

АННОТАЦИЯ

Отчет 84 с., 5 ч., 22 рис., 28 источников, 2 прил.

ПРОГРАММНЫЙ КОМПЛЕКС ПО ХРАНЕНИЮ И ОБРАБОТКЕ ИНТЕРАКТИВНЫХ ПЛАНОВ ПОМЕЩЕНИЙ, ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ, ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ, БАЗЫ ДАННЫХ, СЕРВИСЫ.

Объектом разработки является программный комплекс, предназначенный для обеспечения работоспособности функционала клиента позволяющего отображать, изменять и создавать интерактивные планы помещений.

Цель работы — обеспечить быструю и бесперебойную обработку запросов, поступающих от множества различных типов клиентов.

В процессе выполнения поставленной задачи, были изучены соответствующие существующие алгоритмы и инструменты, их основные особенности и возможность их применения в рамках решения задачи.

В результате был разработан программный комплекс, который позволит в полной мере оценить возможности и перспективы развития данного программного решения на современном рынке недвижимости.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	7
1. ПОСТАНОВКА ЗАДАЧИ	10
1.1. ОПИСАНИЕ НЕОБХОДИМОГО ФУНКЦИОНАЛА MVP	10
1.2. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ	10
2. ВЫБОР ТЕХНОЛОГИЧЕСКИХ РЕШЕНИЙ	13
2.1. ПЛАТФОРМА	13
2.2. ХРАНЕНИЕ ДАННЫХ	16
2.3. АВТОРИЗАЦИЯ И АУТЕНТИФИКАЦИЯ	18
2.4. ВЕБ СЕРВЕР	23
2.5. ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ ТРАФИКА	24
2.6. ОБЕСПЕЧЕНИЕ ГИБКОГО ДОСТУПА К ФУНКЦИОНАЛУ СЕРВИСА И ХРАНИМЫМ ДАННЫМ	24
2.7. ИНСТРУМЕНТ ДЛЯ КЭШИРОВАНИЯ ДАННЫХ ЗАПРОСОВ	27
2.8. ИНСТРУМЕНТ ДЛЯ ОБЕСПЕЧЕНИЯ ВЗАИМОДЕЙСТВИЯ МИКРОСЕРВИСОВ	28
2.9. РЕЗЮМЕ	29
3. РЕАЛИЗАЦИЯ	31
3.1. МОДЕЛИ ДАННЫХ	31
3.2. GRAPHQL	35
3.3. HTTP СЕРВЕР	38
3.4. МЕХАНИЗМ АВТОРИЗАЦИИ И АУТЕНТИФИКАЦИИ	39
3.5. ФАЙЛОВОЕ ХРАНИЛИЩЕ	41
3.6. СЕРВИС РАССЫЛКИ ЭЛЕКТРОННЫХ ПИСЕМ	44
3.7. СЛОЙ КЭШИРОВАНИЯ	46
3.8. ВЕБ СЕРВЕР	48
4. ОПИСАНИЕ ПРОГРАММЫ	51
4.1. ОБЩИЕ СВЕДЕНИЯ	51
4.2. ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ	51
4.3. ОПИСАНИЕ ЛОГИЧЕСКОЙ СТРУКТУРЫ	52
4.4. ИСПОЛЬЗУЕМЫЕ ТЕХНИЧЕСКИЕ СРЕДСТВА	56
4.5. ВЫЗОВ И ЗАГРУЗКА	57
4.6. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ	57

5. ПОЛУЧЕННЫЙ РЕЗУЛЬТАТ И ДАЛЬНЕЙШЕЕ РАЗВИТИЕ	59
СПИСОК ЛИТЕРАТУРЫ.....	60
ПРИЛОЖЕНИЕ А. БЛОК-СХЕМЫ ОСНОВНЫХ АЛГОРИТМОВ РЕАЛИЗОВАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ	62
ПРИЛОЖЕНИЕ Б. ТЕКСТ РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ	66

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

DB – база данных (от англ. Database);

MVP — простейший работающий прототип продукта, которым тестируют спрос до полномасштабной разработки (от англ. minimum viable product);

ПО – программное обеспечение;

Идентификация – заявление о том, кем является пользователь;

Аутентификация – предоставление подтверждения того, что пользователь тот, кем он себя идентифицирует;

Авторизация – проверка, наличия у пользователя доступа к указанному ресурсу;

Токен – некая строка, сгенерированная специальным образом, в которой хранится информация о пользователе;

Монолит – программный комплекс, все функциональные части которого не разнесены на отдельные микросервисы, а являются частью одного целого;

id – уникальная специальным образом сгенерированная строка, которая ставится в соответствие чему-либо;

Виджет – небольшой графический элемент или модуль, вставляемый на сайт или выводимый на рабочий;

API – описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой (англ. application programming interface).

ВВЕДЕНИЕ

Сложно представить взрослого современного человека, который так или иначе бы не сталкивался с проблемой поиска недвижимости для покупки или аренды. Хотя эта проблема и не нова и существует огромное количество различных сервисов для поиска недвижимости – далеко не все проблемы были решены. Так, например, значительную роль при поиске недвижимости играет планировка, однако зачастую по предоставленной арендодателем или владельцем недвижимости информации – о ней можно только догадываться. Еще сильнее проблема увеличивается, если речь идет именно об аренде, так-как недвижимость может быть с мебелью и прочими предметами интерьера, и было бы большим преимуществом знать заранее какая именно мебель, как и где она расположена.

На данный момент не один популярный ресурс не предоставляет пользователям возможно выкладывать объявления с возможностью предоставления планировки и, возможно, схемы расстановки мебели. Соответственно так-как нет возможности выкладывать объявления с подобного рода информацией - у пользователей, которые интересуются покупкой или арендой недвижимости нет возможности осуществлять более гибкий поиск именно того, что им нужно.

На данный момент существует множество сервисов, способных предоставить кадастровый план квартиры или другой недвижимости, но они недостаточно хорошо распространены и не пользуются популярностью. Так же общей чертой всех этих сервисов является то, что они имеют сложный, нагруженный и не понятный интерфейс, либо же, за предоставление информации берется высокая плата.

Из важных недостатков того, как работают современные сервисы, предоставляющие возможность размещений объявлений о продаже, либо же

сдачи в аренду недвижимости, можно выделить фотографии. Так-как, зачастую, арендодатель или владелец недвижимости не являются профессиональными фотографами – возникает проблема, что фотографий либо недостаточно, либо же совершенно не понятно откуда они были сделаны, сколько комнат в квартире, и так далее. Решением данных проблем могла бы стать возможность отметки на плане помещений места откуда была сделана данная фотография в совокупности с отметкой направления, в котором был направлен объектив камеры. Дальнейшим этапом развития данного способа размещений фотографий, мог бы стать виртуальный помощник, который бы на основе предоставленного плана мог бы не опытному пользователю подсказать позиции, из которых лучше всего делать фотографии, так что бы они были максимально информативными для соискателей.

Данная задача очень актуальна, особенно в современном мире, когда, мошенничество и обман являются обычным делом, и многие люди всяческими уловками пытаются обмануть покупателя или арендатора при поиске им недвижимости. Результатом подобного обмана, в лучшем случае станет в пустую потрачено время, в худшем - потеря большого количества денег. С появлением же возможности просмотра плана недвижимости и, возможно, объектов мебели расположенных внутри, с фотографиями, которые сделаны с понятных ракурсов, и дополнительным набором фильтров, при просмотре объявления – потенциальный покупатель или арендатор сможет сэкономить себе огромное количество времени и ресурсов, просто потому что сразу отфильтрует те варианты, которые в противном случае бы его заинтересовали, из-за умело подготовленных фотографий, или неполноты предоставленной информации.

На данном этапе, решение данной задачи можно разбить на следующие этапы, с точки зрения реализации функционала необходимого на серверной части данного программного комплекса:

- 1) В первом разделе осуществлена постановка задачи: представлено описание необходимого функционала MVP, а также определены цели и задачи, решаемые данной работой;
- 2) Второй раздел содержит описание выбора инструментов и алгоритмов, которые применимы для решения поставленной задачи;
- 3) Третий раздел представляет описание фактической реализации компонентов системы, с использованием выбранных инструментов и алгоритмов;
- 4) Четвертый раздел описывает, полученное, в результате выполнения данной работы ПО;
- 5) Пятый раздел подводит итоги и описывает перспективы дальнейшего развития ПО.

1. ПОСТАНОВКА ЗАДАЧИ

1.1. ОПИСАНИЕ НЕОБХОДИМОГО ФУНКЦИОНАЛА MVP

Для того, чтобы оценить необходимость и востребованность подобного программного решения на рынке было принято решение для начала не создавать новое, полностью независимое ПО, а реализовать его, как дополнение к уже существующим решениям, в качестве виджета. Задачей данного ПО будет дополнение функционала сервисов, позволяющих размещать объявления об аренде и продаже недвижимости, следующим спектром возможностей:

- 1) Интерактивное создание плана с помощью виджета;
- 2) Возможность прикрепления фотографий к точкам на плане, где они были сделаны;
- 3) Возможность размещения на плане предметов мебели;
- 4) Отображение созданного плана помещений с помощью виджета;
- 5) Возможность регистрации пользователя в системе;
- 6) Наличие у пользователя доступа к своим данным через данный виджет, из любого сервиса, в котором он использован;
- 7) Возможность хранения фотографий на сервере.

В рамках данной работы предполагается создание серверной части, данного программного комплекса.

1.2. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Целью данной работы является создание программного комплекса, способного дополнить и улучшить работу сервисов, заинтересованных в использовании функционала предоставляемого виджетом.

Для создания данного программного комплекса и достижения поставленной цели - необходимо решить следующие задачи:

- 1) Составить и изучить список платформ, подходящих для реализации данного ПО;
- 2) Сравнить платформы и выбрать из них ту, которая наилучшим образом подходит для реализации MVP;
- 3) Составить и изучить список DB, подходящих для реализации данного ПО;
- 4) Сравнить DB и выбрать из них ту, которая наилучшим образом подходит для реализации MVP;
- 5) Составить и изучить список алгоритмов авторизации и аутентификации, подходящих для реализации данного ПО;
- 6) Сравнить алгоритмы и выбрать из них тот, который наилучшим образом подходит для реализации MVP;
- 7) Составить и изучить список веб серверов, подходящих для реализации данного ПО;
- 8) Сравнить веб сервера и выбрать из них тот, который наилучшим образом подходит для реализации MVP;
- 9) Составить и изучить список протоколов шифрования трафика, подходящих для реализации данного ПО;
- 10) Сравнить протоколы шифрования трафика и выбрать из них тот, который наилучшим образом подходит для реализации MVP;
- 11) Составить и изучить список способов обеспечения гибкого доступа к API, предоставляемому данным ПО;
- 12) Сравнить способы выбранные способы доступа к API и выбрать из них тот, который наилучшим образом подходит для реализации MVP;
- 13) Составить и изучить список технологий для кэширования данных, подходящих для реализации данного ПО;
- 14) Сравнить технологии кэширования данных и выбрать из них тот, который наилучшим образом подходит для реализации MVP;

- 15) Составить и изучить список технологий для реализации взаимодействия между микросервисами, подходящих для реализации данного ПО;
- 16) Сравнить технологии для реализации взаимодействия между микросервисами и выбрать из них тот, который наилучшим образом подходит для реализации MVP;
- 17) Спроектировать гибкую схему базы данных, которая бы могла быстро меняться и легко модифицировалась;
- 18) Исходя из полученной схемы базы данных, описать модели данных на языке запросов GraphQL;
- 19) Реализация HTTP сервера на выбранной платформе;
- 20) Реализация файлового сервиса;
- 21) Реализация сервиса для отправки электронных писем;
- 22) Реализация механизма кэширования данных;
- 23) Реализация механизма кэширования данных;
- 24) Конфигурирование веб сервера.

2. ВЫБОР ТЕХНОЛОГИЧЕСКИХ РЕШЕНИЙ

Описанные ниже технологии были выбраны, основываясь на данных, полученных на просторах интернета, и личного опыта коммерческой и промышленной разработки программного обеспечения. Однако среди всего прочего можно выделить два следующих ресурса [11], [13].

2.1. ПЛАТФОРМА

Выбор правильной платформы для реализации данного проекта на стадии MVP может во многом отразиться на успехе всего проекта в целом. Это вызвано в первую очередь тем, что от этого будет напрямую зависеть то, насколько просто и легко в дальнейшем будет возможно вносить правки и изменять ПО под нужды клиентов. Таким образом можно составить следующий перечень требований к платформе:

- 1) Гибкость – возможность расширения/изменения существующего функционала не требует значительных усилий;
- 2) Наличие менеджера пакетов, с большой базой готовых модулей – возможность по максимуму использовать готовые и, желательно, проверенные временем решения;
- 3) Востребованность на рынке – возможность, в случае необходимости, легко и быстро найти разработчиков;
- 4) Производительность – платформа должна быть в состоянии обрабатывать большое количество запросов в единицу времени, что бы свести время ожидания пользователей к минимуму;
- 5) Надежность – это должна быть сформировавшаяся технология, хорошо зарекомендовавшая себя на уже реализованных проектах;
- 6) Перспективность – эта технология должна удовлетворять требованиям проекта не только на стадии MVP, но и на этапе его дальнейшей поддержки и развития;

- 7) Сообщество – наличие большого сообщества – очень значительный плюс, который внушает надежду на то, что решение очередной проблемы, обнаруженной в ходе разработки, может быть легко найдено на просторах интернета.

После анализа данных представлены на различных интернет ресурсах, а также опираясь на личный опыт – был составлен следующий список платформ, которые могли бы быть использованы для реализации MVP:

- 1) Java;
- 2) PHP;
- 3) Ruby;
- 4) Python;
- 5) NodeJS.

Далее они будут рассмотрены по порядку, опираясь на выбранные выше критерии.

Java - очень мощная и “взрослая” платформа, которая существует на рынке уже более 20 лет, и активно используется и по сей день. Из-за своей высокой надежности Java активно используется везде, в том числе и в банковских сферах. Однако и у Java есть свои недостатки:

- 1) Гибкость – по сравнению со своими конкурентами из списка выше - Java, пожалуй, самый не гибкий язык из-за своей чрезмерной сложности и монструозности;
- 2) Востребованность – по сравнению с другими выбранными платформами, найти хорошего Java разработчика для начинающей компании – не самая простая задача, в основном, потому что, Java не пользуется большой популярностью среди начинающих разработчиков.

PHP – так же, как и Java – платформа с очень богатой историей, которая в свое время сыскала большую популярность на рынке. Однако в реалиях сегодняшнего дня PHP очень сильно сдает позиции и пользуется плохой репутацией в среде разработчиков, потому что язык позволяет слишком многое

и на нем очень легко написать низко производительный код. PHP обладает следующими недостатками:

- 1) Востребованность – хоть PHP и активно развивается, все же он активно теряет свои позиции и популярность, как среди других платформ, так и среди разработчиков;
- 2) Производительность – из-за специфики своей работы: для каждого нового запроса, поступающего на сервер, запускается отдельный интерпретатор. Таким образом становятся невозможными: одновременная обработка большого количества одновременно поступающих запросов, а также не блокирующие операции ввода и вывода.

Ruby – платформа, популярность которой пошла на спад несколько лет назад, из-за чего сообщество разработчиков сильно уменьшилось и количество разрабатываемых библиотек и пакетов пошло на спад. Соответственно можно определить следующие недостатки:

- 1) Востребованность на рынке – востребованность Ruby стремительно уменьшается и все меньше новых проектов начинают писать с использованием данной платформы;
- 2) Сообщество – сообщество вокруг Ruby, относительно других платформ очень мало, и в ближайшее время это не изменится.

NodeJS и Python – в рамках выбранных критериев оба практически лишены недостатков. Обе платформы обладают богатой историей, очень востребованы на рынке и имеют огромные сообщества, а также активно используются на больших и высоконагруженных проектах. Однако NodeJS в данном случае обладает рядом преимуществ перед конкурентом:

- 1) Не смотря на известные проблемы JavaScript при вычислении математических операций с плавающей точкой – в данной задаче это не является критическим местом;

- 2) Так же значительной особенностью NodeJS и Python является динамическая типизация, которая делает язык очень гибким, но ценой этого является уменьшение производительности и большая вероятность ошибок из-за невнимательности разработчика;
- 3) Платформа NodeJS хорошо заточена под асинхронное выполнения кода;
- 4) Имеется возможность осуществления не блокирующих операций ввода/вывода;
- 5) Высокая производительность;
- 6) Низкий порог входа для начинающих разработчиков;
- 7) Огромная база различных библиотек и пакетов, для решения практически любых задач;
- 8) Возможность написания клиентской и серверной части приложения на одном языке;
- 9) Хорошая поддержка технологии серверного рендеринга страниц, перед отдачей клиенту.

Python – лишен недостатка JavaScript возникновения значительных погрешностей при вычислении математических операций, однако порог вхождения для python-разработчика значительно выше, чем для NodeJS, а это очень важный показатель при коммерческой разработке программных решений.

Сравнительные тесты производительности описанных платформ приведены на данном ресурсе [12].

Таким образом выбор был сделан в пользу технологии NodeJS.

2.2. ХРАНЕНИЕ ДАННЫХ

Среди баз данных явным образом выделяются следующие решения, хорошо зарекомендовавшие себя на больших и высоконагруженных проектах:

- 1) Oracle Database;
- 2) Microsoft Database;
- 3) MongoDB;

- 4) MySQL;
- 5) PostgreSQL.

