Описание системы, так сказать

Данный проект представляет собой, по сути, просто онлайн блокнот с некоторыми наворотами, типа совместного доступа к запискам, регистрации, хранении своих записок на сервере и всем таким.

Если покажется, что для такой системы устройство слишком сложное, то так и есть, проект делался как учебный.

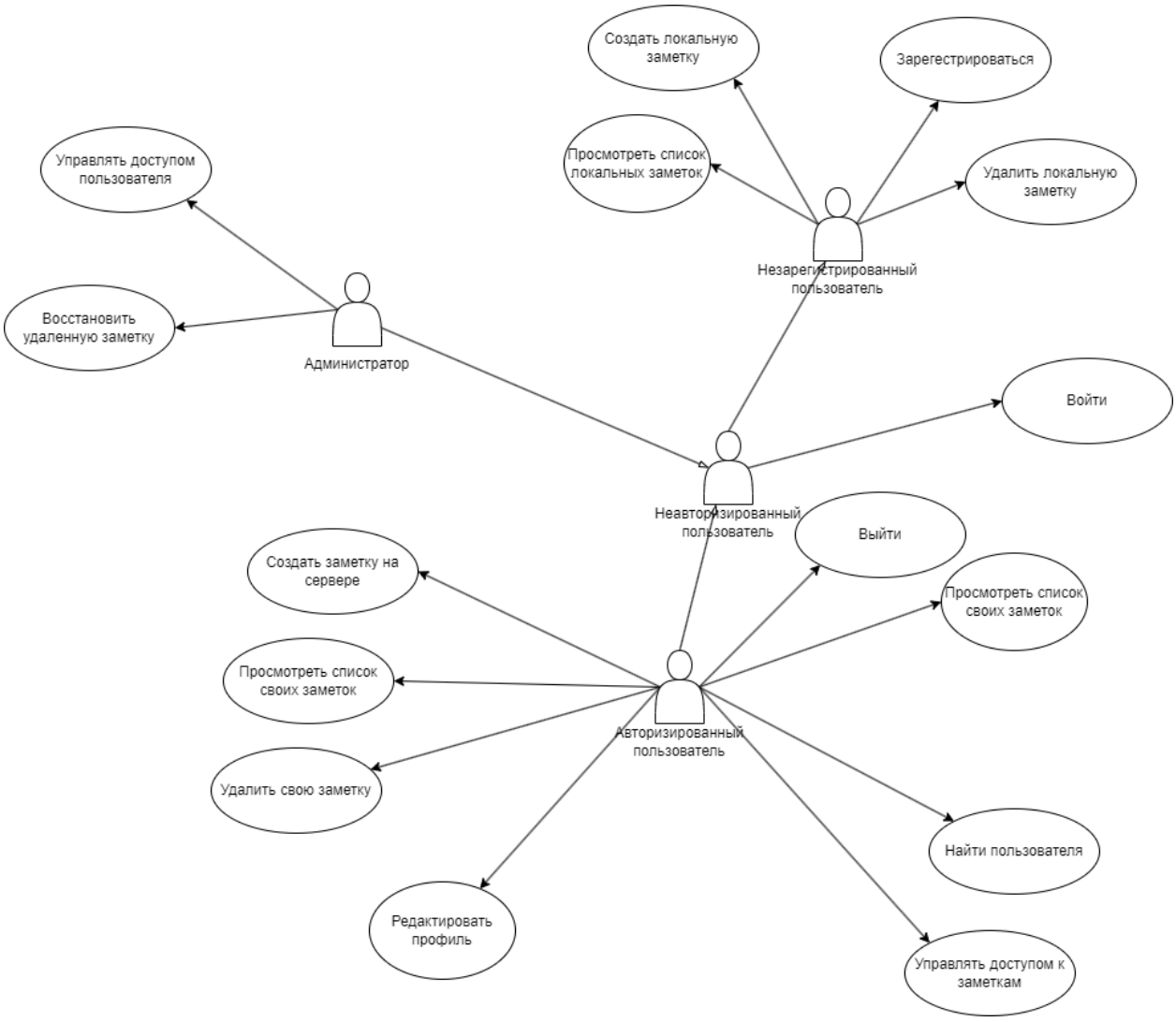

Рисунок 1. Кейсы

На диаграмме выше представлены основные кейсы системы и все они реализованы, даже чуть больше. Админ, например, умеет восстанавливать директории.

