СОДЕРЖАНИЕ

[ГЛОССАРИЙ 2](#_Toc133920771)

[АННОТАЦИЯ 3](#_Toc133920772)

[ВВЕДЕНИЕ 4](#_Toc133920773)

[I. РОЛЬ И РАЗВИТИЕ АРХИТЕКТУРЫ ПО 6](#_Toc133920774)

[1.1 Роль архитектора в разработке программного обеспечения 6](#_Toc133920775)

[1.2 Эволюция архитектуры ПО 8](#_Toc133920776)

[1.3 Преимущества и недостатки микросервисной архитектуры 11](#_Toc133920777)

[II. МИКРОСЕРВИСНАЯ АРХИТЕКТУРА 13](#_Toc133920778)

[2.1 Моделирование сервисов 13](#_Toc133920779)

[2.2 Типы связей 15](#_Toc133920780)

[**2.2.1 Domain coupling (связь по доменной области)** 15](#_Toc133920781)

[**2.2.2 Pass-Through Coupling (промежуточная связь)** 16](#_Toc133920782)

[**2.2.3 Common coupling (общая связь)** 18](#_Toc133920783)

[**2.2.4 Content Coupling (связь контента)** 19](#_Toc133920784)

[2.3 Domain Driven Design 20](#_Toc133920785)

[**2.3.1 Понятный язык** 21](#_Toc133920786)

[**2.3.2 Агрегаты** 21](#_Toc133920787)

[**2.3.3 Ограниченный контекст** 22](#_Toc133920788)

[2.4 Разбиение монолитного приложения 23](#_Toc133920789)

[**2.4.1 Инкрементальная миграция** 24](#_Toc133920790)

[**2.4.2 Strangler Fig** 24](#_Toc133920791)

[**2.4.3 Branch by Abstraction** 25](#_Toc133920792)

[**2.4.4 Parallel Run** 26](#_Toc133920793)

[**2.4.5 Нюансы декомпозиции данных** 27](#_Toc133920794)

[2.5 Способы коммуникации микросервисов 29](#_Toc133920795)

[**2.5.1 Синхронные запросы** 29](#_Toc133920796)

[**2.5.2 Асинхронные запросы** 30](#_Toc133920797)

[**2.5.3 Коммуникация используя общие данные** 31](#_Toc133920798)

[**2.5.4 Коммуникация запрос-ответ** 32](#_Toc133920799)

[**2.5.5 Событийно ориентированная коммуникация** 33](#_Toc133920800)

[**2.5.6 Обработка ошибок** 35](#_Toc133920801)

[III. МИГРАЦИЯ ПРИЛОЖЕНИЯ ''МЕНЕДЖЕР ЗАДАЧ'' НА МИКРОСЕРВИСНУЮ АРХИТЕКТУРУ 36](#_Toc133920802)

[3.1 Обзор изначального приложения 36](#_Toc133920803)

[3.2 Подготовка инфраструктуры 38](#_Toc133920804)

[**3.2.1 Service Registry** 39](#_Toc133920805)

[**3.2.2 API gateway** 40](#_Toc133920806)

[3.3 Миграция доменов на микросервисную архитектуру 42](#_Toc133920807)

[**3.3.1 Миграция профиля пользователя в Identity/Profile микросервис** 43](#_Toc133920808)

[**3.3.2 Создание микросервиса Project Service** 43](#_Toc133920809)

[**3.3.3 Миграция микросервиса Sprint Service** 44](#_Toc133920810)

[**3.3.4 Миграция микросервиса Item Service** 45](#_Toc133920811)

[3.4 Демонстрация работы приложения 46](#_Toc133920812)

[3.5 Обзор структуры полученного приложения 50](#_Toc133920813)

[ЗАКЛЮЧЕНИЕ 51](#_Toc133920814)

[БИБЛИОГРАФИЯ 52](#_Toc133920815)

# **ГЛОССАРИЙ**

DDD Domain Driven Design

ПО программное обеспечение

БД база данных

ACID atomicity, consistency, isolation, durability

SQL structured query language

HTTP hypertext transfer protocol

JMS java message service

API application programming interface

IP internet protocol

CRUD create, read, update, delete

# **АННОТАЦИЯ**

Дипломная работа на тему: “Миграция монолитного приложения на микросервисную архитектуру”.

Автор: Евгений Паненко Александрович, студент 5 курса факультета Математики и Информатики, Прикладная информатика.

Работа состоит из введения, 3 глав, заключения, списка использованной литературы и источников.

Объект исследования – микросервисная архитектура

Цель – изучение проектирования приложения с использованием микросервисной архитектуры и последующая миграция монолитного приложения на изученный тип архитектуры.

В работе были рассмотрены практики, используемые при построении и миграции на микросервисную архитектуру.

В результате изученной теории была произведена миграция приложения “Менеджер задач” на микросервисную архитектуру.

# **ВВЕДЕНИЕ**

**Актуальность и важность объекта изучения.** С ростом количества пользователей интернета нагрузка на продукты и ресурсы, размещенные в интернете, увеличивается, бросая вызов существующим системам. Для адаптации к новым требованиям архитектура программных продуктов адаптируется и постепенно вводит новые методы проектирования. Микросервисная архитектура является одним из самых популярных видом архитектуры на данный момент, преследуя достижения следующих целей:

* Компоненты системы легко поддаются масштабированию
* Изоляция компонентов, предотвращение каскадных сбоев
* Разнообразие в выборе технологий для различных задач
* Облегчение развертывания
* Ранний выход на рынок
* Улучшенная безопасность
* Возможность распределения обязанностей между различными командами

Благодаря особенностям построение систем следуя микросервисной архитектуре, команды и компании способны извлекать вышеперечисленные преимущества, которые в последствии могут определить успех продукта на рынке.

**Целью работы** является изучение микросервисного подхода при реализации продукта и применении теоретических знаний на практике при миграции существующего монолитного приложения на микросервисную архитектуру.

Подцелями данной работы являются:

* Изучение истории развития архитектуры программного обеспечения
* Изучение преимуществ и недостатков микросервисного подхода
* Ознакомление с теорией определения границ микросервисов
* Изучение типов коммуникаций
* Рассмотрение основных паттернов используемых при миграции монолитных приложений на микросервисную архитектуру

**Ожидаемым результатом** является освоение вышеупомянутого метода проектированияи применение полученных знаний для миграции существующих приложений на микросервисную архитектуру.

В качестве методической базы для написания работы использовались следующие ресурсы:

* Teze de master, de licență și anuale. Indicații metodice [1],
* Ghid metodic pentru perfectarea tezelor de licență/master [2].

Основными источниками информации при написании работы являлись:

* Книжные источники [3, 8]
* Интернет-источники [4-7, 9, 10]
* Документации фреймворка [11, 12]

Работы состоит из введения, трех глав, заключения и библиографии.

В первой главе “Роль и развитие архитектуры ПО” рассматривается роль архитектора, история развития архитектуры ПО, а также преимущества и недостатки использования микросервисной архитектуры.

Во второй главе “Микросервисная архитектура” рассматриваются составляющие проектирования микросервисной архитектуры, а именно:

* Основы моделирования микросервисов
* Паттерны, используемые при миграции приложений на микросервисную архитектуру
* Проблемы требующие внимания при проектировании и обзор DDD
* Типы коммуникаций микросервисов

В третьей главе “Миграция приложения ''менеджер задач'' на микросервисную архитектуру” выполнена миграция монолитного приложения на микросервисную архитектуру.

Работы изложена на 52 страницах, включает в себя 28 иллюстраций, глоссарий с использованными аббревиатурами и библиографией, включающей 12 источников.

