

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА/GRADUATION THESIS

Автоматизация разработки хранилищ данных на базе GraphQL

Автор/ Author

Ярошевич Александр Владимирович

Направленность (профиль) образовательной программы/Major

Программно-информационные системы 2017

Квалификация/ Degree level

Бакалавр

Руководитель ВКР/ Thesis supervisor

Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО,
факультет программной инженерии и компьютерной техники, доцент (квалификационная
категория "ординарный доцент")

Группа/Group

P3417

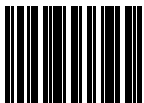
Факультет/институт/кластер/ Faculty/Institute/Cluster

факультет программной инженерии и компьютерной техники

Направление подготовки/ Subject area

09.03.04 Программная инженерия

Обучающийся/Student

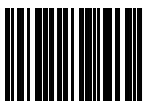
Документ подписан	
Ярошевич Александр Владимирович	
29.05.2021	

(эл. подпись/ signature)

Ярошевич
Александр
Владимирович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Пенской Александр Владимирович	
30.05.2021	

(эл. подпись/ signature)

Пенской
Александр
Владимирович

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Ярошевич Александр Владимирович

Группа/Group Р3417

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет программной инженерии и компьютерной техники

Квалификация/ Degree level Бакалавр

Направление подготовки/ Subject area 09.03.04 Программная инженерия

Направленность (профиль) образовательной программы/Major Программно-информационные системы 2017

Специализация/ Specialization

Тема ВКР/ Thesis topic Автоматизация разработки хранилищ данных на базе GraphQL

Руководитель ВКР/ Thesis supervisor Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Срок сдачи студентом законченной работы до / Deadline for submission of complete thesis 31.05.2021

Техническое задание и исходные данные к работе/ Requirements and premise for the thesis

Изучить существующие инструменты автоматизации разработки систем хранения данных. Исследовать языки взаимодействия с хранилищами данных. Разработать программное средство, генерирующее по входным спецификациям реализацию GraphQL API для взаимодействия и хранения данных. Полученное программное решение должно упростить и ускорить процесс разработки клиент-серверных приложений путем автоматизации разработки серверной части приложения.

Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)/ Content of the thesis (list of key issues)

- 1) Введение. Актуальность темы. Постановка задачи.
- 2) Анализ существующих инструментов автоматизации разработки систем хранения данных.
- 3) Анализ языков взаимодействия с хранилищами данных.
- 4) Разработка демонстрационного проекта.
- 5) Разработка и тестирование генератора GraphQL API для взаимодействия с данными по

входным спецификациям.

6) Выводы.

Перечень графического материала (с указанием обязательного материала) / List of graphic materials (with a list of required material)

Исходные материалы и пособия / Source materials and publications

1) Спецификация GraphQL [Электронный ресурс]. - Режим доступа: <https://spec.graphql.org/June2018/> свободный (дата обращения: 28.04.2021)

2) Документация к PostgreSQL [Электронный ресурс]. - Режим доступа: <https://www.postgresql.org/docs/> свободный (дата обращения: 28.04.2021)

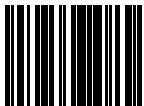
3) Сравнение подходов автоматизации разработки бекенда [Электронный ресурс]. - Режим доступа: <https://core.ac.uk/download/pdf/80989262.pdf> свободный (дата обращения: 28.04.2021)

4) Документация к Golang [Электронный ресурс]. - Режим доступа: <https://golang.org/doc/> свободный (дата обращения: 29.04.2021)

Дата выдачи задания/ Objectives issued on 30.04.2021

СОГЛАСОВАНО / AGREED:

Руководитель ОП/ Head
of educational program

Документ подписан	
Муромцев Дмитрий Ильич	
20.05.2021	

(эл. подпись)

Муромцев
Дмитрий Ильич

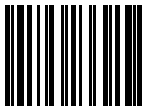
Руководитель ВКР/
Thesis supervisor

Документ подписан	
Пенской Александр Владимирович	
30.04.2021	

(эл. подпись)

Пенской
Александр
Владимирович

Задание принял к
исполнению/ Objectives
assumed by

Документ подписан	
Ярошевич Александр Владимирович	
30.04.2021	

(эл. подпись)

Ярошевич
Александр
Владимирович

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ /
SUMMARY OF A GRADUATION THESIS**

Обучающийся/ Student

Ярошевич Александр Владимирович

Наименование темы ВКР / Title of the thesis

Автоматизация разработки хранилищ данных на базе GraphQL

Наименование организации, где выполнена ВКР/ Name of organization

Университет ИТМО

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ/
DESCRIPTION OF THE GRADUATION THESIS**

1. Цель исследования / Research objective

Упростить разработку клиент-серверных приложений путем автоматической генерации системы хранения данных на основе схемы GraphQL.

2. Задачи, решаемые в ВКР / Research tasks

1) Обзор существующих инструментов автоматизации разработки систем хранения данных. 2) Анализ способов и языков взаимодействия с системами хранения данных. 3) Разработка демонстрационного проекта. 4) Разработка генератора систем хранения данных по входной спецификации. 5) Сравнение результатов работы генератора с демонстрационным проектом.

3. Краткая характеристика полученных результатов / Short summary of results/conclusions

Произведен анализ существующих решений для автоматизации разработки хранилищ данных и взаимодействия с ними. Разработан демонстрационный проект. Разработан генератор позволяющий сгенерировать систему хранения данных на основе входной спецификации в виде схемы GraphQL. Произведено сравнение результатов работы генератора и демонстрационного проекта. Сформулированы планы по развитию проекта.

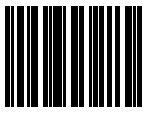
4. Наличие публикаций по теме выпускной работы/ Have you produced any publications on the topic of the thesis

5. Наличие выступлений на конференциях по теме выпускной работы/ Have you produced any conference reports on the topic of the thesis

6. Полученные гранты, при выполнении работы/ Grants received while working on the thesis

7. Дополнительные сведения/ Additional information

Обучающийся/Student

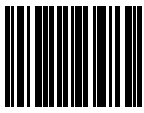
Документ подписан	
Ярошевич Александр Владимирович	
29.05.2021	

(эл. подпись/ signature)

Ярошевич
Александр
Владимирович

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Пенской Александр Владимирович	
30.05.2021	

(эл. подпись/ signature)

Пенской
Александр
Владимирович

(Фамилия И.О./ name
and surname)

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ	6
СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ	8
ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ	9
ВВЕДЕНИЕ	11
1 РЕШЕНИЯ ДЛЯ АВТОМАТИЗАЦИИ СОЗДАНИЯ СЕРВЕРНЫХ ПРИЛОЖЕНИЙ	13
1.1 Мобильный бэкенд как сервис	13
1.1.1 Платформа Backendless	14
1.1.2 Платформа “Parse”	15
1.1.3 Платформа “AWS Amplify”	17
1.1.4 Платформа “Firebase”	18
1.1.5 Хранилище данных “FaunaDB”	18
1.3 Способы взаимодействия с хранилищами данных	19
1.3.1 Архитектурный стиль “REST”	19
1.3.2 Язык запросов “GraphQL”	20
1.4 Выводы	23
2 РАЗРАБОТКА ГЕНЕРАТОРА СЕРВЕРНЫХ ПРИЛОЖЕНИЙ ДЛЯ ОБРАБОТКИ И ХРАНЕНИЯ ДАННЫХ	25
2.1 Требования к генератору	25
2.2 Технологии и инструменты разработки	25
2.3 Демонстрационное приложение	26
2.4 Разработка генератора	30
2.5 Выводы	35
3. РЕЗУЛЬТАТЫ И ПЕРСПЕКТИВЫ РАЗВИТИЯ	36
3.1 Сравнение демонстрационного приложения и сгенерированного	36
3.2 Перспективы развития	38
3.3 Выводы	38
ЗАКЛЮЧЕНИЕ	39
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	40
ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ДЛЯ СОЗДАНИЯ ПОЛЬЗОВАТЕЛЯ В ДЕМОНСТРАЦИОННОМ ПРИЛОЖЕНИИ	43

