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 приложения по поиску наставника для изучения информационных технологий»

Студент ИУ7-66Б _____ Д. В. Варин
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсовой работы _____ Ю. М. Гаврилова
(Подпись, дата) (И. О. Фамилия)

2022 г.

РЕФЕРАТ

Расчетно-пояснительная записка 64 с., 9 рис., 12 табл., 19 источн., 7 прил.

Ключевые слова: Базы Данных, PostgreSQL, NoSQL, Redis, Golang, Docker, REST, Кэширование данных

Объектом разработки является базы данных для приложения по поиску наставника в сфере информационных технологий.

Цель работы — спроектировать и разработать базу данных для приложения по поиску наставника в сфере информационных технологий.

Для достижения данной цели необходимо решить следующие задачи:

- проанализировать и выбрать вариант представления данных для решения задачи;
- проанализировать системы управления базами данных и выбрать подходящую систему;
- спроектировать базу данных, описать ее сущности и связи;
- спроектировать интерфейс приложения, предоставляющего доступ к базе данных;
- реализовать программное обеспечение, позволяющее взаимодействовать с спроектированной базой данных.

В результате выполнения работы была спроектирована и разработана база данных для приложения по поиску наставника в сфере информационных технологий.

По результатам исследования, использование кэширования при получении информации из базы данных позволяет снизить времени отклика системы до 66 раз, при наличии высокой нагрузки на приложение. Для реализованного приложения высокой является нагрузка в 250 запросов в секунду.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие термины с соответствующими определениями.

Ментор — наставник, человек, у которого есть большой опыт в какой-то профессиональной области

Менти — подопечный

Персистентность — возможность долговременного хранения состояния

База данных — совокупность данных, хранимых в соответствии со схемой данных, манипулирование которыми выполняют в соответствии с правилами средств моделирования данных

Система управления базами данных — совокупность программных средств, обеспечивающих управление созданием и использованием баз данных

Structured Query Language — язык структурированных запросов

Not only Structed Query Language — термин, обозначающий ряд подходов, направленных на реализацию хранилищ баз данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных СУБД с доступом к данным средствами языка SQL

PostgreSQL — свободная объектно-реляционная система управления базами данных

MySQL — свободная реляционная система управления базами данных

Oracle Database — объектно-реляционная система управления базами данных компании Oracle

Redis — объектно-реляционная система управления базами данных компании Oracle

Tarantool — объектно-реляционная система управления базами данных компании Oracle

MongoDB — объектно-реляционная система управления базами данных компании Oracle

Метрика программного обеспечения — мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие сокращения и обозначения.

ПО — Программное обеспечение

SQL — Structured Query Language

NoSQL — Not only Structed Query Language

СУБД — Система управления базами данных

ВВЕДЕНИЕ

Сфера информационных технологий (IT) испытывает нехватку кадров. Каждый год университеты страны выпускают специалистов, но их количество не удовлетворяет спрос.

Из-за популярности it-специальностей, в последние годы набирают популярность курсы, на которых за небольшой промежуток времени готовят специалистов в различных направлениях.

На таких курсах существует менторинг - программа поддержки студентов. Ментор - это профессионал, обладающий компетенциями в своей сфере деятельности, источник знаний и ответов. Он помогает развиваться своему подопечному — менти.

Ментор преследует цель - поделиться собственным опытом с менти. Но на образовательных курсах ментор помогает, в основном, в рамках программы, на которой построен курс.

Другое направление менторинга - составление индивидуального плана развития. На программах при образовательных курсах программа менторства не покрывается исчерпывающе, доля персонального развития недостаточна. Индивидуальный подход более действеннен для людей, имеющих некоторый опыт в изучаемой сфере и желающих повысить свой уровень. Например, младший инженер-программист хочет повысить свой уровень до старшего. Для этого ему нужен человек, который сможет составить план, состоящий из конкретных шагов, при достижении которых, уровень компетенции возрастет до желаемого уровня.

Ментор - специалист, работающий в индустрии и знающий, какие навыки востребованы прямо сейчас, как повысить квалификацию. Этот человек способен составить программу и в короткие сроки подготовить менти к будущей работе. Другое направление - повышение уровня квалификации уже работающего специалиста.

Для того, чтобы найти наставника и составить план развития, нужна система, предоставляющая такую возможность.

Реализация системы требует наличия базы данных и методов взаимодействия с ней.

Цель работы – реализовать базу данных для приложения по поиску наставника для изучения информационных технологий.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- проанализировать варианты модели базы данных и выбрать подходящий вариант для решения задачи;
- спроектировать базу данных, описать ее сущности и связи;
- реализовать интерфейс для доступа к базе данных;
- реализовать программное обеспечение, которое позволит получить доступ к данным.

1 Аналитическая часть

1.1 Формализация задачи

Ментор - человек, работающий в IT индустрии. Он имеет опыт и экспертизу в сфере информационных технологий, предлагает свои услуги по обучению других людей.

Для менти необходима возможность найти ментора по навыкам, которые ему необходимо изучить (например, разработка операционных систем). В результате поиска с фильтрацией, в выдачу попадут лишь менторы, удовлетворяющие потребностям менти.

Ментор может просматривать список активных заявок на наставничество. В личном кабинете имеется список активных заявок, среди которых наставник выбирает себе менти.

После утверждения заявки, у ментора появляется возможность составлять планы развития для менти. План состоит из задач, у которых есть состояние готовности, срок выполнения. Менти должен по мере выполнения, отмечать задачи как выполненные.

Задачи ментора:

- предоставление рекомендаций и советов по карьерному росту;
- составление плана развития.

Возможности менти:

- изучить навыки, требуемые в работе;
- повысить профессиональный уровень;
- узнать о работе в разных компаниях, разных сферах;
- эффективно обучаться под руководством профессионалов.

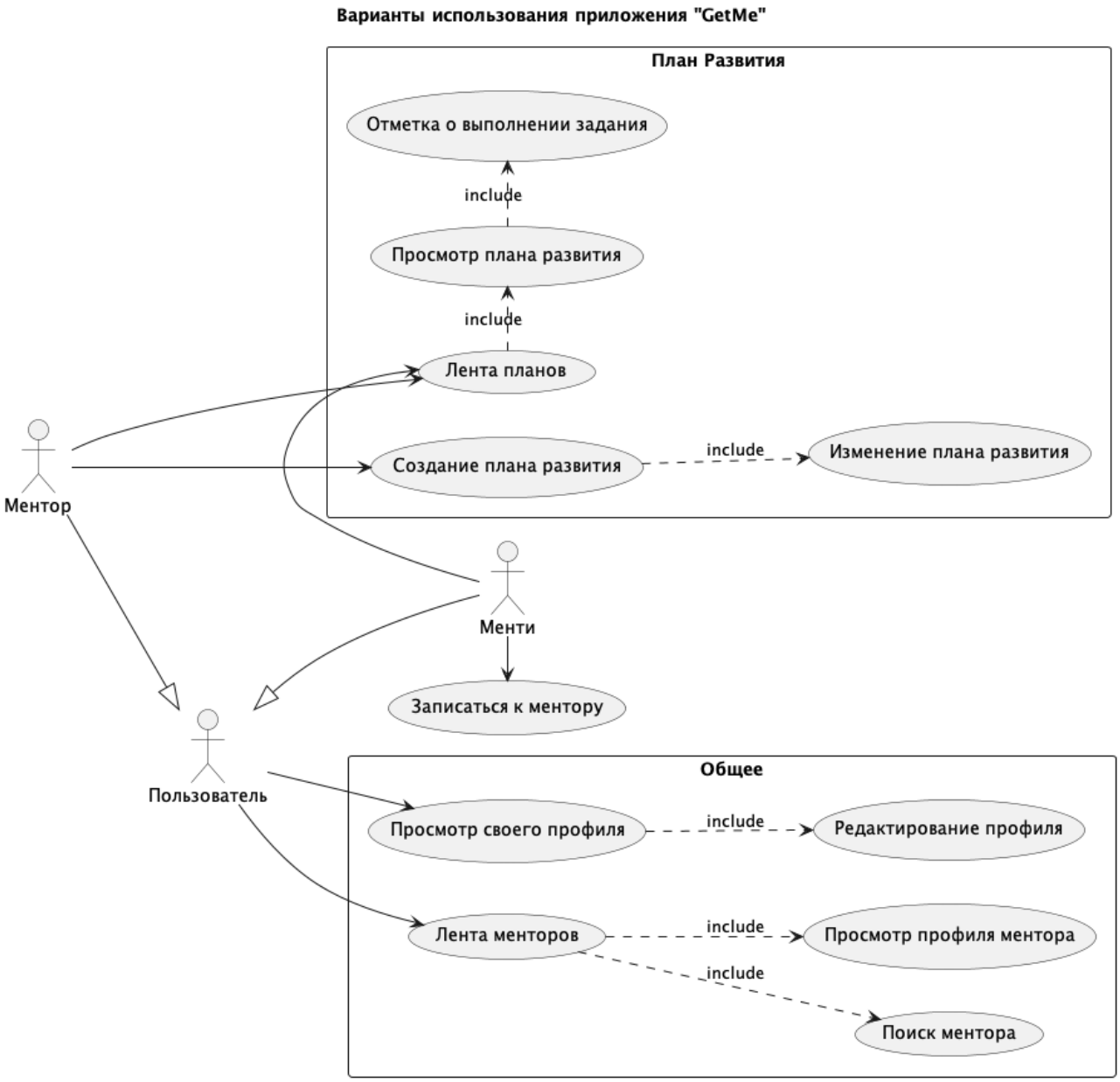

Рисунок 1.1 – Диаграмма использования приложения

1.2 Пользователи и данные в системе

Система, разрабатываемая в рамках курсового проекта, предполагает собой микросервисное приложение [1].

Доступ к системе имеет разграничение по ролям: ментор, менти, администратор. При этом, должно быть ограничение доступа к данным в зависимости от роли пользователя. Администратор имеет доступ к всей системе целиком.

Для хранения данных необходимо использовать строго структурированную и типизированную базу данных, так как:

- все роли имеют отношения между собой;

- доступ к данным разграничен;
- у предметной области есть структура, которую можно предописать заранее.

Данные в приложении делятся на следующие типы:

- сессии;
- пользовательские;
- заявки на обучение;
- планы;
- задачи в рамках плана;
- связи между менторами и менти.

Пользовательские данные и данные о планах обучения
Разрабатываемая система предполагает хранение данных о пользователях, связях между наставниками и подопечными, а также планов развития. Поэтому необходимо предусмотреть наличие нескольких ролей, пользователей, каждый из которых будет иметь возможность загружать свои наборы данных, взаимодействовать с своей частью системы.

