РАЗДЕЛ 2 СРАВНЕНИЕ АНАЛОГИЧНЫХ САЙТОВ И ВЫЯВЛЕНИЕ ПОДХОДЯЩИХ УСЛОВИЙ РАБОТЫ

Этот раздел содержит анализ сайтов на реальных примерах области и определение конкретных условий работы системы.

2.1 Анализ сайтов.

К сожалению, даже у популярных конкурентов могут быть или устаревшие технологии, или не опытные специалисты.

Типичный пример это сайт https://edu.tatar.ru, где нету семантических тегов HTML, а это понижение в репутации поисковых систем от Google и Яндекс.

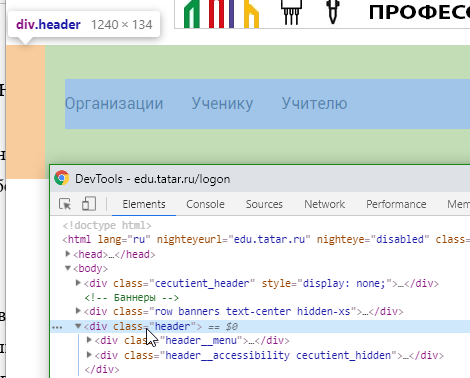
****

Рис. 2.1 –

В данном примере видно, что присутствует навигация, но в HTML обозначена как обычный блок <div>, а должен быть <nav>.

Так же на данном сайт должны присутствовать минимальный набор из семантических тегов таких как <header>, <footer>, <nav>, <section>, <article>, <address>.

Если сайту не так важно место выдачи в поисковике, особенно такому известному сайту как edu.tatar.ru, то он должен позаботиться как минимум о UX и пользователях с медленным интернетом.

К сожалению, даже об этом не позаботились разработчики данного сайта.

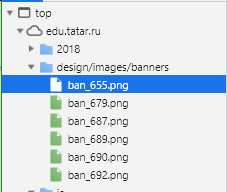
****

Рис. 2.2 - Использование только PNG формата

Как видно на скриншоте выше, картинки отправляются в .png формате, а должны быть в webp, который разработал Google и как они сами пишут:

«WebP lossless images are 26% smaller in size compared to PNGs. WebP lossy images are 25-34% smaller than comparable JPEG images at equivalent SSIM quality index».

****

Рис. 2.3 – Пример правильного использования

Так же вместо обычного <img> тега должен быть специализированный <picture> который помогает браузеру в зависимости от его возможностей загрузить подходящий тип картинки.

Но помимо использования лишнего трафика пользователей, на сайте так же плохо проработан UX то есть опыт пользователя, опыт взаимодействия (англ. User eXperience, UX) - это восприятие и ответные действия пользователя, возникающие в результате использования и/или предстоящего использования продукции, системы или услуги.

Банальным примером является дёргающаяся кнопка в которой не задан размер и в зависимости от размера контента ширина кнопки меняется.

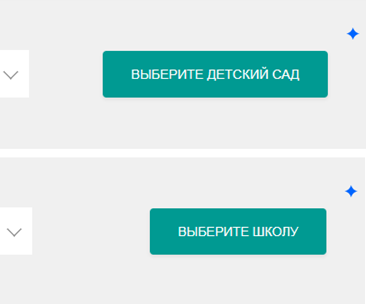


Рис. 2.4 – Одна и та же кнопка разного размера

А можно было бы просто взять самый широкий контент и задать размер по самому широкому контенту.

Единственное что можно взять на заметку с этого сайта, это минификация исходных .js файлов и последующая архивация перед отправкой клиенту и возможно использование nginx для кеширования .js архивов.

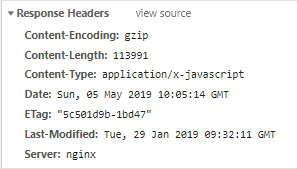
****

Рис. 2.5 – Исходный код в жатом виде

Если не смотреть на Российский сегмент сайтов, то сразу можно увидеть на обычном сайте с новостями использование Семантических тегов.

****

Рис. 2.6 – Использование правильных тегов

Так же использование webP, что дает 30% экономии трафика для каждой картинки.

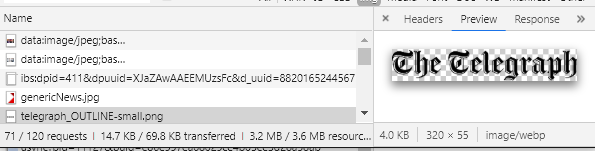
****

Рис. 2.7 – Использование правильного формата

Как можно увидеть «правильные» сайты используют семантику, webp, продуманный UX и современный UI, но это только малая часть того из чего состоят сайты, остальная часть состоит из технологий на чем работает сам сайт.

2.2 Определение конкретных условий работы системы

Конкретными условием работы системы является любое устройство где есть предустановленные программ, такие как:

* Node.js;
* NPM;
* MongoDB.

Так же для корректной работы системы необходим доступ к aws api, для этого используется специальный токен, который можно получить на сайте <https://aws.amazon.com/ru/>

Но этот токен нельзя не кому передавать так как он работает так же как и логин/пароль от аккаунта.

2.3 Определение частных условий

В условиях частных определений входят понятия для конкретной реализации проблемы, например, одна их частных проблем является авторизация и аутентификация пользователя, так же часто проблемой возникает сохранение медиа файлов, обе эти проблемы рассмотрены более подробно ниже.

2.3.1 Авторизация и аутентификация

Для начала нужно определить в чем разница между столь одинаковыми терминами.

Аутентификация - подтверждение подлинности чего-либо или кого-либо. Например, предъявление паспорта - это подтверждение подлинности заявленного имени отчества.

Авторизация является функцией определения прав доступа к ресурсам и управления этим доступом. Авторизация - это не то же самое что идентификация и аутентификация: идентификация - это называние лицом себя системе; аутентификация - это установление соответствия лица названному им идентификатору; а авторизация - предоставление этому лицу возможностей в соответствие с положенными ему правами или проверка наличия прав при попытке выполнить какое-либо действие. Например, авторизацией являются лицензии на осуществление определённой деятельности.

Индустриальным стандартом стали универсальные решения наподобие passport.js. Но, учитывая сложившуюся ситуацию, обычным явлением стало то, что разработчики никогда в полной мере не понимают того, какие именно механизмы принимают участие в работе систем аутентификации.

Организация аутентификации пользователей в среде Node.js рассматривается на практическом примере, где показана организация регистрации пользователей в системе и организация их входа в систему. Здесь будут подняты такие вопросы, как работа с технологией JWT и имперсонация пользователей.

Требования к проекту, в котором реализуется аутентификация:

1. Наличие базы данных, в которой будет храниться адрес электронной почты пользователя и его пароль, либо - clientId и clientSecret, либо - нечто вроде комбинации из приватного и публичного ключей.
2. Использование сильного и эффективного криптографического алгоритма для шифрования пароля.

В тот момент, когда написан материал, считается, что лучшим из существующих криптографических алгоритмов является Argon2. Так же не рекомендуется использовать простые криптографические алгоритмы вроде SHA256, SHA512 или MD5.

Когда в системе создаётся новый пользователь, его пароль необходимо хэшировать и сохранить в базе данных. Пароль в базе сохраняют вместе с адресом электронной почты и другими сведениями о пользователе (например, среди них может быть профиль пользователя, время регистрации и так далее).

Данные учётной записи пользователя должны выглядеть примерно так, как показано ниже.



Рис. 2.8 - Данные пользователя, полученные из MongoDB с помощью Robo3T

Вот как выглядит схема действий, выполняемых в том случае, когда пользователь пытается войти в систему.

