УДК 004.9:004.4
DOI 10.17513/snt.40041

СРАВНИТЕЛЬНЫЙ АНАЛИЗ ПОДХОДОВ К ОРГАНИЗАЦИИ КЛИЕНТ-СЕРВЕРНОГО ВЗАИМОДЕЙСТВИЯ В СОВРЕМЕННЫХ ВЕБ-ПРИЛОЖЕНИЯХ, НА ПРИМЕРЕ REST API И GRAPHQL

[1]Кожанов П.С., [1, 2]Готская И.Б.

[1]ФГАОУ ВО «Национальный исследовательский университет ИТМО», Санкт-Петербург, e-mail: PKozhanov99@inbox.ru;
[2]ФГБОУ ВО «Российский государственный педагогический университет имени А.И. Герцена», Санкт-Петербург, e-mail: iringot@mail.ru

В статье рассматриваются ключевые подходы к организации клиент-серверного взаимодействия в современных веб-приложениях на примере наиболее популярных архитектурных решений REST API и GraphQL. Авторы анализируют основные проблемы в эволюции клиент-серверного взаимодействия и обосновывают научную новизну проведенного исследования. Целью исследования является проведение сравнительного анализа эффективности архитектурных подходов клиент-серверного взаимодействия REST API и GraphQL. В статье также представлена методология исследования и результаты сравнительного анализа архитектурных подходов REST API и GraphQL по следующим критериям: извлечение данных, производительность (скорость), масштабируемость, организация кода, инструментарий, обработка ошибок, безопасность и кэширование. Для наглядного представления преимуществ и недостатков обоих подходов были разработаны два веб-приложения, каждое из которых реализует свой метод получения, обработки и визуализации данных с сервера на клиенте. В результате анализа определены случаи и условия, при которых следует использовать REST API или GraphQL, что позволяет разработчикам делать обоснованный выбор архитектуры для своих проектов. Особое внимание уделено вопросам интеграции с существующими системами и адаптации к изменяющимся условиям, а также вопросам обеспечения безопасности и управления правами доступа в контексте использования данных подходов. Таким образом, результаты исследования вносят значительный вклад в понимание и оптимизацию клиент-серверного взаимодействия в контексте развития современных веб-технологий.

Ключевые слова: клиент-серверное взаимодействие, REST API, GraphQL, архитектурный подход, чрезмерная выборка, масштабируемость, недостаточная выборка, кэширование

COMPARATIVE ANALYSIS OF APPROACHES TO ORGANIZING CLIENT-SERVER INTERACTION IN MODERN WEB APPLICATIONS, USING THE EXAMPLE OF REST API AND GRAPHQL

[1]Kozhanov P.S., [1, 2]Gotskaya I.B.

[1]ITMO University, Saint Petersburg, e-mail: PKozhanov99@inbox.ru;
[2]Herzen State Pedagogical University of Russia, Saint Petersburg, e-mail: iringot@mail.ru

The article discusses in detail the key approaches to the organization of client-server interaction in modern web applications using the example of the most popular architectural solutions REST API and GraphQL. The authors analyze the main problems in the evolution of client-server interaction and substantiate the scientific novelty of the study. The purpose of the study is to conduct a comparative analysis of the effectiveness of architectural approaches of client-server interaction between REST API and GraphQL. The article also presents the research methodology and the results of a comparative analysis of the architectural approaches of REST API and GraphQL according to the following criteria: data extraction, performance (speed), scalability, code organization, instrumentation, error handling, security and caching. To visually present the advantages and disadvantages of both approaches, two web applications have been developed, each of which implements its own method of receiving, processing and visualizing data from the server to the client. As a result of the analysis, the cases and conditions under which the REST API or GraphQL should be used are identified, which allows developers to make an informed choice of architecture for their projects. Special attention is paid to the issues of integration with existing systems and adaptation to changing requirements. Additionally, the issues of security and access rights management in the context of using these approaches are considered. Thus, the results of the study make a significant contribution to understanding and optimizing client-server interaction in the context of modern web technologies.

Keywords: client-server interaction, REST API, GraphQL, architectural approach, scalability, over-fetching, under-fetching, caching

Клиент-серверное взаимодействие является важнейшим аспектом в разработке программного обеспечения, поэтому столь важно правильно выбрать подход для его организации. Изначально все веб-приложения использовали один подход для формирования клиент-серверного взаимодействия – REST (Representational State Transfer). Архитектура REST использовалась с XML, однако вскоре после этого Дугласом Крокфордом была разработана и стандартизирована технология JSON (JavaScript Object Notation), обеспечивающая понятный формат данных. Вскоре данная технология

внедрилась в веб-проекты и продолжает использоваться до настоящего времени [1].

Позже появились альтернативы, позволяющие более гибко настраивать это взаимодействие и получать модифицированные структуры данных. Одним из таких подходов является GraphQL. В этой связи возникает вопрос, какой из данных подходов является лучшим для различных типов веб-приложений.

