



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА  
К КУРСОВОЙ РАБОТЕ  
НА ТЕМУ:**

**«Разработка системы извлечения  
многокомпонентных терминов и их переводных  
эквивалентов из параллельных научно-технических  
текстов»**

Студент      ИУ7-63Б

\_\_\_\_\_ Сапожков А. М.

Руководитель

\_\_\_\_\_ Строганов Ю. В.

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

УТВЕРЖДАЮ

Заведующий кафедрой ИУ7  
(Индекс)

И. В. Рудаков  
(И. О. Фамилия)

« \_\_\_\_ » \_\_\_\_\_ 20 \_\_\_\_ г.

## ЗАДАНИЕ на выполнение курсовой работы

по дисциплине Базы Данных

Студент группы ИУ7-63Б

Сапожков Андрей Максимович  
(Фамилия, имя, отчество)

Тема курсового проекта Разработка системы извлечения многокомпонентных терминов и их переводных эквивалентов из параллельных научно-технических текстов  
Направленность КР(учебная, исследовательская, практическая, производственная, др.)

учебная

Источник тематики (кафедра, предприятие, НИР) кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

### Задание:

Спроектировать и реализовать базу данных, содержащую информацию о терминах. Разработать приложение, предоставляющее интерфейс для загрузки текстов и выделения терминов, которые можно сохранять в базу данных, а также классифицировать и анализировать. Реализовать функциональность для разных категорий пользователей.

### Оформление курсового проекта:

Расчетно-пояснительная записка на 20-40 листах формата А4.

Расчетно-пояснительная записка должна содержать постановку задачи, введение, аналитическую часть, конструкторскую часть, технологическую часть, экспериментально-исследовательскую часть, заключение, список литературы, приложения.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)  
На защиту проекта должна быть предоставлена презентация, состоящая из 15-20 слайдов.  
На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, интерфейс, результаты проведенных исследований.

Дата выдачи задания « \_\_\_\_ » \_\_\_\_\_ 20 \_\_\_\_ г.

Руководитель курсовой работы

Ю. В. Строганов  
(И.О. Фамилия)

Студент

А М. Сапожков  
(И.О. Фамилия)

## **Реферат**

Расчетно-пояснительная записка 79 с., 22 рис., 3 табл., 41 ист., 7 прил.

**Ключевые слова:** научно-технические тексты, многокомпонентный термин, извлечение терминов, базы данных, PostgreSQL, NoSQL, Redis, InfluxDB, REST API, кэширование.

**Цель работы:** разработка системы извлечения многокомпонентных терминов и их переводных эквивалентов из параллельных научно-технических текстов.

Разработанное ПО является приложением для выделения терминов из текстов, их сохранения и анализа. Приложение может использоваться лингвистами для сбора терминологических баз данных и проведения исследований.

В работе было проведено исследование, которое показало, что применение кэширования может повысить производительность приложения более чем в 2 раза.

# Содержание

<b>Введение</b>	<b>6</b>
<b>1 Аналитическая часть</b>	<b>8</b>
1.1 Метод извлечения многокомпонентных терминов на основе структурных моделей . . . . .	8
1.2 Требования к приложению . . . . .	9
1.3 Формализация данных . . . . .	9
1.4 Формализация ролей . . . . .	11
1.5 Анализ моделей данных . . . . .	13
1.5.1 Классификация СУБД по модели данных . . . . .	13
1.5.2 Выбор модели данных . . . . .	15
<b>2 Конструкторская часть</b>	<b>17</b>
2.1 Проектирование базы данных . . . . .	17
2.1.1 Формализация сущностей системы . . . . .	17
2.1.2 Ролевая модель . . . . .	18
2.1.3 Хранимая процедура базы данных . . . . .	20
2.2 Проектирование программного комплекса . . . . .	22
2.2.1 Бизнес-сценарии . . . . .	22
2.2.2 Архитектура приложения . . . . .	22
2.2.2.1 Монолитная архитектура . . . . .	22
2.2.2.2 Микросервисная архитектура . . . . .	22
2.2.2.3 Выбор архитектуры приложения . . . . .	23
2.2.3 Связь компонентов приложения . . . . .	24
2.2.3.1 REST API . . . . .	24
2.2.3.2 gRPC . . . . .	25
2.2.3.3 Выбор способа взаимодействия компонентов приложения . . . . .	27
2.2.4 Структура программного комплекса . . . . .	27
2.2.5 Паттерны проектирования . . . . .	29
<b>3 Технологическая часть</b>	<b>30</b>
3.1 Выбор СУБД . . . . .	30
3.1.1 Долговременное хранение данных . . . . .	30

3.1.2	Кеширование данных . . . . .	30
3.1.3	Хранение логов системы . . . . .	30
3.2	Средства реализации . . . . .	31
3.3	Детали реализации . . . . .	32
3.3.1	Роли базы данных . . . . .	32
3.3.2	Хранимая процедура базы данных . . . . .	32
3.3.3	Интерфейс приложения . . . . .	32
3.4	Сборка и развёртывание приложения . . . . .	35
3.5	Примеры работы ПО . . . . .	35
<b>4</b>	<b>Экспериментально-исследовательская часть</b>	<b>38</b>
4.1	Цель проводимых измерений . . . . .	38
4.2	Описание проводимых измерений . . . . .	38
4.3	Инструменты измерения времени обработки запросов . . . . .	39
4.4	Технические характеристики оборудования . . . . .	42
4.5	Результаты проведённых измерений . . . . .	42
4.5.1	Открытая линейная нагрузка . . . . .	42
4.5.2	Открытая постоянная нагрузка . . . . .	44
4.5.3	Закрытая нагрузка . . . . .	46
<b>Заключение</b>		<b>49</b>
<b>Список использованных источников</b>		<b>50</b>
<b>Приложение А Алгоритм извлечения многокомпонентных терминов</b>		<b>56</b>
<b>Приложение Б Бизнес-сценарии</b>		<b>60</b>
<b>Приложение В Инициализация ролевой модели базы данных</b>		<b>64</b>
<b>Приложение Г Хранимая процедура базы данных</b>		<b>65</b>
<b>Приложение Д Сборка приложения</b>		<b>73</b>
<b>Приложение Е Развёртывание приложения</b>		<b>75</b>
<b>Приложение Ж Презентация</b>		<b>79</b>

## **Определения, обозначения и сокращения**

Единица языка — элемент системы языка, неразложимый в рамках определённого уровня членения текста и противопоставленный другим единицам в подсистеме языка, соответствующей этому уровню.

Терминологическая единица — элемент терминологической системы, которая является языковым выражением системы понятий определенной предметной области.

Термин — слово или словосочетание, являющееся названием определённого понятия в определенной предметной области.

Модель данных — это абстрактное, независимое, логическое определение структур данных, операторов над данными и прочего, что в совокупности составляет абстрактную систему, с которой взаимодействует пользователь.

База данных (БД) — совокупность взаимосвязанных данных некоторой предметной области, хранимых в памяти ЭВМ и организованных таким образом, что эти данные могут быть использованы для решения многих задач многими пользователями.

Система управления базами данных (СУБД) — приложение, обеспечивающее создание, хранение, обновление и поиск информации в базах данных.

Индекс — объект базы данных, создаваемый с целью повышения производительности поиска данных.

Бизнес-сценарий — сценарий взаимодействия пользователя с программным продуктом для достижения конкретной цели.

Сервис — абстракция, определяющая что-то, что предоставляет услугу.

Компонент — единица развёртывания ПО, которая является реализацией сервиса, содержащей поведение.

Микросервис — сервис, отвечающий за один элемент логики в определенной предметной области.

## **Введение**

Стремительное развитие и внедрение технологий искусственного интеллекта и технологий автоматической обработки текстовой информации способствуют развитию лингвистических баз данных как основы создания прикладных программных средств, проведения лингвистических исследований источников информации при решении ряда прикладных задач, где однозначная и упорядоченная терминология имеет особую значимость. Под терминологической базой данных принято понимать организованную в соответствии с определёнными правилами и поддерживаемую в памяти компьютера совокупность данных, характеризующую актуальное состояние некоторой предметной области и используемую для удовлетворения информационных потребностей пользователей [1]. Каждый термин дополнен информацией о его значении, эквивалентах в других языках, кратких формах, синонимах, сведениях об области применения. По целевому назначению терминологические базы данных разделяют на одноязычные, предназначенные для обеспечения информацией о стандартизованной и рекомендованной терминологии, и многоязычные, ориентированные на работы по переводу научно-технической литературы и документации [2] [3].

Создание терминологических баз данных представляет собой сложный и трудоемкий процесс, требующий значительного количества времени на их создание и обновление, что особенно важно для развивающихся терминологий таких предметных областей, как авиация, космонавтика, нанотехнологии, биоинженерия, информационные технологии и многих других. Одним из наиболее время-затратных процессов является ручной сбор иллюстративного материала – извлечение специальной терминологии из коллекций текстов, что требует наличия средств автоматического извлечения многокомпонентных терминов при обработке научно-технических текстов.

Существующие программные средства автоматического извлечения терминов основаны на лингвистических и статистических методах. Могут быть ис-

пользованы и методы машинного обучения [4], сложность их реализации вызвана необходимостью наличия огромных массивов обучающих данных, которые могут отсутствовать для определенной предметной области. В основе лингвистических методов лежит использование грамматики лексико-синтаксических шаблонов, представляющих собой структурные модели лингвистических конструкций [5] [6]. Статистический подход заключается в нахождении n-грамм слов по заданным частотным характеристикам [7] [8]. Гибридный подход для выделения терминологических сочетаний, объединяющий лингвистический и статистический методы, заключается в предварительном описании моделей, по которым могут быть построены термины для последующего поиска их в коллекции текстов [9].

Выравнивание терминологических единиц в параллельных текстах обычно осуществляется в два этапа: сначала выделяют терминологические единицы на каждом языке отдельно, затем один из одноязычных списков терминов-кандидатов интерпретируется как язык источника, и для каждого термина-кандидата на языке источнике предлагаются потенциально эквивалентные термины в списке терминов-кандидатов на языке перевода [10].

Целью данной работы является разработка системы извлечения многокомпонентных терминов и их переводных эквивалентов из параллельных научно-технических текстов. Для достижения поставленной цели необходимо решить следующие задачи.

1. Провести анализ предметной области и формализовать задачу.
2. Спроектировать базу данных и структуру программного обеспечения.
3. Реализовать интерфейс для доступа к базе данных.
4. Реализовать программное обеспечение, которое позволит пользователю создавать, получать и изменять сведения из разработанной базы данных.
5. Провести исследование зависимости времени выполнения запросов от использования кеширования данных текущей сессии пользователя.

# **1 Аналитическая часть**

В данном разделе будет описана постановка задачи, модель и структура базы данных, а также будут определены роли пользователей в системе.

## **1.1 Метод извлечения многокомпонентных терминов на основе структурных моделей**

Предлагаемый метод автоматического извлечения русскоязычных многокомпонентных терминов на основе базы данных структурных моделей терминологических словосочетаний состоит из пяти основных этапов [11].

1. Анализ предложения по частям речи.
2. Удаление частей речи и их сочетаний, которые не входят в состав терминологических словосочетаний:
  - глаголы;
  - союзы;
  - местоимения;
  - частицы;
  - знаки препинания;
  - «наречие + предлог».
3. Удаление из оставшихся терминов-кандидатов стоп-слов, указанных в специальной зоне словаря. Под стоп-словами понимаются слова, которые образуют широко используемые коллокации с терминами, но в совокупности не являются терминами по сути, например «современная химия», «рассматриваемый метод синтеза о-гликозидов».
4. Соотнесение полученных цепочек слов с шаблонами терминологических словосочетаний, которые хранятся в базе структурных моделей терминов.
5. Проверка полученных терминов-кандидатов по базе данных.
  - 5.1. Если термин-кандидат есть в базе данных, то он извлекается как термин.
  - 5.2. Если полученный термин-кандидат отсутствует в базе данных, то он отправляется терминологу для ручной обработки.

5.3. Если термин-кандидат состоит из нескольких слов, то производится попытка разбить его на несколько терминов.

## **1.2 Требования к приложению**

Диаграмма, оформленная в соответствии с нотацией IDEF0 и отражающая декомпозицию алгоритма работы системы извлечения многокомпонентных терминов, представлена в приложении А. Исходя из специфики решаемой задачи, можно сформулировать требования к приложению [12].

1. Задание исходных данных должно производиться из файла или через специально предусмотренные поля ввода.
2. Пользователю должна быть предоставлена возможность перед сохранением терминов в базу данных выполнять над ними следующие операции.
  - 2.1. Редактирование.
  - 2.2. Сопоставление переводных эквивалентов.
  - 2.3. Присвоение характеристик.
3. Пользователь может анализировать и редактировать добавленные в базу данных термины путём их поиска по заданным характеристикам.

## **1.3 Формализация данных**

Разрабатываемая база данных должна хранить информацию о следующих сущностях:

- пользователи;
- единицы русского языка;
- единицы иностранного языка;
- характеристики единиц языка;
- структурные модели слоя;
- элементы структурных моделей слоя;
- контексты употребления единиц языка.

На рисунке 1 представлена ER-диаграмма сущностей в нотации Чена, описывающая сущности, их атрибуты и связи между сущностями в разрабатываемой базе данных.

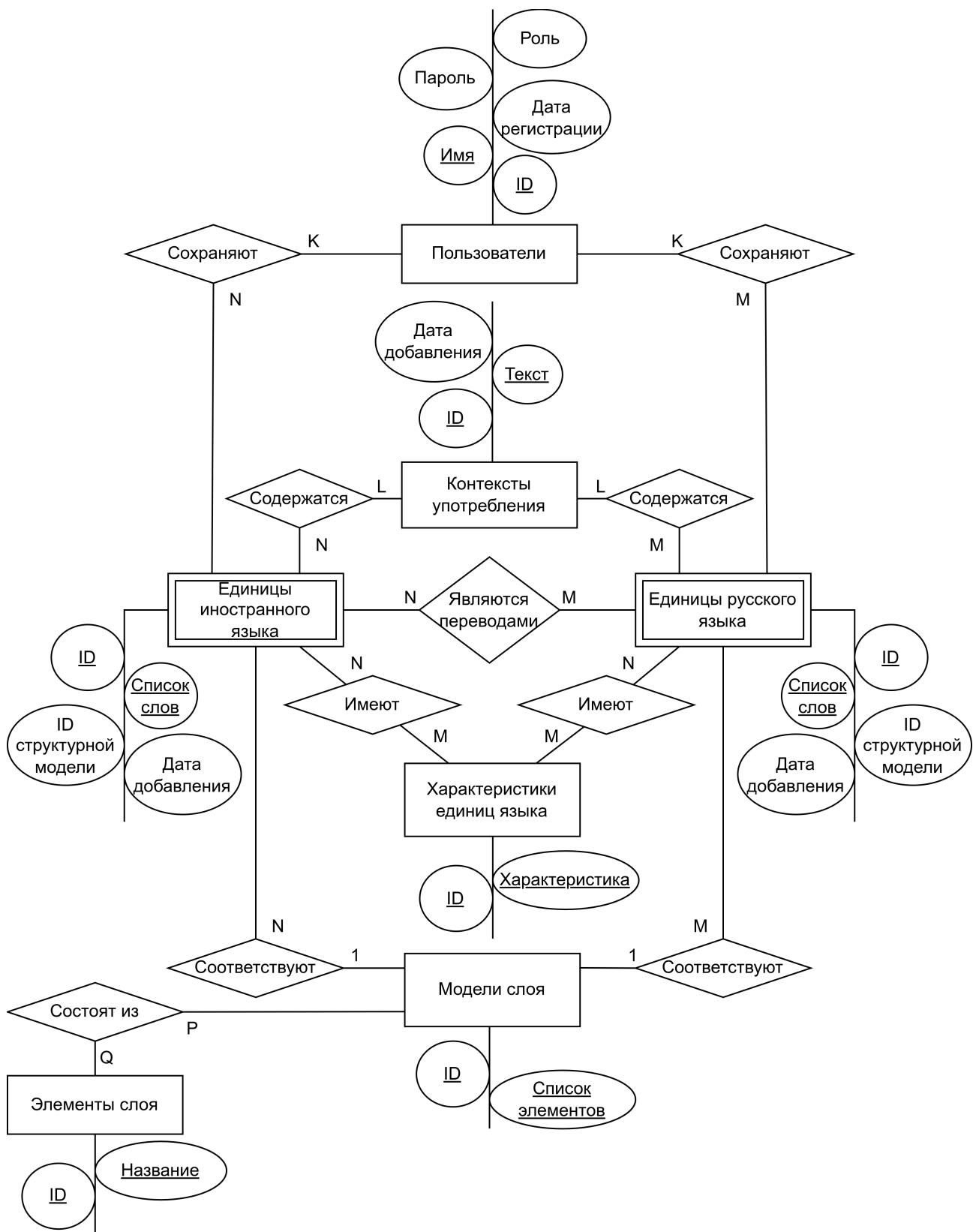


Рисунок 1 – ER-диаграмма сущностей базы данных

Стоит отметить, что тексты могут размечаться на нескольких слоях. Это означает, что текст может разбиваться на разные единицы языка и из него можно выделять отдельные слова, термины, синтаксические деревья, семантические падежи и т.д. Соответственно, разрабатываемая база данных должна иметь многослойную структуру: работа с одними и теми же сущностями должна производиться по отдельности для каждого слоя разметки текстов.