ПРИЛОЖЕНИЕ Б. ФРАГМЕНТ ЛИСТИНГА РАЗРАБОТАННОГО ГЕНЕРАТОРА	45
--	-----------

ПРИЛОЖЕНИЕ В. СХЕМА GraphQL ПОЛУЧЕННАЯ В РЕЗУЛЬТАТЕ РАБОТЫ ГЕНЕРАТОРА	49
--	-----------

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

API - Application Programming Interface

HTTP - HyperText Transfer Protocol

MBaaS - Mobile Backend as a Service

ORM - Object-Relational Mapping

REST - Representational State Transfer

SDK - Software Development Kit

URI - Uniform Resource Identifier

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Android – операционная система для смартфонов, планшетов, электронных книг, наручных часов, фитнес-браслетов, телевизоров, проекторов и других устройств [1]

Golang – компилируемый многопоточный язык программирования, разработанный внутри компании Google [2]

JavaScript – это легковесный, интерпретируемый или JIT-компилируемый, объектно-ориентированный язык с функциями первого класса [3]

Object-Relational Mapping (ORM) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных» [4]

Software development kit (SDK) – набор средств разработки, позволяющий специалистам по программному обеспечению создавать приложения для определённого пакета программ, программного обеспечения базовых средств разработки, аппаратной платформы, компьютерной системы, игровых консолей, операционных систем и прочих платформ [5]

Абстрактное синтаксическое дерево – помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами.

Таким образом, листья являются пустыми операторами и представляют только переменные и константы [6]

Компилятор – программа, которая переводит текст, написанный на языке программирования, в набор машинных кодов [7]

Программный интерфейс приложения – описание способов (набор классов, процедур, функций, структур и констант), которыми одна компьютерная

программа может взаимодействовать с другой программой. Обычно входит в описание какого-либо интернет-протокола, программного каркаса (фреймворка) или стандарта вызовов функций операционной системы [8]

Спецификация OpenAPI – формализованная спецификация и экосистема множества инструментов, предоставляющая интерфейс между front-end системами, кодом библиотек низкого уровня и коммерческими решениями в виде API

ВВЕДЕНИЕ

С каждым днем в мире все больше и больше различных приложений, это как и мобильные и веб приложения, так и приложения для интернета вещей и умных устройств. По отчетом аналитиков из Sensor Tower ежегодное количество первичных скачиваний мобильных приложений из App Store и Play Market вырастет на 61% до 230 миллиардов к 2025 году по сравнению с показателем 2020 года [9]. Большинство из этих приложений состоят из клиентской и серверной части. Это различные форумы, интернет-магазины, приложения ресторанов и кафе. Зачастую, серверная часть приложения заключается только в хранение и обработки данных. Но разработка серверного приложения требует отдельных ресурсов и разработчиков, в том числе и для монотонных и шаблонных задач. Это усложняет и замедляет разработку. В множестве случаев разработку серверного приложения можно автоматизировать, имея четкое описание хранимых сущностей и требований к программному интерфейсу приложения. В качестве такого описания может послужить схема GraphQL API.

Целью настоящей работы является проектирование и разработка инструмента для генерации приложения отвечающего за обработку и хранение данных согласно входным требованиям описанным в формате схемы GraphQL.

Для достижения поставленной цели необходимо решить следующий ряд задач:

- 1) провести обзор существующих решений автоматизации разработки систем хранения данных;
- 2) провести обзор способов и языков взаимодействия с систем хранения данных;
- 3) разработать демонстрационный проект;

- 4) разработать генератор реализующий систему хранения данных доступную GraphQL API на основе входной спецификации.

1 РЕШЕНИЯ ДЛЯ АВТОМАТИЗАЦИИ СОЗДАНИЯ СЕРВЕРНЫХ ПРИЛОЖЕНИЙ

На сегодняшний день существует множество различных программных решений для автоматизации разработки серверных приложений для обработки и хранения данных. В данном разделе будет приведен обзор нескольких приложений с разными подходами к автоматизации. Среди этих приложений можно выделить как коммерческие, которые распространяются на условиях подписки, так и те, которые распространяются бесплатно и имеют открытый исходный код. Также, будут разобраны различные языки и способы для взаимодействия с хранилищами данных.

1.1 Мобильный бэкенд как сервис

Согласно Carney [10], мобильный бэкенд как сервис или “Mobile Backend as a Service” (MBaaS) - это услуга, которая предоставляет способ подключения приложения к облачному серверному хранилищу для людей, разрабатывающих сетевые и мобильные приложения. Функционал MBaaS может включать в себя не только хранение и управление данными, но и такие функции, как регистрация и авторизация пользователей, отправка уведомления на мобильные устройства, хранение файлов, сервисы определения геолокации и различные интеграция с социальными сетями. Набор услуг предлагается с помощью SDK и API. Помимо MBaaS существует понятие “Backend as a Service “(BaaS), которое отличается от MBaaS своей ориентированностью на сетевые приложения, нежели мобильные. Оба понятия предоставляют схожий набор функциональности и зачастую взаимозаменяемы.

1.1.1 Платформа Backendless

“Backendless” это платформа бэкенда как сервис, которая предоставляет готовую облачную серверную инфраструктуру для мобильных и сетевых приложений по платной ежемесячной подписке. Платформа состоит из нескольких программных интерфейсов доступных по средствам REST и SDK для JavaScript, Flutter, Android, iOS. “Backendless” включает в себя различные программные интерфейсы, в том числе и для управления пользователями, взаимодействия с реляционной базой данных, управления пуш уведомлениями, хранения файлов.

Программный интерфейс для управления пользователями отвечает за регистрацию и авторизацию пользователей [11]. Также обеспечивается проверка существования пользователя, поддержка пользовательских сессий, обновление учетной записи пользователя, отправка уведомлений на почту и прочая настраиваемая бизнес логика.

Программный интерфейс для управления данными позволяет хранить, обновлять, удалить и осуществлять поиск по данным [11]. Для создания таблицы в базе данных необходимо описать соответствующую структуру в клиентском приложении, “Backendless” автоматически создаст таблицу и связи в реляционной базе данных. Также, можно создать таблицу используя графический клиент платформы.

Клиентские приложения могут обмениваться сообщениями в режиме реального времени путем создания издателей (publishers) и подписчиков (subscribers) используя программный интерфейс для обмена сообщениями [11]. Сообщениями могут быть любые дискретные данные. Доставка сообщения может быть осуществлена в виде пуш уведомлений путем нативной интеграции с мобильными устройствами на iOS и Android.

“Backendless” включает в себя набор сервисов для потокового видео и аудио. Клиенты могут как и проигрывать аудио и видео в реальном времени

так и вещать потоковые данные, что позволяет реализовать в клиентском приложении видео звонки и стриминговый функционал.

1.1.2 Платформа “Parse”

“Parse” это одна из первых BaaS платформ изначально развивавшиеся как коммерческое приложение предоставляющие большой набор функционала наподобие Backendless. В 2016 году компания Facebook прекратил поддержку Parse и значительная часть приложения стало бесплатным открытым программным обеспечением именуемым Parse Server [12]. Для работы Parse Server требуется собственные ресурсы для хостинга, также не весь функционал Parse доступен в Parse Server [13].

Таблица 1 - Сравнение функционала Parse и Parse Server

Функциональность	Наличие функциональности в Parse	Наличие функциональности в Parse Server
1	2	3
Локальная разработка и тестирование	да	да
Возможность выбора собственного хостинга	нет	да
Контроль над базой данных	нет	да
Контроль над индексами в базе данных	нет	да
Ограничение количества запросов к базе данных	да	нет
Настройка файлового хранилища	нет	да

Продолжение таблицы 1

1	2	3
Открытый исходный код и изменения	нет	да
Средства аналитики	нет	нет
Аутентификация	да	да
Сервис конфигурации	да	нет
Уведомления	да	нет
Поиск	да	да
Валидация покупок внутри приложений	да	нет
Фоновые процессы	да	нет
Информирование по электронной почте	да	нет
Схема API	да	да
Система уведомлений о событиях	да	нет
Система мониторинга сбоев	да	нет
Логирование	да	да
Интерактивная панель управления	да	да
Сервис запросов	да	да
Управления пользователями	да	да
Поддержание пользовательский сессий	да	да

Продолжение таблицы 1

1	2	3
Поддержание пользовательский сессий	да	да
Пользовательский роли	да	да
Хранение файлов	да	да
Сервис геолокаций	да	да
Сервис поиска по данным	да	да

Функционал Parse Server имеет больше возможностей настройки и управления серверным приложениям, но это в тоже время это делает его сложнее и менее удобным. Parse имеет ряд полезных возможностей недоступных в Parse Server, например валидаций покупок внутри приложений и сервис оповещения о событиях.

