Пример отчета по практике «SPbPU-student-thesis-template». Версия от 20 мая 2021 г.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ПЕТРА ВЕЛИКОГО»**
НАЗВАНИЕ ИНСТИТУТА
НАЗВАНИЕ ВЫСШЕЙ ШКОЛЫ

Отчет о прохождении такой-то практики на тему: «Тема практики»

Фамилия Имя Отчество, гр. N

Направление подготовки: XX.XX.XX Наименование направления подготовки.
Место прохождения практики: СПбПУ, ИКНТ, ВШИСиСТ.
Сроки практики: с дд.мм.гггг по дд.мм.гггг.
Руководитель практики от ФГАОУ ВО «СПбПУ»: Фамилия Имя Отчество, должность, степень.
Консультант практики от ФГАОУ ВО «СПбПУ»: Фамилия Имя Отчетство, должность, степень.
Оценка: _____

Руководитель практики
от ФГАОУ ВО «СПбПУ» И.О. Фамилия

Консультант практики
от ФГАОУ ВО «СПбПУ» И.О. Фамилия

Обучающийся И.О. Фамилия

Дата: дд.мм.гггг

СОДЕРЖАНИЕ

Введение 4
0.1. Постановка задачи 5
Глава 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 6
1.1. Технология REST 6
1.2. Технология GraphQL 7
1.3. Сравнение технологий REST и GraphQL 8
1.3.1. Преимущества GraphQL 9
1.3.2. Недостатки GraphQL 10
1.4. Проблемы миграции на GraphQL 11
1.5. Выводы 12
Глава 2. ПРЕДЛАГАЕМАЯ ТЕХНОЛОГИЯ 12
2.1. Предлагаемая технология 13
2.1.1. Достоинства предлагаемой технологии 13
2.1.2. Недостатки предлагаемой технологии 15
2.2. Требования к реализации 16
2.3. Выводы 17
Глава 3. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ СЕРВИСА 17
3.1. Требования к реализации 18
3.2. Выбор технологий 18
3.3. Обзор исходных кодов реализации 20
3.3.1. Пакет controller 20
3.3.2. Пакет service 21
3.3.3. Пакет util 23
3.3.4. Пакет property 23
3.3.5. Пакет bpp 25
3.3.6. Модуль test 26
3.4. Выводы 26
Глава 4. ТЕСТИРОВАНИЕ РЕАЛИЗАЦИИ 26
4.1. Данные для тестирования реализации 26
4.2. Функциональное тестирование 27
4.3. Тестирование производительности 27
4.4. Возможности улучшения производительности 29
4.5. Выводы 30
Заключение 31

Список сокращений и условных обозначений 32
Словарь терминов 33
Список использованных источников 34
Приложение 1. Краткие инструкции по настройке издательской системы LaTeX 35
Приложение 2. Некоторые дополнительные примеры 37

ВВЕДЕНИЕ

В течение многих лет основными архитектурными подходами, использующимися в сфере информационных технологий для обмена данными между системами, являлись такие технологии как SOAP (Simple Object Access Protocol) и REST (Representational state transfer).

Очевидным их отличием является используемый способ представления данных. В случае SOAP единственным поддерживаемым форматом является XML, а в случае REST могут использоваться форматы JSON, HTML и XML. Каждый из подходов имеет свои достоинства и недостатки, однако на данный момент REST является приоритетным выбором для современных разработчиков в силу легковесности и лучшей читаемости формата JSON, гибкости и простоте использования, особенно в Web-разработке, где он обязан своей популярности языку JavaScript.

Однако при работе с REST разработчики также столкнулись с определёнными ограничениями и недостатками. Основными из них считаются так называемые overfetching и underfetching, когда в ответ на запрос REST-клиент получает лишние данные, или наоборот недостаточное их количество, из-за чего клиент вынужден выполнять другой запрос. Это часто случается, когда меняется дизайн приложения, или изменяется или добавляется новая функциональность, переиспользующая ранее существовавшие запросы. В случае недостаточности данных разработчику серверной части приходится вносить изменения в код, чтобы добавить требуемые данные, а избыточность зачастую игнорируется. Проблема усугубляется наличием нескольких типов клиентов – например, Android, iOS и веб-приложений.

Необходимость постоянных небольших изменений на серверной стороне приводит к замедлению процесса разработки. Как один из способов решения этой проблемы в компании Facebook для внутреннего использования был создан язык запросов GraphQL, позволивший разработчикам на стороне клиента самостоятельно выбирать те данные, которые им необходимо получить, с помощью специального языка запросов.

После публикации спецификации в открытый доступ, разработчики заинтересовались новым гибким подходом, однако столкнулись с рядом препятствий при попытке внедрить данную технологию в разрабатываемые ими системы. Основными из них стали:

- невозможность переработки прошлых версий мобильных приложений для использования GraphQL;
- необходимость в поддержке обоих протоколов на время миграции;
- неготовность разработчиков клиентских приложений изучать и внедрять новую технологию.

Для решения указанных проблем автор предлагает создать сервис, который станет посредником между GraphQL-сервером и REST-клиентом. Автор ожидает, что в этом случае будут решены перечисленные проблемы, а также получены дополнительные преимущества по сравнению с использованием каждой технологии в отдельности.

0.1. Постановка задачи

Целью данной работы является создание прототипа указанного сервиса-прослойки. Для этого автор должен решить следующие задачи:

- Изучить технологии REST и GraphQL;
- Осуществить сравнение GraphQL с другими архитектурами и протоколами;
- Составить техническое задание для реализации сервиса;
- Подготовить данные и окружение для тестирования;
- Разработать прототип сервиса и протестировать его на подготовленных данных.

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Хорошим стилем является наличие введения к главе, которое *начинается непосредственно после названия главы, без оформления в виде отдельного параграфа*.

1.1. Технология REST

REST (Representational state transfer) это архитектурный стиль взаимодействия. REST не является протоколом или стандартом, в отличие от SOAP, так как не даёт строгих правил взаимодействия, например позволяя передавать в запрос и ответ как JSON, так и XML или бинарные файлы. Однако он накладывает ряд ограничений [1], при соблюдении которых сервис считается RESTful:

Разделение сервера и клиента

Логика и визуальный интерфейс клиента, за которые отвечает клиентское приложение, не затрагивают то, каким образом данные хранятся и обрабатываются на сервере. Благодаря этому облегчается создание других клиентских приложений и улучшается масштабируемость за счёт упрощения компонентов сервера.