**I. РОЛЬ И РАЗВИТИЕ АРХИТЕКТУРЫ ПО**

## **1.1 Роль архитектора в разработке программного обеспечения**

Индустрия разработки программного обеспечения молодая относительно других отраслей. Разработке программного обеспечения примерно 70 лет и многие термины берутся из других профессий. Одним из таких терминов является ‘'архитектура’' или ‘'архитектор’', но является ли роль архитектора программного обеспечения ответвлением или развитием древней профессией благодаря которой спроектированы наши жилища. Роль архитектора в реальном мире предполагает высокий уровень квалификации и возможность строить постройки с высокой уверенностью в их надежности, что нельзя сказать о программном обеспечении, где даже самые надежные сервисы, в которые были вложены огромные средства и усилия не защищены абсолютной отказоустойчивостью. Так что же на самом деле означает термин, который программисты украли из другой отрасли?

Saw Newman в своей книге “Building Microservices” (Построение микросервисов) проводит другую аналогию. С ним этой идеей поделился Erik Doernenburg, который считает что роль архитектора в ИТ больше похожа на градостроителя. Роль градостроителя изучать множественные источники информации и далее стараться оптимизировать планировку города, который будет наиболее удобен жителям, беря во внимание также будущее развитие города. Он не говорит где построить конкретное здание, а зонирует город и предоставляет выбор какое именно здание должно быть построено другим людям. Вместо того, чтобы волноваться о том что именно будет построено в одной определенной зоне, он в большей степени занят работой над синергией между зонами.

Сравнение с программами должно быть очевидным. Пользователи нашего приложения являются жителями города и со временем город должен меняться, чтобы покрывать новые потребности. Эти изменения не всегда можно предусмотреть заранее, но роль архитектора заключается в том, чтобы организовать процессы таким образом, что изменения не принесут больших неприятностей. С одной стороны архитектор заботится об удобстве и счастии пользователей, но не должен забывать предоставить разработчикам и людям поддерживающим эту систему хорошие условия для выполнения их работы.

Одним из важных этапов в построении микро сервисной архитектуры является зонирование. С точки зрения архитектуры или архитектора фокус должен быть направлен не на детали как работает один определённый сектор, а на то как они общаются друг с другом. Каждая такая зона может быть как одним микро сервисом так и другим проектом, который в свою очередь также разбивается на свои зоны. На данном этапе также могут быть приняты решения на счет используемых технологий в будущем.

Другим важным аспектом могут быть выбор технологий. Хотим ли мы использовать технологию, которая лучше показывает себя в данной задаче но с которой не знакомы разработчики или лучше использовать более знакомый инструментарий несмотря на его уступающую производительность. Выбор, который будет сделан зависит от множества факторов и архитектор берет на себя ответственность определяя границы используемых технологий. Чаще всего нет абсолютно правильного решение и оно будет всегда компромиссное. Например, выбирая малое количество знакомых технологий мы унифицируем решения, делая его очень комфортных для разработчиков, несмотря на то что могут пострадать другие показатели как масштабируемость.

Также на плечи архитектора ПО может лечь задание принципов и практик которые будут преследоваться на проекте. Такие принципы и практики помогут проекту добиваться будущих целей. Например, в определенный момент бизнес выдвинул новое требование по уменьшение времени требуемого на выпуск новой функциональности. В этом случае может быть введен новый принцип, что команда разработки будут иметь полный контроль в ходе разработки и новая функциональность будет публиковаться сразу же как она готова, не дожидаясь важного релиза. Практики помогают нам производить более качественный продукт, как пример это может быть рекомендация по использованию HTTP/REST подхода в интеграции разных частей системы или минимально допустимое покрытие кода тестами.

Обобщая вышесказанное, архитектор - это лицо ответственное за целостность общей картины, которое задает стандарты, обеспечивающие дальнейшее развитие системы с учетом соблюдения достойного качества.

## **1.2 Эволюция архитектуры ПО**

Несмотря на то, что сфера ИТ относительна молода и она успела пройти ряд эволюционных этапов. Каждый такой этап привнес новые практики и методы того, как может быть организован код и как компоненты могут взаимодействовать друг с другом. В этой главе мы пройдемся по основным этапам и нововведениям в архитектуре ПО.

Все началось с эры мейнфреймов в 1960 году, когда комплектующие были безумно дорогие и занимали огромные размеры. Эти машины были мощными серверами способными обрабатывать большие объемы инструкций. Со временем комплектующие начали становится более производительными и доступными, терминалы, которые ранее использовались для послания команд мэйнфреймам заменяются десктопными машинами. Эти терминалы сами по себе имеют неплохую производительность и способны выполнять инструкции самостоятельно. Это привело к появлению клиент серверной архитектуре, при большинство кода связанного с клиентом производится на его стороне, а сервера используются для хранения информации и обработки ресурсоемких процессов. В это время впервые возникли испытания связанные с установкой клиентского приложения  перед получением доступа к серверу на различные машины.

Далее началась эра приложений, доступ к которым осуществляется с использованием веб браузера, без необходимости устанавливать отдельный клиент для каждого сервера. В данном случае веб сервер играл роль приложения, которое отвечает за обработку всех входящих запросов от пользователей. Это изменение позволило решить проблему с установкой клиентов на каждую машину и теперь сервер мог быть доступен с любой машины, если человеку известен домен приложения. На данном этапе сервера отвечали за все операции, начиная с предоставления и генерации html кода, применение бизнес логики, а также доступа к данным. Таким образом эти 3 домена в дальнейшем были размечены как слои из которых состоит приложение и такая слоеная архитектура часто упоминается как MVC (Model-View-Controller). Model в этой архитектуре отвечает за объекты и данные ассоциируемые с ними, View это слой отвечающий за показ данных пользователю, а Controller за обработку запросов к серверу и выдаче ответов клиенту.

С развитием интернета в мире количество пользователей увеличилось в разы и старые приложения начали не справляться нагрузкой. Вначале для того, чтобы обрабатывать большее количество запросов происходило вертикальное масштабирование. Суть такого масштабирования заключалась в увеличении мощностей серверов, доступного объема памяти и ресурсов. Это помогло на время, но вскоре проблему уже было не возможно решить увеличением мощности сервера. Решением проблемы на этот раз стало горизонтальное масштабирование. Основная идея заключается в запуске нескольких экземпляров одного и того же приложения, между которыми будет распределена нагрузка.

Следующим препятствием к наибольшей производительности стали реляционные базы данных, которые было крайне сложно масштабировать. Приложения выполняющие множество запросов с базой данных (такие как системы по обработке банковских транзакций), стало причиной высокой задержки при выполнении запросов. Для того, чтобы решить данную проблему было предпринято многих действий, которые в основном относились к вертикальному масштабированию - использование кэша, оптимизация запросов с использованием индексов, репликация данных для создания реплик исходной базы данных, поддерживающей только чтение информации. Некоторые специалисты начали верить в то, что реляционные базы данных не приспособлены к обработке больших объемов данных из за жестких связей и повышенной нагрузке при обработке транзакций. Итогом стало разработка нереляционных баз данных, в которых данные хранятся в гибкой структуре, имеющей форму документов. Основной идеей было избежать связей между таблицами и хранить все необходимые данные в одном документе. Это с одной стороны решило проблему с масштабированием, но в то же время привнесло новые испытания, связанные с синхронизацией данных в случае обновление одного и того же документа в различных экземплярах БД.

Несмотря на то, что горизонтальное масштабирование слоенного приложения помогло решить проблему с производительностью, данный подход не был идеальным, так как он требовал масштабировать огромное приложение целиком, включая те части которые не были высоко нагружены. Это способствовало идеи разбивать приложения на сервисы, каждый из которых будет ответственен за выполнение определенной задачи и сможет обмениваться информацией с другими сервисами по необходимости. Таким образом стало возможным масштабировать только высоко нагруженные части приложения, тем самым экономя ресурсы. Также такой подход позволил по необходимости использовать различные технологий для различных задач. Говоря о интеграции между такими сервисами, изначально использовалось соединение через БД, но потом они мигрировали на использование Enterprise Service Bus (ESB), которое позволило подключить все разношерстные системы к единому каналу сообщения. Также это способствовало унификации способов аутентификации, безопасности и авторизации среди всей организации.

