DOI 10.37882/2223-2966.2024.12-2.15

ОБЗОР СОВРЕМЕННЫХ ПОДХОДОВ К СОЗДАНИЮ API НА PYTHON

Нагорный Николай Николаевич
главный разработчик ПАО Росбанк, г. Москва
itdevelopernew@gmail.com

Аннотация. В статье рассмотрена проблематика разработки API для веб-приложений. Целью статьи является исследование современных подходов к созданию API при программировании на Python. Показано, что существует несколько видов API, которые используются в различных приложениях и сервисах. Например, RESTful API (Representational State Transfer) является одним из самых популярных типов API. Он использует стандартные HTTP — запросы как GET, POST, PUT и DELETE, для взаимодействия с приложениями и сервисами. Еще одним примером API является SOAP (Simple Object Access Protocol), который использует XML передачи данных. GraphQL описывается как «язык запросов для вашего API» и был построен в ответ на проблемы с производительностью при переходе Facebook на нативные мобильные приложения. Эта технология построения API была разработана как альтернатива REST. Несмотря на свое название, оно не только позволяет запрашивать данные с сервера, но и изменять их.

Ключевые слова: Application Programming Interface, я веб-приложения, разработка, обзор, Python, RESTful, GraphQL API.

OVERVIEW OF MODERN APPROACHES TO CREATING APIS IN PYTHON

N. Nagorny

Summary. The article considers the problems of developing APIs for web applications. The purpose of the article is to study modern approaches to creating APIs when programming in Python. It is shown that there are several types of APIs that are used in various applications and services. For example, RESTful API (Representational State Transfer) is one of the most popular types of APIs. It uses standard HTTP requests such as GET, POST, PUT and DELETE to interact with applications and services. Another example of API is SOAP (Simple Object Access Protocol), which uses XML data transfer. GraphQL is described as «a query language for your API» and was built in response to performance issues when Facebook switched to native mobile applications. This API building technology was developed as an alternative to REST. Despite its name, it not only allows you to request data from the server, but also to change it.

Keywords: Application Programming Interface, I web applications, development, review, Python, RESTful, GraphQL API.

Рост интернет-вычислений привел к росту вычислительных центров, в результате чего корпоративные центры обработки данных, как правило, состоят из большого количества серверов, которые не используются в полной мере [1]. Это, в свою очередь, стало причиной увеличения стоимости обслуживания соответствующих сервисов, что обусловлено прямой зависимостью между количеством серверов и расходами на их аренду, электроэнергию, лицензии на программное обеспечение и администрирование [2].

Серверный пул представляет собой группу серверов в кластере, которые объединены в некоторую логическую единицу (пул). Особенностью информационных систем с клиент-серверной архитектурой, функционирующих в современных сетях, является генерация большого количества сообщений, передаваемых сетями. Произвольный запрос пользователя на выполнение конкретной задачи веб-сервисом может вызвать создание и отправку десятков запросов и ответов другим веб-сервисам.

Одним из подходов к повышению их производительности является увеличение количества веб-серверов, обрабатывающих запросы, сгенерированные пользователями, то есть клиентами, к этому веб-сервису. [3] Клиент должен видеть веб-сервис, к которому обращается как единое целое. Для этого на нескольких веб-серверах устанавливается идентичное программное обеспечение, а маршрутизатор посылает запрос от клиента к одному из серверов, выбираемым по определенным признакам. Распределение запросов между веб-серверами помогает избежать ситуации, когда поток запросов вызывает такую нагрузку на отдельный сервер, что возможная скорость обработки запросов и существующие системные ресурсы потребуют построения очереди [4].

Для решения этих проблем с соблюдением необходимых требований может быть введено использование технологии Serverless для разработки веб-сервера. Ключевые характеристики данной инфраструктуры обеспечивают приложение неограниченным количеством вычисляемых ресурсов, привлекаемых только тогда, когда они нужны.

