любого веб- приложения. Веб-драйверы для этих баз данных поддерживаются множеством языков программирования. В конце концов, любое веб-приложение нуждается в базе данных. Например, и MongoDB и MySQL позволяют распространять, модифицировать или развертывать облачные веб-приложения. Разработчики этих СУБД изначально создавали их базы данных с открытым исходном кодом, где любой может использовать их совершенно бесплатно. На этом их схожесть заканчивается, основная разница заключается в том, как эти базы данных были разработаны.  MongoDB — это система не реляционной базы данных, основанная на документах. Её также называют объектной ориентированной системой. Данная система была разработана для того, что заменить структуру MySQL, как более простой способ работы с данными. С другой стороны, MySQL — это табличная система. Табличный дизайн представляет структуру запроса базы данных для поиска и считается базой данных SQL. По мере роста потребностей в управлении данных и в росте объема данных в предприятиях, предприятия стали воспринимать MySQL, как более жесткую архитектуру, а не как более гибкую для переформатирования данных систему. Это мнение связано с дизайном на основе таблиц, который позволяет сайтам и приложениям применять конечный вариант многовариантных поисковых запросов.  MongoDB использует популярную модель управления доступом на основе ролей с гибким набором разрешений. Пользователям назначается роль, и эта роль предоставляет им определенные разрешения на наборы данных и операции с базой данных. Вся связь зашифрована с помощью TLS, и можно записывать зашифрованные документы в коллекции данных MongoDB. Протокол TLS (transport layer security) основан на протоколе SSL (Secure Sockets Layer), изначально разработанном в Netscape для повышения безопасности электронной коммерции в Интернете. MySQL поддерживает те же функции шифрования, что и MongoDB; его модель аутентификации также аналогична. Пользователям могут быть предоставлены роли, а также привилегии, дающие им разрешения на определенные операции с базой данных и на определенные наборы данных [50].  Преимущества MySQL  MySQL относится к корпоративному классу и эффективен на разных платформах и в сетях. Это лидер в этой области, и он продолжает создавать и выпускать сопоставимые варианты баз данных. Также известно, что он совместим с большим количеством систем из-за своего времени на рынке и мобильности. Поэтому он также рассматривается как масштабируемое решение. Благодаря сочетанию API, серверных, программных и административных инструментов и опций MySQL считается очень доступным и обычно работает без простоев [43].  Наконец, он создается с аутентификацией данных и безопасностью на уровне защищенных сокетов (SSL). Уровень безопасности делает конфиденциальные данные более защищенными при передаче.  Преимущества MongoDB  Использование MongoDB дизайна динамической схемы способствует созданию более гибкой среды для поиска данных, кодирования, интеграции и разработки баз данных.  Кроме того, он имеет поля, которые легко изменить, что позволяет пользователям избежать крупномасштабного пересмотра или повторной калибровки для меняющихся организационных потребностей или потребностей в данных [47].  Модель данных документа также предоставляет компаниям более сложные возможности хранения, доступа, индексирования и объединения любых типов данных как для моделей данных, удобных для кода, так и для собственных моделей данных. Таким образом, отображение преобразования не является необходимым для большей надежности и упрощения масштабирования [40]. **Аутентификация в одностраничных приложениях** Ключевая разница между Single Page application и мультистраничным сайтом, заключается в том, где хранятся данные о сессии - на сервере или в браузере. Другими словами, в мультистраничных сайтах используется statefull подход, а в Single Page Application используется stateless подход. Stateful аутентификация — это аутентификация, основанная на сессиях. Stateful аутентификация — это способ проверки пользователей, когда сервер или серверная часть хранит большую часть о сеансе, например свойства пользователей. В случае stateful аутентификации всякий раз, когда клиент отправляет запрос на сервер, сервер должен искать информацию о сеансе, такую ​​как свойства пользователя, и сопоставлять их с данными, которые были получены. Этот процесс включает в себя получение идентификатора пользователя, и сравнение со значительным объемом сохраненных данных аутентификации, которые есть у всех пользователей. В этой модели вы можете легко представить, что если память сеанса будет удалена на стороне бэкэнда, то идентификатор ссылки сеанса, который держит клиент, будет совершенно бессмысленным [4].  Положительные стороны:   * Отмена сеанса в любое время * Простота реализации и управление для сценария с одним сеансом и сервером * Данные сессии могут быть изменены позже   Отрицательные стороны:   * Увеличение нагрузки на сервер: чем больше количество зарегистрированных пользователей, тем больше ресурсов сервера занято * Трудность в масштабировании: если сессии расположены на разных серверах, нужно создавать определенный алгоритм, чтобы связать определенную сессию пользователя с определенном сервером. Это означает, что, например, если сессия Боба привязана к сервер Х, то тогда все следующие запросы Боба должны обрабатываться сервером Х. Для решения данной проблемы можно использовать прокси, до того, как запрос будет отправлен на сервер. Более того если сервер разворачивается с дублированием для обеспечения аварийного переключения, то задача усложняется поскольку два одноранговых узла, которые дублируют друг друга, должны реализовать алгоритм, который должен обеспечить согласованность этих сеансов [27].   Stateless аутентификация был создан для того, чтобы решить недостатки stateful аутентификации. Эти два способа разные и используются в разных сценариях. Stateless аутентификация хранит информацию о сессии пользователя в браузере. Так как информация о сессии пользователя находится на клиентской части, у сервера есть только одна возможность проверить валидность — это совпадает ли payload и подпись.  Положительные стороны:   * Снижение нагрузки на сервер: большое количество данных о сессиях не хранится на стороне сервера. Мы можем хранить больше пользовательских данных о сеансе на стороне клиента, чтобы уменьшить количество запросов к базе данных. * Простота масштабирования: поскольку данные хранятся на стороне клиента, не имеет значение какой внутренний сервер направляет запрос, если все внутренние серверы используют один приватный ключ, то все серверы имеют одинаковые возможности для проверки достоверности сеанса [10].   Отрицательные стороны:   * Невозможность отменить сеанс в любое время: поскольку сеанс пользователя хранится на стороне клиента – сервер не имеет прав на удаление сеанса * Данные о сессии не могут быть изменены до истечения срока действия сессии. Предположим, мы хотим добавить свойство возраст к данным сеансам, возможно мы попросим клиента обновить возраст, но не можем убедиться, что клиент действительно обновит возраст, поскольку его предыдущая сессия еще не истекла, тогда клиент до сих пор может отправлять запросы со старыми данными сеанса.   Разницу хранений сессий между мульти страничными приложениями и Single Page приложениями можно увидеть на рисунке 2.6: |
| **Рисунок 2.6 Разницу хранений сессий между мульти страничными приложениями и Single Page приложениями.** |

