Пример ВКР «SPbPU-student-thesis-template». Версия от 15 мая 2021 г.

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Название института

Работа допущена к защите
Должность руководителя М
_____ И.О. Фамилия
«_____» _____ 202X г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
РАБОТА БАКАЛАВРА
ТЕМА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

по направлению подготовки XX.XX.XX Наименование направления подготовки
Направленность (профиль) XX.XX.XX_YY Наименование направленности (профиля) образовательной программы

Выполнил
студент гр. N — И.О. Фамилия

Руководитель
должность,
степень, звание[1] — И.О. Фамилия

Консультант[2]
должность, степень — И.О. Фамилия

Консультант
по нормоконтролю[3] — И.О. Фамилия

Санкт-Петербург
202X

[1]Должность указывают сокращенно, учёную степень и звание — при наличии, а подразделения — аббревиатурами. «СПбПУ» и аббревиатуры институтов не добавляют.

[2]Оформляется по решению руководителя ОП или подразделения. Только 1 категория: «Консультант». В исключительных случаях можно указать «Научный консультант» (должен иметь степень). Без печати и заверения подписи.

[3]Обязателен, из числа ППС по решению руководителя ОП или подразделения. Должность и степень не указываются. Сведения помещаются в последнюю строчку по порядку. Рецензенты не указываются.

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Название института

УТВЕРЖДАЮ
Должность руководителя М
_____ И.О. Фамилия
«_____» _____ 202Хг.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы

студенту Фамилия Имя Отчество гр. N

1. Тема работы: Тема выпускной квалификационной работы.
2. Срок сдачи студентом законченной работы[4]: дд.мм.202Х.
3. Исходные данные по работе[5]: статистические данные с сайта [**gosstat**], а также из репозитория [**uci**]; основным источником литературы является монография [**Book**] и статья [**Article**].
4. Содержание работы (перечень подлежащих разработке вопросов):
   4.1. Обзор литературы по теме ВКР.
   4.2. Исследование программных продуктов.
   4.3. Разработка метода/алгоритма/программы.
   4.4. Апробация разработанного метода/алгоритма/программы.
5. Перечень графического материала (с указанием обязательных чертежей):
   5.1. Схема работы метода/алгоритма.
   5.2. Архитектура разработанной программы/библиотеки.
6. Консультанты по работе[6]:
   6.1. Должность, степень, И.О. Фамилия.
   6.2. Должность, степень, И.О. Фамилия (нормоконтроль).

[4]Определяется руководителем ОП, но не позднее последнего числа преддипломной практики и/или не позднее, чем за 20 дней до защиты в силу п. 6.1. «Порядка обеспечения самостоятельности выполнения письменных работ и проверки письменных работ на объем заимствований».

[5]Текст, который подчёркнут и/или выделен в отдельные элементы нумерационного списка, приведён в качестве примера.

[6]Подпись консультанта по нормоконтролю пока не требуется. Назначается всем по умолчанию.

7. Дата выдачи задания[7]: дд.мм.202X.

Руководитель ВКР _____И.О. Фамилия

Консультант[8] _____И.О. Фамилия

Задание принял к исполнению дд.мм.202X

Студент _____И.О. Фамилия

[7]Не позднее 3 месяцев до защиты (утверждение тем ВКР по университету) или первого числа преддипломной практики или по решению руководителя ОП или подразделения (открытый вопрос).

[8]В случае, если есть консультант, отличный от консультанта по нормоконтролю.

РЕФЕРАТ

На 28 с., 0 рисунков, 0 таблиц, 0 приложений

КЛЮЧЕВЫЕ СЛОВА: СТИЛЕВОЕ ОФОРМЛЕНИЕ САЙТА, УПРАВЛЕНИЕ КОНТЕНТОМ, PHP, MYSQL, АРХИТЕКТУРА СИСТЕМЫ.[9]

Тема выпускной квалификационной работы: «Тема выпускной квалификационной работы»[10].

В данной работе изложена сущность подхода к созданию динамического информационного портала на основе использования открытых технологий Apache, MySQL и PHP. Даны общие понятия и классификация IT-систем такого класса. Проведен анализ систем-прототипов. Изучена технология создания указанного класса информационных систем. Разработана конкретная программная реализация динамического информационного портала на примере портала выбранной тематики...[11]

В данной работе изложена сущность подхода к созданию динамического информационного портала на основе использования открытых технологий Apache, MySQL и PHP. Даны общие понятия и классификация IT-систем такого класса. Проведен анализ систем-прототипов. Изучена технология создания указанного класса информационных систем. Разработана конкретная программная реализация динамического информационного портала на примере портала выбранной тематики...

ABSTRACT

28 pages, 0 figures, 0 tables, 0 appendices

KEYWORDS: STYLE REGISTRATION, CONTENT MANAGEMENT, PHP, MYSQL, SYSTEM ARCHITECTURE.