Начну описание системы с бека, потом расскажу про фронт и в конце будет список ошибок, которые я допустил при разработке системы

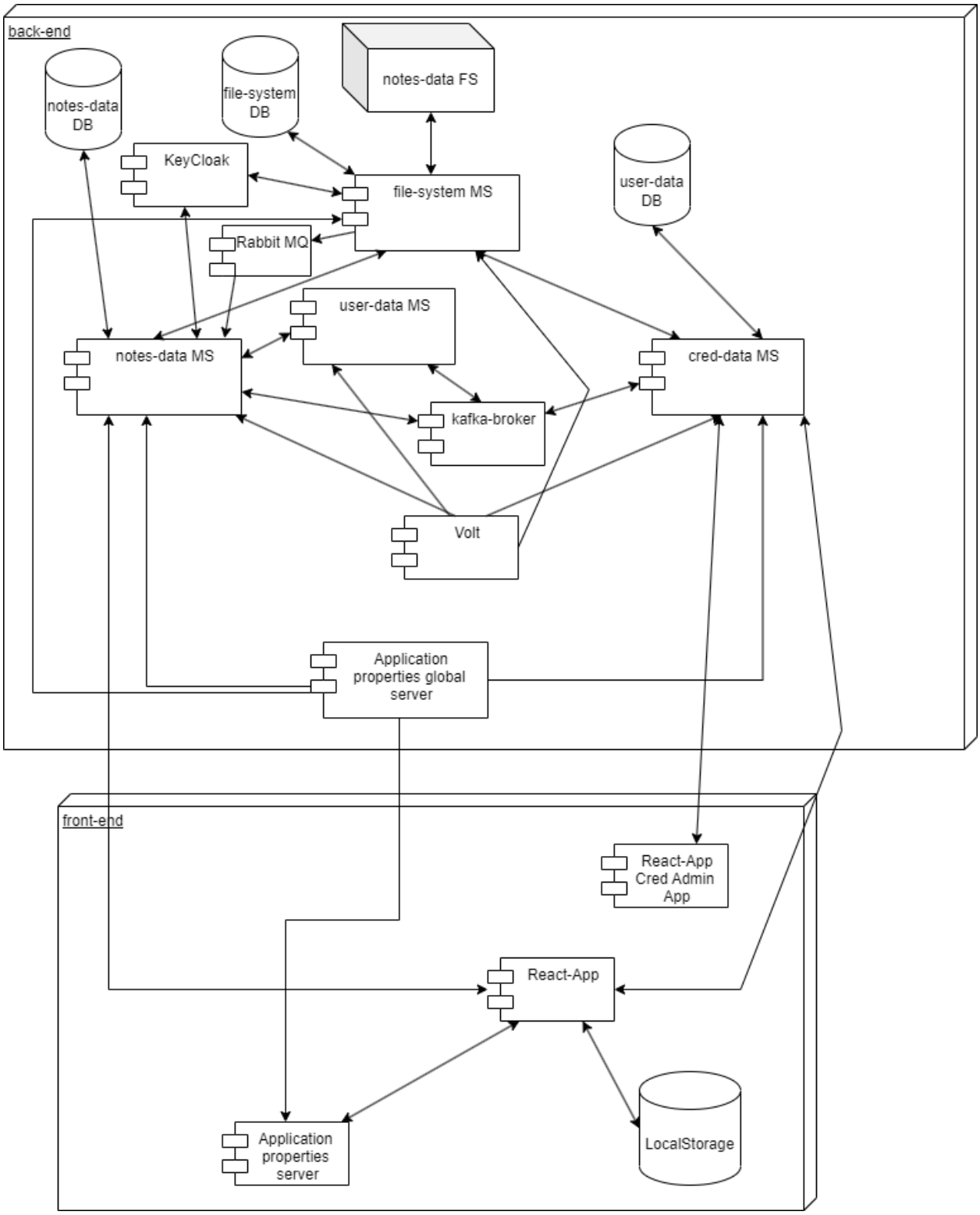

Рисунок 2. Общая схема.

Глядя на эту схему, которую я рисовал в самом начале разработки, а это было больше года назад, я тоже мало что понимаю. В общем, в чем идея – на беке есть 4 микросервиса, которые выполняют непосредственно логику приложения. Вот их список:

1 – file-system-ms

2 – registration-ms

3 – user-data-ms

4 - logic-ms

Их зоны ответственны примерно сопоставимы с их названиями, первый управляет файловой системой, в которой лежат записки пользователей, второй – регистрирует и авторизует, третий – хранит личные данные, четвертый, по сути, управляет общим доступом и некоторыми данными, которые в итоге дублируются. Про каждый подробно ниже.

file-system-ms

Данный микросервис нужен для хранения контента записок. Хранит он его в формате txt файлов, поэтому от него и отходит стрелочка к файловой системе на схеме. Файловая система там упрощенная, есть понятие «кластер» - это корневая папка каждого пользователя сервиса, в ней пользователь может создавать свои папки, а уже в них txt файлы. Создать папку внутри папки нельзя. Файлы только txt. На полноценную файловую систему это не тянет, но для текущей задачи достаточно. Авторизация перед сервисом происходит через keycloak. В этом микросервисе разделения по ролям нет, все методы доступны всем пользователям.

Помимо создания файлов ими можно управлять – перемещать, редактировать, удалять и восстанавливать. Помимо этого, большинство действий пользователя сохраняется.