1.3 Модели баз данных

Модель базы данных - это тип модели данных, которая определяет логическую структуру базы данных и в корне определяет, каким образом данные могут храниться, организовываться и обрабатываться [2]. Рассмотрим основные модели.

Иерархическая модель
В иерархической модели информация организована в виде древовидной структуры. Каждая запись имеет одного "родителя"и несколько потомков.

Такие модели данных графически могут быть представлены в виде перевернутого дерева. Оно состоит состоящее из объектов, каждый из которых имеет уровень. Корень дерево - первый уровень, его потомки второй и так далее.

Основной недостаток иерархической модели данных - невозможно реализовать отношение "many-to-many связь, при которой у потомка существует несколько родителей.

В качестве примера такой модели можно привести каталог в операционной системе Windows.

Сетевая модель

Сетевая модель - это структура, у которой любой элемент может быть связан с любым другим элементом.

Сетевая модель описывается как иерархическая, но в отличие от последней лишена недостатков, связанных с невозможностью реализовать "many-to-many"связь. Разница между сетевой и иерархической заключается в том, что в сетевой модели у потомка может быть несколько предков, когда у иерархической только один.

Основной недостаток - жёсткость задаваемых структур и сложность изменения схем БД из-за реализации связей между объектами на базе физических ссылок (через указатели на объекты).

В качестве примера такой модели можно привести WWW (World Wide Web).

Реляционная модель

Данные в реляционной модели хранятся в виде таблиц и строк, таблицы могут иметь связи с другими таблицами через внешние ключи, таким образом образуя некие отношения.

Реляционные базы данных используют язык SQL. Структура таких баз данных позволяет связывать информацию из разных таблиц с помощью внешних ключей (или индексов), которые используются для уникальной идентификации любого атомарного фрагмента данных в этой таблице. Другие таблицы могут ссылаться на этот внешний ключ, чтобы создать связь между частями данных и частью, на которую указывает внешний ключ.

SQL используют универсальный язык структурированных запросов для определения и обработки данных. Это накладывает определенные ограничения: прежде чем начать обработку, данные надо разместить внутри таблиц и описать.

Нереляционная модель

Данные нереляционных баз данных не имеют общего формата. Они могут пред-

ставляться в виде документов (Mongo [3], Tarantool [4]), пар ключ-значение (Redis [5]), графовых представляниях.

Динамические схемы для неструктурированных данных позволяют:

- ориентировать информацию на столбцы или документы;
- основывать ее на графике;
- организовывать в виде хранилища Key-Value;
- создавать документы без предварительного определения их структуры, использовать разный синтаксис;
- добавлять поля непосредственно в процессе обработки.

1.3.1 Выбор модели базы данных для решения задачи

Для решения задачи будет использоваться реляционная модель данных по нескольким причинам:

- изложение данных будет осуществляться в виде таблиц;
- данные структурированные, структура нечасто измяема;
- возможность исключить дублирование, используя связь между отношениями с помощью внешних ключей;
- разделение доступа к данным от способа их физической организации.

Для хранения состояний системы, в частности, авторизации пользователей, следует использовать нереляционную модель.Особенность таких данных в том, что у них нет отношений и связей, они хранятся парами key-value. Главное требование - данная база должна отличаться быстродействием.

Для решения таких задач следует рассмотреть in-memory СУБД.

In-Memory —- это набор концепций хранения данных, когда они сохраняются в оперативной памяти приложения, а диск используется для бэкапа. В классических подходах данные хранятся на диске, а память — в кэше.

Такие СУБД основаны на нереляционной модели данных.

1.4 Формализация данных

База данных пользователей, планов развития и отношений между ролями

Хранение данных пользователей, планов развития и отношений между ролями должны храниться в одной базе данных. Пользователи должны иметь уникальные идентификаторы, чтобы их можно было однозначно идентифицировать.

Наставники и подопечные связаны отношением N:M, информация об этих связях должна сохраняться.

Каждый план должен хранить идентификатор наставника и подопечного, чтобы идентифицировать роли.

Ниже представлена ER-диаграмма сущностей в нотации Чена.

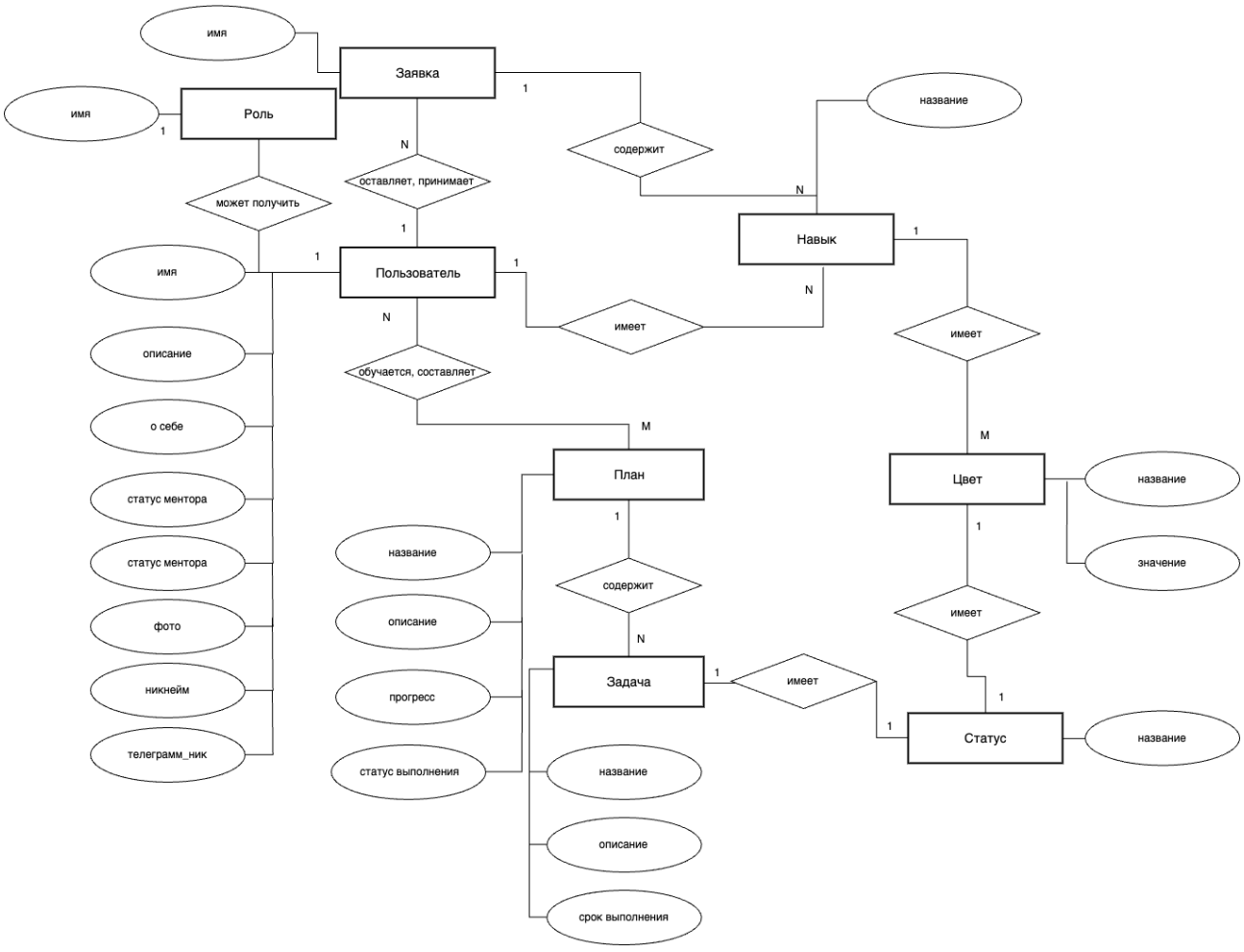

Рисунок 1.2 – Диаграмма использования приложения

База данных сессий

Современные веб приложения построены на основе HTTP протокола, который не сохраняет свое состояние (stateless [6]), каждый запрос является независимой транзакцией. Это значит, что отслеживание взаимодействия с клиентами (пользователями) нужно реализовывать самостоятельно. Это нужно для того, чтобы клиенту не приходилось на каждый запрос проходить аутентификацию в системе, он мог в течение некоторого промежутка времени пользоваться системой без повторной аутентификации.

Сессии [7] позволяют отслеживать "состояния" между сайтом и каким-либо клиентом. Они позволяют хранить произвольные данные и получать их в момент установления соединения между клиентом и сервером. Данные получаются и сохраняются в сессии при помощи "ключа".

База данных сессий должна хранить пары значений: *уникальный идентификатор сессии - уникальный идентификатор пользователя*. Данные должны отдаваться быстро и иметь устойчивость к отказу, например, сохраняться на твердотельный накопитель с возможностью восстановления.

Вывод

В данном разделе:

- рассмотрена структура базы данных для системы;
- привелена диаграмма использования приложения;
- проанализированы и определены модели баз данных для решения задачи;
- формализованны данные, используемые в системе.

2 Конструкторская часть

2.1 Проектирование базы данных

Для основной логики приложения была спроектирована база данных, представленная в виде ER-модели, в нотации Crow's Foot.

