https://github.com/orgs/AgroScience-Team/repositories — ссылка на репозитории с кодом
https://miro.com/app/board/uXjVMiIA-Y0=/ -- сылка на общую архитектуру, здесь выполнялся первый этап разработки с разбором ТЗ и проектированием архитектуры

# Первый этап разработки: архитектура

Было решено разбить проект на несколько сервисов, а не писать один большой REST монолоит и уж тем более не MVC монолит. Одна из причин этого решения, но не самая значимая, это то, что разработчики в компании были знакомы с разными языками программирования, а соответственно и фрейморками. Так как были свои временные рамки, было решено сделать так, что все будут использовать те технологии, что больше всего им знакомы. Это связано с тем, что в одной только ORM можно погрязнуть как минимум на две недели, чтобы начать ей эффективно пользоваться, а у нас было много технологий, которые и без того было необходимо освоить. Ниже приведены плюсы и минусы, которые имели самое большое значение на решение разделить проект на сервисы.

Плюсы разделения проектов на сервисы:
1. Масштабируемость: Разделение на сервисы позволяет горизонтально масштабировать каждый сервис независимо от других. Это означает, что вы можете увеличивать ресурсы только для тех сервисов, которые испытывают большую нагрузку, вместо увеличения ресурсов всего монолитного приложения.
2. Гибкость разработки: Каждый сервис может быть разработан и развернут независимо от других. Это позволяет командам разработчиков работать над разными сервисами независимо, ускоряя процесс разработки и обеспечивая более гибкую архитектуру.
3. Легкость поддержки: При возникновении проблемы в одном из сервисов, вы можете сосредоточиться на его решении без влияния на остальные сервисы. Это упрощает процесс обслуживания и устранения неполадок.

Минусы разделения проектов на сервисы:
1. Сложность управления: С разделением на сервисы, у вас будет больше компонентов, которые нужно управлять и координировать. Это может добавить сложности в развертывании, мониторинге и отладке.

2. Сетевая задержка: При использовании API Gateway для объединения сервисов в одну точку входа, возникает дополнительная сетевая задержка. Каждый запрос должен пройти через API Gateway, что может немного замедлить общую производительность системы. Этот минус мы максимально постарались свести к минимуму, выбрав Nginx для написания api gateway. Собственно за лёгкость и быстроту работы Nginx и ценится.

3. Сложность тестирования: Разделение на сервисы требует более сложного тестирования, поскольку вам нужно проверить взаимодействие между различными сервисами и обработку ошибок при передаче данных между ними. В целом, разделение проектов на различные REST сервисы имеет множество преимуществ, таких как масштабируемость и гибкость разработки. Однако, оно также вносит дополнительную сложность в управление и тестирование системы. При выборе подхода следует учитывать требования и особенности вашего проекта.

В итоге, оценив время необходимое на изучение технологий и изучив ТЗ, было решено ограничиться написанием пяти сервисов: авторизации, профилей, погоды, полей и api gateway.

P. S. API Gateway - это слой, который предоставляет единую точку входа для клиентов, обеспечивая маршрутизацию, контроль доступа и другие функции для взаимодействия с микросервисами или другими внутренними и внешними API.

Основная цель API Gateway состоит в том, чтобы упростить и улучшить управление, контроль и безопасность взаимодействия клиентов с множеством сервисов или API.

Вот некоторые основные причины использования API Gateway:

1. Единая точка входа: API Gateway предоставляет клиентам единую точку входа для доступа к нескольким сервисам или API. Вместо того, чтобы клиентам необходимо было знать и использовать разные URL и конечные точки для каждого сервиса, они могут использовать один URL API Gateway.

2. Маршрутизация и перенаправление: API Gateway может маршрутизировать запросы клиентов к соответствующим сервисам на основе определенных правил. Он также может выполнять перенаправление запросов на другие микросервисы или внешние API.

