МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

Национальный исследовательский ядерный университет «МИФИ»

Институт интеллектуальных кибернетических систем

КАФЕДРА КИБЕРНЕТИКИ (№ 22)

Направление подготовки 09.03.04 Программная инженерия

Пояснительная записка

к научно-исследовательской работе студента на тему:

Разработка серверного компонента клиент-серверной программной системы диагностики щитовидной железы по ультразвуковым снимкам

Группа Б21-504

Студент _____ Попов З.А.

Руководитель _____ Зайцев К.С.

Научный консультант _____ Дунаев М.Е.

Оценка руководителя 12

Оценка комиссии _____

Члены комиссии

_____ _____

_____ _____

_____ _____

_____ _____

Москва 2024

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

Национальный исследовательский ядерный университет «МИФИ»

Институт интеллектуальных кибернетических систем

КАФЕДРА КИБЕРНЕТИКИ (№ 22)

Задание на НИР

Студенту гр. Б21-504 Попов Захар Андеевич

ТЕМА НИР

Разработка серверного компонента клиент-серверной программной системы диагностики щитовидной железы по ультразвуковым снимкам

ЗАДАНИЕ

№ п/п	Содержание работы	Форма отчетности	Срок исполнения	Отметка о выполнении Дата, подпись
1.	**Аналитическая часть**			
1.1.	Анализ и сравнение архитектур построения серверных приложений			
1.2.	Анализ и сравнение паттернов проектирования модулей системы			
1.3.	Анализ инструментов применяемых для реализации системы			
2.	**Теоретическая часть**			
2.1.	Разработка моделей микросервисов			
2.2.	Разработка общей модели системы			
3.	**Инженерная часть**			
3.1.	Проектирование модулей на основе разработанных моделей			
3.2.	Проектирование системы на основе разработанных модели взаимодействия			
4.	**Реализация**			
4.1.	Реализация системы			

ЛИТЕРАТУРА

Дата выдачи задания: 26.12.2024

Руководитель _____ Зайцев К.С.

Студент _____ Попов З.А.

Реферат

Общий объем основного текста, без учета приложений 38 39 страниц, с учетом приложений 44 ??. Количество использованных источников 15. Количество приложений 1.

МИКРОСЕРВИСЫ, GOLANG, РАСПРЕДЕЛЕННЫЕ ТРАНЗАКЦИИ, POSTGRESQL, REDPABDA

Целью данной работы является Разработка серверного компонента клиент-серверной программной системы диагностики щитовидной железы по ультразвуковым снимкам

В первой главе проводится обзор и анализ предметно области и инстсрументов

Во второй главе описываются использованные и разработанные модели системы и модулей.

В третьей главе приводится результаты проектирования по разработанным моделям.

Содержание

Введение

Исследование щитовидной железы на наличие узловых образований является крайне сложной и комплексной задачей, требующей высокой квалификации медицинских работников. Для определения природы узлов используется специальная система классификации EU-TIRADS, которая основана на различных характеристиках узлов, таких как размер, форма и эхогенность. Однако, ручная классификация узлов по системе EU-TIRADS может быть трудоемкой и приводить к ошибкам. В связи с широким использованием методов глубокого обучения в различных сферах жизни, эти методы применяются и для решения задачи диагностики эндокринологических заболеваний. Для того, чтобы эти методы были эффективно использованы в повседневной работе медицинских работников, проводящих эндокринологические исследования, необходимо разработать серверное приложение для интуитивного взаимодействия медицинского персонала с технологиями искусственного интеллекта. Цель работы заключается в разработке серверного приложения для упрощения классификации заболеваний по EU-TIRADS. Для достижения этой цели необходимо разработать архитектуру серверного приложения или доработать уже имеющуюся, развернуть приложение на единой цифровой платформе РФ «ГосТех», а также провести нагрузочное тестирование с замерами скорости обработки и количества одновременно выполняющихся запросов. Подобные технологии уже внедряются в качестве дополнительного программного обеспечения, интегрированного в аппараты ультразвуковой диагностики или в качестве прикладного программного обеспечения. Однако, на данный момент на российском рынке подобные технологии не представлены. Поэтому, разработка серверного приложения для упрощения классификации заболеваний по EU-TIRADS является важным шагом в развитии эндокринологической диагностики. В первом разделе данной работы проводится анализ проблемной области, исследование существующих инструментов для создания серверного приложения, анализ имеющейся архитектуры программной системы, рассматриваются трудности, возникающие при миграции серверного приложения с одного языка программирования на другой. Также ставятся цели и задачи НИР. Во втором разделе проводится сравнительный анализ инструментов для создания серверного приложения и выбор наиболее подходящего, а также анализируются необходимые изменения в базе данных. В третьем разделе описана разработка выбранной архитектуры приложения, компоненты и их взаимодействие меж-

ду друг другом и интеграция с внешними системами. В четвертом разделе произведена программная реализация серверного приложения и проведено его тестирование.

1. Аналитическая часть

1.1 Анализ проблемной области и архитектур построения серверных приложений

Традиционная архитектура на основе клиент-серверной модели является одной из наиболее распространенных архитектур для создания серверных приложений. Эта модель предполагает разделение приложения на две составляющие - клиентскую и серверную. Клиентская составляющая отвечает за интерфейс взаимодействия с пользователем, а серверная - за обработку запросов и предоставление данных. Основным преимуществом клиент-серверной модели является возможность создания распределенной архитектуры, которая позволяет создавать приложения с большими объемами данных или с большим количеством пользователей. Также этот подход позволяет легко масштабировать и обновлять серверное приложение без влияния на клиентскую составляющую. Одним из недостатков клиент-серверной модели является ее зависимость от сети. Отказ сети или недоступность сервера может привести к невозможности доступа к приложению. Еще одним недостатком является необходимость настройки и поддержки серверной инфраструктуры, что может требовать отдельных затрат и увеличить стоимость разработки и поддержки серверного приложения. Для реализации клиент-серверной модели на практике используются различные технологии и инструменты, как проприетарные, так и открытые. Например, для создания серверных приложений могут использоваться языки программирования, такие как Java, Python, Ruby, PHP, а для создания клиентских приложений - HTML, CSS, JavaScript. Существует также множество фреймворков, библиотек и инструментов, которые предназначены для упрощения разработки серверных приложений на основе клиент-серверной модели. Например, Node.js является популярным фреймворком для создания серверных приложений на JavaScript, а Django и Flask - для Python. Также используются такие средства, как базы данных, предназначенные для хранения данных приложения, или сервисы и инструменты для автоматизации и управления развертыванием и масштабированием серверных приложений. Кроме того, существуют такие платформы, как Amazon Web Services и Microsoft Azure, которые предоставляют в облаке готовые инструменты для создания и развертывания серверных приложений на основе клиент-серверной модели.

1.1.1 Микросервисная архитектура

Микросервисная архитектура является относительно новым подходом к разработке серверных приложений. Она характеризуется модульностью, то есть каждый модуль является отдельным сервисом, который может быть развернут и масштабирован отдельно от других модулей. Микросервисная архитектура рассматривает приложение как совокупность небольших сервисов, которые представляют отдельные функции приложения. Каждый сервис обычно работает независимо от других сервисов, используя API для обмена данными между другими сервисами. Основное преимущество микросервисной архитектуры заключается в ее модульности. Это дает возможность быстро отвечать на изменения требований и легко масштабировать приложение по мере необходимости. Другим преимуществом микросервисной архитектуры является возможность использования различных технологий и языков программирования для различных сервисов. Это позволяет использовать наилучшее решение для каждой отдельной задачи. Микросервисная архитектура может быть также более устойчивой и надежной, поскольку имеется возможность аварийного отключения отдельных сервисов, позволяя уменьшить влияние отказа других компонентов на работу приложения в целом. Существует несколько типов микросервисных архитектур, которые различаются по тому, как организованы и взаимодействуют между собой микросервисы. К одному из распространенных типов микросервисных архитектур относится оркестрованная архитектура [8] – это такая архитектура, в которой существует централизованный компонент или слой, называемый оркестратором, который координирует действия и взаимодействие других микросервисов в системе. Оркестратор отвечает за управление жизненным циклом запросов от клиентов, контроль последовательности выполнения операций и обработку бизнес-логики интеграции между сервисами. Оркестрованная архитектура обеспечивает высокий уровень контроля и координации в распределенных системах, позволяя эффективно управлять сложными процессами и задачами. В то же время, она может создавать единую точку отказа, увеличивая зависимость системы от работы оркестратора. Ещё одним из распространённых типов микросервисных архитектур является event-driven архитектура – архитектура, в которой микросервисы взаимодействуют через отправку и прием событий. Каждый сервис может публиковать события, на которые другие сервисы могут, соответственно, подписываться и реагировать. К её преимуществам относятся: гибкость (компоненты могут быть легко добавлены или удалены без влияния на другие части системы), масштабируемость (каждый компонент может масштабироваться независимо для обеспечения высокой произ-

водительности), отказоустойчивость (отказ одного компонента не блокирует работу всей системы). Однако тут возникает сложность в обеспечении согласованности данных между различными сервисами, управлении событиями и обработке ошибок. Также стоит упомянуть про подход API Gateway – данный тип архитектуры использует централизованный шлюз, который является единой точкой входа для всех запросов клиентов к микросервисам или другим системам. Шлюз может выполнять несколько функций, к которым относятся: аутентификация, авторизация, маршрутизация запросов, преобразование протоколов и форматов данных, кеширование, балансировка трафика и другие. Данный подход позволяет сделать архитектуру приложения более гибкой, безопасной и легкой в управлении, а также позволяет централизованно управлять всеми взаимодействиями клиентов с микросервисами, упрощая развертывание и поддержку системы.

Микросервисная архитектура предусматривает разбиение приложения на небольшие независимые сервисы, каждый из которых выполняет строго определённые задачи. Эти сервисы взаимодействуют через сети с использованием API или брокеров сообщений.