|  |
| --- |
| Rest API – широко используемая архитектура, которая в основном используется в связке Single Page Application. Одним из архитектурных ограничений дизайна является, то, что каждый запрос должен содержать все необходимые данные состояния приложения, чтобы изменить состояние ресурса сервера. “ST” в аббревиатуре REST – это Representational State Transfer – это относится к тому факту, что нужно постоянно передавать состояние между сервером и клиентом по протоколы HTTP [7].  Stateless аутентификация требует, чтобы данные были сохранены на стороне браузера. Ключевым компонентом является JWT токен, который хранит дынные об аутентификации, которые могут быть переданы конфиденциально между клиентом и сервером. В Single Page Applications разработчики обычно хранят JWT токен в local storage браузера и включают его в заголовок запроса или передают его в теле запроса.  JSON Web Token – это утверждение, которое представлено в кодировке JSON и может передаваться между двумя сторонами. JWT – это механизм проверки собственника определенных данных в формате JSON. По своей сущности, токен – это закодированная безопасная строка, которая может содержать неограниченное количество данных и криптографически подписана. Когда сервер получает токен, то он может гарантировать, что данному содержимому можно доверять, потому что токен был подписан источником. Ни один посредник не может изменить токен, после того как он был отослан. Очень важно отметить, что JWT гарантирует право собственности на данные, но не гарантирует шифрование. Данные, которые вы отправляете в JWT могут быть увидены кем угодно, кто перехватывает токен, потому что он просто сериализован, а не зашифрован. По этому причине строго рекомендуется использовать протокол HTTPS. JWT полезная технология для аутентификации REST API и аутентификации типа сервер-сервер. Почему JWT хорош для аутентификации? Он очень хорош для реализации, потому что [14]:   * Мы можем хранить любую информацию на клиенте * Сервер может доверять клиенту, потому что JWT токен подписан сервером. * Нет нужды в обращении к базе данных, чтобы получить дынные, так как все данные хранятся в токене * Нет нужды хранить данные о сессиях в базе данных, когда сталкиваемся с проблемой масштабирования   Очень распространённое использование JWT, и возможно одно из самых хороших – это механизм аутентификации с использованием REST API. JWT технология настолько популярно и широко используется, что даже Google использует этот механизм для того что аутентифицировать пользователя, когда мы используем их серверы. Идея очень проста: когда мы настраиваем доступ к API Google мы получаем секретный токен, таким образом Google нас аутентифицирует. Когда мы отсылаем запрос на API сервер, сервер знает, что это за клиент, потому что запрос был подписан уникальным идентификатором [31].  JWT Authentication API Request  **Рисунок 2.7 – Механизм аутентификации JWT**  Как же все-таки аннулировать токен? Одно из самых логичных решений – это изменить секретный ключ сервера, что сделает все токены недействительными. Однако это решение не идеальное для пользователей, у которых токены просто аннулируются без причины. Один из способов аннулировать токен – это добавить дополнительное свойство токену – это указать время и дату создания токена. Затем каждый раз, когда мы отправляем запрос, мы можем проверять это свойство и сравнивать с текущим датой и временем. Если срок годности токена прошёл, то мы можем смело аннулировать токен и не пустить пользователя в систему [12].  Архитектура REST API – хороша тем, что её логика проста и нет нужды в хранении сложных состояния в памяти, эта архитектура имеет дело с ресурсами (вместо того, чтобы иметь дело с функциями, которые не связаны между собой), которые позволяют сделать всю бизнес-логику связанной и простой. Что же случается после того, как пользователь отправляет свои данные? Откуда мы знаем если пользователь правильно зашел в систему или нет? Мы не можем хранить состояние пользователя на стороне сервера, чтобы сигнализировать об этом. Так что же мы можем сделать? [13]  Что же такое токены? JSON Web Tokens – это открытый и стандартный путь для то чтобы безопасно представить личность пользователя во время двустороннего взаимодействия. То есть, когда две системы обмениваются данными, мы можем использовать веб-токен для идентификации пользователя без необходимости отправки при ватных данных при каждом запросе. Если применить эту технологию в контексте REST API, то можно увидеть, как эти взаимодействия могут извлечь выгоду из этого механизма [26].  Вкратце, JWT работает следующим образом:  secure APIs with JWT process  **Рисунок 2.8 – Схема работы JWT**   * Пользователь отправляет запрос авторизации. Другими словами, это логин или юзернэйм с паролем, или другие любые данные. * После проверки, REST API создаст JSON Web Token и подпишет его с помощью секретного ключа * Потом REST API вернет этот токен на клиент * Клиентское приложение получает токен и сохраняет его в local storage, и продолжит использовать его в каждом последующем запросе для аутентификации пользователя, без необходимости больше отправлять свои учетные данные   Звучит достаточно просто, не так ли? Но насколько это безопасно? Давайте разбираться далее? Токен сам по себе это закодированная строка, которая состоит из трех частей, разделенных друг от друга точкой: header.payload.signature. Каждая часть содержит важную часть паззла. После декодирования первые две части будут представлены в формате JSON, содержащую соответствующую информацию, а последняя часть будет использоваться для проверки подлинности токена [20]:   * Header содержит данные связанные с типом токена и алгоритмом, который использовался при его создании, самые распространённые – это HS256 и RS256. Все зависит от уровня безопасности, который нам нужен. Это зависит от стандарта и мер, которые мы ищем. В этих двух примерах, один алгоритм использует приватный ключ и сервером, и клиентом, а второй использует приватный ключ сервера, в комбинации публичного ключа клиента. * Payload содержит данные, которые относятся к запросу и пользователя, который делает этот запрос.   Существуют и другие атрибуты, которые можно добавить к объекту payload, определённому как часть стандарта, но приведенные выше считаются более распространёнными. Конечно, мы можем использовать их или определить свои собственные, если клиент и сервер согласны с реализацией [34].  Signature – это закодированная строка, которая используется и клиентом и сервером для проверки подлинности payload. **Выводы главы** В данной главе мы подробно рассмотрели подробно элементы стека MEVN и принцип работы аутентификации на основе токенов. Подводя итоги данной главы, и рассмотрев все элементы стека MEVN, можно выявить сильные и слабые стороны.  