Термин API используется в различных контекстах в области инженерии программного обеспечения. API (Application Programming Interface) — это набор правил, протоколов и инструментов, позволяющих разным программам взаимодействовать друг с другом. Благодаря API программисты могут создавать более сложные и функциональные программы, используя уже существующие сервисы и программы [5].

API касается интерфейса программного элемента, который можно вызвать или выполнить. Указанные программные элементы появляются на разных уровнях абстракции программной системы. В данном исследовании речь идет только о тех элементах, которые появляются на самом высоком уровне: компоненты и каналы коммуникации.

API является одной из важнейших технологий в мире программирования, поскольку позволяет интегрировать различные компоненты программы и повышать ее эффективность. Без использования API программистам пришлось бы разрабатывать все компоненты программы с нуля, что занимает уйму времени и усилий.

Основными принципами работы API являются простота и надежность. API должно быть легко понятно и просто в использовании, чтобы разработчики могли быстро и без проблем интегрировать его в свои программы. Также API должно быть надежным и стабильным, чтобы приложения, которые его используют, могли работать без сбоев и ошибок.

Есть несколько видов API, которые используются в различных приложениях и сервисах. Например, RESTful API (Representational State Transfer) является одним из самых популярных типов API. Он использует стандартные HTTP — запросы как GET, POST, PUT и DELETE, для взаимодействия с приложениями и сервисами. Еще одним примером API является SOAP (Simple Object Access Protocol), который использует XML передачи данных [6].

Использование API имеет как преимущества, так и недостатки. Среди главных преимуществ можно выделить [7–10]:
- быстрая разработка: использование готовых API может существенно ускорить процесс разработки приложений и сервисов;
- экономия ресурсов: разработчики могут использовать готовые API вместо создания собственных функций и сервисов, что позволяет экономить ресурсы и время;
- улучшенная функциональность: использование API позволяет расширять функциональность приложений и сервисов, добавляя новые возможности и функции.

Среди недостатков использования API можно выделить:
- ограниченность: использование API может быть ограничено определенными правилами и ограничениями, установленными разработчиками приложений и сервисов;
- зависимость от сторонних сервисов: если приложение или сервис зависят от стороннего API, любые изменения могут повлиять на работу приложения или сервиса;
- безопасность: использование API может повысить уязвимость программ и сервисов к кибератакам, если не принять соответствующие меры защиты.

В целом, несмотря на некоторые недостатки, использование API является важным инструментом для разработчиков программ и сервисов. Он позволяет ускорить процесс разработки, расширить функциональность и улучшить пользовательский опыт.

API является важным инструментом для разработки приложений и сервисов, позволяющих ускорить процесс разработки, расширить функциональность и улучшить опыт пользователя. При использовании API необходимо учитывать как его преимущества, так и недостатки и принимать соответствующие решения.

Обзор инструментов для работы с API:
- Postman: это популярный инструмент для тестирования и документирования API. Он позволяет легко посылать запросы и получать ответы, а также создавать документацию;
- Insomnia: это другой популярный инструмент для тестирования и отладки. Он предоставляет широкий набор функций для работы с API, включая поддержку GraphQL и автоматическую генерацию кода;
- Swagger: используется для разработки и документирования API. Это позволяет создавать спецификации в формате OpenAPI и автоматически генерировать документацию и код для клиентских библиотек.

А теперь рассмотрим, где используется API и увидим примеры практики проектирования и документирования API популярных программных интерфейсов:
- социальные сети API позволяют разработчикам создавать программы для социальных сетей и использовать данные из социальных сетей для различных целей, например, для анализа поведения пользователей;
- электронная коммерция: многие электронные магазины используют API для интеграции со сторонними сервисами, такими как платежные системы или логистические компании;
- мобильные приложения: API используются для взаимодействия между мобильными приложениями и внешними сервисами, такими как базы данных или веб-сервисы.

API-интерфейсы являются одним из основных механизмов взаимодействия программных компонентов. Они служат связующим элементом современных мобильных, десктопных и веб-приложений, возможности которых нашли применение в таких сферах деятельности, как розничная торговля, транспорт, социальные сети, экономика, бухгалтерия и др.