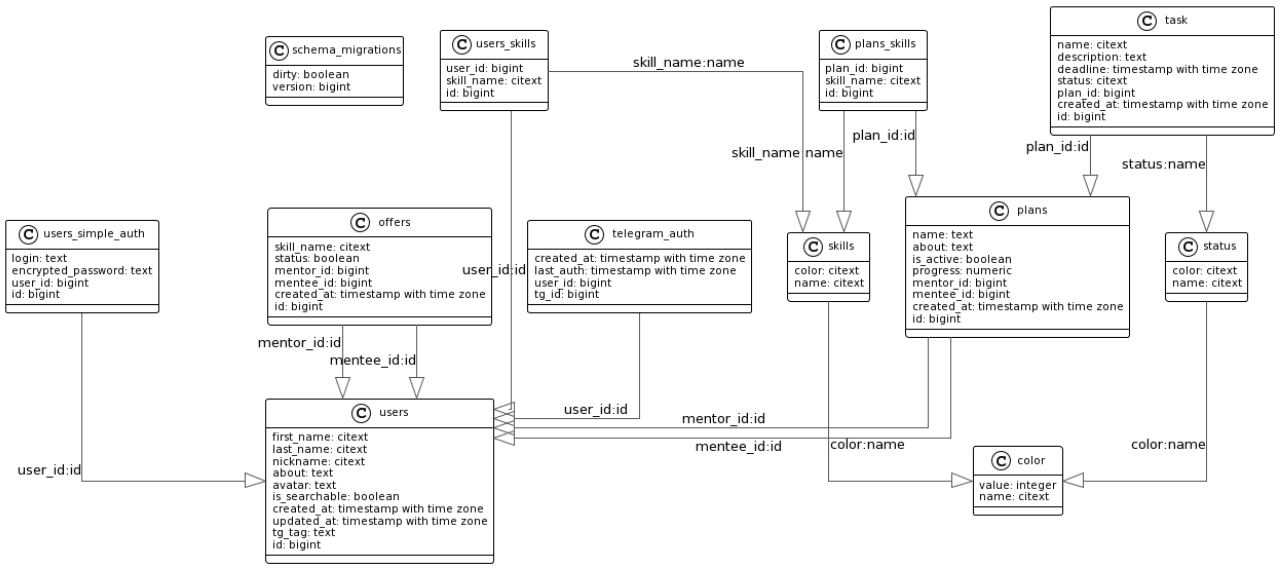

Рисунок 2.1 – ER диаграмма основной логики приложения

База данных будет состоять из следующих сущностей:

1. Users - пользователи;
2. Plans - планы развития;
3. Task - задачи из плана развития;
4. Skills - изучаемые навыки;
5. SimpleAuth - данные аутентификации вида логин-пароль;
6. TelegramAuth - данные о телеграмме пользователя;
7. Status - статус выполнения задачи;
8. Панель администратора - набор сущностей панели администратора
9. Sessions - хранение сессий пользователей

Сущность Users

Сущность Users содержит информацию о пользователях:

Таблица 2.1 – Описание полей таблицы `users`

Поле	**Значение**
`id`	Идентификатор пользователя, уникальный.
`first_name`	Имя пользователя в системе
`last_name`	Фамилия пользователя
`nickname`	Псевдоним, короткая альтернатива имени
`about`	Информация о пользователе
`is_searchable`	Признак отоюражения пользователя в поиске
`tg_tag`	Псевдоним в Telegram
`created_at`	Время регистрации пользователя
`updated_at`	Время последнего редактирования информации о пользователе

Сущность Task

Сущность Task содержит информацию о задачах, которые прикреплены к плану:

Таблица 2.2 – Описание полей таблицы `plans`

Поле	**Значение**
`id`	Идентификатор задачи, уникальный.
`name`	Название задачи.
`description`	Описание задачи
`deadline`	Крайний срок выполнения
`status`	Статус задачи
`plan_id`	Идентификатор плана развития, к которому привязана задача
`created_at`	Время создания задачи

Сущность Plans

Сущность Plans содержит информацию о планах развития:

Таблица 2.3 – Описание полей таблицы `plans`

Поле	**Значение**
`id`	Идентификатор плана, уникальный.
`name`	Название плана развития.
`about`	Описание плана развития
`is_active`	Признак выполнения плана
`progress`	Процент выполнения плана
`mentor_id`	Идентификатор ментора, создавшего план
`mentee_id`	Идентификатор менти, обучающемуся по плану
`created_at`	Время создания плана
`updated_at`	Время последнего изменения информации о плане

Сущность Skills

Сущность Skills содержит информацию о навыках, существующих в системе:

Таблица 2.4 – Описание полей таблицы `skills`

Поле	**Значение**
`name`	Название навыка.
`color`	Цвет навыка (для бейджа)

Сущность Status

Сущность Status содержит информацию о навыках, существующих в системе:

Таблица 2.5 – Описание полей таблицы `status`

Поле	**Значение**
`name`	Текстовое название статуса.
`color`	Цвет статуса (для бейджа)

Сущности Авторизации

В приложении есть два способа авторизации: через мессенджер Telegram[8] и через пару логин-пароль. Сущность SimpleAuth содержит данные авторизации пользователей через логин-пароль:

Таблица 2.6 – Описание полей таблицы `users_simple_auth`

Поле	**Значение**
`id`	Идентификатор записи.
`login`	Логин пользователя.
`encrypted_password`	Зашифрованнный пароль
`user_id`	Идентификатор пользователя к которому привязаны данные авторизации

Сущность TelegramAuth содержит данные авторизации пользователей через мессенджер Telegram:

Таблица 2.7 – Описание полей таблицы `users_simple_auth`

Поле	**Значение**
`tg_id`	Имя пользователя в telegram.
`last_auth`	Время последнего входа в приложение через telegram.
`user_id`	Идентификатор пользователя, к которому привязаны данные авторизации
`created_at`	Дата первого входа в систему через telegram

Сущности панели администратора

Для администрирования базы данных, поддержания ролевой модели будет создан ряд сущностей. ER-модель сущностей, в нотации Crow's Foot, представлена ниже:

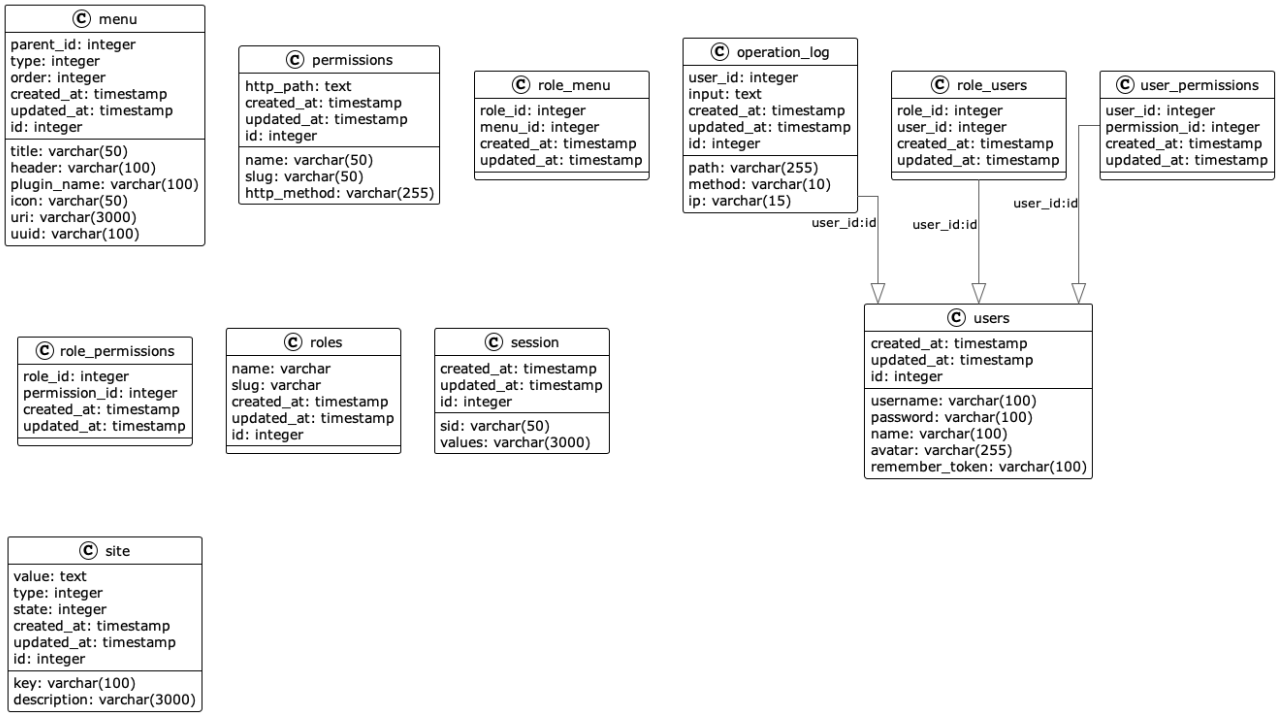

Рисунок 2.2 – ER диаграмма сущностей панели администратора

Таблицы для миграций базы данных

Миграции базы данных - это система контроля версий базы данных. Для обновления базы данных необходимо определить сущность, которая будет контролировать текущую версию.

Сущность SchemaMigrations содержит данные о текущей версии базы данных:

Таблица 2.8 – Описание полей таблицы `schema_migrations`

Поле	**Значение**
`version`	Номер текущей версии БД
`dirty`	Признак успешного обновления БД

2.2 Проектирование базы данных сессий

Для отслеживания "состояния"между клиентом и сервером необходимо хранить данные, которые идентифицирует конкретного пользователя. Как говорилось выше, для таких задач используют in-memory СУБД. Такие базы данных не имеют одной структуры (как реляционные), поэтому можно лишь высокоуровнево спроектировать базу.

Для хранения пользовательские сессий требуются следующие поля:

Таблица 2.9 – Описание полей таблицы `sessions`

Поле	**Значение**
`session_id`	Идентификатор сессии, уникальный
`user_id`	Идентификатор пользователя, которому принадлежит сессия

2.3 Ограничения, связи между сущностями, целостность данных

Для избежания дублирования данных сущности связаны между собой через внешние ключи.

Внешние ключи

- В Таблице telegram_auth поле user_id ссылается на поле id таблицы users;
- В Таблице users_simple_auth поле user_id ссылается на поле id таблицы users;
- В Таблице plans
  - поле mentee_id ссылается на поле id таблицы users;
  - поле mentor_id ссылается на поле id таблицы users;
- В Таблице status поле color ссылается на поле name таблицы color;
- В Таблице task
  - поле plan_id ссылается на поле id таблицы plans;
  - поле status ссылается на поле name таблицы status;
- В таблице skills поле color ссылается на поле name таблицы color;
- В Таблице offers
  - поле mentee_id ссылается на поле id таблицы users;
  - поле mentor_id ссылается на поле id таблицы users;
  - поле skill_name ссылается на поле name таблицы skills;

Помимо внешних ключей, в базе данных присутствуют связи типа many-to-many, реализующиеся через промежуточные таблицы.

Для хранения пользовательские связей навык-пользователь используется таблица users-skills:

Для хранения связей навык-план используется таблица plans-skills:

Таблица 2.10 – Описание полей таблицы `users_skills`

Поле	**Значение**
`id`	Идентификатор записи
`skill_name`	Название навыка, внешний ключ к таблице skills
`user_id`	Идентификатор пользователя, внешний ключ к таблице users

Таблица 2.11 – Описание полей таблицы `plans_skills`

Поле	**Значение**
`id`	Идентификатор записи
`skill_name`	Название навыка, внешний ключ к таблице skills
`plan_id`	Идентификатор плана, внешний ключ к таблице plans

Ролевая модель

В базе данных присутствуют три роли:

1. Администратор - имеет доступ к всем таблицам, доступны все операции над ними;

2. Пользователь - доступ к всем таблицам для CRUD операций, кроме таблиц telegram_auth, schema_migrations и всех таблиц, сявзанных с панелью администратора;

3. Гость - доступ к тем же таблицам, что у пользователя на операции SELECT.