Преимущества:   * Простота переключение между разработкой фронтенд или бэкэнд, так как все написано на одном языке программирования * Единый формат общения между всеми частями приложения. JSON используется для передачи данных, как и в Vue, так и в Node.JS, а также MongoDB позволяет сохранять данные в коллекции в формате JSON. * Лучшая рентабельность. Разработка на данном стеке требует разработчиков, знающий один язык программирования – JavaScript. Это означает, что уменьшается количество разработчиков в команде. * Сохранение времени. Позволяет сохранить время, так как в реестре Node.JS существуют готовые решения для функционала и не требуется создавать весь функционал с нуля. * Гибкость. Проще добавлять новый функционал по мере разработки приложения, чем когда оно уже закончено. * Поддержка сообщества по всему миру. JavaScript – это универсальный язык программирования, который используется практически абсолютно везде, значит можно ожидать помощи, если появятся вопросы от сообщества.   Недостатки:   * В высоконагруженных приложениях загрузка происходит медленно, так как передается большое количество данных, которые нужно обработать. * Single Page приложения не очень хорошо дружат с SEO, так как при сборке страницы у нас всего одна страница. Для устранения этого недостатка, нужно выучить новую технологии – SSR и Nuxt.JS * Отсутствует история браузера. Single page приложения не сохраняют историю браузера, когда пользователь прыгает между состояниями приложения, но эта проблема решается History API браузера. Это означает, что нужно писать чуть больше кода.   JavaScript зарекомендовал себя как один из самых популярных языков в мире с растущей экосистемой, который увеличил рост JavaScript разработчиков. В этой главе мы рассмотрели важные блоки для создания full stack веб приложения, используя всего один язык программирования – JavaScript, используя Node.JS, как среду для запуска backend, Express.JS -–как веб сервер, Vue.JS – как фреймворк для frontend и MongoDB – как базу данных. Все эти технологии формируют стек MEVN. Данный стек идеально подходит для малых бизнесов и для индивидуального использования. Стоить отметить, что обучиться данным технология не составляет большого труда, для того чтобы создавать быстрые и мощные веб приложения. Учитывая тот факт, что данный стек относительно молодой и у него дружелюбное сообщество, которое очень быстро развивается и у которого большое количество участников. У данного сообщества существует много ресурсов для обучения, так что, изучив данные технологии можно без труда найти себе работу. ГЛАВА III. РЕАЛИЗАЦИЯ ОДНОСТРАНИЧНОГО ВЕБ ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ MEVN В данной главе мы подробно рассмотрим, как создаются современные одностраничные веб приложения с использованием стека MEVN. Рассмотрим, как создать серверную часть при помощи Node.JS и Express.JS, также спроектируем модели для базы данных MongoDB с использованием библиотеки mongoose, которая позволяет упростить взаимодействие с базой данных. Далее затронем такие моменты, как создание визуальной части одностраничного приложения с использованием Vue, создадим глобальное хранилище, используя такую технологию, как Vuex. Настроем обмен данными между backend и frontend и реализуем аутентификацию на основе токенов JWT. В конце главы мы конвейеризируем одностраничное веб приложение, сделав так, чтобы backend и frontend запускались на разных, независимых контейнерах. **Проектирование базы данных MongoDB** В данном проекте мы будем проектировать базу данных MongoDB при помощи ORM – mongoose. ORM – (Object-Relational Mapping, объектно-реляционное отображение) – технология, главной целью которой является создание виртуальной базы данных. Для каждого языка программирования она своя. Для Node.JS одной из самых популярных является Mongoose. Mongoose является интерфейсом для MongoDB. Mongoose - это библиотека объектно-ориентированного программирования JavaScript, которая создает соединение между MongoDB и платформой веб-приложений Express.  Разработка схем в Mongoose начинается с создания схемы. Каждая схема прикреплена к коллекции в базе данных и определяем форму документов в этой коллекции. В нашем проекте можно выявить три модели: Статья, Продукт и Пользователь. В каждой модели есть свои специфичные атрибуты. Давайте рассмотрим каждую модель, а так же, как она создается. Начнем с модели статьи. Для того, чтобы создать модель, нам сначала нужно импортировать в файл определения модели библиотеку Mongoose. Далее следует определиться какие атрибуты будут у Статьи. Я выявил следующие: заголовок, дата, картинка, текст статьи и дата создания статьи. В коде это выглядит следующим образом:    **Рисунок 3.1 – Схема модели статьи**  Основными атрибутами данной модели являются заголовок с типом строка, дата с типом строка, картинка – тип строка, тут будет храниться адрес картинки из нашего сервера, содержимое статьи – тоже тип строка, и дата создания статьи с типом дата. Так же указываем метаданные, что коллекция будет называться posts, и чтобы дата в поле создания изменялась автоматически. Далее нам нужно назвать модель и привязать к схеме, потом нам нужно экспортировать нашу модель, чтобы мы могли использовать ее в другим частях приложения.  Следующая модель нашего приложения — это модель продуктов. У продуктов будут следующие атрибуты: заголовок, картинка, цена, описание, категория, рейтинг, размер, код продукта и дата создания. Имя коллекции будет products, и подключим возможность автоматического обновления даты создания продукта. Далее привязываем модель к схеме и экспортируем эту модель для дальнейшего использования. Подробную схему продукта можно увидеть на следующем рисунке:    **Рисунок 3.2 – схема модели продуктов**  Последняя модель, которую мы рассмотрим – это модель пользователя. Принцип построения модели точно такой же, как и у первых двух. К атрибутам этой модели, можно отнести следующие: username, пароль, электронную почту, заказы пользователя, имя и фамилия пользователя, его адрес, мобильный номер телефона, и дата создания пользователя. Имя коллекции – Users, и также добавляем автоматическое автозаполнение даты создания пользователя. Привязываем модель к схеме и экспортируем модель. Подробно схему можно увидеть на следующем рисунке:    **Рисунок 3.3 – схема модель пользователя**  У данной модели тип для username – строка, для пароля тоже тип строка, так же, как и у мейла, заказы пользователя – представлены в виде списка, изначально список пустой, имя и фамилия пользователя – тоже являются строками, адрес – это сложный объект, у которого имеются следующие поля: улица, город, почтовый индекс и страна, у которых изначально значение null.  