|  |  |
| --- | --- |
|  | **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 «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОЙ РАБОТЕ НА ТЕМУ:***

***«Разработка базы данных для авторской игры в слова»***

Студент \_ИУ7-65Б\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_**А. М. Тагилов**\_\_\_\_**

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_**Л. Л. Волкова**\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

*2022 г.*



**РЕФЕРАТ**

Расчетно­-пояснительная записка 42 с., 8 рис., 1 табл., 12 ист., 1 прил. В работе представлена разработка и описание программного обеспечения с базой данных для хранения и изменения карточек и категорий для авторской игры в слова.

Ключевые слова: базы данных, версионирование, backend, серверное приложение, авторизация, контейнеризация.

Проведен анализ существующих решений, подходов к хранению данных с версионированием. Разработана база данных с распределением по ролям и возможностью хранения авторизационных данных. Реализовано серверное приложение на языке Golang для предоставления удобного доступа к базе данных, в приложении также предусмотрена возможность создания пользователей и авторизации. Для развертывания приложения описана система контейнеризации. Проведено исследование времени генерации строк в формате JSON на стороне базы данных и приложения.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc113315747)

[1 Аналитический раздел 4](#_Toc113315748)

[Требования к разрабатываемому ПО 4](#_Toc113315749)

[Выбор базы данных 6](#_Toc113315750)

[Типы баз данных 6](#_Toc113315751)

[Анализ существующих решений 7](#_Toc113315752)

[Логическая модель базы 8](#_Toc113315753)

[2 Конструкторский раздел 10](#_Toc113315754)

[Ролевая модель базы данных 10](#_Toc113315755)

[Структура хранения пользователей 11](#_Toc113315756)

[Структура оригинальной базы 12](#_Toc113315757)

[Структура ревизий 12](#_Toc113315758)

[Изменение и применение ревизий 13](#_Toc113315759)

[Диаграмма компонентов 14](#_Toc113315760)

[Вывод 15](#_Toc113315761)

[3 Технологический раздел 16](#_Toc113315762)

[Программный веб-интерфейс 16](#_Toc113315763)

[Развертывание приложения 16](#_Toc113315764)

[Работа пользователей с приложением 17](#_Toc113315765)

[Авторизация для зарегистрированных пользователей 18](#_Toc113315766)

[Работа редакторов и администраторов с приложением 19](#_Toc113315767)

[Вывод 21](#_Toc113315768)

[4 Исследовательская часть 22](#_Toc113315769)

[Технические характеристики 22](#_Toc113315770)

[Сравнительный анализ работы со строками 22](#_Toc113315771)

[Вывод 24](#_Toc113315772)

[ЗАКЛЮЧЕНИЕ 25](#_Toc113315773)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 26](#_Toc113315774)

[ПРИЛОЖЕНИЕ А 28](#_Toc113315775)

ВВЕДЕНИЕ

Существует немало игр для компаний на основе отгадывания слов: Крокодил, Alias, Шляпа, Activity и другие. Слова для игры обычно написаны на картонных карточках, поэтому далее под словом карточка понимается слово или набор слов для отгадывания.

Безусловно, для хранения набора карточек существует множество решений. Но было бы удобно предлагать пользователю не только набор карточек для игры, но и также предоставить выбор их категории. Чтобы была возможность редактировать набор слов и категорий на стороне модерации игры, необходимо предусмотреть версионирование изменений. Данный функционал может предложить система контроля версий, но для ее использования необходимо иметь немало навыков работы, например, с VCS Git [1].

Цель курсовой работы – разработать базу данных, содержащую информацию о карточках и категориях, позволяющую модерации создание, редактирование и применение ревизий, содержащих в себе набор изменений актуального состояния базы, а также разработать ПО для работы с ней.

Для достижения цели необходимо выполнить следующие задачи:

1. определить требования к разрабатываемому ПО;
2. провести анализ основных видов баз данных, выбрать наиболее подходящий;
3. спроектировать базу данных категорий и карточек с учетом создания ревизий;
4. реализовать базу данных;
5. разработать ПО для работы с базой данных,
6. исследовать производительность генерации строк с помощью базы данных и клиента.

# Аналитический раздел

## Требования к разрабатываемому ПО

Для обращения к базе данных по ролям необходимо предусмотреть возможность обращения к базе данных с разными правами доступа.

Для пользователей доступ предоставляется к просмотру категорий и карточек, идентификатора версии базы, а также к функциям проверки сессии и авторизации для получения сессия.

Для модераторов – к созданию, редактированию и применению своих ревизий. Ревизия представляет из себя набор изменений к текущему состоянию базы, сама база изменяется только после применения ревизии. Ревизия представляет из себя набор категорий и карточек.

Для администраторов – к созданию, редактированию и применению всех ревизий, созданию пользователей.

Необходимо найти подход к разрешению конфликтов в ревизиях разных модераторов, либо к избеганию таковых, это критерий предсказуемой работы с базой.

Для корректной работы ПО необходимо атомарно применять ревизии к текущей базе, а также изменять версию базы: применение любой непустой ревизии должно приводить к изменению версии. Версия может представлять из себя число, либо набор символов и всякий раз должна быть уникальной.

Разные категории могут включать в себя одинаковые карточки, но одна категория не может включать таковые.

Для удобства использования можно предусмотреть запрос на получение и категорий, и карточек ревизии одним запросом.

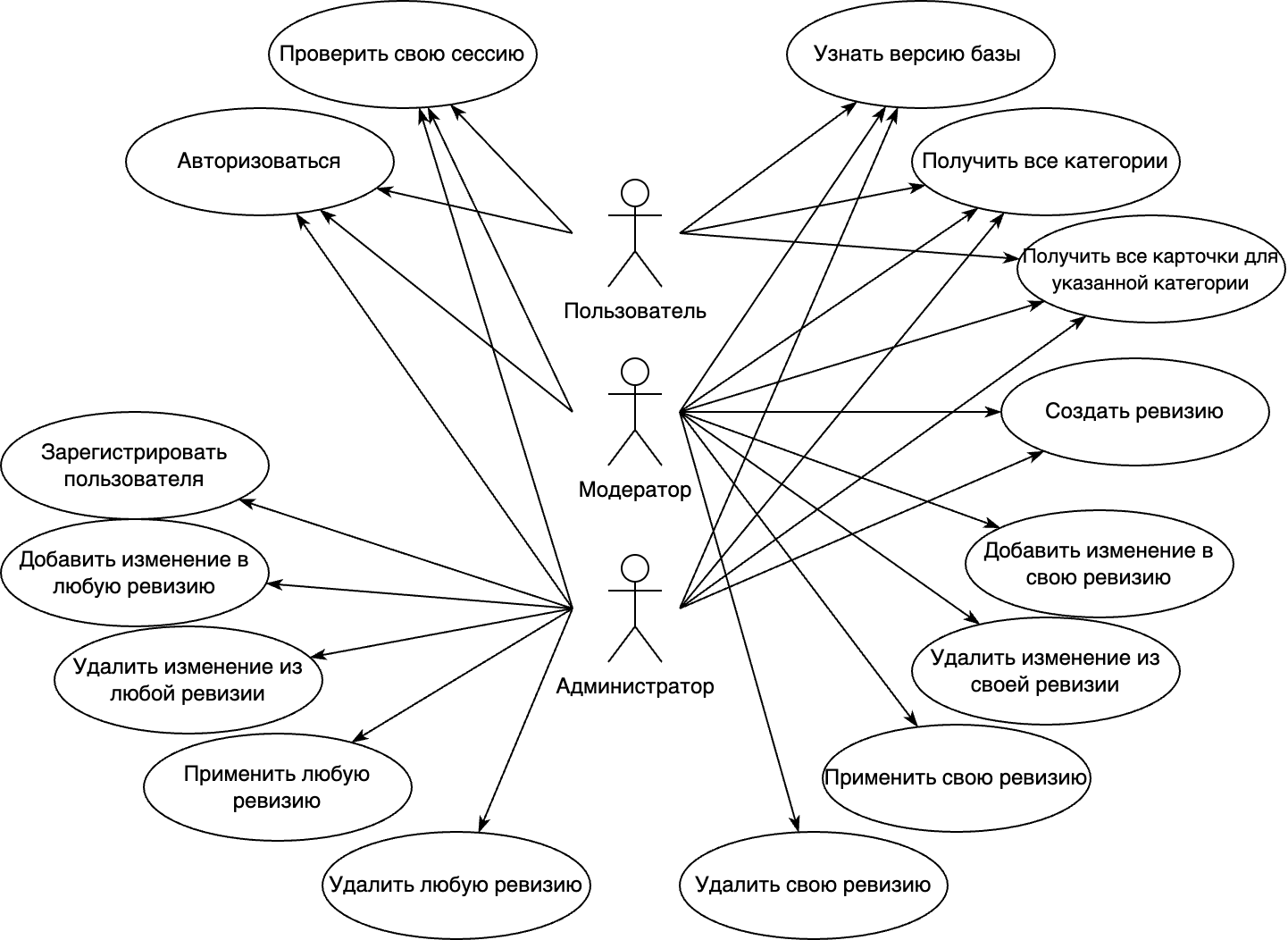


Рисунок 1 – диаграмма прецедентов.

Далее требования будут разделены по приоритетам от меньшего к большему: необходимые, желаемые и возможные.