Триггеры

Для обновления поля progress в таблице task, нужен механизм, который будет при каждом обновлении статуса задачи (добавления новой задачи к плану) обновлять поле progress.

Такую задачу может выполнить триггер, который будет срабатывать при обновлении таблицы plan и добавлении новых записей. После таких действий

будет срабатывать функция, которая будет актуализировать progress у всех записей таблицы.

Ниже представлена схема алгоритма работы выполняемой тригером функции update_progress().

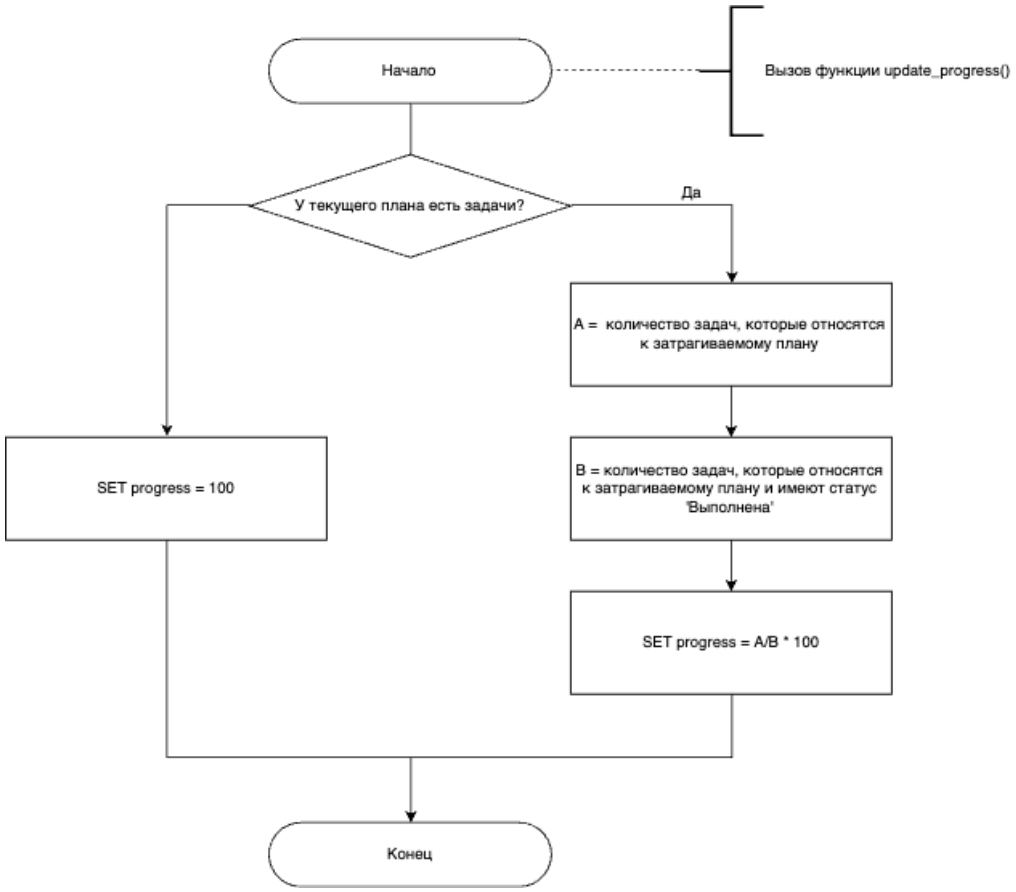

Рисунок 2.3 – Схема алгоритма работы функции триггера

Вывод

В данном разделе:

- спроектированы сущности базы данных;
- описаны сущности для миграции данных;
- спроектирована база данных для хранения сессий;
- описаны требуемые внешние ключи, триггеры;
- приведена ролевая модель для разграничения доступа.

3 Технологическая часть

3.1 Выбор СУБД

Для приложения необходимо выбрать СУБД, которые будут использоваться двумя сервисами: с основным "ядром" приложения, для хранения сессий.

Для основной логики была выбрана СУБД PostgreSQL. Для хранения сессий было выбрано in-memory хранилище Redis. Это хранилище удовлетворяет необходимым требованием (key-value хранилище), помимо этого, оно не требует дополнительной настройки для начала использования.

3.2 Выбор средства реализации

Система, взаимодействующая с базой данных, представляет из себя Web-сервер, доступ к которому осуществляется с помощью REST API [9]. Для реализации будет использоваться язык программирования Golang [10]. Этот язык создан для разработки микросервисных web приложений. В данном случае приложение будет состоять из двух микросервисов: сервис сессий и сервис бизнес-логики.

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

Для реализации REST будет использоваться web фреймворк echo [11]. Панель администратора строится на основе go-admin [12]. Документирование REST API будет осуществляться с помощью swagger [13], который поддерживает протокол openAPI [14].

Для сборки приложения в готовый продукт был выбран docker-compose [15] - оркестратор Docker контейнеров [16].

Docker позволяет изоилировать приложение и разворачивать его на любой машине, независимо от установленных зависимостей. Это реализуется благодаря наличию всех требуемых зависимостей внутри контейнера.

В отличии от виртуальных машин, имеющих хост-ОС и гостевую ОС. Контейнеры размещаются на одном физическом сервере с операционной системой хоста, которая разделяет их между собой. Совместное использование ОС хоста между контейнерами делает их менее требовательными к мощности

компьютера.

Docker-compose связывает контейнеры в одну систему - в отдельных контейнерах будет сообираться два микросервиса приложения, postgresql и redis.

3.3 Детали реализации

Выбор типа приложения

Система является серверным приложением, которое принимает запросы клиентов, обрабатываем их и возвращает ответ.

Клиенты могут взаимодействовать с сервером через REST API интерфейс. Это один из популярных подходов к взаимодейтствию в Web среде.

Для создания объектов в БД использовались миграции, которые приведены в листингах А.1 — А.10.

Для создания триггеров, инициализации ролевой модели использовались миграции, которые приведены в листингах Д.1 — Д.2.

Паттерны взаимодействия с СУБД PostgreSQL и Redis приведены в листингах Б.1 — Б.5 и В.1 — В.2.

Данные для базы данных был получены с помощью Веб скрапинга интернет сайта, в результате чего были сформированы json файлы, загружаемые в БД. Пример заполнения БД приведен в листингах Г.1 — **??**.

Приложение должно предоставлять доступ к разрабатываемой базе данных посредством REST API. Для этого будет предстоит разработать API.

`REST API` проектируемого приложения представлен в Таблице 3.1.

Таблица 3.1 – Описание `REST API` реализуемого приложения

Путь	Метод	Описание
/api/v1/offers	GET	Метод для получения списка подавших заявку на менторство менти
/api/v1/offers	POST	Метод для подачи заявки на занятия с ментором
/api/v1/offer/offer_id/accept	POST	Метод одобрения заявки на менторство (ментор принимает заявку менти)
/api/v1/offer/offer_id/accept	DELETE	Метод отклонения заявки на менторство (ментор отклоняет заявку менти)
/api/v1/plans	GET	Метод получения списка планов развития
/api/v1/plans/plan_id/task	POST	Метод для создания задачи в плане развития
/api/v1/plans/plan_id/status	POST	Метод изменения статуса задачи
/api/v1/user	GET PUT	Методы для взаимодействия с данными пользователя
/api/v1/user/status	GET PUT	Методы для взаимодействия с статусом пользователя (ментор/менти)
/api/v1/user/user_id	GET	Метод для получения информации о пользователе по его id
/api/v1/auth/telegram/register	GET	Проверка авторизации с использованием мессенджера Telegram
/api/v1/auth/telegram/login	GET	Метод получения сессии приложения после авторизации через Telegram

Продолжение на следующей странице

Таблица 3.1 – продолжение

Путь	Метод	Описание
/api/v1/auth/simple/register	POST	Метод для регистрации пользователи с использованием электронной почты и пароля
/api/v1/auth/simple/login	POST	Метод для входа пользователя в систему с использованием электронной почты и пароля
/api/v1/auth/token	GET	Метод для получения токена авторизации
/api/v1/logout	POST	Метод для выхода пользователя из системы
/api/v1/skills	GET	Метод получения всех поддерживаемых навыков
/api/v1/skills/users	GET	Метод получения всех пользователей с подходящими навыками конфигурациям нейронных сетей
		Конец таблицы

Для сборки частей системы использовались docker контейнеры, конфигурации которых представлены в листингах Е.1 - Е.3

Для их развертывания использовался docker-compose. Листинг конфигурации в yaml формате представлен на листингах Ж.1 – Ж.2.

Вывод

В данном разделе была определена СУБД, которая будет использоваться для решения задачи, выбран тип приложения. Кроме того, определены средства реализации приложения и представлен интерфейс взаимодействия с приложением (API), приведены детали реализации разрабатываемого приложения: создания ролевой модели, триггеров, сборка и развертывание системы.

4 Исследовательская часть

4.1 Цель исследования

Цель исследования - сравнить время, которое требуется для получения данных о пользователях приложения с использованием кэширования данных и без него.

Для достижения цели требуется:

- реализовать в приложении кеширование ответов;
- выбрать конечную точку, которая будет под нагрузкой и составить запрос;
- обеспечить наличие данных в базе данных;
- составить ленту запросов и выбрать профиль нагрузки;
- нагрузить выбранную конечную точку с использованием кеширования и без использования;
- проанализировать полученыые результаты (графики, метрики) и сформулирвоать выводы.

4.2 Описание исследования

Для исследования в приложении был реализован механизм кеширования ответов: на каждый запрос проверяются переданные аргументы, после чего в кеше осуществляется поиск ответа. Если ответа нет - происходит обращение к базе данных, данные сохраняются в кеш и отдаются пользователю.

При повторном запросе данные отдадутся из кеша. Запроса в базу данных не будет. За счет этого удается сократить время выполнения запроса, исключив поход в базу данных, который занимает определенное время.

В качестве кеша подключен экземпляр базы данных redis, чтобы не влиять на redis, использующийся для авторизации.

Для тестирования будет использоваться инструмент нагрузки и измерения производительности Web приложений - Yandex.Tank [17].

Для обработки результатов нагрузочного тестирования будет использоваться сервис Overload [18]. Он позволяет анализировать приложение под нагрузкой, визуализируя результаты в виде графиков.

Для тестирования была выбрана конечная точка */api/v1/skills/users* с параметром запроса *skills* равным всем возможным значениям из таблицы **skills**. Это позволит сильнее всего нагрузить приложения. Для получения всех пользователей придется выбрать с записи из таблицы **users** и применить фильтр для всех пользователей.

С помощью миграций реализовано заполнение базы данных.

Для генератора нагрузки (Yandex.Tank) необходимо выбрать тип нагрузки. Существует два типа нагрузки: открытая и закрытая.