Большинство отечественных исследований основывалось исключительно на теоретических выкладках, не подтвержденных экспериментальными данными, что определяет научную новизну исследования для российского научного сообщества – экспериментально подтвержденные сильные и слабые стороны подходов REST API и GraphQL [2].

Цель исследования – проведение сравнительного анализа эффективности архитектурных подходов клиент-серверного взаимодействия REST API и GraphQL.

Для проведения исследования были сформированы конкретные критерии: извлечение данных, производительность (скорость), масштабируемость, организация кода, инструментарий, обработка ошибок, безопасность и кэширование [3].

Материалы и методы исследования

Системное решение поставленных исследовательских задач потребовало использования следующих методов: анализ, сравнение, обобщение, эксперимент. Проанализированы прикладные сценарии, демонстрирующие практическую значимость рассматриваемых архитектурных подходов. Для проведения сравнительного анализа были созданы два приложения, одно из которых использовало REAS API, другое – GraphQL для получения данных. Далее было экспериментально показано, по каким критериям выигрывает каждый из подходов.

Инструментарий REST API и GraphQL. На сегодняшний день используются различные инструменты для отслеживания состояния запросов, посылаемых с клиентской части веб-приложения. В рамках экспериментального проекта REST API подключена библиотека *axios*, которая используется для выполнения HTTP-запросов. Другим важным инструментом для отслеживания состояния запросов является *Fetch API*. Он представляет собой встроенный в браузер API для отправки HTTP-запросов, он требует ручной обработки состояния запросов.

Не менее важной составляющей клиент-серверного взаимодействия является инструментарий для дебага, который дает возможность тестировать запросы. Для REST API таким инструментом является *Postman*, который позволяет создавать и отправлять запросы, а также просматривать ответы.

GraphQL также имеет свои инструменты для отслеживания состояния запросов. Среди таких инструментов главным является *Apollo Client*. Это популярная библиотека для работы с GraphQL в клиентской части. Она предоставляет возможности отслеживания состояния запросов, кеширования и управления состоянием.

Другими важными инструментами для дебага GraphQL являются *GraphQL Playground* и *Apollo Client Devtools*. GraphQL Playground представляет собой интерактивную среду для выполнения запросов и их отладки; она поставляется вместе с большинством серверов GraphQL.

Apollo Client Devtools – это расширение для браузеров, предоставляющее инструменты для отладки Apollo Client, включая просмотр кэша и выполнение запросов. Также в этом инструменте можно выполнять GraphQL-запросы и мутации в интерактивном режиме.

Таким образом, можно сделать вывод, что REST API часто требует больше ручной работы для отслеживания состояния запросов. Инструменты, такие как Postman, предоставляют средства для дебага, но требуют дополнительных шагов. GraphQL, особенно при использовании Apollo Client, предоставляет более удобные инструменты для отслеживания состояния запросов и дебага.

Организация кода для работы с REST API и GraphQL. Организация кодовой базы и ее удобство является важнейшей составляющей для удобной разработки и дальнейшей поддержки веб-приложения. GraphQL является полноценным языком запросов, который разработчику необходимо освоить перед внедрением данного подхода в проект. В случае с экспериментальными проектами данное утверждение полностью подтверждается на практике.

GraphQL ориентирован на типы данных, что позволяет более гибко определять, какие данные клиент хочет получить. Разработчик явно задает типы, которые должны быть задействованы в процессе этого взаимодействия.

Кроме того, один GraphQL эндпоинт обслуживает все запросы, что уменьшает количество точек входа и снижает сложность кода. Также клиент может запрашивать только необходимые данные, избегая избыточной информации.

Что касается второго подхода, REST API часто организуется вокруг конечных точек

(endpoints), что может привести к созданию нескольких точек входа для различных ресурсов. Сами же ресурсы и методы запросов (GET, POST, PUT, DELETE) определяют структуру кода.

Поскольку созданные экспериментальные приложения являются небольшими по объему, то невозможно с точностью указать, какой из данных подходов сработал лучше с точки зрения организации кодовой базы. Однако, несмотря на это, даже в таком случае GraphQL потребовал дополнительных настроек и более сложного написания кода. Если же разработчики имеют дело с огромным проектом, то REST API во многих аспектах может быть неэффективным подходом в силу причин, указанных выше. Таким образом, в общем случае для небольших проектов имеет смысл использовать REST, а для масштабных приложений со сложной структурой данных – GraphQL.

Извлечение данных в REST API и GraphQL. Методология REST и язык запросов GraphQL представляют собой различные подходы к извлечению данных из сервера. REST использует стандартные HTTP методы (GET, POST, PUT, DELETE) для извлечения данных [4].

Как видно из рис. 1, ответ в подходе REST имеет простую структуру и позволяет клиенту запрашивать только данные, которые необходимы, в формате JSON (рис. 1).