Необходимые требования:

1. ролевая модель на уровне базы данных;
2. возможность выполнения типовых запросов для получения данных о текущем состоянии базы, а также запросов для работы с ревизиями;
3. инструмент разрешения, либо недопущения конфликтов в ревизиях;
4. серверное приложение с доступом по клиент-серверному протоколу для пользователей.

Желаемые требования:

1. серверное приложение с доступом по клиент-серверному протоколу для модераторов и администраторов.

Возможные требования:

1. серверное приложение для работы модераторов с доступом через сервис для мгновенного обмена сообщениями;
2. ПО с графическим интерфейсом для пользователей;
3. ПО с графическим интерфейсом для модераторов и администраторов.

## Выбор базы данных

### Типы баз данных

Концептуально базы данных разделяются на дореляционные, реляционные и постреляционные [2]. Дореляционные основаны на инвертированных списках и сетевых моделях, а также почти не используются, поэтому к сравнению представлены не будут.

Данные в реляционной модели представляют набор сущностей-таблиц, связанных между собой, что гарантирует целостность БД. Обработка происходит при помощи реляционной алгебры или реляционного исчисления. Преимущества: связанность сущностей гарантирует целостность, поддержан механизм транзакций, существуют индексы и другие механизмы оптимизации работы с БД, универсальный язык запросов SQL [3]. Недостатки: низкая скорость доступа к сравнительно большим базам ввиду необходимости проверок всех ограничений.

Постреляционные БД созданы с целую упростить масштабируемость базы и ускорить работу с данными [5]. Целостность данных поддержать все еще можно, но БД не всегда сможет гарантировать это. Принято выделять основные подходы к реализации постреляционных баз данных:

1. представление данных в виде графа;
2. ключ-значение;
3. семейство столбцов и другие.

К преимуществам можно отнести: скорость работы за счет отсутствия проверок целостности на этапе выполнения запроса, возможность хранения неструктурированных данных. К недостаткам – собственный API для взаимодействия, отсутствие гарантий целостности данных.

**Вывод**

В результате сравнения реляционных и постреляционных баз данных, был выбран тип реляционных БД по следующим причинам:

1. гарантия целостности данных;
2. существование четкой и заранее определенной структуры данных;
3. оптимизации на основе индексов;
4. универсальный язык запросов SQL.

## Анализ существующих решений

Для сравнительного анализа было была выбрана система контроля версий [4] с файлом для хранения информации в структурированном формате. Данные можно хранить в файле, а текущую версию файла размещать с помощью системы контроля версий на сервере. Преимущества данного решения: возможна реализация всех запланированных для ПО команд почти «из коробки». Недостатки: сложно реализовать разделение по ролям и ограничить права доступа, такое хранение данных не гарантирует целостность (можно создать ПО для ее гарантии), сложное управление командами для пользователя, не знакомого с Git.

Также можно сравнить с онлайн-таблицами. Преимущества: простота реализации таблиц карточек и категорий, удобный интерфейс. Недостатки: сложно реализовать разделение прав доступа по ролям, обеспечить автоматическое создание ревизий по запросу модератора, а также применение этих ревизий.

Сравнение существующих решений представлено на таблице 1.

Таблица 1 – сравнение существующих решений.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Система контроля версий | Онлайн-таблица | БД |
| Четкая ролевая модель | Нет | Нет | Да |
| История изменений | Да | Да | Нет |
| Создание ревизий | Да | Нет | Да |
| Гарантия целостности | Нет | Нет | Да |
| Простота управления | Наименьшая | Наибольшая | Средняя |

## Логическая модель базы

На рисунке 1 отображена логическая модель БД в виде ER-диаграммы.

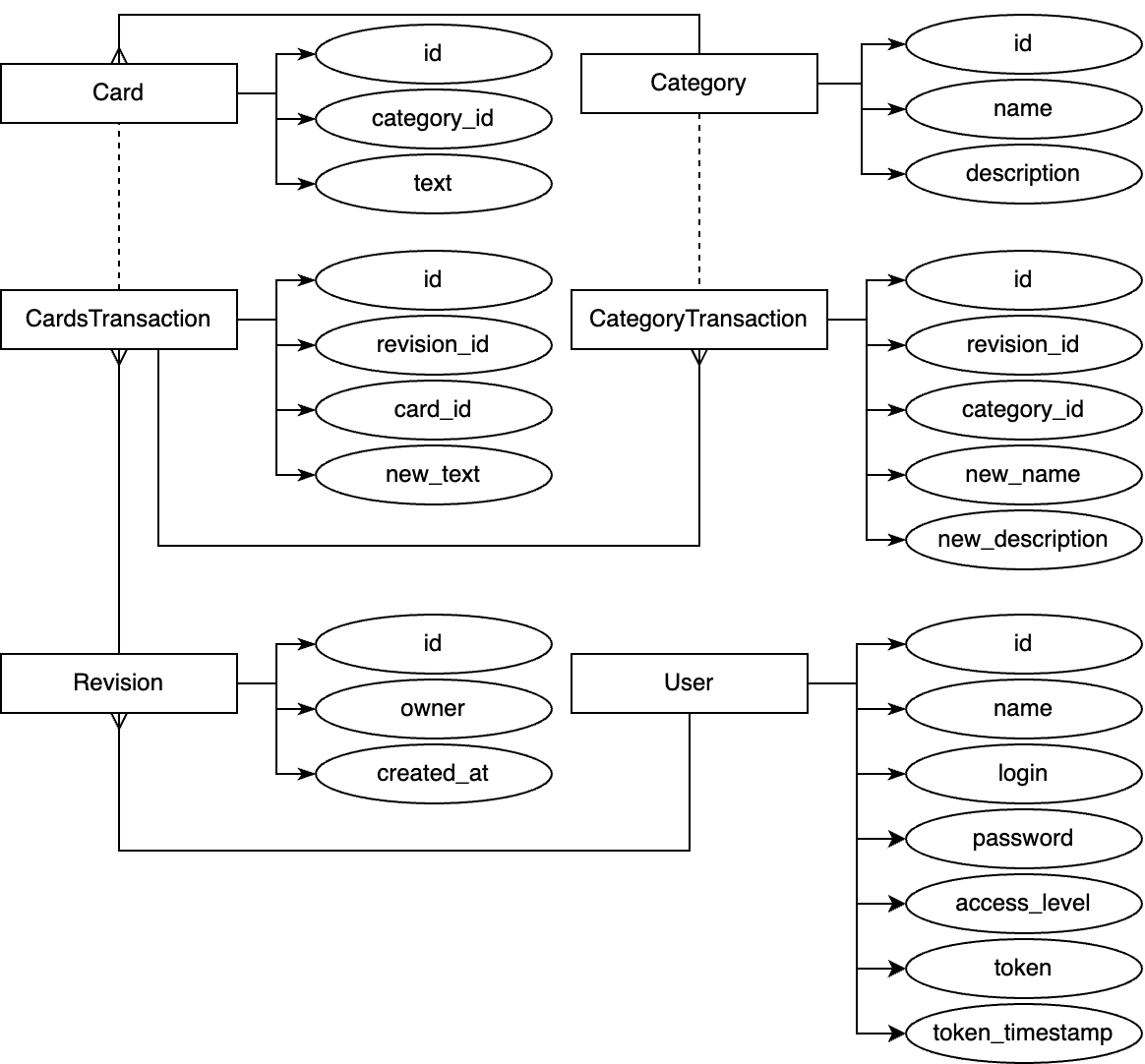


Рисунок 2 – диаграмма сущность-связь базы.

Для связей определены следующие ограничения:

1. для одной карточки/категории может быть определена только одна транзакция, либо не определена вовсе;
2. в рамках транзакции создания карточки/категории, поле card\_id/category\_id не указывается;
3. в рамках транзакции удаления карточки/категории, поля изменений не указываются.

**Вывод**

В данном разделе была проанализирована и формализована задача, а также рассмотрены способы ее реализации. Были рассмотрены и сравнены аналоги. Были описаны основные требования и ограничения к разрабатываемой базе данных.

# Конструкторский раздел

В конструкторском разделе будет проведено проектирование базы данных, приведен подход к хранению в БД паролей и авторизации, выделению ролей на уровне базы данных, а также реализованы все необходимые объекты базы данных: генераторы последовательностей, функции, процедуры и представления.

Большая часть функционала БД, который использует серверное приложение хранится в функциях и хранимых процедурах. Далее, все функции или хранимые процедуры с префиксом *cw* были реализованы в рамках данной работы на уровне базы данных.

## Ролевая модель базы данных

Для обеспечения безопасности работы с данными предусмотрена ролевая модель. Всего используется две роли.

1. Пользователь *watcher\_user*. Доступ к текущему состоянию категорий и карт, просмотру версии базы, авторизации, проверки уровня доступа. Не имеет доступ к ревизиям. Для обеспечения авторизации без предоставления доступа к таблице пользователей используются функции *cw.auth* и *cw.session*.
2. Редактор *editor\_user*. Изменение ревизий, в том числе:
   1. создание ревизии;
   2. удаление ревизии;
   3. применение ревизии;
   4. изменение категорий в ревизии, в том числе:
      1. создание категорий;
      2. редактирование категорий;
      3. удаление категорий;
      4. изменение карт в категории, в том числе:
         1. создание карт;
         2. удаление карт;
         3. редактирование карт.