Отсутствие состояния

Каждый запрос рассматривается отдельно, сервер не хранит контекст для клиента, что улучшает масштабируемость, использование ресурсов и отказоустойчивость, однако может увеличить нагрузку на сеть за счёт того, что часть данных будет передаваться с каждым запросом.

Кэширование

Клиенты и промежуточные узлы должны иметь возможность кешировать ответы сервера, а ответы сервера должны явно или неявно указывать на возможность кеширования. Очевидным образом это улучшает производительность и отказоустойчивость системы.

Единообразие интерфейса

Этот принцип позволяет каждому сервису развиваться независимо.

Вот ряд признаков такого интерфейса:

– Идентификация ресурса – каждый ресурс и каждый управляемый им объект имеет уникальный идентификатор – URI и ID, соответственно. Например, для получения списка выставленных счетов мы можем обра-

титься с запросом `GET /api/invoices`, и в ответе получить информацию о счетах с идентификаторами `123` и `567`.

- Манипуляция ресурсами через представление – клиент имеет возможность модифицировать информацию на основании метаданных, получаемых в других запросах. Например, получив ID объекта в одном запросе, он может удалить этот объект по ID в другом. В предыдущем пункте мы получили счёт с идентификатором `123`, и мы можем удалить этот счёт с помощью запроса `DELETE /api/invoices/123`.
- Самоописываемые сообщения – запрос и ответ содержит достаточно информации, чтобы понять, как его обрабатывать. Например, сообщение может иметь заголовок с MIME-типом: `Content-Type: application/json`.
- Гипермедиа как способ управление состоянием – клиенту не требуется заранее знать, как взаимодействовать с системой за пределами гипермедиа. Например, получая информацию о балансе, клиент получает список возможных с ним действий. В частности, при отрицательном балансе в списке действий не будет вывода денег, а лишь опция пополнить счёт, и данная логика не должна быть заложена приложении-клиенте.

Слои

Структура системы является иерархической, клиент может взаимодействовать с ресурсом не напрямую, что позволяет повысить масштабируемость за счёт кеширования и балансировки нагрузки на промежуточных узлах. А также наличие слоёв позволяет добавить абстракцию от устаревших (legacy) компонентов системы и устаревших клиентов, добавив перед ними сервисы-адаптеры. Например, при переходе на API версии 2, можно создать сервис, предоставляющий API версии 1, который сам будет обращаться к API версии 2.

Код по требованию

Это ограничение указано как «необязательное» ограничение, допускающее в архитектуре системы использование загружаемого кода в виде апплетов.

1.2. Технология GraphQL

GraphQL это открытый язык запросов для получения и манипуляции данными. В отличие от REST, он имеет достаточно строгую спецификацию [2]. Система, работающая с GraphQL, имеет так называемую schema, которая описывает API. Схема состоит из сущностей нескольких типов: object, query и mutation. Object

является самой простой из них и представляет собой набор полей, являющихся другими объектами, коллекциями объектов или скалярами – строками, целыми числами, числами с плавающей запятой, булевыми значениями или ID. Поля также могут принимать аргументы: например, при получении информации о сумме денег на счету пользователя, мы можем передать валюту, в которой хотим получить ответ. Query и mutation являются особыми типами объектов, использующимися для получения и изменения данных соответственно.

Приведём пример запросов и ответов на GraphQL. Примеры будут выполняться на схеме, продемонстрированной в приложении 1. Данная схема имеет сущности User, Account и Card, а также ряд query и mutations, некоторые из которых сейчас будут рассмотрены.

Пример выполнения query

Создадим запрос для получения списка валютных счетов и привязанных к этим счетам карт пользователей, отсортированных по имени в алфавитном порядке.

В приложении 2 представлены выполняемый запрос слева и полученный ответ справа. Разберём его более детально. В строке 2 указываем, что хотим получить список users, отсортированный по полю имя (name). Для каждого пользователя получаем его id, имя (name), а также список счетов (account). Среди счетов нас интересуют те, значения валюты (currency) которых не равно RUR, это условие учтено в строке 5. И для каждого счёта получаем id, остаток (amount), валюту (currency) и список карт (cards). Для каждой карты получаем её маскированный номер (number_masked) и платёжную систему (system).

Как видно из результата выполнения, в ответ были получили только поля, указанные в запросе.

Пример выполнения mutation

Изменим сумму на счету у одного из пользователей, полученных в прошлом запросе.

Для этого воспользуемся mutation под названием updateAccount, установив новое значение amount для счёта с идентификатором 4. В этом же запросе получим в ответ обновлённое состояние счёта, а также имя пользователя, которому этот счёт принадлежит. Запрос и ответ продемонстрированы в приложении 3.

1.3. Сравнение технологий REST и GraphQL

1.3.1. Преимущества GraphQL

Некоторые преимуществ, получаемые при использовании GraphQL и соответствующего ему подхода, уже были перечислены во введении, однако рассмотрим их детальнее [3, 4]:

Клиент может конкретно указать, какие данные ему требуются и в каком виде.

Это позволяет сэкономить количество сетевых вызовов, обращений к базе данных, памяти и файловой системе, сэкономить трафик и избавиться от лишних преобразований, конвертаций и сортировок. Предположим, есть сервис, который выводит топ-25 фотографий пользователя в высоком качестве, отсортированными по количеству лайков и с несколькими лучшими комментариями. Затем на странице профиля потребовалось выводить превью пяти последних фотографий, с сортировкой по дате, для чего был добавлен новый эндпоинт. Затем потребовалось создать мобильные версии для обоих экранов, в которых количество фотографий меньше, не отображаются комментарии, а вместо полноразмерных изображений используются миниатюры. Это потребует двух доработок на стороне сервера, что было посчитано нецелесообразным ввиду недостатка времени у разработчиков. В итоге на стороне сервера осуществляется лишний запрос для получения комментариев к изображению, а пользователь тратит время и трафик на загрузку изображений в большом качестве. В случае с GraphQL разработчику клиента в обоих случаях достаточно будет лишь изменить в запросе способ сортировки и количество запрашиваемых элементов, а в списке получаемых полей поменять ссылку на изображение на ссылку на превью.

Упрощается агрегация данных из нескольких источников в одном запросе.

Допустим, у нас есть сервис, предоставляющий информацию о счетах, и сервис, предоставляющий информацию о картах. Чтобы получить информацию о счетах и привязанных к ним картам, придётся либо сделать два запроса и сопоставить их результаты на стороне клиента, либо создать ещё один сервис, агрегирующий информацию и предоставляющий её в требуемом нам виде. При использовании GraphQL один сервис, называемый BFF (Backend For Frontend), является универсальным агрегатором.