Данный подход присылает на клиент полную структуру данных со всеми полями. Простота данного подхода очевидна, несмотря на то, что часть данных фактически не используется на клиентском интерфейсе. Но в случаях, когда нужно получить данные из нескольких связанных ресурсов, REST может привести к проблеме «избыточной связанности» (или over-fetching). GraphQL, напротив, позволяет клиенту запрашивать именно те данные, которые нужны, используя гибкий и мощный язык запросов, как видно из рис. 2. Однако сложность GraphQL заключается в том, что надо быть внимательным при проектировании схемы данных, чтобы избежать избыточных запросов, а также обеспечить эффективное использование сетевых ресурсов [5].

На рис. 2 видно, что в данном случае на клиентскую часть попадают только поля *id*, *firstName* и *e-mail,* несмотря на то, что в базе данных имеется также поле password. Исходя из этого можно подтвердить, что GraphQL позволяет избежать избыточной связанности данных, обеспечивая клиенту только необходимую информацию.

Таким образом, когда нужно получить простые данные или при работе с уже существующим REST API, можно оставаться при REST. Но если у нас сложная структура данных или большая гибкость в запросах, использование GraphQL будет предпочтительнее.

Обработка ошибок в подходах REST API и GraphQL. Подход REST часто использует стандартные HTTP статус-коды для обозначения успешных или неуспешных операций. Например, при запросе ресурса успешный ответ может иметь статус-код 200, а при ошибке – код 404 (при ошибке на клиентской части) или 500 (при ошибке на сервере). REST также может включать дополнительную информацию об ошибке, например, в теле ответа, чтобы уточнить причину ошибки.

GraphQL, с другой стороны, использует стандартный формат JSON для возврата ошибок. В каждом ответе от сервера в объекте «errors» могут содержаться информация о произошедших ошибках. В экспериментальном веб-приложении исправим запрос так, чтобы на клиентскую часть вернулась ошибка. Например, в запрос *getAllUsers* добавим поле *city*, которое изначально не предусматривалось в GraphQL-схеме.

Когда в GraphQL-запросе запрашивается поле, которое не было предварительно указано в схеме, сервер GraphQL возвращает ошибку с сообщением «Cannot query field <имя_поля> on type <тип>». Например, если в схеме GraphQL не определено поле «city» для типа «User», а клиент пытается запросить это поле, сервер вернет ошибку данного вида. Таким образом, клиент получает информацию о том, что его запрос содержит недопустимое поле, что помогает ему правильно скорректировать запрос.

Такие подробные сообщения об ошибках являются одним из преимуществ GraphQL, поскольку они помогают разработчикам более точно и быстро обнаруживать и исправлять ошибки в запросах, улучшая тем самым процесс разработки.

Таким образом, сложность обработки ошибок в обоих подходах зависит от конкретной реализации. В подходе REST разработчики могут использовать стандартные библиотеки для обработки HTTP статусов и кодов ошибок. В GraphQL можно использовать те же библиотеки, но также нужно быть внимательным при работе с объектом «errors» в формате JSON.

Безопасность в подходах REST API и GraphQL. При сравнении безопасности в подходах REST и GraphQL, следует учитывать несколько аспектов, таких как возможности аутентификации, авторизации, контроля доступа к данным и защиты от нежелательных запросов.

