

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И  
МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ  
Ордена Трудового Красного Знамени федеральное государственное  
Бюджетное образовательное учреждение высшего образования  
«МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ СВЯЗИ И  
ИНФОРМАТИКИ»  
(МТУСИ)

Дисциплина: «Технологии программирования Web-приложений»

Реферат

на тему: «Сравнение подходов к разработке API: GraphQL vs REST API»

Выполнил:

Студенты 1 курса

Группы М092401(75)

Цыганков Р.О.

Проверил:

к.т.н., Гузеев А. В.

Москва 2025

## Содержание

Введение.....	2
1. Теоретические основы API.....	3
2. REST API: архитектурный стиль.....	5
3. GraphQL: язык запросов для API.....	9
4. Практическая реализация REST API.....	14
5. Практическая реализация GraphQL API.....	18
6. Сравнение практических реализаций REST и GraphQL .....	23
7. Рекомендации по выбору подхода .....	26
8. Заключение .....	29
9. Список использованных источников .....	30
10. Скриншоты проекта .....	31

## **Введение**

### **Актуальность темы**

API (Application Programming Interface) играет ключевую роль в современной разработке программного обеспечения, обеспечивая взаимодействие между различными приложениями и системами. С ростом сложности и масштабов приложений требования к API постоянно возрастают - необходимы гибкость, производительность и удобство использования. Традиционный подход REST API, основанный на архитектурных принципах, долгое время был стандартом, однако он имеет ограничения, связанные с избыточной или недостаточной загрузкой данных, множественными запросами для получения связанных ресурсов и необходимостью версионирования при изменениях.

GraphQL, разработанный Facebook, представляет собой язык запросов и спецификацию API, ориентированную на гибкость и производительность. Он позволяет клиенту точно указывать, какие данные нужны, и получать их одним запросом, что снижает сетевой трафик и упрощает работу с данными. Это особенно важно для современных мобильных и веб-приложений с динамическими и сложными интерфейсами.

### **Цель и задачи исследования**

Целью реферата является проведение комплексного сравнительного анализа подходов REST и GraphQL в разработке API, выявление их сильных и слабых сторон, а также определение оптимальных сценариев применения.

Для достижения цели поставлены следующие задачи:

- Рассмотреть теоретические основы REST и GraphQL, их архитектурные принципы и особенности реализации.

- Проанализировать ключевые различия в моделях запросов, форматах данных, производительности и удобстве разработки.
- Продемонстрировать практическую реализацию базовых API на примерах кода для каждого подхода.
- Провести сравнительный анализ практических реализаций с точки зрения эффективности, гибкости и удобства использования.
- Сформулировать рекомендации по выбору подхода в зависимости от конкретных условий и требований проекта.

## **1. Теоретические основы API**

### **Понятие API и его роль**

API (Application Programming Interface, интерфейс программирования приложений) - это набор правил, протоколов и инструментов, который позволяет программам взаимодействовать друг с другом. По сути, API выступает в роли "моста" между разными приложениями, обеспечивая обмен данными и выполнение определённых задач. Например, с помощью API Google Maps можно интегрировать карты и геолокацию в сторонние приложения или сайты [1].

API используются практически во всех современных программных продуктах, обеспечивая их интеграцию, расширяемость и взаимодействие с внешними сервисами. Благодаря API разработчики могут использовать готовые инструменты и функции, не реализуя их с нуля, что ускоряет и удешевляет процесс разработки [2].

### **Основные функции и задачи API**

API выполняет несколько ключевых функций:

- Предоставляет доступ к данным (например, получение информации о погоде или курсах валют).

- Позволяет выполнять операции с ресурсами (создание, изменение, удаление данных).
- Обеспечивает отправку уведомлений (например, оповещения в мобильных приложениях).
- Управляет ресурсами (например, настройка прав доступа или оптимизация серверных мощностей).
- Повышает безопасность, изолируя критичные функции от прямого доступа внешних программ<sup>5</sup>.

## **Протоколы и архитектурные стили API**

API реализуются с помощью различных протоколов и архитектурных стилей, которые определяют правила обмена данными и взаимодействия между программами. К основным преимуществам таких протоколов относятся:

- Стандартизация форматов и структур данных.
- Модульность и расширяемость систем.
- Улучшенное взаимодействие между разными программами и сервисами.
- Встроенные механизмы безопасности (аутентификация, авторизация).

Однако у протоколов API есть и недостатки: сложность реализации, потенциальные проблемы совместимости при обновлениях, а также возможные накладные расходы на передачу данных [3].

## **Виды и примеры API**

Существует несколько основных видов API:

- Веб-API (Web API) - предоставляют доступ к данным и функциям через интернет с помощью HTTP-запросов (например, REST, GraphQL).
- Библиотечные API - наборы функций, предоставляемых программными библиотеками для использования в коде приложений.

- API операционных систем - позволяют приложениям взаимодействовать с функциями ОС (например, работа с файлами, сетью, устройствами).

### **Примеры использования API:**

- Интеграция платёжных систем на сайтах.
- Авторизация через социальные сети.
- Использование сервисов машинного обучения через облачные API.

### **Эволюция подходов к API: от REST к GraphQL**

Исторически одним из самых распространённых архитектурных стилей стал REST (Representational State Transfer), предложенный Роем Филдингом в 2000 году. REST определяет принципы построения распределённых систем и до сих пор широко используется для создания веб-API.

Позднее, с ростом требований к гибкости и эффективности передачи данных, появился язык запросов GraphQL, разработанный Facebook в 2012 году и открытый для сообщества в 2015 году. GraphQL предоставляет клиенту возможность точно указывать, какие данные ему нужны, что позволяет избежать избыточной или недостаточной загрузки информации.

### **Значение API в современной разработке**

API стали неотъемлемой частью современной экосистемы программного обеспечения. Они позволяют быстро интегрировать новые сервисы, расширять функциональность приложений, обеспечивать безопасность и стандартизировать взаимодействие между разными системами. Благодаря API разработка становится более быстрой, гибкой и экономичной.

## **2. REST API: архитектурный стиль**

## Основные принципы REST

REST (Representational State Transfer) - это архитектурный стиль для построения распределённых веб-сервисов, который определяет набор принципов и ограничений для взаимодействия между клиентами и серверами. Ниже подробно раскрываются ключевые аспекты REST API [4].

REST API строится на следующих фундаментальных принципах и ограничениях:

- Клиент-серверная архитектура- взаимодействие строится по модели «клиент–сервер»: клиент отвечает за пользовательский интерфейс и отправляет запросы, сервер - за обработку запросов и управление данными. Это разделение упрощает сопровождение и масштабирование системы.
- Отсутствие состояния (stateless)- каждый запрос от клиента к серверу должен содержать всю необходимую информацию для его обработки. Сервер не хранит состояние клиента между запросами, что облегчает масштабирование и повышает надёжность.
- Кэшируемость- сервер может помечать ответы как кэшируемые или нет, что позволяет клиентам и промежуточным прокси сохранять ответы и снижать нагрузку на сервер, ускоряя доступ к данным.
- Единый интерфейс (Uniform Interface)- все взаимодействия с ресурсами должны происходить через стандартизированный и единообразный интерфейс, что упрощает понимание и использование API.
- Многоуровневая система (Layered System)- Архитектура может быть разделена на слои (например, презентационный, бизнес-логики, хранения данных), каждый из которых отвечает за свою задачу. Это увеличивает гибкость, масштабируемость и безопасность.
- Возможность передачи кода по требованию (Code on Demand)- Сервер может отправлять исполняемый код (например, скрипты JavaScript), который клиент выполняет при необходимости. Это расширяет функциональность, но используется редко.

## Структура REST API

- Ресурсно-ориентированная модель- в REST всё строится вокруг ресурсов - сущностей, которыми управляет API (например, пользователи, товары, заказы). Каждый ресурс идентифицируется уникальным URL [5].

Пример структуры URL:

`http://api.example.com/v1/users/123`

Здесь users - коллекция ресурсов, 123 - уникальный идентификатор конкретного пользователя.

- Использование HTTP-методов- для взаимодействия с ресурсами применяются стандартные HTTP-методы, которые соответствуют операциям CRUD:
  - GET - получение данных о ресурсе (без изменения состояния)
  - POST - создание нового ресурса
  - PUT - полное обновление ресурса или его создание, если не существует
  - DELETE - удаление ресурса

Корректное использование HTTP-методов делает API интуитивно понятным и предсказуемым для разработчиков.