Благодаря успехам сервисно ориентированных систем, дальнейшее их развитие было очевидным. Дальнейшем развитием было не только разбить приложение на различные сервисы, а также разбить эти сервисы на подсервисы, каждый из которых будет отвечать за свою часть бизнес логики, а также может иметь свою собственную базу данных. Входной точкой для всех сервисов стал API gateway - микросервис через который будут проходить все запросы и он будет отвечать за аутентификацию и авторизацию.

Следующим ключевым моментом стало развитие инструментов для масштабирования микросервисов, таких как Kubernetes и развитие облачных провайдеров, которые предоставляют сервера для развертывания системы. Благодаря этому, теперь нагрузку на приложение можно было контролировать и масштабировать отдельные сервисы только по необходимости, а в случае простоя, например в ночное время, количество запущенных сервисов можно было снижать, чтобы добиться меньшей стоимости системы.

В данном разделе были рассмотрены ключевые моменты в развитии архитектуры ПО от эры мейнфреймов до сегодняшнего дня, от больших монолитных приложений до микросервисной архитектуры, легко поддающиеся изменениям и способным выдерживать намного большие нагрузки. В следующем разделе рассмотрим преимущества использования микросервисной архитектуры подробнее.

## **1.3 Преимущества и недостатки микросервисной архитектуры**

Ранее в предыдущем разделе была уже затронута тема появления микросервисов, как и проблемы, которые в своем большинстве они должны были решить. В данной разделе рассмотрим подробнее преимущества и недостатки использования микросервисной архитектуры.

Первым и очевидным плюсом использования микросервисной архитектуры состоит в том, что так как приложение разбито на различные модули, каждый из этих модулей может использовать различные технологии. Таким образом для решения конкретной задачи может быть выбраны наиболее подходящие для этого технологии. С другой стороны, существует множество споров о том, что лучше использовать – технологию, которая может не быть наилучшим выбором для решения задачи, но разработчики имеют большой опыт в ее использовании или технология, которая менее известна разработчикам, но идеально подходит под решение задачи. Выбор зависит от множества факторов и всегда индивидуален, но как минимум используя микросервисную архитектуру, у разработчиков появляется такая опция.

Следующий плюс, как и остальные связан с тем, что в микросервисной архитектуре все модули становятся разбиты и выполняемый код не находится в одном месте, как при монолитной архитектуре. Благодаря этому достигается наиболее высокий уровень отказоустойчивости, так как если один из микросервисов упал, вся остальная система продолжает функционировать. По той же причине масштабирование намного лучше улучшает производительность системы, так как с микросервисной архитектурой можно масштабировать сервисы, которые берут на себя основную часть нагрузки, а те части приложения, которые не подвергаются высоким нагрузкам, могут иметь всего несколько копий на сервере. Это разница позволяет более рационально распределять ресурсы сервера в отличии от монолитных приложений, где при масштабировании придётся также и масштабировать части приложения, которые в этом не нуждаются.

Другим преимуществом, связанным с использованием микросервисной архитектуры, связан с облегчением работы разработчиков на таких проектах. Во-первых, из-за того, что система состоит из множества модулей, менеджеры могут разделить обязанности команд определенной группой микросервисов. Таким образом будет увеличена скорость разработки и внедрения новых функциональностей, так как разработчикам уже хорошо будут известны микросервисы над которыми им предстоит работать. В то же время будет упрощен релиз новых функциональностей, так как с микросервисной архитектурой, релиз будет выпущен для одного или группы микросервисов, а не для всего монолита целиком. Тем самым каждый из релизов будет нести меньше рисков, так как только часть системы будет подвержена изменениям.

Несмотря на преимущества, которые привносит микросервисная архитектура, она не без недостатков. Например, в зависимости от дизайна системы, разработчики могут испытывать трудности в разработке на локальной машине. В случае, если проект становится довольно большим и тестирования задачи локально требуется запустить 10–20 микросервисов, может возникнуть проблема с производительностью на локальной машине. Зачастую разработчики оказываются в ситуации, когда они не могут запустить систему локально и им приходится разворачивать приложение в облаке, чтобы была возможность протестировать новую функциональность или исследовать поведение бага.

Другие недостатки связаны с тем же фактором, который привносит и преимущества, как это иронично не звучало. Адаптация в новому методу работы, а также потребность в новых инструментах может увеличить стоимость проекта по крайней мере на первых шагах. Так как приложение разбито на множество микросервисов усложняется мониторинг и исправление ошибок. Теперь разработчику недостаточно запустить монолит в режиме отладки и выявить причину ошибки. Из-за того, что вместо вызова метода, в микросервисам необходимо общаться через сеть возникает дополнительная задержка для выполнения запроса, а также пересылаемые данные потенциально могут быть украдены злоумышленниками.

Несмотря на минусы и сложности, которые можно встретить при использовании микросервисной архитектуры, в далекой перспективе при построении больших и сложных систем, преимущества использования такого подхода перевешивают недостатки. С каждым годом разработчики получают все больше и больше опыта в использовании такого подхода и начали разрабатываться методы по минимизации недостатков. Выбор использовать данный подход или нет, зависит от множества факторов и индивидуален для каждого проекта в зависимости от сроков выхода на рынок и размеров, но можно сказать с уверенностью, что микросервисная архитектура становится популярнее среди разработчиков с каждым годом.

**II. МИКРОСЕРВИСНАЯ АРХИТЕКТУРА**

## **2.1 Моделирование сервисов**

Перед написанием множества микросервисов в спешке, первоначально необходимо тщательно проработать как они будут выглядеть. В конечном итоге микросервисы, которые планируеться создать должны легко поддаваться изменениям и возможности публиковать эти изменения не затрагивая или нарушая контрактов с другими микросервисами. Эта способность - изменяться независимо от других критически важна при построении микросервисов. Для достижения этой цели необходимо правильно задать границы микросервиса, в чем нам помогут три следующих концепта - сокрытие информации, связанность (coupling), сплоченность (cohesion).

Для начала поговорим о сокрытие информации. Этот концепт был разработан Дэвидом Парнасом (David Parnas) и описывает эффективный способ определять границы модуля. [1] Суть заключается в сокрытие максимально возможного количества деталей модулем. Для этого Парнас посмотрел на преимущества, которые модуль должен теоретически предоставлять:

* Уменьшать время на разработку - благодаря тому что модули могут разрабатываться параллельно, время затрачиваемое на разработку проекта может быть сокращено.
* Понятность - так как каждый модуль может быть рассмотрен отдельно от всей системы, бизнес логику скрывающиеся за ним проще понять.
* Гибкость - модули должны иметь возможность изменяться вне зависимости друг от друга.

Этот список первоначально предназначался для модулей, но так же отлично подходит для микро сервисной архитектуры, ведь если задуматься микросервисная архитектура это лишь иная форма модульной архитектуры.

Одно из других высказываний Парнаса гласит - *Связь между модулями - это предположения, которые они делают друг о друге.*

Уменьшая количество предположений, которые один модуль делает по отношению к другому, мы влияем на их связь. Чем меньше таких предположений, тем вероятнее модуль сможет изменяться не влияя на другие модули. Чем проще и понятнее интерфейс модуля и как он используется другими, тем вероятнее успешная работа программиста с ним.