#### **1.4 Формализация ролей**

Для работы с системой обязательным этапом является прохождение аутентификации. Пользователь может работать в системе под одной из следующих ролей.

1. Студент — пользователь, имеющий возможность обрабатывать тексты, сохранять выделенные термины, а также анализировать и редактировать только те термины, которые он выделил.
2. Преподаватель — пользователь, обладающий функционалом, доступным роли «Студент», и имеющий возможность анализировать и редактировать все термины, хранящиеся в базе данных. Также данной роли доступна функция добавления нового слоя разметки текстов.
3. Администратор — пользователь, имеющий возможности, доступные роли «Преподаватель», а также имеющий в распоряжении специальные функции для анализа состояния системы и настройки её работы. Также администратор имеет возможность создавать аккаунты, соответствующие любой из ролей в системе.

В ходе использования приложения должна быть предусмотрена возможность смены ролей путём прохождения повторной аутентификации.

Также стоит отметить, что до входа в аккаунт пользователь считается неавторизованным и не имеет доступа к функционалу системы, так как любая работа с терминами должна быть персонализирована.

На рисунке 2 представлена диаграмма вариантов использования системы в соответствии с выделенными типами пользователей.

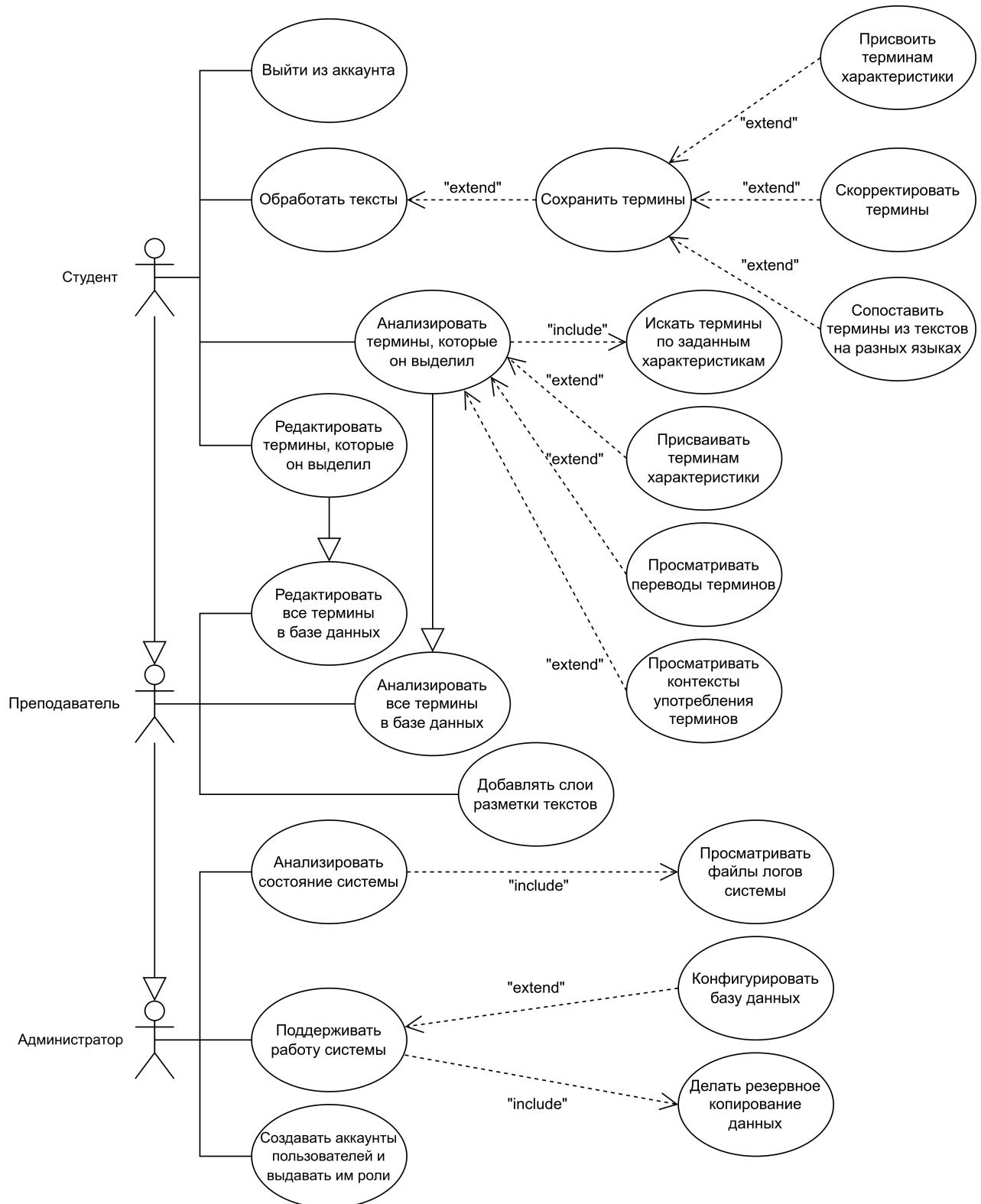


Рисунок 2 – Диаграмма вариантов использования

## **1.5 Анализ моделей данных**

### **1.5.1 Классификация СУБД по модели данных**

По модели данных СУБД можно разделить на следующие типы.

#### **1. Дореляционные.**

Старые (дореляционные) системы можно разделить на три большие категории: системы с инвертированными списками (inverted list), иерархические (hierarchic) и сетевые (network) [14].

##### **1.1. Инвертированные списки (файлы).**

База данных на основе инвертированных списков представляет собой совокупность файлов (таблиц), содержащих записи. Для записей в файле определен некоторый порядок, диктуемый физической организацией данных. Для каждого файла может быть определено произвольное число других упорядочений на основании значений некоторых полей записей (инвертированных списков). Обычно для этого используются индексы. В такой модели данных отсутствуют ограничения целостности как таковые. Все ограничения на возможные экземпляры базы данных задаются теми программами, которые с ней работают. Одно из немногих ограничений, которое может присутствовать — это ограничение, задаваемое уникальным индексом.

##### **1.2. Иерархические.**

Иерархическая модель данных состоит из объектов с указателями от родительских объектов к дочерним, соединяя вместе связанную информацию. Иерархические базы данных могут быть представлены в виде дерева. Их производительность в значительной степени зависит от подхода, выбранного самим пользователем (прикладным программистом и/или администратором базы данных).

##### **1.3. Сетевые.**

К основным понятиям сетевой модели данных относятся элемент (узел)

и связь. Узел — это совокупность атрибутов данных, описывающих некоторый объект. Сетевые базы данных могут быть представлены в виде графа. В сетевой БД логика процедуры выборки данных зависит от физической организации этих данных. Поэтому эта модель не является полностью независимой от приложения. Другими словами, если необходимо изменить структуру данных, то нужно изменить и приложение.

## 2. Реляционные.

Определение реляционной системы требует [14], чтобы база данных только воспринималась пользователем как набор таблиц. Таблицы в реляционной системе являются логическими, а не физическими структурами. Таблицы представляют собой абстракцию способа физического хранения данных, в которой детали реализации на уровне физической памяти скрыты от пользователя.

Реляционные базы данных основаны на информационном принципе: все информационное наполнение базы данных представлено одним и только одним способом, а именно — явным заданием значений, помещенных в позиции столбцов в строках таблицы. Этот метод представления — единственно возможный для реляционных баз данных. В частности, нет никаких указателей, связывающих одну таблицу с другой.

Реляционная модель состоит из следующих компонентов [14].

- Неограниченный набор скалярных типов (включая, в частности, логический тип).
- Генератор типов отношений и соответствующая интерпретация для сгенерированных типов отношений.
- Возможность определения переменных отношения для указанных сгенерированных типов отношений.
- Операция реляционного присваивания для присваивания реляционных значений указанным переменным отношения.
- Неограниченный набор общих реляционных операторов (реляционная

алгебра) для получения значений отношений из других значений отношений.

### 3. Постреляционные.

В связи с быстрым ростом количества данных и их усложнением возникла необходимость в поиске новых подходов к хранению и обработке, отличных от реляционных. Таким решением стала NoSQL-технология (Not Only SQL). Постреляционная модель является расширением реляционной модели. Она снимает ограничение неделимости данных, допуская многозначные поля, значения которых не являются атомарными, и набор значений воспринимается как самостоятельная таблица, встроенная в главную таблицу [15].

Наиболее популярными типами постреляционных СУБД являются:

- ключ-значение (Redis, Tarantool, Oracle NoSQL DB);
- колоночные (Vertica, ClickHouse, HBase);
- документо-ориентированные (CouchDB, MongoDB);
- графовые (InfoGrid, GraphX, Neo4j).

#### 1.5.2 Выбор модели данных

Для долговременного хранения данных будет использоваться реляционная модель по следующим причинам:

- 1) данные имеют чётко заданную структуру;
- 2) исключается дублирование данных за счёт использования связей между отношениями с помощью внешних ключей;
- 3) доступ к данным отделяется от способа их организации на уровне физической памяти;

Для кратковременного хранения данных о текущей сессии пользователя (в частности, сохранённых им терминов) будет использоваться нереляционная модель по следующим причинам:

- 1) данные могут не иметь общей структуры;
- 2) появляется возможность хранения вложенных структур данных;

3) повышается быстродействие за счёт возможности хранения всех данных в оперативной памяти (In-Memory Database);

In-Memory — набор концепций хранения данных, основанных на их сохранении в оперативной памяти сервера и использовании вторичной памяти для хранения резервных копий. Быстродействие In-Memory баз данных по сравнению с реляционными позволит повысить отзывчивость системы, уменьшив время ожидания пользователя при выполнении сохранения и поиска терминов.

### **Вывод из аналитической части**

В данном разделе были описан метод извлечения многокомпонентных терминов из научно-технических текстов, на основе которого были сформулированы требования к приложению. Также были описаны сущности базы данных и определены следующие роли пользователей: студент, преподаватель и администратор. Были проанализированы модели данных: для долговременного хранения данных будет использоваться реляционная модель, а для кэширования данных пользовательских сессий — нереляционная (In-Memory хранилище).

## 2 Конструкторская часть

В данном разделе будет формализована структура базы данных и будут описаны подходы к разработке приложения.

### 2.1 Проектирование базы данных

#### 2.1.1 Формализация сущностей системы

На рисунке 3 представлена диаграмма, отражающая информацию о таблицах проектируемой базы данных.

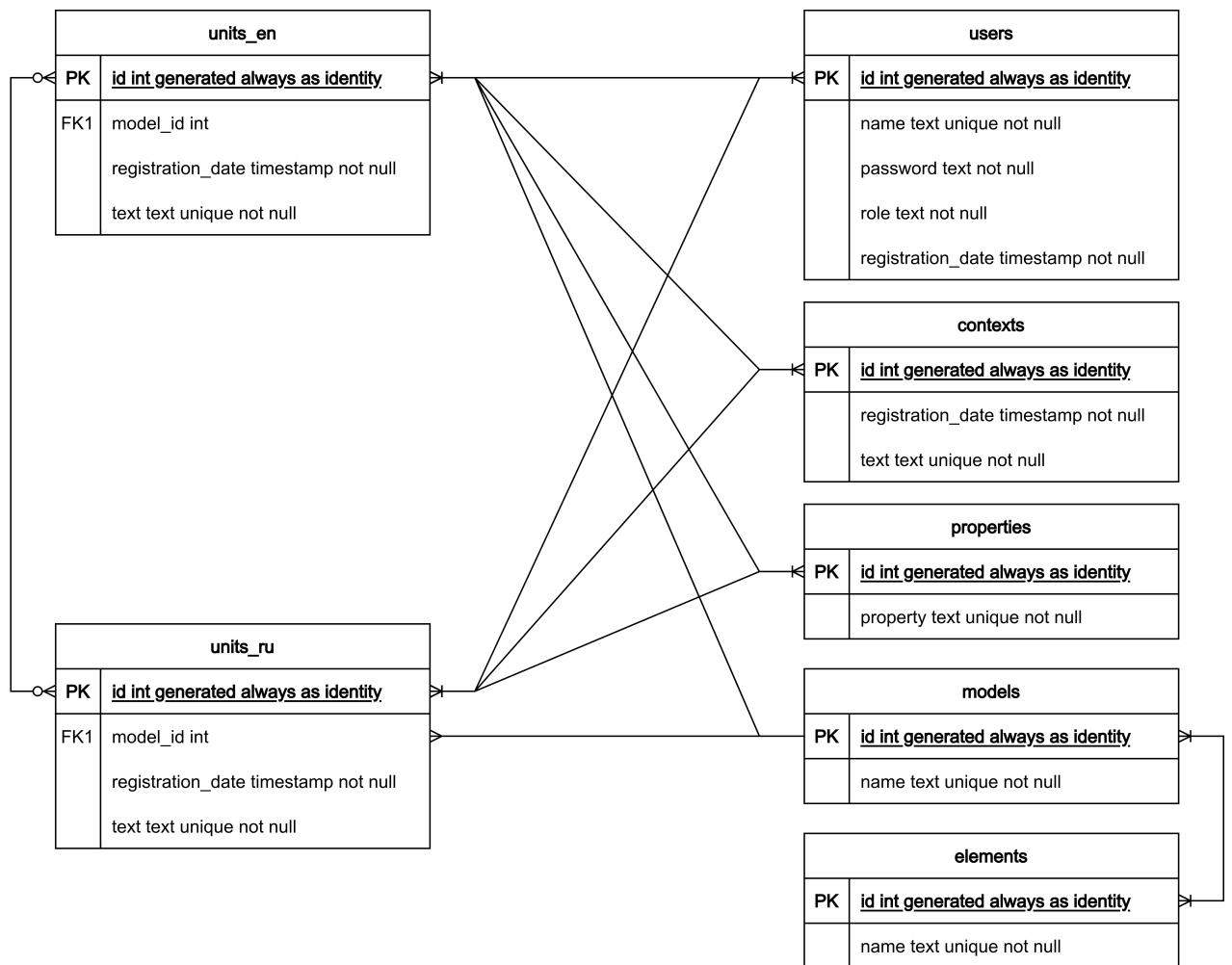


Рисунок 3 – Диаграмма базы данных

Для обеспечения возможности разметки текстов на нескольких слоях необходимо определить, какие таблицы должны создаваться отдельно для каждого слоя. Таблицы **users**, **contexts** и **properties** не должны зависеть от слоя разметки текстов, а все остальные таблицы должны хранить информацию о каждом слое

отдельно.

### **2.1.2 Ролевая модель**

Для поддержания целостности данных необходимо задать ограничения на операции доступа к данным. В таблице 1 представлено описание прав пользователей на работу с таблицами разрабатываемой базы данных.

Таблица 1 – Роли базы данных

Роль	Права на таблицы			
	Выборка	Вставка	Обновление	Удаление
Студент	contexts, properties, units_ru, units_en, models, elements	contexts, properties, units_ru, units_en	contexts, properties, units_ru, units_en	properties
Преподаватель	contexts, properties, units_ru, units_en, models, elements	contexts, properties, units_ru, units_en, models, elements	contexts, properties, units_ru, units_en, models, elements	contexts, properties, models, elements
Администратор	users, contexts, properties, units_ru, units_en, models, elements	users, contexts, properties, units_ru, units_en, models, elements	users, contexts, properties, units_ru, units_en, models, elements	users, contexts, properties, units_ru, units_en, models, elements

Если пользователь имеет право на работу с какой-либо таблицей базы данных, то он также может работать со всеми соответствующими таблицами-связками.

### **2.1.3 Хранимая процедура базы данных**

Для создания нового слоя разметки текстов необходимо совершить последовательность действий над несколькими таблицами базы данных, которые можно объединить в хранимую процедуру. Её задачей будет создание связанных таблиц для хранения данных о новом слое, а также выдача пользователям прав на работу с ними.

На рисунке 4 представлена схема алгоритма работы хранимой процедуры, добавляющей новый слой разметки текстов.

