Geekbrains

**Разработка сервиса обработки запросов для интернет-магазина**

**c использованием GraphQL**

Программа: Разработчик

Специализация: Веб-разработка на Java

ФИО: Поляков Илья Михайлович

Москва

2024

[Содержание](#_heading=h.7mt9x4b37ce)

[Введение](#_heading=h.1fob9te)

[Теоретическая и практическая главы](#_heading=h.hst8x8f1r8tz)

[Заключение](#_heading=h.dhwmmf3wq1kk)

[Список используемой литературы](#_heading=h.m8jvbwhv7ujg)

[Приложения](#_heading=h.s6k7t2lwcp0k)

# Введение

API[[1]](#footnote-1) для доступа к данным используется повсеместно. Наиболее распространенным примером являются приложения с клиент-серверной архитектурой. В данном случае сервер предоставляет клиенту доступ к данным, за обработку и хранение которых он отвечает. Из этого следует, что вопрос проектирования API крайне актуален, так как количество приложений, использующих клиент-серверную архитектуру, настолько велико, что невозможно достаточно точно его оценить.

Одной из распространённых проблем при запросе данных у сервера это представление клиенту избыточных данных (over-fetching). Например, он получает поля, которые не использует. Другая наоборот - представление не всех необходимых данных (under-fetching). Например, в первом запросе пользователь узнает Id связанной сущности, а далее по Id получает ее.

Для решения поставленных проблем предложено API. Ключевой идеей предложенного решения, является использование технологии GraphQL. GraphQL даст возможность получать только те поля сущности, которые нужны пользователю в конкретном запросе, что позволит избежать проблемы over/under-fetching при запросах.

**Тема проекта:** Разработка сервиса обработки запросов для интернет-магазина c использованием GraphQL.

**Цель:** Изучить особенности, преимущества и недостатки работы с технологией GraphQL, а также выработать рекомендации по её использованию в целях улучшения процесса разработки приложений.

**Какие проблемы решает:**

1. Результат работы может быть полезен при проектировании API приложений как один из возможных способов предоставления доступа к данным.

**Задачи:**

1. Изучить литературу, касающуюся технологии GraphQL.

2. Изучить существующие подходы к проектированию API, предоставляющего доступ к данным.

3. Выполнить проектирование серверной части системы, выбрать технологии, которые станут использоваться в дальнейшем.

4. Разработать серверную часть системы.

**Инструменты:** Postman, Maven, Hibernate, Spring boot, JPA.

**Состав команды**: Поляков Илья Михайлович (Разработчик).

# Теоретическая и практическая главы

**Глава 1. Клиент-серверная архитектура.**

Приложениями и сайтами одновременно могут пользоваться сотни и даже миллионы человек. Все они обращаются к одному компьютеру, который должен уметь обрабатывать запросы и присылать ответы. Такой подход называется клиент-серверной архитектурой. Она описывает, как происходит работа с пользователями, где хранятся данные и как обеспечивается их защита.

**1.1 Клиент-серверная архитектура и ее компоненты**

В клиент-серверной архитектуре используется три компонента:

Клиент - программа, которую мы используем в интернете. Чаще всего это браузер, но может быть и другая отдельная программа

Сервер - компьютер, на котором хранится сайт или приложение. Когда мы заходим на сайт магазина, мы обращаемся к серверу, на котором находится сайт.

База данных - программа, в которой хранятся все данные приложения. Для магазина это будет база клиентов, товаров и заказов.