На этом наши модели заканчиваются, конечно, по мере развития проекта модели тоже могут развиваться, а используя библиотеку mongoose и mongodb изменения делаются легко, быстро и без проблем. Далее данные модели будем использовать при построение серверной части нашего приложения. **3.2 Создание REST API сервера** Так как модели у нас уже спроектированы и созданы, можно смело приступать к созданию REST API сервера. REST API – это такой же веб сервер, как и у обычных серверов веб приложений, просто отличается по архитектуре. Для начала нужно разобраться с архитектурой проекта нашей серверной части. В нашем проекте будут использоваться следующие дополнительные библиотеки: bcryptjs – для хэширования паролей, body-parser – для того, чтобы обработать тело запроса, cors – для того, чтобы, слать кросс доменные запросы, express – для создания веб сервера, jsonwebtoken – для создания токена, mongoose – для управления базой данных и nodemailer – для отправки мейлов. В конечном итоге, структура проекта выглядит следующим образом:    **Рисунок 3.4 – структура проекта серверной части**  В папке models – хранятся модели, в папке controllers – хранится вся логика приложения и взаимодействие с моделями. В этой папке контроллер для каждой модели вынесен в отдельный файл, так же хранится контроллер для отправки мейла из формы контактов. В папке node\_modules - хранятся все зависимости проекта, то есть все установленные библиотеки, не стоит забывать, что некоторые библиотеки используют и другие библиотеки, поэтому список зависимостей увеличивается. В папке public хранятся картинки для продуктов и для постов. В папке routes хранятся все url адреса, куда можно сделать запрос, чтобы получить данные. В этом месте связываются url адреса и методы контроллеров. Для каждого контролера, я создал отдельный файлик в этой папке, чтобы в дальнейшем было проще управлять проектом, по мере его развития. В папке util находятся функции, которые являются вспомогательными и те, которые часто используются. В той папке находится файлик с подключением к базе данных и её конфигурацией. Этот файлик выглядит следующим образом:    **Рисунок 3.5 – подключение к базе данных**  Файлик. gitignore относится к управлению контроля версий приложение и к репозиториям. В данном проекте это рассматриваться не будет. Если говорить вкратце, то программное обеспечение контроля версий отслеживает все вносимые в код изменения в специальной базе данных. При обнаружении ошибки разработчики могут вернуться назад и выполнить сравнение с более ранними версиями кода для исправления ошибок, сводя к минимуму проблемы для всех участников команды.  В файле Dockerfile описана вся конфигурация для докера, чтобы сделать контейнер из нашей серверной части, мы рассмотрим его в последней части данной главы.  Package-lock. json если коротко, то предназначен для блокировки зависимостей от определенного номера версии. В package-lock. json файле перечислены зависимости вашего приложения и зависимости всех его зависимостей. Другими словами, он описывает, какую версию каждого отдельного пакета вы установили.  Файл server.js – это начальная точка запуска проекта. Рассмотрим его структуру:    **Рисунок 3.6 – содержимое файла server.js**  В данном файле мы импортируем все нужные библиотеки и нужные функции, такие как подключение к базе данных. Создаем объект express и сохраняем его в переменную, что и будет нашим сервером, а также создаем переменную, где будем хранить порт, на котором будет запускаться наш сервер. Далее мы подключаем body-parser, чтобы он работал каждом запросе, а также подключаем cors – чтобы не было ошибок, когда будем отправлять запросы с домена на домен. Следующее, что нам нужно сделать, это указать, что наша папка images является публичной, чтобы оттуда можно было доставать файлы, без этих строчек, файлы будут недоступны. Далее мы подключаем все наши роуты и подключаемся к базе данных. Последним этапом мы указываем, чтобы сервер слушал наш порт и при запуске команды node server.js, наш сервер успешно откроется.  Рассмотрим какие маршруты доступны в нашем проекте, для того чтобы знать на какие адреса нам отправлять запросы. Рассмотрим, как выглядит классический файл маршрутов на примере файла contact.js. Содержимое файла contact.js можно увидеть на рисунке ниже. |
| **Рисунок 3.7 – пример составления маршрута для запроса.**  В данном файле мы подключаем библиотеку Express.JS, а также подключаем контроллер, где хранится логика, как будет отправляться мэйл формы контактов. Далее мы получаем сущность роутера, и вызываем метод post, тем самым говорим, что мы планируем отправлять данные, а в параметрах этого метода указываем путь, по которому будет доступен этот маршрут, в нашем случае – это «/contact», а вторым параметром указываем контроллер, который отвечает за отправку мэйла. В итоге мы должны экспортировать роутер, чтобы мы могли использовать его в файле server.js. Все остальные маршруты построены по такому же принципу. Давайте рассмотрим какие маршруты доступны в нашем приложении:   * “/contact” – маршрут для отправки мэйла формы контактов * “/fetch-posts” – маршрут для получения все постов для блога * “/fetch-posts/:id” – маршрут для получения одного поста по идентификатору * “/fetch-related-posts” - маршрут для получения связанных постов * “/create-post” - маршрут для создания постов * “/fetch-products” – маршрут для получения всех продуктов * “/fetch-products/:id” – маршрут для получения одного продукта по идентификатору * “/fetch-related-products” – маршрут для получения связанных продуктов * “/fetch-category-products” – маршрут для получения продуктов по категории * “/create-product” – маршрут для создания продуктов * “/get\_latest\_products” – маршрут для получения 8 последних продуктов   Рассмотрим классический пример контроллера в нашем проекте, на примере контроллера для отправки формы обратной связи. Выглядит он следующим образом: |
| **Рисунок 3.8 – пример контроллера отправки мэйла с формы контактов**  В данном контроллере мы импортируем библиотеку nodemailer для отправки мэйлов. Далее мы описываем функцию для отправки мейла, внутри которой мы описываем конфигурацию. В конфигурации мы указываем почтовый сервис, который мы используем, в нашем случае – это Gmail, далее мы указываем логин и пароль от почтового ящика. Затем мы создаём объект письма, которое должно прийти на наш почтовый ящик, где указываем от кого, придет письмо – эти данные мы берем из тела запроса, так же указываем кому должно прийти письмо, тему письма мы тоже берем из тела запроса и само содержимое письма. Далее мы непосредственно создаем функцию, которая отправляет письмо. В качестве параметром данной функции мы передаем наш объект письма и передаем колбэк, для обработки, где мы проверяем на наличие ошибок. Таким образом выглядит самый простой контроллер в нашем приложении. Для того, чтобы посмотреть, как используются модели в контроллерах, может рассмотреть контроллер регистрации пользователя. Он выглядит следующим образом: |