Примеры практики проектирования и документирования API и их возможностей:

- Google Maps API: позволяет использовать картографические данные, такие как расположение, маршруты и географические объекты, в приложениях и на сайтах;
- Twitter API: предоставляет доступ к данным Twitter, таким как твиты, пользователи и хэштеги, для создания инструментов аналитики, мониторинга и управления аккаунтом;
- Facebook API: позволяет получать доступ к данным Facebook, таким как профили пользователей, страницы, комментарии и сообщения, для создания приложений и инструментов для управления аккаунтом.

До 2015 г. задача создания хорошего API почти всегда приводила к тому, что разработчики выбирали REST независимо от фактического направления использования. Fielding определяет REST [5] как архитектурный стиль вокруг модели мышления, используемой при создании стандартов, описывающих Интернет: протокол передачи гипертекста (HTTP), унифицированный идентификатор ресурса (URI) и язык разметки HyperText (HTML) [6]. В 2015 году Facebook открыл спецификацию технологии, называемой GraphQL. Это подход, определяющий декларативный язык запросов для получения данных по API. Он показал, что есть случаи, когда для удовлетворения требований системы необходимо нечто иное, чем HTTP. Наталкиваясь на два подхода к созданию API, возникает вопрос, какой выбрать для конкретного случая? Внедрение систем стоит дорого. Поэтому очень важно заранее правильно определить какой подход выбрать для конкретной задачи.

Основными компонентами архитектуры REST API являются ресурсы и Uniform Resource Identifiers (URI), HTTP методы, форматы данных и Hypermedia as the Engine of Application State (HATEOAS) — рисунок 1.

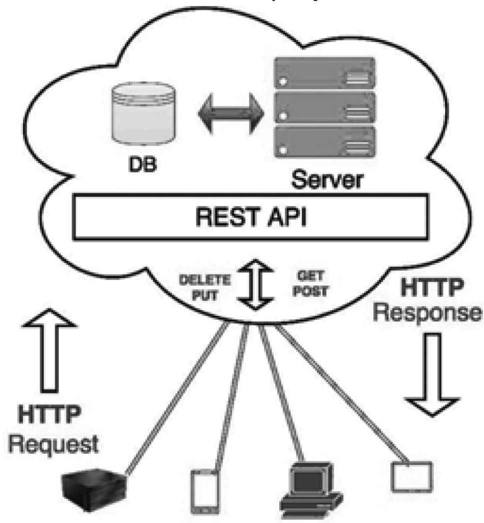

Рис. 1. Архитектура REST API

Комбинация этих компонентов обеспечивает гибкость, масштабируемость и простоту использования REST API. REST API позволяет разрабатывать клиентов и серверы независимо друг от друга, обеспечивая легкую интеграцию и эффективный обмен данными между ними.

Одной из задач исследования является сравнение между технологией GraphQL и принципами, и ограничениями, указанными в архитектурном стиле REST, с целью оценки их ключевых различий. На основе данного анализа осуществлено использование GraphQL для построения приложения прототипа.

Итак, разработка архитектуры веб-сервера требует значительных усилий, направленных на обеспечение гибкой инфраструктуры среды развертывания сервера. Кроме того, необходимо учитывать, что значительная часть выделенных и приобретенных вычислительных ресурсов не используется в течение большей части времени существования сервера. Указанные причины обуславливают необходимость исследования возможности применения бессерверных вычислений (Serverless) и языка запросов GraphQL для создания веб-сервера.

GraphQL Server можно развернуть с помощью любого из трех алгоритмов, перечисленных ниже:

- GraphQL Server с подключенной базой данных;
- GraphQL Server, который интегрирует существующие системы;
- гибридный подход.

Опишем алгоритм GraphQL Server с подключенной базой данных. Эта архитектура имеет GraphQL Server с интегрированной базой данных и часто может использоваться с новыми проектами — рис. 2. При получении запроса сервер считывает полезную нагрузку запроса и извлекает данные из базы данных. Это называется разрешением запроса. Ответ, возвращаемый клиенту, соответствует формату, указанному в официальной спецификации GraphQL.