Используется система типов для описания данных.

Схема является контрактом между клиентом и сервером, позволяет задать для запросов и ответов типы полей, списки возможных значений (enum), обязательность их наличия (nullability).

1.3.2. Недостатки GraphQL

Однако GraphQL имеет и недостатки в сравнении с REST [4, 5]:

Необходимость в управлении дополнительными ограничениями.

Так как потребитель GraphQL API имеет возможность самостоятельно выбирать те данные, которые он хочет получить, встаёт вопрос безопасности. Например, недобросовестный пользователь может отправить на сервер запрос для получения полной информации обо всех пользователях, чтобы использовать их в целях, идущих вразрез с интересами компании. Либо отправить множество ресурсоёмких запросов с целью вызвать отказ в обслуживании этого сервиса. Для предотвращения обоих атак разработчику необходимо задавать дополнительные ограничения, предугадывая возможные способы злоупотребления. Также у некоторых реализаций есть механизм Persistent Queries, позволяющих задавать все возможные запросы, и обращаться к ним по уникальному идентификатору.

Отсутствующее поле и null неотличимы.

Например, существует запрос mutation для обновления информации о пользователе, позволяющий изменять его имя, адрес и номер телефона. Эти поля могут присутствовать или отсутствовать в запросе, чтобы клиенту не приходилось отправлять адрес и номер телефона, если он хочет изменить имя. Однако если пользователь захочет полностью удалить значение адреса, передав null, то сервер решит, что поле адреса не должно быть изменено.

Отличие формата ввода и вывода.

Для иллюстрации воспользуемся одним из принципов REST – отсутствием состояния. Допустим, в одном запросе клиент получил некий контекст, который он должен передать в следующем запросе. В случае REST сервер и клиент обмениваются идентичным объектом. А в случае с GraphQL для этой цели в схеме должны быть определены отдельные типы для ввода и вывода, а на клиенте осуществляться составление запроса с использованием полей из ответа.

Отсутствие поддержки полиморфизма для mutation.

В то время как для object можно задать другой object в качестве интерфейса (либо использовать union), для полей в mutation наследование не предусмотрено.

Например, нужно передать информацию о владельце счёта. Если владельцем является физическое лицо, то нужны его фамилия, имя и отчество, а для юридического лица – наименование предприятия. В случае с REST клиент может передать дополнительное поле, содержащее тип владельца счёта, и сервер сможет осуществить десериализацию в нужный тип. А при использовании GraphQL придётся создать отдельный запрос для каждого типа.

Отсутствие namespaces.

Один GraphQL сервис имеет единственную схему. В случае большого проекта схема может достигать значительных объёмов, и ориентирование в ней потребует наличие дополнительной документации, а также увеличивается вероятность коллизии имён. REST в свою очередь не использует такое понятие как тип передаваемого объекта, и управление ими осуществляется средствами клиента и сервера в отдельности.

Ограничение использования подхода Backend Driven UI.

При использовании этого подхода управление пользовательским интерфейсом передаётся на серверную сторону: приложение представляет собой набор виджетов. А список и порядок виджетов, отображаемых на каждом экране, содержимое каждого виджета, а также способ перехода между экранами, приложение получает от сервера. При использовании GraphQL разработчик должен будет решить, каким образом передавать изменяющийся запрос с сервера на устройство, либо в ответ на каждое изменение редактировать схему, создавая или изменяя имеющиеся query.

Отсутствие механизмов кеширования.

На уровне HTTP технология GraphQL использует метод POST для осуществления запросов. Но механизмы кеширования HTTP не кешируют запросы по методу POST [**graphqlcaching**]. Однако предполагается, что

Как видно, обе технологии имеют свои преимущества и недостатки, но решение об использовании того или иного подхода должно приниматься с учётом специфики конкретного проекта.

1.4. Проблемы миграции на GraphQL

В случае, если разработчики решат начать использовать GraphQL в уже существующем продукте, они могут столкнуться с рядом проблем. Чтобы наглядно продемонстрировать возможные проблемы, рассмотрим пример.

Допустим, что некая компания имеет мобильное приложение для платформ Android и iOS, а также веб-версии приложений для компьютеров и мобильных устройств. Часть пользователей не имеет возможности обновлять приложение из-за того, что их версия операционной системы больше не поддерживается приложением, однако компания не желает терять прибыль от этих пользователей.

В компании периодически производится редизайн приложения, в связи с чем на большом количестве экранов меняется формат представления данных, в связи с чем должны быть изменены и форматы запросов к серверу и ответов от него. Из-за большого количества имеющихся платформ сильно возрастает нагрузка на разработчиков серверной части, так как необходимо разрабатывать сервисы с учётом различий в представлении результата на каждой платформе.

Использование GraphQL позволило бы снизить нагрузку на разработчиков серверной части, так как в этом случае им достаточно создать ряд универсальных источников и предоставить схему, для которой разработчики клиентских приложений будут писать запросы.

Однако внедрение новой технологии означает, что её использованию необходимо обучить каждого разработчика каждой из платформ, а затем в течение длительного времени поддерживать одновременно две версии API – GraphQL для новых версий приложения, и REST для старых.

1.5. Выводы

Таким образом, мы рассмотрели некоторые из современных технологий, используемые для организации клиент-серверного взаимодействия. Были рассмотрены их основные принципы, а также преимущества и недостатки технологий в сравнении друг с другом. В результате

ГЛАВА 2. ПРЕДЛАГАЕМАЯ ТЕХНОЛОГИЯ

Хорошим стилем является наличие введения к главе, которое *начинается непосредственно после названия главы, без оформления в виде отдельного параграфа*.

2.1. Предлагаемая технология

В качестве решения для перечисленных проблем автор предлагает создание дополнительного сервиса в виде адаптера между REST-клиентом и GraphQL-сервером. Данный сервис должен хранить маппинг (соответствие между запросами двух типов), и при получении REST-запроса находить в своей базе соответствующий шаблон GraphQL-запроса, заполнять его данными из REST-запроса и отправлять к GraphQL-сервису для выполнения.

Помимо описанной основной функциональности также возможна реализация следующих функциональностей:

- В случае, если для REST-запроса не было найдено соответствия, то запрос должен быть передан gateway-сервису – таким образом, описываемый сервис может стать единственной точкой доступа к системе.
- При необходимости может быть реализовано приведение ответа GraphQL–сервиса к другому виду с целью сохранения обратной совместимости.