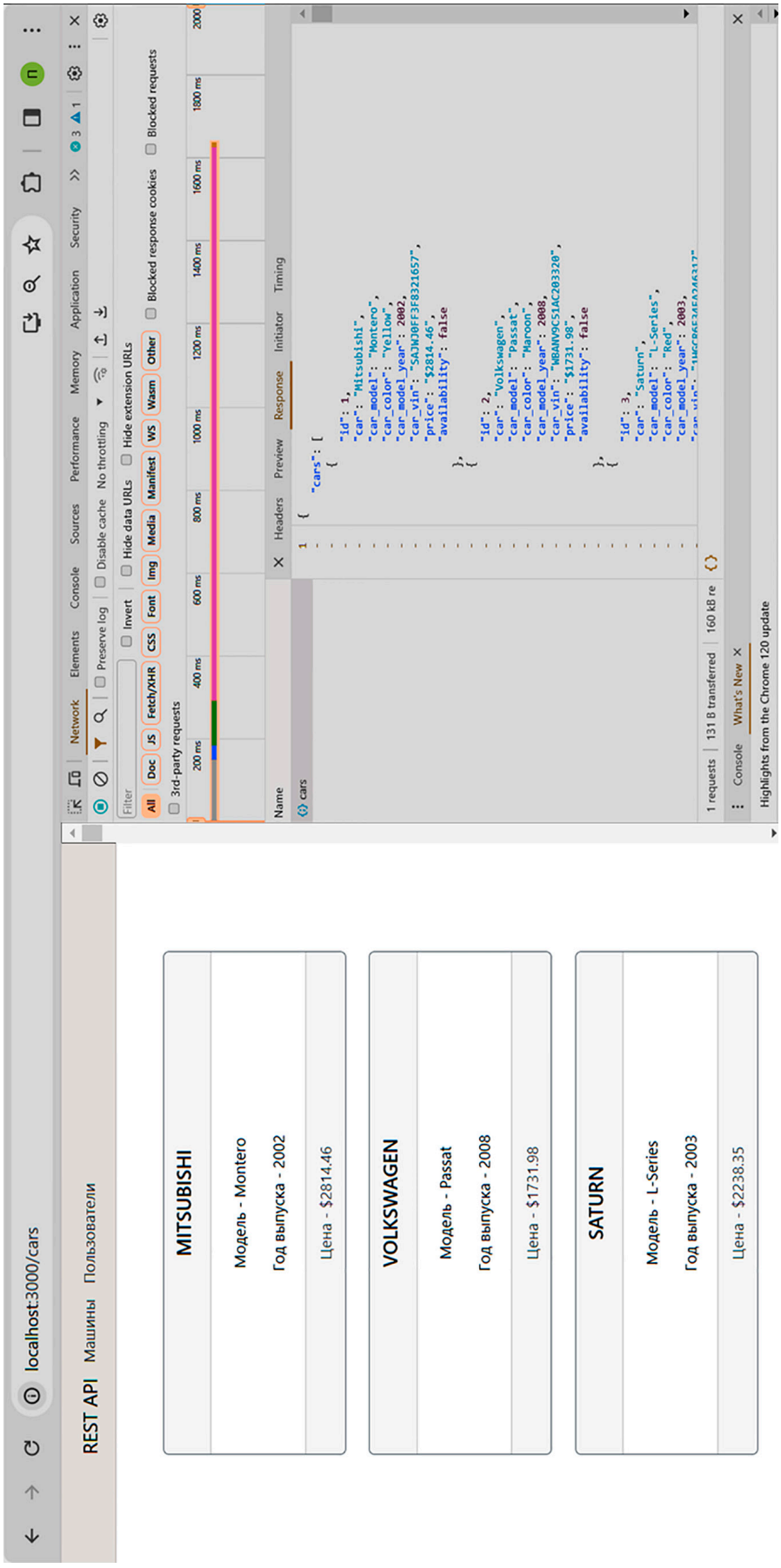

Рис. 1. Ответ от сервера на GET-запрос

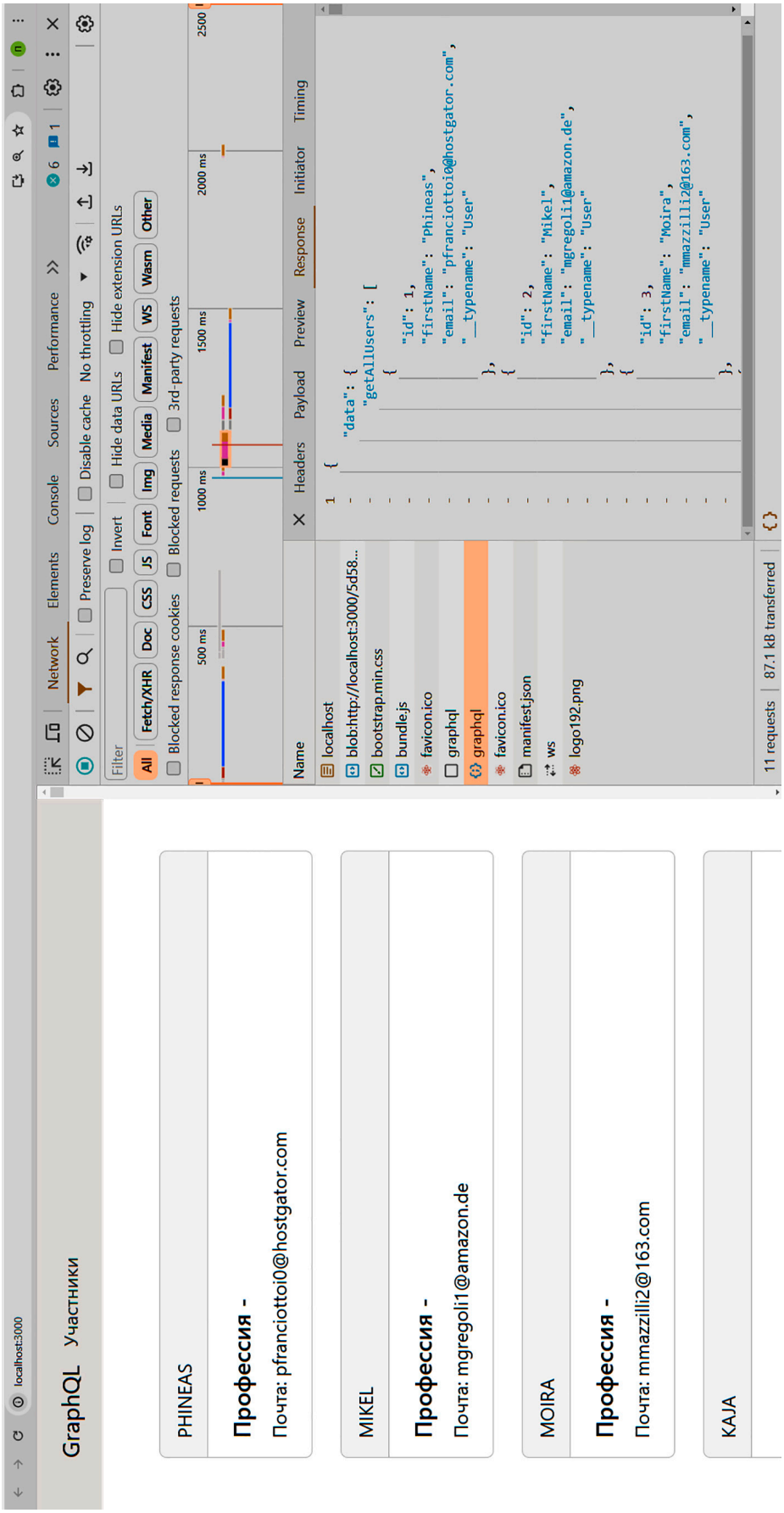

Рис. 2. Структура ответа на запрос GraphQL

В подходе REST безопасность часто реализуется с помощью токенов доступа (например, JWT) и стандартных методов аутентификации и авторизации, таких как OAuth 2.0. Осуществление контроля доступа к конечным точкам API и к данным может потребовать наличия различных ролей и разрешений.

Сложность же REST в безопасности может возникнуть из-за неоднородности методов аутентификации и авторизации, а также необходимости учета всех различных конечных точек и их соответствующих прав доступа.

GraphQL реализует безопасность через права доступа (permissions) и функции аутентификации/авторизации, определяемые на уровне схемы. Одним из преимуществ GraphQL в обеспечении безопасности является возможность четкого контроля над тем, какие данные разрешено запрашивать и модифицировать для каждого конкретного типа запроса. Данный критерий был рассмотрен выше, где описывалась схема данных.

Сложность GraphQL в аспекте безопасности может возникнуть из-за необходимости тщательно определять и управлять правами доступа к различным типам данных и запросам в самой схеме.

Для улучшения безопасности веб-приложения при использовании:

− REST, можно реализовать единый стандарт аутентификации/авторизации для всех конечных точек и аккуратно управлять правами доступа к ресурсам;

− GraphQL, можно использовать строгое управление правами доступа к данным на уровне схемы.

Масштабируемость проектов на REST API и GraphQL. Масштабируемость является важным критерием для выбора подхода при разработке веб-приложений. Масштабирование REST API может быть сложным на этапе разработки, поскольку управление множеством эндпоинтов и обработка различных типов запросов может потребовать дополнительных усилий [6].

Предположим, что в созданном веб-приложении на странице «Машины» необходимо получать пользователей, которые купили тот или иной автомобиль. Архитектура данного приложения построена на принципе изоляции модулей, то есть модули «Машины» и «Пользователи» являются независимыми и имеют свои уникальные api для получения данных. В данном примере разработчикам потребуется создать дополнительные конечные точки для доступа к списку пользователей. Так, если в проекте не используются различные механизмы кэширования данных, такой подход повлечет за собой излишние ресурсные затраты для практически одинаковых запросов.