Преимущества микросервисной архитектуры:

- **Гибкость масштабирования**: отдельные сервисы масштабируются независимо, что оптимизирует использование ресурсов.
- **Свобода выбора технологий**: команды могут использовать разные языки программирования и инструменты для реализации отдельных сервисов.
- **Лучшая управляемость**: изолированные сервисы упрощают внесение изменений и поддержку.
- **Повышенная устойчивость**: сбой одного сервиса не влияет на работу других.
- **Параллельная разработка**: команды могут работать над разными сервисами одновременно.

Недостатки микросервисной архитектуры:

- **Сложность проектирования**: требует тщательного планирования взаимодействия между сервисами.
- **Проблемы с согласованностью данных**: переход к eventual consistency добавляет сложности.
- **Высокие инфраструктурные затраты**: управление множеством сервисов требует дополнительных инструментов, таких как Kubernetes.
- **Увеличение задержек**: сетевое взаимодействие между сервисами медленнее, чем вызовы в памяти.

– **Сложности диагностики**: распределённый характер системы затрудняет выявление и исправление ошибок.

1.1.2 Монолитная архитектура

Монолитная архитектура представляет собой подход, при котором всё приложение разрабатывается как единый, неделимый блок. В нём объединены пользовательский интерфейс, бизнес-логика и уровень работы с данными. Такой подход исторически был основным в разработке ПО.

Преимущества монолитной архитектуры:

– **Простота разработки**: начальный этап разработки требует меньше усилий на планирование структуры.
– **Единая кодовая база**: централизованное хранение и управление кодом упрощает процесс разработки и интеграции изменений.
– **Лёгкость развертывания**: однотипные развертывания минимизируют риск несовместимости.
– **Высокая производительность**: отсутствие сетевого взаимодействия между компонентами ускоряет их взаимодействие.
– **Поддержка транзакционности**: встроенные механизмы работы с транзакциями позволяют обеспечить согласованность данных.

Недостатки монолитной архитектуры:

– **Сложность масштабирования**: увеличение нагрузки требует масштабирования всего приложения, даже если это касается лишь одного его компонента.
– **Ограничения технологического выбора**: единая технологическая платформа ограничивает возможности внедрения новых инструментов.
– **Рост сложности кода**: по мере увеличения объёма функциональности поддержка системы становится сложнее.
– **Долгое развертывание**: внесение малейших изменений требует пересборки и перезапуска приложения.
– **Уязвимость к сбоям**: сбой в одном компоненте приводит к отказу всей системы.

1.1.3 Сравнение монолитной и микросервисной архитектур

Для более наглядного понимания различий между монолитной и микросервисной архитектурами представим их основные аспекты в таблице:

Аспект	Монолитная архитектура	Микросервисная архитектура
Сложность разработки	Низкая	Высокая
Масштабируемость	Ограниченная	Гибкая
Устойчивость к сбоям	Низкая	Высокая
Технологический выбор	Ограниченный	Гибкий
Развертывание	Простое	Сложное
Поддержка транзакционности	Простая	Сложная
Задержки взаимодействия	Минимальные	Увеличенные
Затраты на инфраструктуру	Низкие	Высокие

Таблица 1.1 – Сравнение монолитной и микросервисной архитектур

Выводы: Монолитная архитектура лучше подходит для проектов с ограниченным функционалом и небольшими командами, где важны простота и быстрота разработки. Микросервисы же оправданы в сложных, масштабируемых системах с распределёнными командами и высокими требованиями к отказоустойчивости.

1.2 Анализ и сравнение паттернов проектирования модулей системы

Связанность (англ. Coupling) и согласованность (англ. Cohesion) являются фундаментальными концепциями в области разработки программного обеспечения, которые оказывают значительное влияние на читаемость, поддержку, тестируемость и переиспользование кода. В данной работе исследуются эти понятия, их виды, а также взаимосвязь между ними с целью формирования рекомендаций по их применению для повышения эффективности разработки.

Связанность (Coupling)

Связанность характеризует степень зависимости одного программного модуля от других. Чем выше уровень связанности, тем сильнее изменения в одном модуле отражаются на других. Стремление к низкой связанности (low coupling) является ключевым принципом проектирования, позволяющим минимизировать количество зависимостей между модулями и повысить их автономность.

Существует несколько типов связанности, классифицируемых по степени их зависимости:

- **Content coupling** (содержательная связанность): возникает, когда один модуль полагается на внутреннюю реализацию другого модуля, например, доступ к его локальным данным. Изменения в одном модуле требуют изменений в другом.
- **Common coupling** (общая связанность): модули используют общие данные, например, глобальные переменные. Изменение этих данных затрагивает все модули, ко-

торые с ними работают.

- **External coupling** (внешняя связанность): несколько модулей зависят от общего внешнего ресурса, такого как сервис или API. Изменения в этом ресурсе могут привести к некорректной работе модулей.
- **Control coupling** (управляющая связанность): один модуль управляет поведением другого через передачу управляющих сигналов, например, флагов, изменяющих поведение функций.
- **Stamp coupling** (штампованная связанность): модули обмениваются сложными структурами данных, однако используют лишь их часть. Это может приводить к побочным эффектам из-за неполной или изменяемой информации.
- **Data coupling** (связанность по данным): наименее инвазивный тип, при котором модули взаимодействуют исключительно через передачу конкретных данных. Такой подход минимизирует взаимозависимость.

Согласованность (Cohesion)

Согласованность характеризует степень внутренней связанности функциональных элементов модуля. Высокая согласованность подразумевает, что все компоненты модуля ориентированы на выполнение одной конкретной задачи или функции, что улучшает читаемость и управляемость кода.

Типы согласованности включают:

- **Functional cohesion** (функциональная согласованность): модуль объединяет весь необходимый функционал для выполнения одной задачи.
- **Sequential cohesion** (последовательная согласованность): результаты одной функции используются в качестве входных данных для другой.
- **Communication cohesion** (коммуникационная согласованность): все элементы модуля работают с одними и теми же данными.
- **Procedural cohesion** (процедурная согласованность): элементы модуля выполняются в определенной последовательности, необходимой для достижения результата.
- **Temporal cohesion** (временная согласованность): функции модуля сгруппированы на основе их выполнения в определенный момент времени (например, при обработке ошибок).
- **Logical cohesion** (логическая согласованность): элементы модуля объединены логически общей функцией, но могут отличаться по своей природе (например, обработ-

ка различных типов ввода).

- **Coincidental cohesion** (случайная согласованность): элементы модуля не связаны логически, что делает модуль хаотичным и сложным для понимания.

Взаимосвязь между Coupling и Cohesion

Связанность и согласованность находятся в сложной взаимозависимости. Уменьшение связанности обычно способствует увеличению согласованности и наоборот. Например, избыточная согласованность может привести к росту зависимости между модулями, что увеличивает связанность. Таким образом, ключевым аспектом является нахождение оптимального баланса, который обеспечит поддержку высокого уровня согласованности при минимально возможной связанности.

Примеры применения

Рассмотрим два варианта компоновки модулей:

- Низкая согласованность:

```
./internal
   ./model
      product.go
      user.go
   ./factory
      product_factory.go
      user_factory.go
   ./repository
      product_repository.go
      user_repository.go
```

  В данном случае функционал рассредоточен, что усложняет понимание структуры приложения и взаимодействия его компонентов.

- Высокая согласованность:

```
./internal
   ./product
      product.go
      product_factory.go
```

```
    product_repository.go
./user
    user.go
    user_factory.go
    user_repository.go
```

Здесь модули структурированы по функциональным областям, что повышает согласованность и упрощает работу с кодом.

Применение принципов низкой связанности и высокой согласованности позволяет создавать поддерживаемые, масштабируемые и понятные программные системы. Их использование способствует не только повышению качества кода, но и ускорению разработки, что особенно важно в современных условиях высокой конкуренции и быстрого изменения требований.

1.2.1 Луковая архитектура

Луковая архитектура была предложена Джеффри Палермо в 2008 году как расширение гексагональной архитектуры. Её цель — облегчить поддержку приложений за счёт разделения аспектов и строгих правил взаимодействия слоёв.

Основная идея

Луковая архитектура предполагает многослойную организацию системы, где каждый слой зависит только от внутреннего слоя. Центральным слоем всегда является независимый слой, который представляет собой сердцевину архитектуры. Остальные слои располагаются вокруг него, создавая структуру, похожую на лук.

Ключевые принципы:

- Приложение строится вокруг независимой объектной модели.
- Внутренние слои определяют интерфейсы, внешние — реализуют их.
- Зависимости направлены к центру.
- Код предметной области изолирован от инфраструктуры.

Слои

Domain Model: содержит основные сущности и поведение предметной области, например, валидацию данных.

Domain Services: реализует бизнес-логику, которая не связана с конкретными сущностями, например, расчёт стоимости заказа.

Application Services: оркестрирует бизнес-логику, например, обработку запроса на создание заказа.

Infrastructure: реализует интерфейсы и адаптеры для работы с базами данных, внешними API и другими ресурсами.

Преимущества

- Независимость предметной области от инфраструктуры.
- Гибкость замены внешних слоёв.
- Высокая тестируемость.

Недостатки

- Сложность обучения.
- Необходимость создания множества интерфейсов.

1.2.2 Hexagonal Architecture

Hexagonal Architecture (также известная как Ports and Adapters) была предложена Alistair Cockburn. Её цель — устранить зависимость бизнес-логики от внешних систем и сделать приложение более гибким и тестируемым.

Основная идея

Hexagonal Architecture представляет приложение в виде шестиугольника, где каждая сторона представляет интерфейс (порт) для взаимодействия с внешними системами (адаптерами). Центр архитектуры — это бизнес-логика, изолированная от деталей реализации.

Ключевые принципы:

- Бизнес-логика полностью изолирована от инфраструктуры.
- Взаимодействие с внешним миром происходит через порты и адаптеры.
- Внешние системы подключаются через адаптеры, соответствующие определённым портам.

Слои

Core (Business Logic): центральная часть архитектуры, содержащая доменные сущности и бизнес-правила.