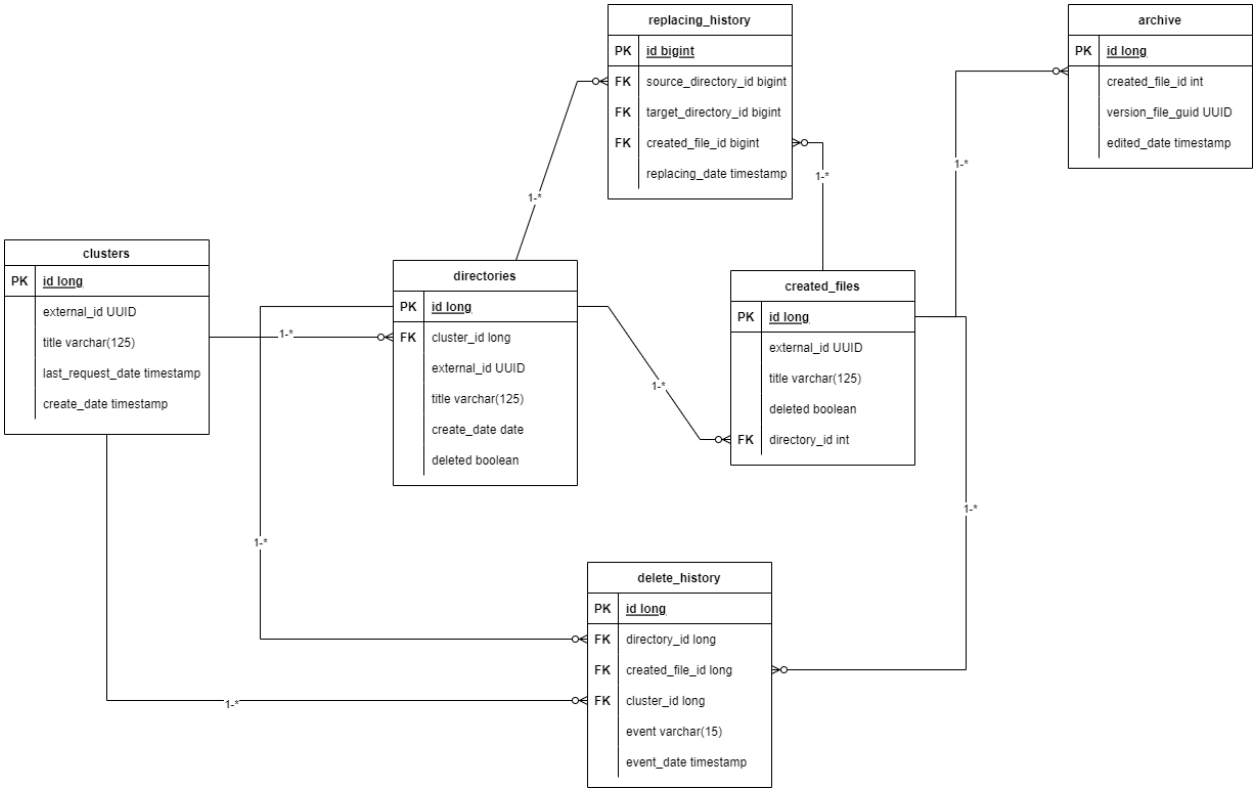

Рисунок 3. База данных.

Как видно из схемы базы данных, есть таблицы, отвечающие за хранение истории удалений (и соответственно восстановлений) объектов, а также, их перемещений. Все это можно посмотреть при желании.
Особенно мне нравится в этом сервисе идея архивирования записей. Условно говоря, можно

сказать, что у файловой системы две ветви – root, в котором хранятся кластеры и все что в них, а archive-root, в котором и лежит архив.

Архив представляет собой набор zip-архивов. Файлы туда попадают двумя путями – при удалении и при редактировании. При удалении файл помечается в бд как удаленный, архивируется и переносится из root в archive-root, при восстановлении его просто разархивируют. При удалении папки архивируются все файлы внутри нее. При восстановлении папки она восстанавливается пустой. Если восстановить файл, папка которого тоже удалена, то папка восстановится автоматически и будет содержать в себе восстанавливаемый файл.

В случае с редактированием файла, его старая версия архивируется и просто лежит в архиве, при этом, она доступа для чтения. Каждой версии назначен GUID и по нему её можно прочитать.

Сам сервис взаимодействует с внешним миром через REST и RabbitMq. В ресте используется JSON. Слать и получать никакие MultipartFile или типа того не надо. Да, контент файла тоже возвращается в JSON. Файловая система получается такая, на половину файловая.

