Санкт-Петербургский государственный университет

Математическое обеспечение и
администрирование информационных систем

Муравьев Кирилл Ильич

Разработка серверного приложения с использованием современных технологий

Отчёт по учебной практике

Научный руководитель:
ассистент кафедры ИАС Чернышев Г. А.

Санкт-Петербург
2021

Оглавление

Введение

Quizlet — разработанное в США образовательное приложение, основой которого являются “флеш-карты”. Флеш-карта — виртуальный аналог бумажной карточки, на которой спереди написан термин, а на обратной стороне его определение. Посредством этих флеш-карт Quizlet позволяет пользователям изучать различные темы и запоминать информацию при помощи игр и тестов, составленных из этих карт [9].

Оригинальный сайт Quizlet не лишён недостатков в функционале, а кроме того, настоятельно продвигает платную подписку. Отсюда возникло желание создать свой аналог для личного использования.

Для создания полноценного аналога сайта и приложения Quizlet необходима серверная часть (backend), так как веб-страницы, данные о пользователе и его коллекциях карточек должны быть доступны с любых его устройств — и, следовательно, храниться и обрабатываться централизованно.

Серверное приложение (веб-сервер) — это программный компонент, который предоставляет для клиентских программ и устройств функциональность, такую как общий доступ к данным и выполнение вычислений для клиента [10]; хранит, обрабатывает и отдаёт веб-страницы, обрабатывает входящие запросы, используя HTTP-протокол и родственные ему [12].

1. Постановка задачи

Целью данной работы является разработка серверного приложения, изучение доступных технологий и представление результатов этой разработки.

Для достижения этой цели были поставлены следующие задачи:

- Выбрать используемый стек технологий, в частности:
  - класс базы данных: реляционная или NoSQL,
  - конкретную СУБД,
  - язык и фреймворк для создания веб-приложения,
  - интерфейс для взаимодействия с клиентским приложением;
- Спроектировать архитектуру приложения;
- Спроектировать схему хранения данных в базе;
- Создать работоспособный API, в частности:
  - Реализовать типичные запросы к базе данных.

2. Выбор технологий

2.1. База данных

База данных — совокупность данных, организованных по определённым правилам, предусматривающим общие принципы описания, хранения и манипулирования данными, независимая от прикладных программ [13]. СУБД (система управления базами данных) — программное средство, предоставляющее приложению контролируемый доступ к данным в базе [6].

2.1.1. Реляционная или NoSQL

NoSQL-СУБД — термин, объединяющий различные системы и решения для хранения неструктурированной информации.

Ключевое отличие любой NoSQL-СУБД от реляционной (РСУБД) — отсутствие закреплённых отношений между данными. Кроме того, с точки зрения разработки продукта имеют значение следующие особенности:

- РСУБД масштабируется вертикально, то есть не может быть легко распределена на несколько серверов без потери целостности и согласованности (не допускает шардирования) [14];
- РСУБД в своей основе имеет таблицы, отношения между данными в которых определяются внешними ключами;
- РСУБД может обеспечить целостность чтения и записи данных при помощи транзакций.

Перечисленные особенности реляционных баз данных становятся ограничениями для некоторых классов приложений, в частности, для веб-приложений, ориентированных на выполнение большого количества простых операций. В свою очередь NoSQL-СУБД отличаются следующим:

- Большинство NoSQL-СУБД основаны на принципе key-value, то есть любая запись (из хранящихся в едином пространстве) имеет уникальный для этого пространства ключ. В ключе, помимо уникального идентификатора записи, сохранены её метаданные и "адрес" в кластере[1] [2] — что способствует быстрому поиску записи;

- NoSQL-СУБД, благодаря отсутствию жёстких связей между данными и key-value архитектуре, может быть скрытно от приложения распределена между многими серверами в кластере.

При разработке аналога приложения Quizlet важно обеспечить хранение и быстрое извлечение крупных объёмов данных (для большого количества пользователей), для чего необходимо предусмотреть возможность разделения базы данных на несколько серверов, связанных в кластер. Также для веб-приложений стандартом стала непрерывная бесперебойная работа, для чего отказоустойчивость базы данных является необходимым компонентом.