Ports: интерфейсы, через которые осуществляется взаимодействие между Core и внешними системами.

Adapters: реализация портов для подключения конкретных технологий, таких как базы данных, веб-сервисы и т.д.

Преимущества

- Гибкость: легко заменять внешние системы, не затрагивая бизнес-логику.
- Тестируемость: изолированная бизнес-логика упрощает написание модульных тестов.
- Чёткое разделение обязанностей.

Недостатки

- Сложность проектирования и реализации.
- Дополнительные накладные расходы на создание портов и адаптеров.

1.2.3 Чистая архитектура

Чистая архитектура, предложенная Робертом Мартином, основывается на идеях Onion и Hexagonal архитектур. Её ключевая цель — изолировать бизнес-логику от деталей реализации.

Базовые принципы

- Использование принципов SOLID.
- Разделение системы на слои.
- Независимость бизнес-логики от инфраструктуры.
- Тестируемость бизнес-логики без внешних зависимостей.

Слои

Domain: содержит общие бизнес-правила, структуры и интерфейсы.

Application: реализует конкретные сценарии использования, например, обработку запросов.

Presentation: отвечает за взаимодействие с пользователем через REST API, CLI и т.д.

Infrastructure: содержит детали реализации, такие как доступ к базе данных или внешние сервисы.

Пересечение границ

В чистой архитектуре зависимости направлены внутрь. Внешние слои реализуют интерфейсы, определённые внутренними слоями. Это достигается за счёт принципа инверсии зависимостей (DIP).

Преимущества

- Независимость от фреймворков и инфраструктуры.
- Простота тестирования.
- Переносимость и возможность разделения на микросервисы.

Недостатки

- Требуется строгое разделение бизнес-правил.
- Возможность излишнего усложнения структуры.

Вывод

Чистая архитектура представляет собой эволюцию луковой и гексагональной архитектур, сохраняя их принципы, но делая больший акцент на SOLID и изоляции бизнес-логики. Она подходит для сложных систем, требующих высокой гибкости и тестируемости, но может быть избыточной для небольших приложений.

1.3 Анализ инструментов применяемых для реализации системы

1.3.1 Взаимодействие микросервисов

Взаимодействие между микросервисами имеет критическое значение для обеспечения их эффективной работы и достижения бизнес-целей. Существует два основных подхода к взаимодействию: синхронное и асинхронное, каждый из которых служит своим целям и задачам.

Синхронное взаимодействие необходимо в ситуациях, когда требуется немедленная обратная связь. Например, когда клиент получает данные из базы данных или выполняет операции, которые требуют актуальной информации в реальном времени. В таких сценариях задержка в получении ответа может нарушить логическую последовательность

действий, и важно обеспечить, чтобы данные были актуальны на момент выполнения запроса.

Асинхронное взаимодействие, в свою очередь, используется в случаях, когда немедленный ответ не требуется. Это позволяет микросервисам отправлять запросы и продолжать выполнять свои задачи, не дожидаясь ответа. Например, в случаях обработки фоновых задач или выполнения операций, которые могут быть выполнены позже, асинхронные подходы позволяют разгружать сервисы и поддерживать их работу без блокировок. Это особенно выгодно в сценариях с высокой нагрузкой, когда множество запросов могут приходить одновременно.

Синхронное взаимодействие

Микросервисы могут синхронно общаться между собой с использованием различных технологий и протоколов. В этом контексте рассмотрим четыре популярных метода взаимодействия: HTTP/REST API, gRPC, GraphQL и WebSocket.

В большинстве случаев для микросервисов, которые обмениваются фиксированными данными, использование GraphQL может быть излишним, поскольку этот подход предназначен для динамического запроса данных. В сценариях, где структуры данных заранее известны и не требуют изменения, REST API будет более простым и понятным решением. То же касается и WebSocket: двунаправленный стриминг данных может быть избыточным для многих бизнес-приложений, где достаточно стандартного запроса и ответа.

Таким образом, основные методы, которые стоит рассмотреть для синхронного взаимодействия микросервисов, — это HTTP/REST API и gRPC.

HTTP/REST API HTTP/REST API — это один из самых распространенных способов синхронного взаимодействия между микросервисами. Каждый микросервис предоставляет набор эндпоинтов, к которым другие сервисы могут обращаться для выполнения операций и получения данных. Используя стандартные методы HTTP (GET, POST, PUT, DELETE), микросервисы обмениваются сообщениями с четко определенными правилами и структурой.

Преимущества использования HTTP/REST

- **Простота**: REST API легко реализовать и документировать. Он основан на понятных стандартах HTTP и может использоваться практически на любой платформе.
- **Читаемость**: Структура URL и использование HTTP-методов делают интерфейс API

интуитивно понятным и удобным для работы разработчиков.

- **Совместимость**: REST API может легко взаимодействовать с разными языками программирования и платформами, что делает его универсальным решением для микросервисной архитектуры.

gRPC gRPC, разработанный Google, представляет собой высокопроизводительный фреймворк для удаленных вызовов процедур (RPC), который поддерживает множество языков программирования. Он использует HTTP/2 для передачи данных, что обеспечивает преимущества, такие как многопоточность и меньшие задержки при передаче информации.

Преимущества использования gRPC

- **Производительность**: Использование HTTP/2 позволяет gRPC эффективно обрабатывать множество параллельных запросов, что делает его более производительным, чем традиционные REST API.
- **Статическая типизация**: gRPC использует Protocol Buffers для описания структуры данных, что позволяет разработчикам строго определять, какой тип данных будет передаваться. Это обеспечивает дополнительную безопасность и позволяет избежать ошибок при взаимодействии между сервисами.
- **Автогенерация кода**: Удобство разработки достигается благодаря автоматической генерации клиентского кода на разных языках, что ускоряет процесс создания микросервисов.

Сравнение HTTP/REST API и gRPC

Характеристика	**HTTP/REST API**
Протокол	Основан на HTTP/1.1 или HTTP/2
Структура данных	Ограничена JSON/XML
Типизация	Нестатическая типизация (JSON)
Производительность	Может иметь более высокую задержку
Поддержка потоковой передачи	Ограниченная
Автогенерация клиента	Обычно требует ручного создания клиентского кода
Простота и читаемость	Прост в реализации, легко понимается
Универсальность	Широко используется и поддерживается, совместим с многим
Тестирование и отладка	Легче тестировать с инструментами для работы с HTTP (напр

Таблица 1.2 – Сравнение HTTP/REST API и gRPC

Заключение Нашей системе не требуется возможность динамической типизации, поэтому выбор сделать в сторону gRPC.

Асинхронное взаимодействие

Асинхронное взаимодействие между микросервисами позволяет увеличить производительность и гибкость распределенных систем. В этом подходе микросервисы могут обмениваться данными без необходимости дожидаться ответа, что снижает задержки и повышает общую эффективность системы. Рассмотрим основные методы асинхронного взаимодействия.

Основные методы асинхронного взаимодействия

- **Очереди сообщений**: Использование систем обмена сообщениями, таких как RabbitMQ, Apache Kafka или Redpanda, позволяет отправлять сообщения между микросервисами без прямого связывания. Один сервис может отправлять сообщения в очередь, а другой — извлекать их и обрабатывать по мере возможности. Это гарантирует, что сервисы могут работать независимо друг от друга и не блокируют друг друга в случае высокой нагрузки.
- **Событийно-ориентированная архитектура**: В этой архитектуре микросервисы реагируют на события, происходящие в системе. События могут генерироваться различными компонентами и служить сигналами для других микросервисов о том, что произошло что-то важное (например, изменение состояния, завершение задачи и т. д.). Это позволяет строить более гибкие и масштабируемые системы.
- **HTTP-события (Webhooks)**: Использование вебхуков позволяет микросервису отправлять HTTP-запросы в другие сервисы при наступлении определённых событий. Это простой способ интеграции, позволяющий уведомлять другие службы о произошедших изменениях или событиях.

Выбор в сторону очередей сообщений также обуславливается необходимостью наличия механизма Dead Letter Queue (DLQ), который позволяет обрабатывать ошибки при отправке и получении сообщений. Кроме того, мы стремимся автоматизировать хранение сообщений, что делает использование очередей сообщений предпочтительным решением.

Описание систем очередей сообщений

- **Apache Kafka**: - Kafka — это распределенная платформа для потоковой передачи данных, которая обеспечивает высокую пропускную способность и низкую задержку. Она работает по принципу публикации и подписки, позволяя множеству клиентов читать и писать сообщения. - **Назначение**: Идеально подходит для обработки потоков данных в реальном времени и хранения больших объемов событий с воз-

можностью их долговременного хранения.

- **Redpanda**: - Redpanda — это высокопроизводительная, совместимая с Kafka, распределенная система сообщений, оптимизированная для работы с потоками данных в реальном времени. - **Назначение**: Обеспечивает низкую задержку и простоту настройки, что делает её предпочтительной для решений, требующих высокой производительности.
- **Apache Pulsar**: - Pulsar — это распределенная система потоковой передачи данных, которая поддерживает многопоточность и управление потоками. Она предоставляет гибкий подход к очередям сообщений и событиям. - **Назначение**: Отличается поддержкой долгосрочного хранения сообщений и сложных сценариев работы с георасределёнными данными.

Сравнение систем очередей сообщений Ниже представлено сравнение преимуществ и недостатков Apache Kafka, Redpanda и Apache Pulsar.

Система	Преимущества	Недостатки
Apache Kafka	- Высокая надежность	- Сложность настройки и администр
	- Масштабируемость	- Требует дополнительных ресурсов
	- Долговременное хранение данных	- Зависимость от Zookeeper
Redpanda	- Низкая задержка	- Меньшая экосистема поддержки
	- Простота настройки	
	- Совместимость с Kafka	
Apache Pulsar	- Гибкость	- Сложность архитектуры и настрой
	- Поддержка георасределённых данных	- Меньшая популярность по сравнен
	- Высокая масштабируемость	

Таблица 1.3 – Сравнение систем очередей сообщений

Заключение Асинхронное взаимодействие, организованное через очереди сообщений, является эффективным способом повышения производительности и устойчивости микросервисов. Учитывая возможности по снижению задержки и простой настройке, мы выбрали Redpanda как предпочтительное решение для организации асинхронного взаимодействия между микросервисами.