| **Рисунок 3.9 – контроллер регистрации пользователя**  В данном контроллере мы импортируем модель пользователя, библиотеки bcryptjs и jsonwebtoken. Далее мы создаем функцию registerUser и сразу экспортируем ее, чтобы у нас был доступ в маршруте. Стоит отметить, что любой контроллер в качестве параметром принимает три параметра – request, response, next – опциональный параметр. Далее мы должны получить из тела запроса следующие параметры: username, email, password, где сохраняем их в переменные. Затем мы должны проверить если существует такой пользователь на основе входящих данных. Пишем проверку, если длинна полученных данных больше, чем ноль, то тогда мы отправляем ответ со статусом 409 – конфликтное обращение к ресурсу и сразу выходим из метода при помощи ключевого слова return. Если такого пользователя не существует, то мы можем пойти дальше и создать объект нового пользователя из данных, которых мы получили из запроса. Сформировав объект, мы можем успешно создать нашего пользователя, используя модель пользователя, а затем мы сохраняем пользователя в базе данных, и если все прошло успешно, то мы отправляем статус 200.  Для того чтобы разобрать каждый из контроллеров более подробно, потребуется очень много времени, поэтому в данной части я рассмотрел на примере одного контроллера, но структура у них одна и таже, просто они решают разные задачи бизнес-логики. Перейдем к созданию фронтенда с использованием Vue и Vuex.  Чтобы сделать контейнер из нашего кода для серверной стороны, нам достаточно создать Dockerfile в папке проекта, и прописать следующих команд, которые представлены на следующем рисунке:    **Рисунок 3.10 – конфигурация для создания контейнера для сервера**  Из данной конфигурации мы можем увидеть, что сначала мы берез из репозитория Docker образ Node.JS, создаем рабочую директорию server копируем из нашего проекта файл с зависимостями, а затем их устанавливаем, затем копируем файлы из нашего проекта в рабочий каталог, освобождаем порт 5000 и запускаем наш сервер. **Создание фронтенда с помощью Vue и Vuex** Структура проекта Vue в нашем проекте состоит из следующих файлов: папка node\_modules. В этой папке хранятся все зависимости и библиотеки нашего проекта, папка паблик, где хранится иконка нашего приложения и один файл HTML. |
| **Рисунок 3.11 – структура файла index.html**  В этом файле мы можем увидеть классическую структуру файла формата HTML, но есть два важных отличия. Эти отличия заключаются в том, что используется тэг noscript, он срабатывает, когда JavaScript в браузере отключен, а также в том что на странице присутствует всего один блок с идентификатором app, вся остальная часть отображается динамически через JavaScript. Файл. browserslistrc — это тот инструмент, с помощью которого можно описать целевые браузеры веб-приложения, используя простые выражения, например, как в нашем случае:> 1%, last 2 versions, not dead. Этот пример файла. browserslistrc означает, что вам нужны: браузеры за последние два года, плюс браузеры, у которых больше 1% пользователей, и все эти браузеры должны быть «живыми». ESLint — инструмент статического анализа кода для выявления проблемных шаблонов, в файле. eslintrc – хранится его конфигурация. В файлах babel.config.js и tsconfig.json хранится конфигурация для JavaScript и для TypeScript – эти файлы генерируются автоматически. Dockerfile хранится описание того, как должен создаваться контейнер для фронтенда нашего приложения. Самое интересное хранится в папке src. В этой папке находится наш главный компонент, с которого начинается наше приложение. В этом файле, мы подключаем все дополнительные файлы стилей, шрифтов и файлов JavaScript. В отличии от привычного способа подключения скриптов, в проекте Vue нужно их подключать динамически в методе жененного цикла mounted. Интересно, что шаблон данного компонента состоит всего лишь из одного тэга - <router-view/>, куда непосредственно будут подгружаться и заменяться компоненты. В папке assets также хранятся дополнительные стили, а также, картинки и дополнительные скрипты. В папке components хранятся все наши компоненты, их очень много (40 штук) и если их рассматривать, то наш диплом превратится в книгу, поэтому предлагаю рассмотреть в данной работе несколько штук, где происходит взаимодействие с Vuex. Сделаем это после того, как рассмотрим подобротнее действия, мутации, гетеры и состояния нашего приложения. Следующая папка – это папка с маршрутами. В нашем проекте доступны следующие маршруты:   * “/home” – наша главная страница приложения, где будет отображаться компонент Home * “/about” - страница About, где будет отображаться компонент About * “/contact” – страница контактной формы, где будет отображаться компонент Contact * “/account-login” – страница Логина, где будет отображаться компонент AccountLogin * “/account-register” – страница Регистрации, где будет отображаться компонент AccountRegister * “/shop-cart” – страница корзины, где будет отображаться компонент Cart * “/shop-checkout” – страница проверки заказа, где будет отображаться компонент Checkout * “/shop” – страница магазина, где будет отображаться компонент Shop * “/account” – страница аккунта, где будет отображаться компонент Account, у которого есть несколько дочерних элементов: Dashboard, Orders, PaymentsMethod, Address, AccountDetails * “/blog” – страница блога, где будет отображаться компонент Blog * “/blog-details/:id” – страница одного поста, где будет отображаться компонент BlogDetails * “/shop-product/:id” – страница одного продукта, где будет отображаться компонент ProductItem * “/order-description/:id” – страница одного заказа, где будет отображаться компонент OrderDescription * “/not-found(.\*)” – страница блога, где будет отображаться компонент NotFound   Стоит отметить, что некоторые из маршрутов должны быть защищены, от тех пользователей, которые не авторизированы. К этим маршрутам можно отнести следующие: “/shop-cart”, “/shop-checkout”, “/account”. Проверкой занимается отдельная функция:    **Рисунок 3.12 – функция для проверки если пользователь авторизирован**  В этой функции мы проверяем если в local storage хранится ключ ‘user’, где мы храним данные о пользователе, включая токен, если он присутствует, то меняет значение переменной isAuthenticated – на значение true и позволяем пользователю пройти дальше, если нет, то следующий маршрут, по которому он пройдет будет “/account-login”, а на страницу аккаунта он не пройдет.  Наше глобальное хранилище будет разделено на модули, чтобы было проще им управлять и расширять код в дальнейшем. Хранилище будет разделено на следующие модули: postsModule, productsModule, cartModule, userModule.  В модуле postModules у нас в состоянии хранятся следующие атрибуты: список постов, выбранный модуль и связанные посты, а также мутации, чтобы изменить состояние постов, когда мы будем получать данные с сервера, геттеры – для того, чтобы кусочки нужного нам состояния, а также действия – для того, чтобы вызвать мутацию. Мутаций у нас всего три – для каждого кусочка состояния своя. Геттеров всего четыре штуки – для получения постов, для получения выбранного поста, для получения связанных постов и для проверки, если посты вообще пришли после совершения запроса. Действий в этом модуле тоже всего три, для загрузки всех постов, для загрузки одного поста и для загрузки связанных постов. В общей сложности действие для получения всех постов выглядит следующим образом: |