В качестве основного критерия для выбора можно отметить модель распространения выбранной базы данных. **Oracle Database** и **Microsoft SQL Server** распространяются только на коммерческой основе с платной лицензией, поэтому они в данном случае не подходящий вариант, несмотря на все их плюсы.

MongoDB – является очень производительной базой данных, распространяемой бесплатно. Однако ее ключевой особенностью является то, что это не реляционная база данных, из чего следует, что в ней становятся не возможны сложные манипуляции с данными. Так же большим минусом является отсутствие поддержки транзакций.

MySQL и PostgreSQL – что бы сделать выбрать между этими двумя технологиями достаточно обратить внимание на ключевые моменты, важные при разработки подобных систем:

- 1) За счёт того, что MySQL поддерживает различные реализации механизмов хранения данных имеется большое количество проблем с транзакциями и репликацией данных;
- 2) Сами по себе репликации, при использовании MySQL, выполняются очень долго, даже в асинхронном режиме;
- 3) Документация PostgreSQL, значительно более полная и подробная, в отличии от документации MySQL;
- 4) PostgreSQL соответствует большому числу стандартов SQL, в то время, как MySQL не поддерживает в полной мере даже стандарт SQL-92;
- 5) MySQL значительно проще и легче администрировать, чем PostgreSQL, но этот плюс является следствием серьезного минуса – в MySQL просто значительно меньше возможностей и функций, поэтому сама по себе настройка и поддержка намного проще.

При сравнении баз данных PostgreSQL и MySQL за основу был взят опыт компании «Mail.ru Group», описанный ее разработчиком в статье «PostgreSQL vs MySQL» [24].

Таким образом выбор был сделан в пользу PostgreSQL.

2.3. АВТОРИЗАЦИЯ И АУТЕНТИФИКАЦИЯ

При анализе способов авторизации и аутентификации были выбраны все самые распространены на сегодняшний день алгоритмы, а именно:

- 1) Отсутствие авторизации и аутентификации;
- 2) HTTP аутентификация;
- 3) Аутентификация с помощью форм;
- 4) Аутентификация по одноразовым паролям;
- 5) Аутентификация по ключам доступа;
- 6) Аутентификация по токенам.

Отсутствие возможности авторизации и аутентификации – прежде чем принять или отклонить данный вариант, нужно понять: почему вообще при решении данной задачи нужен, или не нужен механизм авторизации и аутентификации. Так-как одной из особенностей системы является наличие у пользователя доступа к своим данным из любого сервиса, в котором используется данный виджет – есть несколько вариантов решения данной задачи:

- 1) Сделать данные общедоступными, например по идентификационному номеру. Минус данного подхода состоит в том, что пользователю необходимо где-то хранить этот номер и не потерять его;
- 2) Сделать веб интерфейс, в котором пользователь из общего списка мог бы выбрать свои данные;
- 3) Реализовать механизм аутентификации и авторизации. В таком случае, пользователь всегда имел бы удобный доступ к своим данным из виджета расположенного в любом сервисе. Также пользователь может быть

уверен, что никто кроме него не сможет получить доступ на изменение его данных.

Таким образом становится понятно, что отсутствие механизма авторизации и аутентификации не является приемлемым в данном случае.

HTTP аутентификация – и ее подвиды используются преимущественно в корпоративных средах. Механизм работы данной схемы следующий (Так же на рисунке 1 изображен схематичный пример того, как это может работать):

- 1) При обращении пользователя к серверу – пользователь получает ответ от сервера с статус-кодом 401, что значит, что пользователь не авторизован, после чего браузер автоматически предлагает пользователю авторизоваться;
- 2) После ввода данных для авторизации, ко всем запросам к серверу браузер добавляет еще один заголовок: «WWW-Authenticate», с указанием данных для авторизации. Стоит отметить, что такой способ является безопасным, только при взаимодействии пользователя с сервером по протоку HTTPS.

При условии того, что клиентом для сервиса может быть встраиваемый виджет, такой способ авторизации является неприемлемым, так-как сервис, на котором используется виджет может работать по незащищенному соединению.

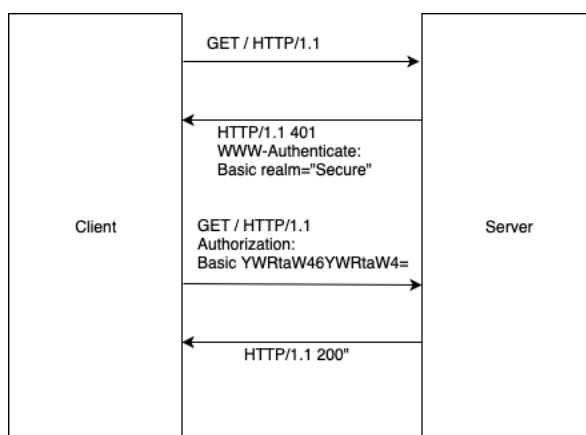


Рисунок 1 – Пример HTTP аутентификации

Аутентификация с помощью форм – наиболее классический подход к аутентификации пользователей. Его суть в следующем (Так же на рисунке 2 показан пример работы данного алгоритма):

- 1) На стороне клиента есть HTML форма, в которую пользователь вводит данные для аутентификации, после заполнения которой, данные отправляются на сервер по средством HTTP POST запроса;
- 2) В случае, если авторизация пройдена успешно – сервер создает запись о текущей сессии, в которой специальным образом сгенерированный токен, ставится в соответствие пользователю;
- 3) При ответе сервера пользователю за запрос об аутентификации сервер дописывает в заголовок «Cookie», только что сгенерированный токен;
- 4) При всех последующих обращениях к серверу используется данный токен, и по нему сервер определяет, что за пользователь к нему обращается.

Данный способ аутентификации, хоть и не является новым, но он все еще широко используется и отлично подходит для данного типа программного обеспечения.

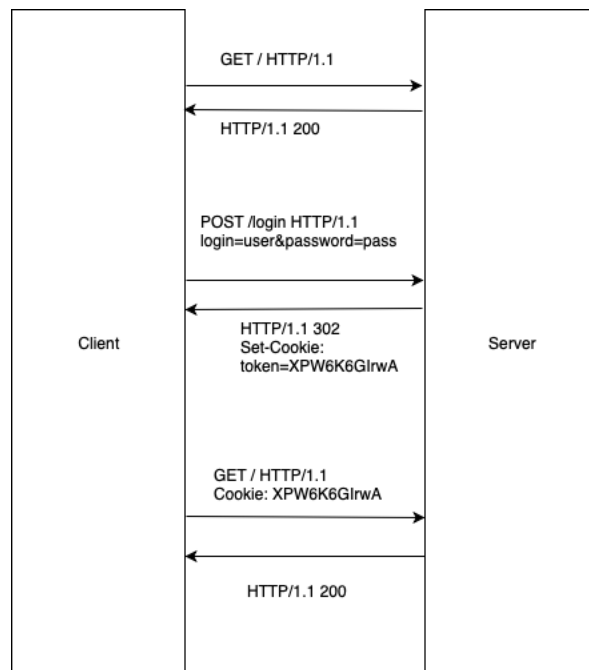


Рисунок 2 – Пример аутентификации с помощью форм

Аутентификация по одноразовым паролям – название данной схемы аутентификации говорит само за себя – ее суть весьма проста при обращении к серверу, генерируется одноразовый пароль, который по средством СМС/электронной почты, или других средств доставляется пользователю, и только с этим паролем пользователь может осуществить вход в систему.

Данный подход широко распространен в современном мире, но на данной этапе создания сервиса он является избыточным.

Аутентификация по ключам доступа – данный тип аутентификации преимущественно применяется при обращении одного веб-сервиса к другому, поэтому он особенно актуален в данном случае, так как предполагается, что один из вариантов распространения сервиса – встраиваемый виджет, а значит обращение к сервису будет происходить со стороннего ресурса. Суть данного подхода очень проста:

- 1) К запросу на сервер дописывается заголовок «Api key», с соответствующим ключом;
- 2) Сервер проверяет ключ, и, если ключ верный и есть в базе данных, значит проходит дальше, иначе он отклоняется.

Смысл данного подхода в создании контроля над входящим трафиком. И наличие так называемого “рубильника”, позволяющего в случае необходимости лишить доступа любой из подключенных сервисов.

Аутентификация по токенам – относительно современный подход к аутентификации пользователей, который применяется повсеместно в наше время. Примерами сервисов с данным механизмом аутентификации могут служить такие сервисы, как «Google», «Facebook», «ВКонтакте» и т.д. Данный подход по своей сути похож на механизм аутентификации с помощью форм, его главное отличие состоит в том, что теперь взаимодействие происходит не только между клиентом и сервером, но сюда добавляется еще и так называемый «identity provider». «identity provider» – по сути своей микросервис, отвечающий за

аутентификацию пользователей. Весь процесс выглядит следующим образом (Так же на рисунке 3 изображен схематичный пример):

- 1) Клиент аутентифицируется любым способом в «identity provider», в ответ на что «identity provider» дает клиенту соответствующий токен (аналог Cookie);
- 2) Теперь при обращении к серверу клиент дописывает соответствующий заголовок к запросу, со значением, равным полученному от «identity provider» токеном;
- 3) Сервер проверяет у «identity provider», достоверность данного токена, и в зависимости от этого обрабатывает или отклоняет запрос клиента.

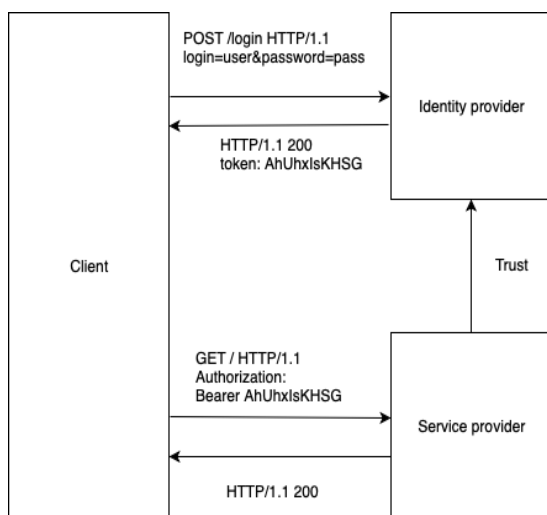


Рисунок 3 – Пример
аутентификации с токеном

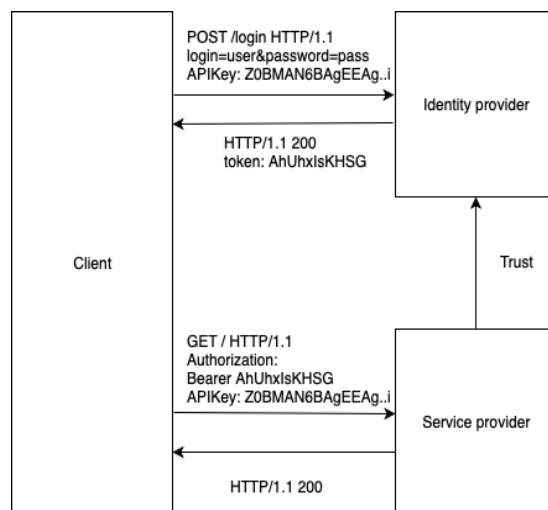


Рисунок 4 – Итоговая схема
аутентификации пользователя в
системе

Данный способ отлично подходит для программных комплексов, которые представляют собой распределенные системы, в которой все части системы будут делегировать функцию аутентификации в единое место.

Таким образом в данном случае выбор пал не на одну конкретную технологию, а сразу на несколько:

- 1) Сторонние сервисы будут аутентифицироваться в системе с помощью ключа доступа;

- 2) Пользователи могут пройти аутентификацию посредством токена, который будет отдаваться «identity provider», аутентификация в котором будет выполняться с помощью HTML форм, схематичное изображение этой схемы представлено на рисунке 4. Единственным допущением на этапе MVP будет тот факт, что функции «identity provider» и «service provider» будут совмещены в одном сервисе.

2.4. ВЕБ СЕРВЕР

Анализ различных ресурсов позволил сделать вывод, что возможны три основных варианта реализации функционала веб сервера:

- 1) Самостоятельная реализация;
- 2) Nginx;
- 3) Apache.

Самостоятельная реализация сервера – не является хорошей идеей, потому что хорошая реализация данного функционала займет значительное количество времени, а количество и качество готовых вариантов окончательно заставляет отказаться от данного варианта.

Apache – некогда очень популярный веб сервер, ныне практически полностью вытесненный **Nginx** (по данным из [11], при написании новых проектов), главным отличием между этими двумя серверами является их подход к выполнению своих функций: Apache для каждого нового запроса создает отдельный независимый процесс, в то время, как Nginx, по своей сути является обычным многопоточным приложением и обрабатывает каждый новый запрос асинхронно в режиме “Reverse proxy” (режим, при котором веб сервер не обрабатывает запросы самостоятельно, а просто перенаправляет их, согласно своей конфигурации), тем самым сильно экономя ресурсы. Таким образом при достижении максимального количества доступных процессов – Apache начнет отклонять новые запросы, а Nginx просто будет добавлять их в очередь. Так же Nginx значительно проще конфигурировать. Таким образом, выбор сделан в

пользу Nginx, из-за более удачного подхода к обработке входящих соединений, в условиях современного мира.

2.5. ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ ТРАФИКА

По сути никакого выбора в данной категории нет, так как протокол TLS пришел на смену протоколу SSL, так как все версии протокола SSL были скомпрометированы.

Суть же протоколов безопасности состоит в следующем:

- 1) Шифрование – шифрование всего трафика на транспортном уровне модели OSI, таким образом, для самого приложения ничего не меняется, однако снаружи невозможно понять, что именно передается;
- 2) Аутентификация – проверка авторства передаваемой информации;
- 3) Целостность – обнаружение подмены информации подделкой.

Не будем останавливаться подробно на том, как работает TLS протокол, отметим лишь тот факт, что для его работы необходимо использовать соответствующий TLS сертификат.

2.6. ОБЕСПЕЧЕНИЕ ГИБКОГО ДОСТУПА К ФУНКЦИОНАЛУ СЕРВИСА И ХРАНИМЫМ ДАННЫМ

REST – практически стандарт в современном мире веб сервисов и за все то время, что REST – существует он отлично себя показал. И хотя сравнивать REST и GraphQL не совсем корректно, так как REST – это подход к проектированию веб сервисов, а GraphQL это новый подход к написанию запросов. Одно из требований к RESTful сервису является разделение ресурсов. В REST ресурсом является все то, чему можно дать имя. Каждый ресурс в REST должен быть идентифицирован посредством стабильного идентификатора, который не меняется при изменении состояния ресурса – URI.

Пример запроса для RESTful сервиса:

GET /heroes/666

Пример ответа для RESTful сервиса на вышеописанный запрос, в формате JSON:

```
{
  "name": "Darth Vader",
  "features": {
    "force": "darkside",
    "swordColor": "red"
  }
}
```

Если же добавить в эту цепочку GraphQL, то запрос изменится следующим образом:

```
{
  hero(id: 666) {
    name
    features
  }
}
```

Теперь, когда стало понятно, чем отличаются запросы в первом и втором случаях, нужно разобраться, так в чем же разница и зачем нужен GraphQL? Вот пример простого кейса, когда явным образом проявятся недостатки стандартного подхода, используемого в RESTful сервисах – допустим нам нужно запросить информацию о трех разных сущностях. При стандартном подходе это бы означало три независимых запроса к серверу, при использовании же GraphQL запрос по-прежнему остался бы один, но принял бы, например такой вид:

```
{
  hero(id: 666) {
    name
    features
  }
  planet(id: 777) {
    name
    features
  }
  spaceship(id: 888)
}
```

С помощью такого запроса было запрошено сразу три ресурса, причем только с той информацией, которая нас интересует. Интересной особенностью данного подхода является то, что ресурсы, которые мы запрашиваем могут

находится в разных базах данных, на разных серверах и т.п., однако с точки зрения клиента – это все еще всего 1 запрос.

Второй важной особенностью GraphQL является то, что на стороне сервера доступные ресурсы описываются в виде графа, с помощью которого мы можем производить практически любые манипуляции с данными. Еще одним важным элементом любого графа являются ребра, связывающие вершины, и в GraphQL они не остались незамеченными, тут мы их можем использовать, например, для ограничений доступа и проверки прав, на доступ к заданному ресурсу.

Пример описания графа:

```
type Root {
  hero: Hero
}
type Hero {
  id: ID
  name: String
  features: Feature
}
type Feature {
  force: String
  swordColor: String
}
```

GraphQL уже сейчас постепенно начинает внедряться в огромное число новых проектов, потому что его преимущества позволят в значительной степени ускорить и упростить разработку клиент-серверных приложений, т.к. при нормальном описании всех сущностей backend и frontend разработчикам не нужно договариваться о том как получить те или иные данные – они доступны все и сразу, а в автоматически генерируемой GraphQL документации, на всякий случай все еще раз описано.