1.3.2 Хранение данных

Реляционные базы данных (RDBMS): Реляционные базы данных, такие как MySQL и PostgreSQL, предлагают структурированный способ хранения данных с использованием таблиц, что позволяет легко реализовать связи между ними. Они подходят для большинства приложений, требующих согласованности и целостности данных.

Описание реляционных баз данных

1. **MySQL**: - **Описание**: MySQL — это популярная реляционная база данных с открытым исходным кодом, которая используется во множестве веб-приложений и сервисов. Она известна своей производительностью, простотой настройки и широким сообществом. - **Назначение**: Широко используется для веб-приложений, особенно тех, которые требуют быстрого доступа к данным.

2. **PostgreSQL**: - **Описание**: PostgreSQL — это мощная объектно-реляционная база данных с открытым исходным кодом, которая поддерживает различные типы данных и расширенные функции, такие как работа с JSON и поиск по полнотекстовому содержимому. - **Назначение**: Подходит для сложных и крупных приложений, требующих расширенной функциональности обработки данных.

Сравнение MySQL и PostgreSQL Ниже представлено сравнение преимуществ и недостатков MySQL и PostgreSQL.

Система	Преимущества	Недостатки
MySQL	- Высокая производительность	- Ограниченная поддержка сло
	- Простота настройки	- Менее гибкая работа с типами
	- Широкое сообщество и поддержка	- Нет полной поддержки ACID
PostgreSQL	- Поддержка сложных запросов и индексов	- Меньшая производительность
	- Расширенные функции для работы с данными	- Сложнее в настройке и админ
	- Полная поддержка ACID и транзакций	

Таблица 1.4 – Сравнение MySQL и PostgreSQL

Заключение

Учитывая наши требования и специфику приложений, мы выбираем PostgreSQL как основное решение для хранения данных в нашей архитектуре.

1.3.3 Авторизация

В современном веб-разработке авторизация пользователей является важной частью обеспечения безопасности и управления доступом к ресурсам. Существует несколько подходов к авторизации, каждый из которых имеет свои особенности, преимущества и недостатки. Два самых популярных метода авторизации — это использование сессий и JSON Web Tokens (JWT). В этом обзоре мы рассмотрим основные аспекты каждого из этих подходов.

Подход 1: Авторизация с использованием сессий Авторизация с использованием сессий предполагает классический подход к управлению состоянием пользователя. Когда пользователь выполняет вход в систему, сервер создает сессию и сохраняет информа-

цию о пользователе, а также уникальный идентификатор сессии (обычно в виде куки) в браузере.

Преимущества

- **Безопасность**: Сессии могут быть безопаснее, так как данные о пользователе хранятся на сервере, и доступ к ним ограничен.
- **Управление сессиями**: Сервер может контролировать время жизни сессии и управлять активными сессиями (например, отключать их по запросу пользователя).
- **Простота реализации**: Для простых приложений реализация сессий может быть проще и привычнее для разработчиков.

Недостатки

- **Сложности с масштабированием**: В распределенных системах требуется дополнительная архитектура для хранения сессий (например, база данных или кэш), что может усложнить разработку.
- **Зависимость от состояния**: Поскольку сессии хранят состояние на сервере, это может ограничить возможность использования микросервисной архитектуры.

Подход 2: Авторизация с использованием JWT

JSON Web Tokens (JWT) — это подход к авторизации, основанный на использовании токенов. Когда пользователь входит в систему, сервер генерирует токен, который содержит зашифрованную информацию о пользователе и отправляет его клиенту. Клиент хранит этот токен и включает его в заголовки запросов при доступе к защищённым ресурсам.

Преимущества

- **Безсостояние**: JWT не требуют хранения состояния на сервере. Токен самостоятельно хранит информацию о пользователе, что упрощает масштабирование и работы в распределенных системах.
- **Безопасность**: Токены можно подписывать и шифровать, что добавляет уровень защиты данных.
- **Гибкость**: JWT можно использовать для разных типов приложений, включая веб-приложения, мобильные приложения и API.

Недостатки

- **Управление сроком действия**: JWT имеют фиксированный срок действия, и их нужно будет обновлять. Отзыв токена может быть сложнее реализовать.
- **Размер**: Токены могут быть крупнее, чем идентификаторы сессий, что может отрицательно сказаться на производительности при частом использовании в заголовках

запросов.

Заключение

JWT более удобный и простой в реализации из-за того что не требуется хранение состояния на сервере.

1.4 Выводы

1. Выполнен анализ архитектурных стилей построения систем, с учетом требований к динамической расширяемости отдельных модулей системы, выбрана микросервисная архитектура с применением API Gateway.
2. Проанализированы основные архитектурные паттерны построения приложений. Clean Architecture за счет преобраладания SOLID, разбиения на слои и преимуществ Dependency inversion выбрана как основа написания микросервисов.
3. Проанализированных основные инструменты для разработки системы, произведены сравнения аналогов

   - для синхронного взаимодействия между микросервисами выбран gRPC, за счет строгой типизации контрактов и производительности за счет сериализации данных.

   - для асинхронного взаимодействия между микросервисами выбран подход с брокером сообщений, за счет универсальности и отказоустойчивости. В качестве реализации выбрана RedPanda, Kafka совместимое API с наибольшей производительностью.

   - для хранения структурированных данных выбрана PostgreSql, за счет развитой надежности и удобства

   - для хранения неструтурированных данных выбрано NoSQL хранилище S3 Minio.

   - в качестве схемы авторизации выбрана схема с JWT токенами, которая позволяет не хранить активное состоянии на серверной стороне.

1.5 Постановка задачи научно иследовательской работы

1. Проектирование и разработка системы серверного приложения системы «Интеллектуальный ассистент врача УЗИ» на основе выбранных инструментов и подходов.
2. Тестирование и оценка разработанной системы и ее производительности. Устранение недостатков при наличии.

2. Теоретическая часть

2.1 Модель микросервиса

К каждому микросервису выделяются общие функциональные требования:

- При необходимости выполнять CRUD операции с базой данных
- При необходимости писать сообщения в логброкер сообщений
- При необходимости читать сообщения из логброкера сообщений

К каждому микросервису выделяются общие нефункциональные требования:

- Соответствовать чистой архитектуре
- Иметь высокий cohesion и низкие coupling

Исходя из этих требований, типовая архитектура нашего сервиса будет соответствовать приведенной.

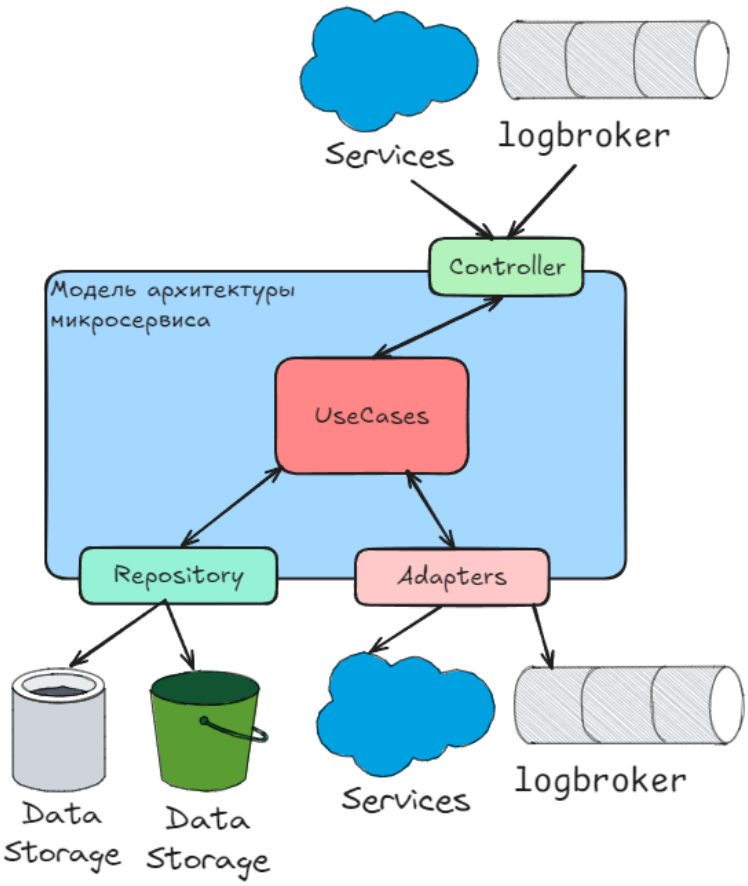


Рисунок 2.1 – Модель типового микросервиса

2.2 Модели модулей системы

2.2.1 Модель сервиса аутентификации

Микросервис аутентификации осуществляет регистрацию новых пользователей, и обновление токенов у уже существующих. Для сущесвующих пользователей он выдает новую пару JWT tokens(access token + refresh token) по refresh token или логину и паролю.