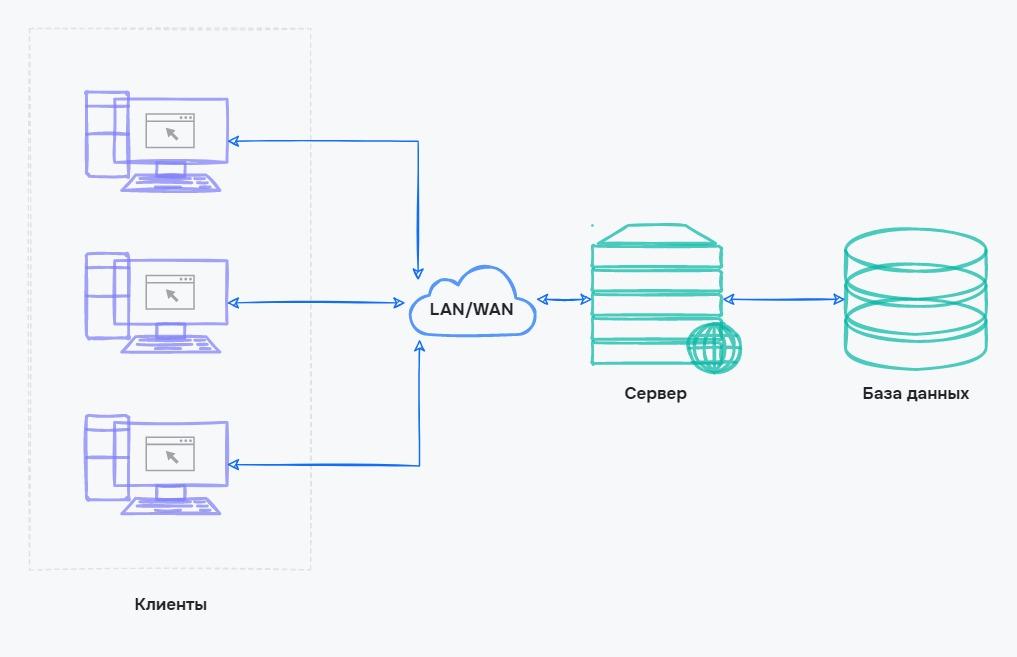


Рисунок 1 Клиент-серверная архитектура и ее компоненты.

**1.2 Особенности клиента**

Клиент - это всегда программа. Ее назначение - дать пользователю удобный способ взаимодействия с сервером.

С точки зрения сервера, выбрать товар - это послать запрос на специальном языке запросов, например GraphQL, как в нашем случае. Но для простого пользователя это сложно. Поэтому клиент в виде браузера дает удобный способ взаимодействия, чтобы не писать код своими руками.

**1.3 Особенности сервера**

Сервер - выделенный или специализированный компьютер для выполнения сервисного программного обеспечения. Основная задача сервера - бесперебойная работа и возможность обрабатывать миллионы запросов от пользователей.

Если сервер выполняет функции приложения и базы данных, то такая архитектура называется **двухуровневой**. Такой подход используют для небольших приложений, где нет большого количества клиентов. Хоть такой способ и проще, но его надежность небольшая. Если сервер взломают, то злоумышленники получат все данные.

Чтобы решить проблему безопасности в клиент-серверной архитектуре, используют базу данных. Она хранится отдельно от сервера. Сервер в этом случае выполняет роль логической машины, которая обрабатывает данные, но не хранит их. Данная модель будет использована в данной работе в рамках реализации интернет-магазина.

**1.4 Особенности клиент-серверного взаимодействия**

Зачастую, взаимодействие между клиентом и сервером происходит по одному из специальных протоколов[[2]](#footnote-2): IP, HTTP, WebSocket и т. д. Более того, инициатором данного взаимодействия в большинстве случаев является именно клиент.

Если абстрагироваться от частных случаев и особенностей протоколов взаимодействия, то можно сказать, что клиент серверное взаимодействие происходит по следующей схеме:

1) клиент отправляет серверу некоторый запрос;

2) сервер выполняет какие-то действия на основе этого запроса;

3) сервер отправляет ответ.

В данном случае, можно сказать, что сервер предоставляет некоторый API для запросов клиента. К этому API предъявляется множество требований разной направленности.

Во-первых, к нему предъявляются функциональные требования. Это означает, что API должен предоставлять полный функционал системы, а именно позволять сделать с помощью запросов все обусловленные на этапе проектирования действия.

Также, API должен быть производительным. Это значит, что он не должен перегружать сеть взаимодействия, не зависимо от протокола, а также увеличивать вычислительную сложность на обработку ответов. Чтобы объяснить, что это значит, давайте рассмотрим следующий пример: сервер возвращает клиенту граф. Естественно, граф можно представить в разном виде: список смежностей, матрица смежностей, список ребер и т.д. Но если клиенту нужен граф в виде списка смежностей, а сервер возвращает его как матрицу смежностей, то клиенту необходимо дополнительно преобразовать его, что является повышением вычислительной сложности.