Для гарантии целостности данных, большая часть изменений ревизий происходит с помощью фунций или хранимых процедур. Разделение доступа модераторов и администраторов происходит на уровне серверного приложения с использованием роли редактора.

Сценарий создания таблиц, наложения на них ограничений и распределения ролей представлен в приложении А.

## Структура хранения пользователей

У каждого пользователя есть логин – уникальный идентификатор пользователя и пароль. На основе этой пары можно получить так называемый ключ сессии: для этого нужно использовать функцию авторизации – *cw.auth*. Ключ сессии позволяет производить запросы на сервер, для которых требуется авторизация: действия модераторов и администраторов, для обычных пользователей не требуется ни регистрация, ни авторизация. Данный ключ действует на протяжении 24 часов, а затем срок его действия истекает, это сделано для защиты от утечек, реализовано с помощью хранения даты и времени успешного обращения к функции авторизации. После истечения срока действия нужно снова обратиться к функции авторизации и получить новый ключ. Схема алгоритма проверки ключа *cw.auth* отображена на рисунке 3.

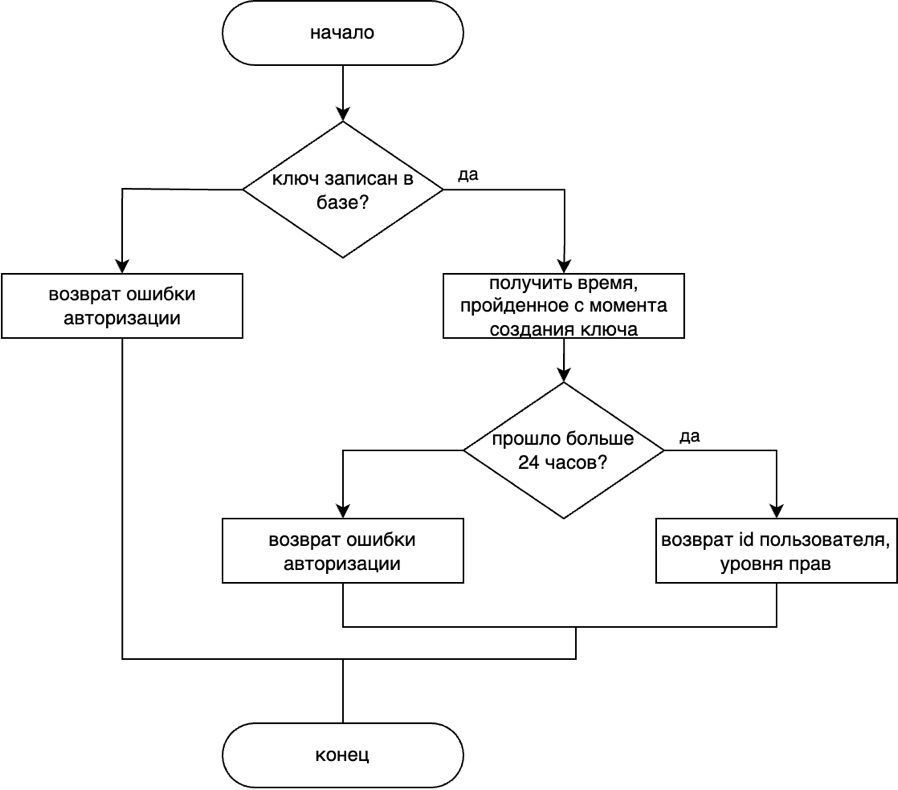


Рисунок 3 – Схема проверки ключа пользователя.

Пароли не стоит хранить в БД в сыром виде – это распространенная уязвимость безопасности согласно отчетам сообщества интернет-безопасности OWASP [6]. Поэтому для их хранения используется алгоритм блочного шифрования Blowfish [7, 8]. Данный алгоритм позволяет хранить пароли в зашифрованном виде, а также получать для одинаковых паролей разные ключи: именно поэтому алгоритм требует обратного преобразования (дешифровки) для проверки соответствия пароля. Для использования алгоритма используется расширение *pgcrypto* из стандартной библиотеки PostgreSQL.

Регистрация пользователя происходит с помощью функции *cw.register*, пароль сохраняется в зашифрованном виде. Данная функция доступна только для пользователей с правами администратора. При попытке входа с помощью функции *cw.auth* пароли сравниваются с помощью алгоритма дешифровки. Листинг функции авторизации и других вспомогательных функций приведен в приложении А.

## Структура оригинальной базы

Оригинальная база – таблицы карточек и категорий, доступных для пользователя любой роли. У каждой карточки есть своя категория. Таблицы соответственно *cw.card* и *cw.category.*

Для создания уникальных идентификаторов категорий используется генератор последовательностей *cw.category\_id\_seq*: при обращении к этому генератору каждый раз создается целочисленное значение, большее на 1 предыдущего. Таким образом, каждая новая категория получает уникальный идентификатор. Таким же образом, но с другим генератором – *cw.card\_id\_seq*, генерируются идентификаторы карточек.

Каждая карточка включает в себя текст, а категория – название и описание.

## Структура ревизий

Ревизия – изменение оригинальной базы. Хранится в таблице *cw.revision.* У ревизии есть имя, идентификатор пользователя-создателя, дата и время создания. Для ревизии можно создать изменение категории, которое хранится в таблице *cw.categoryT*. Для получения ревизии со всеми категориями и карточками есть функция *cw.revision\_json*, которая возвращает все данные о ревизии в формате JSON. Для этой функции используется представление *cw.revision\_categories\_with\_cards*, которое позволяет просматривать категории с карточками в формате JSON, листинг функции и представления представлен в приложении А.

Для обеспечения доступа модераторов существует функция принадлежности ревизии тому или иному модератору – *cw.owns*.

Если изменение категории применяется к оригинальной базе без названия категории и описания, то оригинальная категория удаляется вместе со всеми картами. Если одно из полей пустое, то оно не изменяется. У каждого изменения есть идентификатор категории из таблицы оригинальных категорий. Чтобы создать категорию, используется генератор идентификаторов *cw.category\_id\_seq*. Таким образом получается исключить возникновение конфликтов при создании категории.

Для карточек сценарии похожи: если поле нового имени пустое, то карточка удаляется. Для создания карточки используется генератор идентификаторов карточек *cw.card\_id\_seq*.

## Изменение и применение ревизий

Ревизии нужны для того, чтобы разом применять любое количество изменений над категориями и карточками. Они позволяют не допускать неконсистентности в момент изменения оригинальной базы редакторами. Большинство изменений ревизий производятся с помощью функций в БД. На каждую функцию наложены ограничения, чтобы не допускать некорректных изменений.

Функции БД для ревизий:

1. *cw.revision\_apply* – применение ревизии, не обладает особыми ограчениями, так как использование остальных функций не позволяет приводить ревизию в неконсисистентное состояние;
2. *cw.remove\_revision* – удаление ревизии;
3. для создания ревизий особых ограничений нет, поэтому оно происходит вручную – при помощи добавления строки в таблицу.

Функции БД для категорий, также представлены в приложении А:

1. *cw.add\_category –* добавление категории, запрещает создание категорий с одинаковыми именами;
2. *cw.edit\_category –* редактирование существующей категории, либо редактирование существующего изменения категории, запрещает изменение имени категории на дубликат, установку пустых полей для несуществующей в оригинальной базе категорий;
3. *cw.remove\_category\_edition* – удаление изменения категории, запрещает удаление несуществуществующей категории.

Функции БД для карточек*:*

1. *cw.add\_card* – добавление карточки, запрещает создание карточек с одинаковым названием в рамках одной категории,
2. *cw.edit\_card* – редактирование существующей карточки, либо редактирование существующего изменения карточки, запрещает изменять одну карточку в разных ревизиях.
3. *cw.remove\_card\_edition –* удаление изменения карточки, запрещает удаление несуществующей карточки.

## Диаграмма компонентов

На рисунке 4 изображена диаграмма компонентов.