- Форматы данных- Обычно REST API использует JSON или XML для передачи данных между клиентом и сервером.
- Версионирование- Для поддержки обратной совместимости часто применяют версионирование API, добавляя версию в URL (например, /v1/).
- Статусы ответов и обработка ошибок- Сервер возвращает стандартные HTTP-коды состояния (например, 200 - успех, 404 - не найдено, 500 - ошибка сервера), что облегчает обработку ошибок на стороне клиента.



## Преимущества и недостатки REST

### Преимущества:

- Простота и прозрачность архитектуры.
- Использование стандартных протоколов и методов HTTP.
- Хорошая масштабируемость и кэшируемость.
- Лёгкая интеграция с различными клиентами (веб, мобильные приложения и др.).

### Недостатки:

- Возможен избыточный или недостаточный объём данных (over-fetching/under-fetching): клиент может получать больше или меньше данных, чем требуется.
- Для получения связанных данных часто приходится делать несколько отдельных запросов.
- Необходимость версионирования при изменениях в структуре данных.
- Ограниченная гибкость при сложных или динамических структурах данных.

## Архитектурные стили и паттерны проектирования

REST API может быть реализован с использованием различных архитектурных стилей и паттернов [6]:

- Ресурсно-ориентированный стиль - основной принцип REST, где все взаимодействия строятся вокруг ресурсов и их идентификаторов.
- Многоуровневая архитектура - разделение на слои (например, контроллеры, сервисы, репозитории), что повышает модульность и поддерживаемость кода.
- Паттерны проектирования - такие как Repository (абстрагирование доступа к данным), Singleton (глобальные конфигурации), пагинация (управление большими коллекциями данных).

### 3. GraphQL: язык запросов для API

#### Концепция и основные принципы GraphQL

GraphQL - это современный язык запросов для API, разработанный Facebook и открытый сообществу, который позволяет клиентам получать именно те данные, которые им нужны, и ничего лишнего. В отличие от традиционного REST API, где структура ответа фиксирована сервером, GraphQL даёт клиенту полную гибкость в формировании запросов.

- Выбор полей (field selection)- В GraphQL клиент формирует запрос, указывая конкретные поля объекта, которые он хочет получить. Например, если нужно имя и электронная почта пользователя, запрос будет содержать только эти поля. Это позволяет избежать избыточной передачи данных (over-fetching) и получать ровно необходимую информацию.
- Единый эндпоинт- В отличие от REST, где существует множество URL для разных ресурсов, GraphQL использует один эндпоинт, через который проходят все запросы. Это упрощает архитектуру и управление API.
- Схема и типизация- В основе GraphQL лежит строго типизированная схема, описывающая доступные типы данных и поля. Схема служит контрактом между клиентом и сервером, позволяя клиенту знать, какие данные можно запросить и в каком формате они будут возвращены.
- Три основных типа операций:
  - Query - получение данных (аналог GET в REST).
  - Mutation - изменение данных (создание, обновление, удаление).
  - Subscription - подписка на события для получения обновлений в реальном времени.

## Структура запросов в GraphQL

Запросы GraphQL имеют иерархическую структуру, где указываются поля и вложенные поля, которые нужно получить. Пример простого запроса для получения имени пользователя с `id=1` и списка его подписчиков (ограниченных 50):

```
{
  user(id: "1") {
    name
    followers(limit: 50) {
      name
    }
  }
}
```

Рис 3.1. Пример простого запроса

Здесь `user` - поле запроса с аргументом `id`, `name` и `followers` - вложенные поля. Аргументы позволяют уточнять запросы и получать конкретные данные.

## Фрагменты и повторное использование запросов

GraphQL поддерживает фрагменты - именованные части запроса, которые можно переиспользовать в разных местах. Это помогает структурировать сложные запросы и избежать дублирования.

Пример использования фрагмента:

```
fragment UserInfo on User {  
  name  
  email  
}  
  
query {  
  user(id: 123) {  
    ...UserInfo  
    age  
  }  
}
```

Рис 3.2. Фрагмент

Фрагмент UserInfo содержит поля name и email, которые затем включаются в запрос с помощью ...UserInfo.

### Переменные в запросах

Для динамичности запросов в GraphQL используются **переменные**. Они позволяют параметризовать запросы, делая их более универсальными и удобными для повторного использования.

Пример запроса с переменной:

```
query getUser($id: Int!) {  
  user(id: $id) {  
    name  
    email  
  }  
}
```

Рис 3.3. запроса с переменной

Здесь \$id - переменная типа Int! (обязательный целочисленный параметр). При выполнении запроса переменной присваивается конкретное значение:

```
{  
  "id": 123  
}
```

Рис 3.4. Переменная

## Схема GraphQL

Схема - это описание всех типов данных, которые доступны через API, и их полей. Она задаёт структуру данных и правила, по которым сервер обрабатывает запросы.

Пример определения типа User в схеме:

```
type User {  
  id: ID!  
  name: String!  
  email: String!  
  age: Int  
}
```

Рис 3.5. User в схеме

- Восклицательный знак ! означает, что поле обязательно и не может быть null.
- Типы данных включают базовые (String, Int, Boolean, ID) и сложные (списки, вложенные объекты).

Схема позволяет клиенту и серверу согласованно взаимодействовать, а также служит основой для автодокументации и инструментов разработки.

## Преимущества GraphQL

- Гибкость запросов - клиент получает ровно те данные, которые нужны.
- Снижение количества запросов - можно получить связанные данные за один запрос, в отличие от REST, где нужны множественные вызовы.
- Отсутствие необходимости версионирования - изменения в схеме могут быть добавлены без нарушения существующих запросов.
- Типизация и самодокументируемость - строгая схема облегчает разработку и поддержку API.
- Поддержка подписок - возможность получать обновления в реальном времени.

Пример простого запроса и ответа

Запрос:

```
query {  
  user(id: 1) {  
    name  
    email  
  }  
}
```

Рис 3.6. простой запрос

Ответ сервера: в json

```
{
  "data": {
    "user": {
      "name": "Иван Иванов",
      "email": "ivan@example.com"
    }
  }
}
```

Рис 3.7. json ответ

Структура ответа повторяет структуру запроса, что упрощает обработку данных на клиенте.

GraphQL - это мощный и гибкий язык запросов для API, который позволяет создавать эффективные, удобные и масштабируемые интерфейсы взаимодействия между клиентом и сервером. Благодаря строгой типизации, возможности точного выбора данных и поддержке сложных операций, GraphQL становится всё более популярным инструментом в современной разработке.

## 4. Практическая реализация REST API

### Настройка окружения и инициализация проекта

Реализация REST API требует чёткого следования архитектурным принципам и понимания механизмов работы веб-серверов. Рассмотрим процесс создания полноценного RESTful API на Node.js с использованием фреймворка Express, включая настройку окружения, разработку эндпоинтов и тестирование.

Подробный пошаговый план, как создать и запустить простой REST API на Node.js с использованием Express.

## Создать новую папку проекта и инициализировать npm

```
mkdir rest-api  
cd rest-api  
npm init -y
```

## Установите необходимые зависимости

```
npm install express
```

express — веб-фреймворк для Node.js

## Создайте структуру проекта

```
rest-api/  
|  
├── app.js  
├── routes/  
|   └── users.js  
└── controllers/  
    └── usersController.js
```

**Создайте файл controllers/usersController.js — логика обработки запросов**



```

// Временное хранилище пользователей в памяти (массив)
// Данные будут сбрасываться при перезапуске сервера
const users = [];
// Счетчик для генерации уникальных ID новых пользователей
let idCounter = 1;

// Контроллер для получения всех пользователей
exports.getUsers = (req, res) => {
  // Отправляем весь массив пользователей в формате JSON
  res.json(users);
};

// Контроллер для получения пользователя по ID
exports.getUserById = (req, res) => {
  // Ищем пользователя по ID из параметров запроса
  const user = users.find(u => u.id === parseInt(req.params.id));

  // Если пользователь не найден - отправляем ошибку 404
  if (!user) {
    return res.status(404).json({ error: "Пользователь не найден" });
  }

  // Отправляем найденного пользователя
  res.json(user);
};

// Контроллер для создания нового пользователя
exports.createUser = (req, res) => {
  // Извлекаем данные из тела запроса
  const { name, email, age } = req.body;

  // Проверка обязательных полей
  if (!name || !email) {
    return res.status(400).json({ error: "Необходимы name и email" });
  }

  // Создаем объект нового пользователя
  const newUser = {
    id: idCounter++, // Автоматически генерируем ID
    name,           // Имя из тела запроса
    email,          // Email из тела запроса
    age: age || null // Возраст (если не указан - null)
  };

  // Добавляем пользователя в массив
  users.push(newUser);

  // Отправляем созданного пользователя с кодом 201 (Created)
  res.status(201).json(newUser);
};

// Примечание: В реальных приложениях вместо хранения в памяти
// следует использовать базу данных для постоянного хранения данных