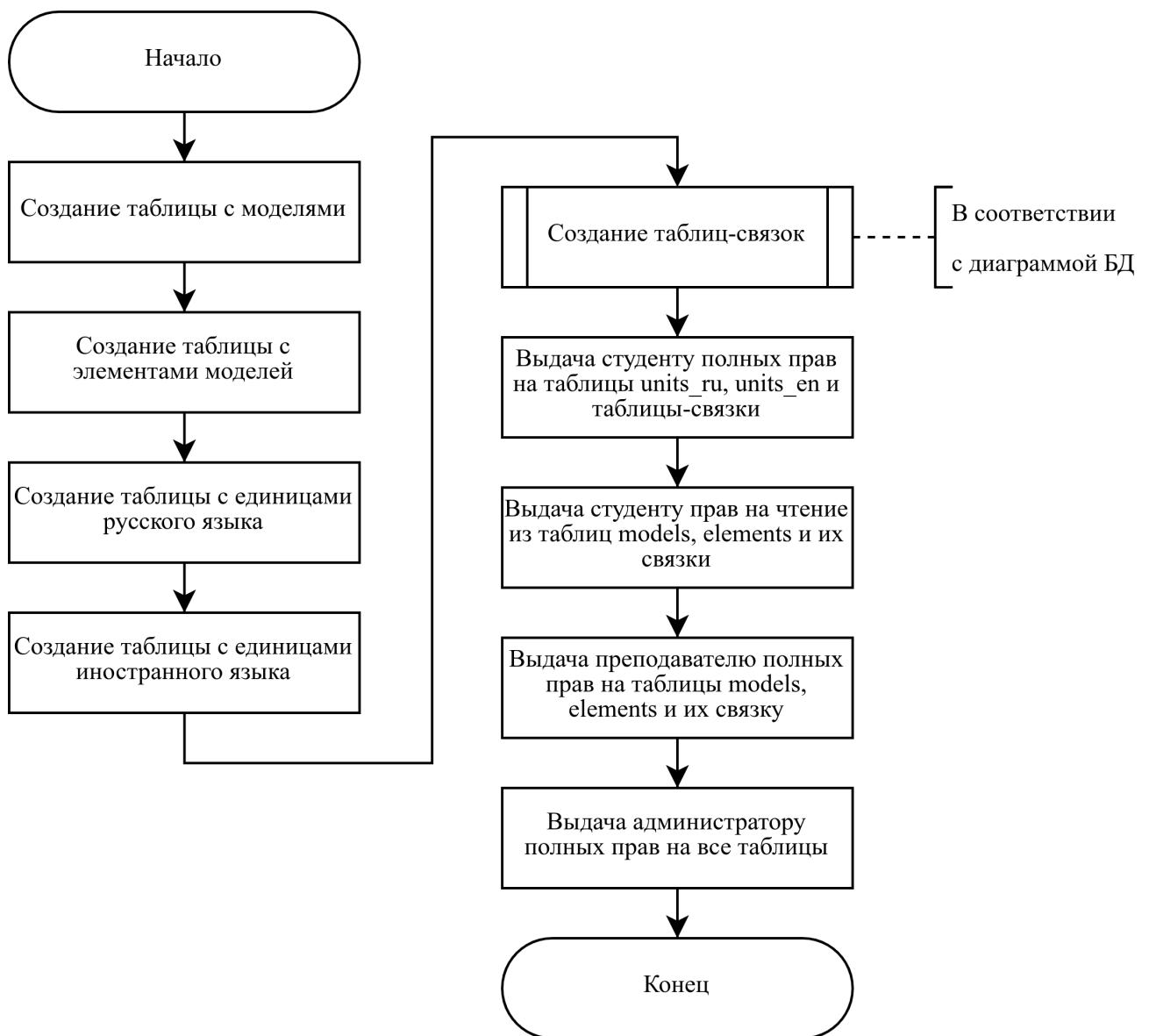


Рисунок 4 – Схема алгоритма добавления нового слоя разметки текстов

## **2.2 Проектирование программного комплекса**

### **2.2.1 Бизнес-сценарии**

Для разработки функций приложения необходимо описать его бизнес-логику. Диаграммы в нотации BPMN, отражающие формализацию бизнес-правил, представлены в приложении Б.

### **2.2.2 Архитектура приложения**

Приложение состоит из нескольких компонентов: пользовательский интерфейс, базы данных и внешние сервисы. Архитектура приложения определяет, как эти компоненты будут взаимодействовать друг с другом, а также устанавливает границы между разными частями приложения и их ответственностьюми.

#### **2.2.2.1. Монолитная архитектура**

Монолитная архитектура [16] — это традиционная модель программного обеспечения, которая представляет собой единый модуль, работающий автономно и независимо от других приложений. Монолитная архитектура представляет собой вычислительную сеть с единой базой кода, в которой объединены все бизнес-задачи. Чтобы внести изменения в такое приложение, необходимо обновить весь стек через базу кода, а также создать и развернуть обновленную версию интерфейса, находящегося на стороне службы. Это ограничивает работу с обновлениями и требует много времени.

Монолитную архитектуру можно использовать на начальных этапах проектов, чтобы облегчить развертывание и управление кодом. Это позволяет сразу выпускать всё, что есть в монолитном приложении.

#### **2.2.2.2. Микросервисная архитектура**

Микросервисная архитектура [16] представляет собой метод организации архитектуры, основанный на ряде независимо развертываемых служб. Эти службы реализуют независимую бизнес-логику и, как правило, имеют собственную базу данных. Обновление, тестирование, развертывание и масштабирова-

ние выполняются внутри каждой службы. Микросервисы разбивают крупные задачи, характерные для конкретного бизнеса, на несколько независимых кодовых баз. Микросервисы в составе приложения должны иметь независимую логику и ограниченную зону ответственности.

При переходе от монолитной архитектуры к микросервисной возникает задача организации взаимодействия компонентов приложения (в частности, транспорта данных). Соответственно, часть проблем переходит из плоскости кода на инфраструктурный и транспортный уровень.

### **2.2.2.3. Выбор архитектуры приложения**

Основные преимущества использования монолитной архитектуры:

- 1) начальная разработка;
- 2) передача данных между компонентами системы;
- 3) единая кодовая база;
- 4) хранение состояния (stateful);
- 5) инфраструктура развертывания;
- 6) единое версионирование проекта;
- 7) производительность;
- 8) интеграционное тестирование.

Основные преимущества использования микросервисной архитектуры:

- 1) независимая разработка и выпуск;
- 2) применение разных технологий для каждой выделенной задачи;
- 3) независимое развёртывание и горизонтальное масштабирование;
- 4) модульное тестирование;
- 5) отказоустойчивость системы;
- 6) повторное использование кода;
- 7) возможность полностью переписать отдельные компоненты приложения.

Для решения поставленной задачи будет использоваться микросервисная архитектура, так как она позволяет независимо разрабатывать и масштабировать компоненты приложения.

### **2.2.3 Связь компонентов приложения**

Программные компоненты приложения могут взаимодействовать друг с другом с помощью интерфейсов прикладного программирования (Application Programming Interface, API). API описывает, как выполнять клиентские запросы, какие структуры данных использовать и каких стандартов должны придерживаться клиенты. В нем также описываются виды запросов, которые один компонент может отправлять другому. Для разработки API широко используются архитектурные стили REST API и gRPC.

#### **2.2.3.1. REST API**

REST, representational state transfer (англ. передача репрезентативного состояния) [19] описывает архитектуру клиент-сервер, в которой данные сервера становятся доступными для клиентов через формат обмена сообщениями JSON или XML.

REST определяется следующими архитектурными ограничениями [19].

1. Модель клиент-сервер.
2. Отсутствие хранения состояния клиента на сервере.
3. Кэширование ответов сервера на стороне клиента.
4. Унифицированный интерфейс.
5. Многоуровневая система.
6. Код по запросу (необязательное ограничение).

Приложение, соответствующее данным архитектурным ограничениям, квалифицируется как «RESTful». Это сеть веб-страниц (виртуальная машина), по которой пользователь перемещается с помощью ссылок (переходы состояний) и в результате попадает на нужную страницу (демонстрация состояния приложения).

Также важно отметить, что REST API практически всегда использует протокол HTTP. Это наиболее распространенный формат, используемый для разработки веб-приложений или соединения микросервисов. Если веб-приложение

реализует REST API, то клиенты могут использовать каждый его компонент в качестве ресурса. Обычно ресурсы доступны через общий интерфейс, который реализует различные HTTP-методы, такие как GET, POST, DELETE и PUT.

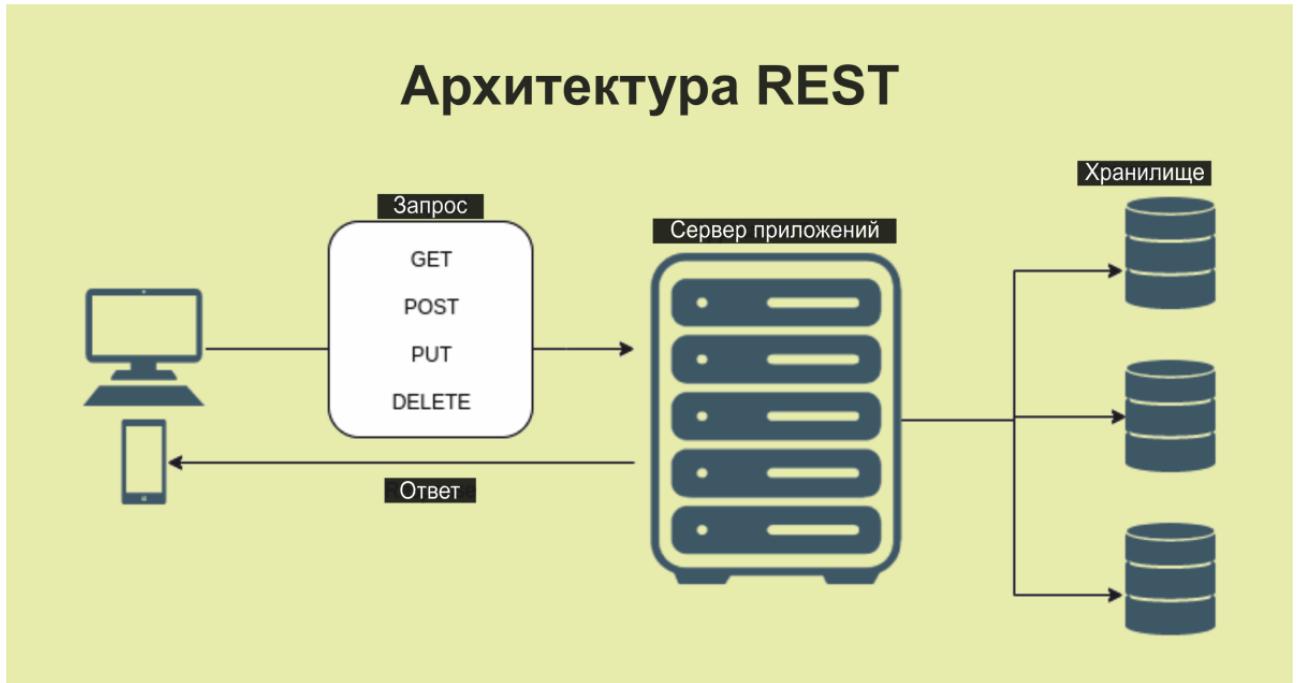


Рисунок 5 – Архитектура REST API

В RESTful API пользователь отправляет запрос на URL-адрес — унифицированный указатель ресурсов, который вызывает ответ с полезной нагрузкой в JSON, XML или любой другой поддерживаемый формат данных. Полезная нагрузка представляет собой ресурс, который нужен пользователю. Общие запросы клиентов включают

- HTTP-метод, указывающий, что должно обрабатываться на ресурсе;
- путь к ресурсу;
- заголовок с данными о запросе;
- полезная нагрузка сообщения для конкретного клиента.

#### 2.2.3.2. gRPC

gRPC, remote procedure call (англ. удалённый вызов процедур) [17] — это протокол RPC, реализованный поверх HTTP/2 (протокол прикладного уровня). Основные особенности gRPC:

- бинарный протокол (HTTP/2);
- мультиплексирование множества запросов на одно соединение (HTTP/2);
- сжатие заголовков (HTTP/2);
- строго типизированный сервис и определение сообщения (Protobuf);
- идиоматические реализации библиотек клиент/сервер на многих языках (Golang, C++, Python и другие);
- интеграция с такими компонентами экосистемы, как обнаружение сервисов, преобразователь имен, балансировщик нагрузки, трассировка и мониторинг и другие.

Удаленный вызов процедур — это веб-архитектура, позволяющая выполнять запросы на сервере с использованием предопределенных форматов сообщений. При этом сервер может быть как локальным, так и удалённым.

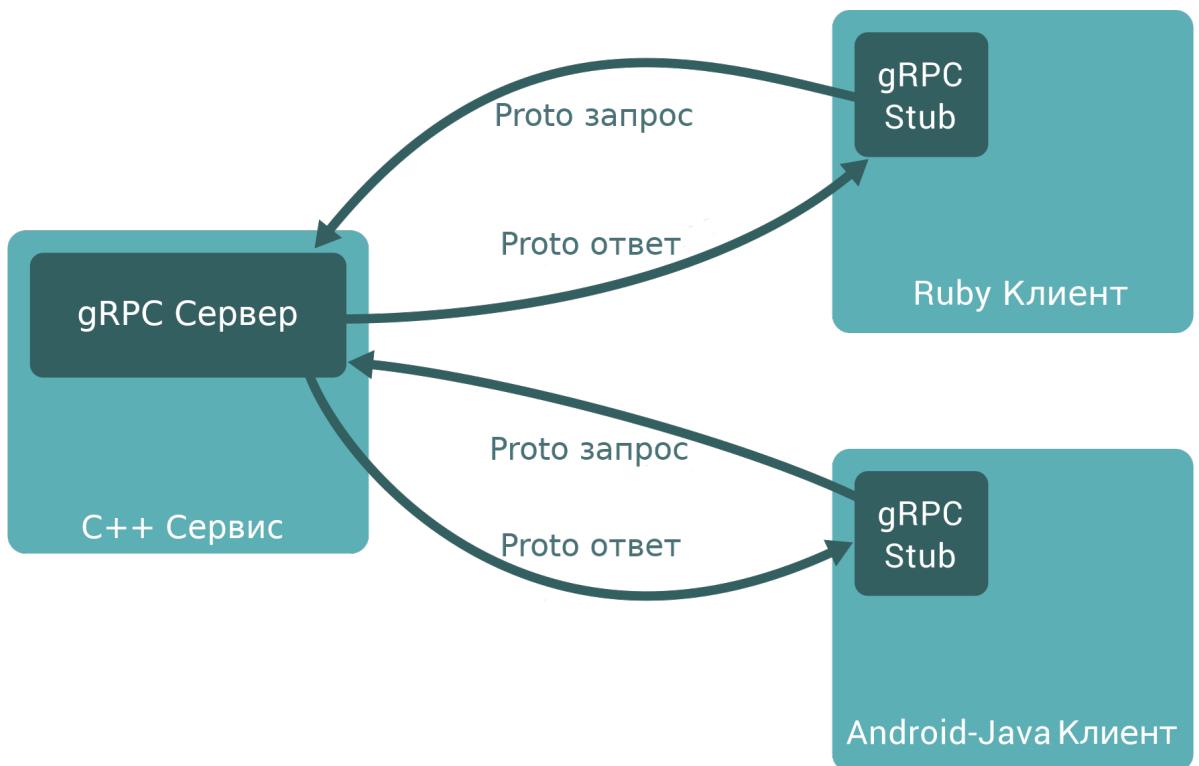


Рисунок 6 – Архитектура gRPC

Эта архитектура API позволяет нескольким функциям, созданным на разных языках программирования, работать вместе благодаря API. gRPC использует формат обмена сообщениями Protobuf (англ. буферы протокола), который

используется для сериализации структурированных данных. Для определенного круга задач API gRPC является более эффективной альтернативой REST API.

### **2.2.3.3. Выбор способа взаимодействия компонентов приложения**

В таблице 2 представлено сравнение gRPC и REST API.

Таблица 2 – Сравнительная характеристика gRPC и REST API

Характеристика	gRPC	REST API
Формат коммуникации	Унарные двусторонние запросы или потоковая передача	Только клиентские запросы
Способ коммуникации	RPC	HTTP-методы
Протокол	HTTP/2	HTTP 1.1
Формат сообщений	Protobuf (protocol buffers)	JSON/XML
Генерация кода	С помощью компилятора Protobuf	Сторонние решения, такие как Swagger [18]

Для решения поставленной задачи будет использоваться REST API по следующим причинам:

1. Универсальность: позволяет связывать любые сервисы, которые могут принимать или отправлять HTTP-запросы.
2. Скорость развёртывания.
3. Поддержка инструментов описания API (Swagger, ReDoc, Apiary и другие).

### **2.2.4 Структура программного комплекса**

На рисунке 7 представлена структура программного комплекса, оформленная в виде диаграммы развёртывания. Она отражает компоненты системы и способы их взаимодействия.