Далее затронем немного термин сплоченности, который можно сформулировать следующим образом - код, который изменяется вместе также и находится вместе. Таким образом, если наше изменение требует всегда требует работы в нескольких модулях, то фактически можно считать, что мы работает над одной кодовой базой, просто разбитой по разным модулям. В реальности мы должны избегать таких ситуаций и строить архитектуру таким образом, чтобы изменение можно было производить воздействуя на минимальное количество модулей. При высокой сплоченности кода относящемуся к единой логике, нам не придется делать изменения во многих частях системы, что упростит развертывание и сократит время на разработку.

Последнее, но не менее важное это связанность компонентов. В проектировании микросервисов мы должны стремится к слабой связанности компонентов, чтобы как уже говорилось ранее изменения не затрагивали большое количество модулей. Это как правило достигается правильным выбором метода интеграции сервисов, а также уменьшение их взаимодействия. В случае, если два микро сервиса очень тесно общаются друг с другом это может в свою очередь привести к сильному связыванию компонентов, чего нам нужно избегать.

## **2.2 Типы связей**

Как уже говорилось ранее необходимо избегать ситуаций, когда один сервис сильно связан с другим. К сожалению в большинстве случаев это не будет достижимо в полной мере, но тем менее при проектировании мы должны стараться как можно больше уменьшить этот показатель. В данном разделе мы постараемся рассмотреть основные типы связей, которые существуют.

### **2.2.1 Domain coupling (связь по доменной области)**

Связь по доменной области представляет собой связь при которой одному микро сервису необходимо взаимодействовать с другим, так как ему необходим функционал второго. В качестве примера приведем следующую диаграмму (рис. 2.1). В данном случае имеет 3 сервиса - обработчик заказов, склад и платежную систему. Сервис обработки заказов в свою очередь зависит от друг других для того чтобы до конца выполнить заказ.



Рис. 2.1 – пример доменной связи

В микро сервисной архитектуре такая зависимость встречается повсеместно, так как суть микро сервисной архитектуры заключается во взаимодействии микро сервисов, тем не менее количество связей между сервисами должно быть сведено к минимуму. В случае, если один из сервисом начинает зависеть от множества других сервисов, возможно он выполняет больше обязанностей чем должен.

### **2.2.2 Pass-Through Coupling (промежуточная связь)**

Данный вид связи описывает ситуацию при которой один микро сервис передает данные другому микро сервису только потому что данные нужно еще одному другому микро сервису далее в цепочке вызовов. Примером такой связи может служить следующая структура (рис. 2.2). В данном случае имеем 3 микро сервиса - обработчик заказов, склад и сервис доставки. В нашем случае обработчик заказов отправляя запрос на склад, также включает информацию куда должен будет доставлен товар и каким способом.

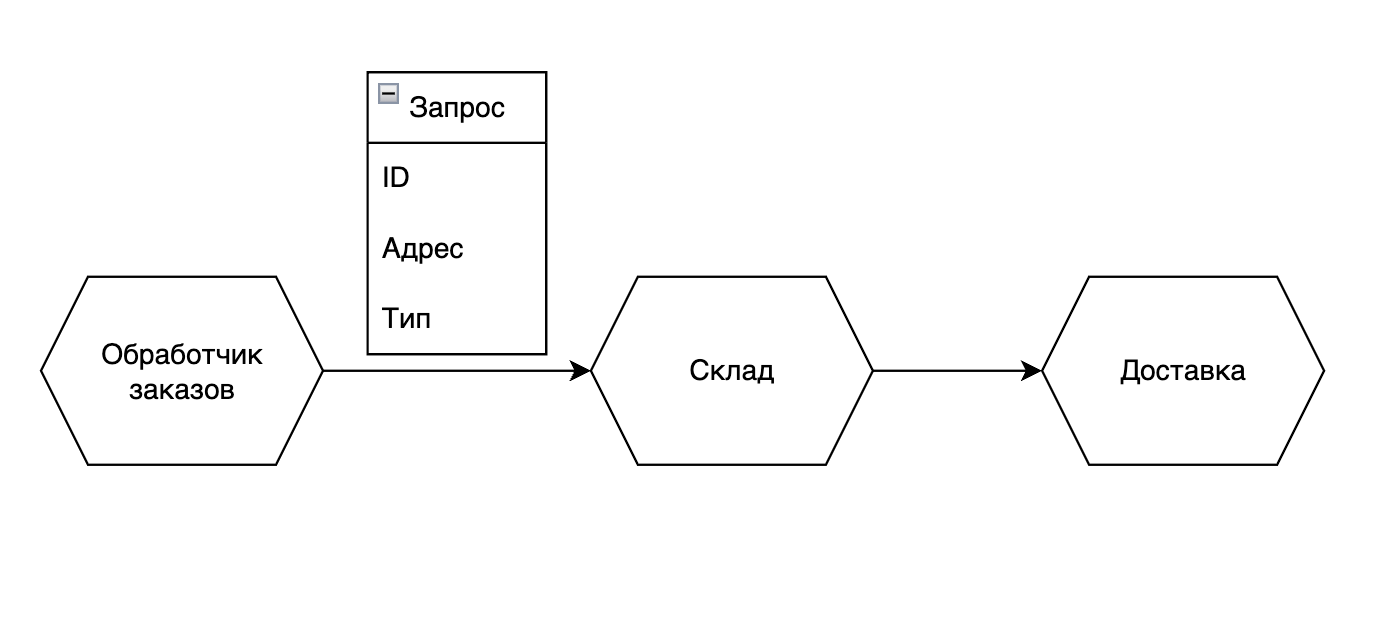


Рис. 2.2 – пример промежуточной связи

В данном примере наибольшей проблемой является то, что в случае если сервису Доставке потребуется изменить формат запроса, в сервисе склад будет необходимо также произвести изменения. Есть несколько способов решить данную проблему. Во первых необходимо пересмотреть действительно ли необходимо передавать данные сервису Доставки через сервис Склад или можно разделить эту цепочку на два запроса в каждый сервис по отдельности. (см. рис. 2.3)

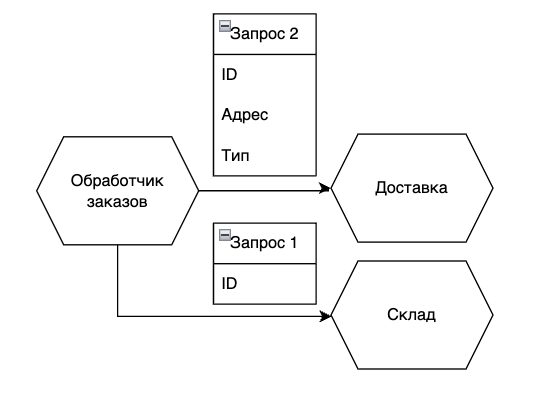


Рис. 2.3 – пример ликвидации промежуточной связи

На первый взгляд проблема решена, но, с другой стороны, наш сервис Обработки заказов увеличил количество связей с другими сервисами, то есть увеличилось количество доменных связей. Также отметим следующее, заказанный предмет должен быть зарезервирован на складе до его отправки. Ранее эта логика находилась в сервисе Склад, а теперь сервису Обработчик заказов придётся управлять этим процессом, что повышает количество логики в одном микросервисе, о которой раньше ему не приходилось заботиться.

Альтернативно можно прибегнуть к другому подходу. Структура будет выглядеть как она была изначально, за исключением того, что информация о доставке будет отослана сервису Склад в таком формате, что он просто ее отправит дальше даже не задумываясь что находиться внутри. Таким образом в случае, если сервис Доставки потребует сделать изменение в контракте, разработчику будет необходимо производить изменения только в двух сервисах, вместо трех как это происходило ранее.