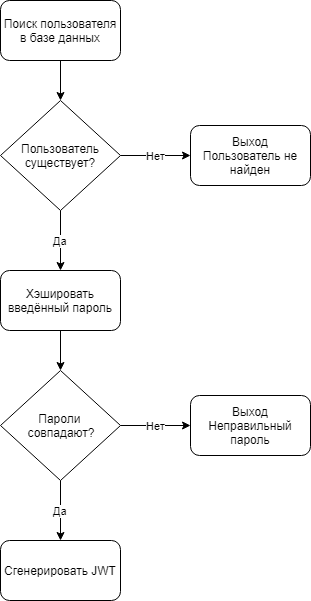


Рис. 2.9 - Вход пользователя в систему

Вот что происходит при входе пользователя в систему:

1. Клиент отправляет серверу комбинацию, состоящую из публичного идентификатора и приватного ключа пользователя. Обычно это - адрес электронной почты и пароль.
2. Сервер ищет пользователя в базе данных по адресу электронной почты.
3. Если пользователь существует в базе данных - сервер хэширует отправленный ему пароль и сравнивает то, что получилось, с хэшем пароля, сохранённым в базе данных.
4. Если проверка оказывается успешной - сервер генерирует так называемый токен или маркер аутентификации - JSON Web Token (JWT).

JWT - это временный ключ. Клиент должен отправлять этот ключ серверу с каждым запросом к аутентифицированной конечной точке.

Верификация пароля производится с использованием библиотеки argon2. Это делается для предотвращения так называемых «атак по времени». При выполнении такой атаки злоумышленник пытается взломать пароль методом грубой силы, основываясь на анализе того, сколько времени нужно серверу на формирование ответа.

Теперь давайте поговорим о том, как генерировать JWT.

JSON Web Token (JWT) - это закодированный в строковой форме JSON-объект. Токены можно воспринимать как замену куки-файлов, имеющую несколько преимуществ перед ними.

Токен состоит из трёх частей. Это - заголовок (header), полезная нагрузка (payload) и подпись (signature). На следующем рисунке показан его внешний вид.



Рис. 2.10 - JWT

Данные токена могут быть декодированы на стороне клиента без использования секретного ключа или подписи.

Это может быть полезным для передачи, например, метаданных, закодированных внутри токена. Подобные метаданные, могут описывать роль пользователя, его профиль, время действия токена, и так далее. Они могут быть предназначены для использования во фронтенд-приложениях.

Вот как может выглядеть декодированный токен.

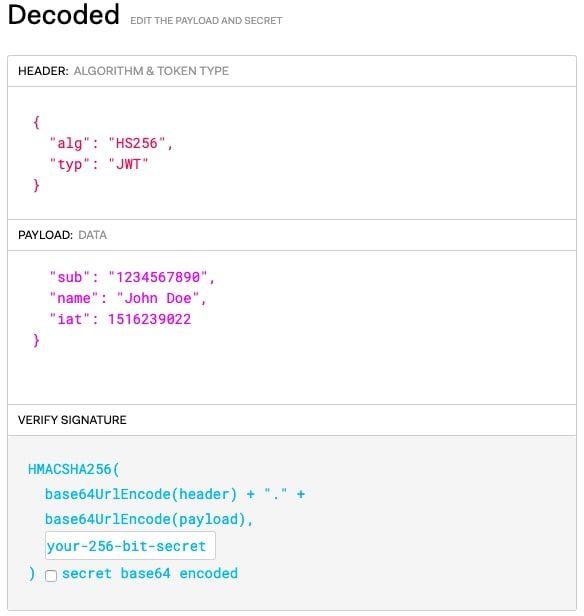


Рис. 2.11 - Декодированный токен

Для завершения работы над сервисом аутентификации пользователей нам нужна функция создания такого токена.

Создавать JWT можно с помощью библиотеки jsonwebtoken. Найти эту библиотеку можно в npm.

Самое важное здесь - это закодированные данные. Не отправляйте в токенах секретную информацию о пользователях.

Подпись - это секретные данные, которые используются для генерирования JWT. Очень важно следить за тем, чтобы подпись не попала в чужие руки. Если подпись окажется скомпрометированной - атакующий сможет генерировать токены от имени пользователей и красть их сессии.

Теперь клиентскому коду нужно отправлять JWT в каждом запросе к защищённой конечной точке.

Рекомендуется включать JWT в заголовки запросов. Обычно их включают в заголовок Authorization.

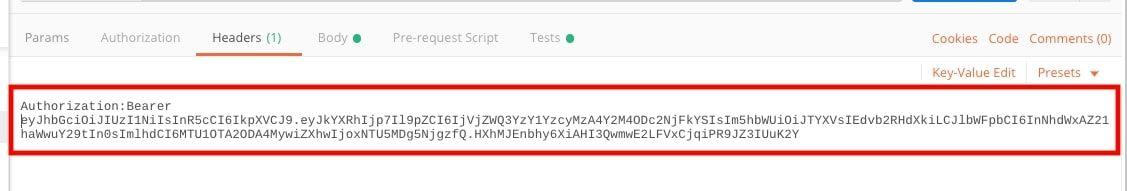


Рис. 2.12 – Код авторизации

Теперь, на сервере, нужно создать код, представляющий собой промежуточное ПО для маршрутов express.

Полезно иметь возможность получать полные сведения об учётной записи пользователя из базы данных и присоединять их к запросу. В нашем случае эта возможность реализуется средствами промежуточного ПО.

После реализации этого механизма маршруты смогут получать сведения о пользователе, который выполняет запрос.

Теперь определенные маршруты защищены. Для доступа к ним пользователь должен иметь валидный JWT. Маршрут, кроме того,

может использовать сведения о пользователе для поиска в базе данных необходимых ему сведений.

Токены защищены от злоумышленников.

Почитав об использовании JWT, вы можете задаться следующим вопросом: «Если данные JWT могут быть декодированы на стороне клиента - можно ли так обработать токен, чтобы изменить идентификатор пользователя или другие данные?».

Декодирование токена - операция очень простая. Однако нельзя «переделать» этот токен, не имея той подписи, тех секретных данных, которые были использованы при подписывании JWT на сервере.

Именно поэтому так важна защита этих секретных данных.

Наш сервер проверяет подпись в промежуточном ПО isAuth. За проверку отвечает библиотека express-jwt.

Теперь, после того, как мы разобрались с тем, как работает технология JWT, поговорим о некоторых интересных дополнительных возможностях, которые она нам даёт.

Как имперсонировать пользователя?

Имперсонация пользователей - это техника, используемая для входа в систему под видом некоего конкретного пользователя без знания его пароля.

Эта возможность весьма полезна для супер-администраторов, разработчиков или сотрудников служб поддержки. Имперсонация позволяет им решать проблемы, которые проявляются только в ходе работы пользователей с системой.

Работать с приложением от имени пользователя можно и не зная его пароля. Для этого достаточно сгенерировать JWT с правильной подписью и с необходимыми метаданными, описывающими пользователя.

Создадим конечную точку, которая может генерировать токены для входа в систему под видом конкретных пользователей. Этой конечной точкой сможет пользоваться только супер-администратор системы.

Для чала нам нужно назначить этому пользователю роль, с которой связан более высокий, чем у других пользователей, уровень привилегий. Это можно сделать множеством различных способов. Например, достаточно просто добавить поле role в сведения о пользователе, хранящиеся в базе данных.

Выглядеть это может так, как показано ниже.



Рис. 2.13 - Новое поле в сведениях о пользователе

Значением поля role супер-администратора будет super-admin.

Далее, надо создать новое промежуточное ПО, которое проверяет роль пользователя.

Оно должно быть помещено после isAuth и attachCurrentUser. Теперь создадим конечную точку, которая генерирует JWT для пользователя, от имени которого супер-администратор хочет войти в систему.

Как видите, тут нет ничего таинственного. Супер-администратор знает адрес электронной почты пользователя, от имени которого нужно войти в систему. Логика работы вышеприведённого кода очень напоминает то, как работает код, обеспечивающий вход в систему обычных пользователей. Главное отличие заключается в том, что здесь не производится проверка правильности пароля.