GraphQL предлагает более гибкий и точно настраиваемый подход к получению данных. Он позволяет клиентам запрашивать только требуемые данные и связывать запросы на стороне сервера. За счет одной конечной точки и возможности делать гибкие запросы, масштабирование GraphQL может быть более простым в сравнении с REST API.

Изменим в веб-приложении, использующем GraphQL для доступа к данным, структуру запроса. Добавим в схему дополнительное поле *info*, которое будет включать в себя поля *city* и *country*. Теперь на клиентской части появилась возможность добавить в запрос новое поле info и выбирать, какие данные клиент должен получить. В данном случае будет запрашивать поле city.

Так, в одном запросе клиентская часть приложения смогла получить дополнительные данные, которые могут быть использованы в интерфейсе приложения. И для этого только изменили существующую схему и ввели дополнительное поле.

GraphQL запросы могут быть гибкими, но неправильное проектирование схемы GraphQL или запросы с глубокой вложенностью могут привести к избыточной нагрузке на сервер и нежелательным эффектам при масштабировании.

Механизмы кэширования в REST API и GraphQL. Кэширование – это важный аспект при разработке веб-приложений, поскольку позволяет улучшить скорость загрузки и снизить нагрузку на сервер. Различного рода манипуляции с данными, в том числе их кэширование, эффективнее всего проводить с помощью state-менеджеров, таких как Redux, Redux Toolkit, Effector, Zustand и др.

В экспериментальном веб-приложении был использован именно Zustand за счет своей простоты внедрения. В коде создается некое хранилище данных, в которое помещается ответ от api. В данном примере это массив users. В функции *getData ()* производится запрос users с использованием метода *get ()* и дальнейшая проверка *if(users.length) return*. Если в массиве уже есть данные пользователей, то попросту игнорируется последующий вызов метода api и клиент использует данные, которые были записаны в кэш.