1.1.3 Платформа “AWS Amplify”

“AWS Amplify” – это набор различных средств и сервисов предназначенные для разработчиков мобильных приложений и веб-интерфейсов для создания серверных приложений на базе инфраструктуры AWS. Данное решение позволяет в короткие сроки сконфигурировать все необходимое для работы мобильного или веб-приложения. Будь то серверная логика, хранение данных, авторизация пользователей или обработка и доставка контента, уведомления и аналитика [14]. Однако основным недостатком данной платформы является сложность настройки и неинтуитивный интерфейс, что может отталкивать неопытных разработчиков. “AWS Amplify” позволяет использовать REST и GraphQL для взаимодействия с хранимыми данными. Платформа является платной и цена формируется в зависимости от используемых ресурсов.

1.1.4 Платформа “Firebase”

Firebase это MBaaS платформа от компании Google. Firebase включает в себя все что может понадобится мобильному приложению. Сервис совмещает в себе такой функционал как хранение данных, синхронизация, аутентификация, уведомления, настраиваемые облачные функции и специальный модуль при помощи которого реализуется в приложении различная функциональность на основе машинного обучения (распознавание текста, объектов на фотографиях и много другое) [14]. Помимо бэкенд функциональности, сервис предлагает и широкий спектр возможностей для аналитики приложения и сбора данных по сбоям и ошибкам в приложении.

Для интеграции Firebase необходимо использовать SDK. Процесс интеграции значительно проще чем с AWS Amplify, так как есть обширная документация на разных языках. Firebase имеет разные тарифные планы, в том числе и бесплатный который подойдет для тестирования или небольшого приложения.

1.1.5 Хранилище данных “FaunaDB”

“FaunaDB” представляет собой облачное хранилище данных взаимодействие с которым осуществляется по GraphQL, FQL (Fauna Query Language) или с помощью одного из драйверов для популярных языков программирования. Создать базу данных можно используя графический интерфейс приложения, а интерфейс для взаимодействия с базой данных будет сгенерирован автоматически. База данных позволяет создавать, получать, редактировать, удалять, индексировать данные. FQL позволяет использовать лямбда выражения и писать собственные функции которые можно использовать в запросах. Также, FaunaDB содержит в себе модуль для регистрации и авторизации пользователей. Функционал FaunaDB не такой обширный как у MBaaS решений, но в тоже время достаточный для

многих приложений которые не требуют отправки уведомлений пользователям и сложной бизнес логики.

1.3 Способы взаимодействия с хранилищами данных

В данном разделе будут рассмотрены основные способы взаимодействия с хранилищами данных с точки зрения клиент серверных приложений.

1.3.1 Архитектурный стиль “REST”

Одним из самых распространенных архитектурных стилей для взаимодействия с серверным приложениям является REST (Representational State Transfer). Представляет собой согласованный набор ограничений, учитываемых при проектировании клиент серверных приложений. В определенных случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры [15]. В данном разделе REST будет рассмотрен с точки зрения взаимодействия с данными.

Ключевой особенностью REST являются ресурсы. Данный архитектурный стиль построен вокруг ресурсов. Взаимодействие с ними происходит через следующие методы HTTP запросов: “GET”, “POST”, “DELETE”, “PUT”, “PATCH”. Каждому ресурсу присваивается уникальные идентификатор, которые используется в URI для взаимодействия с ним. Пример операций над ресурсами:

- Удалить пользователя: DELETE /users/1
- Обновить пользователя: PATCH /users/1
- Получить одного пользователя: GET /users/1
- Получить всех пользователей: GET /users/
- Создать пользователя: POST /users/

Для API работающие по принципам REST нет единой спецификации, но как правило используют спецификацию OpenAPI, которая описывает все доступные ресурсы и методы для взаимодействия с ними. Стоит отметить что в REST невозможно получить несколько ресурсов за раз. Для получение каждого ресурса требуются отдельные запросы к серверу.

1.3.2 Язык запросов “GraphQL”

GraphQL это сильно типизированный язык запросов для работы с данными в клиент серверных приложениях созданный в качестве более эффективной и гибкой альтернативы REST. Основные особенности GraphQL:

- 1) в запросе точно указываются какие данные требуются в ответе;
- 2) позволяет облегчить агрегацию данных из нескольких источников;
- 3) необходима всего одна конечная точка.

GraphQL позволяет запрашивать и получать сразу несколько сущностей, что может сократить количество запросов к серверу и общее время затраченное на написание запросов [16]. В тоже время, можно запрашивать лишь необходимые сущности и необходимые поля этих сущностей. Данная особенность упрощает разработку программного обеспечения с несколькими клиентскими приложениями, так как отсутствует необходимость разработки отдельных методов программного интерфейса для различных клиентов. Например, если использовать GraphQL при разработке новостного приложения с двумя клиентами в виде веб и мобильного приложения, то можно в мобильном приложении одним запросом получить текст новости, картинку и заголовок новости, а в веб приложении помимо текста новости, картинки и заголовка получить отзывы и комментарии к новости. При этом, благодаря GraphQL в серверном приложении не потребуется разрабатывать два различных метода для веб и мобильного клиента.

Сущности и методы доступные через GraphQL описываются в специальной схеме. Данная схема описывается с помощью языка Schema Definition Language (SDL) [17]. SDL это простой язык состоящий из описания объектов, которые определяются ключевым словом “type” или “input” и включают в себя набор определенных типов. Также, есть корневые объекты “Query”, “Mutation” и “Subscription” которые описывают методы для взаимодействия с данными. В “Query” описываются методы для получения данных, в “Mutation” методы для операций над данными и в “Subscription” методы передающие данные через прокол WebSocket [18]. Объекты могут включать в себя другие объекты что позволяет устанавливать связь между экземплярами объектов. В листинге 1.1 приведен пример схемы GraphQL состоящий из двух сущностей автора и новости.

Листинг 1.1 - Схема GraphQL

```
type NewsPost {
  title: String
  description: String
  author: Author
}

type Author {
  # Обязательные поля обозначаются знаком !
  id: Int!
  name: String
  posts: [NewsPost!]
}

type Query {
  news(limit: Int): [NewsPost!]
  authors(limit: Int): [Author]
}

type Mutation {
  addNewsPost(title: String, description: String, AuthorID:
int): NewsPost
  addAuthor(name: String): Author
}
```

Синтаксис объектов “Mutation” и “Query” схож между собой. По порядку описывается методы, в скобках указывается принимаемые на вход значения а после через знак “:” указывается возвращаем объект, как это

описано в листинге 1.1. Восклицательный в конце типа означает что значение обязательно и не может быть пустым, в противном случае значение является необязательным. Квадратные скобки используются для обозначения массивов. В для описания формата данных в GraphQL используется ряд стандартных типов данных:

- Int - знаковое целое число размером 32 бита;
- Float - знаковое числовое значение двойной точности с плавающей точкой;
- String - последовательность символов в формате UTF-8;
- Boolean - “true” или “false”;
- ID - скалярный тип данных используемый для идентификаторов и представляют собой тип “String”;

В GraphQL есть возможность определять свои собственные типы с помощью ключевого слова “scalar”. Обработка, сериализация и десериализация остается за имплементацией схемы в серверном приложении. В листинге 1.2 приведен пример определения и использования типа для работы с датой.

Листинг 1.2 - Определение типа данных “Date” в GraphQL

```
scalar Date

type Post {
  title: String!
  createdAt: Date!
}
```

Язык запросов в GraphQL схож с языком описания схемы, и представляют собой набор запрашиваемых методов и требуемых полей в ответе. Методы в зависимости от типа в изначальной схеме помещаются в соответствующий корневой элемент “Query”, “Mutation” или “Subscription”.