Функциональные требования к сервису аутентификации:

- Возможность регистрировать новых пользователей в сервисе
- Подписывать JWT ключи для авторизации
- Обменивать логин пароль или refresh JWT ключ на новый resfresh JWT ключ

Тогда для соответствия требованиям, модель базы данных:

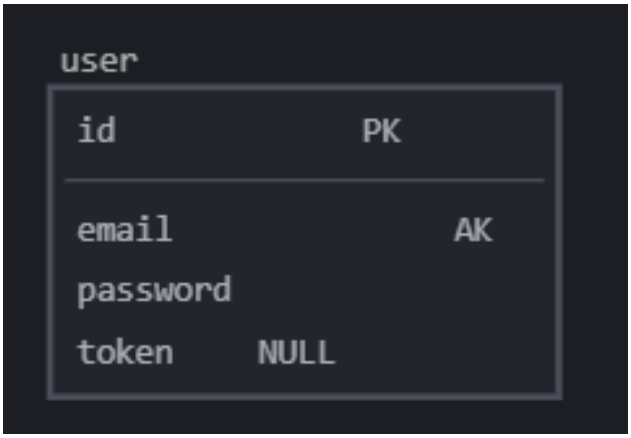


Рисунок 2.2 – Модель базы данных сервиса аутентификации

Для обмена refresh token на новую пару ключей, в токен необходимо зашивать ID пользователя. В соответствии с требованиями, модель сервиса авторизации:

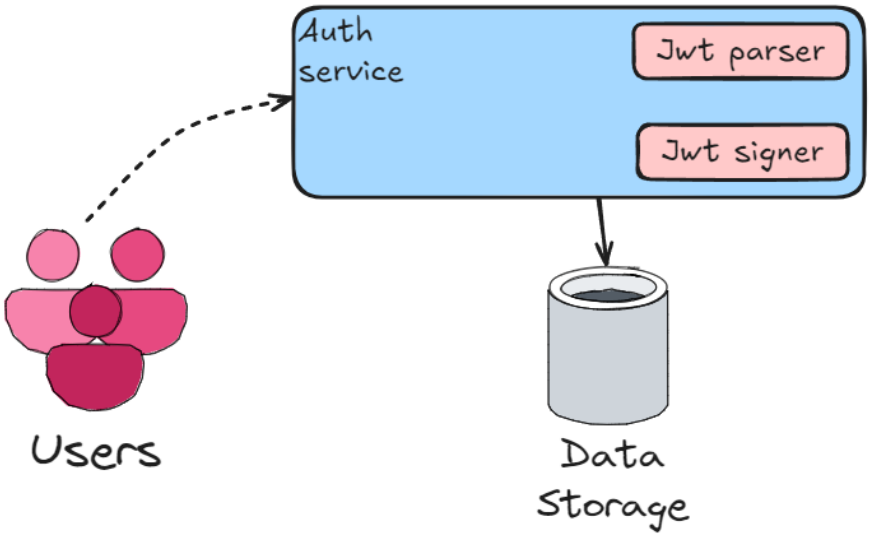


Рисунок 2.3 – Модель сервиса аутентификации

2.2.2 Модель сервиса узи

Для начала нужно определить вид данных которые должен хранить узи.

Узи представляет из себя последовательный набор кадров, конкретной щитовидной железы. На щитовидной железе могут быть обнаружены злокачественные образования, называемыми **узлами**. Узел злокачественного образования может быть запечатлен на нескольких кадрах, такое отображения узла на конкретном кадре называется **сегмент**ом, узлов может быть любое количество. Каждый узел и сегмент в отдельности характеризуется классификационными признаками. На данный момент имеется 3 классификацио-оных признака: вероятности принадлежности узлов к классам **TI-RADS**. TI-RADS, описывает стадию злокачественного образования, существует 3 стадии: TI-RADS23, TI-RADS4, TI-RADS5.

Наглядный пример:

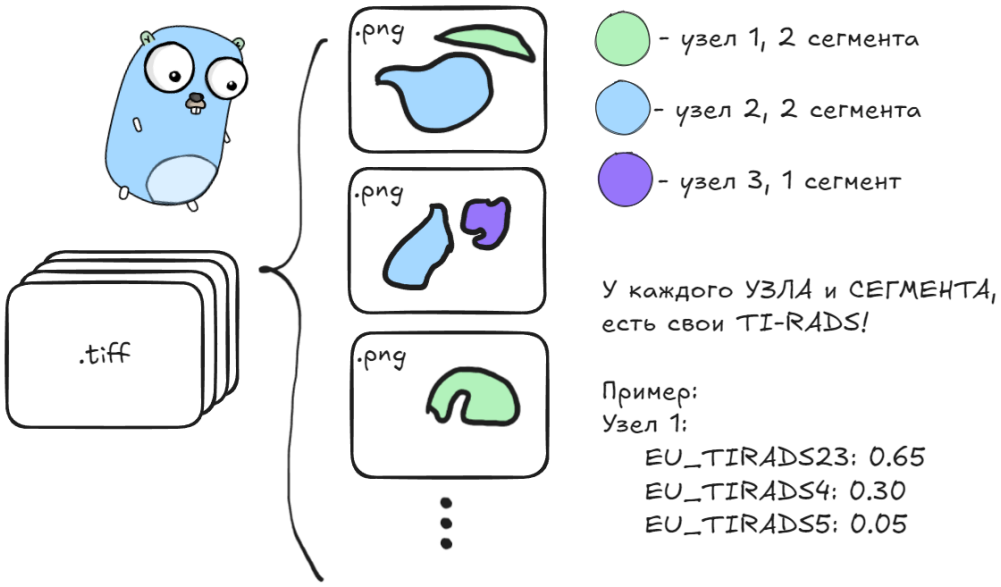


Рисунок 2.4 – Предметная область узлов и сегментов

Функциональные требования к сервису узи:

- Принимать единуб композицию кадров узи, разбивать и сохранять ее по кадрам.
- После разбиения узи по кадрам, оповещать через брокер сообщений, сервис с ml моделью, о доступности обработки изображения
- Предоставлять возможность совершать CRUD операции над узлами, сегментами, узи, ti-rads, images.
- Сохранять результат сегментации и классификации нейронной моделью посредством логброкера сообщений.

Тогда для соответствия требованиям, модель базы данных:

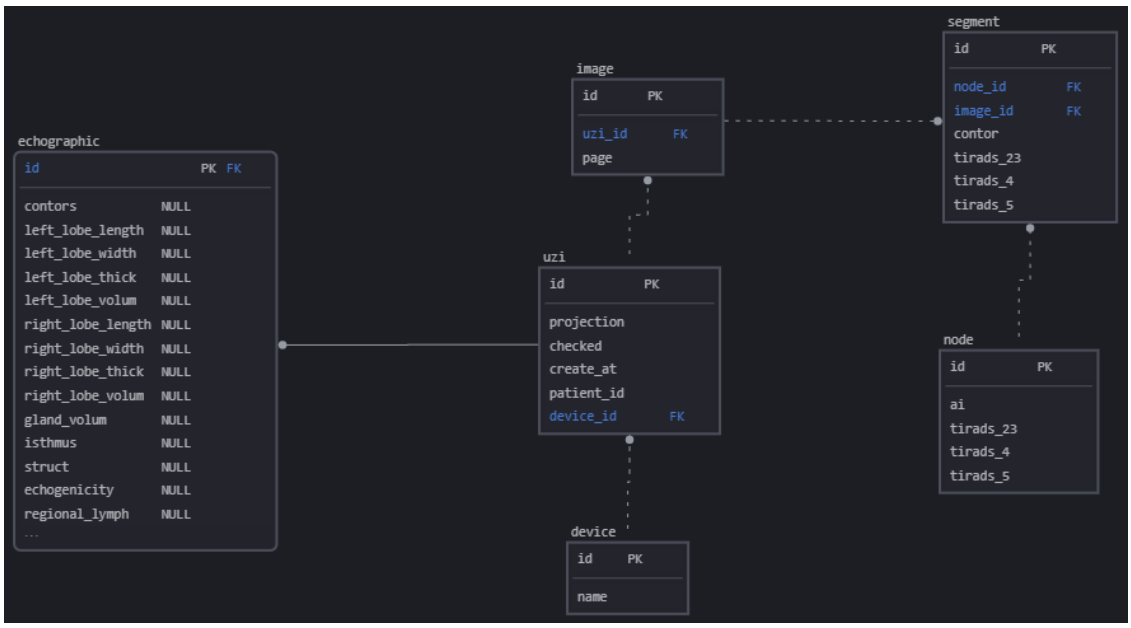


Рисунок 2.5 – Модель базы данных сервиса аутентификации

В соответствии с требованиями, модель сервиса узи:

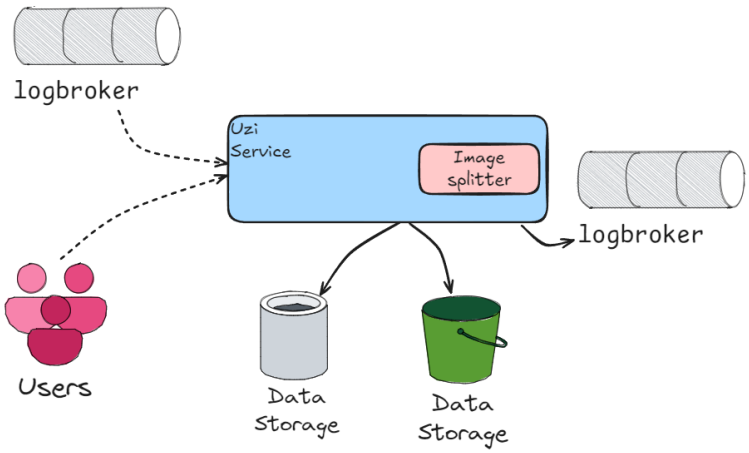


Рисунок 2.6 – Модель сервиса УЗИ

2.2.3 Модель Gateway API

Сервис gateway API, является точной входа для всех пользователей системы. Сервис делает запросы в дочерние микросервисы.