Закрытая нагрузка подразумевает управление соединениями, в каждом из которых генератор подает запросы с нулевой задержкой между ними. Это создает эффект, при котором деградация тестируемой системы влечет за собой уменьшение потока запросов (при неизменном количестве соединений). Данный тип нагрузки используется для определения органической производительности тестируемой системы. Для нагрузки данного типа необходимо задать число N, где N - суммарное количество запросов, которое будет сделано к системе. Данный тип нагруки является менее жестким по сравнению с открытой.

Открытая нагрузка подразумевает управление количеством одновременных запросов в систему, которое не зависит от текущего состояния тестируемой системы и потенциально является более жесткой по сравнению с закрытой нагрузкой. Необходимо задать количество rps, где rps - количество одновременных запросов к приложению в секунду.

Для тестирования был выбран открытый тип, так как такая нагрузка сильнее влияет на систему, нежели закрытая.

В листинге 4.1 представлена конфигурация для генератора нагрузки.

Листинг 4.1 – Пример конфигурации для генератор нагрузки Yandex.Tank

```
phantom:
  address: 127.0.0.1
  load_profile:
    load_type: rps
    schedule: const(100, 1m)
  ammofile: /var/loadtest/ammo.txt
  ammo_type: uri
  timeout: 2s

overload:
  enabled: true
  token_file: /var/loadtest/token.txt
  job_name: load_test
  job_dsc: no cached test /api/v1/skills/users?
    skills=backend const 150 rps 1 min
```

В листинге 4.2 представлена лента запросов, использующаяся для тестирования.

Листинг 4.2 – Пример ленты запросов

```
[Connection: close]
[Host: localhost]
[Cookie: None]
/api/v1/skills/users?skills=backend,frontend,QA,devops,agile,
   recruitment,golang,js,databases,postgres,android,ios,kotlin,
   swift,c++,java,c#,Сообщество Онтико,DevRel,Code Review,Карьера
   ,HR,Project Management,System Design,Другое,Аналитика,
   Entrepreneurship,Product Management,Marketing,UX/UI/Design,
   Data Science/ML,Content/Copy,Cloud,Team Lead/Management,
   Собеседования,DevOps/SRE,Эксперт Авито,Сети,Наставник Яндекс.
   Практикума
/api/v1/skills/users?skills=backend,frontend,QA
/api/v1/skills/users?skills=agile
```

С такой лентой запроса на каждого клиента будет приходиться три запроса к приложению.

Для тестирования была выбрано два профиля нагрузки:

1. линейная - от 1 до 100 rps на протяжении пяти минут;
2. постоянная - 250 rps на протяжении минуты.

4.3 Технические характеристики

Исследование проводилось с использованием одного компьютера. Его технические характеристики:

- процессор: Apple M1 Pro;
- память: 32 Гб;
- операционная система: macOS Monterey [19] 12.4.

Исследование проводилось на ноутбуке, подключенным к сети электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения рабочего стола, окружением рабочего стола, а также непосредственно тестируемой системой.

4.4 Результаты исследования

Ниже приведены графики зависимости времени ответа конечной точки от количества rps, подаваемых генератором нагрузки.

На графиках показаны метрики rps, avg и 98 квантиль времени ответа.

rps (requests per second) - это количество запросов, подаваемых генератором в секунду.

avg (average) - это среднее время ответа конечной точки.

98 квантиль времени ответа - это значение времени ответа, до которого укладывается 98 % всех запросов.

4.4.1 Линейная нагрузка

График зависимости времени ответа конечной точки при линейной нагрузке без использования кеширования.

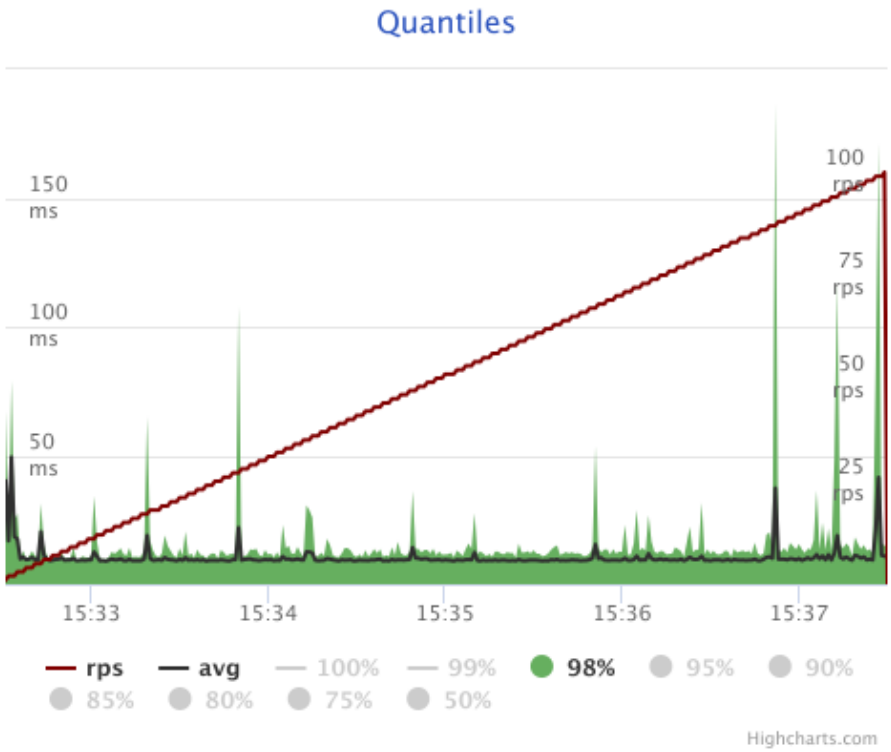

Рисунок 4.1 – График времени ответа конечной точки при линейной нагрузке без использования кеширования

График зависимости времени ответа конечной точки при линейной нагрузке с использования кеширования.

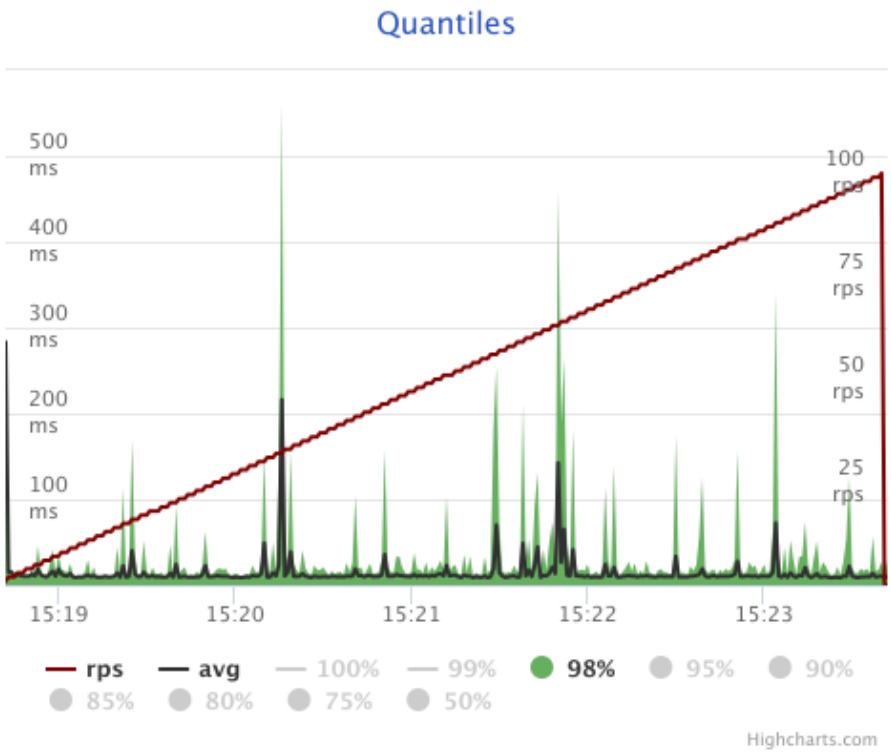

Рисунок 4.2 – График времени ответа конечной точки при линейной нагрузке с использованием кеширования

Тестирование показало, что при заданном количестве запросов (максимум 100 в секунду), кеширование не дает повышение производиельности в сравнении с временем ответа без кеширования.

Наоборот, появляются более частные "пики связанные с задержками в ожидании запросов из кеша и реляционной базы данных.

4.4.2 Постоянная нагрузка

График зависимости времени ответа конечной точки при постоянной нагрузке без использования кеширования.

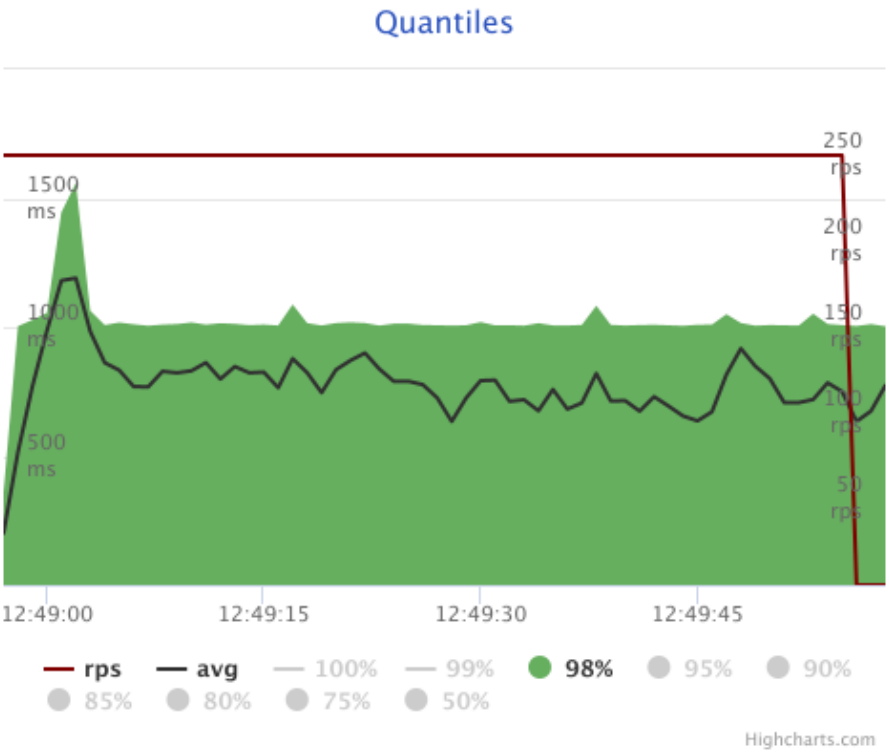

Рисунок 4.3 – График времени ответа конечной точки при постоянной нагрузке без использования кеширования

График зависимости времени ответа конечной точки при постоянной нагрузке с использования кеширования.