```

**Создайте файл routes/users.js — маршруты для пользователей**

```

// Импорт модуля Express и создание экземпляра роутера
// Роутер позволяет группировать связанные маршруты и middleware
const express = require('express');
const router = express.Router();

// Импорт контроллера пользователей, содержащего логику обработки запросов
const usersController = require('../controllers/usersController');

// Маршрут GET / - получение списка всех пользователей
// Обработчик запроса: usersController.getUsers
router.get('/', usersController.getUsers);

// Маршрут GET /:id - получение конкретного пользователя по ID
// :id - динамический параметр, доступный через req.params.id
// Обработчик запроса: usersController.getUserById
router.get('/:id', usersController.getUserById);

// Маршрут POST / - создание нового пользователя
// Данные пользователя передаются в теле запроса (req.body)
// Обработчик запроса: usersController.createUser
router.post('/', usersController.createUser);

// Экспорт роутера для использования в основном приложении
// Этот роутер будет подключен по пути /api/users в основном файле
module.exports = router;

```

## Создайте файл app.js — основной файл сервера

```

// Импорт модуля Express для создания REST API сервера
const express = require('express');
// Создание экземпляра Express-приложения
const app = express();

// Импорт маршрутов для работы с пользователями из отдельного файла
const usersRoutes = require('./routes/users');

// Middleware для автоматического парсинга JSON-тела входящих запросов
// Позволяет работать с данными из POST/PUT/PATCH-запросов в формате JSON
app.use(express.json());

// Подключение маршрутов для работы с пользователями
// Все запросы начинающиеся с /api/users будут перенаправлены в usersRoutes
app.use('/api/users', usersRoutes);

// Порт, на котором будет работать сервер
const PORT = 3000;
// Запуск сервера с выводом сообщения при успешном старте
app.listen(PORT, () => {
  console.log(`REST API сервер запущен на http://localhost:${PORT}`);
});

```

## Запустите сервер

node app.js

### Конечный результат

После запуска сервера можно выполнять HTTP-запросы к API по адресу <http://localhost:3000/api/users>.

### Выполнение запросов

Пример запроса создания пользователя (POST) в другом терминале tty2

```
curl -X POST http://localhost:3000/api/users \  
-H "Content-Type: application/json" \  
-d '{"name": "Иван Иванов", "email": "ivan@example.com", "age": 30}'  
curl -X POST http://localhost:3000/api/users \  
-H "Content-Type: application/json" \  
-d '{"name": "Цыганков Роман", "email": "tsygankov@bk.ru", "age": 23}'
```

Пример запроса получения списка пользователей (GET):

```
curl http://localhost:3000/api/users
```

Пример запроса получения пользователя по id (GET):

```
curl http://localhost:3000/api/users/2
```

## 5. Практическая реализация GraphQL API

Реализация GraphQL API требует глубокого понимания его архитектуры и экосистемы инструментов. Рассмотрим процесс создания полнофункционального GraphQL сервера на Node.js с использованием Express, Apollo Server и MongoDB, включая продвинутые техники оптимизации и обработки ошибок.

Создать новую папку проекта и инициализировать npm

```
mkdir graphql-test  
cd graphql-test
```

```
npm init -y
```

Установите необходимые зависимости

```
npm install apollo-server-express express graphql
```

express — веб-фреймворк для Node.js graphql — ядро GraphQL

**Создайте файл server.js — основной сервер GraphQL**

```
// Импорт необходимых модулей Apollo Server и Express
const { ApolloServer, gql } = require('apollo-server-express');
const express = require('express');

// Временные данные в памяти (замените базой данных в реальном приложении)
const users = [
  { id: '1', name: 'Alice', email: 'alice@example.com', posts: ['1'] },
  { id: '2', name: 'Bob', email: 'bob@example.com', posts: ['2'] },
];

const posts = [
  { id: '1', title: 'First Post', content: 'Hello World!', authorId: '1' },
  { id: '2', title: 'GraphQL Guide', content: 'Learn GraphQL in 5 mins', authorId: '2' },
];

// Определение GraphQL схемы с помощью SDL (Schema Definition Language)
const typeDefs = gql`
  type User {
    id: ID!           # Уникальный идентификатор (обязательное поле)
    name: String!      # Имя пользователя
    email: String!     # Электронная почта
    posts: [Post!]!    # Список постов пользователя (не может быть null)
  }

  type Post {
    id: ID!           # Уникальный идентификатор поста
    title: String!    # Заголовок поста
    content: String!   # Содержание поста
    author: User!      # Автор поста (связь с типом User)
  }

  type Query {
    users: [User!]!    # Запрос для получения всех пользователей
    user(id: ID!): User # Запрос пользователя по ID (может возвращать null)
    posts: [Post!]!    # Запрос для получения всех постов
  }

  type Mutation {
    createPost(        # Мутация для создания нового поста
      title: String!
      content: String!
      authorId: ID!
    ): Post!
  }
`;
```

```

// Резолверы - функции, реализующие логику работы с данными
const resolvers = {
  query: {
    users: () => users,      # Возвращает всех пользователей
    user: (_, { id }) => users.find(user => user.id === id), # Поиск пользователя по ID
    posts: () => posts,      # Возвращает все посты
  },
  mutation: {
    createPost: (_, { title, content, authorId }) => {
      # Создание нового поста и добавление в массив
      const newPost = {
        id: String(posts.length + 1),
        title,
        content,
        authorId,
      };
      posts.push(newPost);
      return newPost;
    },
  },
  # Кастомные резолверы для полей типов
  Post: {
    author: post => users.find(user => user.id === post.authorId), # Поиск автора поста
  },
  User: {
    posts: user => posts.filter(post => user.posts.includes(post.id)), # Фильтрация постов пользователя
  },
};

# Инициализация Apollo Server с настройками схемы и резолверов
const server = new ApolloServer({ typeDefs, resolvers });
const app = express();

# Асинхронная функция запуска сервера
async function startServer() {
  await server.start();          # Ожидаем инициализации Apollo Server
  server.applyMiddleware({ app }); # Интеграция Apollo с Express

  # Запуск сервера на порту 4000
  app.listen({ port: 4000 }, () =>
    console.log(`Server ready at http://localhost:4000${server.graphqlPath}`)
  );
}

startServer();

```

```
/*
Особенности реализации:

1. Единая конечная точка /graphql для всех запросов
2. Строгая типизация через схему
3. Возможность выполнять сложные запросы с объединением данных
4. Автоматическая документация API через GraphQL Introspection

Пример запроса для создания поста:
mutation {
  createPost(
    title: "New Post",
    content: "GraphQL is awesome!",
    authorId: "1"
  ) {
    id
    title
    author {
      name
    }
  }
}
```

## Запустите сервер

```
node server.js
```

## Конечный результат

После запуска сервера перейдите в браузере по адресу: <http://localhost:4000/graphql> Откроется интерфейс GraphQL, где можно выполнять запросы.

## Запросы

Получение пользователей с их постами:

```
query GetUsersWithPosts {
  users {
    name
    email
    posts {
      title
    }
  }
}
```

```
}
```

Создание нового поста:

```
mutation CreatePost {  
  createPost(  
    title: "New Post",  
    content: "GraphQL is awesome!",  
    authorId: "1"  
  ) {  
    id  
    title  
    author {  
      name  
    }  
  }  
}
```

Пример запроса получения пользователя по id:

```
query {  
  user(id: "1") {  
    id  
    name  
    email  
    age  
  }  
}
```

## 6. Сравнение практических реализаций REST и GraphQL

Глава посвящена детальному анализу практических аспектов реализации REST API и GraphQL API, выявлению их преимуществ и недостатков, а также оценке удобства и эффективности использования каждого из подходов в реальных приложениях.

### Модель запросов и ответов

**REST API** строится вокруг концепции ресурсов, каждый из которых доступен по уникальному URL (эндпоинту). Для получения связанных данных, например, информации о пользователе и его постах, необходимо выполнить несколько отдельных запросов к разным конечным точкам:

- `/user/<id>` — получение данных пользователя
- `/user/<id>/posts` — получение списка постов пользователя
- `/user/<id>/followers` — получение списка подписчиков

Это приводит к избыточной передаче данных (over-fetching), когда клиент получает больше информации, чем требуется, и к необходимости делать несколько запросов (проблема N+1).

**GraphQL** решает эти проблемы, позволяя клиенту сформировать один иерархический запрос, в котором можно указать именно те поля и связанные сущности, которые нужны. Например, в одном запросе можно получить имя пользователя, его посты и список подписчиков, без избыточных данных.