Возможность запроса нескольких данных одним запросом или несколькими асинхронными появилась не только на уровне синтаксиса запроса, но и на уровне протокола. Очень большую популярность и распространение набирает протокол **HTTP/2**, который стал очень легко применим в следствии использования шифрования трафика с помощью протокола TLS. Все дело в том, что HTTP/2 –

бинарный протокол и это было существенной проблемой для его применения, но с тех пор, как каждый сервис использует протоколы шифрования – весь трафик уже перешел в бинарный вид и протокол HTTP/2 стал легко применим. Его основные преимущества перед протоколом HTTP/1.1 следующие:

- 1) В HTTP/2 сервер имеет право послать то содержимое, которое ещё не было запрошено клиентом. Это позволит серверу сразу выслать дополнительные файлы, которые потребуются браузеру для отображения страниц, без необходимости анализа браузером основной страницы и запрашивания необходимых дополнений;
- 2) Мультиплексирование множества запросов в одном соединении TCP;
- 3) Сжатия данных в заголовках HTTP.

Таким образом при написании данного программного обеспечения будет сделан выбор в пользу протокола HTTP/2 и языка запросов GraphQL.

2.7. ИНСТРУМЕНТ ДЛЯ КЭШИРОВАНИЯ ДАННЫХ ЗАПРОСОВ

Два основных инструмента, широко используемых многими крупными компаниями для кэширования данных:

- 1) Redis;
- 2) Memcached.

Кэширование это очень востребованная в современном мире технология, которая позволяет существенно снизить нагрузку на любую систему, путем хранения в оперативной памяти «ответов», на частые и сложные запросы к системе. Поэтому, учитывая простоту ее реализации, и тот эффект видимого ускорения работы системы, который она может дать конечному пользователю, имеет смысл добавить ее в систему сразу, еще на этапе MVP, для того что бы в дальнейшем с этим не возникло проблем.

Оба инструмента: Memcached и Redis являются «ключ-значение», базами данных, которые хранят все свои данные в оперативной памяти. За счет такой архитектуры они обеспечивают молниеносный доступ к данным за время $O(1)$.

Memcached – очень мощный инструмент, используемый для кэширования различных данных в оперативной памяти, но в отличие от **Redis**, Memcached обладает значительно меньшим количеством возможностей и значительно более скромной документацией. Из ключевых преимуществ Redis, можно выделить следующие:

- 1) Поддержка большого числа типов данных;
- 2) Гибкая настройка стратегий очистки ключей;
- 3) Наличие возможности создавать вложенные ключи;
- 4) Хорошо приспособлен к хранению множеств и сложных структур данных.

Выше приведен список лишь тех преимуществ Redis, которые имеют значение при реализации задачи кэширования данных, но это далеко не весь список. Поэтому в данном случае выбор будет сделан в пользу Redis.

2.8. ИНСТРУМЕНТ ДЛЯ ОБЕСПЕЧЕНИЯ ВЗАИМОДЕЙСТВИЯ МИКРОСЕРВИСОВ

Для того что бы взаимодействие между отдельными частями системы было наиболее быстрым, нужно создать некий канал связи, который бы позволил каждому элементу системы обмениваться сообщениями с другими ее частями, не тратя на это много времени и ресурсов.

Для решения задачи взаимодействия между элементами системы не редко применяют **Redis**, как и для кэширования данных. Redis способен обеспечить данную возможность, за счет того, что в нем реализован механизм публикации событий и подписки на события. Учитывая все ранее перечисленные особенности Redis, можно прийти к выводу, что он отлично справится с данной задачей.

Однако в данном случае, есть более специализированный инструмент для решения поставленной задачи – **RabbitMQ**. RabbitMQ – брокер сообщений, который давно зарекомендовал себя как надежный инструмент, позволяющий решить задачу взаимодействия между элементами системы. Из ключевых особенностей работы RabbitMQ можно выделить следующие:

- 1) Надежность;
- 2) Гибкая система обработки сообщений;
- 3) Возможность выгрузки данных на диск;
- 4) Высокая стабильность;
- 5) Наличие плагинов;
- 6) Хорошая документация;
- 7) Наличие механизма подтверждения обработки сообщений.

Таким образом в данном случае выбор будет сделано в пользу профильного программного обеспечения – RabbitMQ.

2.9. РЕЗЮМЕ

Проведя анализ технологий, для решения поставленной задачи были выбраны следующие технологии:

- 1) Платформа – NodeJS;
- 2) Хранение данных – PostgreSQL;
- 3) Авторизация и аутентификация:
 - a. Аутентификация с помощью форм;
 - b. Аутентификация по ключам доступа;
 - c. Аутентификация по токенам;
- 4) Веб сервер – Nginx;
- 5) Обеспечение безопасности трафика – протокол TLS;

- 6) Обеспечение гибкого доступа к функционалу сервиса и хранимым данным:
 - a. Язык запросов – GraphQL;
 - b. Протокол – HTTP/2
- 7) Инструмент для кэширования данных запросов – Redis;
- 8) Инструмент для обеспечения взаимодействия микросервисов – RabbitMQ.

3. РЕАЛИЗАЦИЯ

Реализация программного комплекса обычно начинается с проектирования. Первым этапом проектирования станет определение необходимых компонентов системы и способов их взаимодействия. Так как, по изначальной задумке, предполагается создание именно MVP – нет большой необходимости делать приложение полностью построенное на микросервисной архитектуре, поэтому сам сервис будет представлять собой монолит. Данное решение вызвано в первую очередь тем, что плюсы микросервисной архитектуры начинают проявляться только на больших проектах, с большими командами. Если же проект еще на стадии MVP – микросервисная архитектура, кроме сильного увеличения стоимости обслуживания и поддержки продукта ничего не даст.

Теперь нужно определить из каких частей должно состоять MVP:

- 1) HTTP сервер;
- 2) Механизм аутентификации и авторизации;
- 3) Файловое хранилище;
- 4) Механизм подтверждения регистрации пользователей, через email адрес.

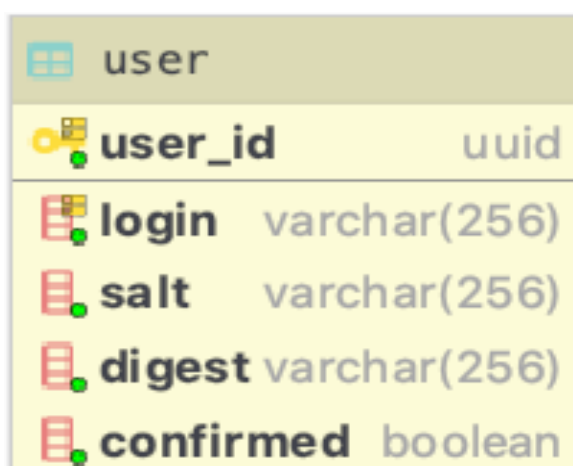
3.1. МОДЕЛИ ДАННЫХ

Для того что бы создать схему базы данных нужно первым делом определиться, как и сущности должны быть созданы:

- 1) Модель пользователя;
- 2) Модель сессии;
- 3) Модель объявления;
- 4) Модель события.

Модель пользователя должна содержать следующий набор данных:

- 1) id – уникальный идентификатор пользователя, служит для того, что бы быстро осуществлять поиск пользователя по базе данных;
- 2) email – уникальное имя пользователя, так же используется для подтверждения аккаунта пользователя, так-как, после регистрации, на данный email будет приходить письмо для подтверждения регистрации;
- 3) Соль – случайная последовательность битов, использованная при генерации дайджеста;
- 4) Дайджест – зашифрованный пароль пользователя;
- 5) Флаг, обозначающий подтвержден данный аккаунт или нет.



The diagram shows a database table named 'user'. It contains five fields: 'user_id' (primary key, uuid), 'login' (varchar(256)), 'salt' (varchar(256)), 'digest' (varchar(256)), and 'confirmed' (boolean).






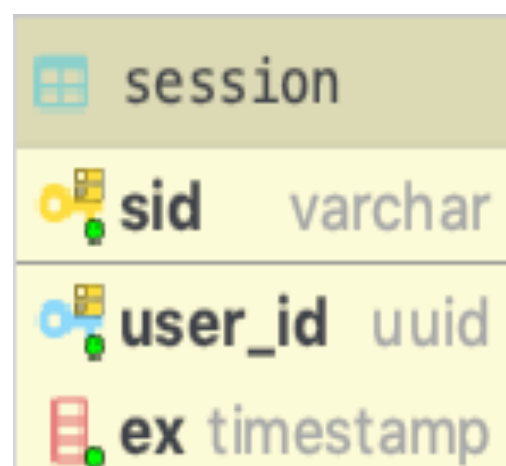
user	
	user_id uuid
	login varchar(256)
	salt varchar(256)
	digest varchar(256)
	confirmed boolean

Рисунок 5 – Модель пользователя



The diagram shows a database table named 'session'. It contains four fields: 'sid' (primary key, varchar), 'user_id' (foreign key, uuid), and 'ex' (timestamp).




session	
	sid varchar
	user_id uuid
	ex timestamp

Рисунок 6 – Модель сессии

Модель сессии состоит из:

- 1) id – уникальный идентификатор сессии;
- 2) id – пользователь, к которому эта сессия относится;
- 3) Временная метка, до которой эта сессия может считаться.

Несмотря на то, что предполагается использование токенов для авторизации и аутентификации пользователей, таблица сессий позволит в дальнейшем обеспечить более высокий уровень безопасности данных в приложении. Так как

не трудно представить себе ситуацию, когда токен пользователя уже был выпущен, но при этом права доступа пользователя были изменены.

Модель объявления включает в себя:

- 1) `id` – уникальный идентификатор объявления;
- 2) `rooms` – массив объектов, хранящий в себе информацию о комнатах;
- 3) `shapes` – информация, необходимая виджету для отрисовки плана;
- 4) Временная метка, когда данное объявление было размещено.

Модель объявления является самым узким местом во всей схеме базы данных, так-как в процессе доработок и модификаций MVP – это самый не постоянная часть базы данных, поэтому в данном случае нужно выбрать наиболее гибкий и удобный способ хранения данных. В данном случае идеальным вариантом будет сохранение объекта “`shapes`”, просто в виде строки, так-как на стороне сервера с ним не нужно производить никаких манипуляций. Но метаинформация, хранимая в массиве “`Rooms`”, может быть использована при поиске и фильтрации объявления на стороне сервиса. Поэтому идеальным решением в данном случае будет сохранение данных в формате “`jsonb`”, данный формат позволяет хранить данные любой структуры и при этом выполнять запросы с учетом данных, которые хранятся в полях сохраненного объекта.

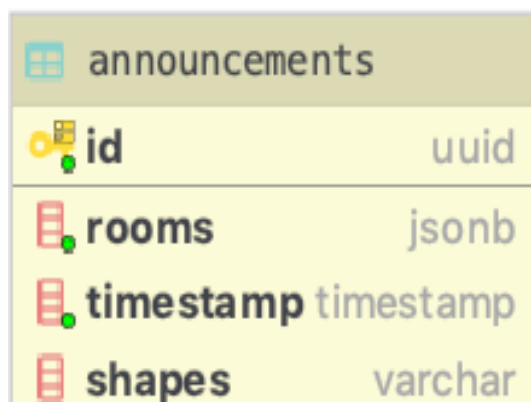


Рисунок 7 – Модель объявления

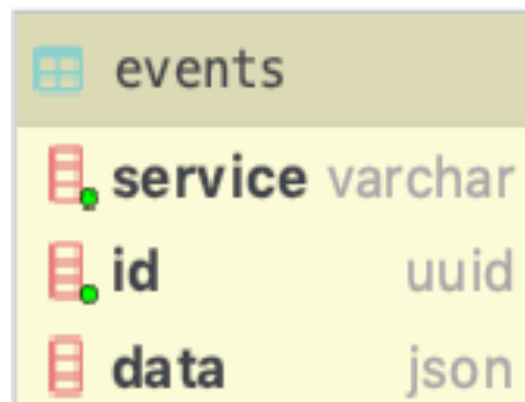


Рисунок 8 – Модель событий

Модель события:

- 1) Название сервиса, получается данного события;
- 2) id – уникальный идентификатор события;
- 3) data – информация, которая должна быть передана сервису, вместе с сообщением о событии. Так-как информация может быть совершенно разная для каждого сервиса, то по аналогии с объявлением данное поле будет храниться в формате “json”.

Несмотря на то, что подразумевается, что архитектура данного программного комплекса будет монолитной, некоторый функционал, все же лучше вынести за его пределы. На этапе MVP это будут только сервис, отвечающий за рассылку писем на электронные почты пользователей, а также сервис, отвечающий за хранение файлов(фотографий).

Исходя из, определенных выше моделей данных, получаем следующую схему базы данных (рисунок 9).

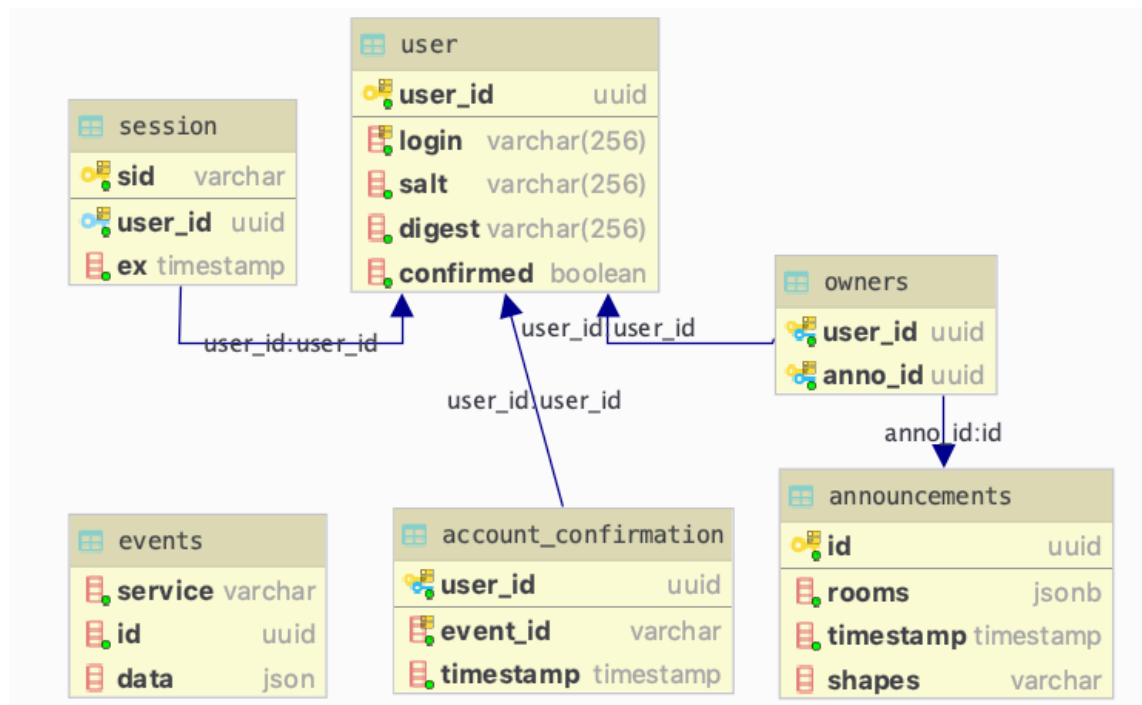


Рисунок 9 – Общая схема базы данных

Дополнительные таблицы “account_confirmation” и “owners”, выполняют вспомогательные функции. Таблица “account_confirmation” хранит соответствие события активации аккаунта пользователя через email и id самого пользователя. Таблица “owners”, помогает установить соответствие между пользователем и его объявлениями.

3.2. GRAPHQL

Теперь, когда есть схема базы данных (рисунок 9), можно адаптировать необходимые модели данных для схемы данных GraphQL. Схема данных в языке запросов GraphQL представляет собой граф, который начинается с корневого запроса на получение данных – “query”, и с корневого запроса на изменение данных – “mutation”.

На языке GraphQL, вершина графа описывается следующим образом:

```
schema {  
  query: RootQuery  
  mutation: RootMutation  
}
```

Где, после “:” идет, тип данных, соответствующий данному полю.

Начнем с рассмотрения ветви запросов на получение данных:

```
type RootQuery {  
  user(login: String!): User  
  announcements(id: String!): [AnnouncementReq]  
}
```

В данном случае тип “RootQuery”, содержит два поля:

- 1) user – поле, при обращении к которому нужно передать обязательный аргумент (об этом говорит восклицательный знак, после описания типа аргумента) “login”, и в случае, если данный пользователь существует, вернет объект типа “User”;
- 2) announcements – поле, при обращении к которому нужно передать обязательный аргумент “id”, и в случае, если данный пользователь

существует, вернет массив относящихся к нему данных типа “AnnouncementReq”.

На языке GraphQL типы данных “User” и “AnnouncementsReq” описываются следующим образом:

```
type User {  
    user_id: String  
    login: String  
}  
  
type AnnouncementsReq {  
    shapes: String  
    rooms: [RoomsType]  
    id: String  
    timestamp: String }
```

Тип “AnnouncementsReq”, содержит еще один сложный составной тип, который описывается следующим образом:

```
type RoomsType {  
    date: int  
    tags: [String]  
    walls: [String]  
    equipment: [String]  
    id: String  
    photos: [String]  
}
```

Типы данных “User” и “AnnouncementReq”, соответствуют описанию аналогичных типов, представленных в базе данных, тип “RoomsType”, в свою же очередь отражает актуальное состояние поля данных “rooms” в схеме базы данных, с типом “jsonb”. Его поля имеют следующее значение:

- 1) date – временная метка в миллисекундах, когда была добавлена данная комната;
- 2) tags – массив тэгов, относящихся к данной комнате;
- 3) walls – массив, описывающий стены, данной комнаты;
- 4) equipment – массив, описывающий предметы, расположенные в данной комнате;
- 5) id – уникальный идентификатор данной комнаты;
- 6) photos – массив id, принадлежащих фотографиям, относящимся к данной комнате.