---

[9]Всего **слов**: от 3 до 15. Всего **слов и словосочетаний**: от 3 до 5. Оформляются в именительном падеже множественного числа (или в единственном числе, если нет другой формы), оформленных по правилам русского языка. *Внимание! Размещение сноски после точки является примером как запрещено оформлять сноски.*

[10]Реферат **должен содержать**: предмет, тему, цель ВКР; метод или методологию проведения ВКР: результаты ВКР: область применения результатов ВКР; выводы.

[11]ОТ 1000 ДО 1500 печатных знаков (ГОСТ Р 7.0.99-2018 СИБИД) на русский или английский текст. Текст реферата повторён дважды на русском и английском языке для демонстрации подхода к нумерации страниц.

The subject of the graduate qualification work is «Title of the thesis».

In the given work the essence of the approach to creation of a dynamic information portal on the basis of use of open technologies Apache, MySQL and PHP is stated. The general concepts and classification of IT-systems of such class are given. The analysis of systems-prototypes is lead. The technology of creation of the specified class of information systems is investigated. Concrete program realization of a dynamic information portal on an example of a portal of the chosen subjects is developed...

In the given work the essence of the approach to creation of a dynamic information portal on the basis of use of open technologies Apache, MySQL and PHP is stated. The general concepts and classification of IT-systems of such class are given. The analysis of systems-prototypes is lead. The technology of creation of the specified class of information systems is investigated. Concrete program realization of a dynamic information portal on an example of a portal of the chosen subjects is developed...

СОДЕРЖАНИЕ

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.

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. 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.

1.5.1. 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.
- 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.

1.5.2. 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.

1.6. Выводы

Параграф с изложением выводов по главе *является обязательным*.

ГЛАВА 2. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ

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

2.1. 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.

2.2. 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 starters, которые включают в себя другие модули и упрощают их конфигурирование.
- Spring Web используется для создания эндпоинтов для обработки входящих REST-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 (легковесных потоков) для повышения производительности и упрощения использования реактивного подхода.

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.

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, а также одной из немногих подобных технологий имеет реализацию в виде библиотеки для Java.

2.3. 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.

2.4. Выводы

Параграф с изложением выводов по главе *является обязательным*.

ГЛАВА 3. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ СЕРВИСА

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

3.1. Требования к прототипу

К прототипу предъявляются не все из требований, перечисленных в требованиях к реализации. В частности, прототип:

- загружает маппинги из настроек, расположенных локально в файле application.yml;
- при перенаправлении запроса к другому эндпоинту обрабатывает только успешные ответы от этого эндпоинта;
- не осуществляет трансформирование ответа в формате JSON, полученного от GraphQL-сервиса;
- допускает любое поведение при наличии нескольких подходящих маппингов.

3.2. Реализация прототипа

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.

3.2.1. Пакет controller

Данный пакет включает в себя классы, ответственные за принятие запроса к сервису и отправку ответа. В прототипе этот пакет имеет единственный класс ApiController, имеющий единственный метод endpoint, который принимает объект запроса и передаёт его в RequestProcessService, который будет рассмотрен далее.

3.2.2. Пакет service

Данный пакет включает в себя классы, ответственные за так называемую бизнес-логику и содержит следующие классы:

- MappingMatcherService – получает метод и uri полученного запроса и находит для него маппинг.
- TemplateFillerService – получает значения переменных из URI, query parameters и body и использует их для заполнения шаблона GraphQL–запроса.
- ForwarderService – ответственнен за перенаправление запроса в Gateway API с сохранением URI, headers, query parameters и body.
- RequestProcessService получает объект запроса и вызывает MappingMatcherService для поиска маппинга. Если маппинг не был найден, то он передаёт объект запроса в ForwarderService и возвращает ответ, полученный от Gateway API. Если маппинг был найден, то он использует TemplateFillerService для создания GraphQL-запроса, выполняет его с помощью GraphClient, который будет рассмотрен далее, и возвращает полученный ответ пользователю.

3.2.3. 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.

3.2.4. Пакет property

Данный пакет включает в себя класс MappingProperties, содержащий маппинги, загруженные из файла application.yml.

3.2.5. Пакет client

Данный пакет включает в себя единственный класс GraphClient, который содержит единственный метод send, который вызывает GraphQL-сервис по переданному адресу с переданным тестом запроса. Запросы отправляются с помощью технологии WebClient, выполняющей запросы с использованием реактивного (неблокирующего) подхода.

3.2.6. Модуль test

Данный модуль содержит в себе юнит-тесты для проверки работы ряда сервисов, а также несколько интеграционных сервисов, проверяющих работоспособность системы вцелом. Интеграционные тесты утилизируют тестовые данные, подготовленные в разделе 2.3, причём вместо вызова внешних систем реализуются заглушки с помощью технологии WireMock, позволяющей указать ожидаемый запрос и соответствующий ему ответ.

3.3. 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. Название параграфа

Для проверки работоспособности прототипа были использованы тестовые данные, подготовленные в разделе 2.3. 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, в результате чего прототип считается работоспособным и будет использован для реализации целевой системы.

4.2. Выводы

Текст выводов по главе 4.

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.

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.

— .

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