### Структура и гибкость ответа

- В REST структура ответа фиксирована сервером и зависит от конкретного эндпоинта. Клиент не может влиять на формат и объем возвращаемых данных. Это ограничивает гибкость и зачастую приводит к избыточной передаче данных.
- В GraphQL клиент полностью контролирует структуру ответа, указывая в запросе, какие поля и вложенные объекты ему нужны. Это снижает сетевой трафик и упрощает обработку данных на клиенте.

### Количество запросов и производительность



- REST требует нескольких запросов для получения связанных данных, что увеличивает задержки и нагрузку на сеть.
- GraphQL позволяет получить все необходимые данные одним запросом, что значительно снижает количество сетевых вызовов и улучшает производительность, особенно в мобильных и слабо подключённых приложениях.

### **Версионирование и эволюция API**

- В REST при изменении структуры данных часто требуется создавать новые версии API (например, /v1/users, /v2/users), что усложняет поддержку и развитие.
- В GraphQL благодаря гибкости схемы и возможности добавлять новые поля без удаления старых, версионирование зачастую не требуется. Клиенты запрашивают только те поля, которые им нужны, что облегчает эволюцию API.

### **Обработка ошибок**

- В REST ошибки обрабатываются через HTTP-коды статуса (например, 404, 500), что является простым и понятным механизмом.
- В GraphQL сервер обычно возвращает HTTP-статус 200, а ошибки передаются в поле errors в теле ответа. Это требует дополнительной обработки на клиенте, но позволяет возвращать частичные данные вместе с ошибками.

### **Кэширование**

- REST изначально поддерживает кэширование на уровне HTTP, что позволяет эффективно использовать CDN и прокси-серверы.
- GraphQL не имеет встроенного механизма кэширования HTTP-ответов из-за единого эндпоинта и динамической структуры запросов, однако существуют клиентские библиотеки (Apollo, Relay), предоставляющие продвинутое кэширование.

### **Удобство разработки и сопровождения**

- REST проще в освоении и реализации, особенно для простых API с фиксированной структурой данных и небольшим числом клиентов.
- GraphQL требует более сложного проектирования схемы и резолверов, но значительно упрощает работу с динамическими и сложными данными, ускоряет разработку клиентской части и снижает количество багов, связанных с избыточной или недостаточной загрузкой данных.

### Инструменты и экосистема

- REST поддерживается огромным количеством инструментов, фреймворков и стандартов (Swagger/OpenAPI, Postman и др.).
- GraphQL имеет развитую экосистему инструментов для разработки, тестирования и мониторинга (GraphiQL, Apollo Studio, GraphQL Playground), а также интеграцию с современными фронтенд-библиотеками.

### Итоговое сравнение

Параметр	REST API	GraphQL API
Количество запросов	Несколько запросов для связанных данных	Один запрос с вложенной структурой
Гибкость структуры данных	Фиксирована сервером	Определяется клиентом
Версионирование	Требуется при изменениях	Обычно не требуется
Кэширование	Встроенное HTTP-кэширование	Требуется клиентских решений
Обработка ошибок	HTTP-коды статуса	Ошибки в поле errors в ответе
Сложность реализации	Ниже, проще освоить	Выше, требует проектирования схемы

Поддержка реального времени	Через WebSocket, отдельные решения	Встроена через подписки (subscriptions)
Инструменты	Широкий набор, зрелые стандарты	Современные, активно развивающиеся

## Практические рекомендации

- REST подходит для простых и стабильных API с небольшим числом клиентов, где важна простота и кэширование.
- GraphQL оптимален для сложных, динамических приложений с разнообразными клиентами, где требуется гибкость запросов и минимизация сетевого трафика.

Практическая реализация REST и GraphQL демонстрирует, что оба подхода имеют свои сильные и слабые стороны. REST остаётся универсальным и простым решением, а GraphQL предлагает новые возможности для оптимизации и гибкости, особенно в современных SPA и мобильных приложениях. Выбор между ними должен базироваться на требованиях проекта, особенностях данных и инфраструктуры, а также опыте команды разработчиков.

## 7. Рекомендации по выбору подхода

Выбор между REST и GraphQL требует комплексного анализа требований проекта, архитектурных особенностей и бизнес-целей. Ниже представлены детальные рекомендации, основанные на практических сценариях и анализе сильных сторон каждого подхода.

### Когда выбирать REST API

Простые CRUD-приложения

REST идеален для систем с предсказуемыми операциями (создание, чтение, обновление, удаление), где структура данных стабильна. Примеры:

- Блоги с фиксированным набором сущностей (посты, комментарии).
- Админ-панели для управления контентом.

Преимущество: Быстрая разработка благодаря стандартизированным HTTP-методам и эндпоинтам.

Высокие требования к кэшированию

REST поддерживает HTTP-кэширование на уровне ресурсов, что критично для:

- Высоконагруженных приложений (новостные порталы, агрегаторы).
- Систем с частыми запросами статических данных (каталоги товаров).

Публичные API для сторонних разработчиков

REST предпочтителен из-за:

- Простоты интеграции (интуитивные эндпоинты, документация в стиле OpenAPI).
- Широкой поддержки инструментов (Postman, Swagger).

Микросервисная архитектура

REST эффективен для:

- Слабо связанных сервисов с чёткими границами.
- Систем, где каждый микросервис управляет своим набором ресурсов.

## **Когда выбирать GraphQL**

Динамические клиентские требования

GraphQL исключает over-fetching/under-fetching, что критично для:

- Мобильных приложений с ограниченным трафиком.
- Сложных интерфейсов (дашборды, персонализированные ленты), где клиент определяет структуру данных.

Агрегация данных из множества источников

GraphQL упрощает объединение данных:

- В гибридных системах (legacy REST API + новые сервисы).
- При интеграции с внешними API (платежные системы, соцсети).

Часто меняющиеся требования к данным

Гибкость схемы GraphQL позволяет:

- Избежать версионирования API при добавлении новых полей.
- Ускорить итерации в стартапах и agile-проектах.

Режим реального времени

Подписки (subscriptions) в GraphQL поддерживают:

- Чат-приложения.
- Трекеры доставки.
- Системы мониторинга.

## Гибридные решения

Постепенная миграция

- Используйте REST для стабильных модулей, GraphQL — для новых функций.
- Пример: eBay внедрял GraphQL поверх существующих REST-сервисов для мобильных клиентов.

Оптимизация производительности

- Кэшируйте часто запрашиваемые данные через REST.
- Сложные запросы обрабатывайте через GraphQL.

Специализация по типам клиентов

- REST для B2B-партнёров (предсказуемость).
- GraphQL для мобильных и веб-клиентов (гибкость).

## Критерии выбора: сводная таблица

Критерий	REST API	GraphQL
Структура данных	Фиксированная	Динамическая
Кэширование	На уровне HTTP	Требует клиентских решений

Сложность запросов	Линейная	Возможны глубоко вложенные запросы
Экосистема	Зрелая (Swagger, Postman)	Активно развивающаяся (Apollo)
Обучение команды	Низкий порог входа	Требует изучения схемы и резолверов

Выбор между REST и GraphQL — не бинарный. Ключевые факторы:

1. Характер данных: Статичные vs динамичные.
2. Клиентские устройства: Мобильные vs десктоп.
3. Скорость разработки: Стандартизация vs гибкость.
4. Инфраструктура: Наличие legacy-систем vs зеленое поле.

Как показывает практика GitHub и Shopify, гибридный подход часто становится оптимальным решением, позволяя сочетать преимущества обеих технологий.

## 8. Заключение

Сравнение подходов к разработке API — REST и GraphQL — демонстрирует, что оба метода имеют уникальные преимущества и оптимальны для разных сценариев. REST, основанный на ресурсно-ориентированной архитектуре, остаётся надёжным выбором для простых приложений с предсказуемыми операциями, где критичны стандартизация и встроенное HTTP-кэширование. Его сила проявляется в публичных API, микросервисных экосистемах и проектах, требующих высокой производительности за счёт кэширования статичных данных. Однако ограничения REST, такие как over-fetching/under-fetching и необходимость множественных запросов для связанных данных, становятся заметными в сложных системах с динамическими требованиями. GraphQL, напротив, предлагает революционную гибкость, позволяя клиентам

точно формулировать запросы и получать связанные данные за один вызов, что особенно ценно для мобильных приложений, дашбордов и систем с часто меняющимися требованиями. Несмотря на сложности с кэшированием и необходимость продуманного проектирования схемы, GraphQL активно набирает популярность в компаниях вроде GitHub и Netflix, где важны скорость итераций и адаптивность. Современные тенденции показывают, что гибридные подходы, сочетающие REST для стабильных модулей и GraphQL для динамических компонентов, часто становятся оптимальным решением. Выбор между технологиями должен основываться на специфике проекта: характере данных, требованиях клиентов и инфраструктурных ограничениях. В конечном итоге, эволюция API-разработки продолжается, и оба подхода будут оставаться востребованными, дополняя друг друга в стремлении к балансу между простотой, гибкостью и производительностью.