2.1.1. Достоинства предлагаемой технологии

Рассмотрим достоинства данного подхода в сравнении с использованием REST либо GraphQL:

- Гибкость и мощь языка GraphQL – разработчики имеют возможность писать полноценные GraphQL-запросы и разрабатывать серверную часть в соответствии со всеми принципами GraphQL.
- Обратная совместимость на клиенте – старая версия приложения может продолжать функционировать после полного перехода на использование GraphQL на сервере.
- Разработчикам клиентских приложений не нужно обучаться новой технологии и внедрять её – взаимодействие с сервером по-прежнему осуществляется через привычный REST.
- Обязанности по написанию, отладке и редактированию запросов могут брать на себя также аналитики и сотрудники отдела сопровождения, не требуя участия разработчиков.
- Возможность постепенной миграции – запросы, которые не были реализованы в GraphQL, передаются старым сервисам, и затем неявно подменяются новой реализацией.

- Такой сервис может обращаться ко множеству различных GraphQL-сервисов, позволяя разграничить зоны ответственности и решить проблему именования 1.3.2.
- Есть возможность использовать существующие механизмы кеширования на клиенте и промежуточных узлах, непригодные для использования с GraphQL.
- Подобная система может быть реализована для любого протокола общения с клиентом – например, вместо REST может использоваться протокол SOAP.
- Для добавления и изменения маппинга необязательна перезагрузка сервиса, что позволяет вносить и проверять изменения гораздо быстрее, чем это происходит при изменении кода. Актуализация маппинга может происходить периодически, при обнаружении изменений или по иному событию.
- Возможность редактирования запроса на сервере в реальном времени, что невозможно при использовании механизма Persistent Queries, при котором клиент использует хэш-код одного из заранее заданных неизменяемых запросов.
- Настройки могут браться из любого типа источника – git или другой системы контроля версий, базы данных, config-серверов, локальных yaml-файлов и т.п.
- Хранение маппингов в системе контроля версий позволит использовать уже имеющуюся ролевую модель, в том числе налаженные процессы для ревью изменений, а также защитить маппинги от несанкционированного изменения и утраты, использовать версионирование.
- Форматом ответа можно легко манипулировать с помощью технологий для форматирования JSON – например, библиотеки JOLT [6].
- В случае миграции с данного решения на использование GraphQL без посредников разработчики будут иметь в своём распоряжении готовые и протестированные GraphQL-запросы. А для поддержки старых версий приложения после миграции не придётся поддерживать старый REST-кластер, а только данный сервис.
- Уменьшается объём кода, отвечающего за отображение, в сервисах. Появляется новый уровень абстракции, позволяющий разработчикам писать более чистый код.

– Данный сервис в качестве отдельного слоя может быть протестирован независимо от остальной системы при помощи автотестов: для этого могут использоваться «заглушки», вызываемые вместо реальных сервисов на тестовом стенде.

2.1.2. Недостатки предлагаемой технологии

– Дополнительный слой в цепочке вызовов замедляет выполнение запросов (увеличивается latency). Фактическое время исполнения запросов предлагаемым сервисом будет измерено и оценено при его апробации.
– Сервис и источники маппингов являются новыми потенциальными точками отказа. При ошибке в маппинге или программном или аппаратном сбое вся система или её часть может оказаться недоступной. Для увеличения отказоустойчивости можно применить горизонтальное масштабирование, увеличивая число реплик, а также размещая их в различных датацентрах. Также можно разделять ответственность за разные части системы между разными кластерами предлагаемого сервиса, чтобы уменьшить возможные последствия.
– В случае недоступности источника маппингов сервис не сможет быть перезапущен. Например, это может произойти при отказе базы данных с маппингами или недоступности GIT-репозитория. Для защиты от данного риска следует предусмотреть резервный источник. Им может быть балансировщик со включенным кешированием или дополнительная база данных.
– При росте количества маппингов может значительно возрасти сложность поддержки системы. Для упрощения управления маппингами следует логически разделять
– Нужен новый механизм тестирования изменений. Маппинги создаются и изменяются в процессе работы сервиса. Поэтому правильность задания каждого конкретного маппинга может быть проверена только через запуск сервиса. Запустить сервис можно как на тестировочном стенде, так и при сборке приложения в рамках интеграционного тестирования. Однако первый вариант достаточно ресурсоёмкий, а второй требует навыков разработки. Поэтому для возможности добавления, изменения и проверки маппингов людьми без опыта разработки следует разработать

специальное программное обеспечение, которое будет получать список возможных запросов, а также имеющихся маппингов, чтобы проверить их корректность, а также то, что для каждого запроса существует не более одного подходящего маппинга.

– Создание, изменения и проверки маппингов требует специальных навыков помимо знания языка GraphQL. Способ решения этого недостатка описан в предыдущем пункте.

Как видно, количество ожидаемых преимуществ значительно превышает количество ожидаемых недостатков, в связи с чем разработку подобной системы считаем целесообразной.

2.2. Требования к реализации

Реализация предлагаемой системы в рамках выпускной квалификационной работы должна обладать следующей функциональностью:

– Загружать маппинги из источника, которым является git-репозиторий, ссылка на который указывается в настройках.
– Принимать REST-запросы, имеющие методы `GET`, `POST`, `PUT`, `DELETE`.
– Находить для запроса соответствующий маппинг по методу запроса и URI запроса. URI запроса, указанный в маппинге, может иметь `path variable`, то есть для следующего запроса
  `GET /api/users/123/accounts?currency=RUR`
  должен быть найден следующий маппинг:
  `GET /api/users/#{userId}/accounts`
– Использовать `path variables`, `query params` и заголовки запроса в качестве переменных для формирования запроса по шаблону, указанному в маппинге. Так, в предыдущем примере на основании запроса будут задаваться значения двух переменных: `userId=123` и `currency=RUR`.
– Также использовать тело для получения значений переменных. Например, при получении запроса со следующим телом
  `{ "name": "Александров" "address": {"street": "Лубянка" "house": "1"}}`
  будут получены значения переменных
  `name=Александров ; address.street=Лубянка ; address.house=1`

– Подставлять значения переменных в шаблон запроса, хранимый в маппинге, на место плейсхолдеров с соответствующий именем. Например, следующий запрос из маппинга:
  `{ users(id: "#{userId}"){ accounts(currency_eq: "#currency") { number } }`
  должен быть заполнен следующим образом:
  `{ users(id: "123"){ accounts(currency_eq: "RUR") { number } }`
– В случае, если маппинг не был найден, запрос должен быть в неизменном виде (с сохранением URI, `query params`, заголовков и тела запроса) направлен по адресу API Gateway, заданному в настройках.
– При наличии нескольких подходящих маппингов, выбирать более специфичный. Например, при наличии маппингов
  `GET /users/#{userId}` и `GET /users/#{userId}/accounts/#{accountId}`
  при поступлении запроса
  `GET /users/123/accounts/456`
  должен быть использован второй маппинг.
– Осуществить преобразование полученного ответа осуществляется при наличии правила для преобразования в маппинге. Для преобразования используется технология, указанная в маппинге. Правила для осуществления преобразования задаются в соответствующем указанной технологии формате. Выбор поддерживаемых технологий осуществляется разработчиком.