Поскольку эти свойства — архитектурные особенности NoSQL-СУБД, была выбрана эта технология.

2.1.2. СУБД

Существуют различные решения по организации баз данных, относимые к категории NoSQL:

- основанные на принципе key-value;

- документоориентированные;

- графовые.

Хранилища key-value представляют собой "бессхемную" (то есть без предопределённой модели хранения) коллекцию пар ключ-значение.

[1]Кластер — объединение нескольких серверов с целью равномерного распределения и/или репликации данных и нагрузки.

Такие пары хранятся в едином пространстве (namespace) и опционально могут быть логически объединены в группы (sets). СУБД оперируют конкретной парой по соответствующему ключу или группой значений из связанных пар [1].

В документоориентированных СУБД (Document Stores) данные представляются в виде групп “документов”, где каждый “документ” — файл в формате JSON или родственном ему, содержащий множество fieldName-value пар [1]. Документ не имеет предопределённой структуры, пара (или список пар) может быть добавлена в любое место. В примере ниже каждое tv_show — отдельный документ.

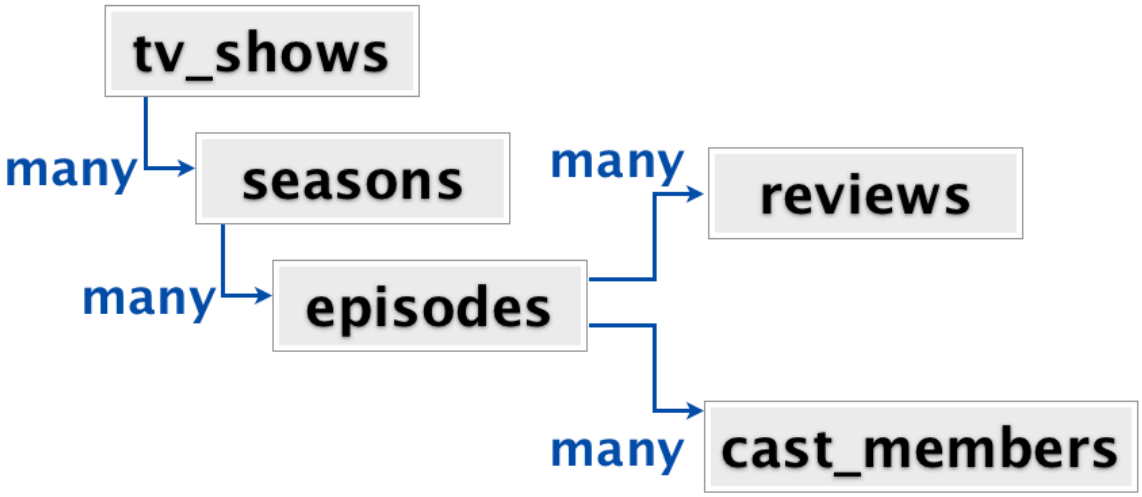

Рис. 1: Пример структуры документоориентированной БД [7].

Графовые БД — альтернативный подход к хранению данных в случае, когда важны связи между ними. Такая база состоит из вершин, рёбер и инвариантов графа. Причём рёбра хранят отношения между данными — они не вычисляются на лету; каждая вершина представляет собой множество key-value пар, аналогично документу в Document Store [11].

Поскольку нам требуется предусмотреть раздельное хранение данных карточек, коллекций карточек и пользователей (3.2), документоориентированная модель, где бы условием эффективной работы [5] было хранение всех карточек пользователя в одном документе с его лич-

ными данными, не подходит для нашего проекта. Поскольку отношения между сущностями могут изменяться (3.1), графовая БД также не удовлетворяет требованиям.

Таким образом, была выбрана NoSQL key-value модель.

Популярные key-value СУБД — Amazon DynamoDB, Redis и Aerospike.

DynamoDB на сервере, отличном от Amazon AWS, использует исполняемый .jar файл, то есть работает в виртуальной машине Java; две другие СУБД запускаются непосредственно. Поскольку не планируется развёртывание на Amazon AWS, DynamoDB нам не подходит.