**Рисунок 3.13 – действие для получения всех постов**

В данном действии мы принимаем контекст этого действия, посылаем запрос на маршрут ‘http://localhost:5000/fetch-posts’, превращаем ответ от запроса в формат JSON, проверяем если не произошла ошибка, если произошла, то мы выбрасывает ошибку, а если нет, то вызываем мутацию, при помощи метода commit, куда передаём название мутации “loadPosts”, а также передает наш ответ от сервера в формате JSON. После этого, когда мы вызовем это действие из шаблона, то состояния должно измениться и данные динамически должны появиться в шаблоне. В нашем случае это выглядит вот так:



**Рисунок 3.14 – вызов действия из шаблона**

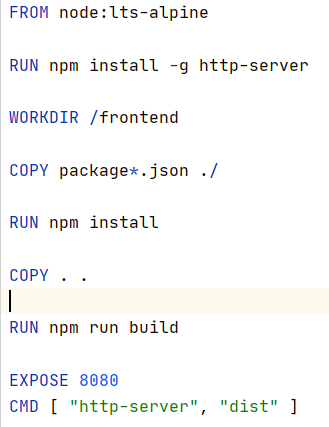
На данном рисунке представлен компонент Blog и секция, где выводится динамически компонент BlogItem, со свойствами key, id, date, title, image. Далее в компоненте жизненного цикла mounted, у глобального объекта $store мы вызываем метод dispatch, в который мы передаем имя нашего действия, в формате имя модуля / имя действия. После того, как мы отправили действие и состояния изменилось, нам нужно вызвать геттер и сохранить его в переменную posts. И теперь, используя директиву v-for мы можем через цикл отрисовать на странице компонент BlogItem, а те параметры, которые передались в цикле, будут доступны в компоненте BlogItem. Таким образом взаимодействуют Vue и Vuex, согласитесь, что довольно-таки просто.

В модуле productsModule хранилище состоит из следующих атрибутов: списка продуктов, одного выбранного продукта, списка связанных продуктов и список последних добавленных продуктов. Из мутаций у нас получение всех продуктов, получение одного продукта, получение связанных продуктов, получение последних добавленных продуктов, поиск продуктов, фильтрация по цене, фильтрация по цене и фильтрация по последним добавленным продуктам. Из геттеров у нас следующие: геттер на получение всех продуктов, на получение количества продуктов, существуют ли вообще продукты, получение одного продукта, получение связанных продуктов и получение всех последних продуктов. Из действий у нас предусмотрены следующие: загрузка всех продуктов, загрузка одного продукта, загрузка связанных продуктов, загрузка последних добавленных продуктов, поиск продуктов, фильтрация по цене, фильтрация по популярности, фильтрация по последним добавленным и сортировка по цене.

В модуле cartModule хранилище состоит всего из одного атрибута – это список товаров в корзине. Из геттеров, нам доступны следующие: список всех товаров, получение количества всех товаров, существуют ли товары вообще и динамический подсчет всех товаров в корзине. Из мутаций существуют следующие – загрузка всех товаров, добавление товара в корзину, удаление товара из корзины и сброс корзины. Действия нам доступны следующие – добавление товара в корзину, удаление товара из корзины, сохранение заказа и сброс корзины.

Перед тем как перейти к последнему модулю userModule, стоит обратить внимание на тот момент, что все модули имеют атрибут namespaced – это значит, что они инкапсулированы в своих модулях и могут работать автономно друг от друга, в случае с последним модулем пользователя, мы хотим, чтобы этот модуль был виден в рамках всего приложения, поэтому у него его нет. В этом модуле состояние состоит из следующих атрибутов: самого токена, где мы проверяем если оно уже нашем браузере, то мы берем значение из него, а если нет, то по умолчанию у него значение null, далее атрибут полного описания пользователя, список заказов нашего пользователя и выбранный заказ. Из геттеров нам доступны следующие: получение токена, получение username пользователя, проверки если есть ли токен у юзера, получение идентификатора юзера, получение полной информации о пользователе, проверка если пользователь прошел аутентификацию, получение списка заказов пользователя, получение выбранного заказа и подсчет стоимости всех заказов. Из мутаций нам доступны следующие: логин юзера, выход пользователя из системы, где мы обнуляем localstorage и убираем токен из хранилища, получение полной информации о пользователе, загрузка всех заказов и загрузка определенного заказа. Из действий нам доступны следующие: регистрация пользователя, логин пользователя, выход пользователя из системы, загрузка полной информации о пользователе, получение всех заказов и получение конкретного заказа.

Для виртуализации нашего фронтенд приложения в Dockerfile мы должны прописать следующую конфигурацию:



**Рисунок 3.15 – конфигурация для создания контейнера для фронтенда**

В данной конфигурации мы можем увидеть, что она немного отличается от конфигурации серверной стороны. Мы также же берем образ Node.JS из репозитория Docker, здесь мы глобально устанавливаем зависимость http-server для того, чтобы этот сервер отображал наше приложение, так как обычный браузер не сможет, создаем рабочую директорию копируем туда наш файлик с зависимостями и устанавливаем эти зависимости, далее копируем файлы нашего проекта, и вызываем команду сборки нашего проекта, освобождаем порт 8080 и запускаем наш сервер через команду: “http-server dist”, то есть в итоге папка dist должна открыться на сервере

* 1. **Выводы главы**

Разработав данное приложение, я понял, что очень удобно создавать веб приложение используя всего один язык программирования. Также, Node.JS играет важную роль в стеке технологий. Это первый и лучший выбор для создания интерактивных приложений. Независимо от того, какой язык вы используете для бэкэнда, вам все равно понадобится JavaScript для фронтэнда, поэтому вместо того, чтобы тратить время на изучение серверного языка, такого как Php, Java или Ruby on Rails, вы можете потратить все свои усилия на изучение JS и освоение в нем. Javascript одного и того же языка может использоваться как на стороне клиента, так и на стороне сервера. Таким образом, разработчик, который знает JS, может выступать в роли разработчика полного стека без изучения дополнительных языков. NodeJS построен на движке Chrome V8 от Google. Это позволяет Node предоставлять среду выполнения на стороне сервера, которая компилирует и выполняет JavaScript с молниеносной скоростью. Движок V8 компилирует JavaScript в машинный код вместо того, чтобы интерпретировать его или выполнять как байт-код, и это делает Node действительно быстрым. Облегченный Javascript достигает высокой производительности с меньшим количеством строк кода по сравнению с Java или C.