Пароль тут не проверяется из-за того, что он здесь просто не нужен. Безопасность конечной точки обеспечивается промежуточным ПО.

2.3.2 Сохранение медиа файлов

Самые известные на мировом рынке объектные хранилища - это Amazon S3 (Simple Storage Service), Google Cloud Storage и Microsoft Blobs Storage. Они надежны, доступны через интернет и масштабируются до сотен петабайт. Наиболее популярный способ доступа к объектному хранилищу - S3 RESTful API от компании Amazon. Есть и альтернатива, OpenStack Swift - поддерживает оба API и даже имеет собственное (как у Microsoft или Google, например). Дополнительно предоставляется доступ по другим файловым и блочным протоколам вроде WebDAV, NFS и FTP, но в контексте этой статьи мы тестируем только доступ по протоколу HTTPS через разнообразные API.

Для сравнения возьмем отечественные публичные объектные хранилища. Их не так много, и мы рассмотрим наиболее заметных игроков российского рынка: Mail.Ru Cloud Solutions, servers.ru, Selectel, Техносерв и Ростелеком.

Предложения хранилищ большой западной тройки можно условно разделить на три части: объемы хранения, операции (запросы), трафик и дополнительные услуги - вроде Amazon S3 Storage Management, Amazon S3 Transfer Acceleration и межрегиональной репликации. Подробные прайсы есть на сайтах Amazon S3 (цены без налогов), Microsoft Blobs Storage (исходящий трафик) и Google Cloud Storage. Нас интересуют стандартные хранилища с «горячими» данными - для редко используемых данных и архивов многие провайдеры предлагают варианты дешевле, которые в этом обзоре мы не затрагиваем.

Основными составляющими стоимости услуг будут объёмы хранения, а также исходящий трафик за пределы сети облачного провайдера. Попробуем их оценить и для удобства переведём цены в рубли по курсу ЦБ РФ на 13 января 2018 г.

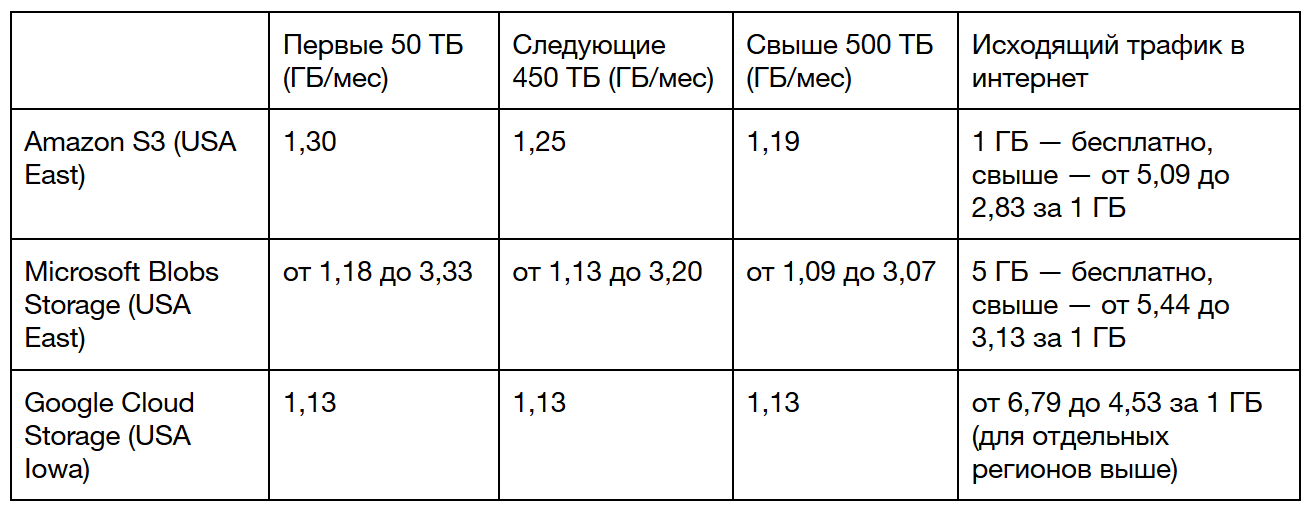


Рис. 2.14 - Таблица сравнения зарубежных провайдеров

У отечественных компаний прайс-листы отличаются большим разнообразием (см. таблицу), мы для удобства пересчитаем их с ограничениями, которые использует большая западная тройка. Все цены в рублях с НДС 18%.



Рис. 2.15 - Таблица сравнения отечественных провайдеров

Прайсы не очень наглядны, к тому же, в стоимость услуг заметную (часто - главную) роль играет исходящий трафик в интернет за пределы сети провайдера, поэтому мы попробуем прикинуть стоимость услуг на примере некоторого использующего объектное хранилище проекта.

Допустим, мы хотим создать сервис и клиентское мобильное приложение к нему. Сразу возникает вопрос хранения статических и пользовательских данных. Есть мнение, что использовать объектные хранилища для статики накладно из-за больших объемов исходящего трафика - существуют отдельные провайдеры, которые позволяют создать сеть доставки и дистрибуции контента (CDN) с кеширующими серверами по всему миру. Если у вас крупный проект с огромным количеством пользователей - это может иметь смысл.

Небольшие стартапы даже статику могут держать в объектном хранилище, но поскольку это все индивидуально, рассматривать такой сценарий мы не будем. Гораздо интереснее пользовательские данные. Благодаря постоянной доступности, возможностям масштабирования и способности выдерживать высокие нагрузки, объектное хранилище отлично подходит для пользовательских файлов любых форматов: фотографий, документов, видеороликов и т. д. Особенно хорошо это заметно на небольших проектах - несколько десятков тысяч файлов и несколько тысяч запросов в сутки обойдутся вам буквально в считанные доллары в месяц. Давайте теперь перейдём к расчётам. Пусть объём пользовательских данных будет достаточно ощутимым (для других вариантов все-таки есть традиционный хостинг) и составляет 1 ТБ - это примерно 5 миллионов фотографий или 50 тысяч небольших видеороликов. Пусть наш условный проект генерирует исходящий трафик в 1 ТБ в месяц. Давайте посчитаем в какую примерно сумму он нам обойдётся.