### **2.2.3 Common coupling (общая связь)**

Общая связь возникает, когда два или более микросервисов использует какой-либо общий ресурс или набор данных. Примером может служить группа микросервисов, использующие одну базу данных, общую память или файловую систему (рис. 2.4). Основная проблема, которая возникает при таком типе связи – если данные, которые используются множеством микросервисов меняются, то в этом случае это потребует изменения во всех этих микросервисах. Например, в какой-то группе микросервисов присутствует компонента, которая хранит в себе статичные данные стран, в которых оперирует бизнес. Если структура данных изменится это повлечёт за собой изменения во всех местах, где эти данные были нужны.

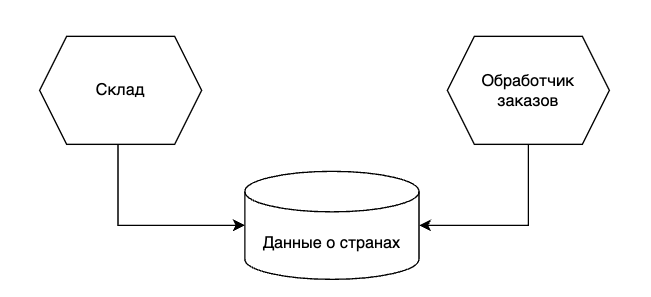


Рис. 2.4 – пример общей связи

В примере, приведенном выше, ситуация не так критична, так как структура данных о различных странах скорее всего будет меняться очень редко или не будет изменяться совсем. Тем не менее в случае с данные, которые могут предполагать частые изменения, такая структура может принести массу неудобств.

Другим примером такой связи, может быть два или более сервисов, которые одновременно работают с одними и теми же данными в другой таблице. В контексте наших примеров это могут быть сервисы Обработчик заказов и Склад, оба выполняющие операции над таблицей “Заказы”, изменяя статус заказа. (рис. 2.5)

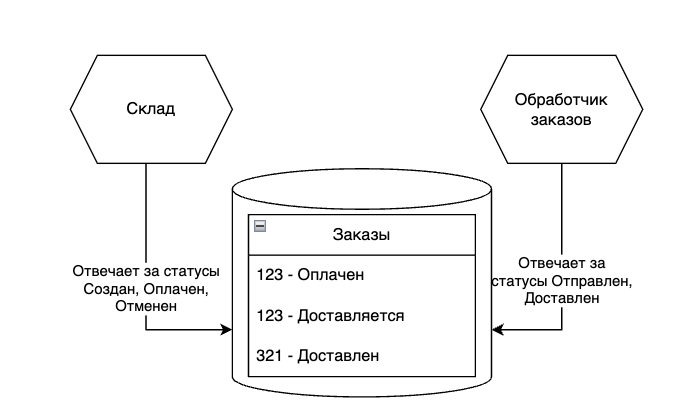


Рис. 2.5 – другой пример общей связи

В данном случае возникает проблема, связанная с тем, что статус заказа может быть некорректно изменен, а также то, что доступ к таблице должен быть у обоих микросервисов. Для решения этой проблемы может быть выделен дополнительный микросервис, который будет отвечать за заказы. В этом случае вместо прямого доступа к таблице Заказы обоим микросервисам придётся посылать запрос для того, чтобы изменить статус заказа. Тем не менее все еще остается проблема некоректного изменение статуса заказа, которую можно будет решить использованием конечного автомата. (рис. 2.6)

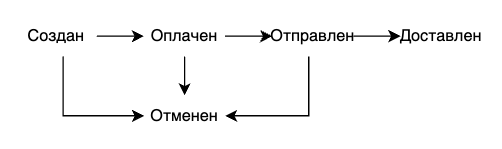


Рис. 2.6 – пример автомата для изменения статуса

После вышеописанных изменений получим систему, в которой невозможно будет некорректно поменять статус заказа. Например, переместить его из состояния Создан сразу в Доставлен. Также логика относящиеся к изменению статуса заказа теперь будет находиться в одном микросервисе вместо двух как было ранее

### **2.2.4 Content Coupling (связь контента)**

Этот тип связи очень похож на описанный ранее. Разница заключается в том, что в предыдущем типе микросервисы знали, что они оба используют один и тот же ресурс, в этом же случае предполагается, что один и более микросервисов могут изменять состояние объекта через обходные пути, например напрямую в базе данных. В нашем примере это может выглядеть следующим образом. Например, существует микросервис Заказ и Обработчик заказов, как в прошлом примере. Когда сервису обработчик заказов необходимо внести изменение в статус заказа, он делает запрос в сервис Заказов и он уже в свою очередь выполняет обновление в базе данных. Добавим в эту же структуру другой уже ранее знакомый сервис – Склад, который также хочет иметь возможность изменять статус заказов, но в данном случае он не будет отправлять запрос в сервис Заказы, а напрямую подключится к базе данных и выполнит обновление. (рис. 2.7)

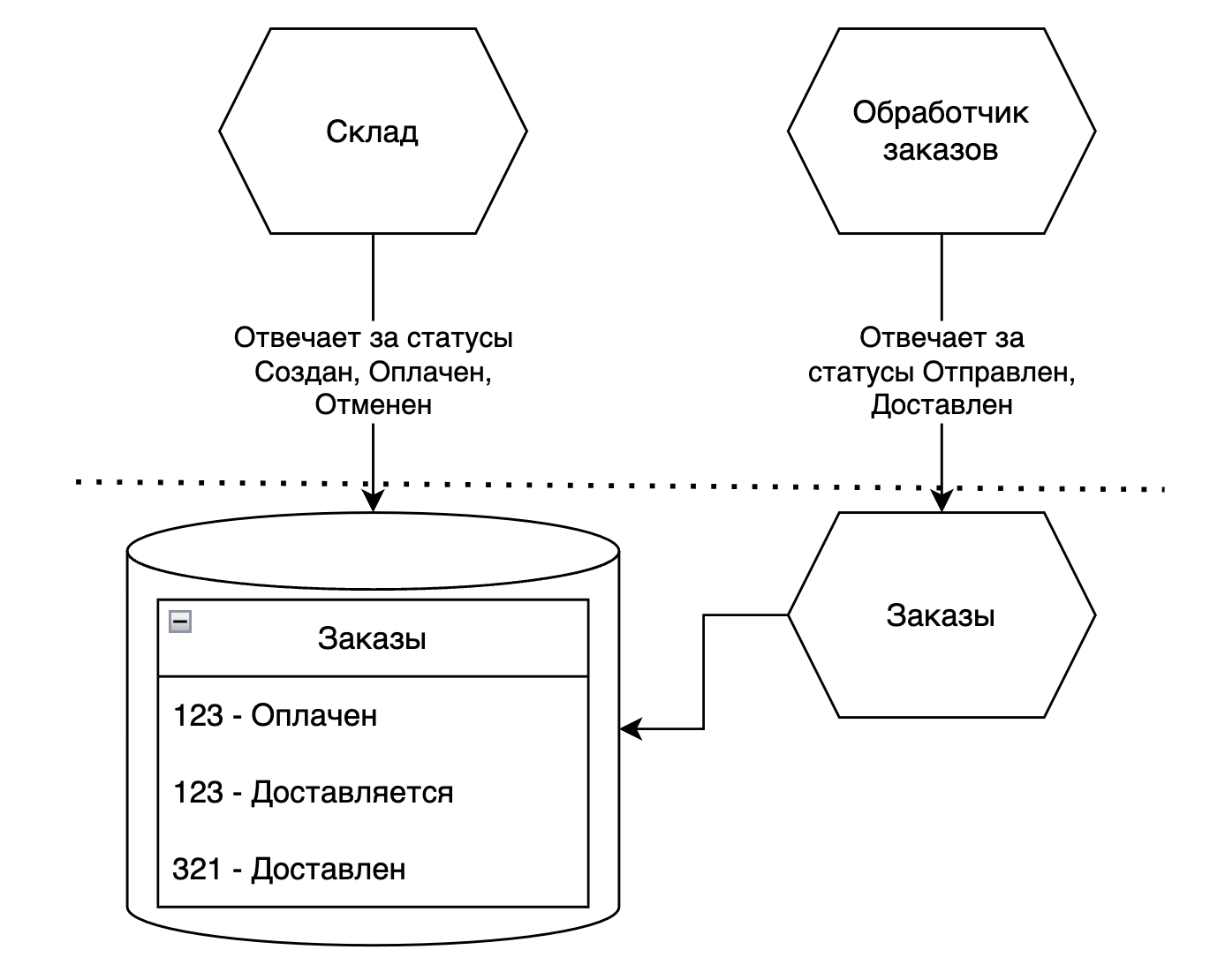


Рис. 2.7 – пример связи по контенту

В данном случае может возникнуть трудности с необоснованным обновлением данных, и такая ситуация при проектировании микросервисов должна избегаться, а если существует, то должна быть исправлена.