Функциональные требования к сервису Gateway API:

- Если дочерний микросервис реализует публичный для пользователя контракт взаимодействия, то сервис должен реализовывать доступ к этому контракту для пользователя
- Предоставлять пользователю по запросу кадры узи

– Запускать пайплайн обработки узи, посредством сообщения в логброкер

В соответствии с требованиями, модель сервиса Gateway:

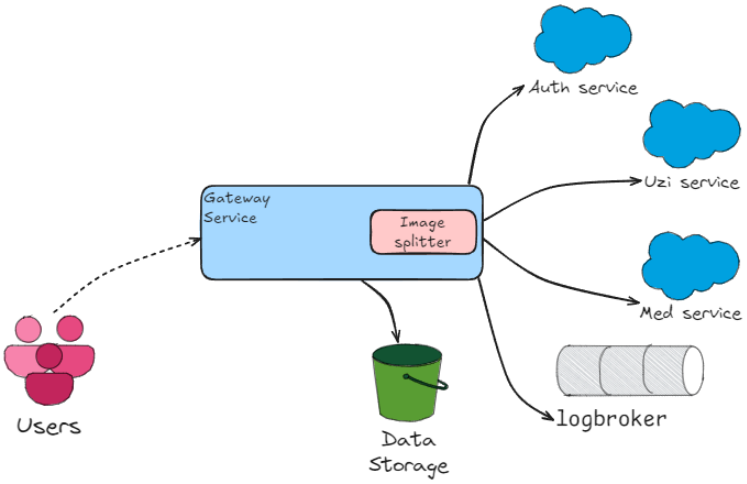


Рисунок 2.7 – Модель сервиса gateway

2.2.4 Модель системы

Общая модель всей системы:

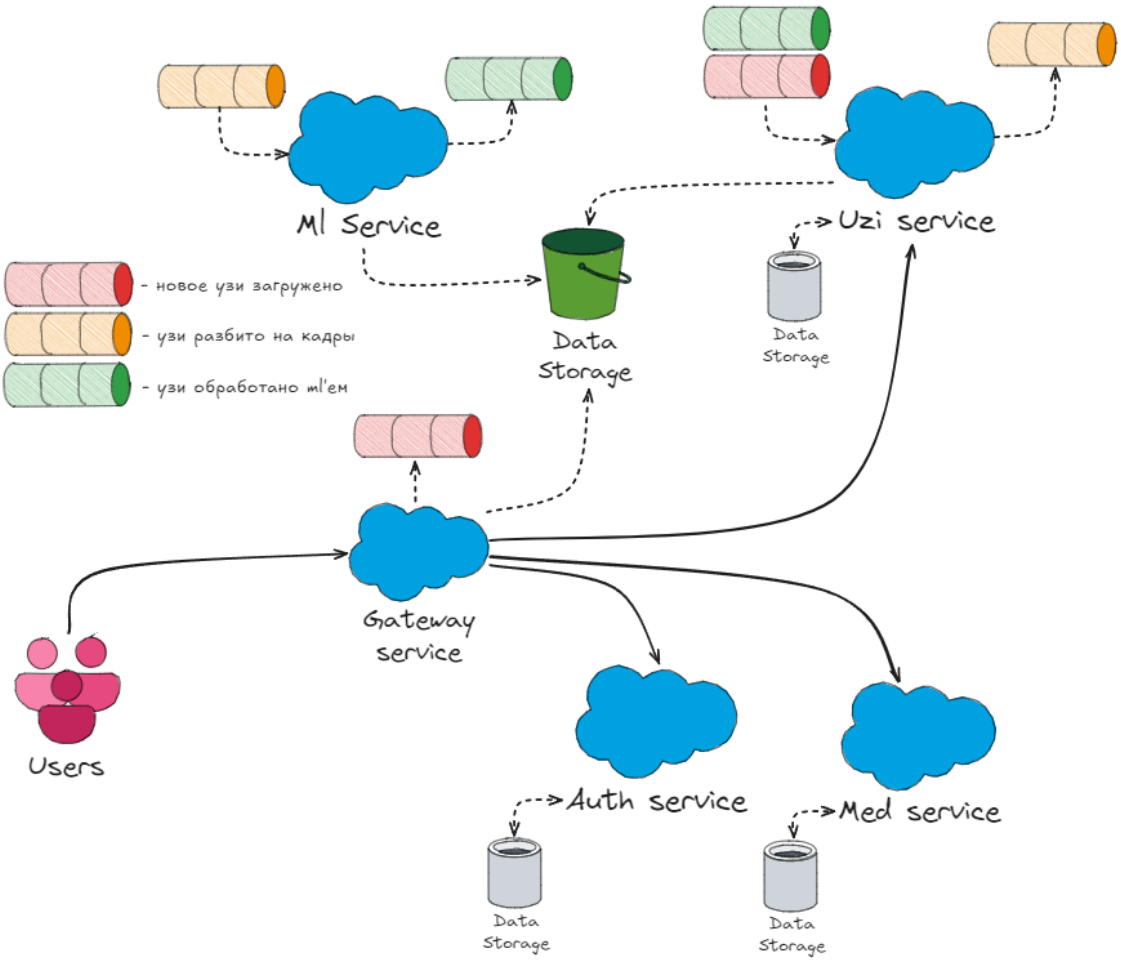


Рисунок 2.8 – Модель системы

2.2.5 Выводы

Были спроектированы модели Auth Service, Uzi Service и Gateway Service. Объединением всех этих моделей является модель самой системы. Модели соответствуют и учитывают все функциональные требования к модулям системы и системе в целом.

3. Результаты проектирования

Для хранения узи снимков и кадров используется S3 minio. Структура бакета minio:

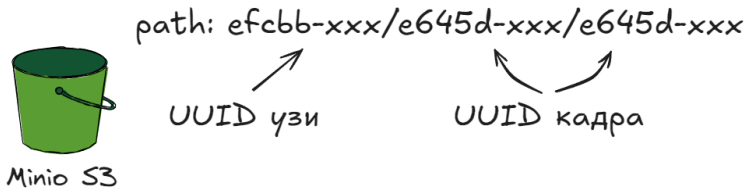


Рисунок 3.1 – Структура S3 хранилища

Для использования асинхронного общения через RedPanda, создадим 3 топика:

- **uziupload** - узи загруженно в S3. Запустит разбиение узи на кадры. А.1
- **uzisplitted** - узи разбито на кадры. Запустит обработку узи нейро моделью. В топик пишет Uzi Service после разбиения узи на кадры. А.2
- **uziprocessed** - узи обработанно нейромоделью. Узи сегментировано и классифицировано А.3

3.0.1 Auth Service

Рисунок 3.2 – Спроектированная база данных Auth Service

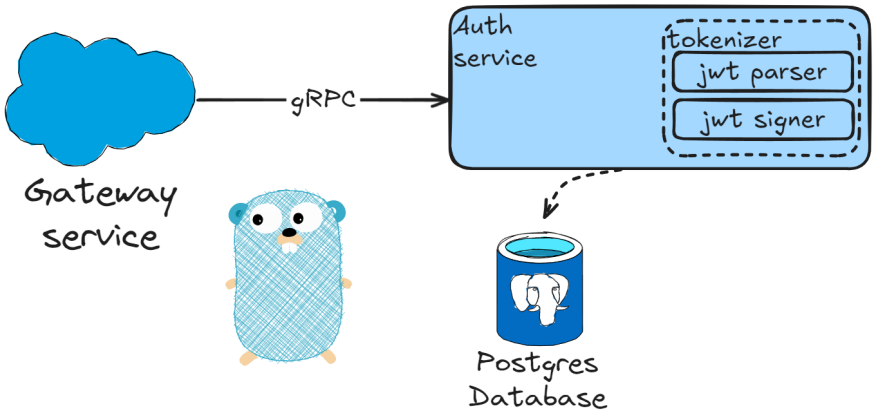


Рисунок 3.3 – Auth Service

3.0.2 Uzi Service

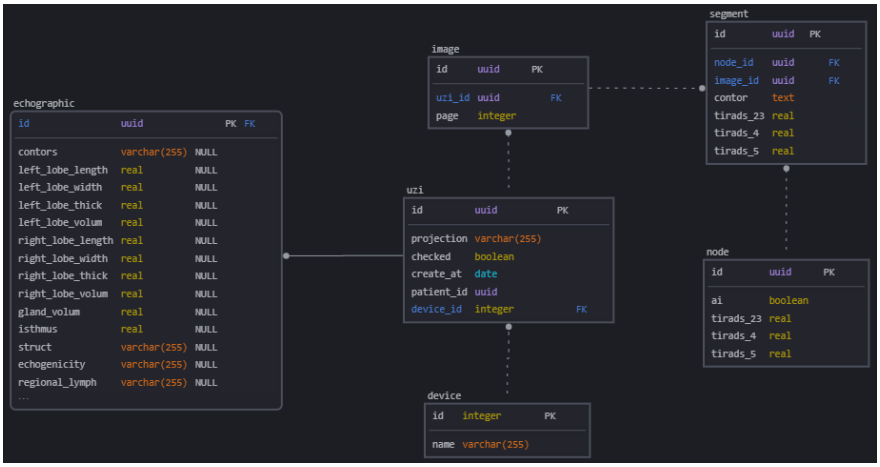


Рисунок 3.4 – Спроектированная база данных Uzi Service

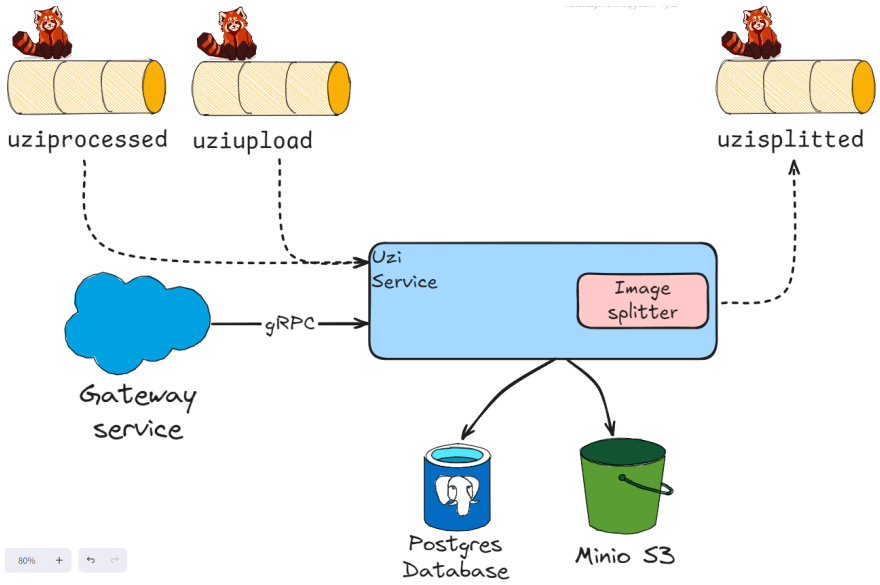


Рисунок 3.5 – Uzi Service

3.0.3 Gateway API

Распределенные транзакции В следствии выбранной микросервисной архитектуры, теряется возможнная атомарность операций, достигаемая за счет транзакций. Решение этой проблемы реализовано в качестве паттерна Saga. С потерей eventual consistency, мы получаем возможнсть производить операцию в транзакции ”поэтапно” с возможнстью откатить все изменения.