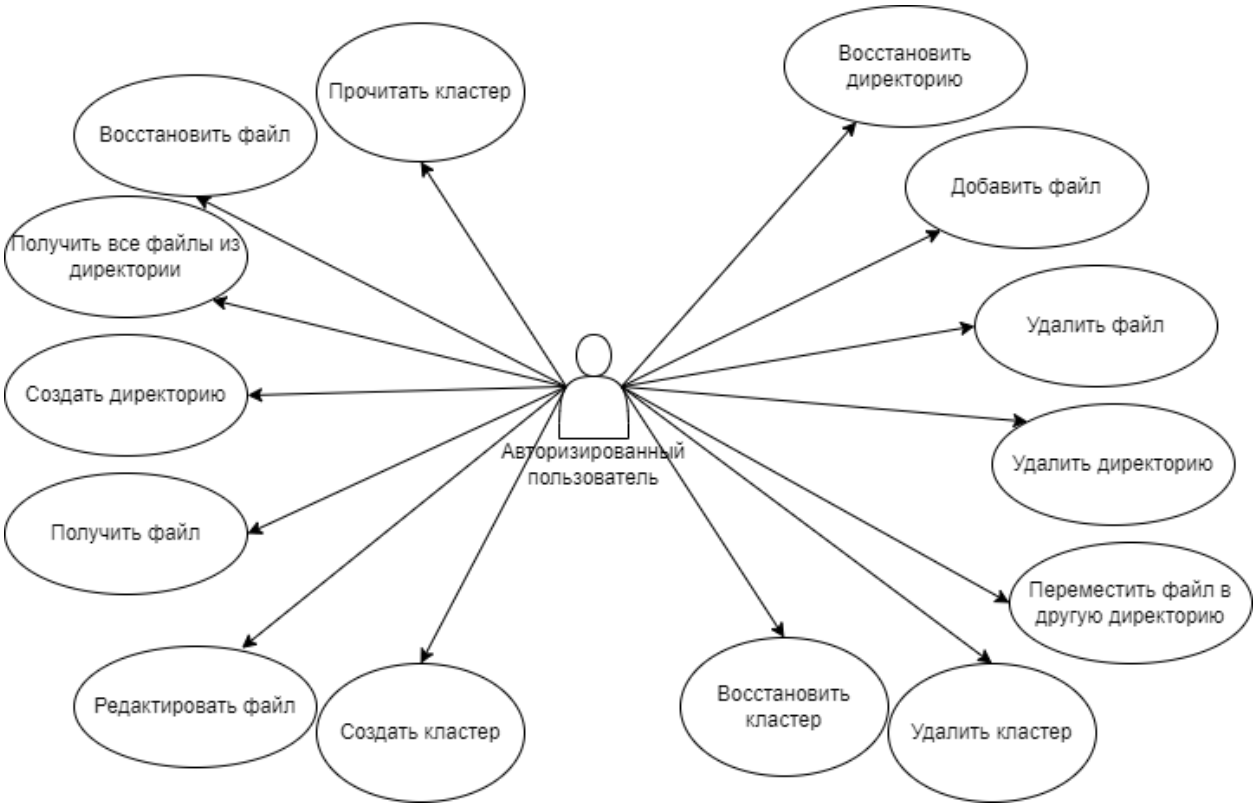

Рисунок 4. Кейсы файловой системы.

Все кейсы с рисунка вызываются через рест методы. Брокер используется для вызова отправки событийных сообщений от джоб. Джобы в сервисе три, запускаются каждый день раз в сутки. Они проверяют поле last_requested_date у кластера, которое обновляется при любом действии пользователя. Если пользователь ничего не делал время равное систему параметру, то весь его кластер архивируется. Это первая джоба. Вторая начинает слать сообщения о том, если до окончательного удаления кластера осталось немного (тоже определяется системным параметром), а третья окончательно удаляет его окончательно. Сообщения об этом и пишутся в брокер.

registration-ms

Этот МС – моя попытка сделать собственный сервис OAUTH2 авторизации. Что я в итоге сделал, но результат мне не нравится совсем. Он вышел крайне переусложненным и основам на спринговом OAUTH2 сервере, который уже давно deprecated и поработав с ним подольше я начинаю понимать почему. Главный косяк, который я с ним заметил, опишу в разделе фронта.

Идея при создании этого МС-а была в том, чтобы сделать сервер, к которому можно подключать разные системы динамически. А системы будут хранить авторизационные данные своих пользователей на этом МС-е и управлять ими.

В итоге это идея вылилась в нечто, которое я назвал многоуровневыми правами. У меня тут есть два понятия пользователя – пользователь моей системы авторизации и пользователь системы, которая подключена к моей системе. В итоге у меня тут есть «супер админ», управляющий доступом систем в этом МС, пользователь этого мс-а, который может настраивать пользователей подключаемой системы и непосредственно пользователи системы. Поэтому и ролей у меня два уровня, внутри моего мс-а, и внутри подключаемой системы. Я накостылил это через Spring Security, используя role для ролей внешних систем (ROLE_ADMIN, ROLE_USER) и authority для действий внутри системы. Роли привязываются к своим правам и уже так передаются к пользователям. Например, может ли пользователь поменять себе пароль, если да, то ROLE_USER должна быть связан с authority CHANGE_PASSWORD в базе данных.