Для добавления нового сервера (node) с целью объединения в кластер в Aerospike достаточно сконфигурировать эту node и присоединение, как и распределение данных, произойдёт автоматически. В Redis для создания кластера необходимо по крайней мере 3 master node, причём он перестанет функционировать в случае обрыва связи хотя бы с одной из них (single point of failure).

По этим причинам в качестве СУБД была выбрана Aerospike Community Edition.

```
# Aerospike installing:
wget -O aerospike-server.tgz 'https://www.aerospike.com/dow
tar -xvf aerospike-server.tgz
wget -O aerospike-tools.tgz 'https://www.aerospike.com/down
tar -xvf aerospike-tools.tgz
sudo service aerospike start
sudo vim /etc/aerospike/aerospike.conf
# clean partition before reusing: (inputFile, outputFile)
dd if=/dev/zero of=/dev/destination/disk
```

2.2. Веб-фреймворк

Из девяти языков программирования, поддерживаемых Aerospike [3], у автора наибольший опыт работы с C# и Python. Поскольку C# обеспечивает статическую типизацию, что упрощает поддержку проекта,

и является компилируемым, то есть на сервере будет запущен непосредственно исполняемый файл приложения, он был выбран в качестве языка разработки.

.NET Framework поддерживает только ОС Windows, тогда как Aerospike — только Linux. Поэтому мы будем использовать .NET Core, обеспечивающий работу .NET в ОС Linux.

В фреймворке .NET Core для разработки web-приложения доступны:

- ASP.NET Blazor Server App,
- ASP.NET Web API,
- ASP.NET Web App.

Первый фреймворк предназначен для разработки клиентской части приложения, третий — для одновременной разработки backend и frontend. Поскольку цель работы — только разработка backend-части, был выбран фреймворк ASP.NET Web API.

2.3. Интерфейс взаимодействия

В настоящее время стандартом разработки Web API является подход Representational State Transfer (REST), однако часто используются альтернативные подходы, такие как язык запросов и среда их выполнения GraphQL.

Недостатки API, основанного на REST [4]:

- Сервер определяет фиксированный набор запросов, которые может отправлять клиент;
- Для получения нескольких объектов потребуется послать несколько запросов.

Недостатки API, основанного на GraphQL [4]:

- Все запросы клиента отправляются на один адрес (single endpoint), таким образом, невозможно использовать HTTP-кеширование и необходимо реализовать собственное;

- Клиентское приложение может запрашивать любой набор данных, предусмотренных оговоренной схемой. Поэтому необходимо разработать схему[2], исключающую запросы, выполнение которых негативно скажется на производительности сервера и заблокирует исполнение остальных.

Поскольку нам понадобится отправлять клиентскому приложению различные множества флеш-карт и различные наборы данных пользователя (включая коллекции и флеш-карты), для уменьшения числа запросов к API будем использовать технологию GraphQL.

В качестве библиотеки, реализующей GraphQL, выберем HotChocolate ввиду наличия подробной документации и возможности работы "code-first" — автоматической генерации GraphQL-схемы на основе классов C#.

[2]Схема — объект, структурно похожий на JSON, перечисляющий и описывающий виды запросов; объекты и их поля, которые могут быть включены в запрос или отправлены на сервер.

3. Проектирование

Поскольку наше API ориентировано на работу с клиентским приложением, первоначально необходимо определить GraphQL-схему, описав всевозможные запросы.

Нам потребуется получать и отправлять данные пользователя:

- Логин,
- Hash пароля,
- Email,
- Его коллекции флеш-карт,
- Его флеш-карты.

При всяком запросе коллекции необходимо предоставить:

- Название,
- Описание,
- Содержащиеся в ней флеш-карты.

Для каждой флеш-карты необходимо предоставить:

- Лицевую сторону,
- Обратную сторону.

Поскольку получение всех коллекций и всех карточек — задача ресурсоёмкая и не всегда требуемая (например, при авторизации пользователя или просмотре информации в личном кабинете достаточно первых трёх полей), необходимо также описать запросы, не запускающие эту задачу.