## **2.3 Domain Driven Design**

Как говорилось ранее одним из ключей к успеху моделирования микросервисов является правильное определение границ микросервиса. Одним из наилучших способ определения границ является определение границы по области применения или домену (domain driven design).

DDD (Domain Driven Design) – это один из важнейших подходов проектирования, в котором производится фокус на моделировании ПО согласно его доменной области. В данном подходе составные программного кода (название классов, методов, переменных) в сервисе должны относиться так или иначе к доменной области этого сервиса. [5]

В своей книге Domain-Driven Design Эрик Эванс представил набор важных идей, которые помогли лучше понять проблему области в программах. Несколько из них представлены далее.

### **2.3.1 Понятный язык**

Основная идея заключается в том, что разработчики при написании кода должны использовать те же термины, что в дальнейшем будут использовать пользователи приложения. Придерживаясь такому принципу, улучшится коммуникация между командой разработки и людьми из реального мира, тем самым улучшится понимание между ними, так как они будут общаться на языке понятным обоим сторонам.

В качестве контрпримера можно предположить платежную систему одного конкретного магазина. Разработчики, желая сделать наиболее красивое и элегантное решение выбрали имплементировать какой-то уже существующий общий дизайн такой системы, который нашла на просторах интернета. В процессе работы выяснилось, что им приходится много общаться с бизнес-аналитиком, чтобы провести аналогию между тем что нужно было бизнесу и тем, что абстрактно задавался дизайном из интернета. По итогам система получилась излишне сложной, так как пыталась реализовать общий подход к проблеме, когда в ее случае отлично бы сработало создать систему отвечающую исключительно требованием проекта, не вводя лишних абстракций, усложняющих понимание работы системы.

### **2.3.2 Агрегаты**

Агрегаты в DDD, это одна из концепций, которая чаще всего может быть понята неправильно. Существует множество определений, каждое из которых имеет место быть и логично, но в большинстве случаев может быть понято только людьми, которые уже знакомы с агрегатами. Для того, чтобы проще понять, что такое агрегаты существует простая аналогия – думать об агрегатах, как о пузырях. Необходимо представить, что проект — это не массивная кодовая база, а набор небольших пузырей, над каждым из которых можно работать по отдельности. Это означает, что в один момент времени вам необходимо думать только об отдельном пузыре, не обо всей системе целиком. [6]

Для примера, предположим, что у нас есть система в области управления проектами, состоящая из трех основных сущностей – проектов, команд и пользователей. Каждая команда имеет множество пользователей, включенных в нее. Каждый пользователь в одно и то же время может быть участником нескольких команд. Каждая команда в свою очередь может относиться к нескольким проектам. Теперь наделим каждую из сущностей свойствами. Для того чтобы наиболее наглядно показать проблему, предположим, что пользователь имеет свойство, отвечающее за его роль на проекте. В ситуации описанной выше появляется следующая проблема, которую и помогают решить агрегаторы. Суть заключается в том, что ранее говорилось то, что пользователь может одновременно находиться в нескольких командах. Обычно для реализации такой связи разработчики создают новую таблицу. Но что, если в будущем появится еще множество аналогичных полей. В этом случае рекомендуется выделить такие сущности, например в сущность “Член команды” и “Член проекта”, где, собственно, уже и будет храниться информация специфичная для члена команды определённого проекта. Таким образом пузыри для нашего примера будут выглядеть следующим образом. (рис. 2.8)

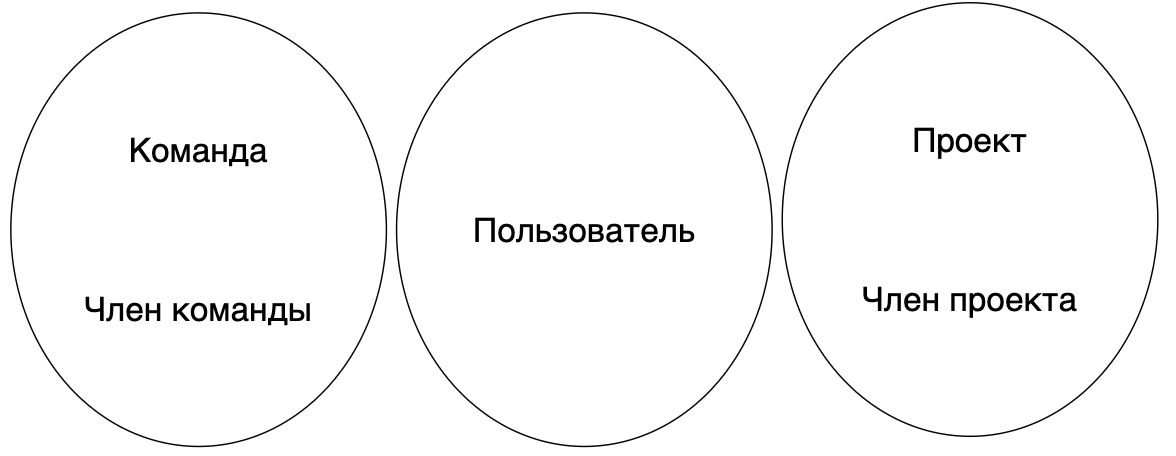
****

Рис. 2.8 – пример распределения по доменам

### **2.3.3 Ограниченный контекст**

Ограниченный контекст — это также один из основных паттернов в DDD. Суть данного принципа заключается в работе с большим количеством моделей и команд. В данном случае такие модели группируются и разбиваются на две или более области. В качестве примера может быть приведен сервис, который разбит на две большие области – область финансов (отвечающая за обработку транзакций) и области учета и доставки товара. Причиной разбиения по сути одного проекта на подмножество проектов заключается в том, что таким образом каждый из проектов сможет задать свои определения. Например, в обоих контекстах может существовать одна и та же сущность по названию, но различаться по смыслу. Для того, чтобы не создавать путаницу при использовании DDD и рекомендуется разбивать большие модели на контексты и далее уже работать с каждым контекстом по отдельности. [7]

В заключении данного раздела стоит уточнить, что были рассмотрены лишь некоторые основные принципы предметно ориентированного дизайна и для того, чтобы раскрыть данную тему потребуется намного больше одного раздела или даже целой книги. Как итог, отметим высокое удобство данного подхода, так как он позволяет интегрировать бизнес в наш код и помогает упростить понимание системы для различных членов команды, а также пользователем.

## **2.4 Разбиение монолитного приложения**

Ранее в данной работе затрагивались такие темы преимущества использования микросервисов и методы их моделирования. В реальности часто разработчики не пишут решение с нуля и уже имеют какой-то работающий монолит. Такое состояние, также хорошо для использования микросервисной архитектуры, так как вместо написания всей системы с нуля, возможна постепенное выделение микросервисов из монолита.