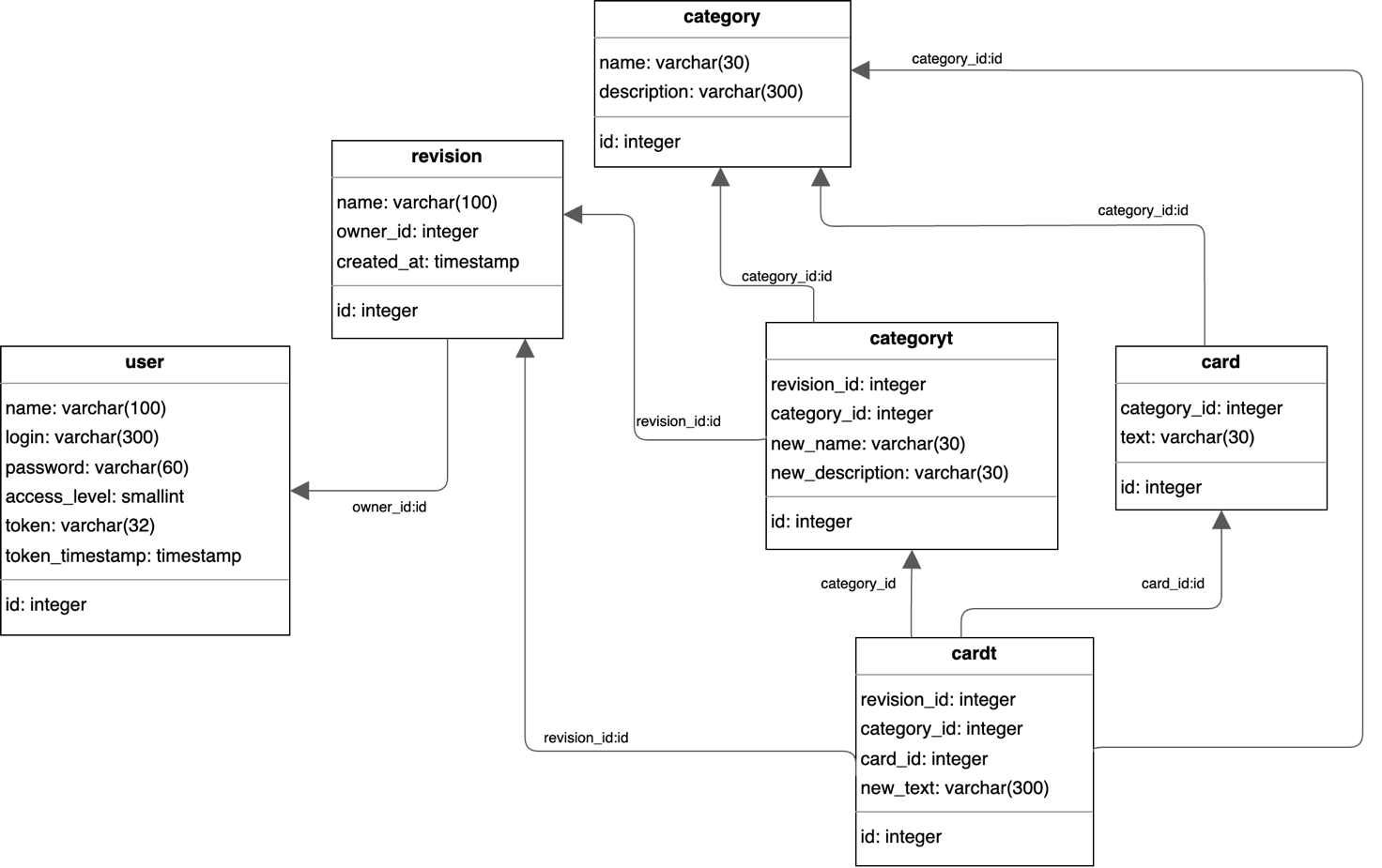


Рисунок 4 – диаграмма компонентов базы данных.

## Вывод

В данном разделе была рассмотрена и разработана схема базы данных, рассмотрены и описаны необходимые ограничения. Были приведены все необходимые объекты базы данных, а также было произведено распределение ролей базы данных.

# Технологический раздел

В технологическом разделе будет рассмотрено серверное веб-приложение, приведен подход к обработке запросов, подключению к базе данных, рассмотрен программный интерфейс приложения. Также будет рассмотрена конфигурация развертывания приложения и контейнеризации.

## Программный веб-интерфейс

Для доступа к базе данных была разработана программа для клиент-серверного взаимодействия. Доступ предоставляется посредством Rest API на основе протокола HTTP [9, 10]. В качестве языка для разработки был выбран язык Golang, его основные преимущества: он является кроссплатформенным, в языке используется строгая статическая типизация [11]. Листинг регистрации одного из обработчиков в приложении представлен в приложении А.

На основе разработанного серверного приложения можно создать как веб-приложение для пользователей веб-браузеров, так и клиентские мобильные приложения под разные платформы, например iOS или Android.

## Развертывание приложения

Для развертывания приложения и базы данных на сервере был использован инструмент контейнеризации Docker. Он позволяет запускать несколько контейнеров на одной машине, управлять жизненным циклом приложения, автоматизировать запуск, развертывание и восстановление, а также обеспечить безопасность изоляции контейнеров [12].

Для начала работы достаточно и выбрать конфигурационный файл и написать команду *docker-compose up*. В моем случае конфигурационный файл запускает два сервиса в одной сети: БД и приложение. У каждого сервиса есть свой конфигурационный файл, описывающий необходимые для работы приложения директории, переменные виртуальной среды, запускаемые файлы. Листинг конфигурационного файла приведен в приложении А.

Подключение к базе данных извне контейнера было решено ограничить для обеспечения безопасности, поэтому конфигурация позволяет подключаться извне только к приложению.

## Работа пользователей с приложением

Для пользователей предусмотрен следующий набор запросов.

1. GET-запрос */version* – получение текущей версии оригинальной базы. Ничего не принимает, возвращает версию в виде строки.
2. GET-запрос */categories* – получение всех категорий из оригинальной базы. Ничего не принимает, возвращает набор категорий в формате JSON. Для каждой категории возвращается уникальный идентификатор, имя и описание. На рисунке 5 изображен пример ответа, в приложении А приведен листинг функции-обработчика.



Рисунок 5 – пример возврата категорий.

1. GET-запрос */cards* – получение карт для выбранной категории. Выбор происходит с помощью передачи JSON с одним полем – category, требуется передать уникальный идентификатор. На рисунке 6 изображен пример запроса и ответа.

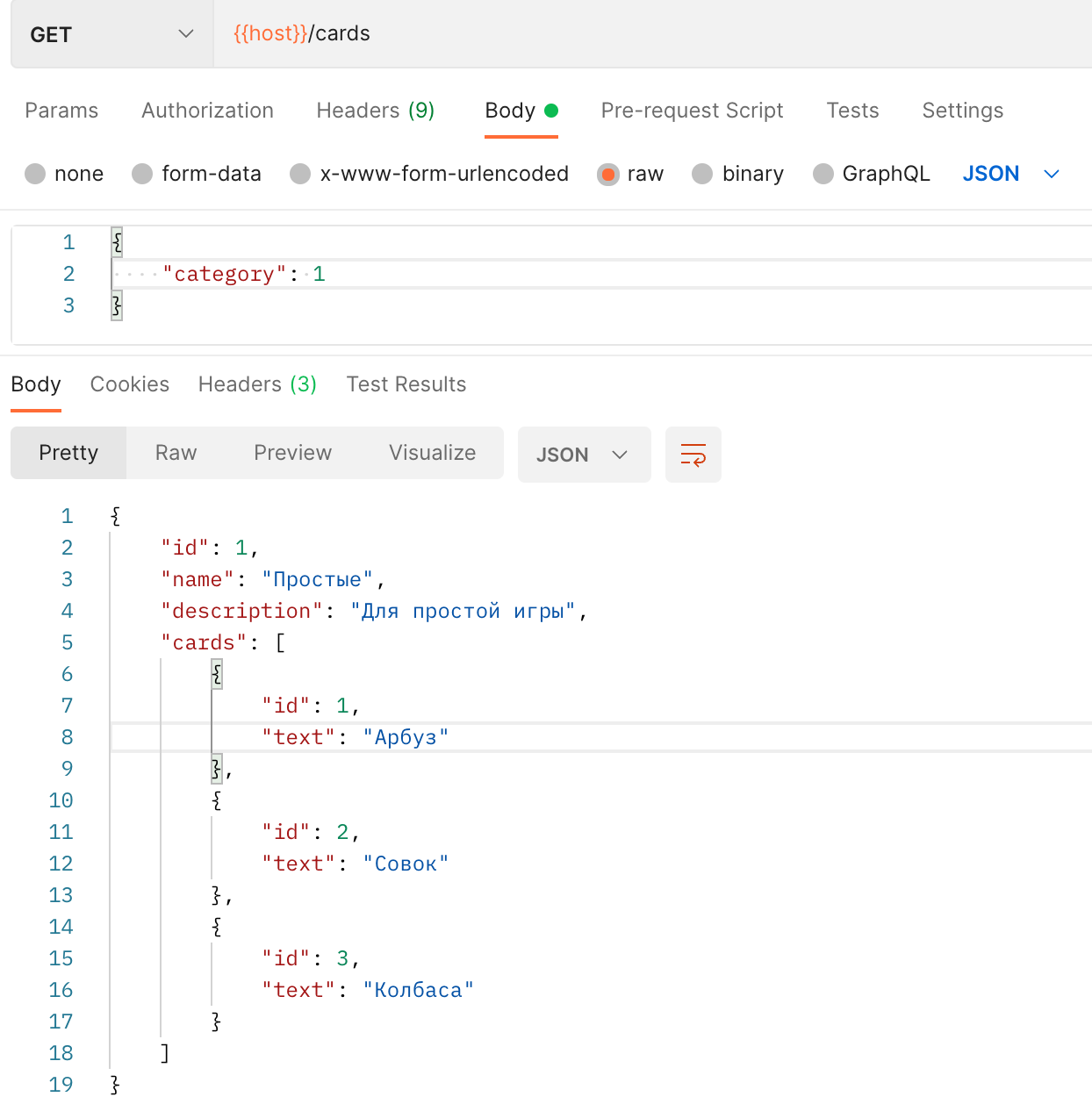


Рисунок 6 – пример запроса карточек.