Для упрощения процесса разработки и развёртывания ПО предлагается использовать контейнеризацию, чтобы изолировать компоненты приложения

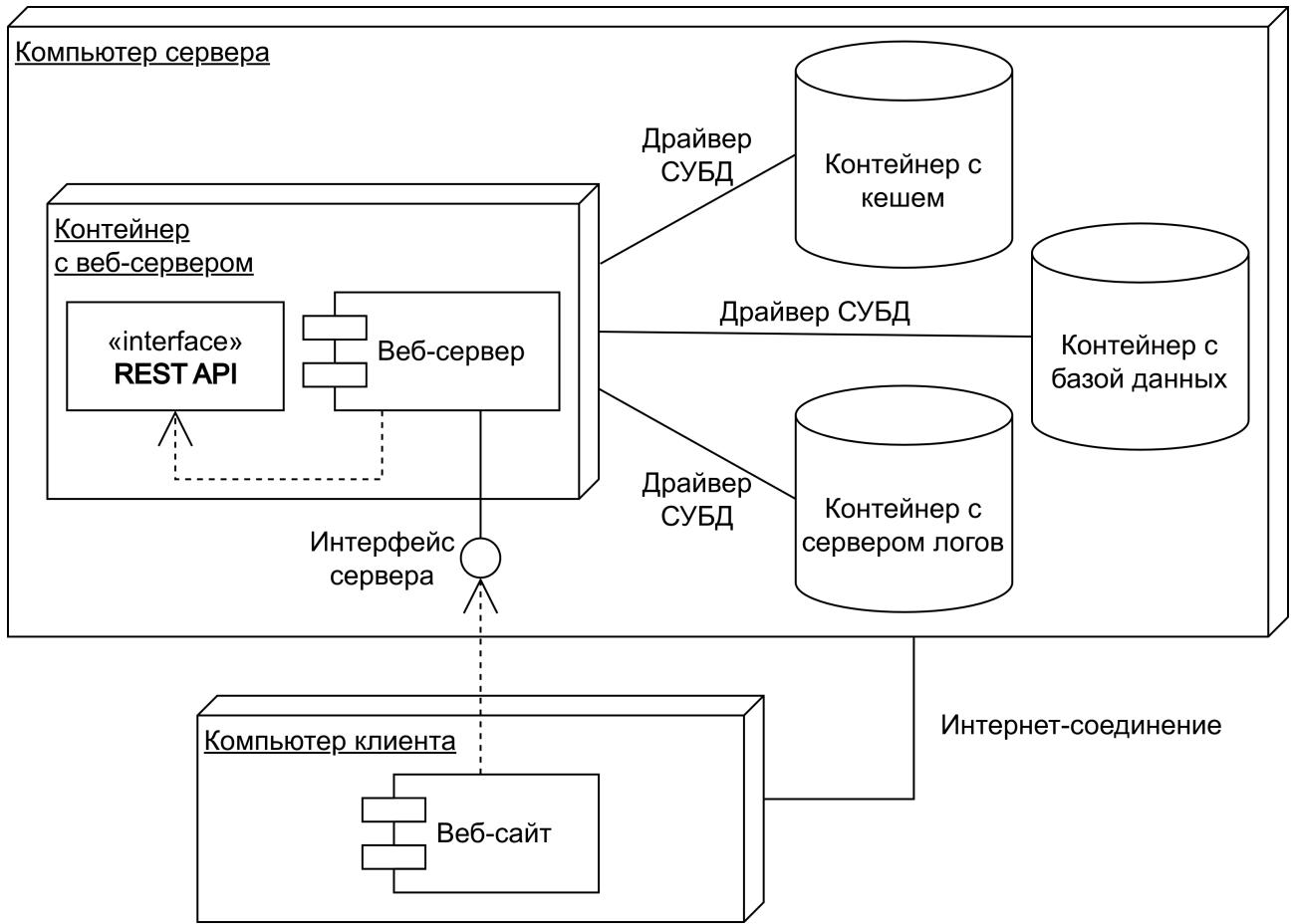


Рисунок 7 – Диаграмма развёртывания программного комплекса

друг от друга. В отличии от виртуальных машин, имеющих ОС хоста и гостевую ОС, контейнеры размещаются на одном физическом сервере с ОС хоста, которая разделяет их между собой. Совместное использование ОС хоста сразу несколькими контейнерами делает их менее требовательными к аппаратному обеспечению.

Также в целях повышения безопасности системы предлагается использовать выделенный сервер логов. Для организации базы данных для этого сервера можно использовать СУБД, ориентированную на обработку временных рядов<sup>1</sup> (СУБД-ВР) [20]. Такой формат хранения данных позволит в динамике отслеживать состояние системы при минимальных затратах ресурсов на хранение и обработку событий.

<sup>1</sup>Временной ряд (time series) — последовательность хронологически упорядоченных числовых значений, отражающих течение некоторого процесса или явления.

## **2.2.5 Паттерны проектирования**

Паттерны проектирования повышают степень повторного использования проектных и архитектурных решений. Они помогают выбрать альтернативные решения, упрощающие повторное использование системы, и избежать тех альтернатив, которые его затрудняют. Паттерны улучшают качество документации и сопровождения существующих систем, поскольку они позволяют явно описать взаимодействия классов и объектов, а также причины, по которым система была построена так, а не иначе [21].

При разработке программного комплекса будут использоваться следующие паттерны.

1. Repository (репозиторий). [22]
2. Dependency injection (инъекция зависимостей). [23]
3. Template Method (шаблонный метод). [21]

## **Вывод из конструкторской части**

В данном разделе была проведена формализация сущностей и ролей системы, разработана хранимая процедура для создания нового слоя разметки текстов. Также были formalизованы основные бизнес-правила и выбраны подходы к разработке приложения. Развёртывание приложения будет осуществляться с помощью контейнеров. Для хранения логов системы будет использоваться СУБД временных рядов (СУБД-ВР).

### **3 Технологическая часть**

В данном разделе будет описан выбор технологий для разработки и развертывания базы данных и приложения, а также будут приведены детали реализации ПО.

#### **3.1 Выбор СУБД**

##### **3.1.1 Долговременное хранение данных**

В качестве основного хранилища данных была выбрана объектно-реляционная СУБД PostgreSQL [24], которая позволит реализовать многослойную структуру хранения данных. Также стоит отметить, что PostgreSQL сертифицирована ФСТЭК России [25] и имеет открытый исходный код.

##### **3.1.2 Кеширование данных**

Для хранения сессий было выбрано in-memory хранилище Redis [26]. Данная СУБД удовлетворяет необходимым требованиям (хранилище типа ключ-значение в оперативной памяти) и не требует дополнительной настройки для начала использования.

##### **3.1.3 Хранение логов системы**

В качестве СУБД для хранения логов. Для данной цели была выбрана СУБД-ВР InfluxDB [27]. В отличие от PostgreSQL, InfluxDB более компактно хранит данные и оптимизирована на запись данных, что подходит для организации сервера логов.

В InfluxDB данные представляются в виде двумерной таблицы [20], называемой измерением (measurement). В измерении имеется столбец с метками времени (timestamp). Остальные столбцы измерения могут принадлежать одной из двух категорий: поле или тег. Поле (field) хранит данные временного ряда и состоит из ключей (field keys) и значений (field values). Тег (tag) представляет собой метаданные поля и состоит из ключей тегов (tag key) и значений тегов (tag values). Поля не индексируются, но для тегов могут быть созданы индексы. В InfluxDB отсутствует явная схема базы данных.

В InfluxDB поддерживаются понятия серии и точки [20]. Серия (series) представляет собой набор данных, имеющих общие измерение, набор тегов и ключи полей. Точка (point) представляет собой элемент данных, состоящий из следующих компонентов: измерение, набор тегов, набор полей, метка времени. Точка однозначно идентифицируется по ее серии и метке времени. При добавлении точек данных метки времени добавляются автоматически.

Хранение данных на физическом уровне в InfluxDB основано на использовании древовидной структуры данных LSM (Log-structured merge-tree) [28], которая используется в реляционных СУБД и обеспечивает быстрый доступ к данным в случае сценария работы, предполагающего частые запросы на вставку данных. В InfluxDB также поддерживается автоматическое сжатие данных для минимизации объема хранимых данных. [20]

В InfluxDB используется SQL-подобный язык запросов InfluxQL [29], который поддерживает непрерывные запросы (continuous query) — запросы, которые запускаются автоматически с заданной периодичностью.

### 3.2 Средства реализации

Система, взаимодействующая с базой данных, представляет собой веб-сервер, доступ к которому осуществляется с помощью REST API. Для реализации был выбран язык программирования Golang [30], созданный для разработки микросервисных веб-приложений. Далее будет описана разработка микросервиса, предназначенного для хранения данных.

Для взаимодействия с базами данных будут использоваться драйвера, написанные для языка Golang, предоставляющие интерфейс взаимодействия посредством языка программирования.

Для реализации REST API будет использоваться веб фреймворк gin [31]. Документирование REST API будет осуществляться с помощью Swagger [18], который поддерживает протокол openAPI [32].

Для развёртывания приложения был выбран оркестратор Docker контейнеров Docker Compose [34]. Docker контейнер позволяет изолировать прило-

жение и разворачивать его на любой машине, независимо от окружения. Это реализуется благодаря инкапсуляции всех требуемых зависимостей внутри контейнера. Docker Compose связывает контейнеры в единую систему и позволяет запускать приложение, состоящее из Docker контейнеров. В отдельных контейнерах будут развёртываться следующие службы.

1. Микросервис приложения, осуществляющего доступ к данным.
2. PostgreSQL для долговременного хранения данных.
3. Redis для кеширования данных пользовательских сессий.
4. InfluxDB для хранения логов системы.
5. Swagger для документирования REST API.
6. PgAdmin для администрирования PostgreSQL.

### **3.3 Детали реализации**

#### **3.3.1 Роли базы данных**

В конструкторской части была разработана ролевая модель, в которой выделены роли студента, преподавателя и администратора. Сценарий создания ролей и выделения им прав, соответствующих описанной ролевой модели, представлен в приложении В.

#### **3.3.2 Хранимая процедура базы данных**

В конструкторской части была разработана хранимая процедура, осуществляющая добавление нового слоя разметки текстов. PL/pgSQL [35] — процедурного расширения языка SQL, используемого в СУБД PostgreSQL. Код функции представлен в приложении Г.

#### **3.3.3 Интерфейс приложения**

Разрабатываемая система является серверным приложением, которое принимает запросы клиентов, обрабатывает их и возвращает ответ. Клиенты могут взаимодействовать с сервером через интерфейс REST API, который обеспечивает доступ к разрабатываемой базе данных.

Описание REST API, соответствующего требованиям к проектируемому приложению, представлено в таблице 3.

Таблица 3 – Описание REST API реализуемого приложения

Путь	HTTP-метод	Описание
/api/v1/login	POST	Вход в систему
/api/v1/logout	POST	Выход из системы
/api/v1/layers/all	GET	Получение всех слоёв разметки текстов
/api/v1/layers	POST	Добавление слоя разметки текстов
/api/v1/elements/all	GET	Получение всех элементов структурных моделей заданного слоя разметки текстов
/api/v1/elements	POST	Добавление элементов структурных моделей заданного слоя разметки текстов
/api/v1/models/all	GET	Получение всех структурных моделей заданного слоя разметки текстов
/api/v1/models	POST	Добавление структурных моделей заданного слоя разметки текстов
/api/v1/properties/all	PUT	Получение всех характеристик единиц языка
/api/v1/properties/unit	PUT	Получение характеристик заданной единицы языка
/api/v1/properties	POST	Добавление характеристик единиц языка
/api/v1/units/all	PUT	Получение всех единиц языка на заданном слое разметки текстов
		Продолжение на следующей странице

Таблица 3 – продолжение

Путь	Метод	Описание
/api/v1/units/models	PUT	Получение единиц языка на заданном слое разметки текстов, соответствующих заданным структурным моделям
/api/v1/units/properties	PUT	Получение единиц языка на заданном слое разметки текстов, соответствующих заданным характеристикам
/api/v1/units	POST, PATCH	Добавление и редактирование единиц языка на заданном слое разметки текстов
/api/v1/users	POST	Регистрация пользователя в системе
/api/v1/admin/stat	GET	Получение информации о состоянии веб-сервера
		Конец таблицы

### 3.4 Сборка и развёртывание приложения

Для сборки частей системы использовались Docker контейнеры, конфигурации которых представлены в приложении Д. Для их развертывания использовался Docker Compose. Листинг конфигурации в формате YAML представлен приложении Е.

### 3.5 Примеры работы ПО

Для разработанного REST API была написана документация с помощью фреймворка [18], который позволяет не только интерактивно просматривать спецификацию, но и отправлять запросы к приложению с помощью Swagger UI [36]. На рисунках 8-10 представлены примеры работы с Swagger UI.

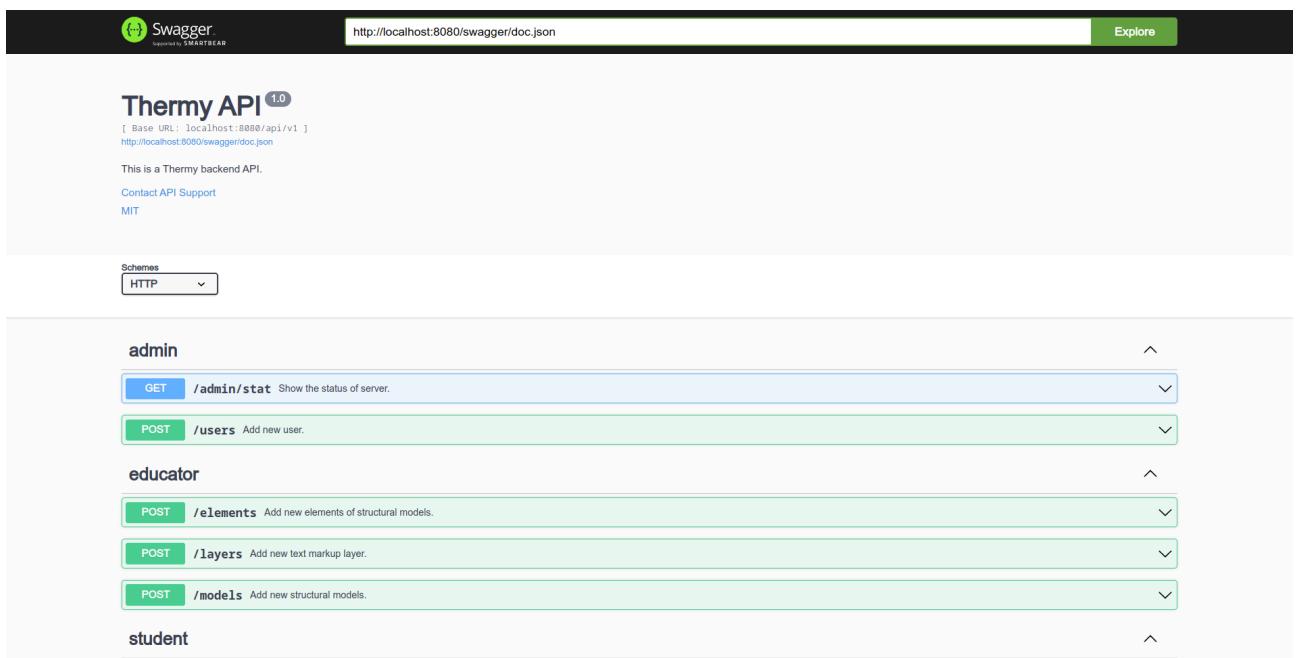


Рисунок 8 – Главная страница Swagger UI

**POST /units** Add new units in the given text markup layer.

add new units in the given text markup layer

**Parameters**

Name	Description
<b>token</b> <small>required</small> <small>string (query)</small>	User authentication token 10063865700249539947
<b>layer</b> <small>required</small> <small>string (query)</small>	Text markup layer newlayer
<b>unitsDTO</b> <small>required</small> <small>object (body)</small>	Information about stored units Edit Value : Model
<pre>{   "contexts": [     {       "ru": "контекст1, содержащий термин1 и термин2",       "en": "context1 with term1 and term2"     },     "units": [       {         "ru": {"text": "термин1", "model_id": 1, "properties_id": [1, 2, 3]},         "en": {"text": "term1", "model_id": 1, "properties_id": [3]}       },       {         "ru": {"text": "термин2", "model_id": 3},         "en": {"text": "term2", "model_id": 3, "properties_id": [3]}       }     ]   } }</pre>	

**Cancel**

Parameter content type  
application/json

Рисунок 9 – Пример заполнения формы для сохранения терминов с помощью Swagger UI

**Responses**

**Curl**

```
curl -X 'POST' \
'http://localhost:8080/api/v1/units?token=10063865700249539947&layer=newlayer' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "contexts": [
    {
      "ru": "контекст1, содержащий термин1 и термин2",
      "en": "context1 with term1 and term2"
    },
    "units": [
      {
        "ru": {"text": "термин1", "model_id": 1, "properties_id": [1, 2, 3]},
        "en": {"text": "term1", "model_id": 1, "properties_id": [3]}
      },
      {
        "ru": {"text": "термин2", "model_id": 3},
        "en": {"text": "term2", "model_id": 3, "properties_id": [3]}
      }
    ]
  }
}'
```

**Request URL**

<http://localhost:8080/api/v1/units?token=10063865700249539947&layer=newlayer>

**Server response**

Code	Details
200	Response headers
	content-length: 0 date: Tue, 16 May 2023 19:48:30 GMT

Рисунок 10 – Пример ответа на запрос сохранения терминов с помощью Swagger UI

## **Вывод из технологической части**

В данном разделе были приведены детали реализации ПО, использующего следующие технологии.