Исходя из этих соображений, была разработана схема запросов, изображенная на Рис. 2. Запрос $hello()$ используется для первоначальной проверки соединения: он всегда выполняется успешно и возвращает “World!”.

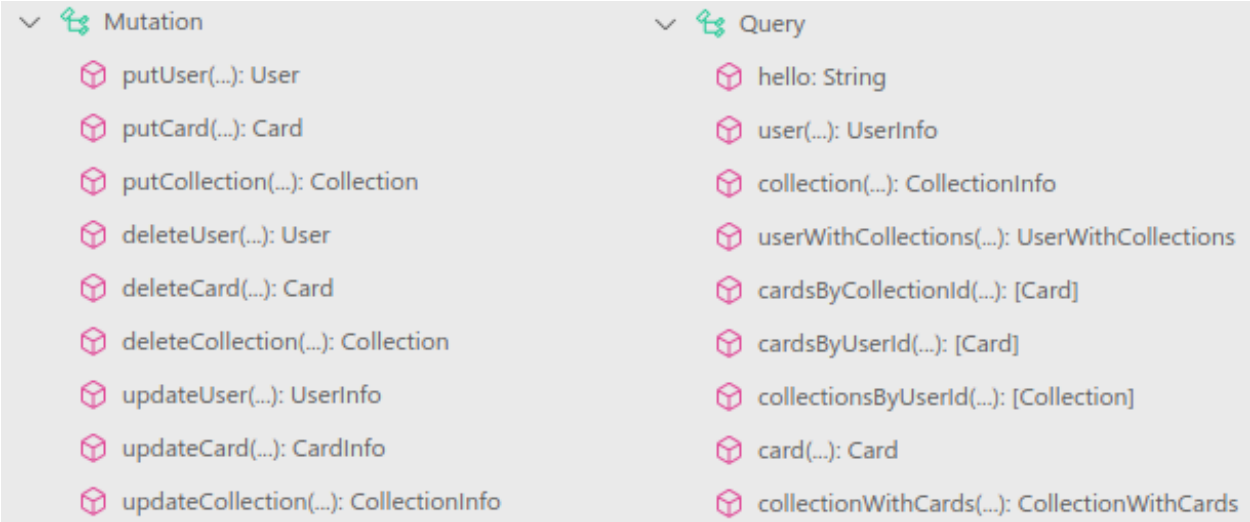

Рис. 2: GraphQL-схема доступных запросов к API.

Кроме того, внутри приложения удобно хранить и передавать Id вместо логинов и имён (в случае с коллекциями и карточками это необходимо, поскольку различные пользователи могут создать коллекции с одинаковыми именами). Соответственно, нужно реализовать возможность получения User.Id по его логину User.Login и наоборот — это обеспечивает перегруженный метод $user(...) : UserInfo$, в качестве аргумента которого может выступать как Id, так и Login.

3.1. Архитектура

Будем использовать монолитную архитектуру, поскольку она проще в разработке и развёртывании. Связь с БД будет реализована при помощи отдельных классов; GraphQL-сервер подключим в качестве сервиса.

Предусмотрим возможность будущей реализации совместного использования коллекций, то есть разрешим пользователю назначать других пользователей, которые получат полный доступ к управлению его коллекцией и карточками в ней. Для этого в модели пользователя (Рис. 3) будем хранить только список Id его коллекций. Тогда когда пользователь захочет поделиться коллекцией, нам будет достаточно добавить её Id в список соответствующего получателя. Аналогично, для обеспечения перемещения карточек между коллекциями, в модели коллекции будем хранить список Id принадлежащих ей карточек.