Диаграмма, иллюстрирующая взаимодействие между клиентом и описываемым сервисом изображена в приложении 4.

2.3. Выводы

Параграф с изложением выводов по главе *является обязательным*.

ГЛАВА 3. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ СЕРВИСА

В данной главе будет рассмотрен процесс выбора технологий и планирования архитектуры системы, в которой будет использоваться предлагаемая технология, а также внутренняя архитектура самой реализации.

3.1. Требования к реализации

Спецификация, описанная в главе 2, не указывает единственный разрешённый источник для получения маппингов. Источником может быть реляционная или нереляционная база данных, система контроля версий, расположенный локально в файловой системе файл, специализированный веб-сервер для получения настроек, или любое другое хранилище данных. Кроме того, сам формат маппингов не является строго определённым, так как они могут храниться в различных источниках. Таким образом, формат хранения настроек В спецификации не определяется ожидаемое поведение системы при получении каких-либо ошибок со стороны GraphQL-сервера. Также не ограничивается список технологий, которые могут быть использованы для трансформации ответа в формате JSON. Таким образом, для данной реализации необходимо определить детали требований, которые будут использоваться при проверке его работоспособности и апробации.

Определим список свойств и особенностей данной, выбранных для данной реализации, следующим образом:

- Маппинги указываются в системе контроля версий GIT
- По умолчанию берутся из локально расположенного текстового файла
- В качестве формата для маппингов используется формат YAML;
- В случае получения ответа с кодом ответа HTTP отличным от кода 200 (успешный ответ), ответ передаётся клиенту в неизменном виде.
-

3.2. Выбор технологий

Для системы, реализуемой в рамках выпускной квалификационной работы, в дальнейшем будет использоваться название R2G (Rest to GraphQL).

Для создания системы будет использован Spring Framework, являющийся одним из самых распространённых и мощных фреймворков для создания веб-сервисов. Spring Framework включает в себя множество различных модулей, которые позволяют значительно ускорить разработку и уменьшить количество необходимого кода за счёт использования многочисленных дополнительных модулей. В частности, будут использованы следующие модули:

- Spring Boot позволяет подключать так называемые стартеры (starters), которые включают в себя другие модули и упрощают их конфигуриро-

вание. За счёт использования стартеров значительно сокращается объём необходимой работы.

- Spring Web используется для создания эндпоинтов для обработки входящих REST-запросов. Данная технология позволяет задать точки входа в веб-сервис, а также предоставляет функциональность для обработки входящих запросов.
- Spring Cloud Config для управления настройками сервиса, в частности для возможности получения настроек и маппингов из GIT-репозитория.

Spring Framework изначально разрабатывался для использования с языком Java, однако на данный момент поддерживает значительное число языков JVM, и в рамках данной работы будет использован язык Kotlin, имеющий среди преимуществ перед Java лаконичность и возможность использования корутин (coroutine, легковесных потоков) для повышения производительности при использовании многопоточной обработки запросов. Также при использовании языка Kotlin упрощается использование реактивной парадигмы программирования. Достоинством этого подхода при создании информационной системы является поддержка неблокирующего ввода/вывода, за счёт которого увеличивается количество запросов, которые сервис может обрабатывать параллельно. Таким образом увеличивается скорость обработки запросов при высокой нагрузке на веб-сервис, а также имеющиеся ресурсы утилизируется более эффективно.

Для взаимодействия с GraphQL-сервисами существуют специальные библиотеки, как например Apollo Client, позволяющие упростить создание клиентских приложений за счёт генерации кода на основании схемы и GraphQL-запросов. Однако подобный подход неприменим в данном случае, так как разрабатываемый сервис должен уметь работать с GraphQL-сервисами в общем случае. Поэтому для обращения к GraphQL-сервисам будет использован обычный HTTP-клиент. В качестве одного из HTTP-клиентов Spring Framework предоставляет технологию WebClient, который позволяет применить ранее упомянутый неблокирующий ввод/вывод, как следствие мы получаем все его достоинства.

Для преобразования результатов используем технологию JOLT, так как она является достаточно мощной технологией, позволяющей осуществлять разнообразные манипуляции с JSON, позволяет писать юнит-тесты для проверки написанных преобразований. Кроме того, JOLT является одной из немногих подобных технологий имеет реализацию в виде библиотеки для Java.

3.3. Обзор исходных кодов реализации

Исходный код прототипа содержит ряд пакетов, логически разделяющих классы. Каждый пакет представляет из себя уровень абстракции над действиями, выполняемыми сервисом при обработке запросов. Исходные коды классов приведены в приложении 8.

При описании реализации будет использоваться термин bean (бин), который обозначает некий объект, которым управляет Spring Framework. Во многих случаях термины бин и объект являются взаимозаменяемыми.

На рис.3.1 изображено дерево пакетов модуля main, содержащего исходные коды приложения.

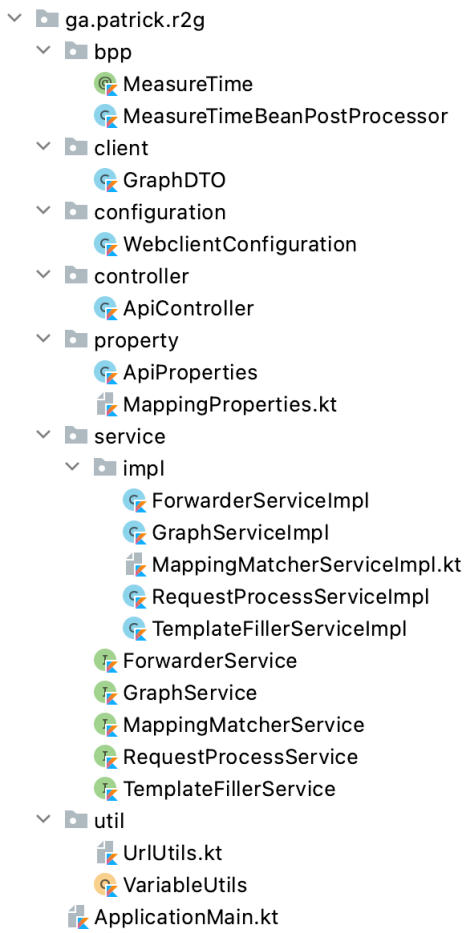

Рис.3.1. Дерево пакетов модуля main

3.3.1. Пакет controller

Данный пакет традиционно включает в себя классы, ответственные за принятие запроса к сервису и отправку ответа. Каждый класс содержит набор методов

с указаниями о том, какие запросы данный метод обрабатывает, и какие данные требуются для его работы. Данные указания обычно даются в виде аннотаций.