1. Golang — язык для написания серверной части приложения.
2. PostgreSQL — основное хранилище данных.
3. PgAdmin — инструмент для администрирования PostgreSQL.
4. PL/pgSQL — расширение языка SQL, использовавшееся для написания хранимых процедур базы данных.
5. Redis — кеш данных пользовательских сессий.
6. InfluxDB — сервер логов.
7. Swagger — инструмент документирования API сервера.
8. Docker — платформа для автоматизации развёртывания и управления приложениями.
9. Docker Compose — оркестратор контейнеров.

## **4 Экспериментально-исследовательская часть**

В данном разделе будет проведено исследование влияния кеширования на производительность разработанного ПО.

### **4.1 Цель проводимых измерений**

Основным сценарием использования разработанного веб-сервера является сохранение терминов после обработки текстов. Это означает, что серверу будут поступать запросы на запись данных, объёмы которых могут превышать объёмы исходных текстов. Чтобы ускорить обработку данных и повысить отзывчивость приложения, можно применить кеширование, то есть запись данных в промежуточное хранилище в оперативной памяти сервера. Далее данные могут быть перемещены в долговременное хранилище.

Целью проводимых измерений будет проведение нагрузочного тестирования и сравнение производительности веб-сервера при обработке запросов на сохранение терминов с использованием кеширования и без него.

### **4.2 Описание проводимых измерений**

Для исследования в приложении был реализован компонент доступа к данным, работающий с Redis. Он предназначен для того, чтобы избежать обращения к долговременному хранилищу (PostgreSQL). Это позволит сократить время выполнения запроса на сохранение терминов. Конфигурация запуска Redis представлена в приложении Е.

Для тестирования была выбрана конечная точка /api/v1/units, через которую осуществляется сохранение терминов.

Для проведения нагрузочного тестирования необходимо выбрать модель нагрузки: открытую или закрытую.

При тестировании с открытой моделью нагрузки количество пользователей системы неисчислимое. Интенсивность нагрузки не зависит от состояния тестируемого сервиса и скорости его ответа на запросы. С увеличением времени ответа нагрузка на сервис растёт, а значит растёт и количество параллельно

обрабатываемых запросов. Для тестирования необходимо задать RPS, то есть количество одновременных запросов к приложению в секунду.

Для тестирования с закрытой моделью нагрузки заранее задаётся число пользователей системы, которые посылают запросы в систему без задержек. Интенсивность нагрузки зависит от того, как быстро отвечает сервис. И чем быстрее отвечает тестируемый сервис, тем большую нагрузку создают пользователи.

При тестировании будут использоваться обе модели нагрузки: открытая модель нагрузки позволит сымитировать поведение пользователей веб-сервиса, а закрытая позволит определить время, за которое система выполнит заданный объём работы.

### 4.3 Инструменты измерения времени обработки запросов

Для проведения нагрузочного тестирования использовался инструмент оценки нагрузки и производительности Yandex.Tank [37]. Для визуализации результатов нагрузочного тестирования использовался сервис для мониторинга и анализа нагрузочного тестирования Overload [38]. В качестве генератора нагрузки использовался Phantom [39].

Для проведения нагрузочного тестирования с помощью Yandex.Tank было выбрано три профиля нагрузки:

- 1) открытая линейная — от 1 до 25 rps на протяжении 3 минут;
- 2) открытая постоянная — 20 rps на протяжении 1 минуты;
- 3) закрытая — 1 пользователь последовательно посылает 10000 запросов.

В листингах 1-3 представлена конфигурация для проведения нагрузочного тестирования с помощью Yandex.Tank.

Листинг 1: Конфигурация для тестирования с открытой линейной нагрузкой

```
1 phantom:  
2     address: 127.0.0.1:8080  
3     load_profile:  
4         load_type: rps
```

```
5      schedule: line(1, 25, 3m)
6      ammofile: /var/loadtest/ammo.txt
7      instances: 1
8 autostop:
9      autostop:
10         - http(5xx,10%,5s)
11 overload:
12     enabled: true
13     token_file: /var/loadtest/token.txt
14     job_name: load_test
15 console:
16     enabled: true
17 telegraf:
18     enabled: false
```

Листинг 2: Конфигурация для тестирования с открытой постоянной нагрузкой

```
1 phantom:
2     address: 127.0.0.1:8080
3     load_profile:
4         load_type: rps
5         schedule: const(20, 1m)
6     ammofile: /var/loadtest/ammo.txt
7     instances: 1
8 autostop:
9     autostop:
10        - http(5xx,10%,5s)
11 overload:
12     enabled: true
13     token_file: /var/loadtest/token.txt
14     job_name: load_test
15 console:
16     enabled: true
17 telegraf:
18     enabled: false
```

### Листинг 3: Конфигурация для тестирования с закрытой нагрузкой

```
1 phantom:
2     address: 127.0.0.1:8080
3     load_profile:
4         load_type: instances
5         schedule: const(1, 1m)
6     ammofile: /var/loadtest/ammo.txt
7     instances: 1
8     loop: 10000
9 autostop:
10    autostop:
11        - http(5xx,10%,5s)
12 overload:
13    enabled: true
14    token_file: /var/loadtest/token.txt
15    job_name: load_test
16 console:
17    enabled: true
18 telegraf:
19    enabled: false
```

В листинге 4 представлен HTTP-запрос, использовавшийся для тестирования.

### Листинг 4: HTTP-запрос для тестирования

```
1 580
2 POST /api/v1/units?layer=newlayer&token=10063865700249539947 HTTP
3 /1.1
4 Host: localhost:8080
5 User-Agent: tank
6 Content-Length: 389
7 Content-Type: application/json
8 Accept-Encoding: gzip
9 { "contexts": { "ru": "контекст1, содержащий термин1 и термин2", "en": "
```

```
"context1 with term1 and term2"}, "units": [{"ru": {"text": "термин1", "model_id": 1, "properties_id": [1, 2, 3]}, "en": {"text": "term1", "model_id": 1, "properties_id": [3]}}, {"ru": {"text": "термин2", "model_id": 3}, "en": {"text": "term2", "model_id": 3, "properties_id": [3]}}]]
```

#### 4.4 Технические характеристики оборудования

Ниже приведены технические характеристики устройства, на котором выполнялось тестирование.

- Операционная система: Manjaro Linux x86\_64 [40], версия ядра 5.15.32.
- Объём оперативной памяти: 16 Гб.
- Процессор: Intel i5-9300H 2.4 ГГц [41].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

#### 4.5 Результаты проведённых измерений

Ниже представлены результаты нагрузочного тестирования разработанного приложения, оформленные в виде графиков зависимости времени ответа веб-сервера от времени и RPS, создаваемого генератором нагрузки. На графиках показана зависимость от времени работы сервиса следующих метрик:

- 1) RPS;
- 2) среднее время ответа веб-сервера (avg, average);
- 3) квантиль уровня 98% времени ответа — промежуток времени, за который веб-сервер отвечает на 98% запросов.

##### 4.5.1 Открытая линейная нагрузка

На рисунках 11 и 12 представлены результаты проведения нагрузочного тестирования при открытой линейной нагрузке с использованием и без использования кеширования.

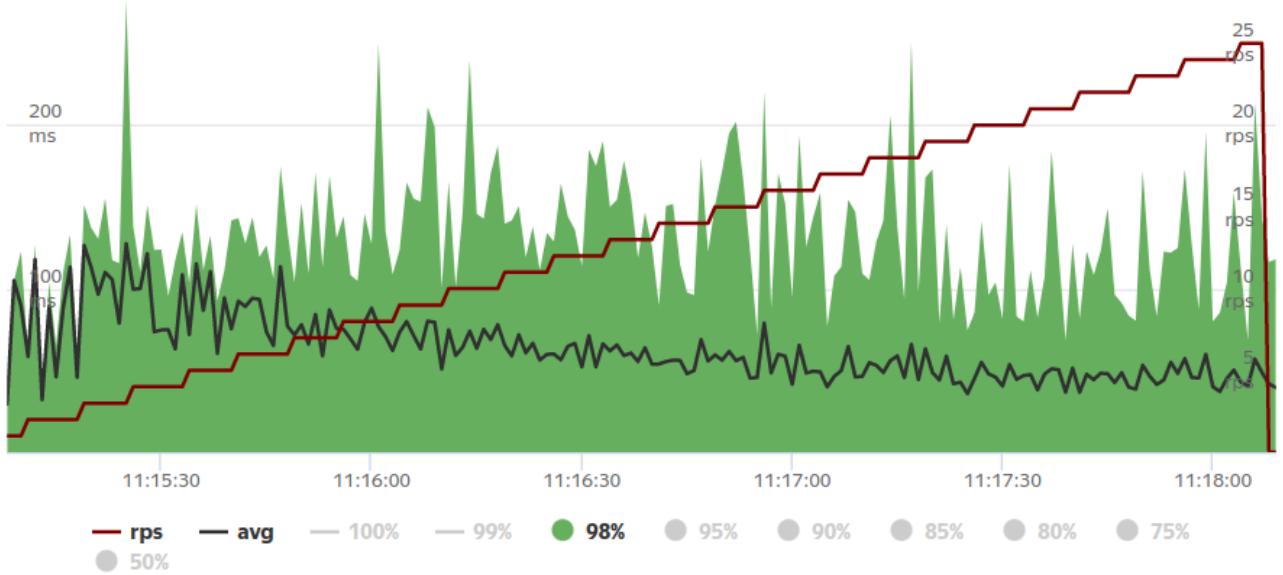


Рисунок 11 – Результат проведения нагрузочного тестирования при открытой линейной нагрузке без использования кеширования

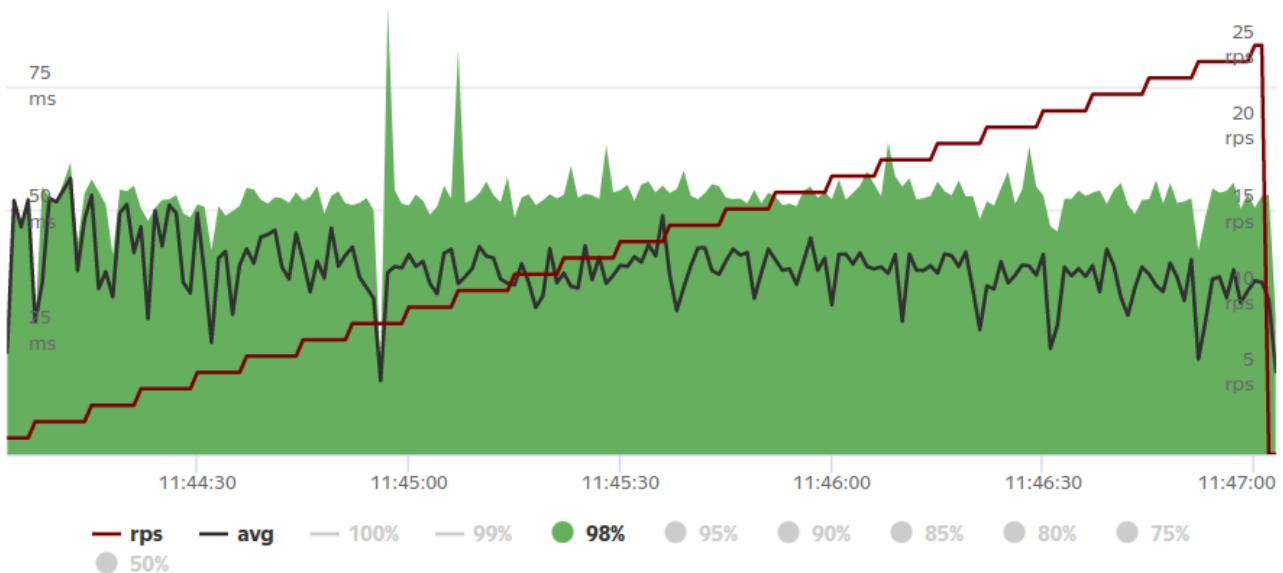


Рисунок 12 – Результат проведения нагрузочного тестирования при открытой линейной нагрузке с использованием кеширования

При использовании кеширования среднее время ответа сервера уменьшилось с 63 мс до 37 мс. Также уменьшился разброс времён ответов, то есть сервер стал отвечать на запросы стабильнее.

#### **4.5.2 Открытая постоянная нагрузка**

На рисунках 13 и 14 представлены результаты проведения нагрузочного тестирования при открытой постоянной нагрузке с использованием и без использования кеширования.

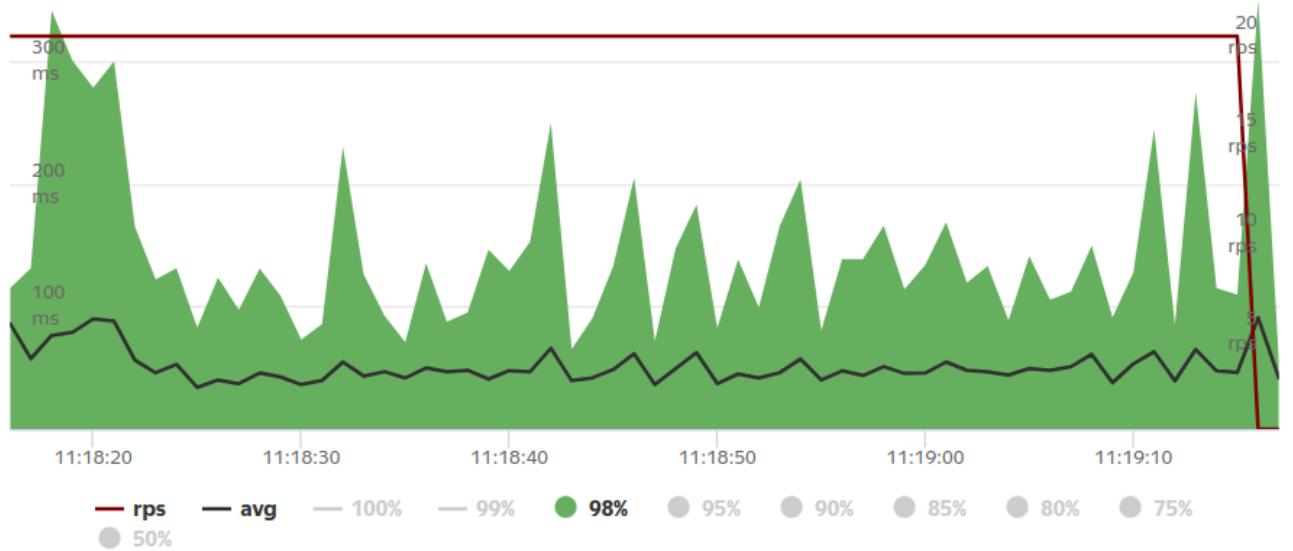


Рисунок 13 – Результат проведения нагрузочного тестирования при открытой постоянной нагрузке без использования кеширования

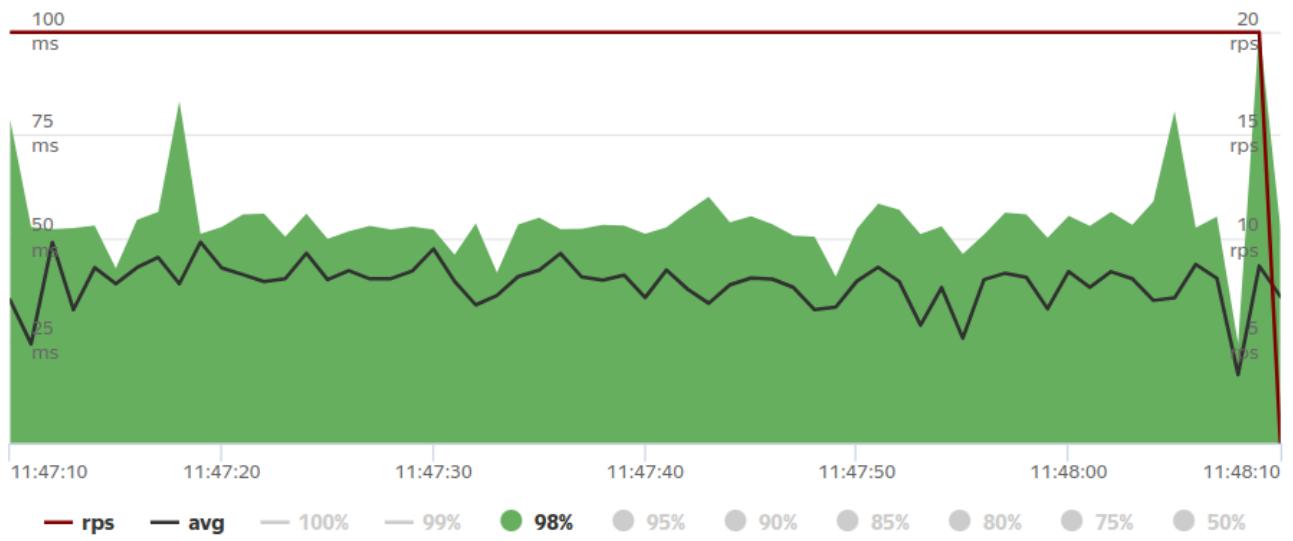


Рисунок 14 – Результат проведения нагрузочного тестирования при открытой постоянной нагрузке с использованием кеширования

При использовании кеширования среднее время ответа сервера уменьшилось с 49 мс до 38 мс. Также уменьшился разброс времён ответов, то есть сервер стал отвечать на запросы стабильнее.