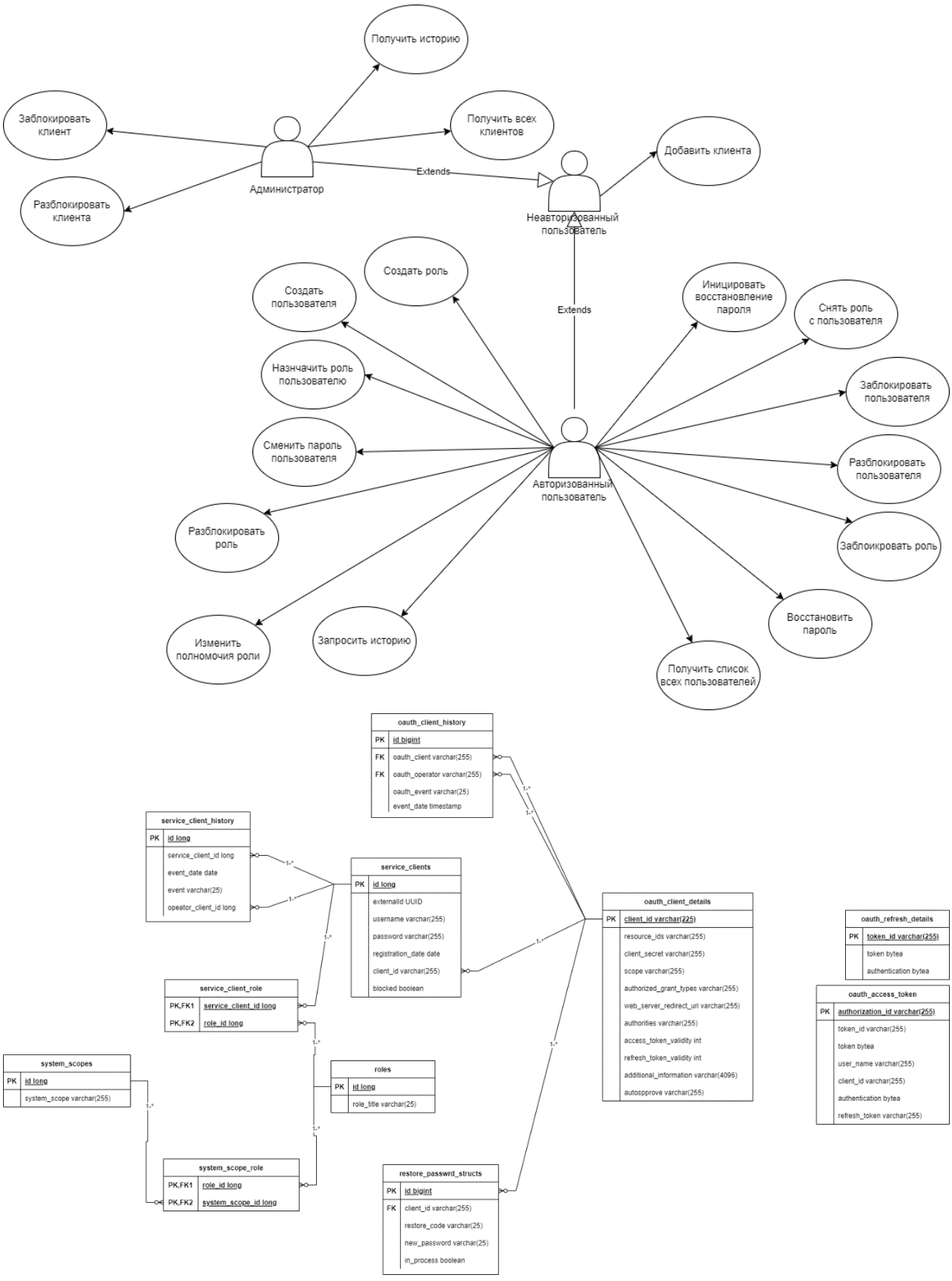

Рисунок 5. Сервис авторизации.

Вот здесь у меня две таблицы – oauth_client_details (она так называется, потому что спринг ее так называет), используется для регистрации систем, которые будут авторизовывать своих клиентов у

меня на сервисе и service_client – уже сами клиенты этого сервиса. В system_scopes лежат «полномочия», которые вяжутся с ролями из таблицы «roles», а те уже связываются с клиентами.

Две одинокие таблички справа создаются спрингом для хранения токенов, я к ним отношения не имею. Тут тоже сохраняется некая история, (бан и разбан систем), но немного. Именно этим и может заниматься супер админ. Системы регистрируются передавая в МС свое название и один из секретных паролей, лежащих в Vault.

В будущем я хотел бы переписать этот МС без использования устаревшей либы спринга, сделать его почище и с более простой структурой.

Этот МС имеет доступ в кафку и отправляет туда сообщение о регистрации «простых» клиентов, то есть клиентов внешних систем, подключенных к этому МС-у.

Но была и фича над которой мне было интересно работать здесь – восстановление пароля. Но, в итоге, получилось не слишком хорошо и как-то «распределенно-монолитно». Поэтому о ней ниже.

user-data-ms

Самый маленький МС.