Разберем на примере регистрации нового врача:

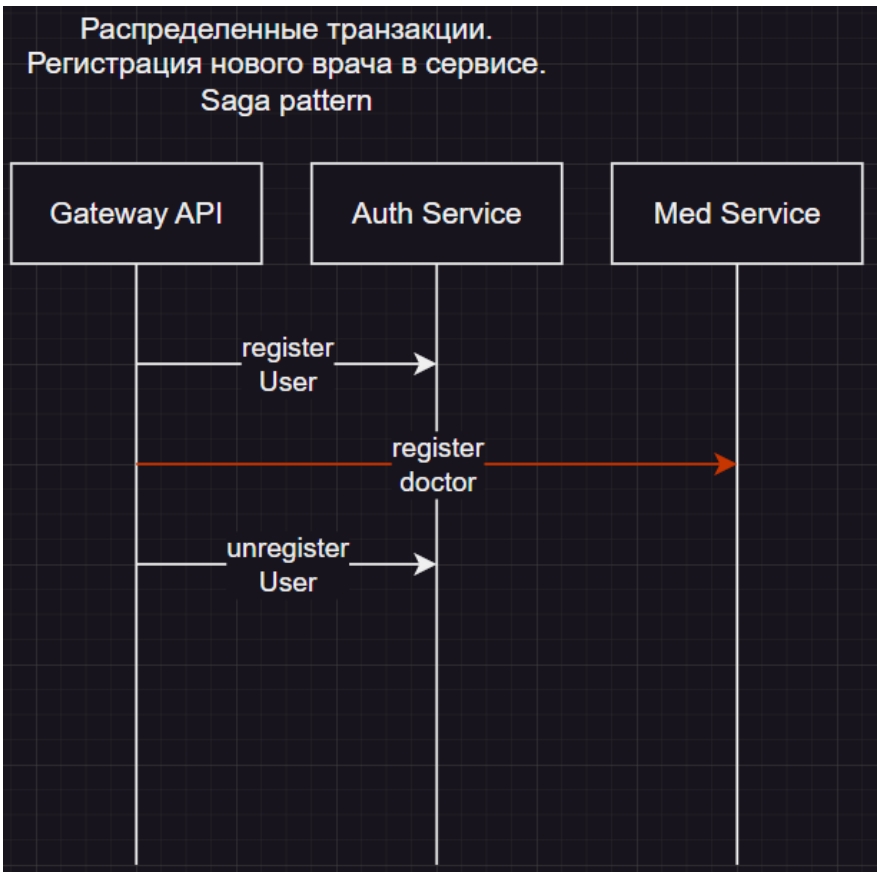


Рисунок 3.6 – Saga pattern

В следующем семестре будет реализованны паттеры transactional outbox и circuit breaker.

3.1 Выводы

В результате были спроектированы модули системы и правила их взаимодействия с учетом выбранных средств разработки на этапе анализа.

4. Реализация

4.1 Состав и структура реализованного программного обеспечения

Реализованно серверное приложение, отвечающие на запросы по HTTP.

4.1.1 Структура приложения

Приложение состоит из 5 микросервисов:

- Gateway API
- Auth Service
- Uzi Service
- Med Service
- Ml Service

Каждый микросервис написанный на golang содержит:

- go.mod и go.sum файлы, описывающие библиотекчные зависимости для приложения
- sql файлы миграций. Сами миграции осуществлялись за счет Goose
- md файлы с описанием технических требований к предоставляемому API сервиса
- proto файлы описывающие контракты для взаимодествия с внешними системами
- taskfile - аналог makefile, файл для локальной сборки и запуска микросервиса
- Dockerfile - файл для сборки docker образа микросервиса
- service.yml - файл содержащий конфигурацию сервиса

4.2 Основные сценарии работы пользователя

Основной сценарий работы ПО - отправка запросов на обработку узи изображений и полученние данных с обработанных изображений. Для взаимодействием с сервисом предоставляется подробное swagger api.

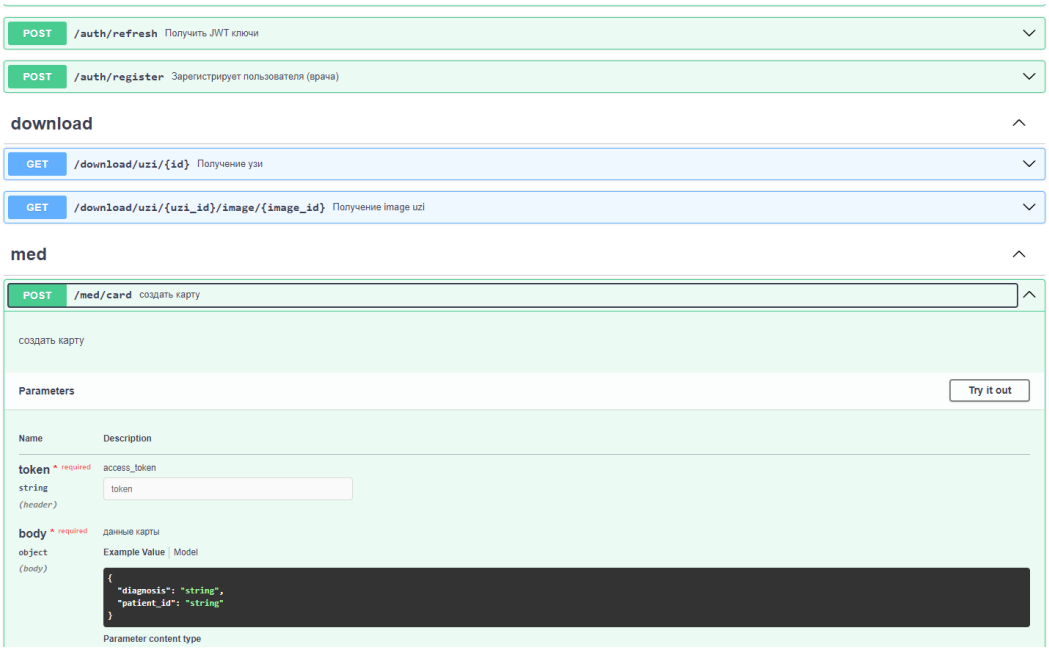


Рисунок 4.1 – Спроектированная база данных Auth Service

Подробнее в листинге.

4.3 Выводы

Таким образом, в рамках НИРа было реализовано 4 микросервиса серверного приложения, а также были проведены некоторые типы тестирования реализации. В будущем планируется развернуть систему на нескольких физическим машинах с использованием оркестратора контейнеризированных приложений kubernetus и повторно провести нагрузочное и интеграционное тестирование.

Заключение

В аналитической части:

1. Выполнен анализ архитектурных стилей построения систем, с учетом требований к динамической расширяемости отдельных модулей системы, выбрана микросервисная архитектура с применением API Gateway.
2. Проанализированы основные архитектурные паттерны построения приложений. Clean Architecture за счет преобраладания SOLID, разбиения на слои и преимуществ Dependency inversion выбрана как основа написания микросервисов.
3. Проанализированных основные инструменты для разработки системы, произведены сравнения аналогов

   - для синхронного взаимодействия между микросервисами выбран gRPC, за счет строгой типизации контрактов и производительности за счет сериализации данных.

   - для асинхронного взаимодействия между микросервисами выбран подход с брокером сообщений, за счет универсальности и отказоустойчивости. В качестве реализации выбрана RedPanda, Kafka совместимое API с наибольшей производительностью.

   - для хранения структурированных данных выбрана PostgreSql, за счет развитой надежности и удобства

   - для хранения неструтурированных данных выбрано NoSQL хранилище S3 Minio.

   - в качестве схемы авторизации выбрана схема с JWT токенами, которая позволяет не хранить активное состоянии на серверной стороне.

В теоретической части:

Были спроектированы модели Auth Service, Uzi Service и Gateway Service. Объединением всех этих моделей является модель самой системы. Модели соответствуют и учитывают все функциональные требования к модулям системы и системе в целом.

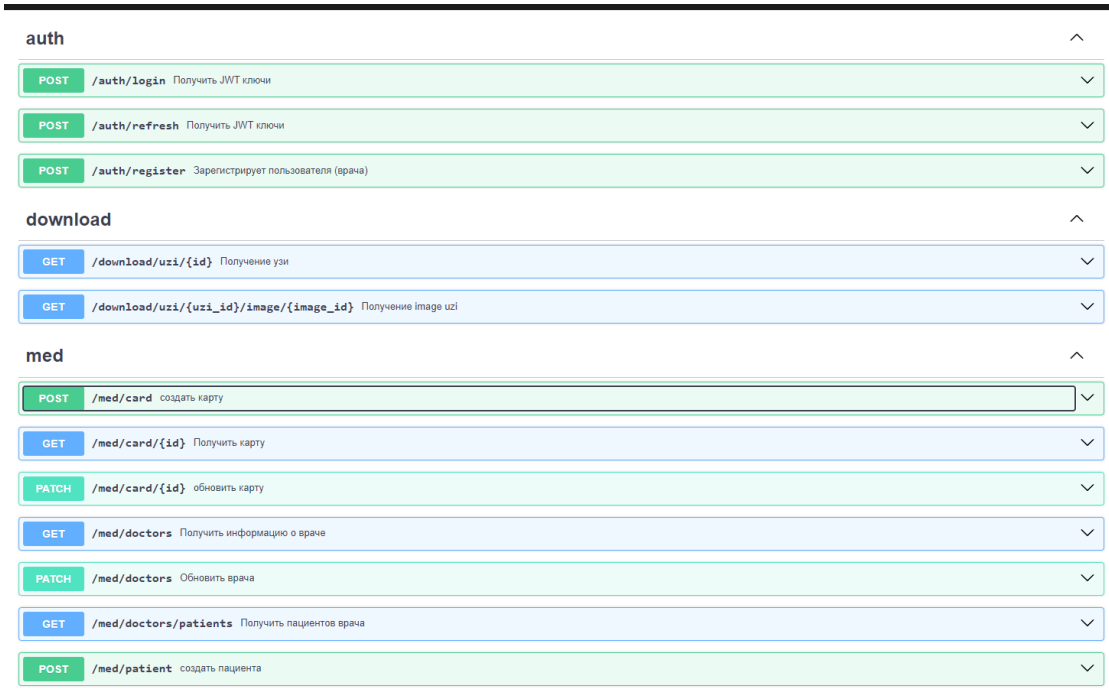
В проектировании:

В результате были спроектированы модули системы и правила их взаимодействия с учетом выбранных средств разработки на этапе анализа.