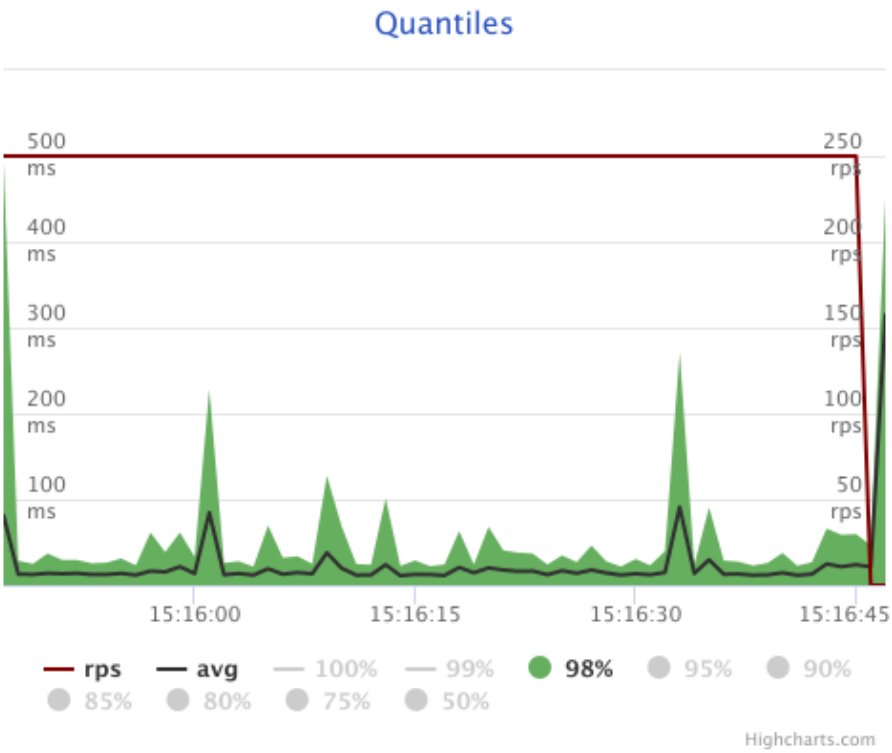

Рисунок 4.4 – График времени ответа конечной точки при постоянной нагрузке с использованием кеширования

По графикам видно, что кеширование при большой нагрузке (250 запросов в секунду) показывает лучшее время ответа в сравнении с тестированием без кеширования. Без кеша среднее время ответа - 1000 миллисекунд. С кешом - 15 миллисекунд.

Вывод

В ходе исследования было реализовано кеширование ответов приложения, выбрана и протестирована конечная точка */api/v1/skills/users*.

Исследование показало, что кеширование позволило уменьшить время ответа конечной точки.
При RPS = 250 время сократилось в 66 раз. Но при линейном изменении RPS от 1 до 100 уменьшения времени ответа не было, производительность ухудшилась: появились более частые пики, связанные с долгими ответами баз данных.

Внедрение кеширования может увеличить производительность на конечную точку при наличии высокой нагрузки. Перед добавление кеширования необходимо проанализировать конечную точку на предмет высокого коэффициента попадания запросов в кеш, иначе кеширование не улучшит производительность, а ухудшит ее.

ЗАКЛЮЧЕНИЕ

При выполнении курсовой работы была достигнута ее цель — спроектирована и разработана база данных для ля приложения по поиску наставника для изучения информационных технологий.

В процессе достижения данной цели были решены следующие задачи:

- проанализированы варианты представления данных и выбраны подходящие для решения задачи;
- проанализированы системы управления базами данных и выбраны подходящие для хранения данных;
- спроектирована базу данных, описаны ее сущности и связи;
- реализован интерфейс для доступа к базе данных;
- реализовано программное обеспечение, позволяющее взаимодействовать со спроектированной базой данных;

В ходе курсовой работы были получены знания в области проектирования баз данных, кеширования данных и нагрузочного приложения, предоставляющего интерфейс взаимодействия с базой данных.

Были изучены различные типы баз данных, способы хранения данных.

В результате проделанной работы, было разработано программное обеспечение, предоставляющее интерфейс для работы с базой данных. Была увеличена производительность с помощью внедрения механизма кеширования ответов.

В ходе выполнения исследовательской части работы было установлено, что кэширование данных при высокой нагрузке (250 запросов в секунду) приводит к росту производительности системы с точки зрения времени ответа. Время ответа уменьшилось в 66 раз по сравнению с вариантом без кэширования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Микросервисная архитектура. — URL: https://www.atlassian.com/ru/continuous-delivery/microservices (дата обр. 13.05.2022).
2. Модель баз данных. — URL: https://clck.ru/323j87 (дата обр. 16.09.2022).
3. MongoDB. — URL: https://www.mongodb.com/ (дата обр. 13.05.2022).
4. Tarantool. — URL: https://www.tarantool.io/ru/ (дата обр. 13.05.2022).
5. Redis. — URL: https://redis.io/ (дата обр. 13.05.2022).
6. Stateful vs stateless web services. — URL: https://nordicapis.com/defining-stateful-vs-stateless-web-services/ (дата обр. 21.08.2022).
7. Что такое сессии? — URL: https://developer.mozilla.org/ru/docs/Learn/Server-side/Django/Sessions#%D1%87%D1%82%D0%BE_%D1%82%D0%B0%D0%BA%D0%BE%D0%B5_%D1%81%D0%B5%D1%81%D1%81%D0%B8%D0%B8 (дата обр. 21.08.2022).
8. Telegram. — URL: https://telegram.org/ (дата обр. 19.08.2022).
9. What is REST? — URL: https://restfulapi.net/ (дата обр. 21.08.2022).
10. The Go Programming Language. — URL: https://go.dev/ (дата обр. 27.08.2022).
11. Echo High performance, extensible, minimalist Go web framework. — URL: https://echo.labstack.com/ (дата обр. 27.08.2022).
12. Backend Data Admin Panel. — 27.08.2022. — URL: https://www.go-admin.com/.
13. Swagger: API Documentation Design Tools for Teams. — URL: https://swagger.io/ (дата обр. 27.08.2022).
14. OpenAPI Specification v3.1.0. — URL: https://spec.openapis.org/oas/v3.1.0 (дата обр. 27.08.2022).
15. Overview of Docker Compose. — URL: https://docs.docker.com/compose/ (дата обр. 27.08.2022).

16. Docker. — URL: https://www.docker.com/ (дата обр. 27.08.2022).

17. Yandex.Tank - load and performance benchmark tool. — URL: https://github.com/yandex/yandex-tank (дата обр. 03.09.2022).

18. Overload a performance analytics service. — URL: https://overload.yandex.net/ (дата обр. 03.09.2022).

19. macOS Monterey. — URL: https://www.apple.com/ru/macos/monterey/ (дата обр. 03.09.2022).

ПРИЛОЖЕНИЕ А
Скрипты инициализации объектов БД

В Листингах А.1-А.10 представлены скрипты создания объектов БД.

Листинг А.1 – Скрипт создания объектов БД. Часть 1

```
CREATE EXTENSION IF NOT EXISTS citext;

CREATE TABLE users
(
    id bigserial not null,
    first_name citext,
    last_name citext,
    nickname citext not null unique,
    about         text          default '',
    avatar        text,
    is_searchable bool not null default false,
    created_at timestamptz default now()::timestamptz not null,
    updated_at timestamptz default now()::timestamptz not null,
    PRIMARY KEY (id)
);
CREATE TABLE telegram_auth
(
    tg_id   bigint not null,
    created_at timestamptz default now()::timestamptz not null,
    last_auth timestamptz default now()::timestamptz not null,
    user_id bigint,
    PRIMARY KEY (tg_id),
    FOREIGN KEY (user_id) references users (id)
);

CREATE TABLE plans
(
    id bigserial,
    name      text    not null,
    about     text,
    is_active boolean not null default false,
    progress  numeric          default 0,
    mentor_id bigint,
    mentee_id bigint,
    created_at timestamptz default now()::timestamptz not null,
    PRIMARY KEY (id),
    FOREIGN KEY (mentee_id) REFERENCES users (id)
);
```

Листинг А.2 – Скрипт создания объектов БД. Часть 2

```
CREATE TABLE color
(
    name citext,
    value int,
    PRIMARY KEY (name)
);

CREATE TABLE status
(
    name citext,
    color citext,
    PRIMARY KEY (name),
    FOREIGN KEY (color) REFERENCES color (name)
);

CREATE TABLE task
(
    id bigserial,
    name citext not null,
    description text not null,
    deadline timestamptz default now()::timestamptz not null,
    status citext not null,
    plan_id     bigint,
    created_at timestamptz default now()::timestamptz not null,
    PRIMARY KEY (id),
    FOREIGN KEY (plan_id) REFERENCES plans (id),
    FOREIGN KEY (status) REFERENCES status (name)
);

CREATE TABLE skills
(
    name citext not null,
    color citext,
    PRIMARY KEY (name),
    FOREIGN KEY (color) REFERENCES color (name)
);

CREATE TABLE users_skills
(
    id bigserial not null,
    user_id bigint not null,
    skill_name citext,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES users (id),
    FOREIGN KEY (skill_name) REFERENCES skills (name)
);
```

Листинг А.3 – Скрипт создания объектов БД. Часть 3

```
CREATE TABLE offers
(
    id bigserial not null,
    skill_name citext not null,
    status    boolean not null default false,
    mentor_id bigint  not null,
    mentee_id bigint  not null,
    created_at timestamptz default now()::timestamptz not null,
    PRIMARY KEY (id),
    FOREIGN KEY (skill_name) REFERENCES skills (name),
    FOREIGN KEY (mentor_id) REFERENCES users (id),
    FOREIGN KEY (mentee_id) REFERENCES users (id)
);
```

Листинг А.4 – Скрипт создания объектов БД. Часть 4

```
CREATE TABLE users_simple_auth
(
    id                 bigserial not null,
    login              text      not null unique,
    encrypted_password text      not null,
    user_id            bigint,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES users (id)
);
```

Листинг А.5 – Скрипт создания объектов БД. Часть 5

```
INSERT INTO skills(name)
VALUES ('backend'), ('frontend'), ('QA'), ('devops'), ('agile'),
       ('recruitment'), ('golang'), ('js'), ('databases'),
       ('postgres'), ('android'), ('ios'), ('kotlin'),
       ('swift'), ('c++'),('java'), ('c#');
```

Листинг А.6 – Скрипт создания объектов БД. Часть 6

```
delete from getme_db.public.skills where name = '';

ALTER TABLE getme_db.public.offers
    alter  skill_name drop default;
```

Листинг А.7 – Скрипт создания объектов БД. Часть 7

```
alter table getme_db.public.plans
    add
        constraint fk_mentor_id foreign key (mentor_id)
            references getme_db.public.users (id);
```

Листинг А.8 – Скрипт создания объектов БД. Часть 8

```
create table plans_skills
(
    id          bigserial not null,
    plan_id     bigint    not null,
    skill_name citext,
    PRIMARY KEY (id),
    FOREIGN KEY (plan_id) REFERENCES plans (id),
    FOREIGN KEY (skill_name) REFERENCES skills (name)
);
```

Листинг А.9 – Скрипт создания объектов БД. Часть 9

```
alter table getme_db.public.offers
alter column status set default true;

alter table getme_db.public.plans
    alter column is_active set default true;
```

Листинг А.10 – Скрипт создания объектов БД. Часть 10

```
INSERT INTO getme_db.public.status(name)
VALUES ('В процессе'),
       ('Выполнена'),
       ('Просрочена'),
       ('Отменена');
```

ПРИЛОЖЕНИЕ Б
Паттерны взаимодействия с PostgreSQL

В Листингах Б.1 – Б.5 приведен пример взаимодействия с PostgreSQL через приложение.