## Базы данных

Было решено использовать СУБД PostgreSQL как реляционная БД для всех микросервисов. Причина проста, в сервисе полей была необходимость использовать расширение PostGIS для хранения контуров полей. Раз сервис полей вынужден без возможности выбора использовать PostgreSQL, то на остальных сервисах мы решили тоже использовать её. В целом это на данный момент одна из самых популярных баз данных среди новых проектов на рынке. Зачастую, при трудоустройстве в наше время, мы встретим именно эту СУБД.

Для каждого сервиса было решено использовать свою базу данных. Так как у нас 4 сервиса, которые содержат определённую бизнес логику, то для них существует соответственно 4 базы данных. Каждый сервис не имеет прямого доступа к базам данных других сервисов.

Существует несколько причин, почему считается хорошей практикой для каждого микросервиса использовать свою базу данных и не обращаться напрямую к базам данных других сервисов:

1. Изоляция данных: Каждый микросервис должен быть независимым и изолированным, имея собственную базу данных. Это позволяет избежать проблем совместного использования данных между сервисами и упрощает разработку, масштабирование и управление каждым микросервисом отдельно.

2. Границы контекста: Каждый микросервис имеет свою собственную область ответственности и определенный контекст. Использование отдельных баз данных помогает ясно определить границы контекста каждого сервиса и упростить его разработку и поддержку.

3. Управление данными: Когда каждый микросервис имеет свою базу данных, это облегчает управление данными, так как каждый сервис может использовать свою схему данных, индексы и оптимизации, наиболее подходящие для его конкретных требований. Это также позволяет легко изменять схему данных для отдельных сервисов без влияния на другие сервисы.

4. Гибкость и масштабируемость: Использование отдельных баз данных для каждого микросервиса обеспечивает гибкость и масштабируемость. Каждый сервис может масштабироваться независимо от других, а также использовать технологии и инструменты, наиболее подходящие для его требований по производительности и масштабируемости.

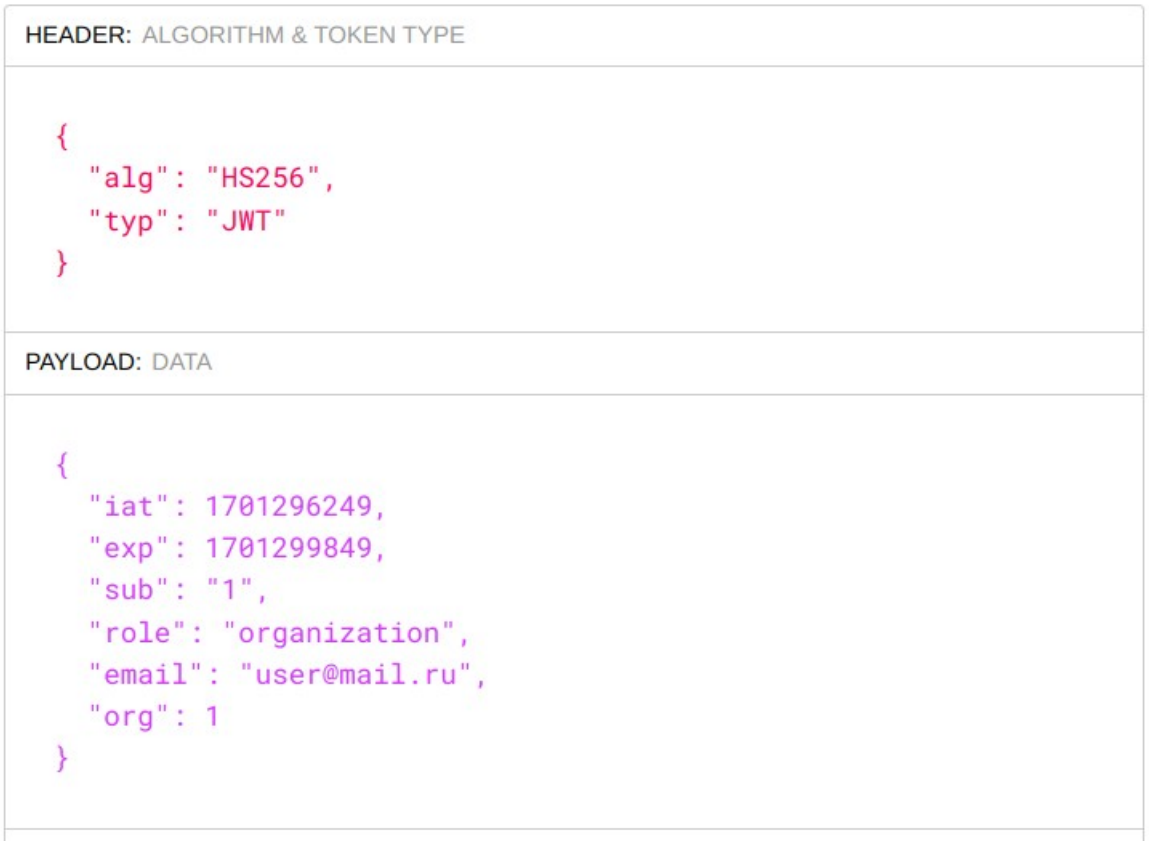
5. Безопасность: Использование отдельных баз данных помогает обеспечить безопасность данных. Каждый микросервис имеет свои собственные учетные данные для доступа к базе данных, и это уменьшает риск несанкционированного доступа к данным других сервисов.