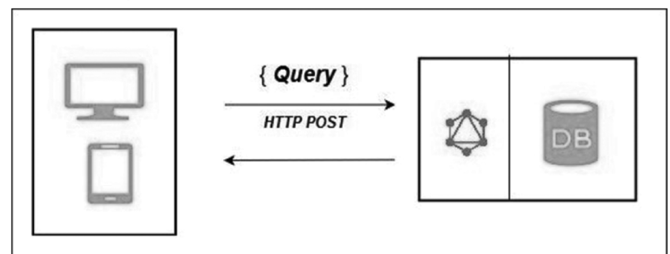

Рис. 2. Алгоритм GraphQL Server с подключенной базой данных

На приведенной выше схеме сервер GraphQL и база данных интегрированы на одном узле. Клиент (настольный/мобильный) взаимодействует с GraphQL Server по протоколу HTTP. Сервер обрабатывает запрос, извлекает данные из базы данных и возвращает их клиенту.

Алгоритм GraphQL Server, который интегрирует существующие системы, полезен для компаний с устаревшей инфраструктурой и различными API. GraphQL можно использовать для объединения микросервисов, устаревшей инфраструктуры и сторонних API в существующей системе — рис. 3.

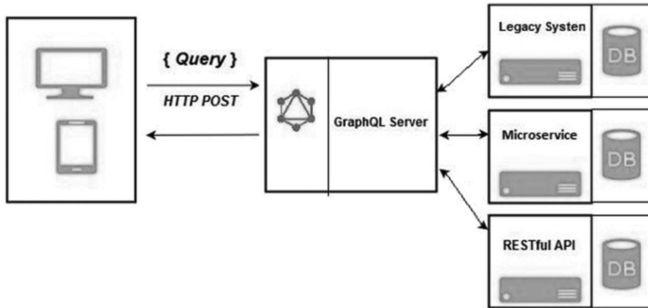

Рис. 3. Интеграция существующих систем сервером GraphQL

На приведенной выше схеме API GraphQL действует как интерфейс между клиентом и существующими системами. Клиентские приложения взаимодействуют с сервером GraphQL, который, в свою очередь, разрешает запрос.

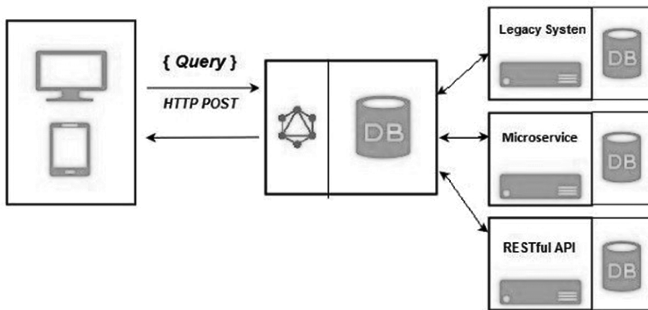

Рис. 4. Гибридный подход GraphQL

Гибридный подход в алгоритме GraphQL можем объединить два вышеуказанных подхода и создать сервер GraphQL. В этой архитектуре сервер GraphQL будет разрешать любой полученный запрос. Он будет либо извлекать данные из подключенной базы данных, либо из интегрированных API. Это представлено на рисунке 6.

Основой для GraphQL является схема. Она описывает существующие типы и их отношения, а также точки входа клиентов — запросы и мутации. Рисунок 5 показывает примерную схему GraphQL. В корне Schema определяется элемент схемы, который делится на запрос и мутацию. Конкретный тип User имеет ряд полей. Восклицательный знак указывает на урегулированность, то есть поле не может быть нулевым. Квадратные скобки вокруг определения типа указывают на массив данного типа. Параметры, как в запросах, так и в мутациях, названы, поэтому их можно указать в любом порядке.