Кэширование в GraphQL может быть реализовано на уровне запросов и ответов, что дает большую гибкость контроля над кэшированием. Если используется библиотека управления состоянием (Apollo Client), можно получить доступ к кэшированным данным и произвести анализ (рис. 3).

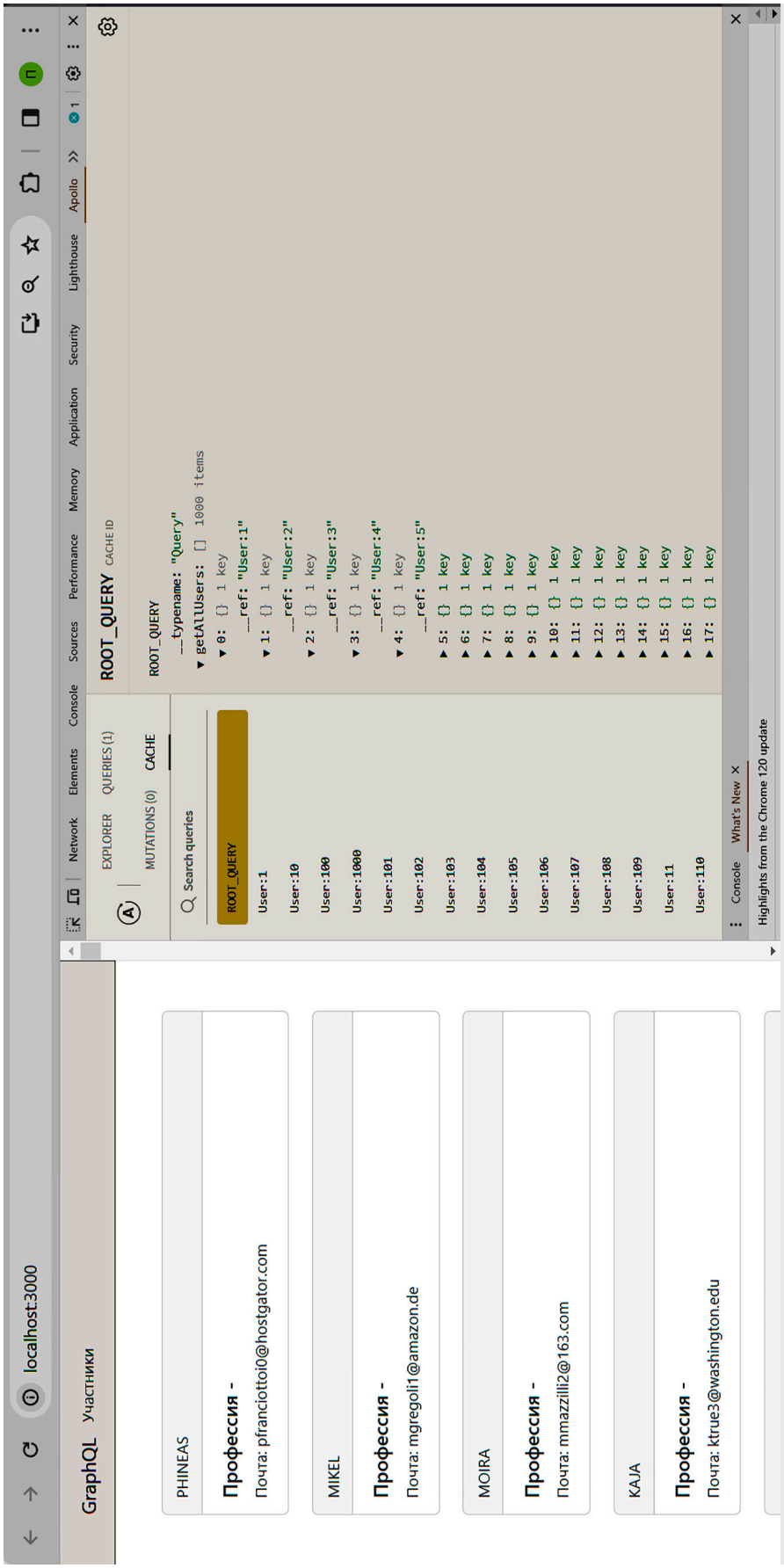

Рис. 3. Организация кэша в GraphQL

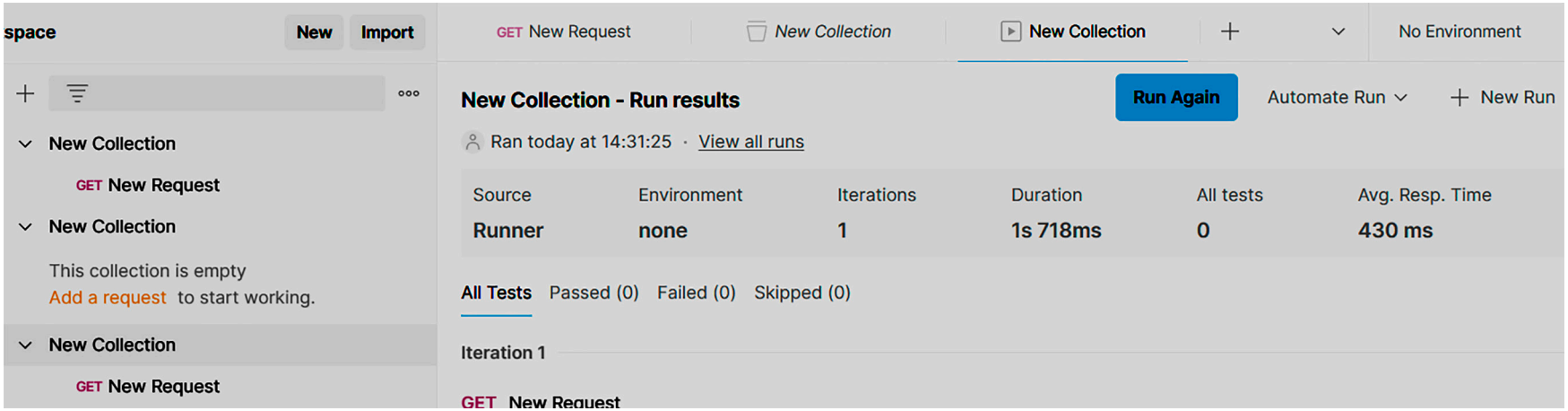

Рис. 4. Результаты нагрузочного тестирования REST-запросов

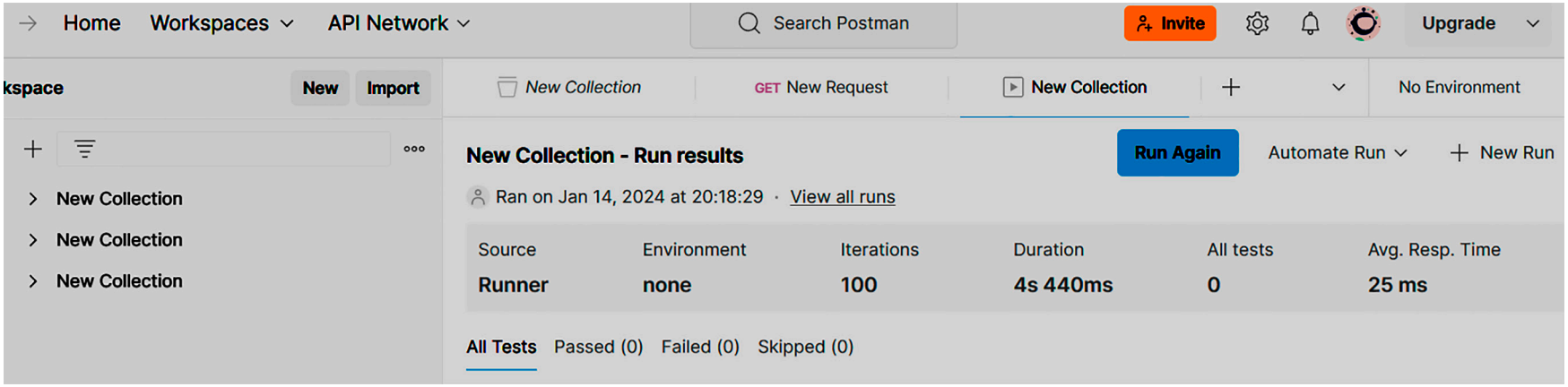

Рис. 5. Результаты нагрузочного тестирования GraphQL-запросов

Таким образом, использование REST API в контексте кэширования данных возможно, если имеются простые и статичные запросы, которые можно хорошо кэшировать на уровне HTTP заголовков. Использование GraphQL же возможно, если требуется более гибкое и точечное управление кэшированием на уровне запросов и ответов, особенно в случае сложных и изменчивых запросов.

Производительность (скорость) запросов REST и GraphQL. Когда рассматривается сравнительный анализ производительности между REST API и GraphQL, особенно при обработке большого количества данных, ситуация зависит от конкретных запросов, структуры данных, типов запросов и условий их использования. Важно учитывать, что в подходе REST каждый HTTP-запрос задействует клиентские ресурсы и приводит к передаче избыточного объема данных. Как результат – замедляется пользовательский интерфейс [7].

В созданных двух веб-приложениях анализ скорости загрузки будет производиться с использованием инструмента Postman как для REST-запросов, так и для GraphQL-запросов. В каждом приложении используется ровно 1000 элементов, которые запрашивает пользователь. Данный сравнительный анализ будет проводиться в контексте *нагрузочного тестирования*. Для простоты эксперимента каждый запрос будет вызываться по 100 раз подряд, а дальше проведен сравнительный анализ скорости отработки данных запросов.

Так, в результате проведения эксперимента были получены следующие данные: общее время выполнения 100 итераций запросов составило *1 мин 20 с*, при этом среднее время выполнения одного запроса составляет *776 мс*. Также результаты показали 0 ошибочных запросов (рис. 4).