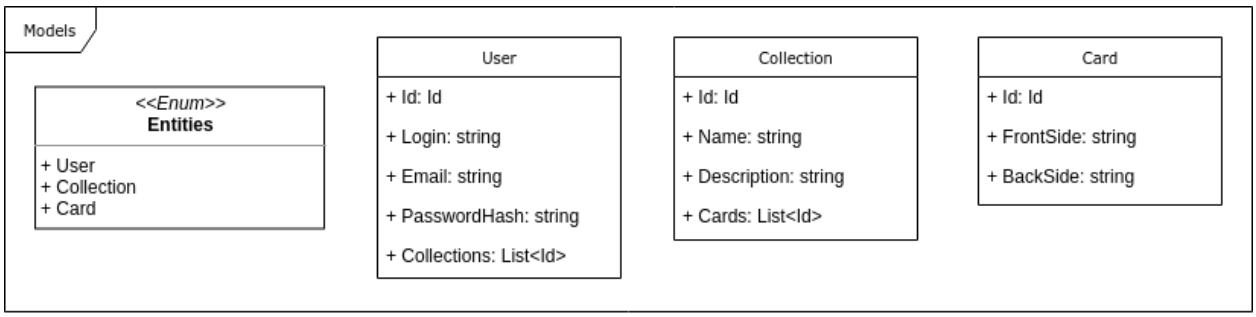

Рис. 3: UML-диаграмма классов используемых моделей.

Разделим работу с БД по назначению: AerospikeManagingClient (Рис. 4) будет использован единожды, чтобы создать вторичные индексы в базе для обеспечения последующей работы приложения. AerospikeQueryClient используется для получения данных из базы, AerospikeWriteClient — для всех операций, изменяющих данные в БД.