1. POST-запрос */auth* – получение ключа сессии. Принимает логин и пароль, возвращает идентификатор пользователя, ключ сессии и уровень прав. Если пара логин-пароль не существует в базе, то будет возвращена ошибка 404 согласно протоколу HTTP. Листинг обработчика представлен в приложении А.
2. POST-запрос */session* – проверка ключа сессии на валидность. Для передачи сессии нужно использовать заголовок *session*. Возвращает либо идентификатор пользователи и уровень прав, либо ошибку 401 Unauthorized, если сессия не является валидной: истекла либо не существует.

## Авторизация для зарегистрированных пользователей

Для всех запросов редакторов обязательно требуется регистрация и последующая авторизация.

Чтобы зарегистрировать пользователя, необходимо сделать POST-запрос */register* и передать имя, логин, пароль и уровень доступа с помощью JSON. Причем, сделать это может только администратор. В ответ может быть получен уникальный идентификатор пользователя, либо ошибка регистрации: 400 Bad Request и сообщение «Пользователь уже существует».

В дальнейшем, для совершения всех запросов необходимо сначала авторизоваться, как это описано выше, а затем к каждому запросу добавлять заголовок.

Так как уровень привилегий в БД разделен только между авторизованными и неавторизованными пользователями, на сервере при каждом запросе проверяется, какие права имеет пользователь. Если это администратор – он может изменить любую ревизию. Если это редактор – только свою.

## Работа редакторов и администраторов с приложением

Для редакторов и администраторов предусмотрены одинаковые запросы, в каждом из них передачи сессии нужно использовать заголовок *session*. Листинг функции проверки авторизации для каждого из запросов приведен в приложении А.

Для изменения, применения и создания ревизий используются следующие запросы.

1. GET-запрос */revision/list* – получение списка всех ревизий в формате JSON. Если передать *revision\_id*, то вместо списка ревизий будет получен набор всех изменений для выбранной ревизии. Пример запроса – на рисунке 7.

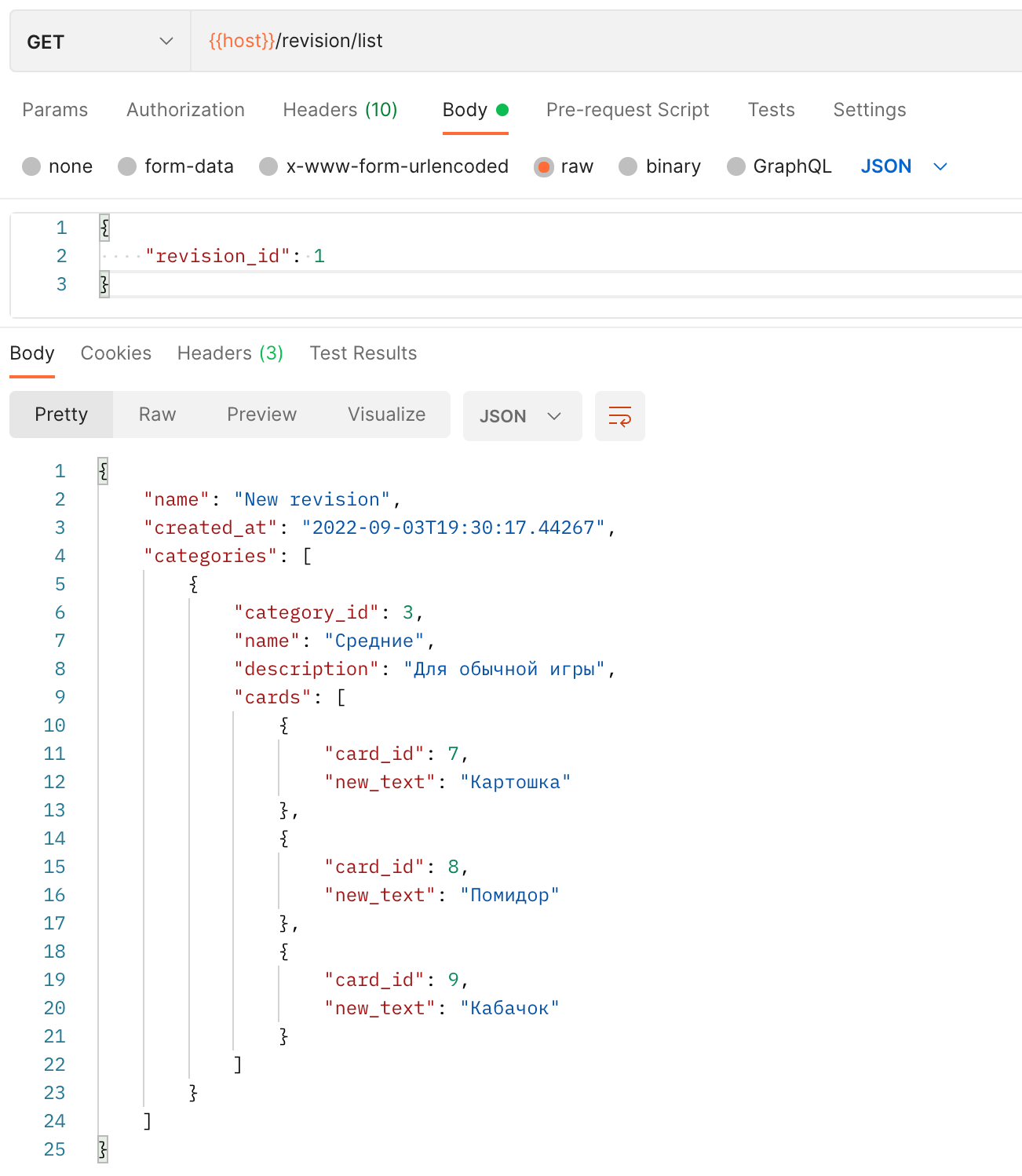


Рисунок 7 – пример запроса списка изменений.

1. POST-запрос */revision/new* – создание ревизии, возвращает уникальный идентификатор ревизии, нужен для последующих запросов. Листинг функции-обработчика представлен в приложении А.
2. POST-запрос */revision/remove* – удаление ревизии, принимает уникальный идентификатор ревизии.
3. POST-запрос */revision/apply* – применение ревизии. Добавляет, изменяет и удаляет все необходимые категории и карточки, добавленные в ревизию.

Для работы с категориями предусмотрено еще три запроса, каждый из них также принимает идентификатор ревизии.

1. POST-запрос */revision/category/new* – создание категории. Подходит, когда нужно создать новую категорию в оригинальной базе. Принимает имя и описание категории, возвращает уникальный идентификатор категории.
2. POST-запрос */revision/category/edit* – редактирование категории. Позволяет отредактировать либо категорию в оригинальной базе, либо категорию в базе ревизии. Принимает имя, описание и идентификатор категории. Если указать пустые и имя, и описание, то категория со всеми карточками будет удалена из оригинальной базы.
3. POST-запрос */revision/category/remove* – удаление изменения категории, подходит для удаления только из базы ревизии, а также для отмены удаления из оригинальной базы. Принимает уникальный идентификатор категории.

Для работы с карточки предусмотрено еще три запроса, каждый из которых также принимает идентификатор ревизии и идентификатор категории. Карточка может быть добавлена в ревизию только в том случае, если категория как-либо изменяется в данной ревизии.

1. POST-запрос */revision/card/new* – создание карты. Подходит, когда нужно создать новую карту в оригинальной базе. Принимает текст на карточке, возвращает уникальный идентификатор карточки.
2. POST-запрос */revision/category/edit* – редактирование карточки. Позволяет отредактировать карточки либо в оригинальной базе, либо в базе ревизии. Принимает текст на карточке и идентификатор карточки. Если указать пустой текст на карточке, то карточка будет удалена из оригинальной базы.
3. POST-запрос */revision/category/remove* – удаление изменения карточки, подходит для удаления только из базы ревизии, а также для отмены удаления из оригинальной базы. Принимает уникальный идентификатор карточки.

## Вывод

В данном разделе была разработана серверная программа и рассмотрено ее развертывание в контейнере вместе с базой данных. Были описаны необходимые действия для старта использования работы с проектом, а также рассмотрены все необходимые действия для непосредственного использования.

# Исследовательская часть

В исследовательском разделе будет рассмотрено сравнение составление JSON объекта – строки с помощью встроенных в PostgreSQL функций и с помощью программы.

## Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

1. процессор – Apple M1 Max;
2. оперативная память – 32 ГБ;
3. операционная система – macOS Monterey 12.4.

### Сравнительный анализ работы со строками

При разработке функционала для генерации строки в формате JSON было решено использовать стандартные функции PostgreSQL для работы с данным типом: *array\_to\_json* и *row\_to\_json*. Однако, данные действия можно выполнить также с помощью клиента БД.

В данном случае для сравнения используется функция *cw.revision\_categories\_with\_cards*, которая позволяет получить все категории и все карточки для каждой из категорий для ревизии, где для каждой категории используется подзапрос, который собирается в JSON, а после сбора всех карт для всех категорий, генерируется JSON со всеми категориями.

Для исследования со стороны клиента была использована функция на языке Golang, которая сначала делает запрос на слияние таблиц и карточек, затем получает объекты, включающие и категории, и карточки, а далее – превращает в структуру: для каждой категории помимо основных атрибутов добавляется список карт. Результат генерируется с помощью стандартной библиотеки *json*.

Для генерации информации о категориях был написан скрипт, позволяющий добавить в БД *x* категорий с *y* карточек в каждой. Замерялось время от запуска функции до получения результата. Результаты исследования представлены в таблице 1, время усреднялось в результате десяти запросов.