В итоге:

Таким образом, в рамках НИРа было реализовано 4 микросервиса серверного приложения, а также были проведены некоторые типы тестирования реализации. В будущем планируется развернуть систему на нескольких физическим машинах с использованием оркестратора контейнеризированных приложений kubernetus и повторно провести нагрузочное и интеграционное тестирование.

A. Приложение 1



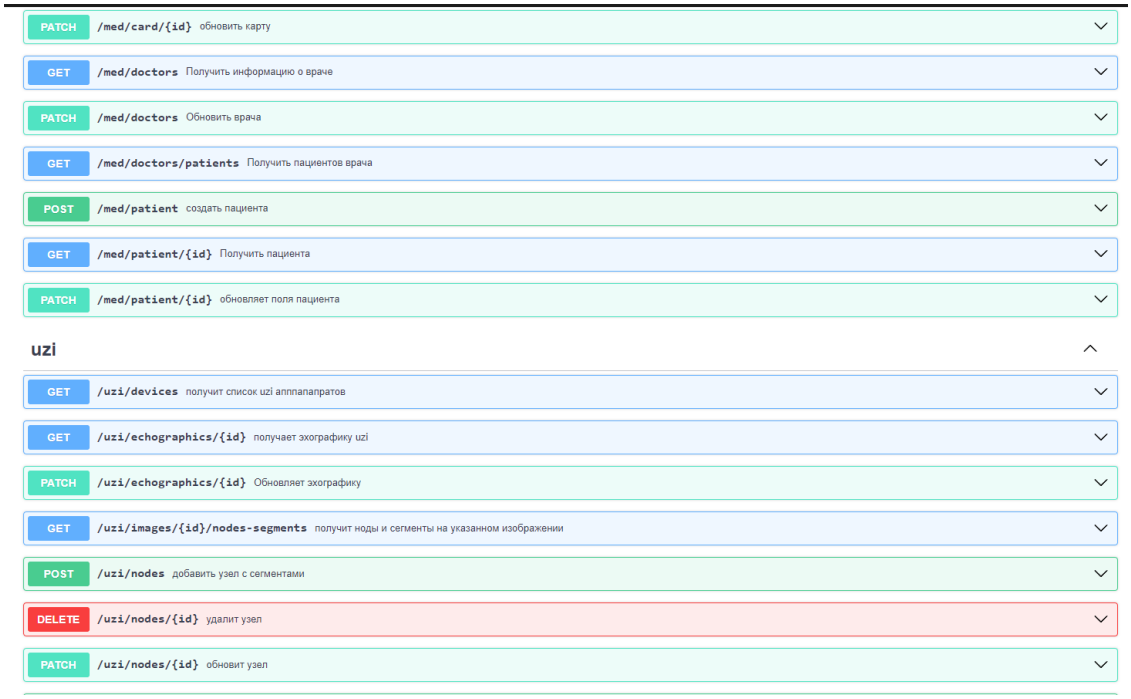
,

Листинг A.1 – Листинг топика `uziupload`

```
syntax = "proto3";

option go_package = "internal/generated/broker/produce/uziupload";

message UziUpload {
  string uzi_id = 100;
}
```

Листинг А.2 – Листинг топика `uzisplitted`

```
syntax = "proto3";

option go_package = "internal/generated/broker/produce/uzisplitted";

message UziSplitted {
    string uzi_id = 100;
    repeated string pages_id = 200;
  }
```



Листинг А.3 – Листинг топика `uziprocessed`

```
syntax = "proto3";

option go_package = "internal/generated/broker/consume/uziprocessed";

message UziProcessed {
    message Node {
        string id = 100;
        double tirads_23 = 400;
        double tirads_4 = 500;
        double tirads_5 = 600;
    }

    message Segment {
        string id = 100;
        string node_id = 200;
        string image_id = 300;
        string contor = 400;
        double tirads_23 = 500;
        double tirads_4 = 600;
        double tirads_5 = 700;
    }

    repeated Node nodes = 100;
    repeated Segment segments = 200;
}
```

Листинг А.4 – Листинг sql миграции Auth сервиса

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE "user"
(
    id       uuid         PRIMARY KEY,
    email    varchar(255) NOT NULL UNIQUE,
    password varchar(255) NOT NULL,
    token    text
);

COMMENT ON TABLE "user" IS     '        ';
COMMENT ON COLUMN "user".token IS 'Refresh           ';
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
DROP TABLE IF EXISTS "user" CASCADE;
-- +goose StatementEnd
```

Листинг А.5 – Листинг sql миграции uzi сервиса

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE device
(
    id   integer      PRIMARY KEY,
    name varchar(255) NOT NULL
);

COMMENT ON TABLE device IS      '                          ';
COMMENT ON COLUMN device.name IS     '     ';

CREATE TABLE uzi
(
    id         uuid         PRIMARY KEY,
    projection varchar(255) NOT NULL,
    checked    boolean      NOT NULL,
    create_at  date         NOT NULL,
    patient_id uuid         NOT NULL,
    device_id  integer      NOT NULL REFERENCES device (id)
);

COMMENT ON TABLE uzi IS      '                 ';
COMMENT ON COLUMN uzi.projection IS     '                ';
COMMENT ON COLUMN uzi.patient_id IS        '                    ';
COMMENT ON COLUMN uzi.device_id IS        '                   ';

CREATE TABLE image
(
    id     uuid PRIMARY KEY,
    uzi_id uuid    NOT NULL REFERENCES uzi (id),
    page   integer NOT NULL
);

COMMENT ON TABLE image IS      '        ';

CREATE TABLE node
(
    id        uuid    PRIMARY KEY,
    ai        boolean NOT NULL,
    tirads_23 real    NOT NULL,
    tirads_4  real    NOT NULL,
    tirads_5  real    NOT NULL
);

COMMENT ON TABLE node IS      '        ';
COMMENT ON COLUMN node.ai IS    '       (  )';
COMMENT ON COLUMN node.tirads_23 IS     '            tirads_23';
COMMENT ON COLUMN node.tirads_4 IS     '            tirads_4';
COMMENT ON COLUMN node.tirads_5 IS     '            tirads_5';

CREATE TABLE segment
(
    id        uuid PRIMARY KEY,
    node_id   uuid NOT NULL REFERENCES node (id),
    image_id  uuid NOT NULL REFERENCES image (id),
    contor    text NOT NULL,
    tirads_23 real NOT NULL,
    tirads_4  real NOT NULL,
    tirads_5  real NOT NULL
```

```
-- +goose Up
-- +goose StatementBegin
CREATE TABLE echographic
(
    id                uuid PRIMARY KEY,
    contors           varchar(255),
    left_lobe_length  real,
    left_lobe_width   real,
    left_lobe_thick   real,
    left_lobe_volum   real,
    right_lobe_length real,
    right_lobe_width  real,
    right_lobe_thick  real,
    right_lobe_volum  real,
    gland_volum       real,
    isthmus           real,
    struct            varchar(255),
    echogenicity      varchar(255),
    regional_lymph    varchar(255),
    vascularization   varchar(255),
    location          varchar(255),
    additional        varchar(255),
    conclusion        varchar(255)
);

COMMENT ON TABLE echographic IS         '     ';
COMMENT ON COLUMN echographic.id IS 'ID uzi';
COMMENT ON COLUMN echographic.contors IS     '';
COMMENT ON COLUMN echographic.left_lobe_length IS    '       ';
COMMENT ON COLUMN echographic.left_lobe_width IS    '       ';
COMMENT ON COLUMN echographic.left_lobe_thick IS    '        ';
COMMENT ON COLUMN echographic.left_lobe_volum IS    '       ';
COMMENT ON COLUMN echographic.right_lobe_length IS    '       ';
COMMENT ON COLUMN echographic.right_lobe_width IS    '       ';
COMMENT ON COLUMN echographic.right_lobe_thick IS    '        ';
COMMENT ON COLUMN echographic.right_lobe_volum IS    '       ';
COMMENT ON COLUMN echographic.gland_volum IS    '    ';
COMMENT ON COLUMN echographic.isthmus IS     '';
COMMENT ON COLUMN echographic.struct IS      '';
COMMENT ON COLUMN echographic.echogenicity IS       '';
COMMENT ON COLUMN echographic.regional_lymph IS        '           ';
COMMENT ON COLUMN echographic.vascularization IS        '     ';
COMMENT ON COLUMN echographic.location IS       '';
COMMENT ON COLUMN echographic.additional IS        '    ';
COMMENT ON COLUMN echographic.conclusion IS      '';

ALTER TABLE echographic
    ADD CONSTRAINT fk_echographic_uzi
    FOREIGN KEY (id)
    REFERENCES uzi (id);
-- +goose StatementEnd

-- +goose Down
-- +goose StatementBegin
ALTER TABLE echographic
    DROP CONSTRAINT fk_echographic_uzi;

DROP TABLE IF EXISTS echographic CASCADE;
-- +goose StatementEnd
```