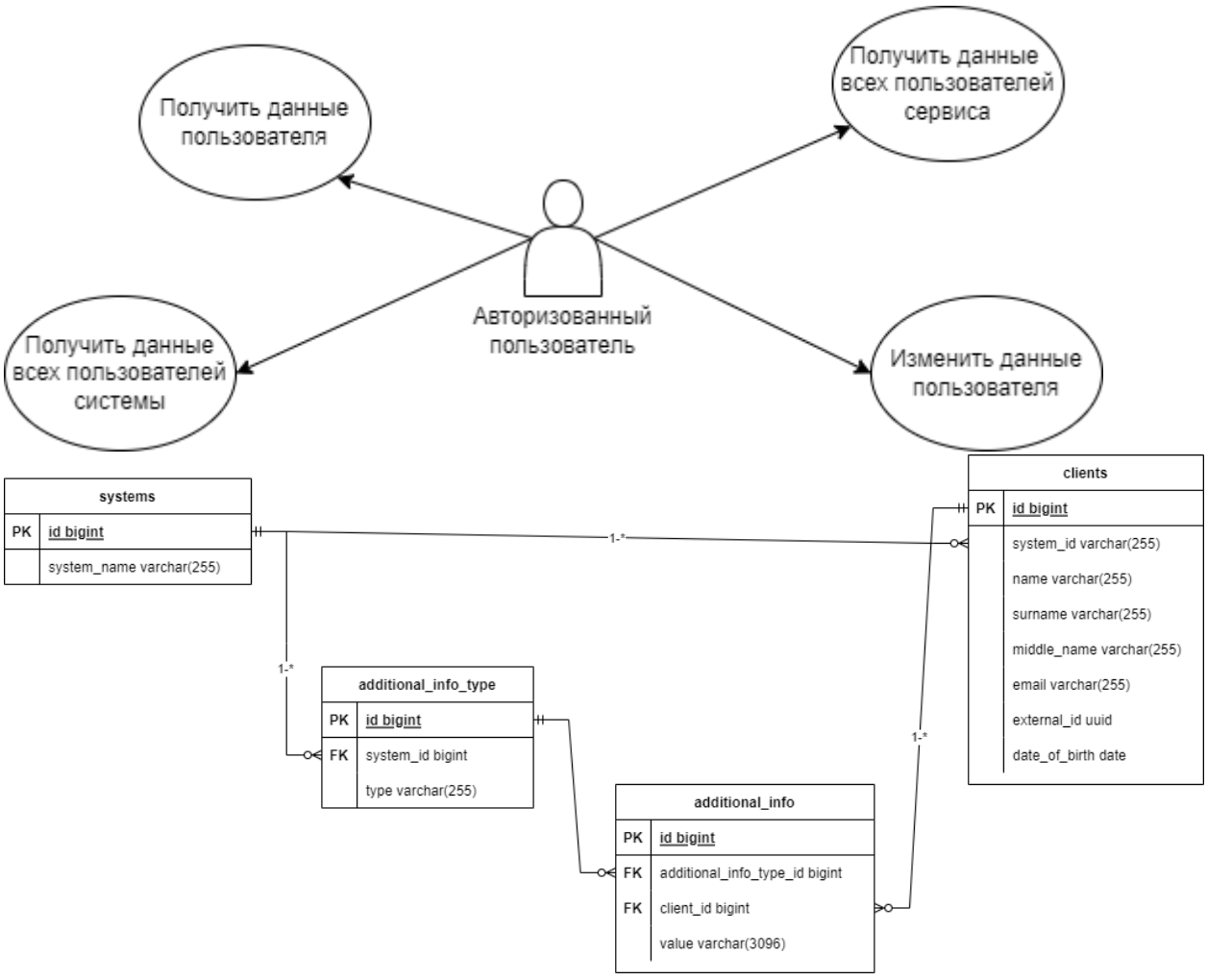

Рисунок 6. МС личных данных.

Собственно, все. Я не придумал что еще засунуть в этом мс, по сути он выполняет роль простого справочника с данными клиента, которые не важны для авторизации. Клиенты лежат внутри систем (таблица system), а так же есть возможность добавить свои типы данных помимо встроенных (таблицы additional_info и additional_info_type)

Собственно, о киллер фиче, восстановлении пароля. Когда пользователь обращается в registration-ms с запросом восстановления забытого пароля, он передает в него свою почту. Сервис авторизации делает об этом запись, генерируя для пользователя уникальный код. Далее все это добро отсылается в кафку, из кафки читается сервисом личных данных (этим), который генерирует ссылку на рест-метод сервиса авторизации и отсылает пользователю её на почту (прямо настоящее электронное письмо на настоящую электронную почту), перейдя по которой пароль успешно меняется.

На этом интересности в сервисе заканчиваются.

logic-ms

Сервис «логики» приложения.