Таблица 1 – сравнение работы со строками с помощью БД и клиента.

|  |  |  |  |
| --- | --- | --- | --- |
| Категорий | Карт | PostgreSQL, мс | Golang, мс |
| 2 | 6 | 2,60 | 10,90 |
| 4 | 12 | 2,60 | 11,50 |
| 8 | 24 | 2,70 | 10,60 |
| 16 | 48 | 4,00 | 13,20 |
| 32 | 96 | 11,90 | 23,10 |
| 64 | 192 | 59,10 | 64,10 |
| 128 | 384 | 345,80 | 290,00 |
| 256 | 768 | 2434,00 | 1710,90 |
| 512 | 1536 | 11931,50 | 2434,60 |

В результате получения измерений был составлен график зависимости количества карт и категорий от времени выполнения запросов на PostgreSQL и Golang, представлен на рисунке 8. Для наглядности по оси Y используется десятичный логарифмический масштаб.

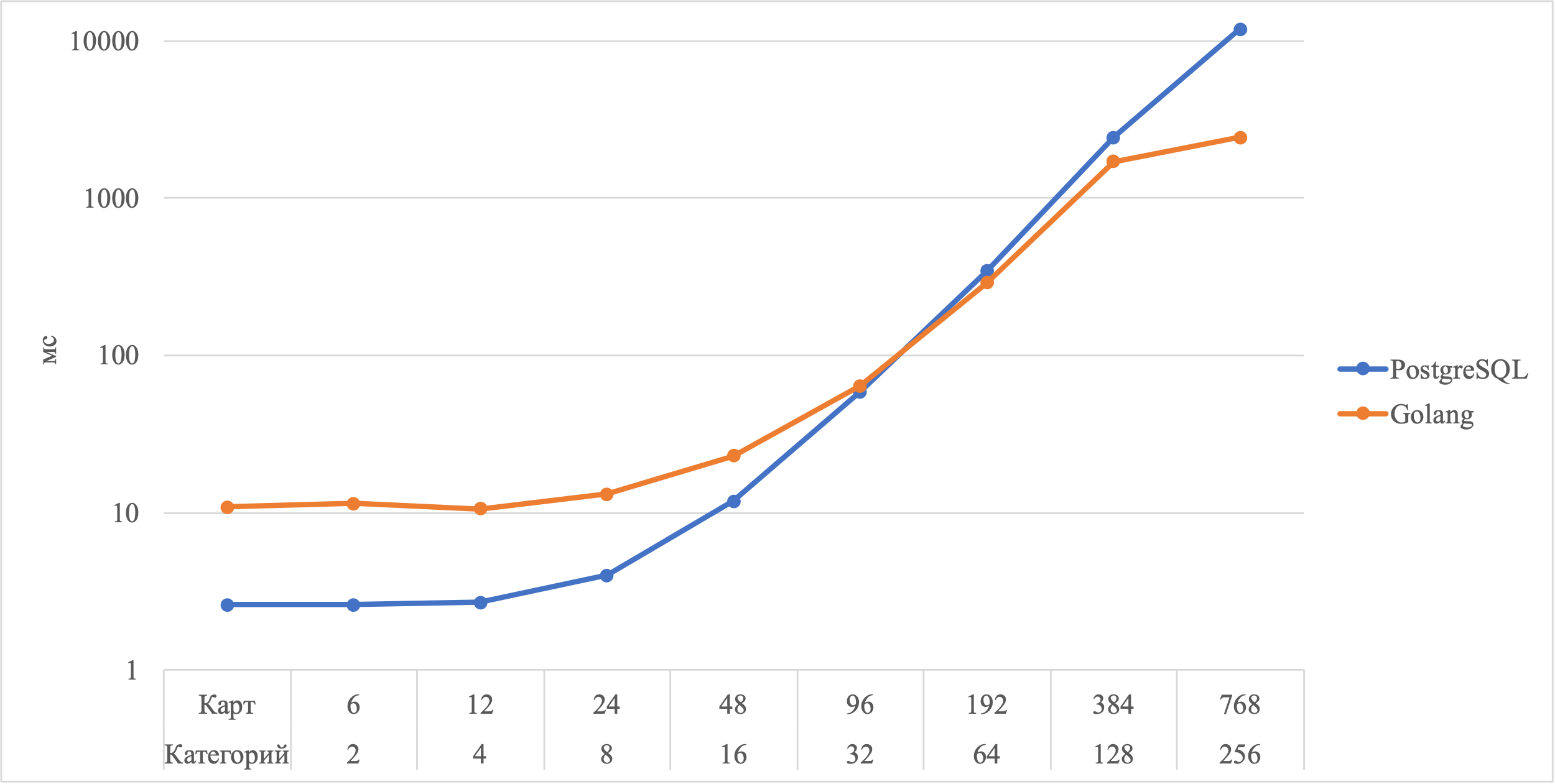


Рисунок 8 – зависимость времени выполнения от размера данных.

Исследуемые функции, как на стороне БД, так и на стороне клиента, приведены в приложении А.

## Вывод

Для полученных данных можно заметить, что до 8 категорий и 24 карт для каждой из них функция с использованием SQL позволяет сократить время почти в 10 раз. Для 64 категорий и 192 карт в каждой время работы функции на Golang сравнивается со временем для PostgreSQL. С возрастанием количества растет и выигрыш функции на Golang: для 512 категорий и 1536 карточек в каждой из них функция на PostgreSQL отстает почти в пять раз.

ЗАКЛЮЧЕНИЕ

В рамках выполнения курсовой работы была проанализирована задача, рассмотрены способы ее реализации и аналоги, описаны требования и ограничения к разработке БД. По результатам анализа была формализована задача и определена ролевая модель базы.   
 Были проанализированы виды баз данных и выбрана наиболее подходящая. Все необходимые запросы и ограничения на эти запросы были описаны, рассмотрены объекты базы данных. База данных была спроектирована и реализована в соответствии со всеми предусмотренными требованиями.

Для работы с базой данных было разработано программное обеспечение, позволяющее обеспечить безопасный ролевой доступ. Программный интерфейс приложения был описан: запросы и их ограничения.

Также исследовано быстродействие выполнение запросов на обработку строк – на уровне БД и на уровне клиента. Выявлено, что при объеме данных до 64 категорий и 192 карточек для каждой категории разумнее использовать генерацию строк JSON с помощью базы данных.

Таким образом, поставленные задачи выполнены и цель достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Loeliger J., McCullough M. Version Control with Git: Powerful tools and techniques for collaborative software development. – " O'Reilly Media, Inc.", 2012.
2. Мартишин, С. А. Базы данных. Практическое применение СУБД SQL и NoSQL-типа для проектирования информационных систем / С. А. Мартишин, В. Л. Симонов, М. В. Храпченко ; Рецензенты: В.А. Сизов, В.И. Гончаренко, А.И. Каптерев. – Москва : ИД «ФОРУМ» : ИНФРА-М, 2017. – 368 с. – (Высшее образование). – ISBN 978-5-8199-0660-6. – EDN GPRRAC.
3. M. J. Egenhofer, "Spatial SQL: a query and presentation language," in IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 1, pp. 86-95, Feb. 1994, doi: 10.1109/69.273029.
4. Сиротенко, Ф. Ф. Принципы работы и возможности системы управления версиями / Ф. Ф. Сиротенко // Наукоемкие технологии в космических исследованиях Земли. – 2010. – Т. 2. – № 1. – С. 15-17. – EDN TAMUSZ.
5. Папуловская, Н. В. Постреляционные хранилища данных : Учебное пособие / Н. В. Папуловская, Ю. П. Парфенов. – 1-е изд.. – Москва : Издательство Юрайт, 2020. – 1 с. – (Высшее образование). – ISBN 978-5-534-09837-2. – EDN BIQSTF.
6. The Open Web Application Security Project®. — Текст : электронный // OWASP : [Электронный ресурс]. — URL: https://owasp.org/ (дата обращения: 29.08.2022).
7. Ковалев, И. И. Защита персональных данных в распределенных системах с использованием стандартагост р 34.12-2015 для шифрования / И. И. Ковалев, А. В. Смакаев // Международная научно-техническая конференция молодых ученых БГТУ им. В.Г. Шухова : Посвящена 165-летию В.Г. Шухова, Белгород, 01–20 мая 2018 года. – Белгород: Белгородский государственный технологический университет им. В.Г. Шухова, 2018. – С. 3918-3922. – EDN ATWADM.
8. Schneier B. Description of a new variable-length key, 64-bit block cipher (Blowfish) // International Workshop on Fast Software Encryption. – Springer, Berlin, Heidelberg, 1993. – С. 191-204.
9. Zhou W. et al. REST API design patterns for SDN northbound API //2014 28th international conference on advanced information networking and applications workshops. – IEEE, 2014. – С. 358-365.
10. Belshe M., Peon R., Thomson M. Hypertext transfer protocol version 2 (HTTP/2). – 2015. – №. rfc7540.
11. Andrawos M., Helmich M. Cloud Native Programming with Golang: Develop microservice-based high performance web apps for the cloud with Go. – Packt Publishing Ltd, 2017.
12. Anderson C. Docker [software engineering] //Ieee Software. – 2015. – Т. 32. – №. 3. – С. 102-c3.