Запрос на получение авторов из схемы описанной в листинге 1.1 представлен в листинге 1.3. В запросе указывается все необходимые нам поля, как и для автора, так и новостей этой автора.

Листинг 1.3 - Запрос GraphQL

```
query {  
  authors(limit: 10){  
    id  
    name  
    posts{  
      title  
      description  
      author  
    }  
  }  
}
```

Запрос на изменение данных работает по схожему принципу что и на получение данных, основное синтаксическое отличие это ключевое слово “mutation” вместо “query”.

1.4 Выводы

В данной главе был проведен обзор различных систем автоматизации разработки систем хранения данных, способов взаимодействия с ними и язык запросов GraphQL и архитектурный стиль REST. В результате обзора платформ автоматизации разработки систем хранения данных сделан вывод, что большинство решений предоставляют большой набор функционала помимо хранения данных. Но эти решения так же требуют изучения их интерфейсов или, в некоторых случаях, отдельных языков для описания системы хранения. Из представленных решений не одно не позволяет полностью сгенерировать готовое серверное приложение имея лишь описание API. Во всех случаях требуется отдельная настройка или подробное описание хранимых сущностей для того чтобы получить схему API.

В результате обзора REST и GraphQL сделан вывод что для задачи генерации серверных приложений лучше подходит GraphQL. GraphQL оказался значительно более гибким, а наличие схемы всего API описанное на языке со строгой типизацией, позволяет составить полное представление о сущностях и взаимосвязей между ними.

2 РАЗРАБОТКА ГЕНЕРАТОРА СЕРВЕРНЫХ ПРИЛОЖЕНИЙ ДЛЯ ОБРАБОТКИ И ХРАНЕНИЯ ДАННЫХ

2.1 Требования к генератору

В результате сравнения различных решений автоматизации разработки систем хранения данных было принято решение разработать генератор использующий описание API приложения в качестве входных данных. В отличие от “FaunaDB” и большинства платформ бэкенда как сервис, такой подход не требует дополнительного изучения специальных языков или особенностей пользовательского интерфейса той или иной платформы. Это позволит ускорить разработку и заранее понимать как будет выглядеть API приложения. В качестве описание API используется схема GraphQL так как она позволяет подробно описать все хранимых сущности и способы взаимодействия с ними. Этот подход не требует изучения отдельного языка для описания API, как например в случае REST и OpenAPI. Также, гибкость GraphQL позволяет эффективно использовать одни и те же методы для различных клиентов.

Желаемый порядок работы с генератором:

- 1) Создание схемы GraphQL.
- 2) Запуск генератора с помощью консольной команды.
- 3) Компиляция сгенерированного кода.
- 4) Конфигурация переменных среды необходимых для подключение к базе данных.
- 5) Запуск скомпилированного приложения.

2.2 Технологии и инструменты разработки

Для разработки был выбран язык Golang поскольку это простой и быстрый язык с большим количеством инструментов для серверной разработки и генерации кода. В качестве базы данных использовались PostgreSQL для

основного окружения и SQLite для тестового окружения. В ходе разработки были использованы следующие библиотеки:

- 1) gqlgen - инструмент генерирующий структуры данных и основную структуру GraphQL сервера на основе входной схемы;
- 2) gorm - популярное ORM для golang поддерживающие работу с большим количеством реляционных баз данных;
- 3) graphql-go - реализация GraphQL на языке golang. Включает в себя набор функционала для анализа схем и реализация методов.

2.3 Демонстрационное приложение

Перед разработкой генератора было реализовано приложение которое демонстрирует на простом примере желаемый функционал ожидаемым от результата работы генератора. Приложение представляет собой список задач назначаемых пользователю. Для этого была создана схема GraphQL API в которой описаны две сущности: пользователь и задача, и методы для взаимодействия с ними. Сущность задачи содержит идентификатор пользователя для обеспечения связи между ними. Связь между сущностями их поля представлены на рисунке 2.1.1

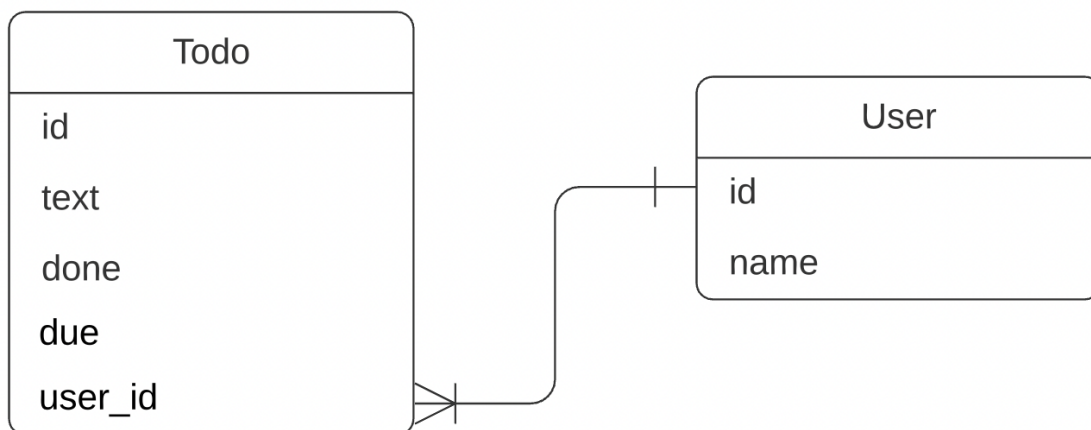


Рисунок 2.2.1 Сущности демонстрационного проекта

Данные структуры сущностей были описаны в виде объектов GraphQL и представлены в листинге 2.2.1. Восклицательные знаки в листинге означают что значения не могут быть пустыми. Более подробно с синтаксисом GraphQL можно ознакомиться в главе 1 в разделе 1.3.2.

Листинг 2.2.1 - Объекты демонстрационных сущностей

```
type Todo {  
  id: ID!  
  text: String!  
  done: Boolean!  
  due: Int!  
  user: User!  
}  
  
type User {  
  id: ID!  
  name: String!  
}
```

Для взаимодействия с данными были созданы вспомогательные объекты для создания, обновления и получения списка задач, описанные в листинге 2.2.2. Эти объекты состоят только из необходимых полей для выполнения операции.

Листинг 2.2.2 - Объекты вспомогательных сущностей

```
input CreateTodoInput {  
  text: String!  
  userId: ID!  
  due: Int!  
}  
  
input UpdateTodoInput {  
  todoId: ID!  
  text: String!  
  userId: ID!  
  done: Boolean!  
  due: Int!  
}  
  
input GetTodosInput {  
  userId: ID  
  text: String  
  dueBefore: Int  
  dueAfter: Int
```

```

    done: Boolean
  }

input UpdateUserInput {
  userId: ID!
  name: String!
}

```

В качестве методов для взаимодействия с данными для каждой из сущностей были описаны методы создания, удаления, обновления, получения, описанные в листинге 2.2.3.

Листинг 2.2.3 - Методы для взаимодействия с сущностями

```

type Mutation {
  createUser(input: String!): User!
  updateUser(input: UpdateUserInput!): User!
  deleteUser(input: Int!): User!
  createTodo(input: CreateTodoInput!): Todo!
  updateTodo(input: UpdateTodoInput!): Todo!
  deleteTodo(input: Int!): Boolean!
}

type Query {
  getTodo(input: Int!): Todo!
  getTodos(input: UpdateTodoInput!): [Todo!]!
  getUser(input: Int!): User!
  getUsers: [User!]!
}

```

Для описанной схемы был успешно написан сервер использующий архитектуру представленную на рисунке 2.2.2. Архитектура состоит из трех слоев: “Resolver”, “Service”, “Repository”.