## Сборка проекта

Для сборки проекта было решено использовать Docker. Каждый сервис имеет свой докер файл, поэтому запуск каждого из сервисов осуществляется с помощью докера по отдельности. Все поднятые контейнеры объеденены в одну docker network под названием agronetwork. Это необходимо, чтобы сервисы могли общаться между собой и были изолированы от ip физической машины.

## Регистрация/Аутентификация

Задача регистрации и аутентификации возложена на auth сервис. Пользовательские сессии контролируются с помощью JWT. Каждый JWT токен содержит следующую информацию.




Каждый токен содержит эту информацию. Sub — id пользователя, org — id организации.

Если в auth сервис придёт старый токен, то сервис заблокирует доступ и пользователю придётся заново проходить процедуру аутентификации.

Любой запрос, который приходит с фронта, содержит токен. В ApiGateway берется этот токен и отправляется в auth сервис и только после удачного ответа изначальный запрос будет перенаправлен туда, куда посылался. Авторизацию по роли уже осуществляет каждый сервис по отдельности путём декодирования JWT токена.

Сервис написан на Python с использованием лёгкого веб фреймворка FastApi и крайне популярный фреймворк SQLAlchiemy для взаимодействия с БД

## Сервис полей

Fields сервис содержит ключевую бизнес логику. Он позволяет работать с культурами, севооборотами, полями и агрохимией. Собственно под это выделяется 4 таблицы. Посевы нельзя менять со стороны фронтенда, это возможно сделать исключительно с помощью терминала с помощью curl при непосредственном подключении к серверу, на котором крутится API. Во время миграций в базу данных записывается чуть больше 200 культур, взятых из приложения OneSoil.

Одна из важных бизнес задач этого сервиса — предоставлять только те данные, которые

принадлежат организации, которая отправила запрос. То есть организация A, не может получить поля и иную информацию организации B.

Сервис полей не осуществляет работу с какими либо снимками. И в целом в проекте на данной итерации это не предусмотрено. Это связано с большой архитектурной сложностью внедрения этой логики в проект, поэтому было решено оставить это на следующие итерации. А добавление возможности хранения маленькой фотографии для превью поля не имено иной ценности, кроме как эстетической, поэтому это было откинуто, чтобы уже полноценно внедрить этот функционал вместе с остальной бизнес логикой аэроснимков. На данной итерации была собрана информация о потенциальных технологиях, которые могут быть использованы для хранения и обработки снимков: это, в первую очередь, QGis как стороннее API и MinIO как S3 хранилище. Непосредственно Александр в течения семестра получил опыт работы с MinIO, что несколько продвинуло исследовательскую составляющую по снимкам.

Сервис написан на Java с использованимем веб фреймворка Spring Boot, Hibernate — ORM для работы с БД и JbdcTemplate для прямых запросов в БД.