Только несколько языков программирования предлагают богатую экосистему пакетов, как NodeJS. Когда вы устанавливаете Node.js, он автоматически устанавливает программы NPM (Node Package Manager). Любой разработчик Node.js может упаковать свои библиотеки и решения в модуль, который каждый может установить с помощью официального менеджера пакетов Node, NPM. Тысячи библиотек и инструментов для разработки Javascript собраны на NPM. С использованием Node.JS легко отправлять и синхронизировать данные между серверной и клиентской кодировкой. Из-за использования одного и того же языка Javascript с обеих сторон ваш исходный код будет чище и последовательнее. Вы будете использовать то же соглашение об именах, те же инструменты и те же лучшие практики. Время разработчиков экономится в большей степени благодаря этой функции.

Касательно Vue.JS хочется сказать по итогу, меня очень зацепила простота и лаконичность этого инструмента, он очень много берет на себя, давая нам максимально комфортную работу, при этом абсолютно не проседая в производительности.

Также хочется заметить, что приложение создавалось быстро и с минимальными трудностями, поэтому рекомендую всем попробовать этот стек. Данный проект еще полностью не дописан, его разработка еще впереди, но я могу сказать, что оставшуюся часть проекта, я допишу без проблем в короткие сроки.

# ВЫВОДЫ

Веб разработка на данный момент очень быстро развивается, я бы даже сказал, что молниеносно. Очень часто появляются новые инструменты, которые хочется внедрить скорее в проект, но стоит быть внимательным. Перед тем как внедрить ту или иную технологию, ее нужно как следует протестировать и понять ее сильные или слабые стороны, вдруг она вообще не подойдет для проекта и сильно повлияет на производительность приложения или занесет новые дефекты в проект.

JavaScript зарекомендовал себя как один из самых популярных языков в мире с растущей экосистемой, который увеличил рост JavaScript разработчиков. В данной работе мы рассмотрели относительно молодой и производительный стек MEVN, который в последнее время очень хорошо себя зарекомендовал, сообщество которого развивается с каждым днем. Данный стек идеально подходит для малых бизнесов, а также для более крупных проектов, поэтому я считаю, что данный стек станут применять все большее количество бизнесов.

Говоря про Docker, хочется сказать, что он является одним из важных инструментом для современного разработчика. Он обладает хорошей документацией и обеспечивает быстрое обучение. Docker хорошо подходит для следующих задач:

* Удобная передача серверного проекта клиенту, ему не нужно настраивать сервер под определённые нужды проекта, так как серверное приложение остается инкапсулировано в контейнере.
* Обеспечение тиражируемости серверов
* Обеспечение преиспользуемости ранее созданных конфигураций
* Идеально подходит и для локального тестирования

Таким образом, анализируя современные технологии, можно сказать, что рынок веб технологий очень быстро растет, создавая новые рабочие места. Новые технологии открывают достаточно большие возможности для новых форм организации труда и занятости, а также не заставят вас скучать, так как всегда будет что изучить новое и быть ценным на рынке. В тоже время, выбор правильных технологий для проекта на текущий момент является самой актуальной задачей сегодняшнего дня.

# БИБЛИОГРАФИЯ

1. Алекс Янг, Брэдли Мек, Майк Кантелон. *Node.JS в действии*. Россия: Питер, 2018. ISBN 978-5-496-03212-4
2. Брэд Дэйли, Брендан Дэйли, Калеб Дейли. *Разработка веб-приложений с помощью Node.JS, MongoDB*. Россия: ДМК Пресс, 2020. ISBN 978-5-6040044-8-7
3. Морно Коскиаро, Лучиано Маммионо. *Шаблоны проектирования Node.JS*. Россия: ДМК Пресс, 2017. ISBN 978-5-97060-485-4
4. Хрусталев Александр Адонисович, Кириченко Андрей Валентинович. *HTML5 + CSS3. Основы современного WEB-дизайна.* Россия: Наука и Техника. ISBN 978-5-94387-750-6
5. Арно Лоре. *Проектирование веб-API*. Россия: ДМК Пресс, 2019. ISBN 978-5-97060-861-6
6. Робин Ноблен, Керри-Лэй Греди. *Эффективный веб-сайт*. Россия: Триумф, 2004. ISBN 5-89392-084-8
7. Бен Фаррелл. *Веб-компоненты в действии.* Россия: ДМК Пресс, 2020. ISBN 978-5-97060-856-2
8. Этан Драун. *Изучаем JavaScript. Руководство по созданию современных веб-прилоэений.* Россиия: Диалектика-Вильямс, 2022

Кириченко Андрей Валентинович. *JavaScript для фронтенд разработчиков. Написание, Тестирование, Развёртывание.* Россия: Наука и Техника, 2020. ISBN 978-5-94387-789-6