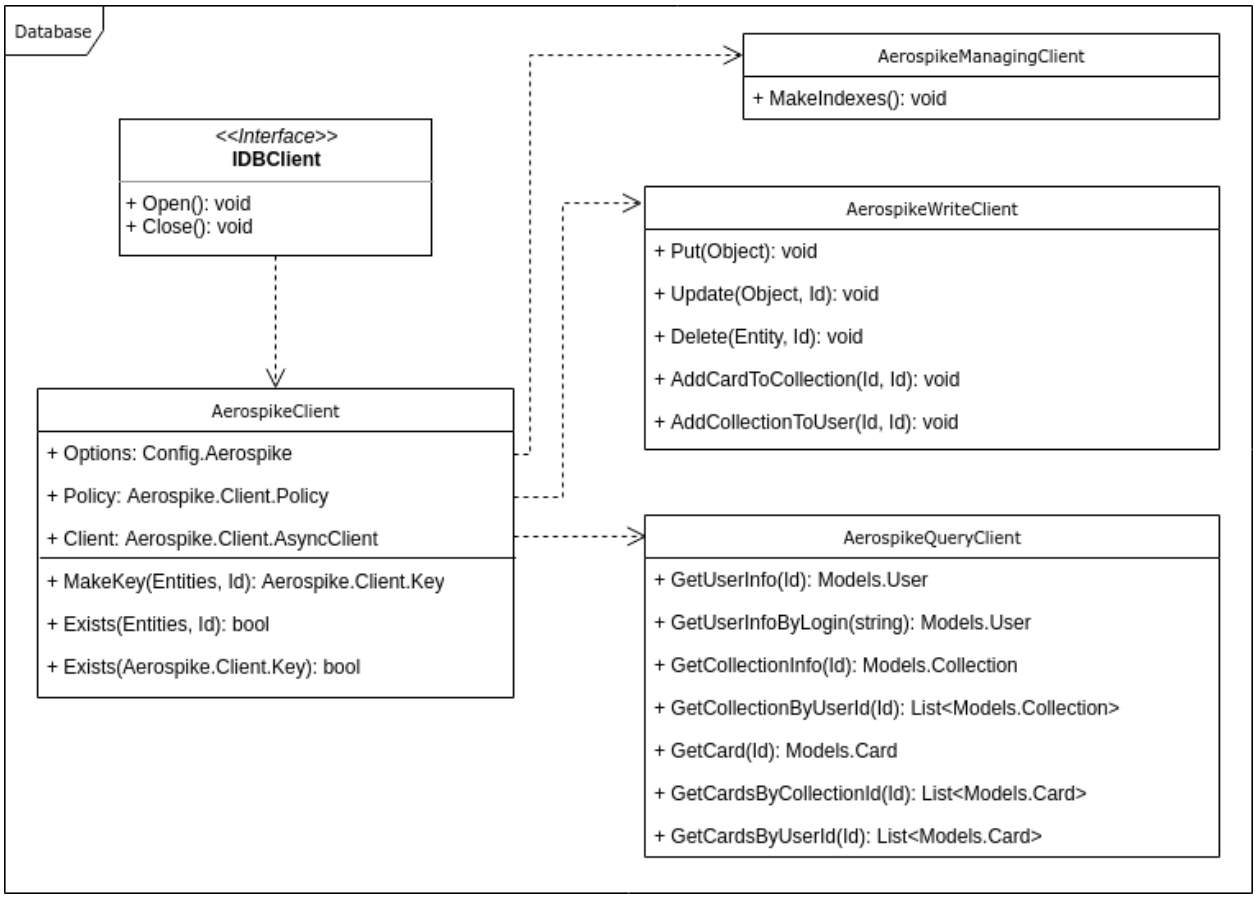

Рис. 4: UML-диаграмма классов работы с БД.

Так будет организовано взаимодействие клиента, подключённого к

нашему API, и серверного приложения. От запроса данных до их получения:

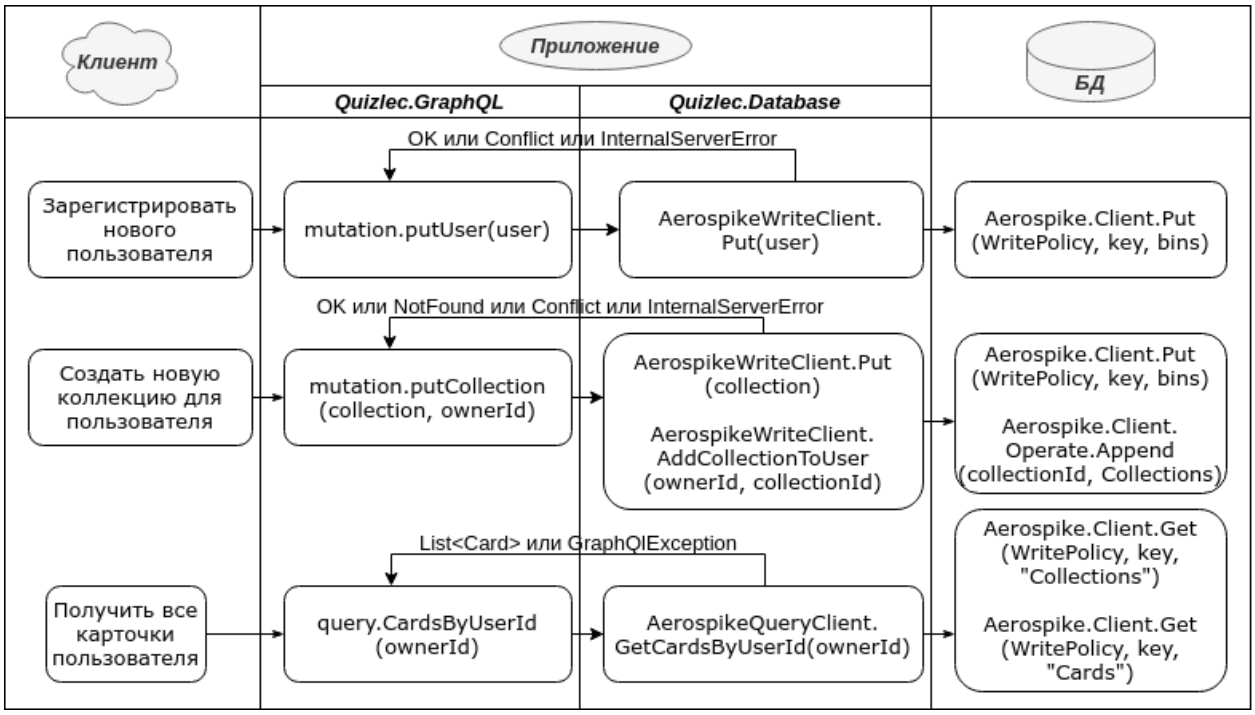

Рис. 5: Диаграмма последовательности примеров запросов.