Листинг Б.1 – Взаимодействие с PostgreSQL с помощью паттерна репозиторий. Часть 1

```
package repository_postgresql

import (
    "database/sql"
    "fmt"

    "github.com/jmoiron/sqlx"
    "github.com/lib/pq"
    "github.com/pkg/errors"

    postgresql_utilits "getme-backend/internal/pkg/utilits/
       postgresql"
)

type UserRepository struct {
    store *sqlx.DB
}

func NewUserRepository(store *sqlx.DB) *UserRepository {
    return &UserRepository{
        store: store,
    }
}

const queryGetUserByNickname = `
SELECT id, first_name, last_name, nickname, about, avatar, tg_tag
   ,
is_searchable, created_at, updated_at from users where nickname =
    ?;`

func (repo *UserRepository) FindByNickname(nickname string) (*
   entities_user.User, error) {
    query := repo.store.Rebind(queryGetUserByNickname)

    user := &entities_user.User{}
    if err := repo.store.QueryRow(query, nickname).Scan(&user.ID,
        &user.FirstName, &user.LastName, &user.Nickname,
```

Листинг Б.2 – Взаимодействие с PostgreSQL с помощью паттерна репозиторий. Часть 2

```
        &user.About, &user.Avatar, &user.TgTag, &user.
           IsSearchable, &user.CreatedAt, &user.UpdatedAt); err
           != nil {
        if err == sql.ErrNoRows {
            return nil, postgresql_utilits.NotFound
        }
        if _, ok := err.(*pq.Error); ok {
            return nil, parsePQError(err.(*pq.Error))
        }
        return nil, postgresql_utilits.NewDBError(err)
    }

    return user, nil
}

const createFilledUserQuery = `INSERT INTO users (first_name,
   last_name, nickname, avatar)
        VALUES (?, ?, ?, ?) RETURNING id;`

func (repo *UserRepository) CreateFilledUser(data *entities_user.
   User) (int64, error) {
    query := repo.store.Rebind(createFilledUserQuery)
    ID := int64(-1)
    if err := repo.store.QueryRow(query, data.FirstName.String,
       data.LastName.String, data.Nickname, data.Avatar.String).
        Scan(&ID); err != nil {
        if _, ok := err.(*pq.Error); ok {
            return ID, parsePQError(err.(*pq.Error))
        }
        return ID, postgresql_utilits.NewDBError(err)
    }

    return ID, nil

}

const queryFindByIDWithSkill = `SELECT users.id, first_name,
   last_name, about, tg_tag,
            avatar, is_searchable, skill_name from users
    left join users_skills us on users.id = us.user_id
    left join skills s on us.skill_name = s.name
    where users.id = ?
`
```

Листинг Б.3 – Взаимодействие с PostgreSQL с помощью паттерна репозиторий. Часть 3

```
func (repo *UserRepository) FindByIDWithSkill(id int64) (*[]
   entities_user.UserWithSkill, error) {
    query := repo.store.Rebind(queryFindByIDWithSkill)
    user := &[]entities_user.UserWithSkill{}

    err := repo.store.Select(user, query, id)
    if err != nil {
        return nil, postgresql_utilits.NewDBError(err)
    }
    if len(*user) == 0 {
        return nil, postgresql_utilits.NotFound
    }

    return user, nil
}

const queryUpdateUser = `update users set
    first_name = COALESCE(NULLIF(TRIM(?), ''), first_name),
    last_name = COALESCE(NULLIF(TRIM(?), ''), last_name),
    about = COALESCE(NULLIF(TRIM(?), ''), about),
    tg_tag = COALESCE(NULLIF(TRIM(?), ''), tg_tag)
WHERE id = ?
returning first_name, last_name, nickname, about, tg_tag, avatar,
    is_searchable;`

const queryUpdateUserDeleteOldSkills = `DELETE FROM users_skills
   WHERE user_id = ?`

const queryUpdateUserAddSkills = `
INSERT INTO users_skills(user_id, skill_name) VALUES (:user_id, :
   skill_name)`

func (repo *UserRepository) UpdateUser(user *entities_user.
   UserWithSkills) (*entities_user.UserWithSkills, error) {
    query := repo.store.Rebind(queryUpdateUser)
    queryDelete := repo.store.Rebind(
       queryUpdateUserDeleteOldSkills)
    userFromDB := &entities_user.UserWithSkills{}
```

Листинг Б.4 – Взаимодействие с PostgreSQL с помощью паттерна репозиторий. Часть 4

```
    tx, err := repo.store.Beginx()
    if err != nil {
        return nil, postgresql_utilits.NewDBError(err)
    }

    err = tx.QueryRowx(query, user.FirstName.String, user.
       LastName.String, user.About.String, user.TgTag.String,
       user.ID.Int64).
        Scan(&userFromDB.FirstName, &userFromDB.LastName, &
           userFromDB.Nickname, &userFromDB.About, &userFromDB.
           TgTag, &userFromDB.Avatar, &userFromDB.IsSearchable)
    if err != nil {
        _ = tx.Rollback()
        return nil, postgresql_utilits.NewDBError(err)
    }

    if len(user.Skills) != 0 {
        if _, err = tx.Exec(queryDelete, user.ID); err != nil {
            _ = tx.Rollback()
            return nil, postgresql_utilits.NewDBError(err)
        }

        args := entities_user.ToUsersSkills(user.ID.Int64, user.
           Skills)
        if _, err = tx.NamedExec(queryUpdateUserAddSkills, args);
            err != nil {
            _ = tx.Rollback()
            return nil, postgresql_utilits.NewDBError(err)
        }
    }

    if err = tx.Commit(); err != nil {
        return nil, postgresql_utilits.NewDBError(err)
    }

    return userFromDB, nil
}

const queryGetMenteeByOffers = `
SELECT offers.id as offer_id, users.id as id, first_name,
last_name, tg_tag, about, avatar, is_searchable from users join
   getme_db.public.offers
```

Листинг Б.5 – Взаимодействие с PostgreSQL с помощью паттерна репозиторий. Часть 5

```
on users.id = offers.mentee_id and offers.mentor_id = ? and
   offers.status = true;‘

func (r *UserRepository) GetMenteeByMentorWithOfferID(mentorID
   int64) ([]entities_user.UserWithOfferID, error) {
    users := &[]entities_user.UserWithOfferID{}

    query := r.store.Rebind(queryGetMenteeByOffers)

    if err := r.store.Select(users, query, mentorID); err != nil
       {
        return nil, postgresql_utilits.NewDBError(
            errors.Wrap(err,
                fmt.Sprintf(
                    "UserRepository: GetMenteeByMentorWithOfferID
                       (%v)", mentorID)))
    }

    return *users, nil
}
```

ПРИЛОЖЕНИЕ В
Паттерны взаимодействия с Redis

В Листингах В.1 – В.2 приведен пример взаимодействия с Redis через приложение.

Листинг В.1 – Взаимодействие с Redis в сервисе авторизации. Часть 1

```
package repository

import (
    "github.com/pkg/errors"

    "github.com/gomodule/redigo/redis"
    "github.com/sirupsen/logrus"

    "getme-backend/internal/microservices/auth/sessions/models"
)

type RedisRepository struct {
    redisPool *redis.Pool
    log       *logrus.Logger
}

func NewRedisRepository(pool *redis.Pool, log *logrus.Logger) *
   RedisRepository {
    return &RedisRepository{
        redisPool: pool,
        log:       log,
    }
}

func (repo *RedisRepository) Set(session *models.Session) error {
    con := repo.redisPool.Get()
    defer con.Close()

    res, err := redis.String(con.Do("SET", session.UniqID,
       session.UserID,
        "PX", session.Expiration))
    if res != "OK" {
        return errors.Wrapf(err,
            "error when try create session with uniqId: %s, and
               userId: %s",
```

Листинг В.2 – Взаимодействие с Redis в сервисе авторизации. Часть 2

```
            session.UniqID, session.UserID)
    }
    return nil
}

func (repo *RedisRepository) GetUserId(uniqID string,
   updExpiration int) (string, error) {
    con := repo.redisPool.Get()
    defer con.Close()

    res, err := redis.String(con.Do("GET", uniqID))
    if err != nil {
        return "", errors.Wrapf(err,
            "error when try get session with uniqId: %s", uniqID)
    }

    _, err = redis.Int64(con.Do("EXPIRE", uniqID, updExpiration
       /100))
    if err != nil {
        return "", errors.Wrapf(err,
            "error when try update expire session with uniqId: %s
               ", uniqID)
    }
    return res, nil
}

func (repo *RedisRepository) Del(session *models.Session) error {
    con := repo.redisPool.Get()
    defer con.Close()

    _, err := redis.Int(con.Do("DEL", session.UniqID))
    return errors.Wrapf(err,
        "error when try delete session with uniqId: %s, and
           userId: %s",
        session.UniqID, session.UserID)
}
```

ПРИЛОЖЕНИЕ Г
Скрипт заполнения БД

В Листингах Г.1 – Г.3 приведен скрипт с примером заполнения базы данных:

Листинг Г.1 – Скрипт с примером заполнения БД. Часть 1

```
CREATE TABLE json_table (
                            id int PRIMARY KEY,
                            airtableId text,
                            slug text,
                            name text,
                            workplace text,
                            description text,
                            about text,
                            competencies text,
                            experience text,
                            price text,
                            menteeCount int,
                            photo json,
                            photo_url text,
                            tags text[],
                            sortOrder int,
                            is_visible bool,
                            sponsors text,
                            calendarType text
);

WITH json (doc) AS (values ('[
  {
    "id": 336,
    "airtableId": "recBcoVPUSaOKLEqe",
    "slug": "kseniia-pomogaeva-336",
    "name": "Ксения Помогаева",
    "job": "Методолог IT обучения",
    "workplace": "Тинькофф Банк, сертифицированный скрам-мастер",
    "description": "Выстраивание стратегии личного и командного
       обучения в зависимости от целей (как hard, так и soft),
       помощь в настройке процессов agile-команд, помощь в
       реализации программ менторинга, наставничества и обучения
       внутри компании.",
    "about": "",
    "competencies": "",
    "experience": "2-5",
    "price": "По договоренности",
    "menteeCount": 0,
    "photo": {
      "id": "attBPUGR0pXxjGdYw",
      "width": 709,
```

Листинг Г.2 – Скрипт с примером заполнения БД. Часть 2

```
      "height": 1057,
      "url": "https://dl.airtable.com/.attachments/
         d8607779d3ada78c9fad792ed2ece2ea/968154fa/
         photo5287487268799493337.jpg?ts=1652723915\u0026userId=
         usrW6Ciyt4Mp0Nk6M\u0026cs=40ef2657e89cfca1",
      "filename": "photo5287487268799493337.jpg",
      "size": 193224,
      "type": "image/jpeg",
      "thumbnails": {
        "small": {
          "url": "https://dl.airtable.com/.attachmentThumbnails
             /568b737de7b03ef5bc2b87f0076e2911/2a0499a6?ts
             =1652723915\u0026userId=usrW6Ciyt4Mp0Nk6M\u0026cs=46
             bbf92159b35348",
          "width": 24,
          "height": 36
        },
        "large": {
          "url": "https://dl.airtable.com/.attachmentThumbnails
             /318c8a94e2a386ef8b4a2aea9954b232/cdab9fc6?ts
             =1652723915\u0026userId=usrW6Ciyt4Mp0Nk6M\u0026cs=
             cd53673135752e05",
          "width": 512,
          "height": 763
        },
        "full": {
          "url": "https://dl.airtable.com/.attachmentThumbnails
             /07fd1c6d9bb51efe25623d47b84531f3/bcce4de0?ts
             =1652723915\u0026userId=usrW6Ciyt4Mp0Nk6M\u0026cs
             =94784d2481c820d8",
          "width": 3000,
          "height": 3000
        }
      }
    },
    "photo_url": "https://dl.airtable.com/.attachments/
       d8607779d3ada78c9fad792ed2ece2ea/968154fa/
       photo5287487268799493337.jpg",
    "tags": [
      "Agile",
      "HR",
      "Product Management",
      "Team Lead/Management",
      "Другое"
    ],
    "sortOrder": 4,
    "is_visible": true,
    "sponsors": "none",
    "calendarType": "none"
  }
]'::json))
```

Листинг Г.3 – Скрипт с примером заполнения БД. Часть 3

```
INSERT INTO json_table
    (id, airtableId, slug, name, workplace,
    description, about, competencies,
     experience, price, menteeCount,
    photo, photo_url, tags, sortOrder,
     is_visible, sponsors, calendarType)
SELECT
    id, airtableId, slug, name, workplace,
    description, about, competencies,
    experience, price, menteeCount,
    photo, photo_url, tags,
    sortOrder, is_visible, sponsors,
    calendarType from json l
cross join lateral
    json_populate_recordset(null::json_table, doc) as p;


INSERT INTO users (first_name, last_name, nickname,
                   avatar, about, is_searchable)
SELECT (string_to_array(p.name, ' '))[1],
       (string_to_array(p.name, ' '))[2],
       p.slug, p.photo_url,
       concat(p.about, ' ', p.description), p.is_visible
From json_table as p;
INSERT INTO skills (name)
SELECT DISTINCT UNNEST(tags) FROM json_table ON CONFLICT DO
   NOTHING;

INSERT INTO users_skills (user_id, skill_name) (
    SELECT u.id, UNNEST(json_table.tags) from users as u
             JOIN json_table on json_table.slug = u.nickname
)
```

ПРИЛОЖЕНИЕ Д

Скрипт инициализации ролевой модели базы данных

В Листинге Д.1 приведен скрипт создания ролевой модели базы данных, а именно - создания и настройки соответствующих прав доступа для трех ролей:

Листинг Д.1 – Скрипт инициализации ролевой модели базы данных. Часть 1

```
-- Администратор
CREATE ROLE administrators;
GRANT USAGE ON SCHEMA public TO administrators;
GRANT SELECT, INSERT, UPDATE, DELETE
    ON ALL TABLES
    IN SCHEMA public
    TO administrators;

ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT SELECT, INSERT, UPDATE, DELETE
    ON TABLES TO administrators;

CREATE USER admin
    WITH
    CREATEDB
    CREATEROLE
    ENCRYPTED PASSWORD 'qwerty'
    IN ROLE administrators;
-- Гость
CREATE ROLE guest;
CREATE USER visitor
    WITH ENCRYPTED PASSWORD 'qwerty'
    IN ROLE guest;

GRANT SELECT
    ON TABLE users, users_simple_auth, users_skills,
       telegram_auth,task,status,skills,plans_skills,plans,offers
       ,color
    TO visitor;
-- Пользователь
CREATE ROLE users;
CREATE USER getme
    WITH ENCRYPTED PASSWORD 'getme-app'
    IN ROLE users;

GRANT SELECT, INSERT, UPDATE, DELETE
    ON TABLE users, users_simple_auth, users_skills,
       telegram_auth,task,status,skills,plans_skills,plans,offers
       ,color
    TO getme;
```

В Листинге Д.2 приведен скрипт создания триггера для обновления прогресса выполнения плана.

Листинг Д.2 – Скрипт создания триггера для обновления прогресса выполнения плана.

```
create or replace function update_progress() returns trigger as
$psql$
begin
    UPDATE plans
    SET progress = COALESCE((
(select count(*) from task where plan_id = new.plan_id and
    status = 'Выполнена')::float * 100::float) / NULLIF(
                    (select count(*)
                        from task
                        where plan_id = new.plan_id),
                        0)::float, 100)::numeric
    WHERE id = new.plan_id;
    return new;
end;
$psql$ language plpgsql;

create trigger update_progress
    after update
    on task
    for each row
execute procedure update_progress();

create trigger insert_progress
    after insert
    on task
    for each row
execute procedure update_progress();

alter table plans
    ALTER progress SET default 100;

alter table task
    ALTER status SET default 'В процессе';
```

ПРИЛОЖЕНИЕ Е
Сборка приложения

В Листинге Е.1 представлен скрипт создания docker-образа базы данных.

Листинг Е.1 – Dockerfile для базы данных

```
FROM postgres:14

RUN apt-get update && apt-get install -y --no-install-recommends
   apt-utils
RUN apt-get update \
      && apt-cache showpkg postgresql-$PG_MAJOR-rum \
      && apt-get install -y --no-install-recommends \
           postgresql-$PG_MAJOR-rum  postgresql-$PG_MAJOR-rum-
              dbgsym\
      && rm -rf /var/lib/apt/lists/*

RUN apt-get update && apt-get install -y git && apt-get install -
   y make
RUN git clone https://github.com/postgrespro/hunspell_dicts
WORKDIR /hunspell_dicts/hunspell_ru_ru
RUN make USE_PGXS=1 install
RUN rm -rf ./hunspell_dicts

WORKDIR /


RUN mkdir -p /docker-entrypoint-initdb.d
```

В Листингах Е.2 и Е.3 представлены скрипты создания docker-образов микросервисов приложения.

Листинг Е.2 – Dockerfile для сервиса основного приложения

```
FROM golang:1.17.1 as builder

WORKDIR /app

EXPOSE 80

COPY . .

RUN apt-get update && apt-get install jq -y && chmod +x ./wait
# Если что-то не собирается из-за CGO, может быть, при проверке
   сертификатов из гошки.
# Убрать CGO_ENABLED
# Итоговый image взять с gcc, например ubuntu
RUN CGO_ENABLED=0 make build

FROM alpine

COPY --from=builder /app /app

WORKDIR /app

CMD ls && ./wait && ./server.out
```

Листинг Е.3 – Dockerfile для сервиса авторизации

```
FROM golang:1.17.1 as builder

WORKDIR /app

EXPOSE 8080 443 80 5001

COPY ../../../../../app/scripts .

RUN CGO_ENABLED=0 make build-sessions

FROM alpine

WORKDIR /app

COPY --from=builder /app/sessions.out ./
COPY --from=builder /app/configs ./app/configs
COPY --from=builder /app/logs ./app/logs


CMD ./sessions.out
```

ПРИЛОЖЕНИЕ Ж
Развертывание приложения

Для развертывания приложения использовался docker-compose. С помощью него запускается приложение, состоящее из docker контейнеров.

В Листингах Ж.1 – Ж.2 представлена конфигурация развертывания приложения.

Листинг Ж.1 – Конфигурация развертывания приложения. Часть 1

```
version: "3"

volumes:
  postgis-data:
  session-redis:
services:
  main:
    image: getme-main
    expose:
      - "80"
    ports:
      - "80:80"
    networks:
      - default
    volumes:
      - ./api:/app/api
      - ./logs:/app/logs
      - ./media:/app/media
      - ${CONFIG_DIR}:/app/configs
    depends_on:
      - getme-db
      - session-service
    environment:
      WAIT_HOSTS: getme-db:5432

  session-redis:
    image: "redis:alpine"
    expose:
      - "6379"
    ports:
      - "6379:6379"
    volumes:
      - session-redis:/data
    restart: always
```

Листинг Ж.2 – Конфигурация развертывания приложения. Часть 2

```
  session-service:
    image: session-service
    expose:
      - "5001"
    ports:
      - "5001:5001"
    networks:
      - default
    volumes:
      - ./logs-sessions:/app/logs
      - ${CONFIG_DIR}:/app/configs
    depends_on:
      - session-redis
    environment:
      WAIT_HOSTS: getme-db:5432
  getme-db:
    image: pg-14
    expose:
      - "5432"
    command: "postgres -c shared_preload_libraries='
       pg_stat_statements'"
    volumes:
      - postgis-data:/var/lib/postgresql
    environment:
      - POSTGRES_PASSWORD=${PG_BD_PASSWORD}
      - POSTGRES_USER=${PG_BD_USERNAME}
      - POSTGRES_DB=${PG_BD_NAME}
    ports:
      - "5432:5432"
    restart: on-failure
    healthcheck:
      test: "exit 0"
    deploy:
      resources:
        limits:
          cpus: '2'
          memory: 4G
networks:
  default:
    driver: bridge
```