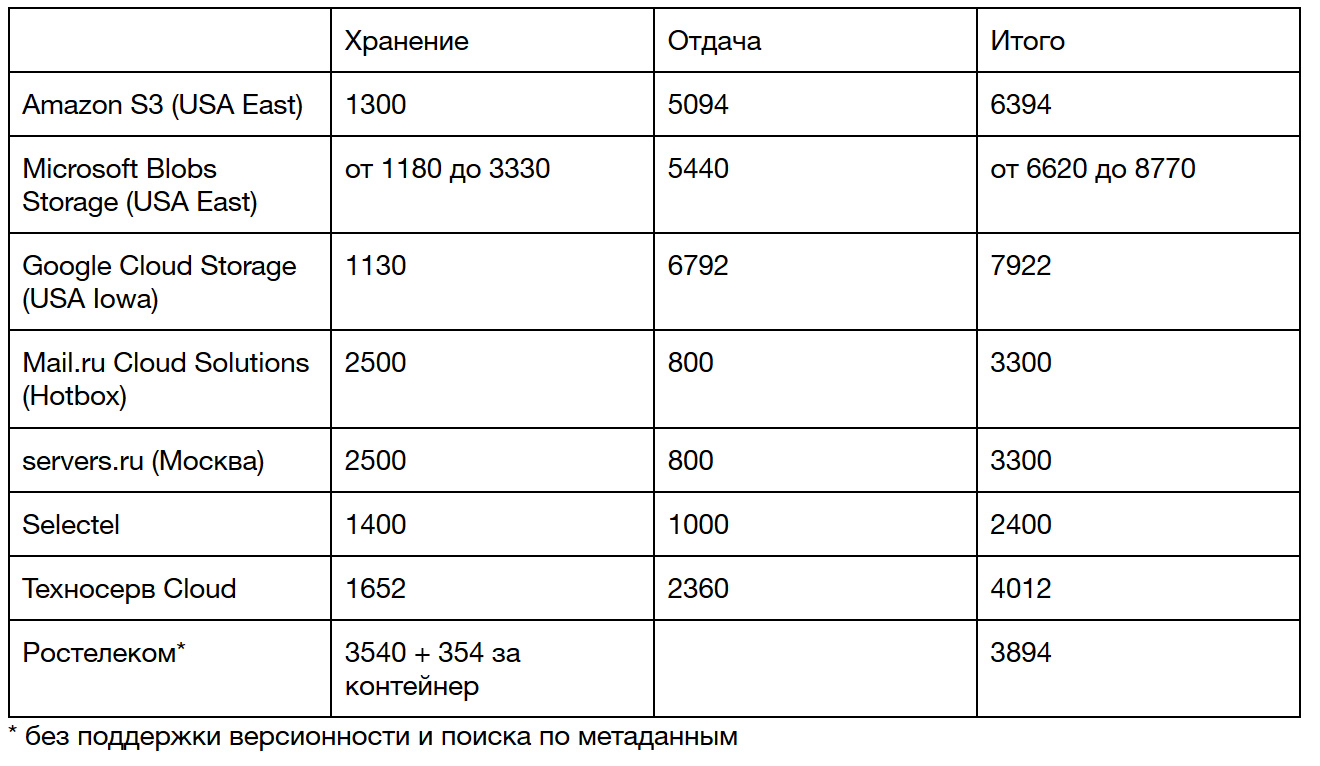


Рис. 2.16 - Таблица с подсчетом стоимости

Операции и запросы мы вынесли за скобки, потому что на начальном этапе понять их количество проблематично - в любом случае, это далеко не основная статья расходов. Сразу становится заметно, что у крупных западных провайдеров очень дорогой исходящий трафик при сходных ценах за объемы хранения. У Amazon, например, он составляет $0,09 за ГБ (в пределах 10 ТБ в месяц) или $90 для нашего условного проекта - весьма недешевое удовольствие. У Google ещё дороже - $0,12 за гигабайт. Хороший повод

выбрать отечественного поставщика услуг.

Mail.ru Cloud Solutions развивается аналогично западным сервисам. Помимо Hotbox заказчикам предлагают хранилище Icebox для редко используемых данных. S3-совместимый API позволяет подключаться к хранилищу при помощи множества разнообразных приложений и использовать коннекторы к разным языкам программирования.

Хранилище servers.ru использует OpenStack Swift (S3 API не поддерживается). Недорогих тарифных планов для редко используемых данных и архивов провайдер не предлагает - по сути дела это хороший вариант для хостинга, позволяющий, например, хранить статические данные сайтов и решать другие сходные задачи. Для корпоративной виртуальной

инфраструктуры список услуг все-таки бедноват. Есть сведения, что провайдер строит приватные облака для клиентов и, в частности, делает там выделенные ceph-инсталляции. На сайте об этом не сообщается.

В Selectel все аналогично: доступ через OpenStack Swift (S3 API хранилище не поддерживает) и отсутствие отдельных тарифных планов для редко используемых данных, хотя очень вкусная цена сглаживает этот момент. Основное применение, как и в предыдущем случае - хостинг статических данных, сайтов и т. д.

Компания Техносерв запустила полноценное корпоративное объектное хранилище для бизнеса и вполне может поспорить с Mail.Ru Cloud Solutions. Заявлена полная совместимость с S3 API и есть отдельный тариф для хранения «холодных» данных. Недостаток один - с физическими лицами провайдер не работает.

Надежда национальной облачной платформы - хранилище Ростелеком использует Hitachi Content Platform и поддерживает S3 API, OpenStack Swift и некий REST API. Отдельные тарифные планы для редко используемых данных у Ростелекома отсутствуют, а настройка хранилища и доступа к нему, как бы это помягче выразиться, неочевидна. Для примера - статья [http://help.cloud.rt.ru/virthran/start/instrukcyas3browver] про доступ через S3 Browser - необходимость вручную кодировать login в base64 и считать хеш md5 для пароля радуют несказанно. С физическими лицами Ростелеком также не работает - все облачные услуги компании доступны только бизнесу и госструктурам.

Увы, на стороне объектного хранилища мы не можем запускать программы, поэтому клиент-серверные измерялки пропускной способности канала (iperf и т. д.) использовать не получится. Мы пропингуем свои хранилища сотней маленьких (64 байта) и сотней больших (1 килобайт) пакетов с внешней по отношению ко всем подопытным виртуальной машины, работающей в одном из российских датацентров. Пакеты не бьются (0% потерь), но время отклика разное. У отечественных сервисов оно гораздо

меньше, как и ожидалось - при обмене большими пачками мелких файлов это свою роль сыграет. Пропинговать Техносерв нет технической возможности, провайдер фильтрует ICMP.

Датацентры наших сервисов размещаются в Москве, большой западной тройки - в США.

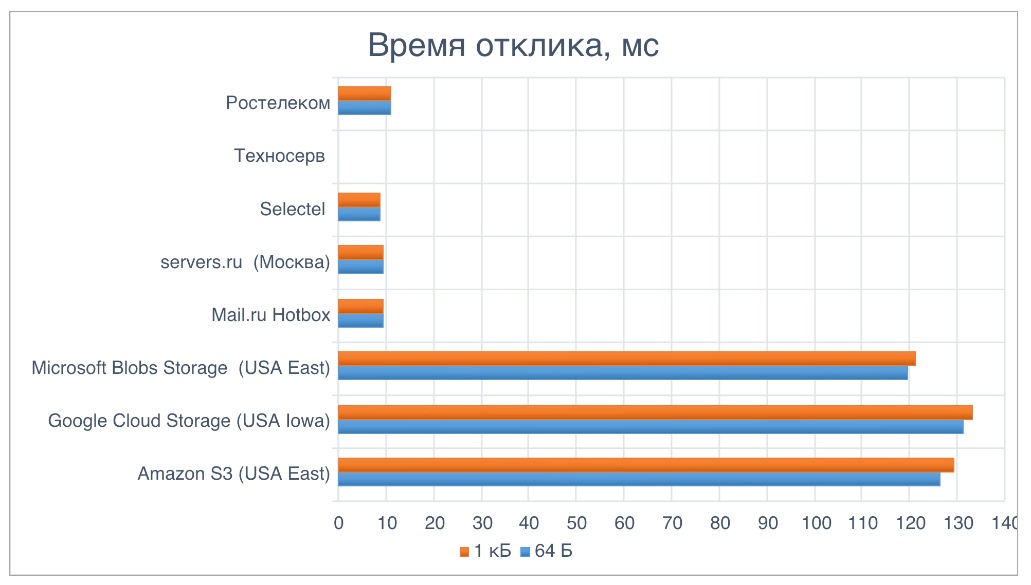
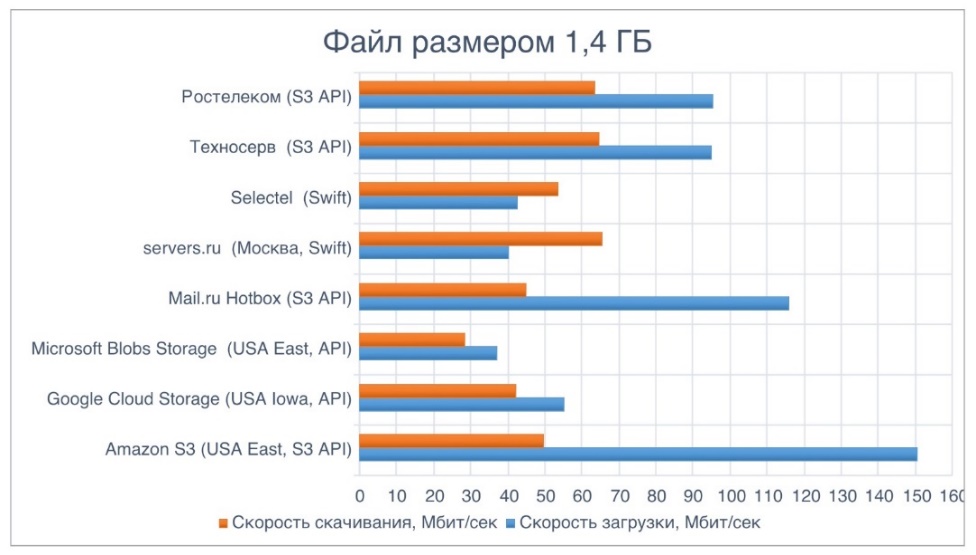