3.2. Схема хранения

Схема хранения данных в базе будет соответствовать схеме, используемой внутри приложения, с добавлением поля “IsActive” типа integer. При работе с C#-библиотекой integer неявно приводится к bool и обратно, что позволяет использовать это поле как bool внутри приложения. Значение этого поля будет проверяться при каждом запросе к строке: в случае false данные считаются удалёнными. Таким образом, при запросе на удаление данных вместо их фактического удаления из базы мы только помечаем их удалёнными. Это обеспечивает возможность последующего восстановления данных и накопления статистики использования нашего приложения.

Поскольку в Aerospike нет такого понятия как внешний ключ, для сохранения связей между данными рекомендуется [8] использовать списки. Будем использовать списки из Id, как и в самом приложении.

В Aerospike доступны следующие типы данных: integer, double, string, bytes и, соответственно, List<type> и Map<type : type>, где type от каждого из четырёх. Id реализуем типа integer, что в сравнении с типом string позволит более быстро выполнять запросы по вторичному индексу.

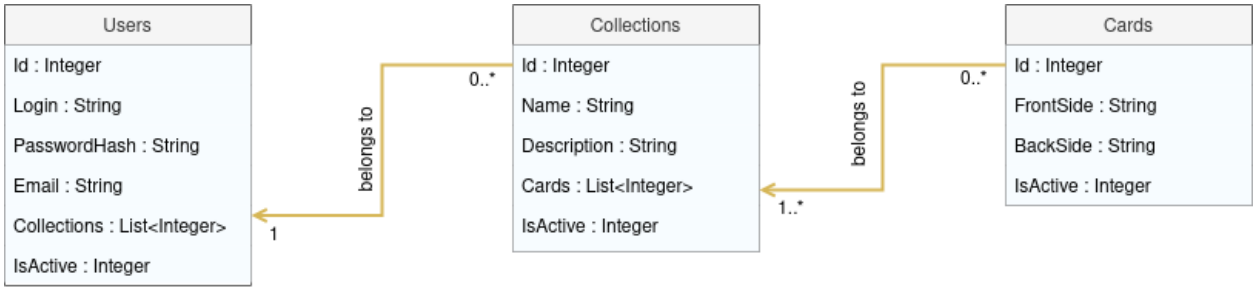

Рис. 6: Диаграмма хранения данных в БД.

4. Разработка

4.1. Реализация запросов

Каждый запрос выполняется внутри конструкции try-catch, что позволяет продолжать работу приложения даже в случае внутренней ошибки. Кроме того, реализованы пользовательские исключения и их обработчики, благодаря чему API возвращает корректный HTTP-код ошибки с пояснением, что позволяет пользователю локализовать проблему и исправить запрос.

```
// GraphQL.Query
public UserWithCollections GetUserWithCollections
    (string login)
{
    try
    {
        var client = new AerospikeQueryClient();
        var user = client.GetUserInfoByLogin(login);
        var collections =
            client.GetCollectionsByUserId(user.Id);
        client.Close();
        var res = new UserWithCollections()
        {
            Id = user.Id, Login = user.Login,
            Email = user.Email,
            Collections = collections,
            CollectionsCount = collections.Count
        };
        return res;
    }
    catch (NotFoundException e)
    {
        throw new GraphQlException("User not found. "
```

```
            + e.Message, HttpStatusCode.NotFound);
        }
        catch (Exception)
        {
            throw new GraphQlException(
                "Could not get user.",
                HttpStatusCode.InternalServerError);
        }
}
```