ПРИЛОЖЕНИЕ А

Листинг А.1 – сценарий создания таблиц и наложения на них ограничений.

|  |
| --- |
| 1. DROP SCHEMA IF EXISTS cw CASCADE; 2. CREATE SCHEMA cw; 3. CREATE SEQUENCE cw.category\_id\_seq; 4. CREATE TABLE cw.Category 5. ( 6. id INT NOT NULL PRIMARY KEY DEFAULT nextval('cw.category\_id\_seq'), 7. name varchar(30) NOT NULL UNIQUE, 8. description varchar(300) NOT NULL 9. ); 10. CREATE SEQUENCE cw.card\_id\_seq; 11. CREATE TABLE cw.Card 12. ( 13. id INT NOT NULL PRIMARY KEY DEFAULT nextval('cw.card\_id\_seq'), 14. category\_id INT NOT NULL, 15. text varchar(30) NOT NULL, 16. FOREIGN KEY (category\_id) REFERENCES cw.Category (id) 17. ); 18. CREATE TABLE cw.User 19. ( 20. id SERIAL PRIMARY KEY, 21. name varchar(100) not null, 22. login varchar(300) unique not null, 23. password varchar(60) unique not null, 24. access\_level smallint not null, 25. token varchar(32) unique, 26. token\_timestamp timestamp 27. ); 28. CREATE TABLE cw.Revision 29. ( 30. id SERIAL PRIMARY KEY, 31. name varchar(100), |

Продолжение листинга А.1.

|  |
| --- |
| 1. owner\_id INT, 2. created\_at timestamp, 3. FOREIGN KEY (owner\_id) REFERENCES cw.User (id) 4. ); 5. CREATE TABLE cw.CategoryT 6. ( 7. id SERIAL PRIMARY KEY, 8. revision\_id INT, 9. category\_id INT unique, 10. new\_name varchar(30), 11. new\_description varchar(30), 12. FOREIGN KEY (revision\_id) REFERENCES cw.Revision (id) 13. ); 14. ALTER TABLE cw.CategoryT 15. ADD UNIQUE (revision\_id, new\_name); 16. CREATE TABLE cw.CardT 17. ( 18. id SERIAL PRIMARY KEY, 19. revision\_id INT, 20. category\_id INT, 21. card\_id INT, 22. new\_text varchar(300), 23. FOREIGN KEY (revision\_id) REFERENCES cw.Revision (id), 24. FOREIGN KEY (category\_id) REFERENCES cw.CategoryT (category\_id) 25. ); 26. CREATE TABLE cw.key\_values 27. ( 28. key varchar primary key, 29. value VARCHAR68: ); |

Листинг А.2 – функции БД для авторизации.

|  |
| --- |
| 1. --авторизация 2. drop type if exists cw.auth\_result CASCADE; 3. CREATE type cw.auth\_result AS 4. ( 5. id int, 6. token varchar(32), 7. access\_level smallint 8. ); 9. CREATE or replace FUNCTION cw.auth(login\_in varchar, pw\_in varchar) 10. RETURNS cw.auth\_result 11. AS 12. $$ 13. DECLARE 14. res cw.auth\_result; 15. begin 16. IF EXISTS(SELECT \* FROM cw.user WHERE login = login\_in AND password = crypt(pw\_in, password)) THEN 17. update cw.user 18. set token = replace(gen\_random\_uuid()::text, '-', ''), 19. token\_timestamp = now() 20. where login = login\_in 21. returning id, token::text, access\_level::text into res.id, res.token, res.access\_level; 22. ELSE 23. select '', -1, -1 into res.token, res.access\_level, res.id; 24. END IF; 25. return res; 26. END 27. $$ LANGUAGE plpgsql SECURITY DEFINER; 28. --проверка сессии, возвращает либо id и уровень доступа, либо -1 29. drop type if exists session\_result CASCADE; |

Продолжение листинга А.2.

|  |
| --- |
| 1. CREATE type session\_result AS 2. ( 3. id int, 4. access\_level smallint 5. ); 6. CREATE or replace FUNCTION cw.session(token\_in varchar) 7. RETURNS session\_result 8. AS $$ 9. declare 10. ret session\_result; 11. begin 12. select id, access\_level 13. from cw.user 14. where token = token\_in 15. and EXTRACT(EPOCH FROM now() - token\_timestamp) < 86400 16. into ret.id, ret.access\_level; 17. if ret.id is null then 18. select -1, -1 into ret.id, ret.access\_level; 19. end if; 20. return ret; 21. END 22. $$ LANGUAGE plpgsql SECURITY DEFINER; |

Листинг А.3 – функции БД для изменения категорий.

|  |
| --- |
| 1. --изменение категории или ее удаление из базы 2. CREATE or replace FUNCTION cw.edit\_category( 3. revision\_id\_in int, 4. category\_id\_in int, 5. name varchar default null, 6. description varchar default null 7. ) 8. RETURNS int 9. AS $$ 10. declare 11. res int; |

Продолжение листинга А.3.

|  |
| --- |
| 1. begin 2. if (exists(select \* 3. from cw.categoryt 4. where revision\_id = revision\_id\_in 5. and new\_name = name 6. and not category\_id = category\_id\_in)) then 7. return -1; 8. end if; 9. if (not exists(select \* from cw.category where id = category\_id\_in) and (name is null or description is null)) then 10. return -2; 11. end if; 12. if (exists(select \* from cw.category where id = category\_id\_in) and 13. not exists(select \* from cw.categoryt where category\_id = category\_id\_in)) then 14. insert into cw.categoryt (revision\_id, category\_id, new\_name, new\_description) 15. values (revision\_id\_in, category\_id\_in, name, description) 16. returning id into res; 17. else 18. update cw.categoryT 19. set revision\_id = revision\_id\_in, 20. category\_id = category\_id\_in, 21. new\_name = name, 22. new\_description = description 23. where category\_id = category\_id\_in 24. returning id into res; 25. end if; 26. return res; 27. END 28. $$ LANGUAGE plpgsql SECURITY DEFINER; 29. --добавление категории 30. CREATE or replace FUNCTION cw.add\_category( |

Продолжение листинга А.3.

|  |
| --- |
| 1. revision\_id\_in int, 2. name varchar, 3. description varchar 4. ) 5. RETURNS int 6. AS $$ 7. declare 8. res int; 9. begin 10. if (exists(select \* from cw.categoryt where revision\_id = revision\_id\_in and new\_name = name)) then 11. return -1; 12. end if; 13. insert into cw.categoryT(revision\_id, category\_id, new\_name, new\_description) 14. values (revision\_id\_in, nextval('cw.category\_id\_seq'), name, description) 15. returning category\_id into res; 16. return res; 17. END 18. $$ LANGUAGE plpgsql SECURITY DEFINER; 19. --удаление изменения категории 20. CREATE or replace FUNCTION cw.remove\_category\_edition( 21. revision\_id\_in int, 22. category\_id\_in int 23. ) RETURNS bool AS 24. $$ 25. begin 26. if (not exists(select \* from cw.categoryt where revision\_id = revision\_id\_in and category\_id = category\_id\_in)) then 27. return false; 28. end if; 29. delete from cw.cardt where revision\_id = revision\_id\_in and category\_id = category\_id\_in; 30. delete from cw.categoryt where revision\_id = revision\_id\_in and category\_id = category\_id\_in; |

Продолжение листинга А.3.

|  |
| --- |
| 1. return true; 2. END 3. $$ LANGUAGE plpgsql SECURITY DEFINER; |

Листинг А.4 – функция и представление для получения ревизии в формате JSON.

|  |
| --- |
| 1. create or replace view cw.revision\_categories\_with\_cards as 2. select id, 3. category\_id, 4. new\_name, 5. new\_description, 6. revision\_id, 7. (select array\_to\_json(array\_agg(row\_to\_json(cards.\*))) as array\_to\_json 8. from (select c.card\_id, c.new\_text 9. from cw.cardt c 10. where c.revision\_id = r.revision\_id 11. and c.category\_id = r.category\_id) cards) as cards 12. from cw.categoryt r; 13. --ревизия в формате json 14. CREATE or replace FUNCTION cw.revision\_json(revision\_id\_in int) RETURNS varchar as 15. $$ 16. DECLARE 17. res varchar; 18. begin 19. select json\_build\_object( 20. 'name', r.name, 21. 'created\_at', r.created\_at, 22. 'categories', j.list) 23. from (select array\_to\_json(array\_agg(json\_build\_object( 24. 'category\_id', c.category\_id, 25. 'name', c.new\_name, |

Продолжение листинга А.4.

|  |
| --- |
| 1. 'description', new\_description, 2. 'cards', c.cards))) list 3. from cw.revision\_categories\_with\_cards c 4. where c.revision\_id = revision\_id\_in) j 5. join cw.revision r on r.id = revision\_id\_in 6. into res; 7. return res; 8. END 9. $$ LANGUAGE plpgsql SECURITY DEFINER; |

Листинг А.5 – пример регистрации обработчика API в программе.

|  |
| --- |
| 1. app.Get("/categories", func(c \*fiber.Ctx) error { 2. return getCategoriesHandler(c) 3. }) |

Листинг А.6 – функция приложения для получения категорий для неавторизованного пользователя.