Рис. 2.17 - Отклики разных провайдеров

Для тестирования производительности из России мы с внешней по отношению ко всем подопытным машины (та же, что и в предыдущем опыте) посмотрим скорость закачки файлов в хранилище и скорость скачивания. Для первого теста возьмём большой файл, а для второго - кучу мелких файлов россыпью. Сведём все результаты в диаграмму и взглянем на результат. Работать, разумеется, будем через REST API, которое поддерживает тот или иной сервис - при том разнообразные файловые и блочные протоколы нам неинтересны

Рис. 2.18 - Нагрузочный тест с большим файлом

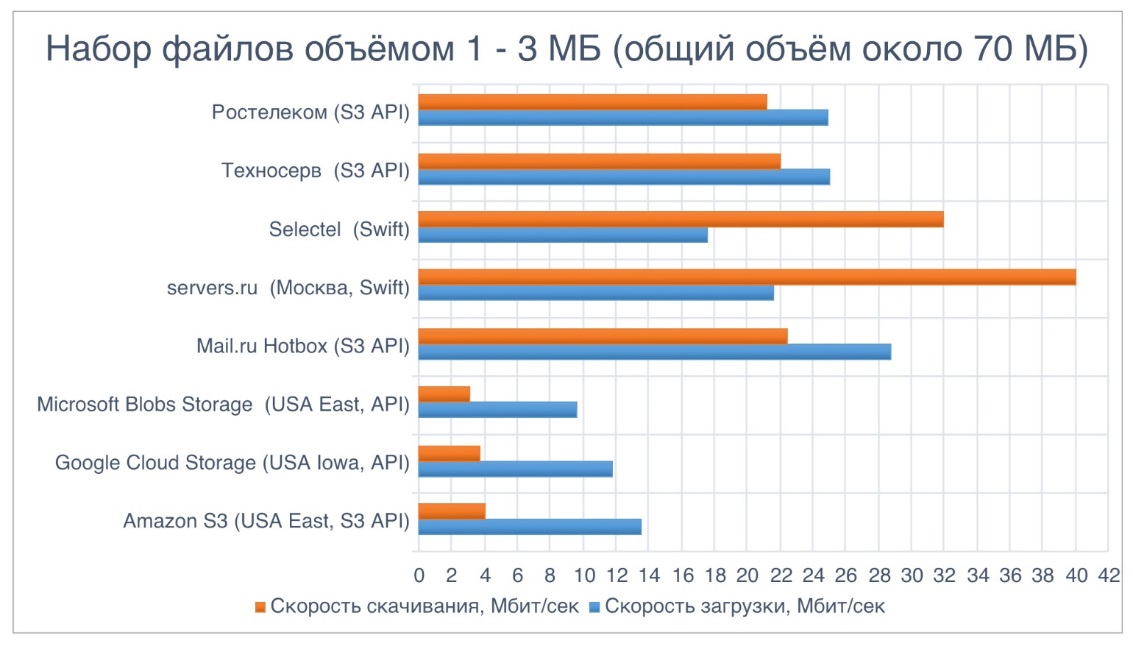


Рис. 2.19 - Нагрузочный тест с маленькими файлами

Для клиентов из России наши сервисы явно быстрее американских Google и Microsoft, а вот Amazon вполне способен с ними поспорить и даже серьезно обогнать по скорости загрузки большого файла. Хотя скачивается такой файл с отечественных хранилищ ощутимо быстрее.

С набором мелких файлов картина совершенно иная - отечественные сервисы серьезно обгоняют зарубежные, что было ожидаемо. У публичных хранилищ Selectel и servers.ru обнаружилась интересная особенность: контент скачивается с них гораздо быстрее, чем закачивается (у других сервисов

все наоборот). Это лишний раз подтверждает, что эти сервисы оптимизированы под раздачу статического контента и хороши для разнообразного хостинга, но хуже подходят для корпоративной ИТ-инфраструктуры.

Отечественные провайдеры достойно конкурируют с большой западной тройкой. Мы тестировали публичные хранилища для горячих данных, и наличие датацентров в России (читай - в Москве) здорово им помогает. Хотя Amazon в состоянии дать высокую скорость на больших (1,4 ГБ) файлах, с пачками мелких наши сервисы работают гораздо быстрее, что важно: основные сценарии применения все-таки предполагают работу с большим количеством мелких файлов. По ценам все не так однозначно - есть варианты гораздо дешевле иностранных аналогов, есть и дороже (разброс достаточно большой). У servers.ru и Selectel отсутствуют недорогие тарифные планы для редко используемых данных. Это станет проблемой, если вы соберётесь развернуть на их площадке виртуальную ИТ-инфраструктуру компании (не хотелось бы по полной программе оплачивать хранение бекапов и архивов). Зато они очень быстро отдают статический контент и хороши для хостинга. Услуги Ростелекома сложны в настройке, но список поддерживаемых API у него самый широкий и скорости хранилище обеспечивает впечатляющие. Хотя отдельного тарифа для редко используемых данных там тоже нет. Для корпоративного применения наиболее интересны предложения Mail.ru и Техносерв - тут тебе и поддержка S3 API, и отдельные тарифы для редко используемых данных, и приличные скорости доступа. В общем, есть из чего выбрать.

Еще один плюс российских сервисов - отсутствие конфликтов с законодательством (все мы помним о работе с персональными данными). Даже собирая анкетные сведения о пользователях сайта, вы можете ненароком получить серьезный штраф, если информация будет храниться за пределами страны.

РАЗДЕЛ 3 РАЗРАБОТКА ТЕХНИЧЕСКОГО ПРОЕКТА И ЕГО РЕАЛИЗАЦИЯ

Изм.

Лист

№ докум.

Подпись

Дата

Лист

1

*2020 395 09.02.05 ПЗ*

Разраб.

*Бакакин Р.И.*

Провер.

Устименко Н.Ф.

Реценз.

*Усанин И.Л.*

Н. Контр.

*Ярославлева И.А.*

Утверд.

*Ярославлева И.А*

*РАЗДЕЛ 3 РАЗРАБОТКА ТЕХНИЧЕСКОГО ПРОЕКТА И ЕГО РЕАЛИЗАЦИЯ*

Лит.

Листов

14

*ГАПОУ «НИТ» гр.395*

52

Этот раздел содержит емкое описание функциональной части приложения с описанием работы внутренних алгоритмов и визуализацией этих частей, а также описание конфигурации для запуска.

3.1 Модули

Для создания программы, Node.js необходимо использовать как один из многочисленных модулей платформы CommonJS. Модуль представляет собой набор файлов JavaScript. В Node.js, если у нас есть файл с именем «index.js», то все модули должны размещаться в папке «node\_modules». А в индексном файле необходимо импортировать подключенные модули, указав имя на его название. Например, в предыдущем листинге мы импортировали модуль с именем «HTTP», который является уже встроенным в Node.js.

На сегодняшний день существует около 18 000 модулей для node.js, которые можно найти на сайте NPM https://npmjs.org. В день осуществляется более 300 000 загрузок.

все модули должны размещаться в папке «node\_modules». А в индексном файле необходимо импортировать подключенные модули, указав имя на его название. Например, в предыдущем листинге мы импортировали модуль с именем «HTTP», который является уже встроенным в Node.js.