Обращение к библиотеке работы с БД также использует конструкцию try-catch и передаёт “наверх”, в том числе в сервис работы с GraphQL, пользовательские исключения, что позволит нам оперативно исправлять возможные ошибки на стороне сервера и скрывать от пользователя технические подробности внутренней работы приложения.

```
// Database.AerospikeQueryClient
public User GetUserInfoByLogin(string login)
{
    try
    {
        var res = new List<User>();
        var stmt = new Statement();
        stmt.SetNamespace(Options.Namespace);
        stmt.SetSetName(Options.Set.User);
        stmt.SetBinNames("Id", "Login",
            "PasswordHash", "Email", "IsActive");
        stmt.Filter = Filter.Equal("Login", login);
        RecordSet rs = Client.Query(
            (QueryPolicy) Policy, stmt);
        while (rs.Next())
        {
            var r = rs.Record;
            if (r.GetBool("IsActive"))
```

```
                res.Add(new User()
                {
                    Id = r.GetInt("Id"),
                    Login = login,
                    Email = r.GetString("Email"),
                });
            }
            rs.Dispose();
            if (res.Count == 1)
                return res[0];
            if (res.Count == 0)
                throw new UserNotFoundException
                    ($"The user with login={login}
                    doesn't exist.");
            if (res.Count > 1)
                throw new DatabaseQueryException
                    ($"Too many users with
                    login={login}: {res.Count}.");
            else
                throw new DatabaseQueryException();
        }
        catch (AerospikeException e)
        {
            throw new DatabaseQueryException
                ("Please create User:Login index
                first.", e);
        }
        catch (Exception e)
        {
            throw new DatabaseQueryException(
                "Cannot get user by login", e);
        }
    }
```

Полный код проекта можно найти на странице `https://github.com/studokim/Quizlec/`.

Заключение

В процессе разработки были достигнуты следующие результаты:

- Выбраны, изучены и использованы технологии работы с базой данных, разработки серверного приложения и его взаимодействия с клиентским приложением;
- Разработана архитектура приложения
- Разработана схема хранения данных;
- Реализованы типичные запросы к данным;
- Создан работоспособный API.

Список литературы

[1] Atzenia Paolo, et al. Data modeling in the NoSQL world // Computer Standards & Interfaces. — 2020. — P. 3–5.

[2] Data Model // AS101 Intro to Aerospike Community Edition. — Access mode: https://academy.aerospike.com/as101-intro-to-aerospike-community-edition/299211 (online; accessed: 10.01.2021).

[3] Development Overview // Aerospike Docs. — Access mode: https://www.aerospike.com/docs/client/index.html (online; accessed: 10.01.2021).

[4] Gilling Derric. REST vs GraphQL APIs, the Good, the Bad, the Ugly. — Access mode: https://www.moesif.com/blog/technical/graphql/REST-vs-GraphQL-APIs-the-good-the-bad-the-ugly/ (online; accessed: 10.01.2021).

[5] Klein John, et al. Performance Evaluation of NoSQL Databases: A Case Study // PABS '15: Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems. — 2015. — P. 5–10.

[6] Liu Ling, Özsu M. Tamer. Encyclopedia of Database Systems. — Springer, Boston, MA, 2009. — Access mode: https://link.springer.com/referencework/10.1007/978-0-387-39940-9.

[7] Mei Sarah. Why You Should Never Use MongoDB. — Access mode: http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/ (online; accessed: 10.01.2021).

[8] Modelling Conceps // Aerospike Docs. — Access mode: https://www.aerospike.com/docs/guide/cdt-list-examples.html#modeling-concepts (online; accessed: 10.01.2021).

[9] Quizlet // Wikipedia, the free encyclopedia. — Access mode: https://en.wikipedia.org/wiki/Quizlet (online; accessed: 10.01.2021).

[10] Server (computing) // Wikipedia, the free encyclopedia. — Access mode: https://en.wikipedia.org/wiki/Server_(computing) (online; accessed: 10.01.2021).

[11] Unal Yelda, Oguztuzun Halit. Migration of Data from Relational Database to Graph Database // ICIST '18: Proceedings of the 8th International Conference on Information Systems and Technologies. — 2018. — P. 1–5.

[12] Web server // Wikipedia, the free encyclopedia. — Access mode: https://en.wikipedia.org/wiki/Web_server (online; accessed: 10.01.2021).

[13] Организация данных в системах обработки данных. Термины и определения // ГОСТ 20886-85. — 2018.

[14] Подольский Даниил. SQL vs NoSQL: проблема выбора // ТК Conf. — Режим доступа: https://www.youtube.com/watch?v=zOEvutrfYt0 (дата обращения: 10.01.2021).