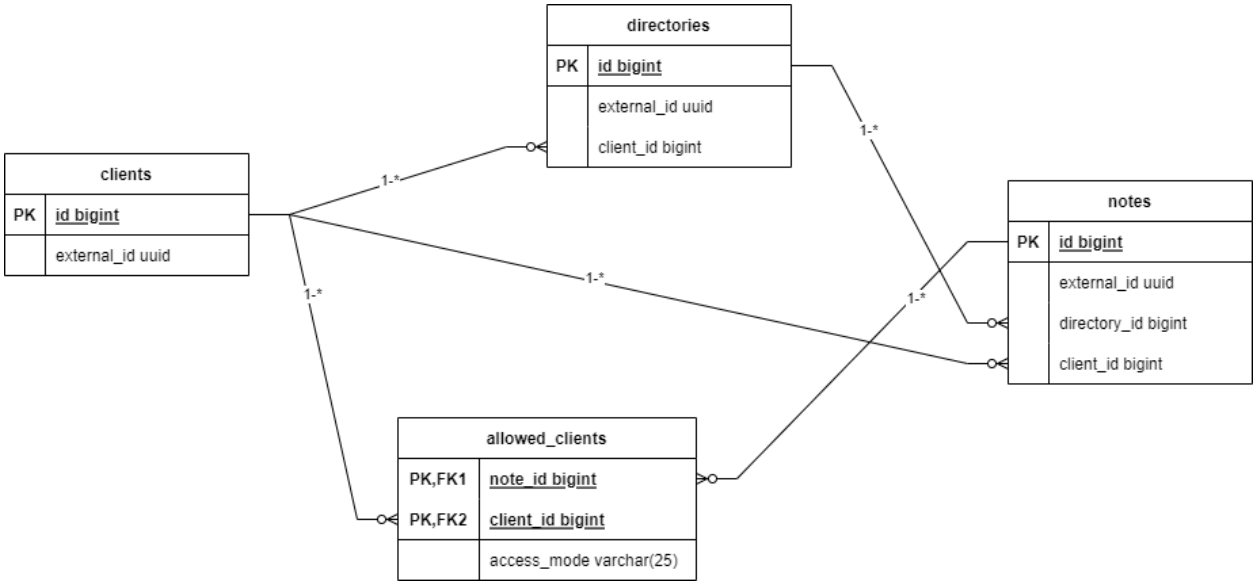

Рисунок 7. Логика.

Для него я кейсы не рисовал, так как они, по сути, повторяют кейсы всей системы.

Самой логики в этом сервисе мало, он нужен, в первую очередь, для управления общим доступом, а также представляет из себя прокси для сервиса личных данных и сервиса файловой системы. Ответы от них кешируются с помощью спринговой либы, а так же этот сервис отправляет на почту пользователю сообщения о том, что его скоро удалят.

Общее описание системы

Имея представление о всех 4 главных узлах системы, можно подняться на уровень выше.

Помимо этих 4 мсов в системы есть кейклок, волт, реббит, кафка, спринг конфиг сервер. Кейлок нужен для авторизации сервиса логики перед сервисом файловой системы, они же общаются по реббиту между собой в случае сообщений об удалении кластера.

Кафка позволяет коммуницировать сервису регистрации с сервисами личных данных и логики, волте лежат всякие пароли от баз данных, данные почтового ящика, с которого пользователям отсылаются письма и все в таком роде, в спринг конфиг сервере лежат остальные конфиги. Я использовал подход, когда конфики хранятся в самом сервисе, а не на гитхабе.

Для общего представления опишу как работает регистрация

Пользователь передает свои данные в сервис регистрации – там создается новая запись, в которой хранится логин и пароль, способы авторизации и прочее. Но здесь сохраняются не все данные, которые передал пользователь, а только те, которые нужны для авторизации. Далее сервис регистрации создает сообщение с личными данными и GUID-ом новой записи и отсылает его в кафку. Это сообщение читается сервисами личных данных и логики, сервис личных данных записывает остальные данные себе, сервис логики также создает у себя пользователя, после чего идет в кейклок, запрашивает там токен и с этим токеном идет в сервис файловой системы, создавая там кластер под нового пользователя.

А в целом любой запрос выглядит так – пользователь идет в сервис авторизации и получает там токен, в этим токеном он идет в сервис логики с запросом, тот проверяет этот токен через сервис авторизации и идет по остальным сервисам (если ему надо идти в сервис личных данных, то он авторизуется опять через сервис авторизации, если в сервис файловой системы – то через кейклок) и после отдает ответ. Конкретно здесь уже видна проблема, что один метод может затрагивать всю системы, а, следовательно, вероятность, что где-то что-то сломается становится выше. Детальнее о том, какие ошибки я сделал при разработке будет описано ниже.

Фронт

Фронт сделан на чистом React JS. Сторонних библиотек я использовал мало, первое что приходит на ум – библиотека для работы с jwt.

Перед началом разработки фронта я сделал несколько маленьких проектов для освоения html, css и базового js, но непосредственно перед разработкой основного фронта я сделал небольшое приложение из двух экранов для админки сервиса авторизации. По идее, она тоже запущена и работает, но на момент написания этого документа, я еще не придумал, на каком адресе, так что это я могу вам сказать лично.