Этот пакет имеет единственный класс ApiController, имеющий единственный метод endpoint. Аннотация `@RequestMapping("/**")` указывает на то, что данный метод принимает запросы с любыми `path` и `method` (`GET`, `POST` и прочие). Также, данный метод принимает в качестве аргумента объект `HttpServletRequest`, который содержит информацию о пришедшем запросе, такую как `path`, `method`, параметры `query`, тело запроса, словарь заголовков. Таким образом, при получении любого запроса к данному веб-сервису информация об этом запросе будет передана в качестве аргументов при вызове этого метода. Метод не имеет никакой логики, кроме вызова метода `processRequest` объекта `RequestProcessService`, который будет рассмотрен далее.

3.3.2. Пакет service

Данный пакет включает в себя классы, ответственные за так называемую бизнес-логику. В корне данного пакета находится ряд интерфейсов. Интерфейсы используются в качестве контрактов для сервисов, которые планируется реализовать. Подход, при котором сначала создаются интерфейсы, и только затем имплементации, широко используется при использовании Test Driven Development. При использовании этого подхода котором прежде чем приступить к реализации продумывается внутренняя архитектура системы, создаются классы-интерфейсы, и для них пишутся тестовые сценарии, описывающие ожидаемое поведение системы. Затем, классы, реализующие эти интерфейсы, помещяются в отдельных файлах, обычно в подпакетах с названием impl и именем, состоящим из названия интерфейса и постфикса Impl. Также, данный подход позволяет создавать несколько реализаций для каждого интерфейса, позволяя подменять эти реализации перед запуском с помощью файла настроек, или даже в процессе работы программы. Также, использование интерфейсов позволяет немного увеличить производительность и ускорить автоматизированное тестирование за счёт использования механизмов динамического проксирования JDK (Java Developer Kit), вместо технологии CGLIB. Обе технологии используются для создания так называемых Proxy-объектов и Mock-объектов.

Proxy-объекты используются для неявного применения дополнительного поведения к какому-либо классу или методу, а также при использовании парадигмы

АОП (Аспектно-ориентированное программирование). Например, Proxy-объекты могут использоваться для кеширования ответа какого-либо метода, логирования аргументов и ответов методов, или для подсчёта времени выполнения метода. В частности, последнее применение будет описано далее при обзоре аннотации `@MeasureTime` и класса `MeasureTimeBeanPostProcessor`.

Mock-объекты используются для эмуляции поведения какого-либо объекта в процессе проведения автоматизированного тестирования. Например, какой-либо сервис в процессе своей работы осуществляет обращение к базе данных через некий бин. Создание, наполнение и очистка базы данных для каждого теста являются значительными накладными расходами. А в случае с сетевыми вызовами в большинстве случаев использование реального обращения к внешним сервисам (например, на тестовом или разработческом стенде) является либо невозможным, либо слишком медленным. Кроме того, согласно методологии тестирования модули программы должны быть протестированы отдельно. Поэтому для модульных (компонентных) тестов реализация зависимого бина подменяется Mock-объектом, для которого вручную задаётся определённое поведение. Например, если бин получил в качестве аргумента строку `"a"`, то он должен будет ответить строкой `"b"`. Применение Mock-объектов будет продемонстрировано в обзоре модуля `test`.

Технология CGLIB является более медленной, чем механизм динамического проксирования JDK, но механизм JDK работает за счёт реализации тех же интерфейсов, что и исходный класс. Соответственно, для использования этого механизма JDK наш проксируемый/мокируемый класс должен реализовывать некий интерфейс.

Рассмотрим интерфейсы пакета service. Согласно принципу Single Responsibility, каждый компонент имеет строгую зону ответственности. Ответственность каждого из них описана ниже:

- RequestProcessService – является ключевым компонентом системы. Имеет единственный метод `processRequest`, который принимает объект входящего запроса `HttpServletRequest` и осуществляет отправку исходящего запроса к нужному сервису. В частности, этот метод получает соответствующий запросу маппинг, и в случае успеха формирует и отправляет запрос к соответствующему GraphQL-сервису. А в случае, если маппинг не был найден, перенаправляет запрос к некому серверу по умолчанию, указанному в настройках.

- MappingMatcherService – используется для поиска маппинга, соответствующего входящему запросу. Он имеет единственный метод `findMapping`, который принимает объект запроса `HttpServletRequest`, находит и отдаёт соответствующий ему маппинг из настроек.
- TemplateFillerService – заполняет шаблон GraphQL-запроса параметрами, полученными из входящего запроса, в соответствии с маппингом. Компонент получает значения переменных для заполнения из URI, query parameters и тела запроса.
- GraphService – осуществляет отправку GraphQL-запроса. Имеет единственный метод send, принимающий URL запрашиваемого сервиса и объект GraphDTO, который будет отправлен в GraphQL-сервис.
- ForwarderService – перенаправляет запросы, для которых не нашлось маппинга, в Gateway API, указанный в настройках. Имеет единственный метод send, принимающий объект `HttpServletRequest`. Запрос передаётся с сохранением URI, headers, query parameters, тела и прочих параметров входящего запроса в исходном виде.

3.3.3. Пакет util

Данный пакет включает в себя единственный класс VariableUtils, который содержит утилитарные методы для:

- получения маски маппинга, которая будет использована для сравнения с пришедшим запросом;
- получения списка переменных, которые ожидается найти в пришедшем запросе;
- получения значения переменных из пришедшего запроса.

Для получения значений из тела запроса, представляющего из себя JSON, использована библиотека JsonPath.

3.3.4. Пакет property

Данный пакет содержит файл MappingProperties, содержащий следующие классы, по своей структуре повторяющие структуру маппингов, записанных в виде YAML-файла (например, application.yml):

- MappingProperties – корневой класс настроек. Он имеет в себе следующие поля:

defaultEndpoint – URL эндпоинта по умолчанию

mappings – список объектов типа Mapping, каждый из которых представляет собой описание правила для маппинга входящего запроса к исходящему, а также правило преобразования ответа.

endpoints – список доступных GraphQL-серверов. В этом списке указываются псевдонимы для URL этих серверов, чтобы избежать их многократного дублирования в маппингах.