#### **4.5.3 Закрытая нагрузка**

На рисунках 15 и 16 представлены результаты проведения нагрузочного тестирования при закрытой нагрузке с использованием и без использования кеширования.

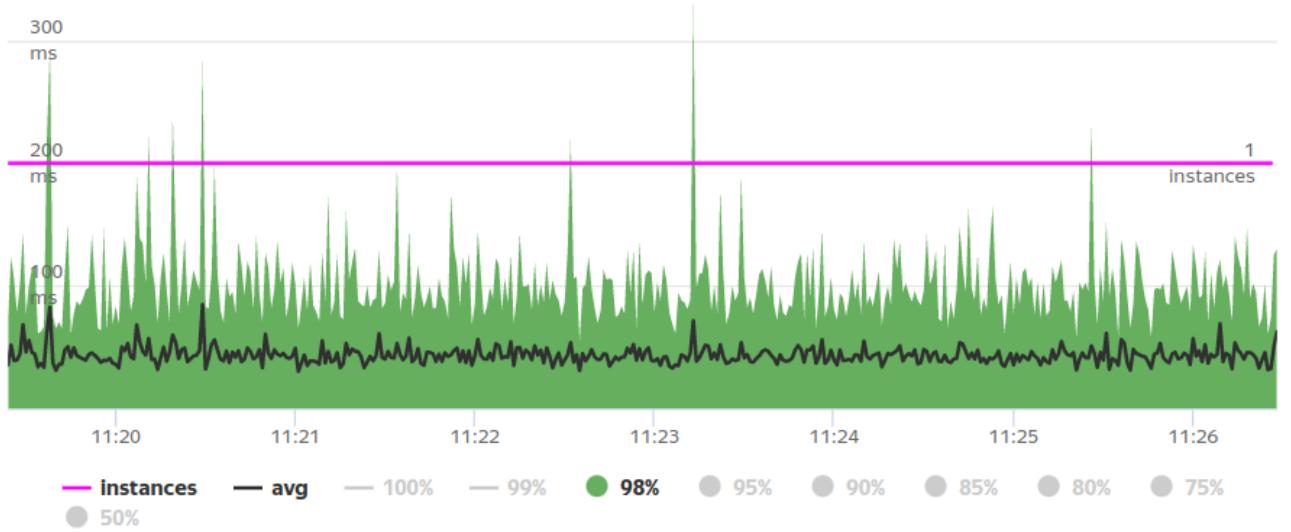


Рисунок 15 – Результат проведения нагрузочного тестирования при закрытой нагрузке без использования кеширования

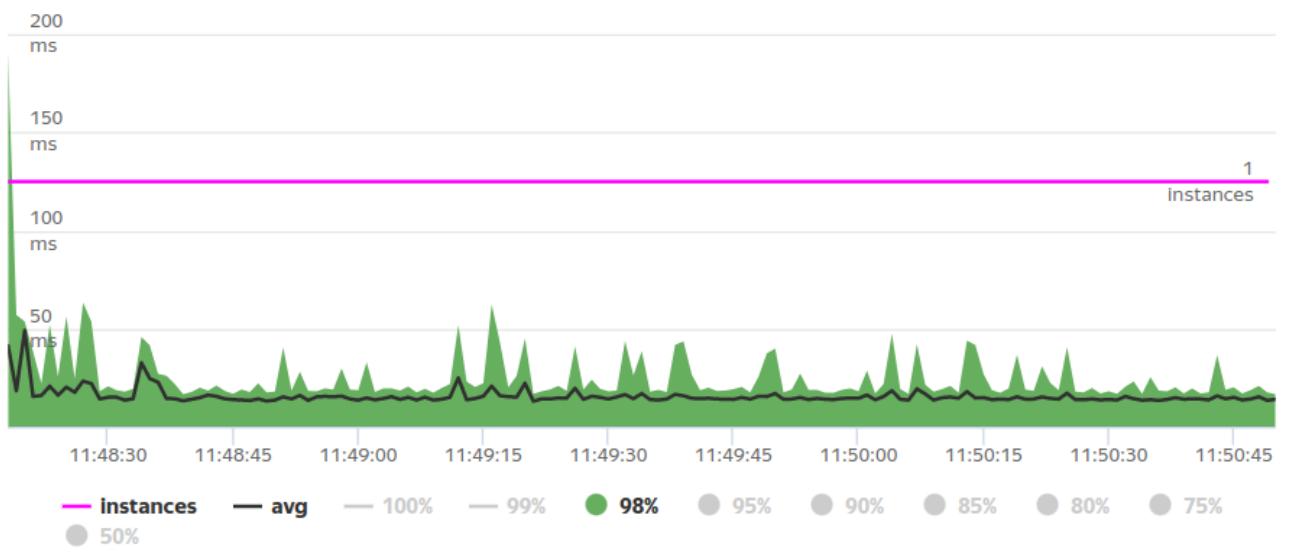


Рисунок 16 – Результат проведения нагрузочного тестирования при закрытой нагрузке с использованием кеширования

При использовании кеширования среднее время ответа сервера уменьшилось с 44 мс до 15 мс. Также уменьшился разброс времён ответов, то есть сервер стал отвечать на запросы стабильнее. Суммарное время обработки 10000 запросов от 1 пользователя уменьшилось с 6 минут 53 секунд до 2 минут 17 секунд, то есть на 66%.

### **Вывод из экспериментально-исследовательской части**

В данном разделе было проведено исследование влияния кеширования на производительность разработанного ПО.

Измерения показали, что кеширование позволило повысить производительность веб-сервера, а именно его конечной точки /api/v1/units. При тестировании с использованием открытой модели нагрузки среднее время ответа сервера уменьшилось на 41% при линейной нагрузке и 22% при постоянной нагрузке. При тестировании с использованием закрытой модели нагрузки среднее время ответа сервера уменьшилось на 66%. Также использование кеширования позволило устраниТЬ пиковые нагрузки на сервис и сделать его работу более стабильной.

## **Заключение**

В рамках курсового проекта была разработана система извлечения много-компонентных терминов и их переводных эквивалентов из параллельных научно-технических текстов.

В ходе выполнения данной работы были решены следующие задачи.

1. Проведён анализ предметной области и формализована задачу.
2. Спроектирована база данных и структура программного обеспечения.
3. Реализован интерфейс для доступа к базе данных.
4. Реализовано программное обеспечение, которое позволит пользователю создавать, получать и изменять сведения из разработанной базы данных.
5. Проведено исследование зависимости времени выполнения запросов от использования кеширования данных текущей сессии пользователя.

Были рассмотрены и проанализированы существующие модели данных и области их применения. С учётом требований к приложению была спроектирована база данных, а именно formalизованы сущности и их связи, выбрана СУБД и реализован интерфейс для доступа к данным.

Разработанный веб-сервер имеет задокументированное API и позволяет сохранять и редактировать единицы языка, а также получать информацию о них.

В ходе выполнения экспериментально-исследовательской части работы было установлено, что применение кеширования может существенно повысить производительность приложения не зависимо от типа и профиля нагрузки.

В дальнейшем разработанное приложение будет развиваться в рамках микросервисной архитектуры. В частности, планируется добавить микросервисы для автоматической разметки текстов по описанному алгоритму и для перемещения данных из кеша в основное хранилище. Также планируется добавить графический интерфейс, позволяющий вручную размечать тексты и сохранять полученные данные при помощи разработанного веб-сервера.

## **Список использованных источников**

1. Когаловский М.Р. Энциклопедия технологий баз данных. – Москва: Финансы и статистика, 2002. – 800 с.
2. Алферова Т.К., Леонов А.В., Филиппов К.В. О состоянии автоматизированной базы данных терминов и определений в области ОП // Компетентность. – 2014. – № 7(118). – С. 10-14.
3. Горбач Т.А., Грибова В.В., Окунь Д.Б., Петряева М.В., Шалфеева Е.А., Шахгельян К.И. База терминов нейрохирургии для интеллектуальной обработки биомедицинских данных // Сборник материалов XIII международной научной конференции: «Системный анализ в медицине». – Благовещенск, 2019. – С. 82-85.
4. Кузнецов И.О. Автоматическое извлечение двусловных терминов по тематике «Нанотехнологии в медицине» на основе корпусных данных // Научно-техническая информация. Сер. 2. – 2013. – № 5. – С. 25-33.
5. Becerro F. B. Phraseological variations in medical-pharmaceutical terminology and its applications for English and German into Spanish translations // SciMedicine Journal. – 2020. – № 2(1). – Р.22-29. DOI: 10.28991/SciMedJ-2020-0201-4.
6. Simon N. I., Kešelj V. August. Automatic term extraction in technical domain using part-of-speech and common-word features // Proceedings of the ACM Symposium on Document Engineering. – 2018. – Р. 1-4. DOI:10.1145/3209280.3229100.
7. Клышинский Э.С., Кочеткова Н.А., Карпик О.В. Метод выделения коллокаций с использованием степенного показателя в распределении Цифа // Новые информационные технологии в автоматизированных системах. – 2018. – № 21. – С. 220-225.

8. Кочеткова Н.А. Метод извлечения технических терминов с использованием усовершенствованной меры странности // Научно-техническая информация. Сер. 2.– 2015. –№ 5. – С. 25-32;
- Kochetkova N.A. A Method for Extracting Technical Terms Using the Modified Weirdness Measure // Automatic Documentation and Mathematical Linguistics. – 2015 – Vol. 49, № 3. – P. 89-95.
9. Захаров В.П., Хохлова М.В. Автоматическое извлечение терминов из специальных текстов с использованием дистрибутивно-статистического метода как инструмент создания тезаурусов // Структурная и прикладная лингвистика. – 2012. – № 9. – С. 222-233.
10. Terryn A., Hoste V., Lefever E. In no uncertain terms: a dataset for monolingual and multilingual automatic term extraction from comparable corpora // Language Resources and Evaluation. – 2020. – Vol. 54, № 2. – P. 385-418. DOI:10.1007/s10579-019-09453-9.
11. Бутенко Ю.И. Строганов Ю.В., Сапожков А.М. Метод извлечения русскоязычных многокомпонентных терминов в корпусе научно-технических текстов // Прикладная информатика. – 2021. – № 6. – С. 21-27. DOI: 10.37791/2687-0649-2021-16-6-21-27.
12. Бутенко Ю.И. Строганов Ю.В., Сапожков А.М. Система извлечения многокомпонентных терминов и их переводных эквивалентов из параллельных научно-технических текстов // НТИ. Сер. 2. ИНФОРМ. ПРОЦЕССЫ И СИСТЕМЫ. – 2022. – № 9. – С. 12-21. ISSN 0548-0027.
13. К. Дж. Дейт SQL и реляционная теория. Как грамотно писать код на SQL. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 480 с., ил. ISBN 978-5-93286-173-8

14. Дейт, К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом «Вильяме», 2005. — 1328 с.: ил. — Парал. тит. англ. ISBN 5-8459-0788-8 (рус.)
15. Маркин, А. В. Системы графовых баз данных. Neo4j : учебное пособие для вузов. — Москва : Издательство Юрайт, 2021. — 178 с. — (Высшее образование). ISBN 978-5-534-13996-9
16. Ньюмен, С. Создание микросервисов, 2-е издание.: Пер. с англ. — СПб.: Издательство «Питер», 2023. — 624 с.: ил. ISBN 978-5-4461-1145-9
17. gRPC: A high performance, open source universal RPC framework [Электронный ресурс]. — Режим доступа: <https://grpc.io/> (дата обращения: 27.03.2023).
18. Swagger: API Documentation & Design Tools for Teams [Электронный ресурс]. — Режим доступа: <https://swagger.io/> (дата обращения: 15.04.2023).
19. Roy Thomas Fielding Architectural Styles and the Design of Network-based Software Architectures [Электронный ресурс]. — Режим доступа: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> (дата обращения: 27.03.2023).
20. Иванова Е.В., Цымблер М.Л. Обзор современных систем обработки временных рядов // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2020. Т. 9, № 4. С. 79–97. DOI: 10.14529/cmse200406.
21. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — 448 с.: ил. — (Серия «Библиотека программиста»). ISBN 978-5-4461-1595-2

22. The Repository Pattern Explained [Электронный ресурс]. — Режим доступа: <https://blog.sapiensworks.com/post/2014/06/02/The-Repository-Pattern-For-Dummies.aspx> (дата обращения: 06.02.2023).
23. Seemann, M. and van Deursen, S. Dependency Injection. Principles, Practices, and Patterns. — Manning, 2019. — 552 с. ISBN 978-1-6172-9473-0
24. PostgreSQL: The World’s Most Advanced Open Source Relational Database [Электронный ресурс]. — Режим доступа: <https://www.postgresql.org/> (дата обращения: 15.04.2023).
25. Государственный реестр сертифицированных средств защиты информации [Электронный ресурс]. — Режим доступа: <https://fstec.ru/tekhnicheskaya-zashchita-informatsii/dokumenty-po-sertifikatsii/153-sistema-sertifikatsii/591-gosudarstvennyj-reestr-sertifikatsii-v-rossii-0001-01bi00> (дата обращения: 15.04.2023).
26. Redis. The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker. [Электронный ресурс]. — Режим доступа: <https://redis.io/> (дата обращения: 15.04.2023).
27. InfluxData: InfluxDB Times Series Data Platform [Электронный ресурс]. — Режим доступа: <https://www.influxdata.com/> (дата обращения: 15.04.2023).
28. O’Neil P., Cheng E., Gawlick D., O’Neil E. The log-structured merge-tree (LSM-tree) // Acta Informatica. 1996. Vol. 33. P. 351–385.
29. InfluxDB Query Language [Электронный ресурс]. — Режим доступа: [https://docs.influxdata.com/influxdb/v1.3/query\\_language/](https://docs.influxdata.com/influxdb/v1.3/query_language/) (дата обращения: 15.04.2023).

30. The Go Programming Language [Электронный ресурс]. — Режим доступа: <https://go.dev/> (дата обращения: 15.04.2023).
31. Gin Web Framework [Электронный ресурс]. — Режим доступа: <https://gin-gonic.com/> (дата обращения: 15.04.2023).
32. OpenAPI Specification - Version 3.0.3 - Swagger [Электронный ресурс]. — Режим доступа: <https://swagger.io/specification/> (дата обращения: 15.04.2023).
33. Docker: Accelerated, Containerized Application Development [Электронный ресурс]. — Режим доступа: <https://www.docker.com/> (дата обращения: 15.04.2023).
34. Docker Compose overview [Электронный ресурс]. — Режим доступа: <https://docs.docker.com/compose/> (дата обращения: 15.04.2023).
35. PL/pgSQL — SQL Procedural Language [Электронный ресурс]. — Режим доступа: <https://www.postgresql.org/docs/current/plpgsql.html> (дата обращения: 16.04.2023).
36. Swagger UI [Электронный ресурс]. — Режим доступа: <https://swagger.io/tools/swagger-ui> (дата обращения: 16.05.2023).
37. Yandex.Tank’s documentation [Электронный ресурс]. — Режим доступа: <https://yandextank.readthedocs.io/en/latest/#> (дата обращения: 14.05.2023).
38. Overload [Электронный ресурс]. — Режим доступа: <https://overload.yandex.net> (дата обращения: 14.05.2023).
39. Phantom [Электронный ресурс]. — Режим доступа: <https://cloud.yandex.com/en-ru/docs/load-testing/concepts/payloads/phantom> (дата обращения: 14.05.2023).