## 9. Список использованных источников

1. Что значит API: принцип работы, функции, виды, примеры // Макхост [Электронный ресурс]. – Режим доступа: <https://mchost.ru/articles/chto-takoe-api-i-kak-eto-rabotaet/>
2. Что такое API и как он работает // Skillbox [Электронный ресурс]. – Режим доступа: [https://skillbox.ru/media/code/chto\\_takoe\\_api/](https://skillbox.ru/media/code/chto_takoe_api/)
3. API от А до Я (теория и практика) // Habr [Электронный ресурс]. – Режим доступа: <https://habr.com/ru/articles/768752/>
4. REST API Tutorial: What is REST?// restfulapi [Электронный ресурс]. – Режим доступа: <https://restfulapi.net/>
5. Understanding REST API: Basics, Methods, and Best Practices // Data Science Dojo [Электронный ресурс]. – Режим доступа: <https://datasciencedojo.com/blog/understanding-rest-api/>
6. Understanding REST API Architectural Styles and Design Patterns - Xapi // blog.xapihub.io [Электронный ресурс]. – Режим доступа:

<https://blog.xapihub.io/2023/08/18/Understanding-REST-API-Architectural-Styles-and-Design-Patterns.html>

7. GraphQL: A query language for your API [Электронный ресурс]. – Режим доступа: <https://graphql.org>.
8. Frigård E. GraphQL vs REST: A Comparison of Runtime Performance: Master's thesis / Uppsala University. – 2022. – 45 p.
9. Santosa B., Pratomo A.H., Wardana R.M. et al. Performance Optimization of GraphQL API Through Advanced Object Deduplication Techniques // Journal of Computer Science and Engineering. – 2023. – Vol. 17, № 4. – P. 195-206.
10. Pautasso C., Zimmermann O., Leymann F. RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision // Proc. 17th Int. Conf. World Wide Web. – 2008. – P. 805–814.
11. Richardson L., Ruby S. RESTful Web Services / Leonard Richardson, Sam Ruby. – O'Reilly Media, 2007. – 240 p.
12. Hartmann H., Timmermann J. REST vs. GraphQL: An Empirical Comparison of API Performance // Proc. IEEE Int. Conf. Web Services. – 2022. – P. 401-408.
13. Fielding R.T. Architectural Styles and the Design of Network-based Software Architectures/ University of California. – 2000. – 180 с.

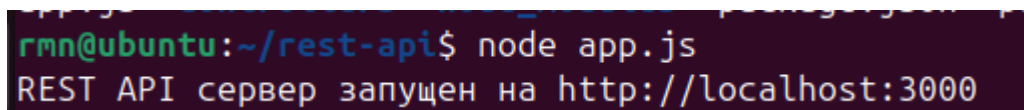
## 10. Скриншоты проекта

Документация:

<https://github.com/mdx9/guzeev>

### Rest API

запуск приложения на ОС ubuntu 22.04 LTS



```
rmn@ubuntu:~/rest-api$ node app.js
REST API сервер запущен на http://localhost:3000
```

Рис 10.1. запуск приложения

Начальная страница <http://localhost:3000/api/users>



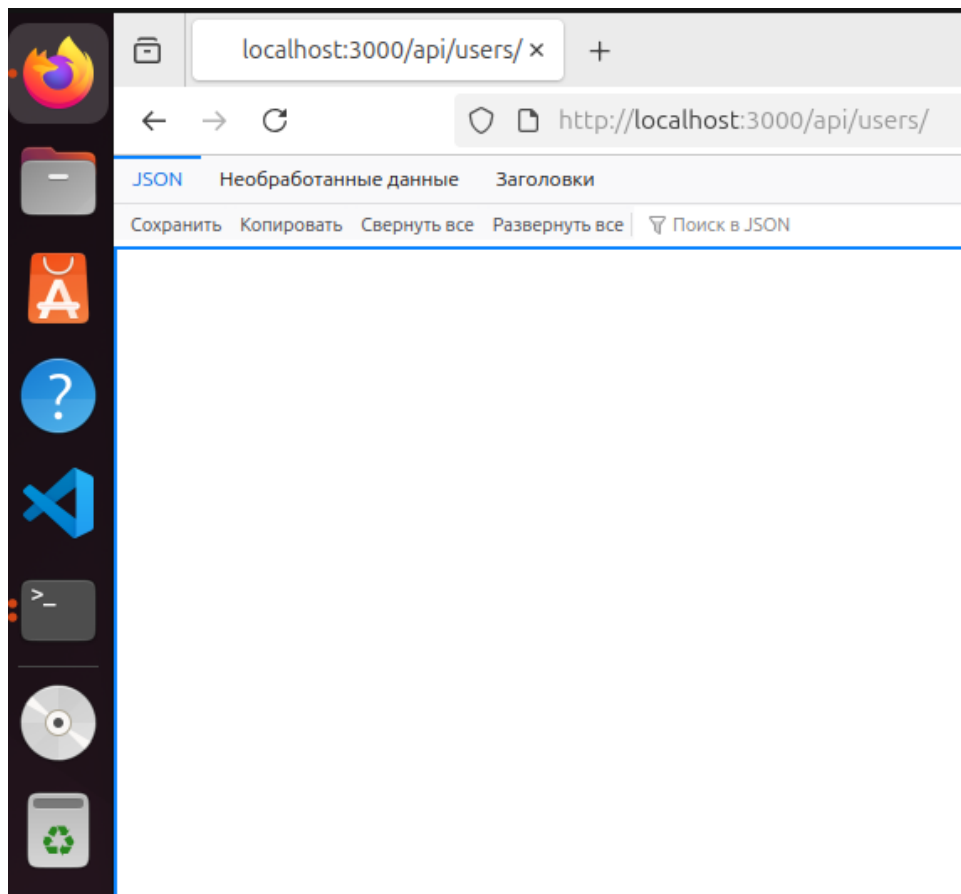


Рис 10.2. Начальная страница

Открываем новый терминал `tty`, и вписываем запрос создания пользователя (POST)

```
curl -X POST http://localhost:3000/api/users \  
-H "Content-Type: application/json" \  
-d '{"name": "Иван Иванов", "email": "ivan@example.com", "age": 30}'
```

```
rmn@ubuntu:~/rest-api$ curl -X POST http://localhost:3000/api/users \  
-H "Content-Type: application/json" \  
-d '{"name": "Иван Иванов", "email": "ivan@example.com", "age": 30}'  
{ "id": 1, "name": "Иван Иванов", "email": "ivan@example.com", "age": 30 }rmn@ubuntu:~/re  
st-api$
```