– Mapping – описание правила для маппинга входящего запроса к исходящему:

  path – шаблон пути, который может содержать переменные;

  methods – HTTP методы запроса;

  endpointName – псевдоним GraphQL-сервера, на который нужно отправить запрос;

  template – шаблон для запроса, представляющий собой GraphQL-запрос, возможно, содержащий в себе переменные, которые будут заполнены с помощью данных из входящего запроса.

  variables – список объектов Variable – описаний переменных, поддерживаемых GraphQL, которые используются в шаблоне, и значения для которых мы также получаем из входящего запроса.

– Endpoint – объект, содержащий два поля:

  name – псевдоним GraphQL-сервера;

  uri – адрес этого сервера.

– VariableDefinition – список переменных, которые следует отправить в составе GraphQL-запроса.

  name – название переменной, используемой в GraphQL-запросе;

  source – данные для получения значения этой переменной из входящего запроса;

  type – тип данных переменной. Возможные значения содержатся в перечислении GraphVariableType.

– GraphVariableType – набор возможных типов переменных. На данный момент поддерживаются строковые, целочисленные и дробные переменные.

3.3.5. Пакет bpp

Данный пакет включает в себя класс `MeasureTimeBeanPostProcessor` и аннотацию `MeasureTime`. Аннотация создана для замера скорости выполнения запроса, и должна быть поставлена на классе, скорость выполнения функций которого нас интересует. Результаты использования этой аннотации будут более подробно рассмотрены в главе 4.

В данный момент эта аннотация стоит на классах `RequestProcessServiceImpl` и `GraphServiceImpl`. Соответственно, мы сможем засечь время выполнения функции `processRequest` и вычесть из него время выполнения фунции `send`, таким образом получив чистое время обработки запроса кодом приложения.

Класс `MeasureTimeBeanPostProcessor` отвечает за имплементацию логики засекания времени выполнения функций для классов, над которыми установлена аннотация `MeasureTime`. Он является имплементацией `BeanPostProcessor`, которые в Spring Framework нужны для донастройки созданных бинов на разных этапах инициализации приложения.

Функция `postProcessBeforeInitialization` выполняется первой, и объект, который передаётся в качестве аргумента к этой фунции, является оригинальным бином (а не Proxy-объектом). Вторым аргументом фунции является название (уникальный идентификатор) бина. С помощью него мы сможем найти оригинальное описание класса, из которого был создан оригинальный бин. Записываем название бина и информацию о его типе в словарь.

Функция `postProcessBeforeInitialization` выполняется второй. Объект, получаемый в качестве аргумента, уже необязательно является оригинальным бином, а может являться Proxy-объектом, созданным другим `BeanPostProcessor`. В свою очередь, Proxy-объекты могут не содержать всю нужную информацию, в частности, информацию об аннотациях, установленных над классом, его методами и полями. Получаем информацию об оригинальном классе из словаря по имени бина. Чтобы замерить время выполнения метода, создадим Proxy-объект, и зададим поведение для каждого метода. При вызове каждого метода будем получать текущее системное время в миллисекундах, затем выполнять оригинальный метод проксируемого объекта, и после этого снова получать текущее системное время. Вычтя из второго первое, получим время выполнения метода. Запишем результат выполнения в словарь для последующей обработки, и также выведем в стандартный вывод (консоль).

3.3.6. Модуль test

На рис.3.2 изображено дерево пакетов модуля tests, содержащего исходные коды тестов, использующихся для автоматизированной проверки работоспособности приложения в целом, а также его компонентов по отдельности.

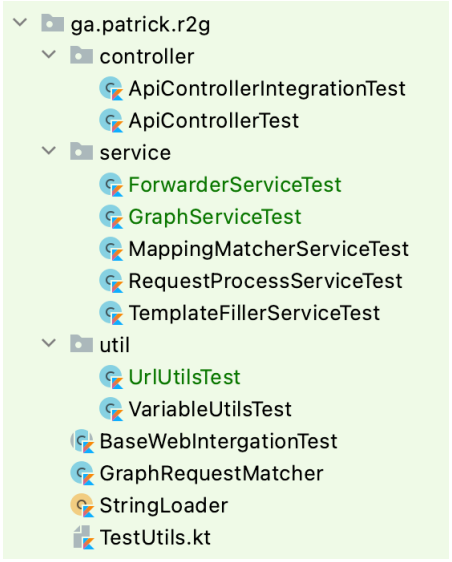

Рис.3.2. Дерево пакетов модуля test

Рассмотрим тестовые классы более подробно:

3.4. Выводы

Параграф с изложением выводов по главе *является обязательным*.

ГЛАВА 4. ТЕСТИРОВАНИЕ РЕАЛИЗАЦИИ

Хорошим стилем является наличие введения к главе, которое *начинается непосредственно после названия главы, без оформления в виде отдельного параграфа*.

4.1. Данные для тестирования реализации

Для тестирования сервиса R2G планируется использовать публичные GraphQL-сервисы. В частности, сервис FakeQL позволяет создать GraphQL-сервис на основании данных в виде JSON. Для этого используем JSON, представленный

в приложении 5. Данный JSON загружается на сайт FakeQL, после чего мы получаем URL эндпоинта, к которому можно отправлять GraphQL-запросы. Схема получившегося сервиса была продемонстрирована ранее (см. приложение 1).

В приложении 6 продемонстрирован пример маппинга. для получения списка счетов в валюте из query parameter, для пользователя с идентификатором, переданным в URI. В приложении 7 представлен пример запроса, который соответствует указанному маппингу, а также GraphQL-запрос, который должен быть отправлен в GraphQL-сервер при получении такого запроса.

4.2. Функциональное тестирование

Для проверки работоспособности были использованы тестовые данные, подготовленные в главе 2. Проверяются следующие сценарии:

- Для запроса существует маппинг, не содержащий переменных. Система отправляет GraphQL-сервису шаблон из маппинга в неизменном виде и возвращает пользователю ответ.
- Для запроса существует маппинг, содержащий несколько переменных. Система заполняет шаблон этими переменными и отправляет результат GraphQL-сервису, возвращает пользователю ответ.
- Для запроса не существует маппинг. Запрос в неизменном виде перенаправляется по адресу Gateway API, указанному в настройках. В данном случае использовался сервис Postman Echo, возвращающий body и headers запроса в качестве ответа. Ответ этого сервиса возвращается пользователю.