### Сервис профилей

На данном этапе этот сервис осуществляет хранение пользовательских данных, планируется, что он будет использован для генерации различных популярных документов в рамках предметной области.

Сервис написан на Python с использованимем лёгкого веб фреймворка FastApi и крайне популярный фреймворк SQLAlchiemy для взаимодействия с БД

### Тестирование

Проводилось ручное тестирование посредством Postman.

В сервисе полей присутствуют интеграционное тестирование с покрытием 54 % кода бизнес логики. Тестами не покрыта логика, связанная с получением данных. В первую очередь была цель протестировать ту логику, которая влияет на персистентность БД. Собственно, эта цель была выполнена в рамках сервиса полей.

### Сервис метео

Данный сервис представляет собой ETL-систему. Она занимается тем, что берет данные с внешнего сервиса, который предоставляет информацию о погоде в какой-то местности, затем она эту информацию преобразовывает в тот вид, который требуется в соответствии с определенной в проекте бизнес-логикой и сохраняет ее в таком преобразованном виде в своей локальной БД. Так же данная система имеет достаточную степень автоматизации, суть которой состоит в том, чтобы сервис по заданному расписанию обновлял информацию о каждом поле, существующем в системе. Целью данного сервиса было настроить доступ к данным о

погоде для конкретного поля так, чтобы это было быстро и не ухудшало пользовательский опыт, что было в полной мере достигнуто благодаря выбранному подходу.

Сервис написан на Java с использованимем веб фреймворка Spring Boot, Hibernate — ORM для работы с БД.

**Второй этап разработки: Написание сервисов**

Было необходимо изучить технологии, которые были выбраны на этапе проектирования. Это заняло некоторое время. Основные проблемы возникли с докером и настройкой Nginx в качестве ApiGateway.

1) Были проблемы с общением между докер контейнерами из-за малого знания сей технологии. Впоследствии стало ясно, что контейнеры могут обращаться друг другу не по ip в качестве хоста, а по имени, если они обёрнуты в одну докер сеть

2) Были проблемы с настройкой Nginx, так как никто в коллективе не имел опыта работы с этой технологией. Основные проблемы, которые возникли и были решены это:

2.1) Было неизвестно как отделить заголовок с JWT от запроса с фронта, чтобы отправить его на сервис авторизации для прохождения процесса аутентификации. Это решилось путём чтения документации и упорного труда.

2.2) Была проблема с CORS на этапе разработки фронтенда. Проблема решалась перехватыванием HTTP OPTIONS запроса внутри Nginx и его обработкой.

3) Было также множество мелких проблем/задач, описание которых есть в github issues.

4) Была серьзеная архитектурная проблема из-за, которой было потеряно время. Изначально было запланировано не 5, а 6 сервисов. То есть сервис fields был изначально разделён на сервис fields, который содержал только таблицу полей, агрохимии и логику работы с ними, и сервис crops, который содержал логику работы с севооборотами и культурами. Мы полагали, что это две малосвязные бизнес логики, но на этапе непосредственной разработки, мы выяснили, что они по итогу сильно связные и вынуждены часто общаться по HTTP, что создавало проблемы с разработкой, поддержкой. К тому же это перегружало внутреннюю сеть.

Благо эту проблему быстро заметили, поэтому слияние двух сервисов в один заняло не много времени, чем могло бы.

## Третий этап: интеграционное тестирование, ручное тестирование, отладка багов

На этом этапе была проведена работа по написанию интеграционных тестов для сервиса полей посредством фреймворка Junit 5 с покрытием в 54%. Также было проведено полное ручное тестирование посредством Postman всей работы API части. То есть поднимались все контейнеры и слались запросы на ApiGateway.

Соответственно возникающие баги приходилось исправлять. В частности исправлялись баги, о которых сообщали товарищи фронтендеры, как например, ранее описанная проблема с CORS.

Также была проведена оптимизационная работа: были накинуты недостающие индексы в базы данных, переписаны некоторые запросы в БД с учётом логирования, чтобы не нагружать БД. Оптимизационный этап проводился в течении всей работы, нельзя сказать, что он был отделён во времени от остальных процессов.