Теперь рассмотрим ветвь с запросами на изменение данных, которая имеет тип “RootMutation”, который на языке GraphQL описывается следующим образом:

```
type RootMutation {  
  registration(input: UserInput!): String  
  login(input: UserInput!): String  
  logout: String  
  loginGoogle(accessToken: String!): String  
  announcement(input: AnnouncementInput!): AnnouncementResp  
  oneTimeLink(input: OneTimeLinkInput!): String  
}
```

Где ключевое слово “input” обозначает тип данных, который нужно указать в качестве аргумента для данного запроса;

Рассмотрим более подробно значение каждого поля:

- 1) registration – с типом обязательного аргумента “UserInput”, и возвращаемым значением типа String. Данный запрос позволяет зарегистрировать нового пользователя, и в случае успешной регистрации, в качестве ответа возвращает email только что зарегистрированного пользователя;
- 2) login – запрос за создание токена и записи в таблицу сессий, входным параметром является аргумент типа “UserInput”;
- 3) logout – в случае, если авторизированный пользователь обратится к данному полю, то запись о его сессии будет удалена из таблицы сессий, и его токен станет не действителен;
- 4) loginGoogle – поле аргументом которого, является строка: “accessToken”, предоставляемая OAuth серверами компании “Google”. В случае, если предоставленный “accessToken” окажется верным, то пользователь будет авторизован в системе, с помощью своего аккаунта Google;
- 5) announcements – поле с аргументом типа “AnnouncementInput”, и типом возвращаемого значения “AnnouncementResp”, запрос к данному полю позволяет создать новое объявление в базе данных;

- 6) oneTimeLink – запрос на получение от файлового сервиса одноразовой ссылки на скачивание или загрузку файла, с аргументом типа “OneTimeLinkInput”.

Ниже приведено описание типов входных данных и возвращаемых значений, описанные выше, на языке GraphQL:

```
input UserInput {
  login: String!
  password: String!
}

input AnnouncementInput {
  shapes: String!
  rooms: JSON!
}

type AnnouncementResp {
  id: String
}

input OneTimeLinkInput {
  type: String!
  name: String
  isZipped: String
  directory: String
  id: String
  lifeTime: String
}
```

3.3. HTTP СЕРВЕР

Теперь, когда понятно, какие данные хранятся в базе данных и описана схема взаимодействия с ними, нужно реализовать сам HTTP сервер, который будет непосредственно всем этим управлять. Так-как в качестве платформы был выбран “NodeJS”, для реализации функционала HTTP сервера доступно чрезвычайно большое число различных фреймворков. Однако в данном случае выбор будет остановлен на фреймворке “Express”. Данный выбор обуславливается в первую очередь тем, что данный фреймворк достаточно популярен, имеет свое собственное большое сообщество и очень серьезную систему различных плагинов. Так же большим его плюсом является гибкость масштабируемость и документация.

3.4. МЕХАНИЗМ АВТОРИЗАЦИИ И АУТЕНТИФИКАЦИИ

Рассмотрение данного вопроса начнем с процесса создания нового пользователя. При получении запроса на регистрацию, на сервер, вместе с запросом, приходят и данные пользователя в виде email-адреса и пароля. Если email-адрес и пароль проходят проверку, и являются допустимыми, то далее производится попытка создания нового пользователя и сохранения его в базу данных. Процесс создания нового пользователя состоит из следующих этапов:

- 1) Получение запроса на регистрацию нового пользователя;
- 2) Проверка предоставленных email-адреса и пароля:
 - a. Проверка email-адреса с помощью соответствующего регулярного выражения;
 - b. Проверка, что пароль состоит более чем из 4 символов, и не содержит запрещенных символов;
- 3) Проверка, не существует ли в базе данных пользователя с указанным адресом электронной почты;
- 4) Если все предыдущие проверки пройдены, тогда происходит шифрование пароля, перед сохранением пользователя в базу данных. Шифрование происходит с использованием алгоритма: “SHA256”, после чего, полученная последовательность битов приводится к строке, в кодировке “base64”, которую уже можно сохранить в базу данных;
- 5) Пользователь сохраняется в базу данных;
- 6) Создается событие на подтверждение аккаунта пользователя (генерируется одноразовая ссылка, по которой пользователь должен перейти, чтобы подтвердить свой аккаунт), которое с помощью шины данных RabbitMQ, доставляется к сервису по рассылке электронных писем;
- 7) После того как пользователь подтвердит свой аккаунт – регистрация считается завершенной.

Пример регистрации нового пользователя представлен на рисунке 10.

После прохождения процесса регистрации пользователь может осуществить вход в систему. Для этого пользователь предоставляет те данные, которые он указывал при регистрации, и если они верны, то в ответ сервер отправит



Рисунок 10 – Регистрация нового пользователя,
через интерфейс GraphiQL

соответствующий JWT-токен. JWT-токен, это специальным образом сгенерированная строка, которая генерируется на основании следующих данных:

- 1) id пользователя;
- 2) email адрес пользователя;
- 3) Уровень доступа пользователя пользователя (только чтение, только запись или, и то и другое);
- 4) Секретное кодовое слово, нужно для расшифровки данного токена.

Пример аутентификации пользователя представлен на рисунке 11.

Теперь, если при обращении к ресурсам, хранимым на сервере, пользователь укажет в запросе следующий заголовок:

Authorization: "Bearer <JWT-token, полученный после аутентификации>"

Сервер сразу определит, что за пользователь обращается за данными, а также его уровень доступа.

Уровень доступа, содержащийся в теле JWT-token'а, используется созданными директивами GraphQL. При попытке перейти по ребру графа к

запрашиваемому полю, директивы GraphQL проверяют, соответствует ли требуемый вершиной графа уровень доступа с предоставленным пользователям уровнем доступа, если нет, то пользователь получит соответствующую ошибку.

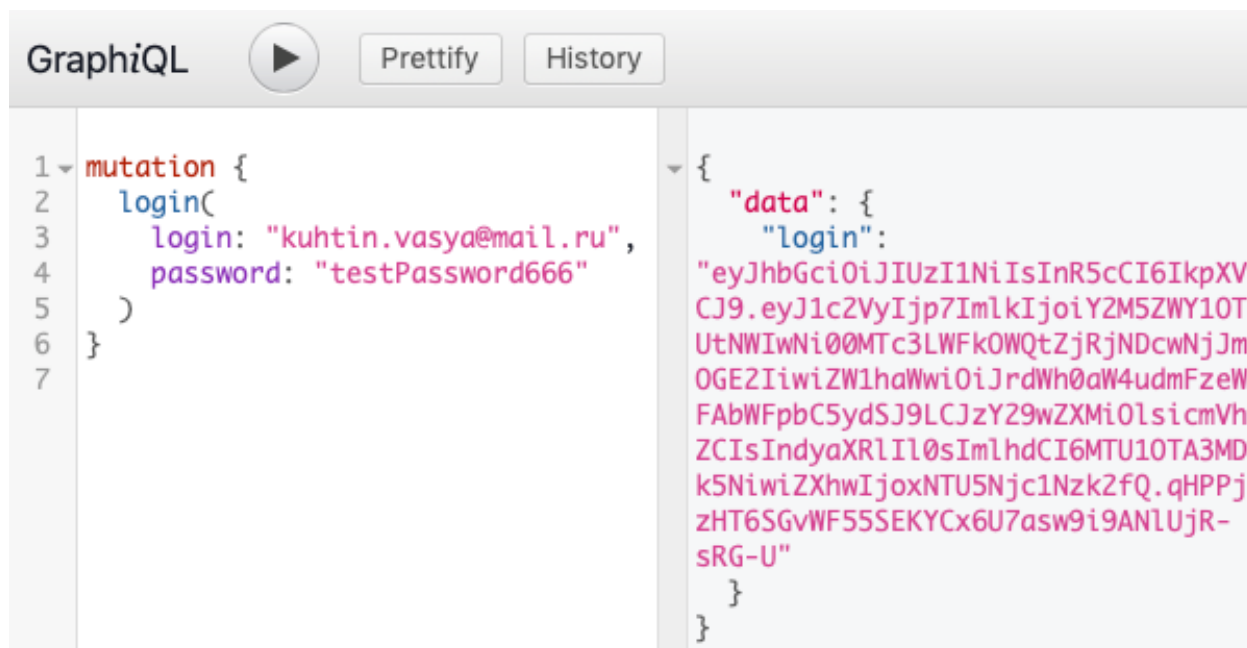


Рисунок 11 – Пример аутентификации пользователя, через интерфейс GraphQL

3.5. ФАЙЛОВОЕ ХРАНИЛИЩЕ

Для реализации возможности прикрепления к планам помещения фотографий – необходимо иметь возможность безопасно сохранять фотографии на сервере. Для этого был реализован соответствующий файловый сервис, так же на базе платформы NodeJS, с фреймворком “Express”. Отличиям от основного сервера в данном случае стала используемая база данных: нет необходимости выполнения сложных выборок из базы и совершения операция в режиме транзакции, поэтому в качестве базы данных была выбрана база данных MongoDB. Суть данного сервиса состоит в том, что он предоставляет удобный интерфейс загрузки и скачивания файлов по одноразовым ссылкам, при этом получить эти одноразовые ссылки можно только через главный сервер. Если

рассматривать поэтапно, то процесс загрузки файла на сервер будет выглядеть следующим образом:

- 1) Аутентификация пользователя в системе;
- 2) Запрос пользователем одноразовой ссылки на загрузку файла;
- 3) Прикрепление файла к запросу при переходе по полученной ссылке.

Процесс скачивания файла отличается лишь типом запрашиваемой одноразовой ссылке, и отсутствием необходимости что-либо дополнительно указывать, при переходе по одноразовой ссылке. Пример запроса на получение одноразовой ссылке указан на рисунке 12.

Для того что бы получить одноразовую ссылку, нужно отправить на главный сервис запрос подобный тому, который продемонстрирован на рисунке 12. В зависимости от того, какой тип операции клиент запрашивает у файлового сервиса – нужно указать правильное значение поля “type”, и указать соответствующие сопутствующие параметры. На данный момент возможно получение двух типов одноразовых ссылок: ссылка на скачивание файла и ссылка на загрузку файла на сервер.

```
1 mutation {  
2   oneTimeLink(  
3     type: "download"  
4     id: "5cebab76065a5e337fa4dae9"  
5   )  
6 }
```

schema 

```
1 {  
2   "data": {  
3     "oneTimeLink":  
      "/public/file/cGkhJnfzPxP2h5nWnQxmW3E0A9itAPGr  
      U3vg1jDhFrSLBt4fxohbXTZbAeUBxUeNl19yfINSCPYYt  
      1Y1KhwwlPAjwPTwVxHdEjwd00ttTxcUunxzypl1bQWtXG  
      BCKBFWc6UdyZsErVzzpLAqGI3Rr3Y6FIoaGaLzT9sZ2nxM  
      8xmtbNCRk5GqX19wWWhMuMNYeG5lueqeQZjM9yzRJfrPow  
      D7Dv8GCfpI4oa8Q6HHMLYszMy9Yhht4xynoMgltNu0VlZ0  
      CdjxfIqMKmLCJDq9nQypNlb3GfeJSrLigwKMrAHyAFLanP  
      Z5H3fACM8XvEiQUCBRGvw"  
4   }  
5 }
```

Рисунок 12 – Пример получения одноразовой ссылки на скачивание файла по id

Как видно из примера, продемонстрированного на рисунке 12 – запрос на скачивание файла имеет следующий вид:

```
mutation {
  oneTimeLink(
    type: "download"
    id: "<file_id>"
    lifeTime: "<timestamp>"
  )
}
```

Запрос же на загрузку файла содержит следующий набор возможных параметров:

```
mutation {
  oneTimeLink(
    type: "upload"
    name: "<file_name>"
    isZipped: <true/false>
    directory: "<directory_name>"
    lifeTime: "<timestamp>"
  )
}
```

Как видно, в данном случае указываются следующие параметры:

- 1) name – имя файла, с которым он будет сохранен на сервере;
- 2) isZipped – флаг, который говорит о том, нужно ли сжимать полученный файл;
- 3) directory – указывается, если на сервере файл необходимо хранить в определенной директории;
- 4) lifetime – время жизни одноразовой ссылки.

В ответ на данный запрос, клиент получит одноразовую ссылку, с помощью которой на сервер можно загрузить нужный файл.

Механизм работы по одноразовым ссылкам позволяет сделать использование хранилища безопасным, потому что из вне нет никакой возможности загрузить или выгрузить данные с файлового сервиса или в него, без валидной одноразовой ссылки. Все это достигается за счет разделения API файлового сервиса на 2 части: внешнюю и внутреннюю. Внешняя часть отвечает

непосредственно за выполнение функций файлового сервиса через одноразовые ссылки и только через них. Процесс генерации одноразовых ссылок происходит только через внутренние API, к которому есть доступ только у главного сервиса.

3.6. СЕРВИС РАССЫЛКИ ЭЛЕКТРОННЫХ ПИСЕМ

Сервис для рассылки электронных писем, как и файлов сервис представляет собой отдельную программную единицу. Однако в отличии от файлового сервиса с сервисом рассылки электронных писем имеет только внутренний API, потому что во внешнем API – тут объективно нет необходимости. Взаимодействие с сервисом рассылки электронных писем (далее mailer), осуществляется через брокера сообщений – RabbitMQ. Сам по себе mailer – является NodeJS приложением, использующим популярную библиотеку для отправки электронных писем, существующую на рынке уже около 9 лет – “nodemailer”, данная библиотека предоставляет простой и удобный интерфейс для реализации функционала сервиса, поэтому нет смысла на этом останавливаться. На данном этапе из функций mailer’a реализован только функционал рассылки одноразовых ссылок для подтверждения аккаунта пользователя. Если рассматривать пошагово, то данная процедура выглядит следующим образом:

- 1) Пользователь регистрирует новый аккаунт;
- 2) В таблицу “events”, заносится запись, содержащая информацию о данном событии, а в таблицу “account_confirmation” заносится запись, ставящая в соответствие данное событие и конкретного пользователя, чей аккаунт необходимо подтвердить. Это все необходимо для того, чтобы максимум информации, который можно было бы получить из одноразовой ссылки – информация о том, что это за событие, т.к. с конкретным пользователем оно связывается уже только на сервере. Так же эта информация сохраняется с той целью, что если по каким-то причинам mailer не

получил запрос на отправку письма – была возможность это понять и повторить попытку через определенное время;

- 3) Далее запрос на отправку электронного письма с помощью брокера сообщений попадает в mailer, который отправляет письмо на указанный пользователем адрес электронной почты;
- 4) После того как пользователь перейдет по ссылке для подтверждения почты – все записи о данном событии удаляются из базы данных.

Пример текста письма, которой отправляется mailer’ом:

Подтверждение регистрации:

Для подтверждения учетной записи пройдите по следующей ссылке
<http://localhost:8080/verify/9dc05910-73e4-42c2-b217-0523ecbd1fcf>

Не отвечайте на это сообщение, т.к. оно было сгенерировано автоматически!

Однако в целом данный процесс выглядит довольно тривиально. Главная сложность в данном случае заключается именно в создании надежной шины данных и организации взаимодействия между частями системы. Для того что бы все работало именно так, как это ожидается – нужно провести детальную настройку работы RabbitMQ. Рассмотрим спектр настроечных параметров, которые были задействованы при создании шины данных, а также причины их использования:

- 1) Для каждого сервиса, взаимодействующего с другими через шину данных, необходимо создать как минимум две очереди для обмена сообщениями: одна для сообщений, поступающих в сервис, другая для исходящих сообщений. Это нужно для того, чтобы отличать сообщения по тому, для кого они предназначены. Иначе может получиться так, что сервис будет их отправлять и сам же получать, либо другие путаницы;
- 2) Далее необходимо активировать опции “durable” и “persistent” данные параметры позволят не потерять отправленные сообщения в тех случаях, если получатель будет не доступен, или в том случае, если сам брокер по какой-либо причине прекратит работу;
- 3) При отправке, каждое сообщение дополняется специальным системным параметром: “correlationId”, в который, в данном случае, записывается id

события, он необходим для того, чтобы четко понимать к какому именно событию относится данное сообщение;

- 4) И последний параметр, однако являющийся одним из наиболее важных является “messageAcknowledgement”, смысл этого параметра предельно прост – когда получатель забирает сообщение из очереди, сообщение продолжает храниться в ней, до тех пор, пока получатель не отправит в ответ сообщение о подтверждении получения сообщения. Данный подход позволяет гарантировать факт того, что сообщение было получено.

3.7. СЛОЙ КЭШИРОВАНИЯ

Учитывая то, какую большую роль в жизни любого современного человека играет интернет, и объемы информации, с которыми приходится работать – трудно представить современный интернет сервис, в котором не был бы так или иначе реализован механизм кэширования. Механизм кэширования схож с механизмом селекторов, широко используемых в профессиональной среде веб-разработчиков. Селектор – функция, которая при получении одинаково набора параметры считает результат всего лишь раз, а все последующие обращения возвращает ссылку на предыдущий результат своей работы. В случае же с кэшированием – обычно происходит обращение к базе данных, хранящей все свои данные в оперативной памяти по принципу ключ-значение. Из-за такого принципа работы, как правило время поиска по ключу в таких системах равно $O(1)$. И так теперь стало понятно, что кэширование по своей сути все не сложная – при очередном обращении к базе данных мы просто сначала проверяем – а нет ли в Redis результат этого запроса, и если он есть – сразу его отдаем, избегая взаимодействия с базой, как продемонстрированное на рисунке 13, где “pg” – библиотека для взаимодействия с базой данных PostgreSQL, а “node-redis”, библиотека для взаимодействия с базой данных Redis.