### **2.4.1 Инкрементальная миграция**

Миграция монолитного приложения на микросервисную архитектуру длинный процесс, в котором не рекомендуется спешить для того, чтобы минимизировать риск прийти в какой-то момент времени к неработающей системе. Посредством инкрементальной миграции разработчики будут иметь возможность научиться на своих ошибках и при следующем шаге уже не допускать их. В случае, если один из таких шагов окажется неверным, будет намного проще переделать все, так как это был небольшой функционал. Таким образом, двигаясь к цели небольшими шагами и совершенствуясь с каждым этапом, миграция будет становиться все проще и проще.

Перед началом любой миграции необходимо определить какие цели должны быть достигнуты в первую очередь. Это может быть желание начать с выделения функционала, который подвержен высоким нагрузках или части приложения для изменения в которой уходит много сил и времени.

Какой бы ни был выбор, скорее всего все сведется к балансу между легкостью выделения функциональности и преимуществам, которые последуют за этим. Также порой в качестве первых изменений стоит рассматривать изменения, которые пройдут безболезненно. Это критически важно обеспечить миграцию небольшими победами перед тем, как приступать к функционалу, который займет намного большее количество ресурсов и, если начать с него, можно и вовсе оставить данную идею о миграции.

Далее рассмотрим одни из наиболее популярных паттернов, используемых при миграции монолита на микросервисную архитектуру.

### **2.4.2 Strangler Fig**

Данная техника часто используется при переписи исходного приложения, а сама идея была позаимствована Мартином Фловлером у разновидности фикуса. Семена этого дерева переносят птицы и обычно начинает прорастать на верхушке других деревьев. Со временем он спускает свои корни вниз к земле по пути убивая или замещая оригинальное дерево. Со временем умершее дерево исчезает, оставляя только новое дерево. [9]

В контексте разработки программного обеспечения аналогия заключается в одновременной поддержке старой и новой части нашего приложения, причем новая часть будет постепенно вымещать предыдущую. Несмотря на то, что данный паттерн чаще всего использовался для миграции одного монолитного приложения на другое более новое, но также монолитное приложение, в нашем случае рассмотрим вариант миграции на группу микросервисов.

Применение данного паттерна можно разбить на три этапа:

1. Выбор функциональности – вначале необходимо выбрать какая функциональности будет перенесена первая.
2. Имплементация – далее необходимо переместить из монолита или реализовать с нуля желаемый функционал в новом микросервисе.
3. Перенаправить запросы – запросы, относящиеся к новой функциональности, должны быть перенаправлены на ранее созданный микросервис, все остальные должны, как и раньше обрабатываться монолитом.

Одним из важнейших преимуществ данного паттерна является то, что в случае неполадки или проблемы с новым микросервисом, разработчики всегда имеют возможность перенаправить все запросы на монолит, тем самым давая себе время на исправление неполадки.

### **2.4.3 Branch by Abstraction**

Предыдущий подход отлично подходит в случае, если функционал легко извлекается из существующего монолита, но в случае, если желаемая функциональность глубоко заложена в коде и ее извлечение не такая тривиальная задача, а также может повлечь за собой неполадки необходим другой подход. В данном разделе обсудим стратегию, которая позволяет минимизировать влияние таких изменений на работу других разработчиков.

Суть данного паттерна заключается в создании отдельной ветки, где будет разрабатываться вторая версия приложения и может быть сформулирована в пяти следующих шагах:

1. Создание абстрактной модели для функциональности, которая будет заменена (в большинстве языков программирования это достигается через интерфейсы)
2. Перенаправить пользователей использовать старую версию используя абстракцию
3. Реализация переработанной функциональности
4. Перенаправить пользователей использовать новую версию
5. Удаление старой реализации

Важно отметить, что при выполнении предпоследнего шага необходимо дополнительно подготовить механизм переключения на старую имплементацию в случае ошибки. Это может быть реализовано с использованием прокси или, например наличием логики, меняющую используемую реализацию в зависимости от параметра в конфигурации. Также альтернативно может быть применен следующий прием – изначально направлять запросы на новую реализацию и в случае ошибки перенаправлять тот же запрос на старую реализацию. Таким образом конечный пользователь не будет затронут неудачным первым запросом, а инженерам, следящим за приложением, не придётся вручную перенаправлять траффик на старую реализацию.

### **2.4.4 Parallel Run**

Несмотря на то, что два предыдущих паттерна позволяют выполнять инкрементальную миграцию, в большинстве случаев при их использовании разработчику приходиться выбирать какую из реализаций использовать в определенный момент времени. Следующий паттерн, который будет рассмотрен, открывает возможность одновременного использования двух систем. Это особенно удобно для того, чтобы сравнить насколько правильно работает новая реализация. Обычно при таком подходе, необходимо выбрать какая из реализаций будет так называемым источником правды (на практике чаще всего на эту роль выбирают старую реализацию).

Стоит отметить, что данная техника может быть использована не только для сравнения работы двух систем, но также для сравнения двух функциональности разных версий одной и той же системы. В ходе проверки обычно вначале разработчики должны удостовериться в правильности работы новой функциональности, но на этом проверка не заканчивается, так как далее можно также сравнить и не функциональные требования такие как время выполнения запроса.

В связи с тем, что в процессе тестирования будут использоваться обе системы, существует риск дупликации данных или действий. Например, представим, что работа ведет над функционалом обе версии которого отправляют оповещение пользователю через сторонний сервис. Если обе версии приложения будут одновременно использоваться, то данное действие будет дублировано и пользователь получил дублирующее оповещение. Для того чтобы избежать такой ситуации можно воспользоваться шпионами (Spy), которые так же популярны в тестировании. Основная идея заключается в оборачивании сервиса отправки оповещений для одной из версий функциональности, который будет регистрировать получение запроса на отправку сообщения, но не отправлять его.

В заключении можно сказать, что реализация данного подхода является трудозатратой и чаще всего используется для тестирования функциональности с высоким уровнем риска и высокими требованиями к работе компоненты.

### **2.4.5 Нюансы декомпозиции данных**

В ходе разбиения монолита на микросервисы помимо работы над выделением кода, также будет необходимо разбить базу данных, что может быть не такой тривиальной задаче. В этом разделе будут обсуждены нюансы разбиения базы данных.

В наше время базы данных, в особенности реляционные базы данных, имеют множество оптимизаций и способно быстро и эффективно извлекать данные, распределенные по разным таблицам. В процессе создания очередного микросервиса зачастую разработчики перемещают данные связанные с данным микросервисом в отдельную базу данных, тем самым таблицы, которые прежде были связаны на уровне базы данных, не могут быть больше объединены базой данных автоматически, а разработчикам приходится реализовать эту логику на уровне кода в микросервисах.

В результате производительность системы падает, так как вместо одного SQL запроса в базу данных, производится два запроса в базу данных, а также еще от одного и более запросов в другие базы данных. (рис. 2.9)

**Diagram

Description automatically generated**

Рис. 2.9 – сравнение БД в различных архитектурах

Для решение возникшей проблемы с производительностью существует разные подходы, самые популярные из которых это кеширование данных и одновременные запрос данных по большому количеству идентификаторов. Например, вместо ста запросов на получение информации о продукте по идентификатору, посылается один запрос, содержащий все интересующие продукты.

Другой проблемой, которая появляется при таком разбиении, это невозможность использовать преимущества транзакции на уровне базы данных, следовательно разработчик теряет все преимущества ACID транзакций. Частичным решением данной проблемы является использование распределенных транзакций, но их реализация зачастую нетривиальна, а их наличие все равно не может гарантировать все преимущества, которые дает реляционная база данных.