Для получения данных с сервера GraphQL посылается запрос, код на рис. 2 показывает такой запрос, в то время как Рисунок 6 иллюстрирует образец результата.

```
schema {
  query: Query
  mutation: Mutation
}

type User {
  id: ID!
  nickName: String
  posts: [Post]!
  followees: [User]!
  followers: [User]!
}

type Post {
  id: ID!
  author: User
  content: String
  # The ISO representation of the date when the post was created.
  createdAt: String
  replies: [Post]!
  replyTo: Post
}

type Query {
  # Retrieve the timeline of a user identified by their nickname.
  timeline(of: String): [Post]!
  user(nickName: String): User
  users: [User]!
}

type Mutation {
  writePost(authorNick: String, content: String, replyTo: String = null): Post
  newUser(nickName: String): User
  followUser(me: String, other: String): User
}
```

Рис. 5. Определение схемы GraphQL, иллюстрирующей определение типов и их отношений

```
query {
  timeline(of: "MaxMustermann") {
    ...basicPostFields
    replies {
      ...basicPostFields
    }
  }
}

fragment basicPostFields on Post {
  content
  author {
    nickName
  }
}
```

Рис. 6. Запрос GraphQL по схеме, определенной на рисунке 5

```
{
  "data": {
    "timeline": [
      {
        "content": "My first blogpost!",
        "author": {
          "nickName": "JohnDoe"
        },
        "replies": [
          {
            "content": "Super cool!",
            "author": {
              "nickName": "MaxMustermann"
            }
          }
        ]
      }
    ]
  }
}
```

Рис. 7. Пример результата выполнения запроса, приведенного на рисунке 6

Он получает содержимое и фамилию автора всех публикаций на временной шкале пользователя MaxMustermann. Обратите внимание, как запрос начинается с одного из полей типа Query: временная шкала. В схеме определяется, что запрос «временная шкала» принимает единый параметр с именем и возвращает массив сообщений.

Рисунок 6 также иллюстрирует использование фрагментов — используя набор полей с описанием желаемых полей каждой публикации на шкале 8, чтобы получить эти поля для каждого сообщения. Применение фрагмента аналогично оператору Spread в ECMAScript 2015 г. [9]. При выдаче запроса GraphQL требует, чтобы все запрашиваемые поля запроса были примитивными типами для улучшения предсказуемости [6].

Транспортный протокол GraphQL сам по себе является лишь спецификацией для «среды выполнения запросов» [8]. Поэтому она не касается транспортного протокола, используемого между клиентом и сервером для передачи этих запросов и мутаций. Поскольку GraphQL передает всю необходимую информацию в самом теле запроса, он работает с другими транспортными протоколами, такими как TCP или UDP. При использовании с HTTP обычно создается единственная конечная точка, принимающая клиентские запросы POST, передавая полезную нагрузку в теле HTTP.

Однако для HTTP API в целом детализация не обязательно определяется методом HTTP. Системы, использующие HTTP исключительно для туннельных целей, могут определять интерфейс, позволяющий вызвать несколько операций в одном запросе HTTP, таким образом, поддерживая операции массового использования [4]. Следовательно, вызов операции всегда приводит к взаимодействию двух компонентов, но несколько операций можно объединить в одно взаимодействие. Взаимодействия описывают необходимую коммуникацию в распределенных системах между компонентами только в том случае, если они хотят сотрудничать в достижении определенных целей. Важно учесть, что возможность вызвать несколько операций в рамках одного взаимодействия полностью зависит от конструкции системы. Например, при работе с HTTP-протоколом на уровне приложений разработчики, как правило, стараются однозначно сочетать операции и взаимодействия, поскольку HTTP достаточно богат, чтобы описать семантику операции на уровне протокола. GraphQL, с другой стороны, позволяет выполнять несколько мутаций в одном взаимодействии. Наибольшим следствием использования GraphQL для проектирования API является перенос ряда обязанностей с сервера на клиента.