В общем и целом, все выглядит довольно просто:

- 1) Выполнить запрос один раз;
- 2) Сохранить результат выполнения запроса в Redis;
- 3) Все последующие ответы на данный запрос получать из Redis.

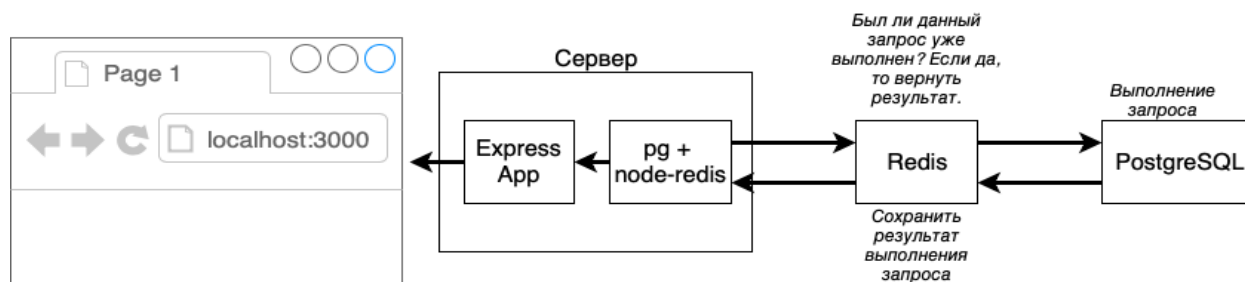


Рисунок 13 – Общая схема взаимодействия компонентов с базой данных и слоем кэширования

Сложностью данного подхода является придумывание алгоритмов генерации ключей для каждого запроса, однако в данном случае ключом для каждого запроса может являться текст самого запроса. Но даже при таком подходе к генерации ключей у данного подхода все еще хватает проблем. Итак, данный подход обладает следующей проблемой: кэшированные значения никогда не удаляются и не обновляются, что может стать проблемой, если в базе данных были произведены какие-либо манипуляции с данными. На первый взгляд решением данной проблемы может стать простое добавление проверки при каждом запросе на изменение данных в базе удалять соответствующие пары ключ-значение из Redis. Но если в будущем зависимости между данными окажутся гораздо более сложными чем сейчас, то как понять какие именно записи нужно удалить? Однако в Redis подобного рода проблемы решаются очень просто:

- 1) Установить время жизни каждой записи можно простым добавлением параметра “ex”, к запросу, со значением равным времени жизни записи в секундах. Например, 10 секунд, в нашем случае;
- 2) Решением проблемы с сложными зависимостями в данных – стали вложенные ключи. Суть способа проста: есть внешний ключ, с помощью которого можно удалить все связанные с ним записи. В данном случае внешним ключом будет служить id пользователя, и в случае, если какие-то из данных этого пользователя изменятся, все связанные с ним сохраненные данные будут сброшены.

Итоговый пример того, как может выглядеть хранение данных в Redis представлен на рисунке 14.

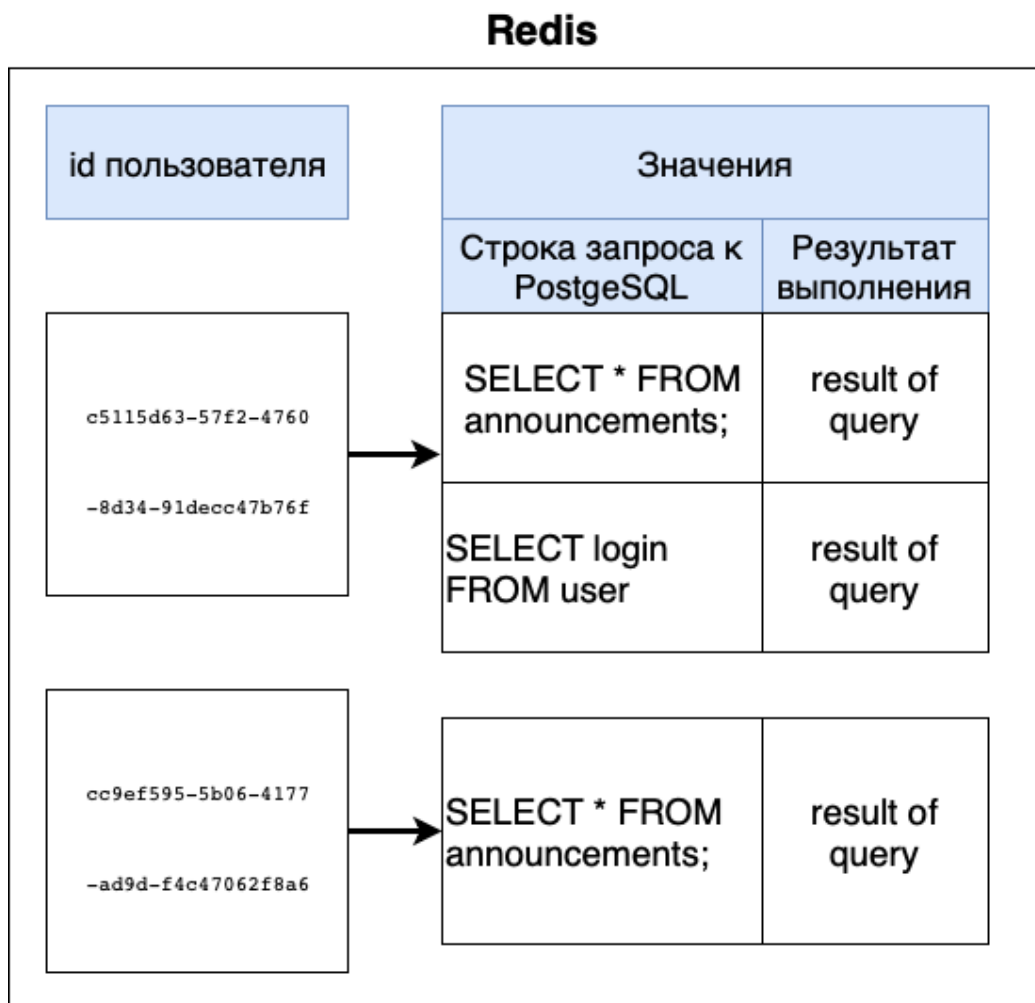


Рисунок 14 – Пример хранения данных в базе данных Redis

В перспективе, наличие слоя кэширования позволит значительно снизить нагрузку на базу данных и уменьшить время отклика сервера на запросы пользователей.

3.8. ВЕБ СЕРВЕР

Веб сервер по своей сути является прослойкой между внешним миром, и всем программным комплексом, развернутом на физическом или виртуальном сервере – в данном случае веб сервером является Nginx. Именно с помощью Nginx’а будет обеспечиваться шифрование трафика и работа по протоколу HTTP/2. По сути весь

процесс настройки веб сервера сводится к написанию его конфигурационного файла, поэтому рассмотрим ключевые его моменты:

Сжатие – для ускорения работы сервера при данные отправляются в сжатом виде, это обеспечивается добавлением в конфигурационный файл следующих строк:

```
gzip on;  
gzip_comp_level 3;  
gzip_types text/css;  
gzip_types text/javascript;
```

Параметр “gzip_comp_level” позволяет указать нужную степень сжатия данных при отправке. Но с этим параметром следует быть аккуратнее, т.к. после 3 уровня сжатия эффект от сжатия усиливается довольно слабо, но время, затрачиваемое на само сжатие, растет в разы.

Следующим шагом следует описать доступные веб серверу маршруты для обращения к представляемому API, потому как все остальное будет заблокирована через файрвол, установленный на сервере:

```
limit_req_zone $request_uri zone=REQLIM:10m rate=60r/m;  
server {  
    listen 80;  
    server_name domen.ru;  
    location ~ /api/file/* {  
        limit_req zone=REQLIM burst=5;  
        proxy_pass 'http://localhost:8081/public/file';  
    }  
}
```

Данные строки позволят перенаправить все входящие по адресу ["http://domen.ru/api/file/*"](http://domen.ru/api/file/*) соединения на внешнее api файлового сервиса. Так же стоит обратить внимание на то, что тут используется директива: “limit_req”, данный параметр позволяет устанавливать ограничения на количество одновременных соединений от одного пользователя. В данном случае этот параметр использует настройки ранее определенной зоны с названием “REQLIM”, и размером очереди равным 5, после заполнения которой все последующие запросы от данного клиента будут отклоняться. В самой же зоне

“REQLIM” стоит ограничение на 60 запросов в минуту, т.е. не более 1 запроса в секунду.

Далее перейдем к описанию части конфигурации, отвечающей за доступ к основному API:

```
server {
    listen 443 ssl http2;
    server_name domen.ru;
    ssl_certificate /etc/letsencrypt/live/ domen.ru/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/ domen.ru/privkey.pem;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
    ssl_prefer_server_ciphers on;
    ssl_ciphers
ECDH+AESGCM:ECDH+AES256:ECDH+AES128:DH+3DES:!ADH:!AECDH:!MD5;
    ssl_dhparam /etc/nginx/ssl/dhparam.pem;
    add_header Strict-Transport-Security "max-age=31536000" always;
    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-XSS-Protection "1; mode=block";
    ssl_session_cache shared:SSL:40m;
    ssl_session_timeout 10m;
    ssl_session_tickets on;
    location ~ /api/* {
        limit_req zone=REQLIM burst=5;
        proxy_pass 'http://localhost:8080/api';
    }
}
```

В данной конфигурации сразу же можно найти существенные отличия от предыдущего образца:

```
listen 443 ssl http2;
```

Директива “listen” говорит о том, с какого порта следует принимать входящие соединения и с какими параметрами. В данном случае выбран порт 443, и это не случайно, потому как 443 это стандартный порт для HTTPS, т.е. защищенного соединения, о чем как раз и говорит второй аргумент – “ssl”. Ну и так-как из-за того, что работа так или иначе будет производится с зашифрованными данными, передаваемым в бинарном виде - ничего не мешает делать это по протоколу HTTP/2, так-как это в значительной степени может уменьшить время отклика сервера. Далее в конфигурационном файле

описываются параметры HTTPS соединения и параметры перенаправления запроса непосредственно на главное API.

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. ОБЩИЕ СВЕДЕНИЯ

В ходе проделанной работы был реализован программный комплекс, который получил название: «Huplex». Для корректного функционирования данного программного обеспечения необходим набор следующего программного обеспечения:

- 1) NodeJS – версии 8 или новее;
- 2) PostgreSQL – версии 9.6 или новее;
- 3) Nginx – версии 1.15 или новее;
- 4) Redis – версии 5.0.4 или новее;
- 5) RabbitMQ – версии 3.7.14 или новее.

Данное программное обеспечение написано на языке программирования JavaScript, с использованием языка структурированных запросов – SQL, для написания запросов к базе данных PostgreSQL.

4.2. ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ

Данное программное обеспечение создано в первую очередь для того, чтобы обеспечить решение следующих задач:

- 1) Сохранение данных пользователя;
- 2) Осуществление выборки из сохраненных данных;
- 3) Обеспечение безопасности канала связи между клиентом и сервером;
- 4) Безопасное хранение файлов;
- 5) Безопасная отдача файлов.

Главной целью данного ПО в первую очередь является обеспечение выполнения минимального необходимого набора функций нужного для корректного функционирования клиента. Это требование сформировано, потому

что, прежде чем начать вкладывать серьезные ресурсы в развитие данного проекта необходимо оценить его потенциал и перспективы развития в будущем.

4.3. ОПИСАНИЕ ЛОГИЧЕСКОЙ СТРУКТУРЫ

Общий алгоритм работы программного комплекса изображен на четырех блок-схемах, приведенных в приложении А. Разбиение выполнено исходя из основных типов сценариев, которые могут быть выполнены данным программным комплексом.

Само же ПО состоит из следующих составных частей, исходный код которых приведен в приложении Б (более подробно структура модулей рассматривается в п. 4.6.):

- 1) HTTP сервер (далее: сервер);
- 2) Сервис, отвечающий за работу с файлами (далее: filesjs);
- 3) Сервис, отвечающий за рассылку электронных писем (далее: mailer).

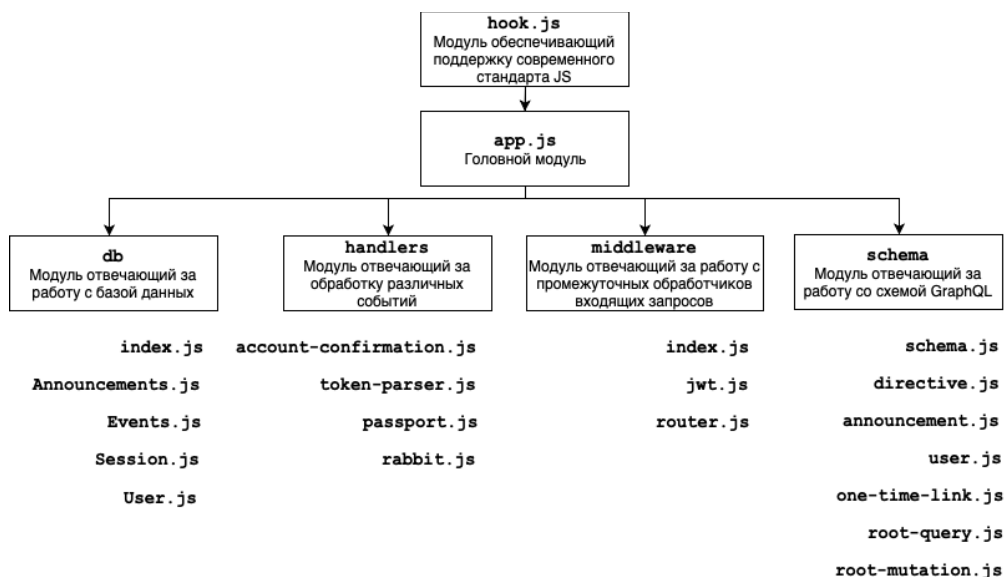


Рисунок 15 – Логическая структура HTTP сервера

Сервер структурно состоит из четырех модулей (рисунок 15):

- 1) db – модуль в котором описаны методы взаимодействия с моделями данных, хранящимися в базе данных:

- a. `index.js` – модуль, отвечающий за формирование и отправку запросов в базу данных;
 - b. `Announcements.js` – модуль, описывающий методы взаимодействия с моделью данных объявления;
 - c. `Events.js` – модуль, описывающий методы взаимодействия с моделью данных события;
 - d. `Session.js` – модуль, описывающий методы взаимодействия с моделью данных сессии;
 - e. `User.js` – модуль, описывающий методы взаимодействия с моделью данных пользователя;
- 2) `handlers` – модуль, содержащий описание различных обработчиков событий:
- a. `account-confirmation.js` – модуль описывающий взаимодействие с сервисом отправки электронных писем через RabbitMQ;
 - b. `token-parser.js` – модуль отвечающий за дешифровку токена;
 - c. `passport.js` – модуль отвечающий за аутентификацию пользователя, через OAuth сервера компании “Google”;
 - d. `rabbit.js` – модуль, описывающий методы для взаимодействия с шиной данных RabbitMQ;
- 3) `middleware` – модуль, отвечающий за подключение промежуточных обработчиков для входящих запросов:
- a. `index.js` – модуль, описывающий подключение всех базовых промежуточных обработчиков;
 - b. `jwt.js` – модуль, описывающий промежуточный обработчик, отвечающий за определение уровня доступа пользователя, если он предоставил корректный токен;
 - c. `router.js` – модуль, отвечающий за подключения схемы GraphQL к HTTP серверу;

- 4) **schema** – модуль, описывающий схему моделей данных на языке запросов GraphQL:
- a. **schema.js** – модуль, формирующий итоговую схему данных;
 - b. **directive.js** – модуль, описывающий директивы, благодаря которым происходит проверка соответствия уровня доступа пользователя и требуемого уровня доступа для доступа к запрашиваемым данным;
 - c. **announcement.js** – модуль, описывающий модель данных объявления на языке GraphQL;
 - d. **user.js** – модуль, описывающий модель данных пользователя на языке GraphQL;
 - e. **one-time-link.js** – модуль, описывающий модель данных одноразовой ссылки на языке GraphQL;
 - f. **root-query.js** – модуль, описывающий модель данных корневого запроса на получение данных, на языке GraphQL;
 - g. **root-mutation.js** – модуль, описывающий модель данных корневого запроса на изменение данных, на языке GraphQL.