40. Manjaro [Электронный ресурс]. Режим доступа: <https://manjaro.org> (дата обращения: 14.05.2023).
41. Процессор Intel® Core™ i5-9300H [Электронный ресурс]. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/191075/intel-core-i59300h-processor-8m-cache-up-to-4-10-ghz.html> (дата обращения: 14.05.2023).

## **Приложение А Алгоритм извлечения многокомпонентных терминов**

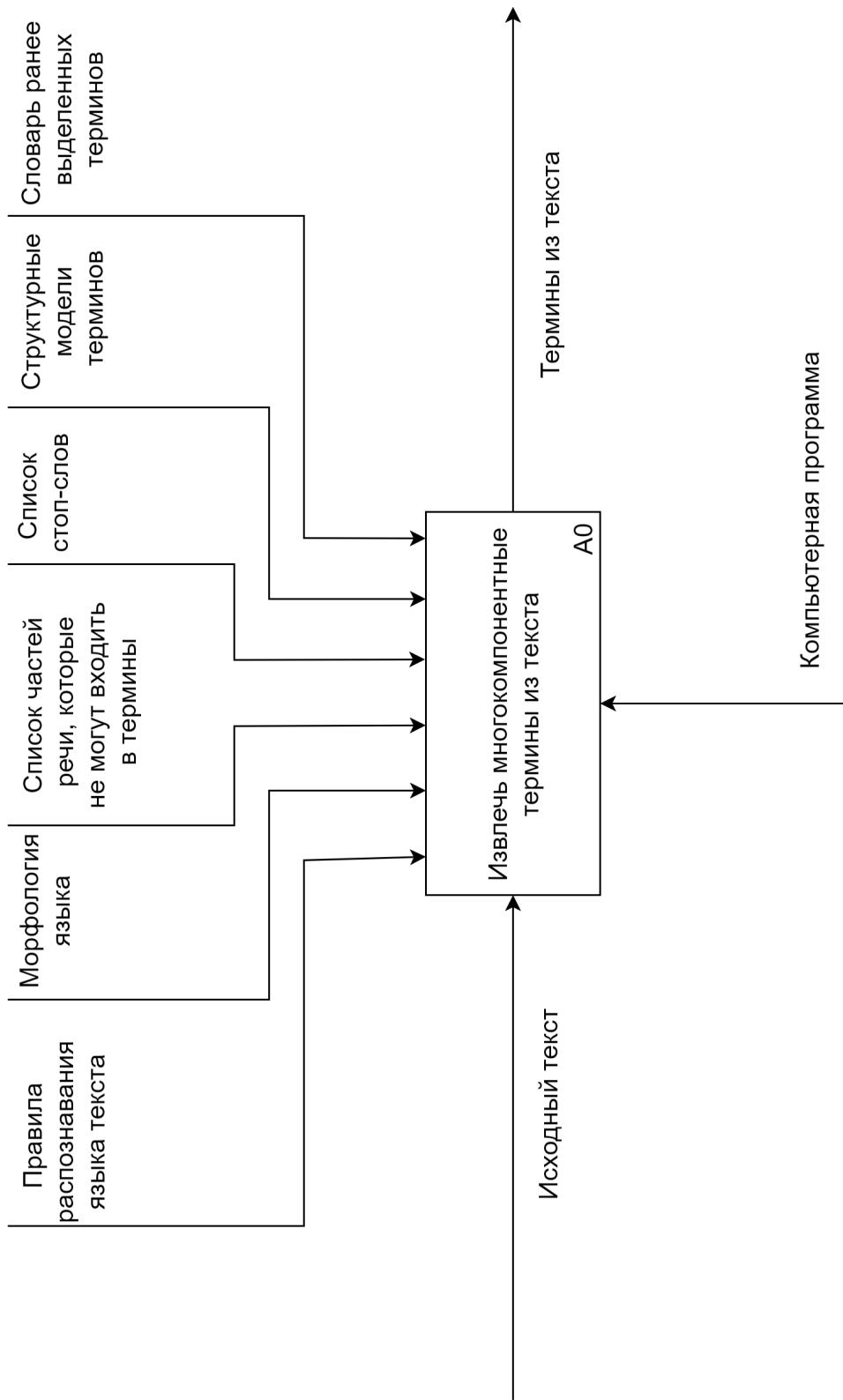


Рисунок 17 – Функциональная схема работы системы извлечения много компонентных терминов, верхний уровень

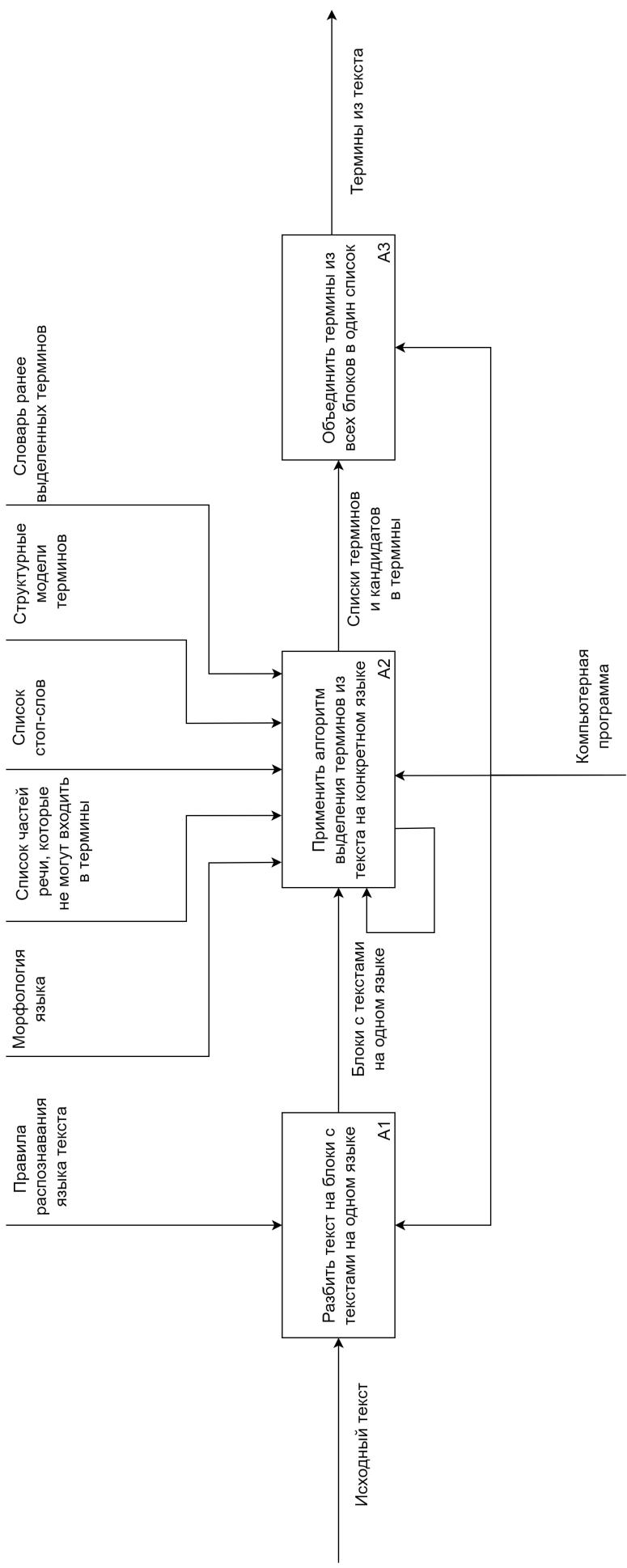


Рисунок 18 – Функциональная схема работы системы извлечения многокомпонентных терминов, декомпозиция уровня А0

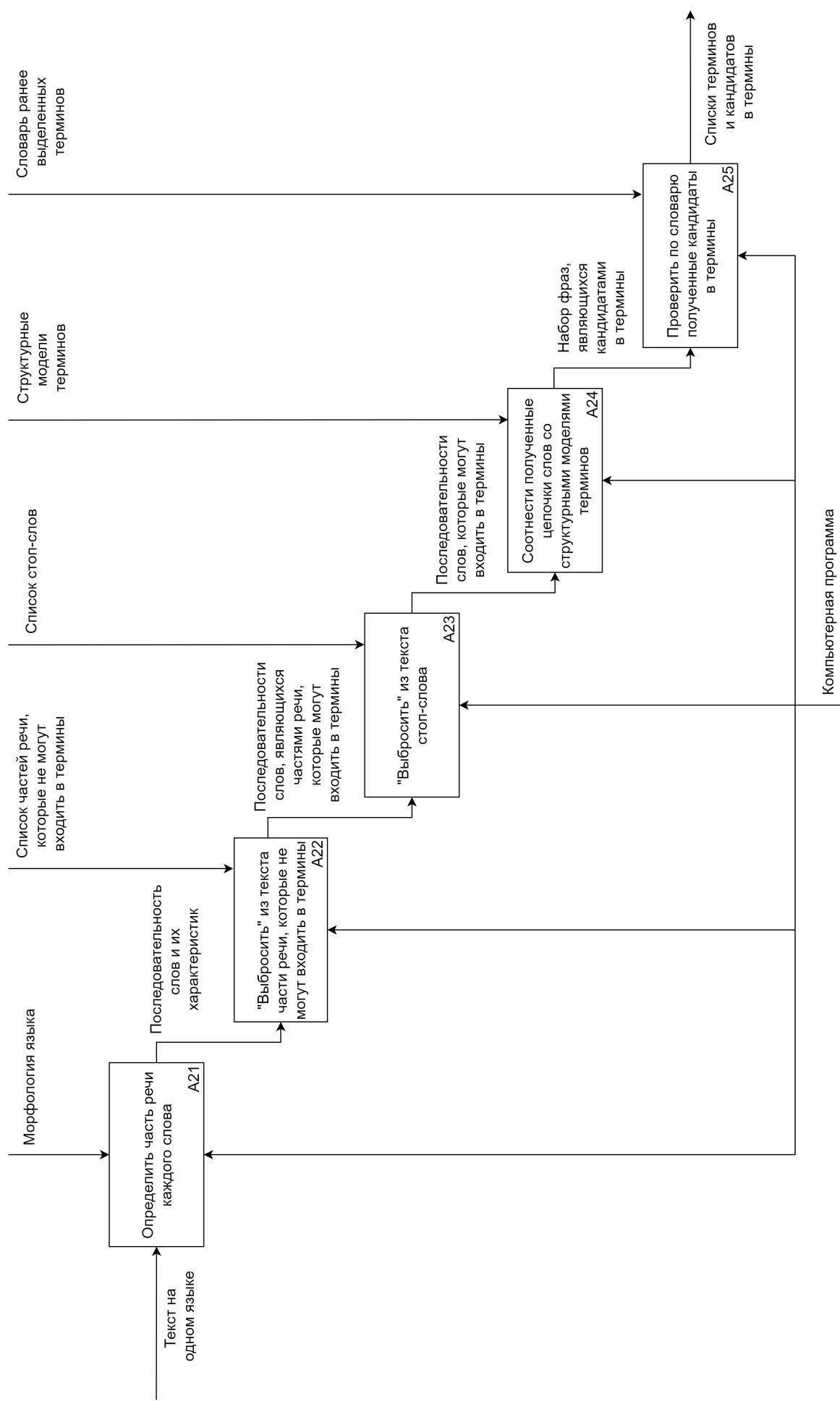


Рисунок 19 – Функциональная схема работы системы извлечения многокомпонентных терминов, декомпозиция уровня