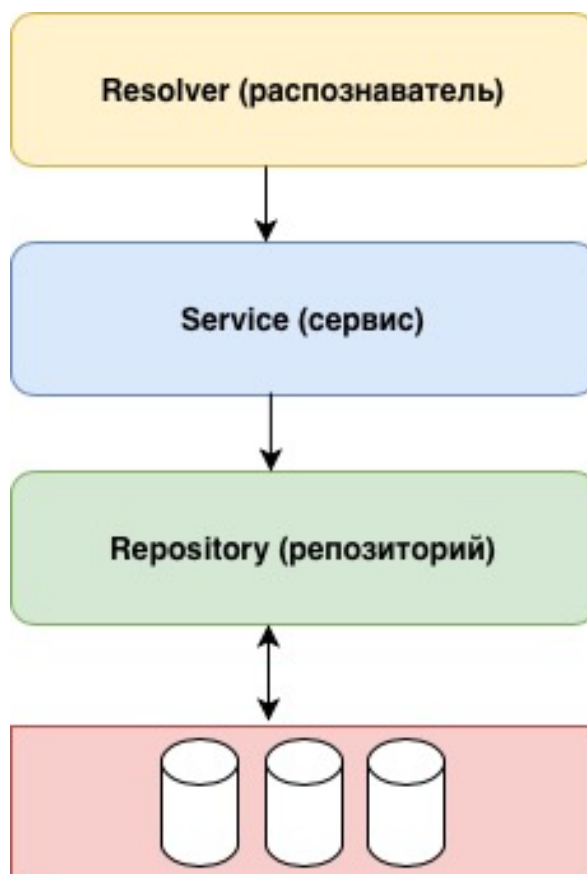


Рисунок 2.2.2 Архитектура демонстрационного проекта

Слой “Repository” отвечает за логику работы с базой данных и другими источниками данными. В этом слое используется паттерн “репозиторий”, где для каждой хранимой сущности описывается и реализуется интерфейс для взаимодействия с данными [19]. В слое “Service” реализуется вся бизнес-логика приложения, и к этот слой взаимодействует с интерфейсам слоя “Repository“. Слой “Resolver” отвечает за обработку запросов и соединения с слоем “Service”.

Примеры успешных запросов на создание пользователя и задания представлены ниже на рисунках 2.2.3 и 2.2.4. Листинг слоя “Repository” и слоя “Service” для создания пользователя представлен в приложении А.

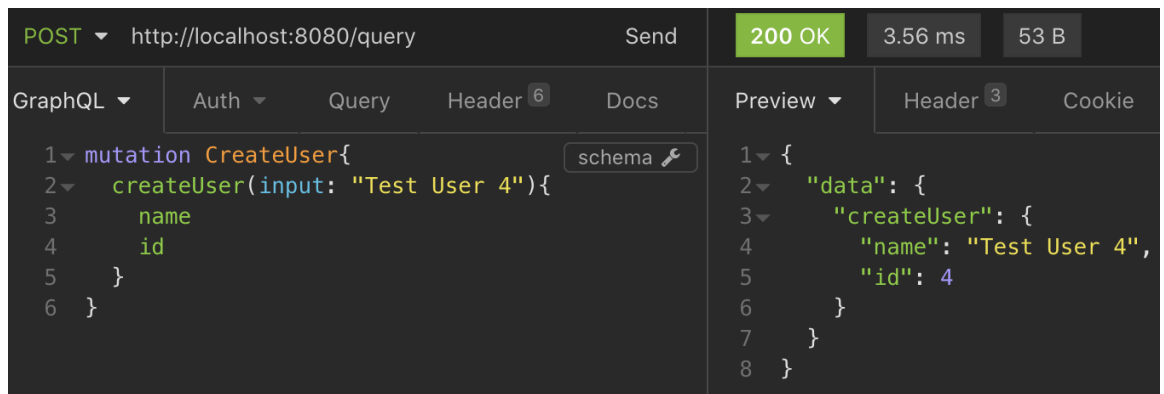


Рисунок 2.2.3. Пример запроса и ответа для создания пользователя

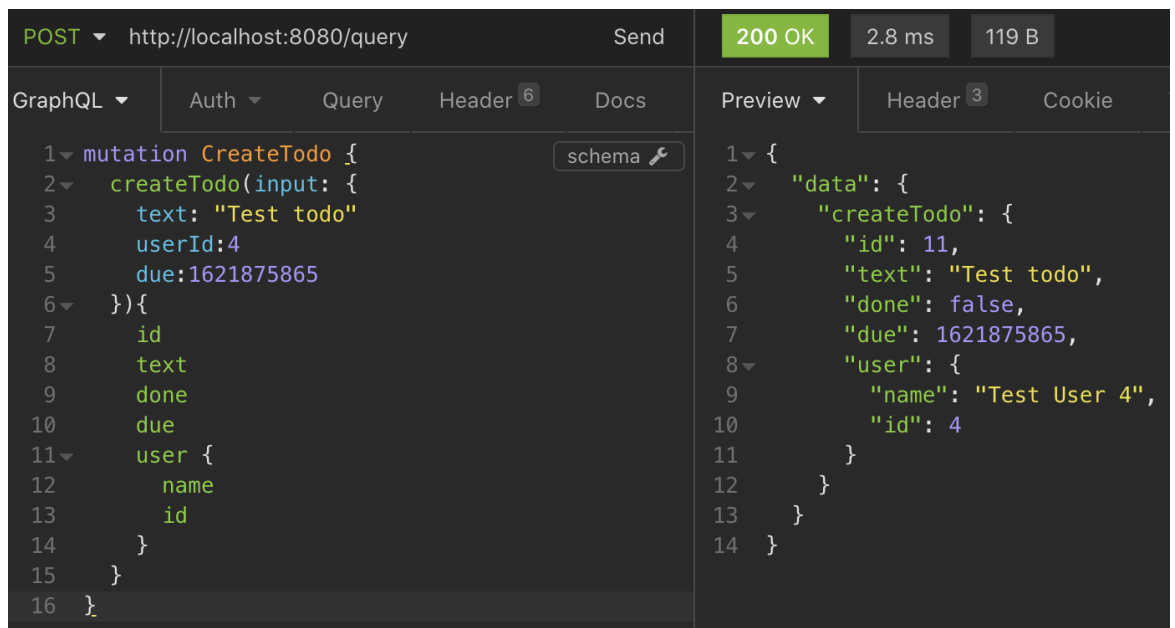


Рисунок 2.2.4. Пример запроса и ответа для создания задания

2.4 Разработка генератора

Задача генератора сгенерировать код который обеспечить хранение и предоставить методы для взаимодействия с сущностями описанными в входной спецификации. Архитектура сгенерированного генератором приложения должна быть схожа с архитектурой демонстрационного приложения. Входная GraphQL схема используемая генератором меньше чем в демонстрационном проекте, т.к. вспомогательные объекты и основные методы для взаимодействия с данными можно сгенерировать при помощи

генератора. Для этого работу генератора потребовалось разделить на несколько этапов:

- 1) Синтаксический анализ и валидаций входной спецификации в виде схемы GraphQL.
- 2) Преобразование сущностей в структуры и добавление вспомогательных полей.
- 3) Генерация вспомогательного кода для взаимодействия с базой данных.
- 4) Генерация новой GraphQL схемы с методами для взаимодействия с данными.

Для синтаксического анализа была использована библиотека из проекта `go-graphql` [20]. Данная библиотека позволяет валидировать GraphQL схему и построить абстрактное синтаксическое дерево для анализируемой схемы. При успешном анализе возвращается объект абстрактного синтаксического дерева, в ином случае ошибка. Пример работы с данной библиотекой приведен в листинге 2.4.1

Листинг 2.4.1 - Получение абстрактного синтаксического дерева

```
func Parse(input string) (Model, error) {
    var model Model

    astDocument, err := parser.Parse(parser.ParseParams{
        Options: parser.ParseOptions{
            NoLocation: true,
        },
        Source: input,
    })
    if err != nil {
        return model, err
    }

    model = Model{astDocument}
    return model, nil
}
```

После построение абстрактного синтаксического дерева выделяются объекты обозначенные директивой “entity”. Поля этих объектов которые

необходимо хранить и являются колонкой в реляционной базе данных обозначаются директивой “column”. Эти объекты являются хранимыми сущностями. Сущностям добавляются дополнительные вспомогательные поля как идентификаторы, время создание и обозначение об удаление. Пример добавление вспомогательных полей с комментариями показан в листинге 2.4.2

Листинг 2.4.2 - Добавление вспомогательных полей

```
func EnrichEntites(model *Model) error {
    // Создаем вспомогательные поля createdAt, updatedAt, id
    createdAt := defineColumn("createdAt", "Time", true)
    updatedAt := defineColumn("updatedAt", "Time", false)
    id := defineColumn("id", "ID", true)

    // Получаем объекты с директивой entity
    entities := model.Entities()

    // Итерация по объектам
    for _, object := range entities {
        object.Definition.Fields = append(
            []*ast.FieldDefinition{id},
            object.Definition.Fields...,
        )

        // Создаем вспомогательные поля для связи между
        // объектами
        for _, relation := range object.Relationships() {
            if relation.IsToOne() {
                object.Definition.Fields = append(
                    object.Definition.Fields,
                    defineColumn(relation.Name()+"Id",
                        "ID", false),
                )
            }
        }
        object.Definition.Fields = append(
            object.Definition.Fields,
            updatedAt,
            createdAt)
    }
    return nil
}
```


После того как модели готовы к генерации, они генерируются вместе с основным кодом для взаимодействия с базой данных. Для этого используется стандартная библиотека для работы с шаблонами "text/template" и используются заранее написанные шаблоны [21]. Пример шаблона для генерации структур сущностей приведен в листинге 2.4.3.