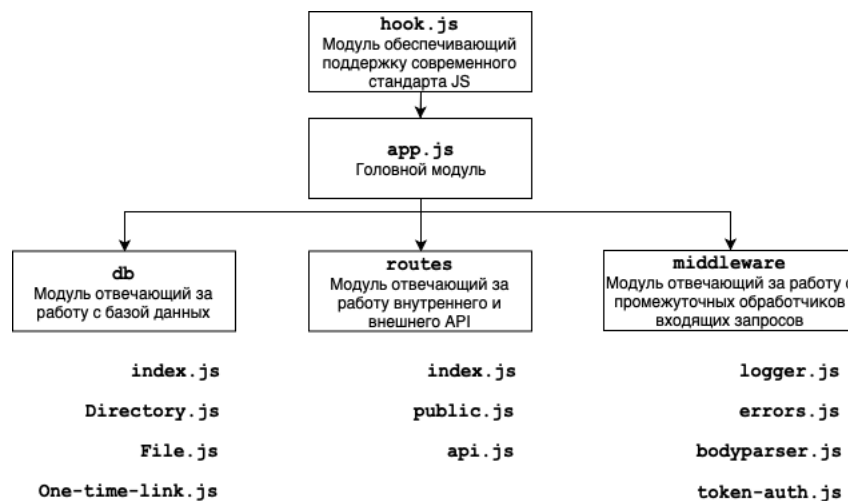


Рисунок 16 – Логическая структура файлового сервиса

Filesjs, структурно состоит из 3 модулей (рисунок 16):

- 1) **db** – модуль в котором описаны методы взаимодействия с моделями данных, хранящимися в базе данных:

- a. `index.js` – модуль, отвечающий за формирование и отправку запросов в базу данных;
 - b. `Directory.js` – модуль, описывающий методы взаимодействия с моделью данных директории;
 - c. `File.js` – модуль, описывающий методы взаимодействия с моделью данных файла;
 - d. `One-time-link.js` – модуль, описывающий методы взаимодействия с моделью данных одноразовой ссылки;
- 2) `routes` – модуль, в котором определены методы взаимодействия с сервисом:
- a. `index.js` – модуль, отвечающий за подключение методов в цепочку промежуточных обработчиков всех входящих запросов;
 - b. `public.js` – модуль, описывающий методы скачивания и загрузки файлов на сервер, через одноразовые ссылки;
 - c. `api.js` – модуль, описывающий внутренние методы взаимодействия с сервисом, доступные только для обращения только с других сервисов;
- 3) `middleware` – модуль, отвечающий за подключение промежуточных обработчиков для входящих запросов:
- a. `logger.js` – модуль, отвечающий за запись всех входящих запросов в файл;
 - b. `errors.js` – модуль, отвечающий за обработку ошибок, возможных входе выполнения запросов;
 - c. `bodyparser.js` – модуль, отвечающий за загрузку файлов на сервер;
 - d. `token-auth.js` – модуль, отвечающий за проверку ключа доступа, без которого все входящие запросы к внутреннему API будут отклоняться.

Mailer, структурно состоит из 2 модулей (рисунок 17):

- 1) `handlers` – модуль, содержащий описание различных обработчиков событий:

- a. `rabbit.js` – модуль, описывающий методы для взаимодействия с шиной данных RabbitMQ;
- 2) `mailer` – модуль, отвечающий за отправку электронных писем:
 - a. `mailer.js` – модуль, описывающий метод отправки электронного письма.

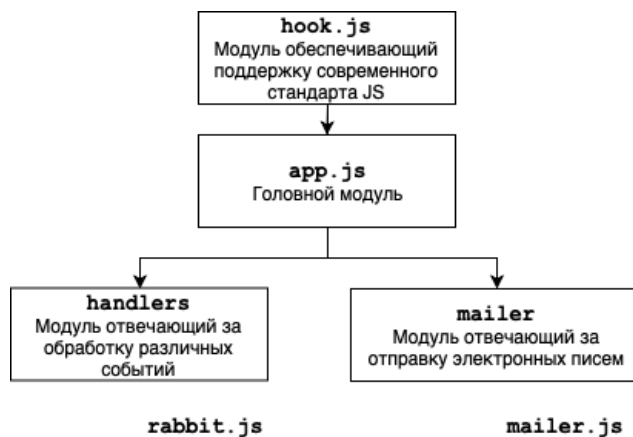


Рисунок 17 – Логическая схема сервиса по рассылке электронных писем

Между собой части ПО связаны следующим образом (рисунок 18):

- 1) Взаимодействие между сервером и `filesjs`;
- 2) Взаимодействие между сервером и `mailer`, осуществляется посредством брокера сообщений RabbitMQ.

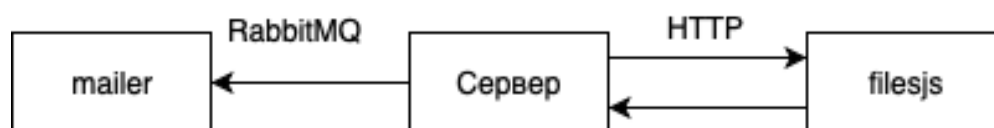


Рисунок 18 – Схема связей частей ПО

4.4. ИСПОЛЬЗУЕМЫЕ ТЕХНИЧЕСКИЕ СРЕДСТВА

Для корректного использования данного программного комплекса, помимо приведенного в п. 4.1. перечня программного обеспечения, необходимо, что бы сам компьютер соответствовал следующим системным требованиям:

- 1) Операционная система: Windows 7/8/10 /Linux 4.0 /OSX 10.2, или новее;

- 2) Процессор с тактовой частотой не ниже 1 ГГц;
- 3) Оперативная память объемом не ниже 1 Гб.

4.5. ВЫЗОВ И ЗАГРУЗКА

Для запуска данного программного обеспечения необходимо:

- 1) Создать новую базу данных в PostgreSQL, с использованием схемы базы данных, хранящейся в каталоге “support”, расположенном в корневой директории ПО;
- 2) Создать перейти в корневой каталог данного программного обеспечения;
- 3) Создать файл “.env”, и заполнить его в соответствии с указаниями в файле “dotEnvTemplate”;
- 4) Открыть командную строку в директории с файлом “.env” и выполнить команду: “npm start”.

4.6. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

Так-как, реализованный программный комплекс представляет из себя веб сервер, то входными данными в данном случае будут являться HTTP запросы, приведенные в таблице 1, выходные данные описаны в таблице 2.

Таблица 1 – Входные данные

HTTP запрос	Смысловое содержание	Параметры
1	2	3
GET /verify/<eventId>	Подтверждение регистрации аккаунта пользователя	- eventId – уникальный идентификатор, события, связывающий данную ссылку с аккаунтом пользователя;
POST /api	Запрос, который должен быть обработан с помощью обработчика GraphQL	- token – если в заголовках запроса НЕ содержится токен, то клиенту будут доступны только запросы на регистрацию и аутентификацию; - body – строка, представляющая собой тело запроса, написанное на языке запросов GraphQL;

1	2	3
POST /public/file/<hash>	Запрос на загрузку файла на сервер	- hash – уникальный идентификатор одноразовой ссылки;
GET /public/file/<hash>	Запрос на скачивание файла с сервера	- hash – уникальный идентификатор одноразовой ссылки;

Таблица 2 – Выходные данные

HTTP запрос	Смысловое содержание	Ответ
1	2	3
GET /verify/<eventId>	Подтверждение регистрации аккаунта пользователя	- statusCode – статус код, сигнализирующий об успешности выполнения запроса: 200 – если запрос выполнен успешно, 400 – если что-то пошло не так;
POST /api	Запрос, который должен быть обработан с помощью обработчика GraphQL	- data – результат выполнения запроса, обработанного GraphQL, возвращаемый в формате JSON;
POST /public/file/<hash>	Запрос на загрузку файла на сервер	- statusCode – статус код, сигнализирующий об успешности выполнения запроса: 200 – если запрос выполнен успешно, 404 – если ссылка не существует и 500 – если произошла внутренняя ошибка сервера;
GET /public/file/<hash>	Запрос на скачивание файла с сервера	- file – файл, которому соответствовала одноразовая ссылка; - statusCode – статус код, сигнализирующий об успешности выполнения запроса: 200 – если запрос выполнен успешно, 404 – если ссылка не существует и 500 – если произошла внутренняя ошибка сервера;

5. ПОЛУЧЕННЫЙ РЕЗУЛЬТАТ И ДАЛЬНЕЙШЕЕ РАЗВИТИЕ

Результатом проделанной стало очень гибкое и достаточно производительно API, которое легко расширять и модернизировать под нужды потенциальных клиентов. В случае если проект окажется достаточно успешным то в дальнейшем он будет переработан в агрегатор объявлений, который позволит пользователям “в один клик” выложить свое объявления на все популярные ресурсы, и также легко их редактировать, отличительной особенностью данного агрегатора так же останется возможность создания планов помещений, которая продолжит развиваться с использование возможностей искусственного интеллекта.

СПИСОК ЛИТЕРАТУРЫ

1. Шаблоны проектирования Node.js / пер. с англ. А.Н. Киселева. — М.: ДМК Пресс, 2017. — 396с.: ил.
2. JavaScript и jQuery. Интерактивная веб-разработка / Джон Дакетт ; [пер. с англ. М.А. Райтмана]. — Москва : Издательство «Э», 2017. — 640 с.: ил.
3. Изучаем Java / К. Сиерра, Б. Бейтс. — М.: Эксмо, 2018. — 720с.: ил.
4. Node.js в действии. 2-е издание. / пер. с англ. Е. Матвеев. — СПб.: Питер 2018. — 332с.: ил.
5. React и Redux: функциональная веб-разработка. — СПб.: Питер, 2018. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).
6. Язык программирования Go.: Пер. с англ. — М.: ООО «И.Д. Вильямс», 2016. — 432 с.: ил. — Парал. тит. англ.
7. node.js [Электронный ресурс]. — Режим доступа: <https://nodejs.org> — Описание технологии. — (Дата обращения: 27.04.2019).
8. PostgreSQL [Электронный ресурс]. — Режим доступа: <https://postgresql.org> — Описание технологии. — (Дата обращения: 27.04.2019).
9. MySQL [Электронный ресурс]. — Режим доступа <https://www.mysql.com> — Описание технологии. — (Дата обращения: 27.04.2019).
- 10.nginx [Электронный ресурс]. — Режим доступа: <https://nginx.org> — Описание технологии. — (Дата обращения: 27.04.2019).
- 11.similartech [Электронный ресурс]. — Режим доступа: <https://www.similartech.com> — Описание технологий. — (Дата обращения: 27.04.2019).
- 12.benchmarksgame [Электронный ресурс]. — Режим доступа: <https://benchmarksgame-team.pages.debian.net/benchmarksgame> — Сравнительные тесты производительности. — (Дата обращения: 27.04.2019).
- 13.tiobe [Электронный ресурс]. — Режим доступа: <https://www.tiobe.com/> — Описание технологий. — (Дата обращения: 27.04.2019).
- 14.apache [Электронный ресурс]. — Режим доступа: <https://apache.org> — Описание технологии. — (Дата обращения: 27.04.2019).
- 15.GraphQL [Электронный ресурс]. — Режим доступа: <https://graphql.org> — Описание технологии. — (Дата обращения: 27.04.2019).
- 16.MongoDB [Электронный ресурс]. — Режим доступа <https://www.mongodb.com> — Описание технологии. — (Дата обращения: 27.04.2019).
- 17.python [Электронный ресурс]. — Режим доступа: <https://python.org> — Описание технологии. — (Дата обращения: 27.04.2019).

- 18.memcached [Электронный ресурс]. — Режим доступа: <https://memcached.org> — Описание технологии. — (Дата обращения: 27.04.2019).
- 19.redis [Электронный ресурс]. — Режим доступа: <https://redis.io> — Описание технологии. — (Дата обращения: 27.04.2019).
- 20.RabbitMQ [Электронный ресурс]. — Режим доступа: <https://www.rabbitmq.com> — Описание технологии. — (Дата обращения: 27.04.2019).
- 21.Java [Электронный ресурс]. — Режим доступа <https://www.oracle.com> — Описание технологии. — (Дата обращения: 27.04.2019).
- 22.PHP [Электронный ресурс]. — Режим доступа <https://www.php.net> — Описание технологии. — (Дата обращения: 27.04.2019).
- 23.Ruby [Электронный ресурс]. — Режим доступа <https://www.ruby-lang.org> — Описание технологии. — (Дата обращения: 27.04.2019).
- 24.PostgreSQL vs MySQL [Электронный ресурс]. — 2015. — Режим доступа: <https://habr.com> . — (Дата обращения: 27.04.2019).
- 25.Что такое TLS [Электронный ресурс]. — 2015. — Режим доступа: <https://habr.com> . — (Дата обращения: 27.04.2019).
- 26.Обзор способов и протоколов аутентификации в веб-приложениях [Электронный ресурс]. — 2015. — Режим доступа: <https://habr.com> . — (Дата обращения: 27.04.2019).
- 27.Использование memcached и redis в высоконагруженных проектах [Электронный ресурс]. — 2016. — Режим доступа: <https://habr.com> . — (Дата обращения: 27.04.2019).
- 28.Brad Peabody Server-side I/O Performance: Node vs. PHP vs. Java vs. Go [Электронный ресурс]. — 2017. — Режим доступа: <https://www.toptal.com> . — (Дата обращения: 27.04.2019).

ПРИЛОЖЕНИЕ А. БЛОК-СХЕМЫ ОСНОВНЫХ АЛГОРИТМОВ РЕАЛИЗОВАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

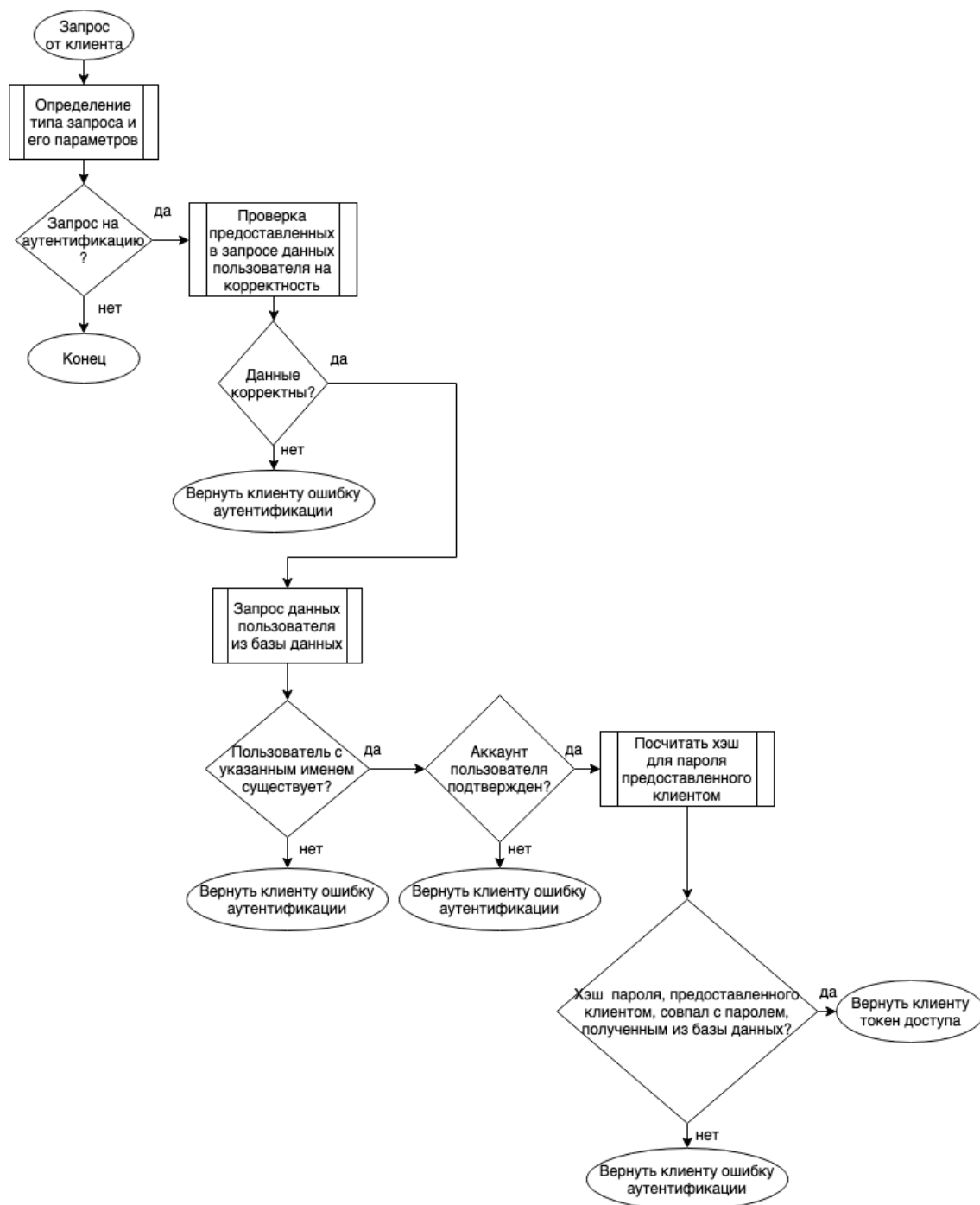


Рисунок А.1 – Алгоритм обработки запроса аутентификации пользователя в системе.