## **Приложение Б Бизнес-сценарии**

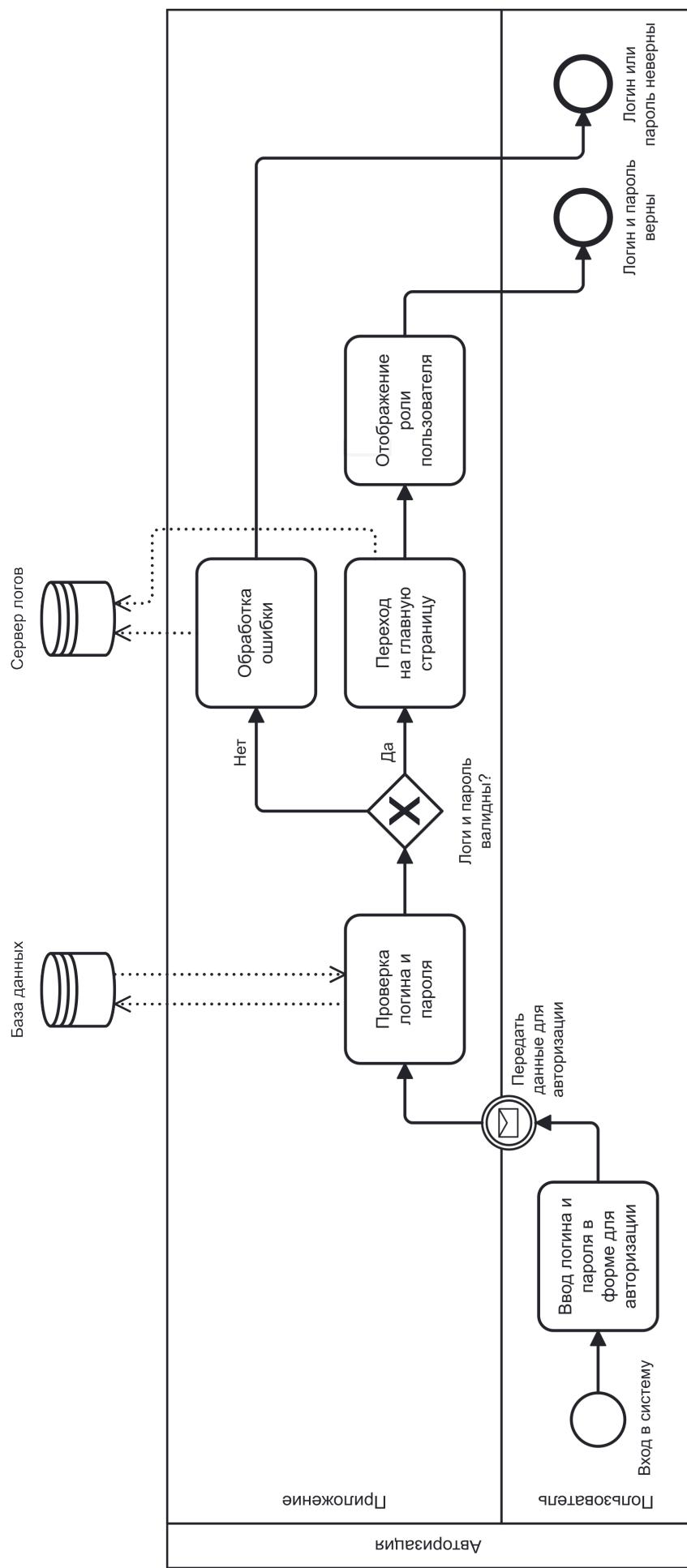


Рисунок 20 – Авторизация пользователя

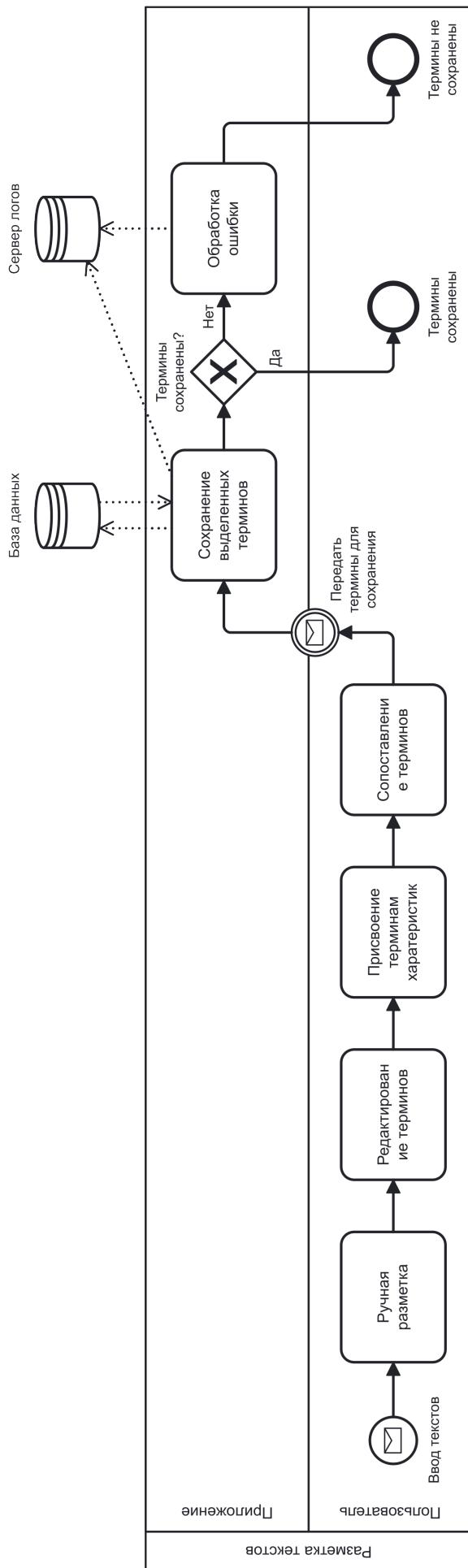


Рисунок 21 – Разметка текстов

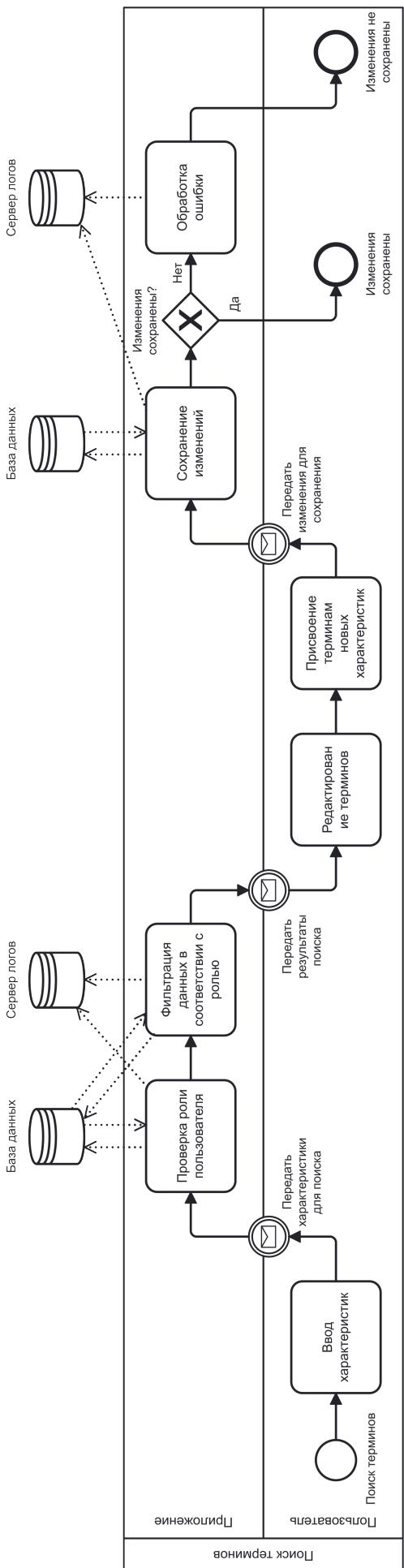


Рисунок 22 – Анализ терминов

## Приложение В Инициализация ролевой модели базы данных

Листинг 5: Скрипт инициализации ролевой модели базы данных для постоянных таблиц

```
1 create role student;
2 grant usage on schema public to student;
3 grant select on public.users to student;
4 grant select, insert, update on public.contexts to student;
5 grant select, insert, update, delete on public.properties to
   student;
6 grant select on information_schema.schemata to student;
7 grant select on information_schema.tables to student;
8
9 create role educator inherit;
10 grant student to educator;
11 grant delete on public.contexts to educator;
12 grant create on database :dbname to educator;
13 grant usage, create on schema public to educator;
14 grant references on all tables in schema public to educator;
15
16 create role admin with inherit CREATEDB CREATEROLE;
17 grant educator to admin;
18 alter database :dbname owner to admin;
19 grant usage, create on schema public to admin;
20 grant references, select, insert, update, delete on all tables in
   schema public to admin;
```

## Приложение Г Хранимая процедура базы данных

Листинг 6: Скрипт создания хранимой процедуры для добавления нового слоя

### разметки текстов

```
1 CREATE OR REPLACE PROCEDURE public.create_layer_tables(layer text)
2 AS
3 $func$
4 DECLARE layer_name text;
5 BEGIN
6     select (layer || '_layer') into layer_name;
7     EXECUTE format(
8         '-- Структурные() Модели слоя
9         create table if not exists %I.models(
10             id int generated always as identity primary key,
11             name text unique not null
12         );
13
14         -- Элементы слоя справочная( таблица)
15         -- drop table if exists %I.elements;
16         create table if not exists %I.elements(
17             id int generated always as identity primary key,
18             name text unique not null
19         );
20
21         -- Единицы русского языка
22         create table if not exists %I.units_ru(
23             id int generated always as identity primary key,
24             model_id int,
25             foreign key (model_id) references %I.models(id),
26             registration_date timestamp not null,
27             text text unique not null
28         );
29
30         -- Единицы иностранного языка
```

```

31      create table if not exists %I.units_en(
32          id int generated always as identity primary key,
33          model_id int,
34          foreign key (model_id) references %I.models(id),
35          registration_date timestamp not null,
36          text text unique not null
37      );
38
39      -- Таблица связка - модели ( слоя и элементы слоя)
40      -- drop table if exists %I.models_and_elem;
41      create table if not exists %I.models_and_elems(
42          model_id int,
43          foreign key (model_id) references %I.models(id),
44          elem_id int,
45          foreign key (elem_id) references %I.elements(id),
46          unique(model_id, elem_id)
47      );
48
49      -- Таблица связка - единицы ( русского языка и единицы
50      -- иностранного языка)
51      -- drop table if exists %I.units_ru_and_en;
52      create table if not exists %I.units_ru_and_en(
53          unit_ru_id int,
54          foreign key (unit_ru_id) references %I.units_ru(id)
55      ,
56          unit_en_id int,
57          foreign key (unit_en_id) references %I.units_en(id)
58      ,
59          unique(unit_ru_id, unit_en_id)
60      );
61
62      -- Таблица связка - характеристики ( и единицы русского
63      -- языка)
64      -- drop table if exists %I.properties_and_units_ru;
65      create table if not exists %I.properties_and_units_ru(

```

```

62         property_id int,
63             foreign key (property_id) references public.
64 properties(id),
65         unit_id int,
66             foreign key (unit_id) references %I.units_ru(id),
67             unique(property_id, unit_id)
68 );
69
70
71     -- Таблица связка - характеристики ( и единицы
72     -- иностранного языка)
73
74     -- drop table if exists %I.properties_and_units_en;
75     create table if not exists %I.properties_and_units_en(
76         property_id int,
77             foreign key (property_id) references public.
78 properties(id),
79         unit_id int,
80             foreign key (unit_id) references %I.units_en(id),
81             unique(property_id, unit_id)
82 );
83
84
85     -- Таблица связка - контексты ( и единицы русского языка)
86     -- drop table if exists %I.contexts_and_units_ru;
87     create table if not exists %I.contexts_and_units_ru(
88         context_id int,
89             foreign key (context_id) references public.contexts
90         (id),
91         unit_id int,
92             foreign key (unit_id) references %I.units_ru(id),
93             unique(context_id, unit_id)
94 );
95
96
97     -- Таблица связка - контексты ( и единицы иностранного
98     -- языка)
99     -- drop table if exists %I.contexts_and_units_en;
100    create table if not exists %I.contexts_and_units_en(

```

```

92         context_id int,
93             foreign key (context_id) references public.contexts
94             (id),
95             unit_id int,
96             foreign key (unit_id) references %I.units_en(id),
97             unique(context_id, unit_id)
98     );
99
100
101    -- Таблица связка - пользователи ( и единицы русского
102    языка)
103
104    -- drop table if exists %I.users_and_units_ru;
105    create table if not exists %I.users_and_units_ru(
106        user_id int,
107            foreign key (user_id) references public.users(id),
108            unit_id int,
109            foreign key (unit_id) references %I.units_ru(id),
110            unique(user_id, unit_id)
111    );
112
113
114    -- Таблица связка - пользователи ( и единицы иностранного
115    языка)
116
117    -- drop table if exists %I.users_and_units_en;
118    create table if not exists %I.users_and_units_en(
119        user_id int,
120            foreign key (user_id) references public.users(id),
121            unit_id int,
122            foreign key (unit_id) references %I.units_en(id),
123            unique(user_id, unit_id)
124    );
125
126    layer_name, layer_name, layer_name, layer_name,
127        layer_name,
128    layer_name, layer_name, layer_name, layer_name,
129        layer_name,
130    layer_name, layer_name, layer_name, layer_name,
131        layer_name,

```

```

121      layer_name, layer_name, layer_name, layer_name,
122          layer_name,
123      layer_name, layer_name, layer_name, layer_name,
124          layer_name,
125      layer_name, layer_name, layer_name
126  );
127 END
128
129 CREATE OR REPLACE PROCEDURE public.
130     grant_student_rights_to_layer_tables(layer text)
131 AS
132 $func$
133 DECLARE layer_name text;
134 BEGIN
135     select (layer || '_layer') into layer_name;
136     EXECUTE format(
137         'grant usage, create on schema %I to student;
138             grant select, insert, update on %I.units_ru to student;
139             grant select, insert, update on %I.units_en to student;
140                 grant select, insert, update, delete on %I.
141 properties_and_units_ru to student;
142                     grant select, insert, update, delete on %I.
143 properties_and_units_en to student;
144                         grant select, insert, update on %I.
145 contexts_and_units_ru to student;
146                             grant select, insert, update on %I.
147 contexts_and_units_en to student;
148                                 grant select, insert, update on %I.users_and_units_ru
149 to student;

```

```

145      grant select, insert, update on %I.users_and_units_en
146      to student;
147
148      grant select on %I.models to student;
149      grant select on %I.elements to student;
150      grant select on %I.models_and_elems to student;',
151      layer_name, layer_name, layer_name, layer_name,
152      layer_name,
153      layer_name, layer_name, layer_name, layer_name,
154      layer_name, layer_name, layer_name
155      );
156
157 END
158
159 $func$ LANGUAGE plpgsql;
160
161 CREATE OR REPLACE PROCEDURE public.
162
163     grant_educator_rights_to_layer_tables(layer text)
164
165 AS
166
167 $func$
168
169 DECLARE layer_name text;
170
171 BEGIN
172
173     select (layer || '_layer') into layer_name;
174
175     EXECUTE format(
176
177         'grant insert, update, delete on %I.models to educator;
178
179         grant insert, update, delete on %I.elements to educator
180
181         ;
182
183         grant insert, update, delete on %I.models_and_elems to
184         educator;',
185
186         layer_name, layer_name, layer_name
187
188     );
189
190 END
191
192 $func$ LANGUAGE plpgsql;
193
194
195 CREATE OR REPLACE PROCEDURE public.
196
197     grant_admin_rights_to_layer_tables(layer text)
198
199 AS

```

```

173 $func$  

174 DECLARE layer_name text;  

175 BEGIN  

176     select (layer || '_layer') into layer_name;  

177     EXECUTE format(  

178         'grant create on schema %I to admin;  

179             grant select, insert, update, delete on all tables in  

180             schema %I to admin;',  

181             layer_name, layer_name  

182     );  

183 END  

184 $func$ LANGUAGE plpgsql;  

185  

186 CREATE OR REPLACE PROCEDURE public.grant_rights_to_layer_tables(  

187     layer text)  

188 AS  

189 $func$  

190 BEGIN  

191     EXECUTE format(  

192         E'call public.grant_student_rights_to_layer_tables(\'%s  

193             \');  

194             call public.grant_educator_rights_to_layer_tables(\'%s\'';  

195             ');  

196             call public.grant_admin_rights_to_layer_tables(\'%s\'';  

197             ',  

198             layer, layer, layer  

199     );  

200 END  

201 $func$ LANGUAGE plpgsql;  

202  

203 CREATE OR REPLACE PROCEDURE public.create_layer(layer text)  

204 AS  

205 $func$  

206 BEGIN  

207     EXECUTE format(

```

```
203      E' set role educator;
204      create schema if not exists %I;
205      call public.create_layer_tables(\'%s\');
206      call public.grant_rights_to_layer_tables(\'%s\');',
207      (layer || '_layer'), layer, layer
208    );
209 END
210 $func$ LANGUAGE plpgsql;
```

## Приложение Д Сборка приложения

Листинг 7: Dockerfile для сервиса основного приложения

```
1 # syntax=docker/dockerfile:1
2
3 ## Build
4 FROM golang:1.20.2-alpine3.17 AS build
5 WORKDIR /app
6
7 # Install dependencies
8 COPY go.mod ./
9 COPY go.sum ./
10 RUN go mod download
11
12 # Copy source code
13 COPY ./cmd/backend/main.go ./
14 COPY ./internal ./internal
15 COPY ./pkg ./pkg
16 COPY ./swagger ./swagger
17
18 # Build the binary
19 RUN go build -o /backend
20
21 ## Deploy
22 FROM scratch
23
24 # Copy our static executable
25 COPY --from=build /backend /backend
26 COPY --from=build /app/swagger ./swagger
27 COPY backend.env /
28
29 EXPOSE ${BACKEND_PORT}
30 # USER nonroot:nonroot
31
```

```
32 # Run the binary  
33 ENTRYPOINT [ "/backend", "-env=backend.env" ]
```

## Приложение E Развёртывание приложения

Листинг 8: Конфигурация развертывания приложения

```
1 version: '3.8'
2
3 services:
4     influxdb: # TODO: configure dashboards
5         container_name: ${INFLUXDB_CONTAINER_NAME}
6         image: influxdb:2.6.1-alpine
7         environment:
8             DOCKER_INFLUXDB_INIT_MODE: setup
9             DOCKER_INFLUXDB_INIT_USERNAME: ${INFLUXDB_USERNAME}
10            DOCKER_INFLUXDB_INIT_PASSWORD: ${INFLUXDB_PASSWORD}
11            DOCKER_INFLUXDB_INIT_ORG: ${INFLUXDB_ORG}
12            DOCKER_INFLUXDB_INIT_BUCKET: default
13            DOCKER_INFLUXDB_INIT_ADMIN_TOKEN: ${INFLUXDB_TOKEN}
14         restart: always
15         healthcheck:
16             # test: "curl -f http://localhost:8086/ping"
17             test: ${INFLUXDB_SERVICE_HEALTHCHECK_CMD}
18             interval: 3s
19             timeout: 10s
20             retries: 5
21         ports:
22             - ${INFLUXDB_PORT}:${INFLUXDB_PORT}
23     postgres:
24         container_name: ${POSTGRES_CONTAINER_NAME}
25         image: postgres:15.2-alpine3.17
26         environment:
27             POSTGRES_USER: ${POSTGRES_USER}
28             POSTGRES_DB: ${POSTGRES_DBNAME}
29             POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
30             PGDATA: /data/postgres
31         restart: always
```

```

32    healthcheck:
33        test: [ "CMD-SHELL", "pg_isready -U postgres" ]
34        interval: 5s
35        timeout: 5s
36        retries: 5
37
38    volumes:
39        - postgres:/var/lib/postgresql/data
40
41    ports:
42        - ${POSTGRES_PORT}:${POSTGRES_PORT}
43
44    networks:
45        - persistent_bridge_network
46
47    depends_on:
48        - influxdb
49
50 pgadmin:
51     container_name: pgadmin4
52     image: dpage/pgadmin4
53     restart: always
54
55     environment:
56         PGADMIN_DEFAULT_EMAIL: admin@admin.com
57         PGADMIN_DEFAULT_PASSWORD: root
58
59     ports:
60         - "5050:80"
61
62     depends_on:
63         postgres:
64             condition: service_healthy
65
66 redis:
67     container_name: redis
68     image: redis:7.0.8-alpine3.17
69
70     environment:
71         REDIS_HOST: ${REDIS_HOST}
72         REDIS_PORT: ${REDIS_PORT}
73         REDIS_PASSWORD: ${REDIS_PASSWORD}
74
75     restart: always
76
77     healthcheck:
78         test: [ "CMD", "redis-cli", "ping" ]

```

```

67      interval: 5s
68      timeout: 5s
69      retries: 5
70
71      ports:
72          - ${REDIS_PORT}:${REDIS_PORT}
73
74      command: redis-server --save 20 1 --loglevel warning --
75          requirepass ${REDIS_PASSWORD}
76
77      volumes:
78          - redis:/data
79
80      depends_on:
81          postgres:
82              condition: service_healthy
83
84      backend:
85          container_name: backend
86
87          build:
88              context: ./
89              dockerfile: api.Dockerfile
90
91          image: golang-backend
92
93          restart: always
94
95          ports:
96              - ${BACKEND_PORT}:${BACKEND_PORT}
97
98          depends_on:
99              influxdb:
100                  condition: service_started
101
102          redis:
103              condition: service_healthy
104
105          postgres:
106              condition: service_healthy
107
108      volumes:
109          redis:
110              driver: local
111
112          postgres:
113              driver: local
114
115      networks:
116
117          persistent_bridge_network:

```

101

driver: bridge

## **Приложение Ж Презентация**