Листинг 2.4.3 - Шаблон для генерации структур

```
type {{.Name}} struct {
    {{range $col := $object.Columns}}
        {{ $col.MethodName }} {{ $col.GoType }} ` + "`" +
    `{{ $col.ModelTags }}` + "`" + `{{end}}

    {{range $rel := $object.Relationships}}
        {{ $rel.MethodName }} {{ $rel.GoType }} ` + "`" +
    `{{ $rel.ModelTags }}` + "`" + `
        {{if $rel.Preload}}{{ $rel.MethodName }}Preloaded bool ` +
    "`gorm:\"" - "\"`" + `{{end}}
    {{end}}
}
```

Шаблоны нужны чтобы сгенерировать общий код использующиеся различными частями приложения. Например запросы к базе данных или методы в рамках слоя "Resolver" по аналогии с демонстрационным приложением. В шаблоны передаются структуры полученные шагом ранее. Вся основная логика приложения реализована с помощью шаблонов. После успешной генерации файлов с кодом, вызывается утилита "goimports" входящая в стандартный пакет языка golang [13]. Данная утилита импортирует в файлы все недостающие зависимости и убирает неиспользуемые зависимости. Примеры листинги других частей генератора продемонстрированы в приложении Б.

После генерации основной части кода необходимо сгенерировать новую GraphQL схему, которая включает в себя дополнительные объекты и новые методы для получения, удаления, редактирования сущностей поля приложения. Для этого используется библиотека go-graphql, которая

использовалась ранее для анализа входной схемы. Принимая на вход абстрактное синтаксическое дерево генерируется новая схема.

Для работы генератора требуется установить `golang` и инициализировать систему модулей в проекте с помощью команды `“go mod init”`. Входная схемы описывается в корне проекта в файле `“model.graphql”`. Генератор запускается с помощью команды `“init”`. Сгенерированный код можно редактировать и компилировать при помощи команды `“go build”`. В результате работы генератора для входной схемы схожей с демонстрационным проектом, представленной в листинге 2.4.4, было успешно сгенерировано приложение предоставляющие такой же функционал как и демонстрационный приложение. Финальная схема полученная в результате работы генератора представлена в приложении В.

Листинг 2.4.4 - Схема GraphQL для генератора

```
type User @entity {
  name: String @column

  todos: [Todo!]! @relationship(inverse:"user")
}

type Todo @entity {
  text: String @column
  done: Boolean @column
  due: Int @column

  user: User @relationship(inverse:"todos")
}
```

Для сравнения с демонстрационным проектом на рисунке 2.4.1 продемонстрирован запрос и ответ на создание “задания”. Запрос схож запросом из демонстрационного проекта и возвращает ожидаемый результат.

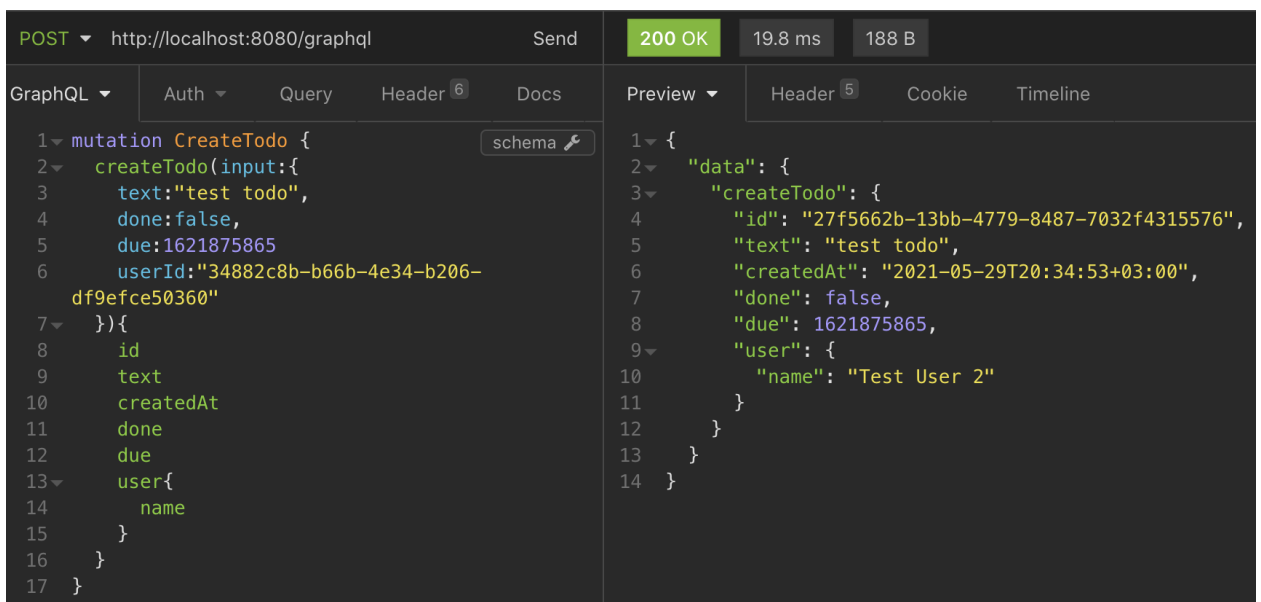


Рисунок 2.4.1 Запрос и ответ на создание “задания”

2.5 Выводы

В данном разделе были подобраны технологии и инструмент для разработки генератора систем хранения данных, описана архитектура генерируемого приложения, разработано демонстрационный приложение демонстрирующие желаемый результат работы генератор и был разработан и описан сам генератор. Разработанным генератором было сгенерирована приложение использующие те же сущности в качестве входных данных что и демонстрационное приложение. В результате было полученные приложение реализующие все те же методы для взаимодействия данных что и демонстрационное приложение.

3. РЕЗУЛЬТАТЫ И ПЕРСПЕКТИВЫ РАЗВИТИЯ

3.1 Сравнение демонстрационного приложения и сгенерированного

В демонстрационном и сгенерированном приложениях использованы одинаковые сущности. Но во входных данных для генератора нет необходимости описывать вспомогательные объекты, так как они генерируются самим генератором. При сравнение сгенерированных вспомогательных объектов для создания и обновлений задач, представленных в листинге 3.1.1, и вспомогательных объектов созданных в ходе разработки демонстрационного приложения, представленных в листинге 3.1.2, можно увидеть что объекты очень похожи, но небольшие различия все же присутствуют. Например, в сгенерированных объектах есть возможность выставить свой уникальный идентификатор при создание объекта. Это сделано для большей гибкости.

Листинг 3.1.1 - Сгенерированные вспомогательные объекты

```
input TodoCreateInput {
  id: ID
  text: String
  done: Boolean
  due: Int!
  userId: ID
}

input TodoUpdateInput {
  text: String
  done: Boolean
  due: Int
  userId: ID
}
```

Листинг 3.1.2 - Демонстрационные вспомогательные объекты

```
input CreateTodoInput {
  text: String
  userId: ID
  due: Int!
}

input UpdateTodoInput {
```

```

    todoId: ID!
    text: String
    userId: ID
    done: Boolean
    due: Int
}

```

Запросы на создание и обновление объекта обрабатываются схожим образом. В обоих случаях после десериализации входных данных вызывается метод слоя “resolver”, который в свою очередь при необходимости преобразует данные в нужные тип и вызывает соответствующие методы сервиса. Пример приведен в листинге 3.1.3.