1. Энди Бадд, Камерон Молд, Саймон Коллинзон. *CSS. Профессиональное применение веб стандартов.* Россия: Вильямс, 2009. ISBN 978-5-8459-1199-5, 1-59059-614-5
2. Марейн Хавербеке. *Выразительный JavaScript.* Россия: Питер, 2019. ISBN 978-5-4461-1226-5
3. Мэтт Фрисби. JavaScript для профессиональных разработчиков. Россия: Питер, 2020. ISBN 978-5-4461-1740-6
4. Иан Милл, Эйдан Хобсон Сейерс. *Docker на практике*. Россия: ДМК Пресс, 2019. ISBN 978-5-97060-772-5
5. Малышев Константин Викторович. *Построение пользовательских интерфейсов.* Россия: ДМК Пресс, 2021. ISBN 978-5-97060-962-0
6. Русак Александр Михайлович, Храмцов Павел Брониславович. *Основы веб технологий.* Россия: Интернет-Университет информационных технологий, 2003. ISBN 5-9556-0001-9
7. Павел Козловский, Питер Бэкон Дарвин. *Разработка одностраничных веб приложений.* Россия: ДМК Пресс, 2020. ISBN 978-1-78216-182-0
8. Nilson Jacques. *Jump Start Vue.JS.* United Kingdom: SitePoint, 2014. ISBN 978-1925836448
9. Callum Macrae. *Vue.js: Up and Running: Building Accessible and Performant Web Apps*. Массачусетс: O’Reilly, 2021. ISBN 978-1491997246
10. Chris Minnik, Nat Dunn. *Actionable Vue.JS.* United Kingdom: Packt publishing, 2021. ISBN 1951959019
11. Lionel Lopez. *Vue: Step-By-Step Guide to Mastering Vue.js From Beginner to Advanced.* United Kingdom: SitePoint, 2017. ISBN 1976214386
12. Brett Nelson. *Getting to Know Vue.js: Learn to Build Single Page Applications in Vue from Scratch.* *Нью-Йорк*: Apress, 2022. ISBN: 978-1484237809
13. Marina Mosti. *Building Forms with Vue.js: Patterns for building and scaling complex forms with great UX.* United Kingdom: Packt publishing, 2022. ISBN: 978-1839213335
14. Heitor Riberio. *Vue.js 3 Cookbook: Discover actionable solutions for building modern web apps with the latest Vue features and TypeScript. Нью-Йорк*: Apress, 2022. ISBN 183882622X
15. Greg Lim, Daniel Correa*. Beginning Vue Stack: Build and Deploy a Full Stack MongoDB, Express, Vue.js, Node.js App.* United Kingdom: Packt publishing, 2021. ISBN 978-1-78883-114-7
16. Ajdin Imsirovic. *Vue.js Quick Start Guide: Learn how to build amazing and complex reactive web applications easily using Vue.js.* United Kingdom: Packt publishing, 2020. ISBN 978-1789344103.
17. Philippe Hong. *Practical Web Design: Learn the fundamentals of web design with HTML5, CSS3, Bootstrap, jQuery, and Vue.js.* Packt publishing, 2018. ISBN 1788395034.
18. Andrea Koutifaris. Vuex Quick Start Guide: Centralized State Management for your Vue.js applications. Packt publishing, 2020. ISBN 978-1788999939.
19. Basarat Syed. Beginning Node.js. *Нью-Йорк*: Apress, 2020. ISBN 978-1484201886
20. Sandro Pasquali. Mastering *Node.js - Second Edition: Build robust and scalable real-time server-side web applications efficiently.* United Kingdom: Packt publishing, 2019. ISBN 978-1785888960.
21. Andrew Mead. *Learning Node.js Development: Learn the fundamentals of Node.js, and deploy and test Node.js applications on the web.* United Kingdom: Packt publishing, 2020. ISBN 978-1788395540.
22. Andrew Mead. *Advanced Node.js Development: Master Node.js by building real-world applications.* United Kingdom: Packt publishing, 2021. ISBN 978-1788393935.
23. Marc Harter. *Node.js in Practice.* New York: Manning Publications, 2020. ISBN 978-1617290930
24. Maya Shavin, Philip Kirkbride, Clifford Gurney, Hugo Di Francesco*. Front-End Development Projects with Vue.js: Learn to build scalable web applications and dynamic user interfaces with Vue*. United Kingdom: Packt publishing, 2021. ISBN ISBN: 978-1838984823
25. Sean P. Kane. *Docker: Up & Running: Shipping Reliable Containers in Production.* New-York: O’Reilly, 2022. ISBN 978-1492036739
26. Arun Kumar. Docker: A Complete Guide. India: Arun, 2021. ISBN B087SG2H7Y
27. Richard Bullington-McGuire, Andrew K. Dennis. *Docker for Developers: Develop and run your application with Docker containers using DevOps tools for continuous delivery.* United Kingdom: Packt publishing, 2020. ISBN 1789536057
28. Engy Fouda. *A Complete Guide to Docker for Operations and Development: Test-Prep for the Docker Certified Associate.* Нью-Йорк: Apress, 2020. ISBN 978-1484281161
29. Brayden Smith. DOCKER: *A Step-by-Step Guide to Learn and Master Docker. . Нью-Йорк*: Apress, 2022. ISBN 1083161709
30. Saibal Ghosh. Docker Demystified: *Learn How to Develop and Deploy Applications Using Docker (English Edition)*. ISBN 978-9389845877
31. Stephen Kuenzli, Jeffrey Nickoloff. *Docker in Action.* New York: Manning Publications, 2021. ISBN 978-1617294761
32. Shannon Bradshaw. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. Массачусетс: O’Reilly, 2021. ISBN 978-1491954461
33. Amit Phaltankar. *MongoDB Fundamentals: A hands-on guide to using MongoDB and Atlas in the real world.* United Kingdom: Packt publishing, 2021. ISBN 978-1839210648
34. Rick Copeland*. MongoDB Applied Design Patterns: Practical Use Cases with the Leading NoSQL Database.* Массачусетс: O’Reilly, 2022. ISBN 978-1449340049
35. Alex Giamas. *Mastering MongoDB 4.x: Expert techniques to run high-volume and fault-tolerant database solutions using MongoDB 4.x.* United Kingdom: Packt publishing, 2021. ISBN 978-1789617870
36. Eelco Plugge. *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB.* Нью-Йорк: Apress, 2022. ISBN 978-1484211830
37. Manu Sharma. *Full Stack Development with MongoDB: Covers Backend, Frontend, APIs, and Mobile App Development using PHP, NodeJS, ExpressJS, Python and React Native (English Edition).* United Kingdom: Packt publishing, 2021. ISBN 978-9355510143
38. Ajit Singh, Sultan Ahmad. *MongoDB Simply in Depth*. Нью-Йорк: Apress, 2019. ISBN 1701055929
39. Mithun Satheesh, Bruno Joseph D'mello, Jason Krol. *Web Development with MongoDB and NodeJS.* United Kingdom: Packt publishing, 2021. ISBN 978-1785287527
40. Subhashini Chellappan, Dharanitharan Ganesan. *MongoDB Recipes: With Data Modeling and Query Building Strategies.* Нью-Йорк: Apress, 2018. ISBN 978-1484248904
41. Nicholas Cottrell. *MongoDB Topology Design: Scalability, Security, and Compliance on a Global Scale.* Нью-Йорк: Apress, 2019. ISBN 978-1484258163
42. David Hows, Eelco Plugge, Peter Membrey, Tim Hawkins. *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB.* Нью-Йорк: Apress, 2019. ISBN 978-1430258216
43. Daniel Perkins. MongoDB: *Master MongoDB with Simple Steps and Clear Instructions.* Нью-Йорк: Apress, 2020. ISBN 152396300X
44. Deepak Vohra. *Pro MongoDB Development.* Нью-Йорк: Apress, 2021.ISBN 978-1484215999
45. Ben Frain. *Responsive Web Design with HTML5 and CSS: Develop future-proof responsive websites using the latest HTML5 and CSS techniques.* United Kingdom: Packt publishing, 2021. ISBN 1839211563.