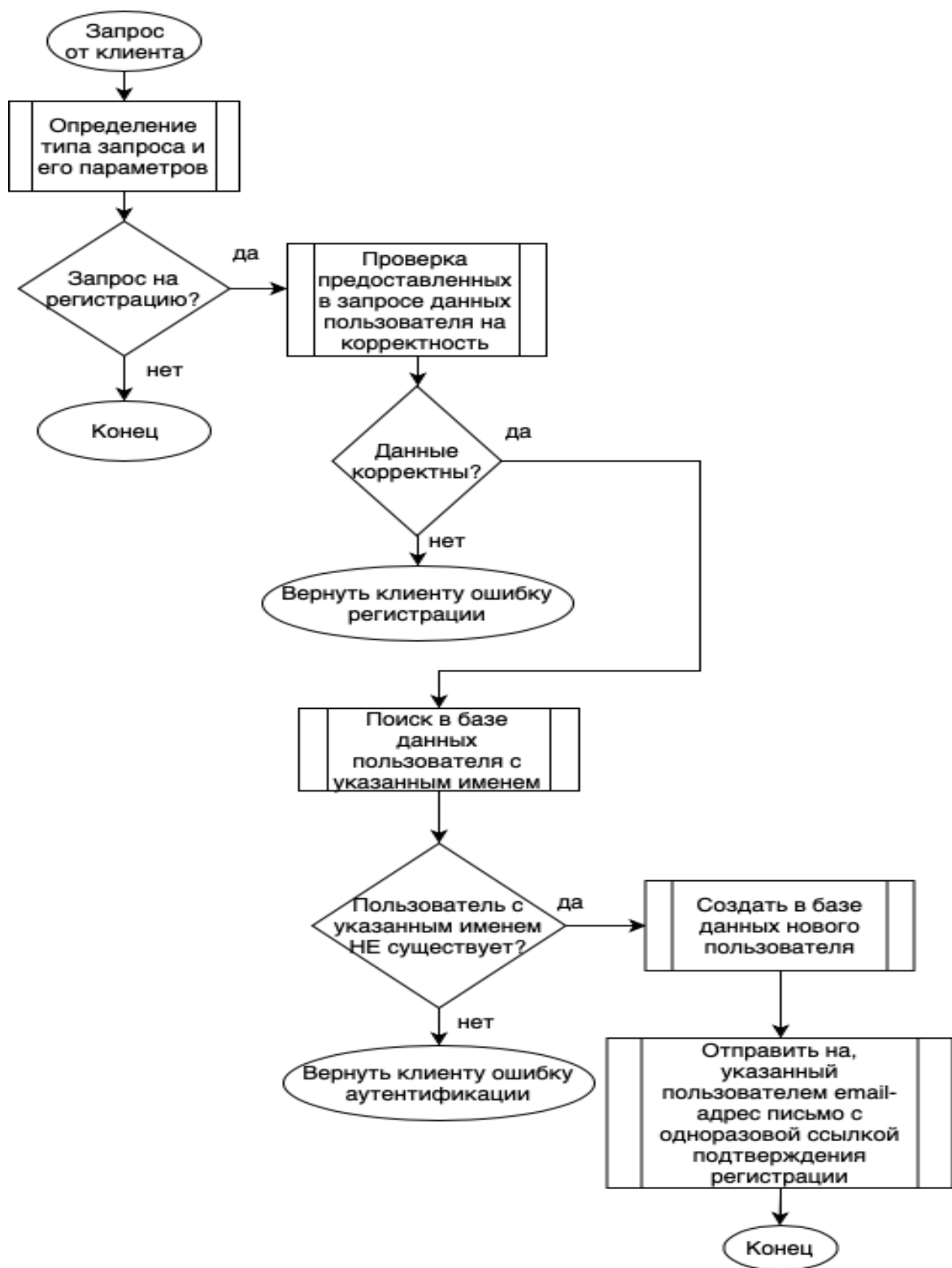


Рисунок А.2 – Алгоритм обработки запроса регистрации пользователя в системе

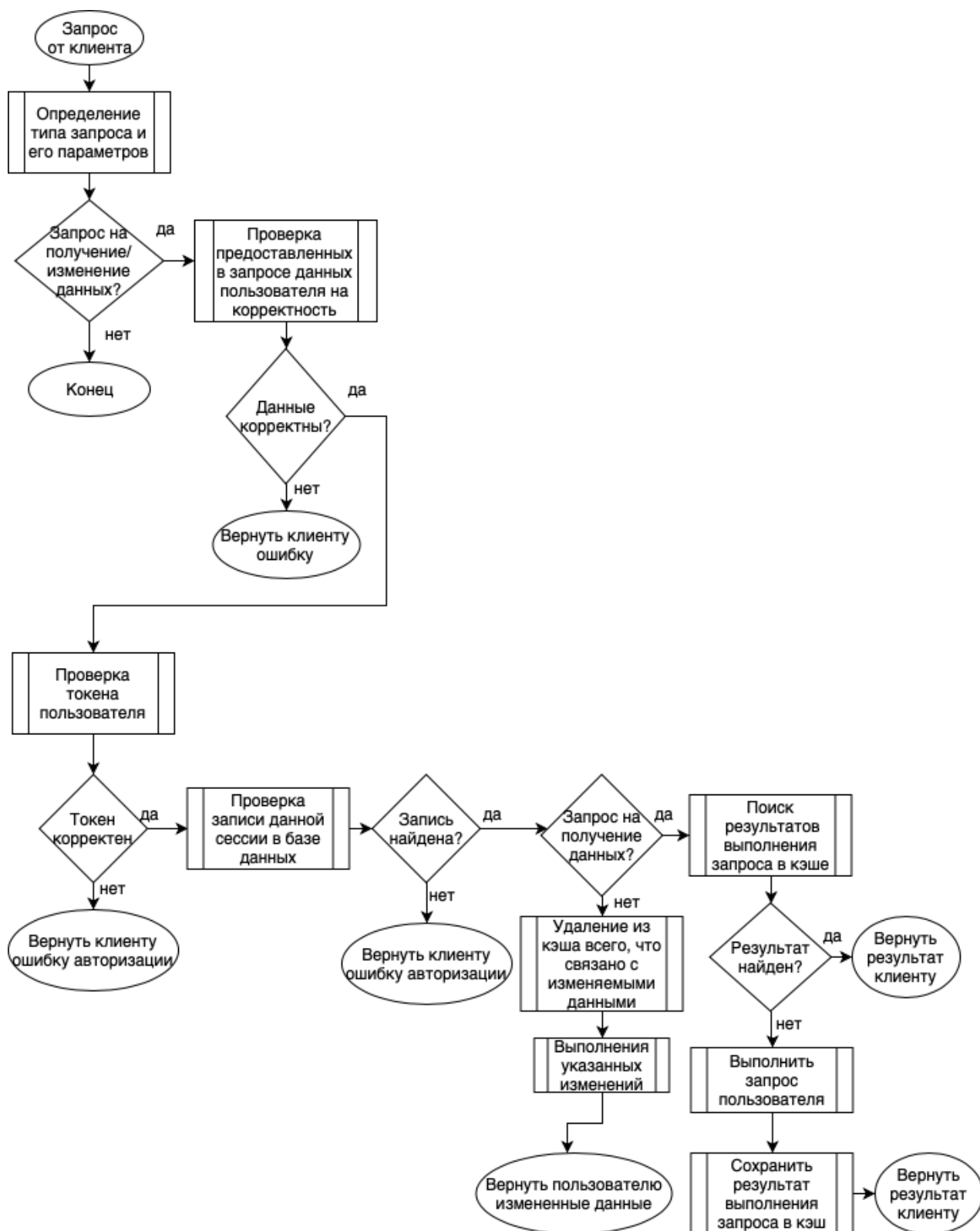


Рисунок А.3 – Алгоритма обработки запроса на получение или изменение данных пользователя

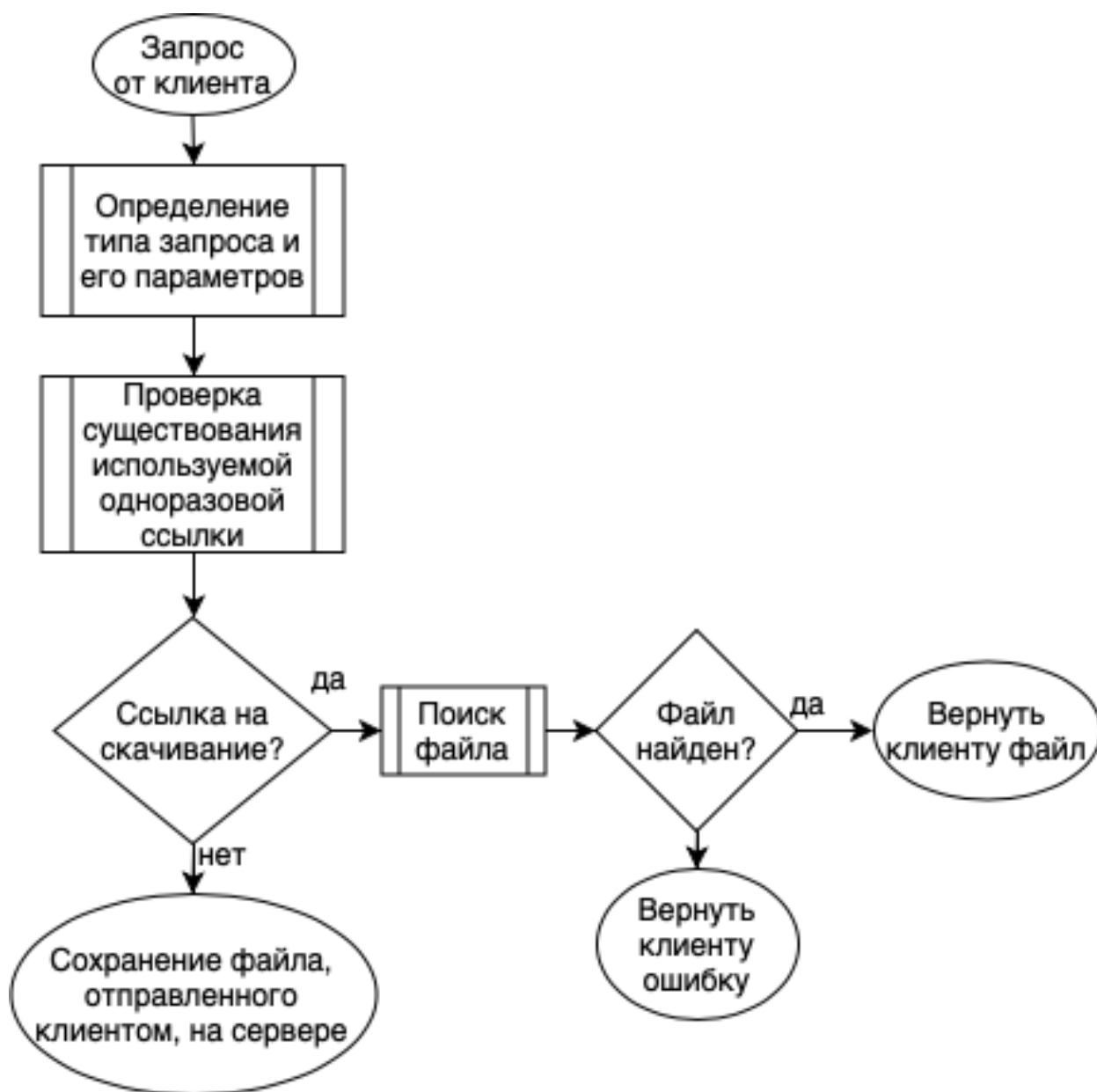


Рисунок А.4 – Алгоритм обработки запросов на скачивание файлов или загрузки файлов на сервер

ПРИЛОЖЕНИЕ Б. ТЕКСТ РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В приложение представлены основные модули реализованного программного обеспечения.

APP.js

```
import http from 'http'
import config from 'config'
import * as log4js from 'log4js'
import app from './src/server'
import pool from './src/db'

const port = config.server.port || 8080
const server = http.createServer(app)
const log = log4js.getLogger('app')
log.level = config.logger.level

server.listen(port, () => {
  log.info(`Server listen on port: ${port}`)
})

server.pool = pool

export default server
```

SERVER.js

```
import express from 'express'
import GraphQL from 'express-graphql'
import applyMiddleware from './middlewares'
import GraphQLSchema from './schema/schema'

const app = express()

applyMiddleware(app)

// GraphQL
app.use('/api', GraphQL((req, res) => ({
  schema: GraphQLSchema,
  graphiql: process.env.NODE_ENV === 'development',
  context: { req, res },
})))

export default app
```


SCHEMA.js

```
import { GraphQLSchema } from 'graphql'
import { applySchemaCustomDirectives } from 'graphql-custom-directive'
import RootQuery from './types/query/root-query'
import RootMutation from './types/mutations/root-mutation'
import directives from './directives'

const schema = new GraphQLSchema({
  directives,
  query: RootQuery,
  mutation: RootMutation,
})

applySchemaCustomDirectives(schema)

export default schema
```

ROOTQUERY.js

```
import { GraphQLObjectType } from 'graphql'
import UserType from '../user'
import AnnouncementsType from '../announcement'

const RootQuery = new GraphQLObjectType({
  name: 'RootQuery',
  fields: {
    user: UserType,
    announcements: AnnouncementsType,
  },
  directives: {
    hasScope: {
      scope: ['read'],
    },
  },
})

export default RootQuery
```

USERTYPE.js

```
import { GraphQLObjectType, GraphQLString } from 'graphql'

const UserType = new GraphQLObjectType({
  name: 'User',
  fields: {
    user_id: {
      type: GraphQLString,
    },
    login: {
      type: GraphQLString,
    },
  },
})
```

```
    },
  })
}
```

```
export default UserType
```

USERRESOLVE.js

```
import * as log4js from 'log4js'
import config from 'config'
import User from '../../../../db/models/User'

const log = log4js.getLogger('schema.user-type.resolver>')
log.level = config.logger.level

export default async (parentValue, { login }) => {
  const user = await User.getUserByLogin(login)
  return user
}

import * as log4js from 'log4js'
import config from 'config'
import User from '../../../../db/models/User'

const log = log4js.getLogger('schema.user-type.resolver>')
log.level = config.logger.level

export default async (parentValue, { login }) => {
  const user = await User.getUserByLogin(login)
  return user
}
```

ROOTMUTATION.js

```
import { GraphQLObjectType } from 'graphql'
import NewUserType from '../user'
import LoginType from '../common/login'
import LogoutType from '../common/logout'
import LoginGoogleType from '../common/login-google'
import AnnouncementType from '../announcement'
import OneTimeLinkType from '../one-time-link'

const RootMutation = new GraphQLObjectType({
  name: 'RootMutation',
  fields: {
    registration: NewUserType,
    login: LoginType,
    logout: LogoutType,
    loginGoogle: LoginGoogleType,
    announcement: AnnouncementType,
    oneTimeLink: OneTimeLinkType,
  },
  directives: {
```

```

      hasScope: {
        scope: ['write'],
      },
    },
  },
})

export default RootMutation

```

USERREGISTRATIONRESOLVER.js

```

import * as log4js from 'log4js'
import config from 'config'
import { GraphQLError } from 'graphql'
import Joi from 'joi'
import User from '../../db/models/User'
import rabbit from '../../handlers/rabbit'
import { services } from '../../handlers/rabbit/config'

const log = log4js.getLogger('schema.mutation.newuser-type.resolver>')
log.level = config.logger.level

const schema = Joi.object().keys({
  password: Joi.string().min(4).required(),
  login: Joi.string().email({ minDomainSegments: 2 }).required(),
})

export default async (parentValue, { login, password }) => {
  const schemaCheck = Joi.validate({
    login,
    password,
  }, schema)
  if (schemaCheck.error) {
    log.error('> Error: invalid credentials!')
    return new GraphQLError('> Invalid credentials!')
  }
  try {
    const userId = await User.registerNewUser(login, password)
    const eventId = await rabbit.sendMsg(services.mailer, {
      mail: login,
    })
    await User.makeNewUserAccountConfirmationTicket(userId.user_id, eventId.id)
    return login
  } catch (e) {
    log.error('> Error, while trying to create a new user!\n', e)
    return new GraphQLError('User with this email already exist!')
  }
}

```

ISAUTHENTICATEDDIRECTIVE.js

```

import {

```

```

    DirectiveLocation,
  } from 'graphql/language/directiveLocation'

  import {
    GraphQLCustomDirective,
  } from 'graphql-custom-directive'

  import resolve from './resolver'

  export default new GraphQLCustomDirective({
    name: 'isAuthenticated',
    description:
      'check user token',
    locations: [
      DirectiveLocation.FIELD,
      DirectiveLocation.QUERY,
    ],
    resolve,
  })

```

JWTMIDDLEWARE.js

```

import * as log4js from 'log4js'
import jwt from 'jsonwebtoken'
import config from 'config'
import Session from '../db/models/Session'

const log = log4js.getLogger('middleware.jwt>')
log.level = config.logger.level

export default async (req, res, next) => {
  const { token } = req
  let session
  let payload
  if (token) {
    try {
      payload = jwt.verify(
        token,
        process.env.JWT_SECRET,
      )
    } catch (e) {
      log.error('> Error, while trying to verify token!')
      return res.boom.unauthorized('Error, while trying to verify token!')
    }

    try {
      session = await Session.getSessionByUserToken(token)
    } catch (e) {
      log.error('> Error, while trying to get session from DB!\n', e)
      return res.boom.badImplementation('Internal server error!')
    }

    if (!Array.isArray(session)) {

```

```

    const checkExpTime = new Date(session.ex)
    const expTime = new Date(payload.exp * 1000)
    const currDate = new Date()
    if (currDate < checkExpTime && currDate < expTime) {
      req.ctx = {
        id: payload.user.id,
        scopes: payload.scopes,
      }
    }
  }
}
next()
}