Листинг 3.1.3 - Обработчик запроса на создание пользователя

```

func (r *mutationResolver) CreateUser(ctx context.Context, input
string) (*model.User, error) {
    user := &models.User{
        Name: input,
    }
    err := r.userService.CreateUser(ctx, user)
    if err != nil {
        return nil, err
    }

    return &model.User{
        Name: user.Name,
        ID:   user.ID,
    }, nil
}

```

Генератор генерирует код соответствующий функционалу и архитектуре демонстрационного проекта. Код полученный в результате работы генератора можно редактировать и дополнять собственными методами, но стоит учитывать что при повторной генерации изменения будут утеряны. Так, при необходимости реализации более сложных приложений, можно использовать генератор для создание основы приложения.

3.2 Перспективы развития

Отталкиваясь от проведенного обзора решений в первой главе, можно сделать вывод что существует множество возможностей развития данного генератора. Например, можно добавить функционал для авторизации и управления пользователями или реализовать файловое хранилище. Полезными могут быть сервисы стриминга потокового аудио и отправки уведомлений. Для этого потребуется разработать отдельные сервисы взаимодействующие с генерируемым приложением, но в таком случае возможно придется использовать дополнительные спецификации помимо схемы GraphQL. Наличие графического интерфейса могло бы сделать данное решение более привлекательным для неопытных пользователей.

Помимо этого, стоит предпринять ряд действия для улучшения производительности и читаемости генерируемого кода. Например можно кэшировать запросы к базе данных или сгенерировать более эффективные сериализаторы и десериализаторы.

3.3 Выводы

В данном разделе было произведено сравнение разработанного и сгенерированного приложения, которое показало схожесть основного функционала и архитектур обоих приложений. Также, был произведен обзор возможных улучшений и перспектив развития полученного генератора.

ЗАКЛЮЧЕНИЕ

В результате исследования был произведен обзор существующих решений автоматизации разработки систем хранения данных, а также способов взаимодействия с облачными хранилищами данных. После этого был разработан демонстрационный проект представляющий собой систему хранения с двумя сущностями, взаимодействия с которым осуществляется посредством GraphQL API. Далее был разработан генератор позволяющий генерировать хранилища данных по входной спецификации в виде GraphQL схемы. При помощи генератора было создано приложение наподобие демонстрационного приложения. Было произведено сравнение демонстрационного и сгенерированного хранилища данных.

Разработанный генератор позволяет эффективно создавать простейшие хранилища данных, которые могут быть доработаны и модифицированы пользователями и скомпилированы под большинство современных устройств.

В дальнейшем предполагается разработка нового функционала для генератора и повышение производительности генерируемого кода.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Операционная система Android [Электронный ресурс] – URL: - <https://ru.wikipedia.org/wiki/Android> (дата обращения: 21.05.2021).
2. Язык программирования Golang – URL: <https://ru.wikipedia.org/wiki/Go> (дата обращения: 10.05.2021).
3. Язык программирования JavaScript [Электронный ресурс] – URL: <https://developer.mozilla.org/ru/docs/Web/JavaScript> (дата обращения: 12.05.2021).
4. “Object-Relation Mapping” [Электронный ресурс] – URL: [https://ru.bmstu.wiki/ORM_\(Object-Relational_Mapping\)](https://ru.bmstu.wiki/ORM_(Object-Relational_Mapping)) (дата обращения: 21.05.2021).
5. SDK - Википедия [Электронный ресурс] 2021. – URL: <https://ru.wikipedia.org/wiki/SDK> (дата обращения: 15.05.2021).
6. “Абстрактное синтаксическое дерево” – URL: https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%BE%D0%B5_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE (дата обращения: 15.05.2021).
7. Компилятор - Википедия [Электронный ресурс] 2021. – URL: <https://ru.wikipedia.org/wiki/Компилятор> (дата обращения: 15.05.2021).
8. Программный интерфейс приложения [Электронный ресурс] – URL: <https://ru.wikipedia.org/wiki/Go> (дата обращения: 10.05.2021).
<https://ru.wikipedia.org/wiki/API>
9. Анализ рынка мобильных приложений – URL: <https://sensortower.com/blog/sensor-tower-app-market-forecast-2025> (дата обращения: 18.05.2021).

10. Сравнение подходов автоматизации разработки бекенда [Электронный ресурс] – URL: <https://core.ac.uk/download/pdf/80989262.pdf> (дата обращения: 28.04.2021)
11. Обзор BaaS платформы «Backendless» [Электронный ресурс] – URL: <https://habr.com/ru/company/backendless/blog/180367/> (дата обращения: 10.05.2021)
12. Платформа “Parse” [Электронный ресурс] – URL: [https://en.wikipedia.org/wiki/Parse_\(platform\)](https://en.wikipedia.org/wiki/Parse_(platform)) (дата обращения: 10.05.2021)
13. Миграция платформы “Parse” [Электронный ресурс] – URL: <https://blog.back4app.com/parse-%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D1%8B-%D0%BC%D0%B8%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D0%B8-%D0%BA%D1%80%D0%BE%D0%BC%D0%B5-%D1%85%D0%BE%D1%81%D1%82%D0%B8%D0%BD%D0%B3%D0%B0-parse-server%2F> (дата обращения: 10.05.2021)
14. Обзор облачных сервисов для разработки бэкенда [Электронный ресурс] – URL: <https://habr.com/ru/company/surfstudio/blog/463435/> (дата обращения: 10.05.2021)
15. REST API [Электронный ресурс] – URL: <https://ru.wikipedia.org/wiki/REST> (дата обращения: 25.05.2021)
16. Gleison Brito, Marco Tulio Valente “Rest vs GraphQL: A Controlled Experiment” - URL: <https://arxiv.org/pdf/2003.04761.pdf> (дата обращения: 29.05.2021)
17. Документация GraphQL [Электронный ресурс] – URL: <https://graphql.org/learn/> (дата обращения: 25.05.2021)
18. “Schema definition language” [Электронный ресурс] – URL: <https://www.apollographql.com/docs/apollo-server/schema/schema/> (дата обращения: 25.05.2021)

19. Паттерн “Репозиторий” [Электронный ресурс] – URL: <http://design-pattern.ru/patterns/repository.html> (дата обращения: 10.05.2021)
20. Реализация GraphQL на языке Golang [Электронный ресурс] – URL: <https://pkg.go.dev/github.com/graphql-go/graphql> (дата обращения: 10.05.2021)
21. Документация языка программирования Golang [Электронный ресурс] – URL: <https://golang.org/doc/> (дата обращения: 10.05.2021)

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ДЛЯ СОЗДАНИЯ ПОЛЬЗОВАТЕЛЯ В ДЕМОНСТРАЦИОННОМ ПРИЛОЖЕНИИ

```
package repos

import (
    "context"

    "github.com/AYaro/ThesisTestProject/internal/models"
    "gorm.io/gorm"
)

type UserRepo struct {
    db *gorm.DB
}

func NewUserRepo(db *gorm.DB) *UserRepo {
    return &UserRepo{db: db}
}

func (repo *UserRepo) FindUsers(ctx context.Context)
([]*models.User, error) {
    var users []*models.User
    err := repo.db.WithContext(ctx).Find(&users).Error
    return users, err
}

func (repo *UserRepo) FindUserByID(ctx context.Context, userID
int) (*models.User, error) {
    var user models.User
    err := repo.db.WithContext(ctx).First(&user, userID).Error
    return &user, err
}

func (repo *UserRepo) CreateUser(ctx context.Context, user
*models.User) error {
    return repo.db.WithContext(ctx).Create(user).Error
}

func (repo *UserRepo) UpdateUser(ctx context.Context, user
*models.User) error {
    err := repo.db.WithContext(ctx).Save(user).Error
    return err
}

func (repo *UserRepo) DeleteUser(ctx context.Context, userID
int) error {
    err := repo.db.WithContext(ctx).Delete(&models.User{},
userID).Error
}
```

```

        return err
    }
    func (r *mutationResolver) CreateUser(ctx context.Context, input
string) (*model.User, error) {
        user := &models.User{
            Name: input,
        }
        err := r.userService.CreateUser(ctx, user)
        if err != nil {
            return nil, err
        }

        return &model.User{
            Name: user.Name,
            ID:   user.ID,
        }, nil
    }
}