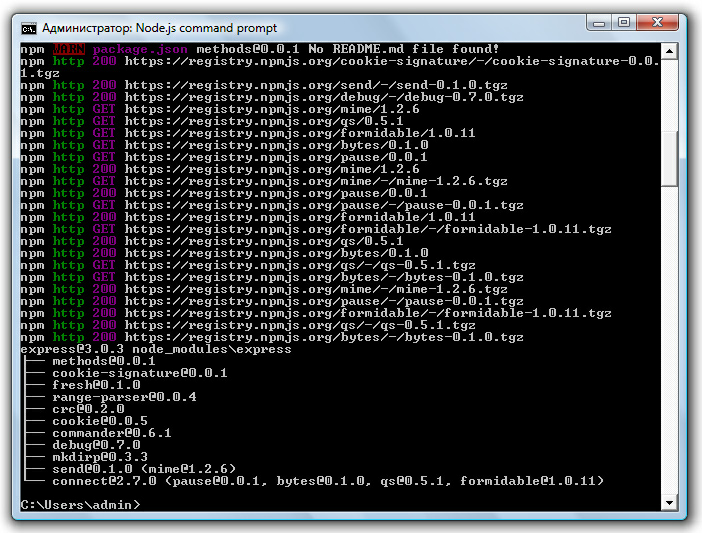


Рис. 3.1 – Установка модулей

3.2Принцип работы подключенных модулей

Общая схема выглядит следующим образом:

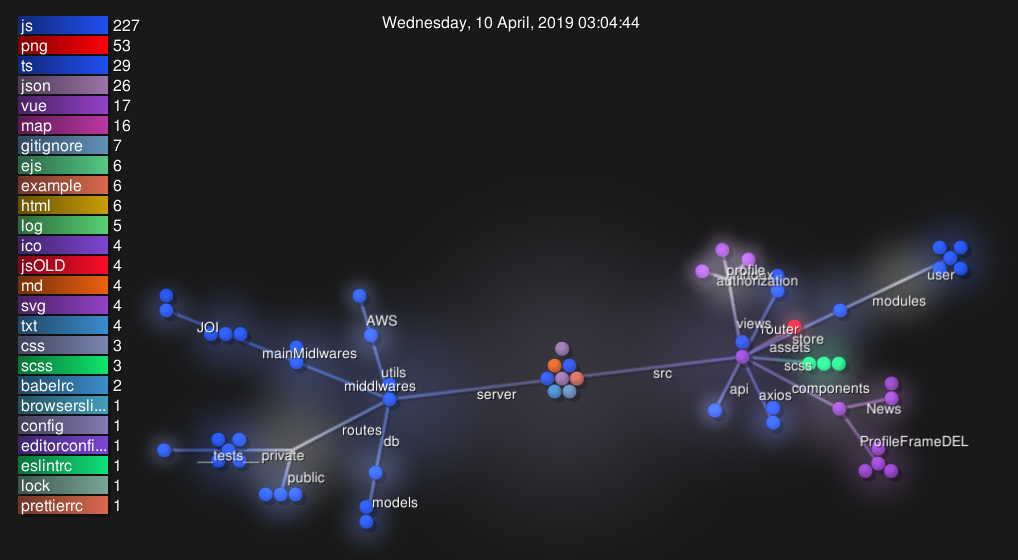


Рис. 3.2 – Схема структуры папок

Слева расположен серверная часть, справа клиентская часть. Скопление по середине это конфигурационные файлы.

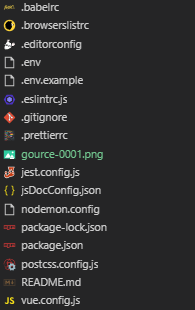


Рис 3.3 – Конфигурационные файлы

При запуске всего приложения сначала собирается клиентская часть через webpack и сервер отдает не всю клиентскую часть целиком, а только кусками по роуту приложения.

Компоненты клиенской части:

1. Базовый – загружается сразу так как хранит базовую логику приложения, в котором прописаны роуты и взаимосвязь данных для других компонентов.
2. Новости – загружается только если нужен общий компонент всех новостей или частные компоненты отдельных новостей.
3. Профиль – загружается при переходе на страницу личного профиля пользователя, обычно после авторизации.
4. Авторизация – загружается при переходе на страницу авторизации.
5. Мероприятия - загружается только если нужен общий компонент всех мероприятий.

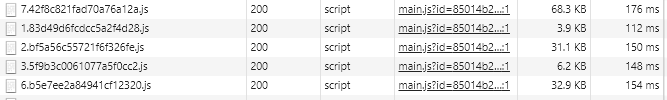


Рис 3.4 – Подгрузка каждого компонента в зависимости от необходимости

Визуализация клиентской части выглядит следующим образом:

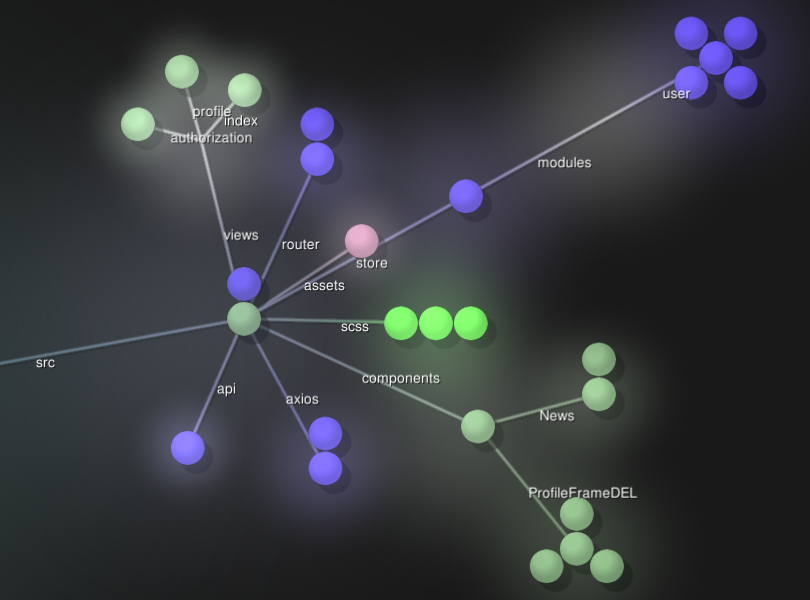


Рис 3.5 – клиентская часть

* views: отображает нужный компонент по маршрутизации;
* router: отвечает за подгрузку компонента по маршрутизации;
* store: хранит в себе модули и связь данных внутри каждого модуля;
* scss: это стили css;
* components: глобальные компоненты для всей клиентской части;
* axios: контролирует общение с сервером;
* api: реализация общения с сервером.

После того как клиентская часть собираться через webpack, происходит запуск серверной части в следующем алгоритме:

1. Подключение всех промежуточных слоев для запросов (koa-body, koa-cookies, и так далее).
2. Загрузка схем моделей для базы данных.
3. Попытка подключения к базе данных.
4. Загрузка схем моделей Joi для валидации входных данных.
5. Открытие портов для запросов.

Каждый новый запрос проходит не определенное количество проверок.

Например, при запросе на «/\*» – запрос пройдет через:

1. Все промежуточные слои для запросов.
2. Проверку на koa-router.
3. Отдачу статических данных, то есть клиентской части приложения.

При запросе на «/api» – запрос пройдет через:

1. Все промежуточные слои для запросов.
2. Проверку на koa-router.
3. При найденном совпадении с koa-router, будет поиск с Joi схемами.
4. При успешной валидации Joi схемами, будет отработана функция api.
5. Возвращение результата функции api.

При не удачном запросе на ‘/api’ будет возвращено 404 HTTP.

При не удачном отработке функции api будет возвращено 500 HTTP.