```

MIDDLEWAREINIT.js

```

import path from 'path'
import helmet from 'helmet'
import cors from 'cors'
import logger from 'morgan'
import config from 'config'
import bodyParser from 'body-parser'
import glob from 'glob'
import lusca from 'lusca'
import * as log4js from 'log4js'
import expressBoom from 'express-boom'
import tokenParser from '../handlers/common/token-parser'

const log = log4js.getLogger('middlewares>')
log.level = config.logger.level

const applyMiddleware = (app, ...middleware) => {
  middleware.forEach(m => app.use(m))
}

export default app => {
  const middleware = glob
    .sync(path.join(__dirname, '/*.js'))
    .filter(m => m.indexOf('index.js') === -1)
    .map(m => m.replace(path.extname(m), ''))

  log.info(middleware.map(m => `middleware injected ${path.basename(m)}`))
  if (process.env.NODE_ENV !== 'test') {
    applyMiddleware(
      app,
      expressBoom(),
      lusca(config.lusca),
      helmet(),
      cors(),
      logger(config.restLogLevel),
    )
  }
}

```

```

    bodyParser.json(),
    bodyParser.urlencoded({ extended: false }),
    tokenParser,
    /* eslint-disable-next-line */
    ...middleware.map(m => require(m).default),
  )
} else {
  applyMiddleware(
    app,
    expressBoom(),
    lusca(config.lusca),
    helmet(),
    cors(),
    logger(config.restLogLevel),
    bodyParser.json(),
    bodyParser.urlencoded({ extended: false }),
    tokenParser,
    /* eslint-disable-next-line */
    ...middleware.map(m => require(m).default),
  )
}
}

```

RABBIT.js

```

import * as log4js from 'log4js'
import config from 'config'
import amqp from 'amqplib/callback_api'
import Events from '../db/models/Events'

const log = log4js.getLogger('handlers.rabbit>')
log.level = config.logger.level

class Rabbit {
  constructor() {
    this.channels = {}
    this.queues = {}
  }

  connect(connectionURL = process.env.RABBITMQ_URL) {
    return new Promise((res, rej) => {
      amqp.connect(connectionURL, (err0, connection) => {
        if (err0) {
          return rej(err0)
        }
        this._connection = connection
        return res(connection)
      })
    })
  }

  _makeNewChannel(channelName) {

```

```

// eslint-disable-next-line consistent-return
return new Promise(async (res, rej) => {
  let { _connection } = this
  if (_connection === undefined) {
    try {
      _connection = await this.connect()
    } catch (e) {
      log.error('> Error, while trying to create a connection with rabbit mq!\n', e)
      return rej(e)
    }
  }

  _connection.createChannel((err1, channel) => {
    if (err1) {
      return rej(err1)
    }

    // eslint-disable-next-line security/detect-object-injection
    this.channels[channelName] = channel
    return res(channel)
  })
})
}

makeNewQueue(channelName) {
  return new Promise(async (res, rej) => {
    let channel = this.channels[channelName]
    if (!channel) {
      try {
        channel = await this._makeNewChannel(channelName)
      } catch (e) {
        log.error('> Error, while trying to make a new channel!\n', e)
        rej(e)
      }
    }

    const queueIn = `${channelName}_in`
    const queueOut = `${channelName}_out`
    channel.assertQueue(queueIn, {
      durable: true,
    })
    channel.assertQueue(queueOut, {
      durable: true,
    })
    channel.prefetch(1)

    channel.consume(queueOut, msg => {
      const id = msg.properties.correlationId
      Events.deleteEvent(id)
      .then(() => {
        channel.ack(msg)
      })
    })
  })
}

```

```

        .catch(reject => log.error('> Error, while trying to delete event from DB!\n',
reject))
    })

    const send = async message => {
        let msg
        try {
            msg = JSON.stringify(message)
        } catch (e) {
            log.error('> Error, while trying to stringify message!\n', e)
        }
        const id = await Events.makeNewEvent(channelName, msg)
        channel.sendToQueue(queueIn, Buffer.from(msg), {
            persistent: true,
            correlationId: id.id,
        })
        return id
    }
    res(send)
    this.queues[channelName] = send
})
}

sendMsg(queueName, msg) {
    return new Promise(async (res, rej) => {
        let queue = this.queues[queueName]
        if (!queue) {
            try {
                queue = await this.makeNewQueue(queueName)
            } catch (e) {
                log.error('> Error, while trying to make a new channel!\n', e)
                rej(e)
            }
        }
        res(await queue(msg))
    })
}

const rabbit = new Rabbit()

export default rabbit

```

GOOGLEOAUTH.js

```

import passport from 'passport'
import { Strategy as GoogleTokenStrategy } from 'passport-google-token'

const GoogleTokenStrategyCallback = (accessToken, refreshToken, profile, done) =>
done(null, {
    accessToken,
    refreshToken,

```



```

    profile,
  })

passport.use(new GoogleTokenStrategy({
  clientID: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
}, GoogleTokenStrategyCallback))

const authenticateGoogle = (req, res) => new Promise((resolve, reject) => {
  passport.authenticate('google-token', { session: false }, (err, data, info) => {
    if (err) reject(err)
    resolve({ data, info })
  })(req, res)
})

export default authenticateGoogle

```

DBQUERY.js

```

/* eslint-disable no-param-reassign,prefer-destructuring */
import { Pool } from 'pg'
import * as log4js from 'log4js'
import config from 'config'
import redis from 'redis'
import util from 'util'
import _ from 'lodash'

const log = log4js.getLogger('db>')
log.level = config.logger.level

let pool = new Pool()
const redisClient = redis.createClient(config.redis.URL)
redisClient.hget = util.promisify(redisClient.hget)

function reqPreparer(reqStrings, ...reqParams) {
  let resultQueryString = reqStrings[0]
  for (let i = 1; i < reqStrings.length; i += 1) {
    resultQueryString += `&${reqStrings[i]}=${reqParams[i]}`
  }

  return {
    text: resultQueryString,
    values: reqParams,
  }
}

function getHashKey(preparedReq) {
  const isHashKeyExists = _.split(preparedReq.text, ';hashKey=')
  let hashKey = null
  let needToResetCachedValue = false
  if (isHashKeyExists.length > 1) {

```

```

    needToResetCachedValue = preparedReq.values[preparedReq.values.length - 1] ===
    'reset'
    const offset = needToResetCachedValue ? 2 : 1
    hashKey = preparedReq.values[preparedReq.values.length - offset]
    preparedReq.text = isHashKeyExists[0]
    preparedReq.values = _.dropRight(preparedReq.values, offset)
  }
  // eslint-disable-next-line no-sequences
  return [preparedReq, hashKey, needToResetCachedValue]
}

async function query(reqStrings, ...reqParams) {
  if (pool === null) {
    pool = new Pool()
  }

  let preparedReq
  let key

  if (Array.isArray(reqStrings)) {
    preparedReq = reqPreparer(reqStrings, ...reqParams)
  } else {
    preparedReq = reqStrings
  }

  const temp = getHashKey(preparedReq)
  preparedReq = temp[0]
  const hashKey = temp[1]
  const needToReset = temp[2]
  if (hashKey && !needToReset) {
    try {
      key = JSON.stringify(preparedReq)
      const cacheValue = await redisClient.hget(hashKey, key)
      if (cacheValue) {
        log.debug('SERVED BY CACHE')
        return JSON.parse(cacheValue)
      }
    } catch (e) {
      log.error('> Error, while trying to cache db request!\n', e)
    }
  } else if (hashKey) {
    redisClient.del(hashKey)
  }

  let result = null
  const client = await pool.connect()

  try {
    await client.query('BEGIN')
    result = await client.query(preparedReq)
    await client.query('COMMIT')
  } catch (e) {

```

```

    await client.query('ROLLBACK')
    log.error('> Error while making a query to DB\n', e)
  }
  client.release()
  log.debug('SERVED BY DB')

  if (hashKey && !needToReset) {
    if (result.rows && result.rows.length === 1) {
      redisClient.hset(hashKey, key, JSON.stringify(result.rows[0]), 'EX',
config.redis.EX)
      return result.rows[0]
    }
    redisClient.hset(hashKey, key, JSON.stringify(result.rows), 'EX', config.redis.EX)
    return result.rows
  }

  if (result.rows && result.rows.length === 1) {
    return result.rows[0]
  }
  return result.rows
}

export default {
  query,
  makeQuery: reqPreparer,
  pool,
}

```

USERMODEL.js

```

import config from 'config'
import crypto from 'crypto'
import * as log4js from 'log4js'
import db from '../index'

const log = log4js.getLogger('model.user>')
log.level = config.logger.level

class User {
  static generateNewHashForPassword(password) {
    if (password !== undefined) {
      if (password.length < 4) {
        throw new Error('The password must contain more than 4 characters')
      }
    }
  }
  const salt = crypto.randomBytes(config.crypto.hash.length).toString('base64')

  if (password) {
    const hashPassword = crypto.pbkdf2Sync(
      password,
      salt,
      12000,

```

```

        config.crypto.hash.length,
        'sha256',
    ).toString('base64')
    return {
        salt,
        hashPassword,
    }
}
return null
}

static async checkPassword(login, password) {
    const result = await db.query`
        SELECT "digest", "salt"
        FROM "user"
        WHERE "login" = ${login}
    `
    if (!password) return false
    if (!result || !result.digest) return false
    return crypto.pbkdf2Sync(
        password,
        result.salt,
        12000,
        config.crypto.hash.length,
        'sha256',
    ).toString('base64') === result.digest
}

static async checkPasswordById(id, password) {
    const result = await db.query`
        SELECT "digest", "salt"
        FROM "user"
        WHERE "user_id" = ${id}
    `
    if (!password) return false
    if (!result || !result.digest) return false
    return crypto.pbkdf2Sync(
        password,
        result.salt,
        12000,
        config.crypto.hash.length,
        'sha256',
    ).toString('base64') === result.digest
}

static async checkLogin(email) {
    const result = await db.query`
        SELECT "login"
        FROM "user"
        WHERE "login" = ${email}
    `
    return result == null || result.email !== email
}

```

```

}

static async registerNewUser(email, password) {
  if (password) {
    const { salt, hashPassword } = this.generateNewHashForPassword(password)
    return db.query`
      INSERT INTO "public"."user" ("user_id", "login", "salt", "digest")
      VALUES (DEFAULT, ${email}, ${salt}, ${hashPassword})
      RETURNING "user_id";
    `
  }
  return db.query`
    INSERT INTO "public"."user" ("user_id", "login", "salt", "digest")
    VALUES (DEFAULT, ${email}, '', '')
    RETURNING "user_id";
  `
}

static async updateUserPassword(id, password) {
  const resPass = this.generateNewHashForPassword(password)
  await db.query`
    UPDATE "public"."user"
    SET "salt" = ${resPass.salt}, "digest" = ${resPass.hashPassword}
    WHERE "user_id" = ${id}
  `
}

static async getUserByLogin(login) {
  const result = await db.query`
    SELECT "login", "user_id", "confirmed"
    FROM "user"
    WHERE "login" = ${login}
  `
  return result
}

static async getUserInfoByLogin(login) {
  const result = await db.query`
    SELECT "login", "user_id"
    FROM "user"
    WHERE "login" = ${login}
  `
  return result
}

static async getUserInfoById(id) {
  const result = await db.query`
    SELECT "login", "user_id"
    FROM "user"
    WHERE "user_id" = ${id}
  `
  return result
}

```

```

}

static confirmAccount(id) {
  return db.query`
    UPDATE "public"."user"
    SET "confirmed" = true
    WHERE "user_id" = ${id}
  `
}

static async getUserById(id) {
  const result = await db.query`
    SELECT "login", "user_id"
    FROM "user"
    WHERE "user_id" = ${id}
  `
  return result
}

static makeNewUserAccountConfirmationTicket(userId, eventId) {
  return db.query`
    INSERT INTO "public"."account_confirmation" ("user_id", "event_id", "timestamp")
    VALUES (${userId}, ${eventId}, DEFAULT)
  `
}

static getUserIdByEventId(eventId) {
  return db.query`
    DELETE FROM public.account_confirmation
    WHERE event_id=${eventId}
    RETURNING user_id;
  `
}

static async loginViaGoogle({ accessToken, refreshToken, profile }) {
  const email = profile.emails[0].value
  let user

  try {
    user = await this.getUserByLogin(email)
  } catch (e) {
    log.error('> Error, while trying to get user by email!\n', e)
    return null
  }

  if (Array.isArray(user)) {
    try {
      user = await this.registerNewUser(email)
    } catch (e) {
      log.error('> Error, while trying to make a new user!\n', e)
      return null
    }
  }
}

```

```

    }
  }

  return user
}
}

export default User

```

MAILER.js

```

require('@babel/register')
require('babel-polyfill')
require('dotenv').config()
const config = require('config')
const log4js = require('log4js')
const mailer = require('./src/mailer')
const rabbit = require('./src/handler/rabbit').default
const { services } = require('./src/handler/rabbit/config')

const log = log4js.getLogger('mailer>')
log.level = 'all'

const auth = {
  user: process.env.DIST_MAIL,
  pass: process.env.DIST_MAIL_PASS,
}

const linkTemplate = config.linkTemplate

async function sendMail(mail, message, auth) {
  try {
    const transport = await mailer.transportInit(auth)
    await mailer.sendMail({
      subject: 'Подтверждение регистрации',
      to: mail,
      message,
    }, transport)
  } catch (e) {
    log.error('> Error, while trying to send mail!\n', e)
  }
}

rabbit.makeNewQueue(services.mailer, (id, content) => {
  return new Promise(async (res, rej) => {
    const { mail } = content
    if (!mail) {
      log.error('> Error, while trying to get mail from new msg')
      return rej()
    }
    if (!id) {

```

```

    log.error('> Error, while trying to get id from new msg')
    return rej()
  }

  try {
    await sendMail(mail,
      'Для подтверждения учетной записи пройдите по следующей ссылке' +
      ` ${linkTemplate}${id}`
      , auth)
    res()
  } catch (e) {
    log.error(`> Error, while trying to send message to ${mail}!\n`, e)
    rej(e)
  }
})
})
})

```

FILESJSPUBLICAPI.js

```

import Router from 'koa-router'
import fs from 'fs-extra'
import Boom from 'boom'
import {
  hashValidation, uploadFile as uploadFileValidation,
} from '../filters/api-joi'

const sendFileByOneTimeLink = async ctx => {
  const { hash } = ctx.params
  const OneTimeLink = ctx.db.model('OneTimeLink')
  const link = await OneTimeLink.checkExistence(hash)
  ctx.body = await link.target.getFileStream()
  link.remove()
}

const uploadFile = async ctx => {
  const { hash } = ctx.params
  const OneTimeLink = ctx.db.model('OneTimeLink')
  const link = await OneTimeLink.checkExistence(hash)
  const file = link.target

  if (file.directory) {
    await ctx.db.model('Directory').checkExistence(file.directory)
  }

  try {
    await file.uploadFile(fs.createReadStream(ctx.request.files.file.path))
    file.set('status', 'uploaded')
    await file.save()
    await link.delete()
  } catch (e) {
  }
}

```



```

    if (e.typeof !== Boom.internal() || e.message !== 'Could not write file on disk') {
      await file.removeFileFromDisk()
    }
    throw e
  }
  ctx.body = JSON.stringify(file)
}

const router = new Router()
router.prefix('/public')
router.get('/file/:hash', hashValidation, sendFileByOneTimeLink)
router.post('/file/:hash', hashValidation, uploadFileValidation, uploadFile)

export default router

```

FILESJSAPI.js

```

import Router from 'koa-router'
import Boom from 'boom'
import {
  generateFileOneTimeLink as oneTimeLinkValidation,
} from '../filters/api-joi'

async function downloadOneTimeLinkHandler(ctx) {
  const { id } = ctx.request.body
  if (id === undefined) {
    throw Boom.badRequest('Id is undefined!')
  }
  const file = await ctx.db.model('File').checkExistence(id)
  const OneTimeLink = ctx.db.model('OneTimeLink')
  const oneTimeLink = new OneTimeLink({
    lifeTime: ctx.request.query.lifeTime || null,
    target: file.get('_id'),
  })
  await oneTimeLink.save()
  ctx.body = JSON.stringify(oneTimeLink)
}

async function uploadOneTimeLinkHandler(ctx) {
  const OneTimeLink = ctx.db.model('OneTimeLink')
  const File = ctx.db.model('File')
  const { name, isZipped, directory } = ctx.request.body
  const file = new File({
    name,
    isZipped,
    directory,
  })
  await file.save()

  const oneTimeLink = new OneTimeLink({
    lifeTime: ctx.request.query.lifeTime,
    action: ctx.request.body.type,
  })

```

```

        target: file._id,
    })
    await oneTimeLink.save()
    ctx.body = JSON.stringify(oneTimeLink)
}

const generateFileOneTimeLink = async ctx => {
    switch (ctx.request.body.type) {
        case 'upload':
            await uploadOneTimeLinkHandler(ctx)
            break
        case 'download':
            await downloadOneTimeLinkHandler(ctx)
            break
        default:
            throw Boom.badRequest('Invalid type of request!')
    }
}

const router = new Router()
router.prefix('/api')
router.post('/file/generate-one-time-link', oneTimeLinkValidation,
generateFileOneTimeLink)

export default router

```