Рис 10.3 создание пользователя

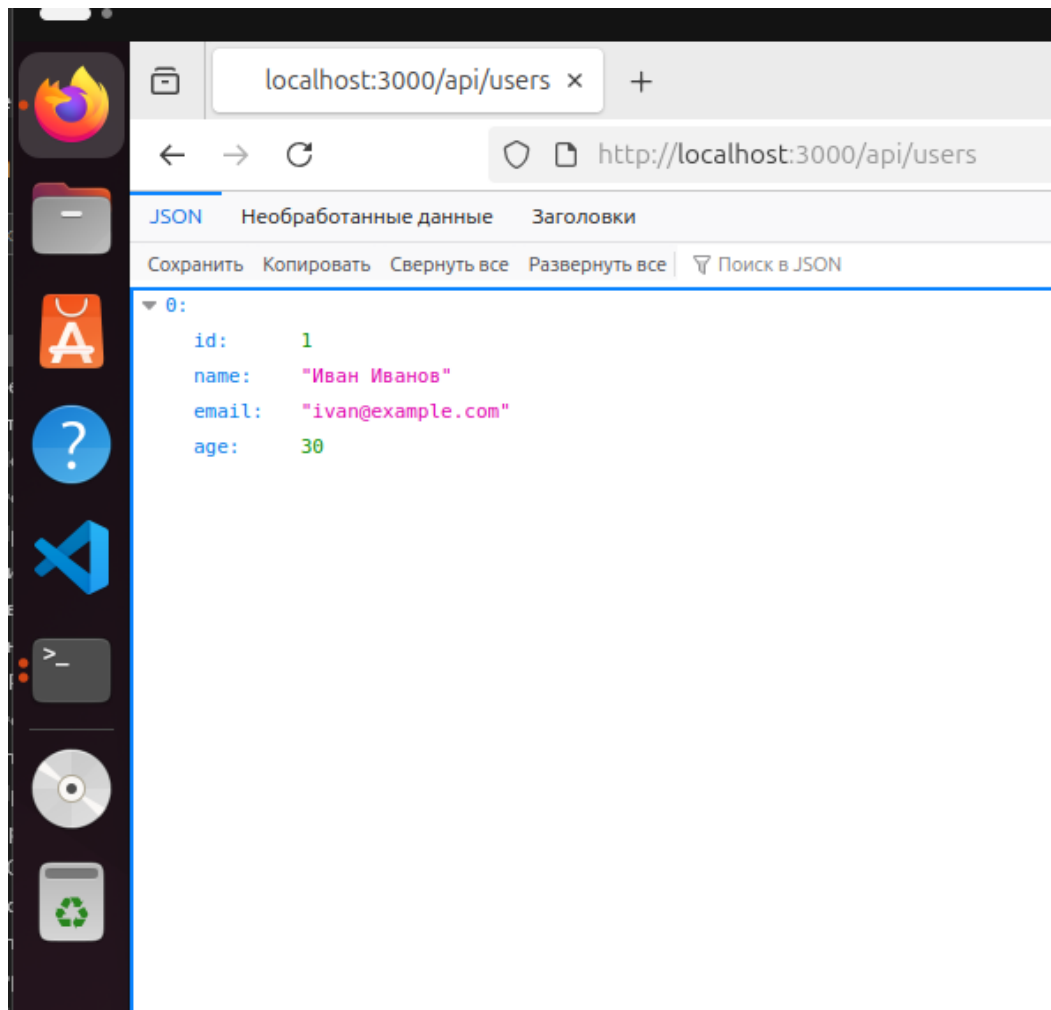


Рис 10.4 результат в браузере

Создадим второго пользователя

```
curl -X POST http://localhost:3000/api/users \  
-H "Content-Type: application/json" \  
-d '{"name": "Цыганков Роман", "email": "tsygankov@bk.ru", "age": 23}'
```

Выведем всех пользователей на экран

Пример запроса получения списка пользователей (GET):

```
curl http://localhost:3000/api/users
```

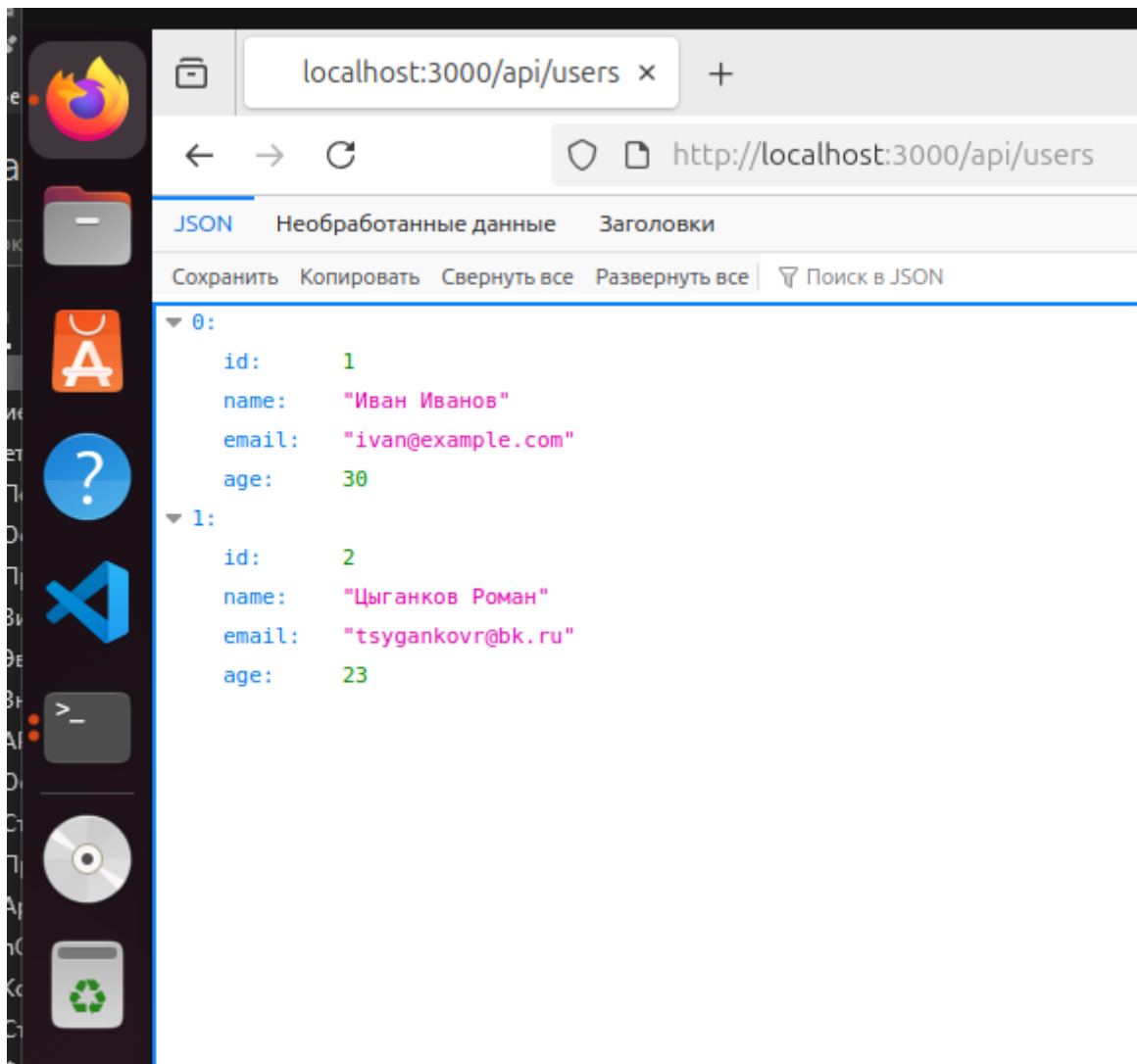


Рис 10.5 Список всех пользователей

Выведем одного пользователя

Пример запроса получения пользователя по id (GET):

```
curl http://localhost:3000/api/users/2
```