Следует отметить, что серверные API делятся на два вида: публичные и приватные. Публичным API называется API, который находится в открытом доступе и каждый может им воспользоваться. Такие API предоставляют крупные компании для работы с их данными или приложениями. К приватному же API зачастую имеет доступ только несколько серверов frontend и он создается специально для них в рамках некоторого одного проекта.

Последнее требование к API, которое следует обсудить – это его удобство. Под этим подразумевается то, что пользователи API должны без особых проблем использовать его после небольшой инструкции, или же полностью интуитивно.

**1.5 Анализ существующих подходов к проектированию API**

Для рассмотрения были выбраны следующие подходы: REST-API и GraphQL со статической схемой данных.

Данные подходы будут рассмотрены исходя из следующих критериев:

1) возможность избежать over-fetching;

2) возможность избежать under-fetching;

3) структура кода.

Начнем мы с одного из старейших подходов в вебе - REST, который использует все возможности HTTP[[3]](#footnote-3).

REST - Representational state transfer - архитектурный подход, который описывает набор правил, как организовать общение систем. У данного подхода есть ряд требований - отсутствие состояния, единообразие интерфейсов. В простом понимании это URI, который описывает, что за ресурс мы запрашиваем: список элементов или один элемент. Кроме того, используются HTTP-глаголы или HTTP-методы - какие действия мы произведем с ресурсом.

Методов достаточно много. Как правило, мы имеем дело с:

POST /users для создания нового пользователя;

GET /users для получения списка пользователей;

GET /users/1 для получения одного пользователя с ID=1;

PUT или PATCH /users/1 для изменения пользователя;

DELETE /users/1 для удаления.

В ответе от сервера мы получаем запрошенный ресурс / сообщение об ошибке и трехзначный код ответа. Код ответа показывает, что произошло с запросом пользователя и что, в случае ошибки, пошло не по плану. К примеру:

200 – запрос выполнен успешно;

401 – необходимо авторизоваться;

404 – ресурс не найден;

505 – сервер временно недоступен.

Такой подход предполагает наличие некоторого количества конечных точек серверного API, через которые и происходит взаимодействие с клиентом. Зачастую, REST-API подразумевает реализацию CRUD-интерфейса, используя перечисленный выше протокол. В расширенном REST-API обычно добавляются вспомогательные поля для GET-запросов, такие как фильтр, пейджинг, сортировка и т.д. Стоит отметить, что добавление этих полей позволяет частично решить проблему over-fetching, т.к. клиент может запрашивать только те сущности, которые ему нужны, но при этом не позволяет выбирать поля, нужные клиенту, отправляя вместо этого все. Несмотря на то, что кажется, что этот подход достаточно строгий и структурированный, на самом деле ничего не мешает использовать эти методы и коды не по назначению, не структурировать код.

GraphQL – это язык запросов, который начал использоваться внутри одной крупной компании в 2012 году, в 2015 был представлен сообществу разработчиков, а с 2018 года находится под крылом Linux Foundation.

В отличие от REST API, в котором точка входа /users возвращает пользователей с заранее оговоренным набором полей, GraphQL позволяет на стороне клиента определить набор данных, которые ему нужны. Например, запрос user { name age } вернет только имя и возраст пользователя.

В рамках GraphQL появляются несколько новых терминов, которых не было при работе с REST API:

мутация (mutation) - запросы на изменение и добавление данных;

распознаватель (resolver) - функция обращения к источнику данных, например, к базе;

схема (schema) - типизированное описание источника данных или, по-другому, структура ответа;

подписка (subscription) - способ автоматического обновления данных с сервера.

GraphQL со статической схемой позволяет выполнять запросы только на те поля, которые выбирает сам пользователь. Также он поддерживает связи между сущностями в рамках одного запроса. Это позволяет решить проблему over и under fetching. Также фреймворки[[4]](#footnote-4) GraphQL и способ взаимодействия с конечной точкой достаточно строги и имеют четкую структуру, что повышает maintainability[[5]](#footnote-5) и надежность.