Визуализация серверной части выглядит следующим образом:

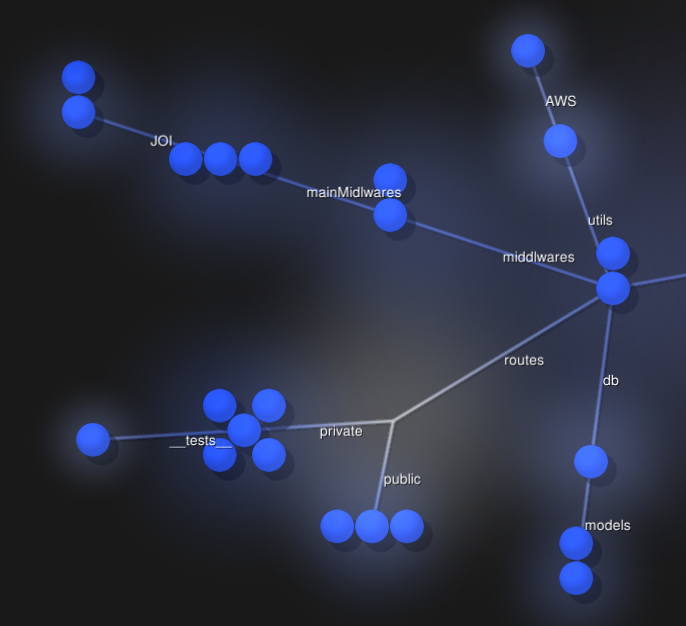


Рис 3.6 – Архитектура папок серверной части

1. db – хранит в себе модели базы данных, подключение к test, dev, prod вариантам баз дынных, отлов ошибок базы данных.
2. routes – имеет private и public пути, private для авторизованных пользователей, public для любого пользователя. Так же имеет тесты в папке \_\_tests\_\_
3. middlewares – хранит в себе все возможные промежуточный слой для любых запросов.
4. utils – содержит в себе вспомогательные модули, например, для работы с AWS.

3.3 Конфигурация приложения

Для корректного запуска приложения необходимы следующие программы:

1. Node.js
2. NPM

Обе программы устанавливаются после скачивания Node.js из официального сайта Node https://nodejs.org/en/ желательно скачивать LTS версию так как она является самой стабильной.

Так же используется база данных MongoDB через ORM Mongoose.

Базу данных можно использовать от стороннего хостинга, например mLab или развернуть локально, например для windows подойдет этот мануал <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

После установки необходимого окружения можно назначить необходимые переменные внутри файла .env

# AMASON S3

AWS\_ACCESS\_KEY\_ID: access key из AWS;

AWS\_SECRET\_ACCESS\_KEY= secret access key из AWS;

AWS\_REGION= Регион хранилища;

AWS\_BUCKET\_NAME= Имя хранилища.

# app

DEBUG= Вывод абсолютно всех логов.

## CORS

ORIGIN\_FOR\_CORS= Пропускной origin для CORS (String).

# VUE

VUE\_APP\_SERVER\_PORT=порт сервера для клиентской части;

VUE\_APP\_SERVER\_HOST=домен сервера для клиентской части.

# mail

MAIL\_USERNAME= Имя почты;

MAILGUN\_API\_KEY= Секретный ключ почты;

MAILGUN\_DOMAIN= Домен почты.

# mongodb+srv or mongodb

DB\_TYPE= тип базы данных (сейчас только mongodb).

## dev | Данные Базы данных для development режима приложения

DEV\_DB\_USERNAME= Имя пользователя;

DEV\_DB\_PASSWORD= Пароль пользователя;

DEV\_DB\_NAME= Имя базы данных;

DEV\_DB\_HOSTNAME= Домен базы данных;

DEV\_DB\_PORT= Порт базы данных.

## test | Данные Базы данных для test режима приложения

TEST\_DB\_USERNAME= Тоже самое что и в данных для dev;

TEST\_DB\_PASSWORD= Тоже самое что и в данных для dev;

TEST\_DB\_NAME= Тоже самое что и в данных для dev;

TEST\_DB\_HOSTNAME= Тоже самое что и в данных для dev;

TEST\_DB\_PORT= Тоже самое что и в данных для dev.

## prod | Данные Базы данных для production режима приложения

PROD\_DB\_USERNAME= Тоже самое что и в данных для dev;

PROD\_DB\_PASSWORD= Тоже самое что и в данных для dev;

PROD\_DB\_NAME= Тоже самое что и в данных для dev;

PROD\_DB\_HOSTNAME= Тоже самое что и в данных для dev;

PROD\_DB\_PORT= Тоже самое что и в данных для dev.

# JWT

SECRET\_OR\_KEY= Секретный ключ сервера для криптографии паролей пользователей;

EXPIRES\_IN= срок жизни временных ключей пользователей.

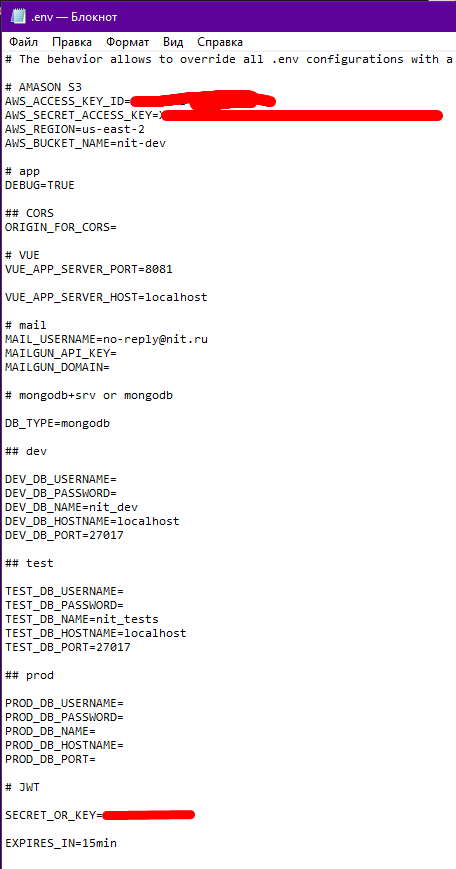


Рис. 3.7 – Пример корректного заполненного .env файла

3.4 Запуск приложения

Перед началом запуска нужно установить нужные данные в .env файле

Запуск приложения описан в файле Readme.md и может меняться, но основной алгоритм выглядит следующим образом и прописываются внутри папки проекта в консоли.

1. npm i –g vue-cli.
2. npm i.
3. npm run start.

Данный алгоритм запускает приложение в production режиме, со всеми оптимизациями для реального сервиса.

Так же вместо npm run start есть еще 12 режимов:

* «serve:front»: Виртуальная сборка клиенской части с hot-reload;
* «build:front»: Реальная сборка клиенской части с исходными файлами;
* «lint:front»: Проверка клиентской части на визуальные ошибки;
* «build:server»: Реальная сборка серверной части с исходными файлами;
* «serve:server»: Виртуальная сборка клиенской части с hot-reload;
* «build»: Тоже самое если запустить одновременно build:server и build:front;
* «serve»: Тоже самое если запустить одновременно serve:server и serve:front;
* «start»: Тоже самое если запустить одновременно build:server и build:front и последующий запуск сервера;
* «start:dist»: запуск сервера;
* «sf»: сокращение от serve:front;
* «ss»: сокращение от serve:server;
* «bf»: сокращение от build:front;
* «bs»: сокращение от build:server.

3.5 Размещение в интернете

На время разработки всегда используется локальное окружение, которое дает быстрый доступ к просмотру только что отредактированного сайта, но обычно только после перезапуска сервера и перезагрузки страницы.  
Для того чтобы избежать постоянных перезапусков используют hot-reload - режим работы локального сервера при котором любое изменение

пересобирает все исходные файлы и отдает измененный вид сайта и новые данные с сервера.

Но такой способ для размещения в интернете и как говорят в «продакшн» не подходит из-за большого количества ограничений.

Самым основным способом размещения в интернете является размещение на хостинге или VDS сервере.