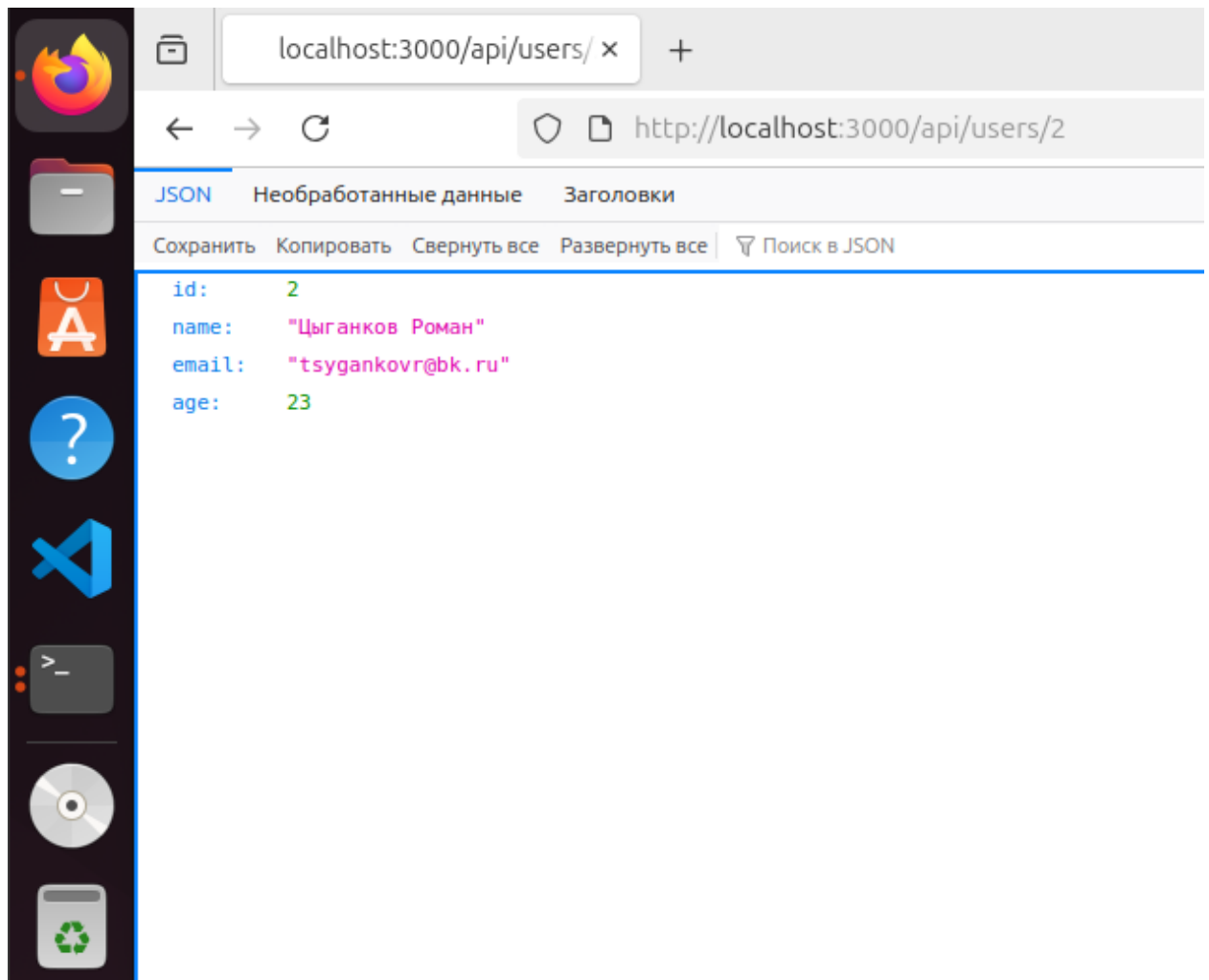


Рис 12.6 Один пользователь

## GrahpQL

### Запуск приложения

```
rmn@ubuntu:~/graphql-test$ node server.js
Server ready at http://localhost:4000/graphql
S
```

Рис 10.7 Запуск приложения

### Начальная страница apollo-server

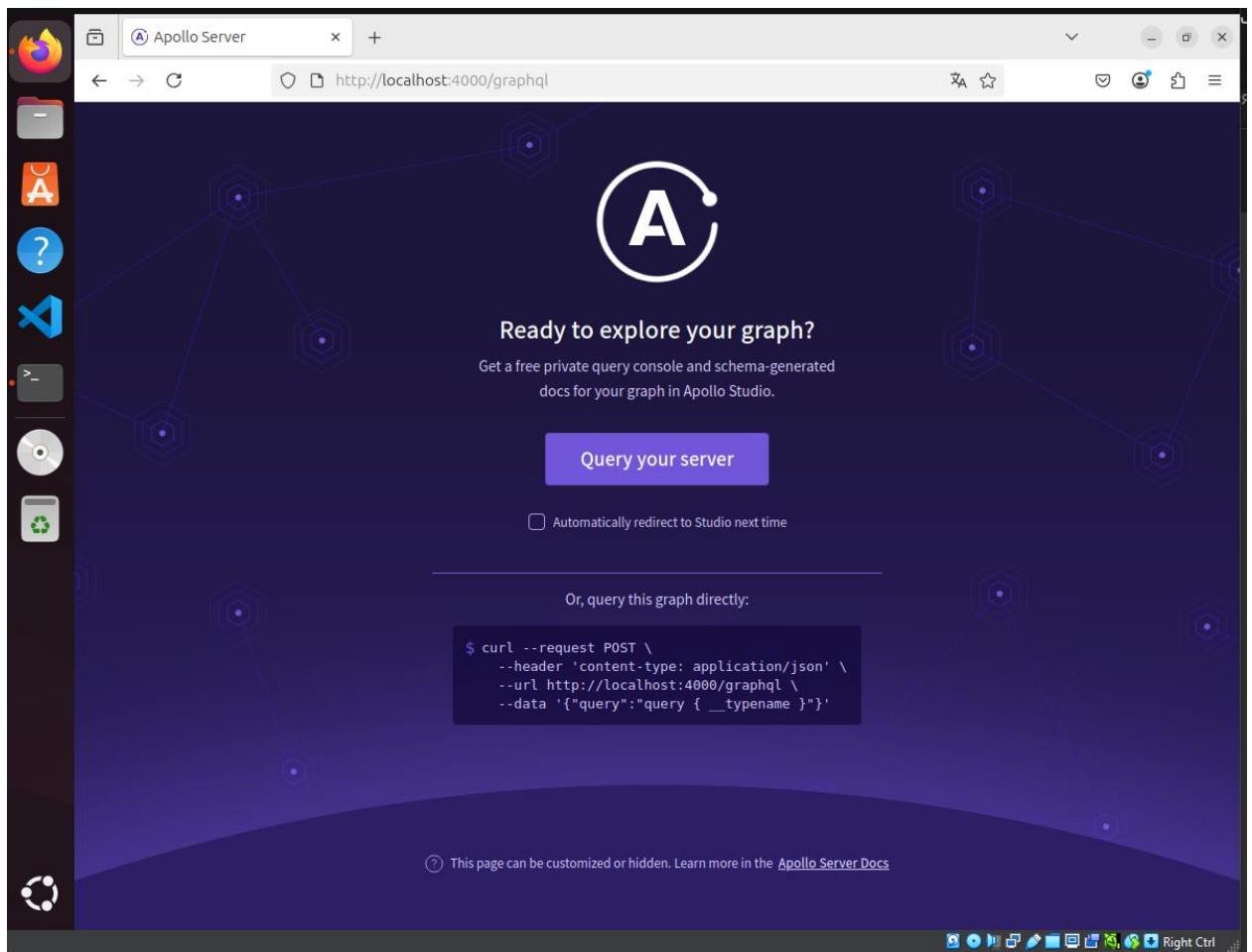


Рис 10.8 Начальная страница

После нажатия «Query your server», переносит нас в sandbox с пользовательским интерфейсом

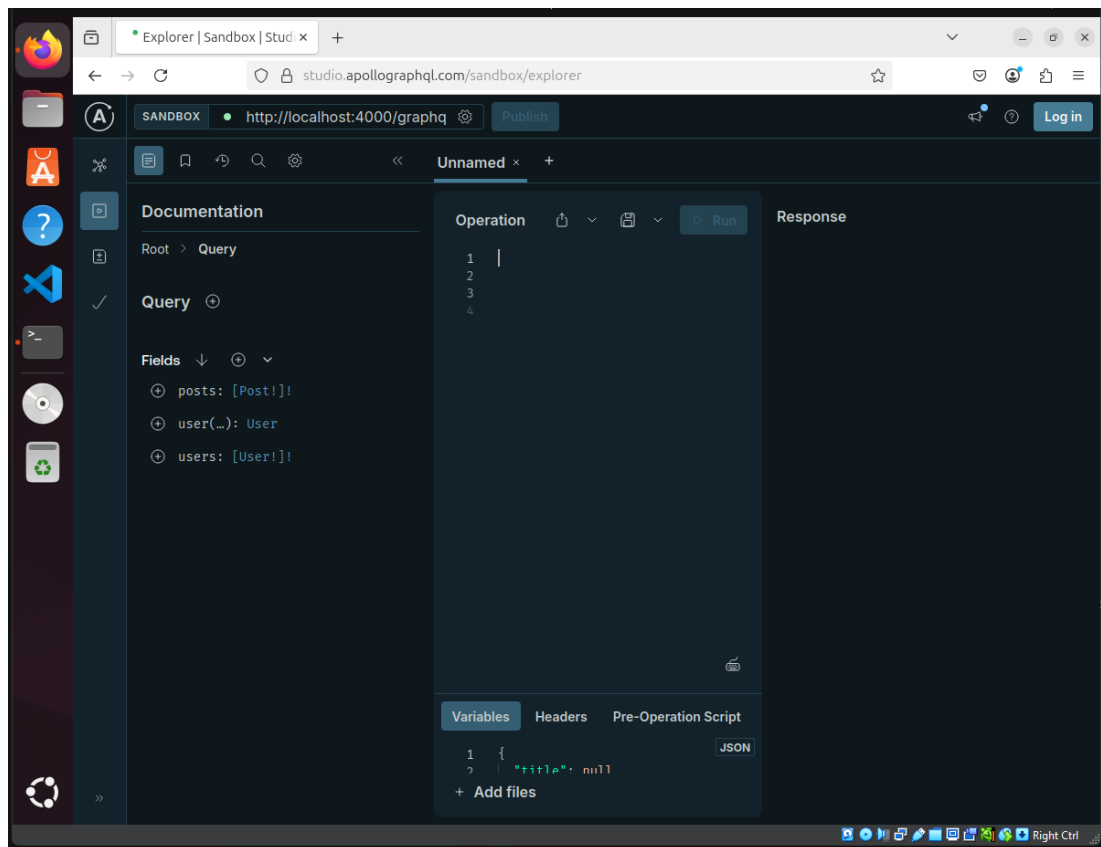


Рис 10.9 Пользовательским интерфейс graphql

Делаем несколько запросов

Получение пользователей с их постами:

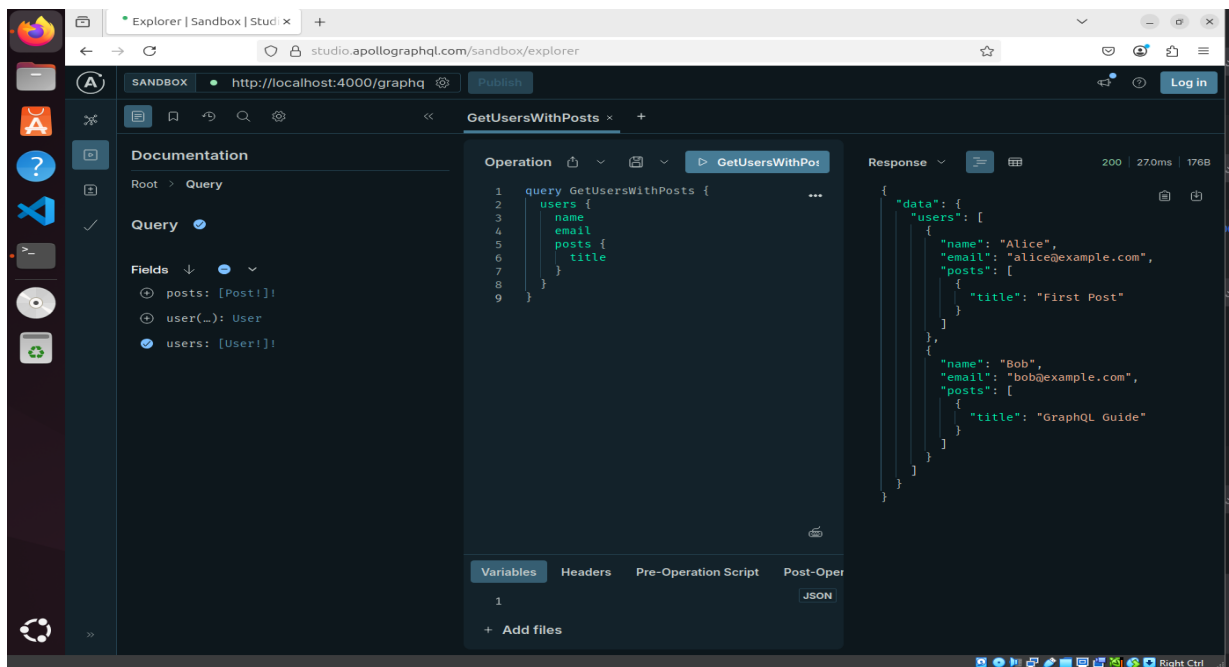


Рис 10.10 Получение пользователей

Создание нового поста:

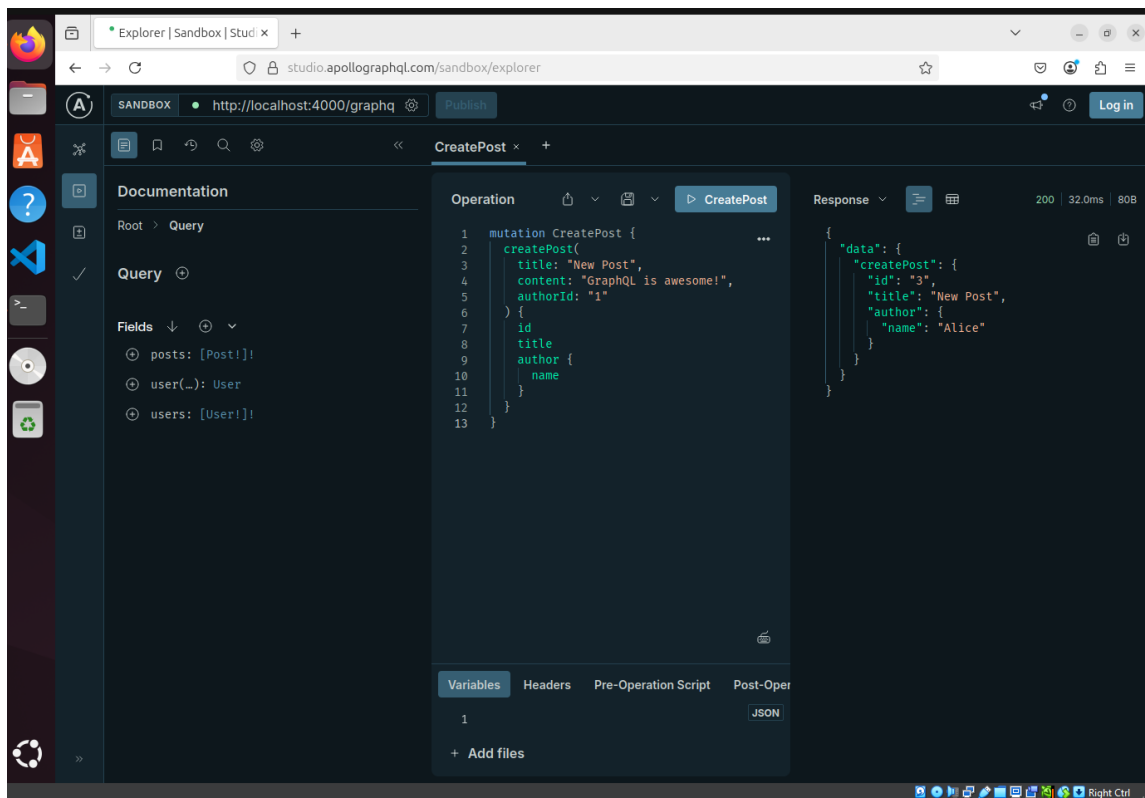


Рис 10.11 Новый пост

Пример запроса получения пользователя по id:

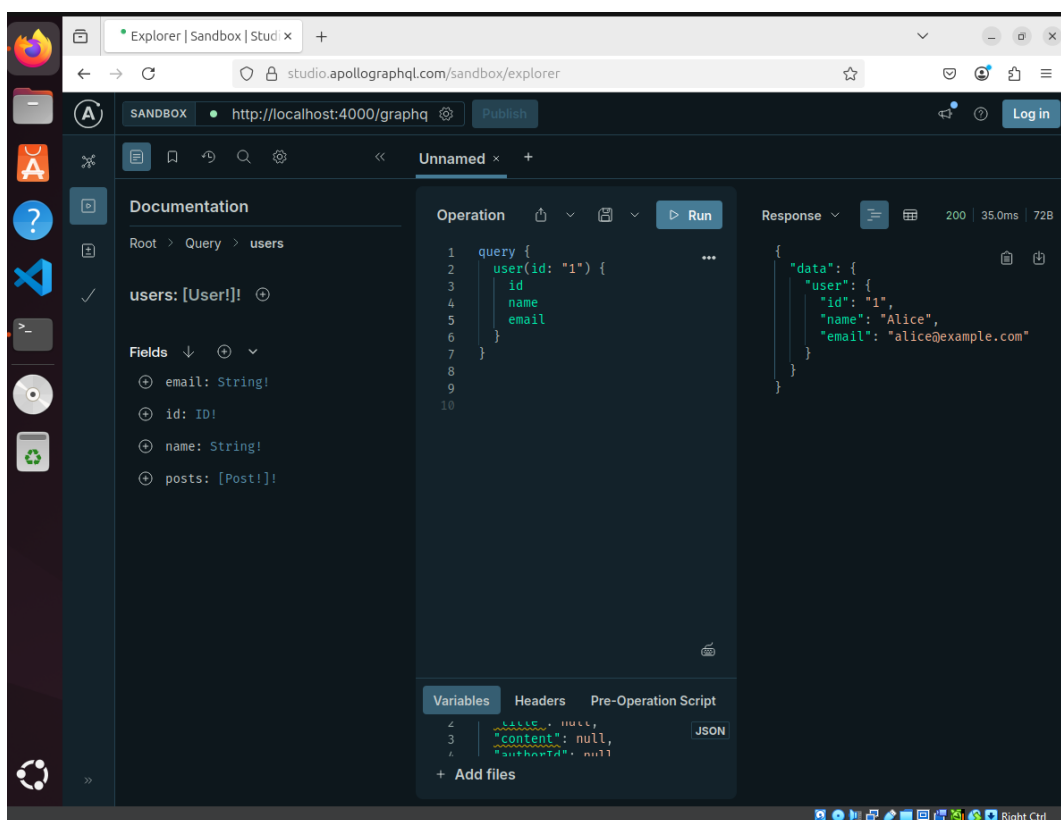


Рис 10.12 получения пользователя по id