```

ПРИЛОЖЕНИЕ Б. ФРАГМЕНТ ЛИСТИНГА РАЗРАБОТАННОГО ГЕНЕРАТОРА

```
// Основная функция работы генератора
func generate(sourceFile, genPath string) error {
    source, err := ioutil.ReadFile(sourceFile)
    if err != nil {
        return err
    }

    m, err := model.Parse(string(source))
    if err != nil {
        return err
    }

    if _, err := os.Stat(genPath); os.IsNotExist(err) {
        err = os.Mkdir(genPath, 0777)
        if err != nil {
            panic(err)
        }
    }

    err = model.EnrichEntites(&m)
    if err != nil {
        return err
    }

    err = generateFiles(genPath, &m)
    if err != nil {
        return err
    }

    err = model.EnrichModel(&m)
    if err != nil {
        return err
    }

    schema, err := model.PrintSchema(m)
    if err != nil {
        return err
    }

    schema = "# NOTICE: THIS FILE WAS GENERATED\n\n" + schema

    if err := ioutil.WriteFile(path.Join(genPath,
"/schema.graphql"), []byte(schema), 0644); err != nil {
        return err
    }
}
```

```

        if err := tools.RunCommand("go run
github.com/99designs/gqlgen", genPath); err != nil {
            return err
        }

        return nil
    }

    // генерация из шаблона
    func WriteFromTemplate(t, filename string, data interface{})
    error {
        temp, err := template.New(filename).Parse(t)
        if err != nil {
            return err
        }
        var content bytes.Buffer
        writer := io.Writer(&content)

        err = temp.Execute(writer, &data)
        if err != nil {
            return err
        }
        err = ioutil.WriteFile(filename, content.Bytes(), 0777)
        if err != nil {
            return err
        }

        return tools.RunCommand(fmt.Sprintf("goimports -w %s",
filename))

        return nil
    }

    // Шаблон соединения с бд
    type DB struct {
        db *gorm.DB
    }

    func NewDB(urlStr string) *DB {
        urlStr := os.Getenv("DATABASE_URL")
        if urlStr == "" {
            panic("NO DATABASE STRING")
        }

        u, err := url.Parse(urlStr)
        if err != nil {
            panic(err)
        }

        db, err := gorm.Open(u.Scheme, urlStr)
        if err != nil {
            panic(err)
        }
    }

```

```

    }

    if urlStr == "sqlite3://:memory:" {
        db.DB().SetMaxIdleConns(1)
        db.DB().SetConnMaxLifetime(time.Second * 300)
        db.DB().SetMaxOpenConns(1)
    } else {
        db.DB().SetMaxIdleConns(10)
        db.DB().SetMaxOpenConns(10)
    }

    return &DB{db: db}
}

// Шаблон для получения объектов
type GenQueryResolver struct{ *GenQueryResolver }

{{range $obj := .Model.Entities}}
    // Query{{$obj.Name}}HandlerOptions ...
    type Query{{$obj.Name}}HandlerOptions struct {
        ID *string
        Filter *{{$obj.Name}}FilterType
    }

    // {{$obj.Name}} ...
    func (r *GenQueryResolver) {{$obj.Name}}(ctx
context.Context, id *string, filter *{{$obj.Name}}FilterType)
(*{{$obj.Name}}, error) {
        opts := Query{{$obj.Name}}HandlerOptions{
            ID: id,
            Filter: filter,
        }
        return r.Handlers.Query{{$obj.Name}}(ctx,
r.GenQueryResolver, opts)
    }

    // Query{{$obj.Name}}Handler ...
    func Query{{$obj.Name}}Handler(ctx context.Context, r
*GenResolver, opts Query{{$obj.Name}}HandlerOptions)
(*{{$obj.Name}}, error) {
        selection := []ast.Selection{}
        for _, f := range graphql.CollectFieldsCtx(ctx, nil) {
            selection = append(selection, f.Field)
        }
        selSet := ast.SelectionSet(selection)

        offset := 0
        limit := 1
        res := &{{$obj.Name}}ResultType{
            EntityResultType: EntityResultType{
                Offset: &offset,
                Limit: &limit,
                Filter: opts.Filter,

```

```

        SelectionSet: &selSet,
    },
}

qdb := r.GetDB(ctx)
if opts.ID != nil {
    qdb = qdb.Where(TableName("{{$obj.TableName}}") +
".id = ?", *opts.ID)
}

var results []*{{$obj.Name}}
plOpts := GetItemsOptions{
    Alias:TableName("{{$obj.TableName}}"),
    Preloaders:[]string{ {{range $r :=
$obj.PreloadableRelationships}}
    {{$r.MethodName}}",{{end}}
    },
}

err := res.GetResults(ctx, qdb, plOpts, &results)
if err != nil {
    return nil, err
}

if len(results) == 0 {
    return nil, nil
}

return results[0], err
}
{{end}}

```


ПРИЛОЖЕНИЕ В. СХЕМА GraphQL ПОЛУЧЕННАЯ В РЕЗУЛЬТАТЕ РАБОТЫ ГЕНЕРАТОРА

```
scalar Time

schema {
  query: Query
  mutation: Mutation
}

type Query {
  user(id: ID, filter: UserFilterType): User
  users(offset: Int, limit: Int, sort: [UserSortType!], filter:
UserFilterType): UserResultType!
  todo(id: ID, filter: TodoFilterType): Todo
  todos(offset: Int, limit: Int, sort: [TodoSortType!], filter:
TodoFilterType): TodoResultType!
}

type Mutation {
  createUser(input: UserCreateInput!): User!
  updateUser(id: ID!, input: UserUpdateInput!): User!
  deleteUser(id: ID!): User!
  deleteAllUsers: Boolean!
  createTodo(input: TodoCreateInput!): Todo!
  updateTodo(id: ID!, input: TodoUpdateInput!): Todo!
  deleteTodo(id: ID!): Todo!
  deleteAllTodos: Boolean!
}

enum ObjectSortType {
  ASC
  DESC
}

type User {
  id: ID!
  name: String
  todos: [Todo!]!
  updatedAt: Time
  createdAt: Time!
  todosIds: [ID!]!
}

type Todo {
  id: ID!
  text: String
  done: Boolean
  due: Int!
  user: User
  userId: ID
```

```

    updatedAt: Time
    createdAt: Time!
}

input UserCreateInput {
  id: ID
  name: String
  todosIds: [ID!]
}

input UserUpdateInput {
  name: String
  todosIds: [ID!]
}

input UserFilterType {
  id: ObjectSortType
  name: ObjectSortType
  updatedAt: ObjectSortType
  createdAt: ObjectSortType
  todosIds: ObjectSortType
  todos: TodoSortType
}

input UserFilterType {
  AND: [UserFilterType!]
  OR: [UserFilterType!]
  id: ID
  name: String
  updatedAt: Time
  createdAt: Time
  todos: TodoFilterType
}

type UserResultType {
  items: [User!]!
  count: Int!
}

input TodoCreateInput {
  id: ID
  text: String
  done: Boolean
  due: Int!
  userId: ID
}

input TodoUpdateInput {
  text: String
  done: Boolean
  due: Int
  userId: ID
}

```

```

input TodoSortType {
  id: ObjectSortType
  text: ObjectSortType
  done: ObjectSortType
  due: ObjectSortType
  userId: ObjectSortType
  updatedAt: ObjectSortType
  createdAt: ObjectSortType
  user: UserSortType
}

input TodoFilterType {
  AND: [TodoFilterType!]
  OR: [TodoFilterType!]
  id: ID
  text: String
  done: Boolean
  due: Int
  userId: ID
  updatedAt: Time
  createdAt: Time
  user: UserFilterType
}

type TodoResultType {
  items: [Todo!]!
  count: Int!
}

```