После этого я написал небольшую либу для создания форм, она лежит на npm, ссылка будет в конце, но в итоге я отказался от нее, так как там достаточно сложная работа со стилями, после я перепишу её, чтобы было получше и попроще.

Основной фронт представляет себя приложение, объединяющие два разных приложения по технологии микрофронтов. Первое приложение из этих двух – это клиентский фронт блокнота, второе – админский. Все сделано без библиотек, упрощающих разработку, сверстано и стилизовано вручную. В будущем повторять такое, наверное, не буду)

Не вижу смысла сильно расписывать фронт, потому что если вы это читаете, то он работает, может кое-как, но работает.

Ошибки мои ошибки ошибочки

Тут без какой-либо логической последовательности, просо список того, что мне в моей системе кажется неправильным.

Архитектура приложения выглядит не как микросервисная, а как распределенный монолит. МС-ы очень сильно связаны друг с другом, нет возможности заменить какой-то из них не переписав половину системы, особенно мне не нравится, что очень много важной логики повязано на брокерах, что усложняет отлов ошибок. Думаю, это вышло потому, что, когда я начинал разработку, я думал не сколько о качестве архитектуры, столько о том, чтобы напихать всего да побольше.

Данные которые позволяют идентифицировать пользователя (и которые должны быть уникальны, по идее) размазаны по двум мс-ам. Так, в мс-е авторизации лежит логин, а в мс-е личных данных, почта, при этом ничего не запрещает иметь двум разным юзерам одинаковую почту, но метод, который отправляет письмо с восстановлением пароля определяет юзера именно по почте. Если почта повторяется, то он просто упадет.

Избыточность последнего мс-а. В нем есть пара косяков, первый на уровне написания его кода. Большая часть методов в нем – проксирование запросов к другим мс-ам. HTTP запросы сделаны с помощью FeignClient и весьма шаблонны. Эти методы не выполняют никакой логики и просто передают данные дальше. Все они реализованы как полноценные методы, но, учитывая, какие они одинаковые, стоило бы поиграться с функциональными интерфейсами. Второй – по сути, 80 процентов работы этого мс-а – это кеширование. Своей логики управления общими доступами у него мало. Поэтому сейчас, если бы я проектировал систему заново, я бы оставил в этом мс-е только управление общим доступом, а для кеширования ответов нашел другой способ, например нашел бы стороннюю систему для этого и в целом бы избавился от проксирования и фронт бы ходил в каждый нужный мс самостоятельно. Таким образом я бы отвязал кеширование от конкретного сервиса логики и, вынеся его в еще одну систему, смог бы кешировать вообще любой запрос к любому мс-у системы, если бы она росла, без добавления новых методов проксирования в мс-е логики, что сделало бы систему более «микросервисной»

Сервис авторизации. Что я понял. Использовать deprecated штуки – скорее всего ошибка. Данный мс у меня основан на спринговой либе для OAUTH2, которая как раз такая. Инфы по ней не слишком много, но я разобрался и заставил это все работать. Но произошла такая ситуация – токены у этой либы хранятся в базе данных, в то же время, в реакте, если используется StrictMode, то хук useEffect отрабатывает два раза. Если в этом хуке будет запрос токена, то в первый раз он отрабатывает нормально, а во второй раз система думает, что в таблицу к токенам надо вписать еще один и падает, потому что запись с таким первичным ключом уже существует, хотя это точно такой запрос, который был мгновение назад. Я не нашел как это починить и просто отключил стрикт мод, но осадочек остался. Поэтому больше никаких deprecated либ (только если очень надо), а этот мс я, наверное, перепишу.

Фронт. Перед тем как начать его делать, я слабо его продумал, поэтому в процессе утонул в контекстах реакта. Перед разработкой следовало нарисовать дерево объектов или какую-то схему, а также, в идеале, использовать либы для упрощения разработки, но я в целом доволен

полученным опытом. Я освоил реакт на начальном уровне и фронт в целом. В следующем приложении хочу добавить к этому TypeScript, изучить как работают гриды в верстке и использовать их (тут был только flex) и изучить state of art фронтовой разработки в целом. И, скорее всего, не буду делать сложный бек для этого. Так что, если у Компании есть потребность в неком несложном и несрочном приложении я с удовольствием возьмусь).

В самом приложении я забыл сделать восстановление кластера клиента админом, но на данный момент я делать его уже не буду, потому что за ним потянется еще что-нибудь, и я так никогда не закончу.

Ссылки

Схемы и доки - https://github.com/carakazov/notes-projects-diagrams

Сервис файловой системы - https://github.com/carakazov/file-system-ms

Сервис авторизации - https://github.com/carakazov/registration-ms

Сервис личных данных - https://github.com/carakazov/logic-ms

Фронт админа сервиса авторизации - https://github.com/carakazov/registration-ms-front-app

Сурсы библиотеки - https://github.com/carakazov/react-js-forms-library

Npm библиотеки - https://www.npmjs.com/package/react-js-forms-library

Фронт основного клиента - https://github.com/carakazov/notes-projcet-front

Фронт админа - https://github.com/carakazov/notes-admin-front