GraphQL позволяет клиентам запрашивать только те данные, которые им нужны. Это может привести к передаче большого объема данных за один запрос и облегчить проблему n+1 запросов. Результаты тестирования GraphQL запроса представлены на рис. 5.

Как видно из рис. 5, общее время выполнения 100 итераций запросов составило 4 с 440 мс, при этом среднее время выполнения одного запроса составляет 25 мс. Также результаты показали 0 ошибочных запросов.

Таким образом, в результате проведения нагрузочного тестирования было установлено, что GraphQL в значительной степени выигрывает у подхода REST по скорости выполнения запросов.

Результаты исследования и их обсуждение

Представленный анализ на базе двух веб-приложений, одно из которых использует REST API, а другое – GraphQL, показал различные результаты.

По критерию «извлечение данных» REST API обычно использует конечные точки, которые возвращают фиксированные структуры данных, и клиент должен выполнить несколько запросов для извлечения связанных данных. С другой стороны, GraphQL позволяет клиентам запрашивать и получать именно те данные, которые им нужны, в рамках одного запроса, что делает его намного более эффективным и сокращает сетевой трафик [4].

По критерию «масштабируемость»: при выборе между REST API и GraphQL важно учитывать, что REST API может быть сложным и запутанным при масштабировании из-за управления большим количеством конечных точек, особенно в сложных приложениях. GraphQL обеспечивает более гибкий способ получения данных и может быть более простым в масштабировании за счет одной конечной точки, но требует аккуратного проектирования схемы и запросов для предотвращения избыточной нагрузки на сервер.

По возможности кэширования GraphQL демонстрирует наиболее гибкую систему управления кэшем. Такой инструмент, как Apollo DevTools, предоставляет дополнительную информацию о текущем состоянии кэша GraphQL, что позволяет производить анализ кэшированных данных и их использование прямо в браузере разработчика.

По критерию «производительность»: если имеются простые или независимые запросы, то подход REST является наиболее оптимальным. Если же есть сложные и/или связанные запросы на получение данных, когда требуется гибкое управление, то в данном случае лучше выбрать подход GraphQL.

При выборе между REST и GraphQL с точки зрения критерия безопасности, GraphQL может предложить более гибкий контроль над данными, что делает его предпочтительным.

Заключение

Комплексное исследование архитектурных стилей REST и GraphQL на основе экспериментального анализа позволило выявить и продемонстрировать ключевые преимущества и недостатки каждого из подходов. В зависимости от конкретных потребностей проекта, команды разработки могут с должной эффективностью использовать как

REST, так и GraphQL для достижения оптимальной производительности, расширяемости и соответствия требованиям клиентов.

При выборе между данными архитектурными подходами следует учитывать конкретные потребности проекта, такие как структура данных, клиентские требования, уникальные характеристики проекта и др.

Результаты данного исследования позволяют говорить о поступательном, динамическом развитии веб-технологий и, в частности, клиент-серверного взаимодействия. Развитие облачных технологий и распределенных систем приводит к созданию более сложных клиент-серверных моделей. Различные архитектурные подходы позволяют оптимизировать это взаимодействие, обеспечивая выбор между созданием гибких, масштабируемых API и управлением данными на клиентской стороне.

Список литературы

1. Тонкушин М.В., Гудков К.В. Сравнительный анализ технологий GraphQL и REST // Современные информационные технологии. 2019. № 29. С. 127–131.

2. Wittern Erik, Cha Alan, Laredo Jim A. Generating GraphQL-Wrappers for REST(-like) APIs // Lecture Notes in Computer Science. 2018. Vol. 10845. P. 65–83. DOI: 10.1007/978-3-319-91662-0_5.

3. Кожанов П.С. Сравнительный анализ подходов к организации клиент-серверного взаимодействия в современных React-приложениях (на примере REST API и GraphQL) // Технические и технологические системы: материалы XIV Международной научной конференции (Краснодар, 22–24 ноября 2023 г.). Краснодар: ООО ИД «Юг», 2023. С. 97–101.

4. Mikuła M., Dzieńkowski M. Comparison of REST and GraphQL web technology performance // Journal Computer Science Institute. 2020. P. 309–316. DOI: 10.35784/JCSI.2077.

5. Гридин В.Н., Анисимов В.И., Васильев С.А. Методы повышения производительности современных веб-приложений // Известия ЮФУ. Технические науки. 2020. № 4. С. 193-200. DOI: 10.18522/2311-3103-2020-2-193-200.

6. Hartina D.A., Lawi A., Enrico B.L. Panggabean. Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University / The 2nd East Indonesia Conference on Computer and Information Technology. 2018. [Электронный ресурс]. URL: https://clck.ru/32vGR3 (дата обращения: 15.03.2024).

7. Ананченко И.В., Чуриков Е.А. Оптимизация http-запросов с помощью перехода с REST API на GraphQL // Актуальные вопросы современной науки: сборник статей III Международной научно-практической конференции (Пенза, 25 сентября 2022 г.). Пенза: Наука и Просвещение (ИП Гуляев Г.Ю.), 2022. С. 11–14.