**Глава 2. Проектирование архитектуры и структуры приложения.**

**2.1 База данных**

# Заключение

В заключение необходимо включить следующее:

1. Краткие и ёмкие теоретические и практические выводы, которые были получены во время анализа теоретической базы и практического исследования.
2. Оценка проведённого исследования, описание его результатов.
3. Практическая значимость работы, рекомендации и планы на дальнейшие исследования.
4. Общий итог - достижение цели, выполнение задач, доказательство гипотезы.
5. Предложения по совершенствованию объекта исследования.

Таким образом, REST-подход в большинстве случаев закрывает все потребности при разработке приложений, если знать его функционал и им пользоваться. И GraphQL и REST могут соперничать только в вопросах публичного API. Так что научитесь грамотно выстраивать приложение, не гонитесь за хайпом и с умом подходите к вопросу выбора инструментов.

Как определить, в каком случае лучше использовать REST, а когда нам нужен GraphQL?

Если вы пишете фронт-приложение, которое использует API – GraphQL избыточен. Проще и лаконичнее договориться о формате запроса и ответа с бэкэнд-стороной.

Если вы пишете публичный API, но знаете, что понадобится пользователю и как избавить его от N+1 запросов – выбирайте, что душе угодно, но REST выйдет дешевле.

Но если вы пишете публичный API и хотите дать пользователю полную свободу в рамках системы – в этом случае действительно лучше подойдет GraphQL.

# Список используемой литературы

Здесь нужно будет указатьсписок используемой литературы, ссылки на все ресурсы, которые нужны были для создания проектной работы.

Основные правила оформления использованной литературы и ресурсов:

1. Каждый источник упоминается единожды, независимо от того, насколько часто на него ссылаются.
2. Список литературы оформляется в алфавитном порядке по фамилии автора, сначала русскоязычная литература, затем иностранная, далее интернет-сайты.
3. Библиографическая запись обязательно включает:
   * Фамилию автора или фамилии их группы, инициалы (при наличии).
   * Название статьи, книги, справочника, закона, иного документа.
   * Населённый пункт, в котором был издан источник, наименование издательства.
   * Год публикации.
   * Число страниц.

## Пример

* *Книга: Автор. Название книги. Город: Издательство, Год.*
* *Статья: Автор. "Заголовок статьи." Название журнала Том, номер (Год): страницы.*
* *Статья: ru.wikipedia.org. «Сервер (аппаратное обеспечение)».*
* *Статья: ru.hexlet.io «Клиент-серверная архитектура».*
* *Доклад Владимира Цукуры - GraphQL - API по-новому: «https://www.youtube.com/watch?v=YgRmgHPTXr4».*
* *Документация: docs.spring.io «Spring for GraphQL».*

# Приложения

В **приложения** обычно входят артефакты, получившиеся в процессе создания проекта:

1. Объёмные графики и таблицы, которые не помещаются на лист А4.
2. Длинные математические формулы и расчёты по ним.
3. Характеристики аппаратуры, которая использовалась для проведения исследования.
4. Авторские методики.
5. Вспомогательный материал: тесты, карточки, схемы, рисунки.
6. Материалы, полученные на предприятии: отчёты, прочие документы.

1. API (application programming interface) - контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это». [↑](#footnote-ref-1)
2. Протокол - набор правил, по которым передаются данные. [↑](#footnote-ref-2)
3. HTTP - протокол уровня приложений для распределённых, объединённых, гипермедийных информационных систем, используемый в глобальной информационной инициативе Всемирной паутины с 1990 года. [↑](#footnote-ref-3)
4. Фреймворк (англ. framework - «остов, каркас, структура») - готовая модель в IT, заготовка, шаблон для программной платформы, на основе которого можно дописать собственный код. [↑](#footnote-ref-4)
5. Maintainability - легкость изменения программного продукта для исправления дефектов и соответствия новым требованиям, с целью облегчения последующего сопровождения или для адаптации к изменившемуся окружению. [↑](#footnote-ref-5)