VDS сервер выбирают опытные пользователи, которые могут настроить сервер так как это голая система, в которую нужно все устанавливать самостоятельно.

Для временного размещения в интернете я использовал хостинг evennode в котором был 30 дневное бесплатное размещение node.js и python кода.

Так же вместо evennode можно было использовать решение от google, yandex, amazon и других крупных хостингов.

Единственная проблема evennode заключалась в невозможности собрать весь проект, серверную часть и клиентскую часть, так как evennode не воспринимал devDependencies как обязательными к установки на свой сервер, а зависимости для разработки по правильному не могут быть в зависимостях для постоянной работы приложения.

Из-за этой проблемы необходима «предсборка» всего проекта, которая осуществляется по команде npm run build.

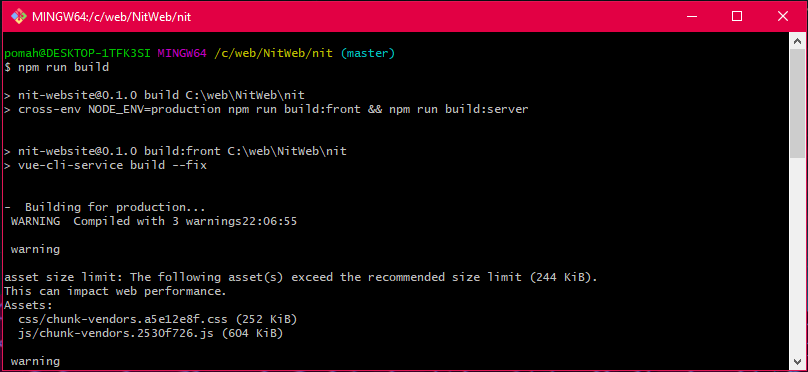


Рис. 3.8 – Работа скрипта «npm run build»

После запуска этого скрипта создается папка с названием dist, в котором находиться уже собранный проект.

Так же в этой папке имеется структура приложения и эта структура повторяет структуру исходного кода так как сборка проходит поочередно, сначала клиентская часть и только потом серверная часть из-за особенности реализации технологий в каждой из частей.

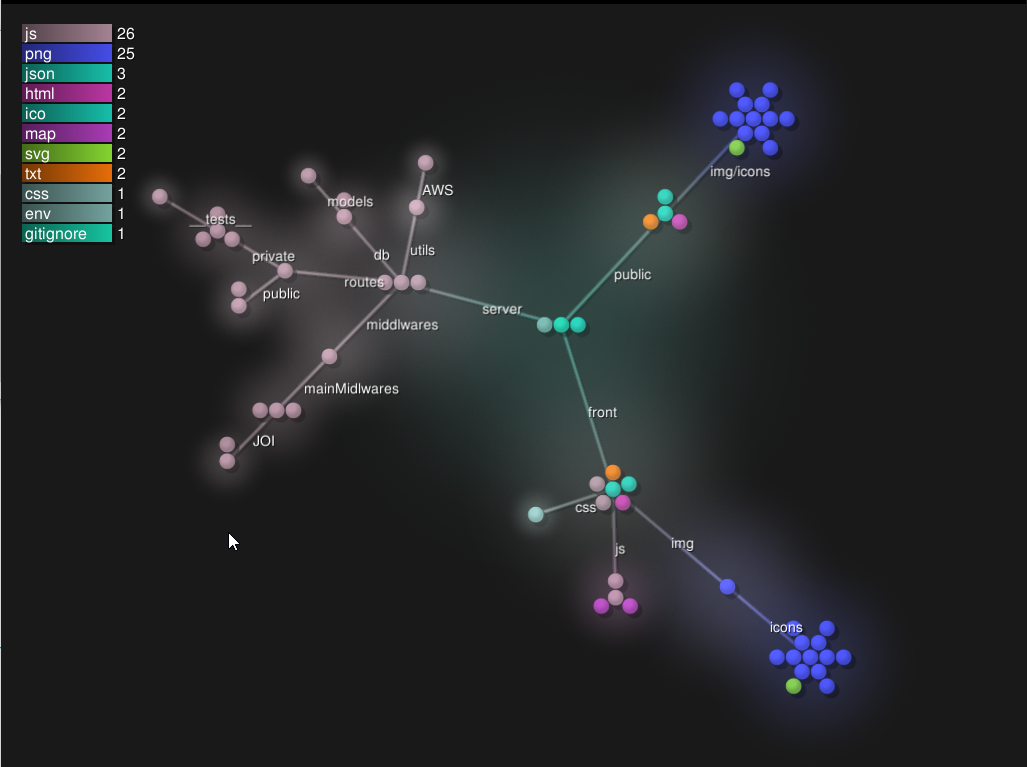


Рис. 3.9 – Структура собранного приложения.

Так же собранное приложение проходит через этапы минификации и обфускации.

Минификация (англ. minification) - процесс, направленный на уменьшение размера исходного кода путём удаления ненужных символов без изменения его функциональности. Минификация особенно полезна для программ на интерпретируемых языках (например, JavaScript), потому что она уменьшает объём данных, которые должны быть обработаны (касаемо сайтов, достигается ускорение загрузки и оптимизация запросов).

Обфускация (англ. obfuscate - делать неочевидным, запутанным, сбивать с толку) или запутывание кода - приведение исходного текста или исполняемого кода программы к виду, сохраняющему её функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию при декомпиляции.

«Запутывание» кода может осуществляться на уровне алгоритма, исходного текста и/или ассемблерного текста. Для создания запутанного ассемблерного текста могут использоваться специализированные компиляторы, использующие неочевидные или недокументированные возможности среды исполнения программы. Существуют также специальные программы, производящие обфускацию, называемые обфускаторами (англ. obfuscator).

После всех этих этапов можно загружать сборку готового приложения на сервера evennode.

Всего есть два варианта загрузки:

1. FTP.
2. Git.

FTP (англ. File Transfer Protocol) - протокол передачи файлов по сети, является одним из старейших прикладных протоколов, появившихся задолго до HTTP, и даже до TCP/IP, в 1971 году; в первое время он работал поверх протокола NCP. Он и сегодня широко используется для распространения ПО и доступа к удалённым хостам. В отличие от TFTP, гарантирует передачу (либо выдачу ошибки) за счёт применения квотируемого протокола.

Протокол построен на архитектуре «клиент-сервер» и использует разные сетевые соединения для передачи команд и данных между клиентом и сервером. Пользователи FTP могут пройти аутентификацию, передавая логин и пароль открытым текстом, или же, если это разрешено на сервере, они могут подключиться анонимно. Можно использовать протокол SSH для безопасной передачи, скрывающей (шифрующей) логин и пароль, а также шифрующей содержимое.

SSH (англ. Secure Shell - «безопасная оболочка») - сетевой протокол прикладного уровня, позволяющий производить удалённое управление операционной системой и туннелирование TCP-соединений (например, для передачи файлов). Схож по функциональности с протоколами Telnet и rlogin, но, в отличие от них, шифрует весь трафик,

включая и передаваемые пароли. SSH допускает выбор различных алгоритмов шифрования. SSH-клиенты и SSH-серверы доступны для большинства сетевых операционных систем.

SSH позволяет безопасно передавать в незащищённой среде практически любой другой сетевой протокол. Таким образом, можно не только удалённо работать на компьютере через командную оболочку, но и передавать по шифрованному каналу звуковой поток или видео (например, с веб-камеры). Также SSH может использовать сжатие передаваемых данных для последующего их шифрования, что удобно, например, для удалённого запуска клиентов X Window System.

После выбора доступа по SSH был написан скрипт для автоматического обновления версии приложения на хостинге.

После запуска этого скрипта начинается:

1. Установка зависимостей на хостинге.
2. Запуск приложения на хостинге.
3. Подключение к базе данных.
4. «Слушание» 80 порта.

После всех этих этапов можно через любое устройство, которое имеет выход в интернет, зайти на домен от хостинга и пользоваться сервисом.