Все сценарии были успешно проверены.

4.3. Тестирование производительности

Так как данный адаптер будет обрабатывать каждый запрос в системе, то среднее время отклика (latency) системы увеличится на величину, равную среднему времени на получение, обработку и отправку запроса разрабатываемой системой (overhead). Измерим это время для разных сценариев. Так как время ответа сервера FakeQL непостоянно, то замер общего времени выполнения запроса не будет показательным. Чтобы определить overhead разработанного адаптера, необходимо замерить время выполнения в нескольких срезах.

Для каждого тестового запроса будем производить замеры в 8 моментах времени:

A. при отправке запроса из клиента, в качестве которого будем использовать Postman;
B. при получении запроса Spring Framework;
C. при передаче управления в компонент RequestProcessService;
D. при отправке запроса из GraphService
E. при получении ответа в GraphService
F. при возврате обработанного ответа из RequestProcessService;
G. при отправке результата клиенту;
H. при получении результата клиентом Postman.

Соответственно, мы получаем четыре среза, на основании которых сможем сделать выводы о производительности приложения, а также рассмотреть возможные пути повышения его скорости.

Таким образом, мы сможем высчитать overhead при предобработке запроса Spring Framework, так также время логики нашего адаптера. Все отправляемые в данном случае запросы абсолютно идентичны, что позволит оценить влияние каких-либо случайных факторов на время выполнения каждого этапа работы адаптера. Результаты замеров изображены в табл.4.1, значения представлены в миллисекундах.

Таблица 4.1

Результаты замеров

№ запроса	1-8	2-7	3-6	4-5
1	734	685	673	669
2	162	154	148	147
3	184	182	179	178
4	415	396	387	385
5	648	644	638	638

Соответственно, в столбце **1-8** время с момента отправки запроса из Postman до момента получения ответа. В столбце **2-7** время с момента получения запроса фреймворком до момента отправки им ответа. В столбце **3-6** время с момента начала обработки запроса нашим кодом до момента возвращения результата фреймворку для отправки клиенту. В столбце **4-5** время с момента отправки запроса в GraphQL-сервис до момента получения ответа от него.

На табл.4.2 посчитана разницу между каждой парой соседних столбцов, в миллисекундах.

Таблица 4.2

Результаты замеров

№ запроса	сеть	фреймворк	наша логика
1	35	12	4
2	8	6	1
3	2	3	1
4	19	9	2
5	4	6	0

В столбце **сеть** посчитано общее время на передачу запроса от клиента к адаптеру. Среднее значение составило 13.6 мс.

В столбце **фреймворк** указано время, потраченное Spring Framework на преобразование HTTP-запроса и его предобработки для передачи нашему коду. Среднее значение составляет 7.2 мс.

В столбце **наша логика** указано время, потраченное на поиск маппинга, подготовку запроса из шаблона, а также на обработку ответа от GraphQL-сервера. Среднее значение составляет 1.6 мс.

Как видно, время работы бизнес-логики при обработке запроса (1.6 мс) – поиска маппинга, заполнения запроса, отправки его в GraphQL-сервер и постобработки результата запроса – является относительно незначительным по сравнению с другими. Данные в этом столбце на самом деле не являются показательными – время передачи данных между клиентом и адаптером зависит от скорости интернет-соединения.

4.4. Возможности улучшения производительности

Рассмотрим способы улучшения показателей скорости работы адаптера:

– **Время обработки зарпоса фреймворком**. Данный показатель можно улучшить, используя вместо Spring Framework более легковесные фреймворки, например KTor или Netty, и неблокирующий ввод/вывод. Также, в качестве веб-сервера вместо Tomcat может использоваться более оптимальный Jetty.

– **Время обработки запроса нашим кодом**. Это время может значительно возрасти при увеличении количества маппингов, так как для каждого запроса осуществляется поочерёдное сравнение пути запроса с путём в маппинге с помощью регулярных выражений. То есть с ростом количества маппингов будет линейно расти количество необходимых сравнений для нахождения нужного. Таким образом, поиск маппингов можно считать "узким горлышком"нашей логики. Улучшить это время можно за счёт использования иных структур данных и логики поиска для хранения маппингов. Например, маппинги можно разделить на группы по HTTP-методу, сузив область для поиска до нескольких раз. А с помощью префиксного дерева можно находить среди них маппинги по пути (`path`) с логарифмической сложностью.

4.5. Выводы

Таким образом, .

ЗАКЛЮЧЕНИЕ

В процессе выполнения научно-исследовательской работы были изучены принципы технологий REST и GraphQL, определены их преимущества и недостатки, произведено сравнение между собой. Смоделирован процесс миграции между данными технологиями и определены проблемы, возникающие при этом.

На основании полученных данных сформулированы принципы, лежащие в основе системы, которая должна стать альтернативой использованию каждой из указанных технологий в отдельности, а также упростить осуществлении миграции.

Для описанной системы определены её предполагаемые преимущества и недостатки, на основании чего принято решение о целесообразности реализации данной системы в процессе выполнения выпускной квалификационной работы. Для этого было сформулировано техническое задание, включающее в себя требования к готовой системе, а также подготовлен пример тестовых данных, которые могут быть использованы для проверки работоспособности готовой системы.

По основным требованиям, сформулированным в техническом задании, был реализован и проверен на тестовых данных прототип веб-сервиса, который ляжет в основу готовой системы.

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

DOI Digital Object Identifier.

СЛОВАРЬ ТЕРМИНОВ

REST — архитектурный стиль построения взаимодействия между клиентом и сервисом.

SOAP — протокол удалённого вызова процедур, передачи сообщений между системами в формате XML.

GraphQL — язык запросов к серверу, использующий строгий синтаксис и схемы данных.

Фронтенд (frontend) — клиентская часть системы, с которой напрямую осуществляет взаимодействие конечный пользователь. Им может являться например мобильное, десктопное или веб-приложение.

Бэкенд (backend) — серверная часть системы, обычно представляет собой набор сервисов.

Сервис — серверное приложение, обрабатывающее запросы клиента.

Эндпоинт (endpoint) — одна из, или единственная точка входа для запросов к сервису.

Маппинг (mapping) — описание соответствия между двумя сущностями.

Бин (bean) — объект, который управляется по принципу инверсии контроля (Inversion of Control) неким IoC-контейнером. В данной работе роль IoC-контейнера выполняет Spring Framework.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Приложение 1

Краткие инструкции по настройке издательской системы LaTeX

Приложение 2

Приложение 2

Некоторые дополнительные примеры

Приложение 2