|  |
| --- |
| 1. func getCategoriesHandler(ctx \*fiber.Ctx) error { 2. var categories = make([]Category, 0) 3. rows, err := watcherConnection.Query("SELECT c.id, c.name, c.description FROM cw.Category c;") 4. if err != nil { 5. return databaseError(err) 6. } 7. defer closeRows(rows) 8. for rows.Next() { 9. var row Category 10. err := rows.Scan(&row.Id, &row.Name, &row.Description) 11. if err != nil { 12. return databaseError(err) 13. } 14. categories = append(categories, row) 15. } 16. return ctx.JSON(fiber.Map{"categories": categories}) 17. } |

Листинг А.7 – функция авторизации в приложении.

|  |
| --- |
| 1. func authHandler(ctx \*fiber.Ctx) error { 2. cred := struct { 3. Login string `json:"login"` 4. Password string `json:"password"` 5. }{} 6. if err := ctx.BodyParser(&cred); err != nil || cred.Login == "" || cred.Password == "" { 7. return fiber.NewError(400, "Please, specify 'login' and 'password' via body") 8. } 9. rows, err := watcherConnection.Query( 10. "SELECT a.id, a.token, a.access\_level FROM cw.auth($1, $2) as a", 11. cred.Login, cred.Password, 12. ) 13. if err != nil { 14. return databaseError(err) 15. } 16. defer closeRows(rows) 17. if !rows.Next() { 18. return fiber.ErrUnauthorized 19. } 20. authResult := struct { 21. Id int `json:"id"` 22. Token string `json:"token"` 23. AccessLevel int `json:"access\_level"` 24. }{} 25. err = rows.Scan(&authResult.Id, &authResult.Token, &authResult.AccessLevel) 26. if err != nil { 27. panic(err) 28. } 29. if authResult.Id == -1 { 30. return fiber.ErrNotFound 31. } 32. return ctx.JSON(authResult) 33. } |

Листинг А.8 – функция проверки авторизации для каждого запроса авторизованного пользователя.

|  |
| --- |
| 1. func authorizeRequest(ctx \*fiber.Ctx) (\*SessionResult, error) { 2. token := ctx.GetReqHeaders()["Session"] 3. if len(token) != 32 { 4. if len(token) == 0 { 5. return nil, fiber.ErrUnauthorized 6. } else { 7. return nil, fiber.NewError(400, "Invalid token") 8. } 9. } 10. session, err := checkAuth(token) 11. if err != nil {return nil, fiber.ErrUnauthorized} 12. return &session, nil 13. } 14. func checkAuth(token string) (SessionResult, error) { 15. rows, err := watcherConnection.Query( 16. "SELECT s.id, s.access\_level FROM cw.session($1) as s", 17. token, 18. ) 19. if err != nil { 20. return SessionResult{}, err 21. } 22. defer closeRows(rows) 23. if !rows.Next() {return SessionResult{}, err} 24. var sessionResult SessionResult 25. err = rows.Scan(&sessionResult.Id, &sessionResult.AccessLevel) 26. if err != nil || sessionResult.Id == -1 || sessionResult.AccessLevel == -1 { 27. return SessionResult{}, errors.New("user not exists") 28. } 29. return sessionResult, err 30. } |

Листинг А.9 – функция создания ревизии для авторизованного пользователя.

|  |
| --- |
| 1. func revisionCreateHandler(ctx \*fiber.Ctx) error { 2. body := struct { 3. Name string `json:"name"` 4. }{} 5. if err := ctx.BodyParser(&body); err != nil { 6. return fiber.NewError(400, "Please, specify 'name' via body") 7. } 8. session, err := authorizeRequest(ctx) 9. if err != nil { 10. return err 11. } 12. var rows \*sql.Rows 13. if session.AccessLevel < 1 { 14. return fiber.ErrForbidden 15. } 16. rows, err = editorConnection.Query( 17. "INSERT INTO cw.revision (name, owner\_id, created\_at) VALUES ($1, $2, now()) RETURNING id", 18. body.Name, session.Id) 19. if err != nil { 20. return databaseError(err) 21. } 22. defer closeRows(rows) 23. if !rows.Next() { 24. return fiber.NewError(500, "Unexpected, cannot create") 25. } 26. var id int 27. err = rows.Scan(&id) 28. if err != nil { 29. return databaseError(err) 30. } 31. return ctx.JSON(fiber.Map{"id": id}) 32. } |

Листинг А.10 – конфигурация контейнеров для развертывания приложения и базы данных.

|  |
| --- |
| 1. version: '3.1' 2. networks: 3. local: 4. driver: bridge 5. services: 6. db: 7. image: postgres 8. restart: always 9. build: 10. dockerfile: Dockerfile 11. context: db 12. environment: 13. - POSTGRES\_USER=postgres 14. - POSTGRES\_PASSWORD=postgres 15. networks: 16. - local 17. ports: 18. - "5432:5432" 19. go: 20. build: 21. dockerfile: Dockerfile 22. context: src 23. networks: 24. - local 25. ports: 26. - "3000:3000" |

Листинг А.11 – функция для генерации ревизии в формате JSON из приложения-клиента БД.

|  |
| --- |
| 1. func revisionCategoriesListByCode() (error, string) { 2. var revisionId = -1 3. rows, err := adminConnection.Query("select c.category\_id cat\_id, cat.new\_name cat\_name, cat.new\_description cat\_desc, c.card\_id c\_id, c.new\_text c\_text from cw.categoryt cat join cw.cardt c on cat.category\_id = c.category\_id where c.revision\_id = $1 order by c.category\_id", revisionId) 4. defer closeRows(rows) 5. var categories = make([]CategoryWithCards, 0) 6. if err != nil { 7. return databaseError(err), "" 8. } 9. for rows.Next() { 10. var row CategoryWithCards 11. err := rows.Scan(&row.CatID, &row.CatName, &row.CatDescription, &row.CardID, &row.CardText) 12. if err != nil { 13. return databaseError(err), "" 14. } 15. categories = append(categories, row) 16. } 17. currentCategoryId := 0 18. categoriesWithList := make([]CategoryWithCardsList, 0) 19. cards := make([]Card, 0) 20. for i, s := range categories { 21. if i == 0 { 22. currentCategoryId = s.CatID 23. } 24. if currentCategoryId != s.CatID || i+1 == len(categories) { 25. var cat CategoryWithCardsList 26. cat.CatID = s.CatID 27. cat.CatName = s.CatName 28. cat.CatDescription = s.CatDescription 29. cat.Cards = cards 30. categoriesWithList = append(categoriesWithList, cat) |

Продолжение листинга А.11.

|  |
| --- |
| 1. cards = make([]Card, 0) 2. currentCategoryId = s.CatID 3. } 4. var card Card 5. card.CardId = s.CardID 6. card.CardText = s.CardText 7. cards = append(cards, card) 8. } 9. rows, err = adminConnection.Query("select name, created\_at from cw.revision where id = $1", revisionId) 10. type RevisionWithCategories struct { 11. Name string `json:"name"` 12. CreatedAt string `json:"created\_at"` 13. Categories []CategoryWithCardsList `json:"categories"` 14. } 15. var revision RevisionWithCategories 16. if !rows.Next() { 17. return databaseError(errors.New("empty function output")), "" 18. } 19. err = rows.Scan(&revision.Name, &revision.CreatedAt) 20. if err != nil { 21. return databaseError(err), "" 22. } 23. revision.Categories = categoriesWithList 24. result, err := json.Marshal(revision) 25. if err != nil { 26. return err, "" 27. } 28. return nil, string(result) 29. } |

Листинг А.12 – распределение ролей в БД.

|  |
| --- |
| 1. --пользователь watcher 2. create role watcher\_user login password '8B137DEC7A74463EB1836CA141BEADB3'; 3. revoke all on all tables in schema public from watcher\_user; 4. revoke all on all tables in schema cw from watcher\_user; 5. revoke execute on all functions in schema cw from watcher\_user; 6. grant usage on schema public to watcher\_user; 7. grant usage on schema cw to watcher\_user; 8. grant select on cw.Category, cw.Card to watcher\_user; 9. grant EXECUTE on function cw.v, cw.auth, cw.session to watcher\_user; 10. --пользователь editor 11. create role editor\_user LOGIN password 'FBEC626988EE4A02949A95C8B5BB113A'; 12. revoke all on all tables in schema public from editor\_user; 13. revoke all on all tables in schema cw from editor\_user; 14. revoke execute on all functions in schema cw from editor\_user; 15. grant usage on schema public to editor\_user; 16. grant usage on schema cw to editor\_user; 17. grant select on cw.Category, cw.Card, cw.CardT, cw.CategoryT, cw.Revision to editor\_user; 18. grant select on cw.revision\_categories\_with\_cards to editor\_user; 19. grant insert on cw.Revision to editor\_user; 20. grant EXECUTE on function cw.v, cw.auth, cw.session, cw.register, cw.owns 21. to editor\_user; 22. grant EXECUTE on function cw.revision\_json to editor\_user; 23. grant EXECUTE on procedure cw.remove\_revision, cw.revision\_apply to editor\_user; 24. grant EXECUTE on function cw.add\_category, cw.edit\_category, cw.remove\_category\_edition to editor\_user; 25. grant EXECUTE on function cw.add\_card, cw.edit\_card, cw.remove\_card\_edition to editor\_user; 26. GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA cw TO editor\_user; |