Во-первых, клиент отвечает за разработку рабочего процесса, который должен быть представлен в приложении — клиент отвечает за реализацию фактических правил, определяющих, какой из доступных путей является действительным. Преимущественно из-за отсутствия метаданных в ответах с сервера, клиент должен самостоятельно разобраться, какие шаги или последовательности операций действительны.

Во-вторых, если клиент использует библиотеку, которая добавляет кэш, он также должен управлять его инвалидацией, задачей, которую не следует недооценивать, когда приложение увеличивается в размерах. В зависимости от проблемной области и требований программы влияние этих последствий различно. Иногда кэши не нужны или их использование даже не целесообразно из-за требований предоставления данных в режиме реального времени.

ЛИТЕРАТУРА

1. Аксютина Е.М., Белов Ю.С. Обзор архитектур и методов машинного обучения для анализа больших данных // Электронный журнал: наука, техника и образование. 2016. №1 (5). С. 132–139.
2. Двуреченский И.О., Симонов И.Н., Гаев Л.В. ВЕБ-ПРИЛОЖЕНИЯ: ОСНОВЫ, ТЕХНОЛОГИИ И РАЗРАБОТКА // Инновационная наука. 2023. №6-1. URL: https://cyberleninka.ru/article/n/veb-prilozheniya-osnovy-tehnologii-i-razrabotka (дата обращения: 17.04.2024).
3. Ковалев В.В., Храмков В.В., Семенова Е.И. Актуальность использования архитектуры REST для обмена данными между клиент-серверными приложениями. В: Сборник материалов V Международной научно-практической конференции, посвященной Дню космонавтики «Актуальные проблемы авиации и космонавтики». В 3-х томах. Том 2. Красноярск; 2019. С. 339–340.
4. Нестеренко В.Р., Маслова М.А. Современные вызовы и угрозы информационной безопасности публичных облачных решений и способы работы с ними. Научный результат. Информационные технологии. 2021;6(1):48–54.
5. Путято М.М., Макарян А.С., Лещенко В.В., Немчинова В.О. АНАЛИЗ ТИПОВЫХ УЯЗВИМОСТЕЙ ПРИ ПОСТРОЕНИИ ВЕБ-ПРИЛОЖЕНИЙ // Вестник Адыгейского государственного университета. Серия 4: Естественно-математические и технические науки. 2022. №3 (306). URL: https://cyberleninka.ru/article/n/analiz-tipovyh-uyazvimostey-pri-postroenii-veb-prilozheniy (дата обращения: 17.04.2024).
6. Садовая Е.Н. СОВРЕМЕННЫЕ ВЫЗОВЫ И УГРОЗЫ ИНФОРМАЦИОННОЙ БЕЗОПАСНОСТИ REST API И СПОСОБЫ ИХ ПРЕДОТВРАЩЕНИЯ // Молодой исследователь Дона. 2023. №3 (42). URL: https://cyberleninka.ru/article/n/sovremennye-vyzovy-i-ugrozy-informatsionnoy-bezopasnosti-rest-api-i-sposoby-ih-predotvrascheniya (дата обращения: 17.04.2024).
7. Baker O., Nguyen Q. A Novel Approach to Secure Microservice Architecture from OWASP vulnerabilities. In: Proceedings of the 10th Annual CITRENZ Conference. Wellington; 2019. p. 56–61.
8. Fielding R. Architectural Styles and the Design of Network-based Software Architectures // UNIVERSITY OF CALIFORNIA, IRVINE. 2000.
9. OWASP API Security Project. Top 10. OWASP. URL: https://github.com/OWASP/API-Security/blob/master/2019/en/dist/owasp-api-security-top-10.pdf (дата обращения 17.04.2024).
10. 2022 CWE Top 25 Most Dangerous Software Weaknesses. Common Weakness Enumeration. URL: https://cwe.mitre.org/top25/archive/2022/2022 cwe top25.html (дата обращения 17.04.2024).

© Нагорный Николай Николаевич (itdevelopernew@gmail.com)