Аналогично микросервисам, как бы разработчикам не хотелось обратного, базы данных также требуют изменений со временем. Для контроля версий кода приложения существует множество различных инструментов, но самым популярным является Git. В мире баз данных самыми популярными аналогами Git, но для баз данных являются Flyway и Liquibase. Основная идея работы данных инструментов заключается в создании скриптов, которые будут описывать дельты между версиями схемы базы данных. Далее эти скрипты будут выполняться специальными плагинами, встроенными в эти инструменты, которые будут контролировать, что каждая из дельт будет применена только единожды.

В данном разделе были затронуты наиболее часто встречающиеся паттерны и практики, используемые при миграции монолитного приложения на микросервисную архитектуру, как с точки зрения кода, так и с точки зрения базы данных. Другим важным аспектом, от которого зависит успех такой миграции, является успешная коммуникация созданных микросервисов между собой, что будет рассмотрено в следующей главе.

## **2.5 Способы коммуникации микросервисов**

В данном разделе будет рассмотрен другой важный аспект построение успешного проекта с помощью микросервисной архитектуры – способ коммуникации. Чаще всего разработчики выбирают тот способ, который наиболее им знаком, не беря во внимание преимущества других подходов. В последующих секциях рассмотрим преимущества и недостатки синхронного и асинхронного способа коммуникации, а также сравнив http запросы с событие ориентированным подходом.

### **2.5.1 Синхронные запросы**

Суть данного типа коммуникации заключается в отправке запроса и блокированию дальнейшего выполнения программы, пока не будет получен ответ или не пройдет определенное количество времени.

Преимуществом такого способа коммуникации является его простота и распространенность среди разработчиков, так как чаще всего изучение программирования начинается с написания скриптов, которые по своей натуре выполняются в синхронном стиле. С другой стороны, использование этого метода хоть и является простым в использовании, но может стать настоящей проблемой. Во-первых, в случае ошибки система, делающая запрос должна содержать логику, как реагировать на такие ситуации. Во-вторых, как ранее упоминалось, при таком методе блокируется дальнейшее выполнение программы, что в случае необходимости выполнения множество запросов при отработке одного сценария, сильно ухудшает производительность системы и увеличивает отклик, так как вместо отправки второго и возможно третьего запроса, микросервис должен вначале дождаться ответа на первый запрос.

Несмотря на недостатки данного подхода, он все равно является популярными и часто используемым из-за его простоты. Ведь разработчику не зачем усложнять себе жизнь, если микросервис не делает множество запросов в другие сервисы и сложность, которые привнесут асинхронные запросы перевешивают получаемые преимущества.

### **2.5.2 Асинхронные запросы**

В сравнение асинхронных запросов с прошлым типом они в свою очередь не блокируют выполнение программы при отправке запроса. Данный тип коммуникации имеет несколько возможных реализаций, которые будут рассмотрены:

* Коммуникация через общий ресурс, например файловое пространство
* Общение используя HTTP протокол, но не блокируя дальнейшее выполнение программы
* Взаимодействие, ориентированное на события чаще всего посылаемые через очереди (например JMS)

Как говорилось ранее, основным преимуществом такого типа коммуникации является не блокирование работы программы в ожидании ответа от сервера. Это особенно полезно в случае, если разработчику известно, что такой запрос занимает большое количество времени, а также если дальше логика приложения включает в себя отправку других запросов. Главным же недостатком такой системы является сильно возрастающая сложность при дебагинге ошибок.

В качестве примера можно привести следующую ситуацию (рис. 2.10). Сервис Обработчик заказывает отправляет сообщение об необходимости отправке товара, далее сервис Склад возьмем данный запрос на обработку, когда он будет доступен и по успешному завершению отправит обратное сообщение сервису Обработчику заказов, о том, что товар был отправлен пользователю. Обработчик заказов в свою очередь также может не сразу принять запрос, а когда будет доступен.

Diagram

Description automatically generated

Рис. 2.10 – пример асинхронного запроса

### **2.5.3 Коммуникация используя общие данные**

В данном типе коммуникации один микросервис сохраняет необходимые данные в месте, а далее другие микросервисы могут использовать эти данные в своих целях. По своей натуре данный метод коммуникации является асинхронным. В качестве примера можно привести следующую ситуацию (рис. 2.11). Представим, что каждый месяц в онлайн магазин сторонний клиент предоставляет новый список спонсорских товаров загружая их в облачное хранилище и дает к нему доступ. Далее микросервисы онлайн магазина могут подключаться к облачному хранилищу и использовать эти данные для обновления каталога.

A picture containing graphical user interface

Description automatically generated

Рис. 2.11 – пример коммуникации через общее облачное хранилище

Для реализации данного типа коммуникации не обязательно использовать облачное хранилище, во многих ситуациях подойдет пространство куда можно сохранить данные и откуда впоследствии их можно будет прочитать. Это может быть, как и облачное хранилище в примере выше, так и любая файловая система или база данных.

Главными преимуществами использования данного паттерна заключается в том, что он прост в реализации, а также способен переносить большие объемы данных, что будет затруднительным используя http протокол, поэтому данный подход часто применяют для коммуникации старой легаси системы с другими сервисами. Недостатком такого подхода является то, что сервисам нужно как-то узнавать добавление новых данных для обработки. Это может быть сделано с помощью отправки оповещения, реализации процесса, который будет раз в определенное время сканировать файлы и выявлять новые данные доступные для обработки.

### **2.5.4 Коммуникация запрос-ответ**

Скорее всего данный способ коммуникации микросервисов является одним из наиболее распространённых, который можно встретить, как в синхронной реализации, так и в асинхронном виде. Основная идея заключается в отправке запроса другому сервису и ожидании ответа от него.

Как говорилось ранее данный тип коммуникации может носить как синхронный, так и асинхронный вариант. В случае синхронной работы при отправке запроса будет открыто соединение с другим сервисом. После обработки запроса второму микросервису достаточно воспользоваться тем же соединением, которое было открыто ранее. В случае, если соединение рушится по причине недоступности одного из микросервисов, то и запрос или ответ не сможет быть получен.

Асинхронный вариант общения не настолько прост. Для примера рассмотрим следующий пример (рис. 2.12), где запрос и ответ посылается не с помощью http запроса, а через очереди. Сервис Обработки заказов будет отсылать запрос сервису Склад через очередь. Сервис Склад будет читать из этой очереди, производить необходимые операции и далее отправлять сообщение назад через другую очередь. Когда сервис Обработки заказов будет снова доступен он прочитает сообщение и выполнит необходимые операции с его стороны.

Diagram

Description automatically generated

Рис. 2.12 – пример коммуникации запрос-ответ

Данный тип коммуникации чаще всего используется в ситуациях, когда для продолжения выполнения процесса необходимо знать результат выполнения запроса и в случае ошибки среагировать соответствующе, например попробовав еще раз через определенное время или сгенерировав соответствующую ошибку.

### **2.5.5 Событийно ориентированная коммуникация**

Событийно ориентированная коммуникация может на первый взгляд выглядеть странно по сравнению с предыдущим способом. Вместо того, чтобы отсылать запрос и ждать ответа, микросервис просто посылает запросы, которые могут или не могут быть выполнены другим микросервисом.

Событие — это факт происхождения определенной ситуации в микросервисе, которую другие миросервисы могут использовать по своему усмотрению, причем сервис, генерирующий это событие может и не знать о том, кто и как будет использовать эту информацию.

В качестве примера возьмем микросервис Доставка, который отправляет, публикует сообщения, когда заказанный объект прошел тот или иной пункт при доставке. Далее в зависимости от события, оно, может быть, просто сохранено в качестве лог сообщения или при определенном условии может быть запущен другой процесс. Например, при прибытии товара на таможенный контроль и если стоимость товара превышает необлагаемую налогом сумму, то с получателя должны дополнительно запросить средства. (рис. 2.13)

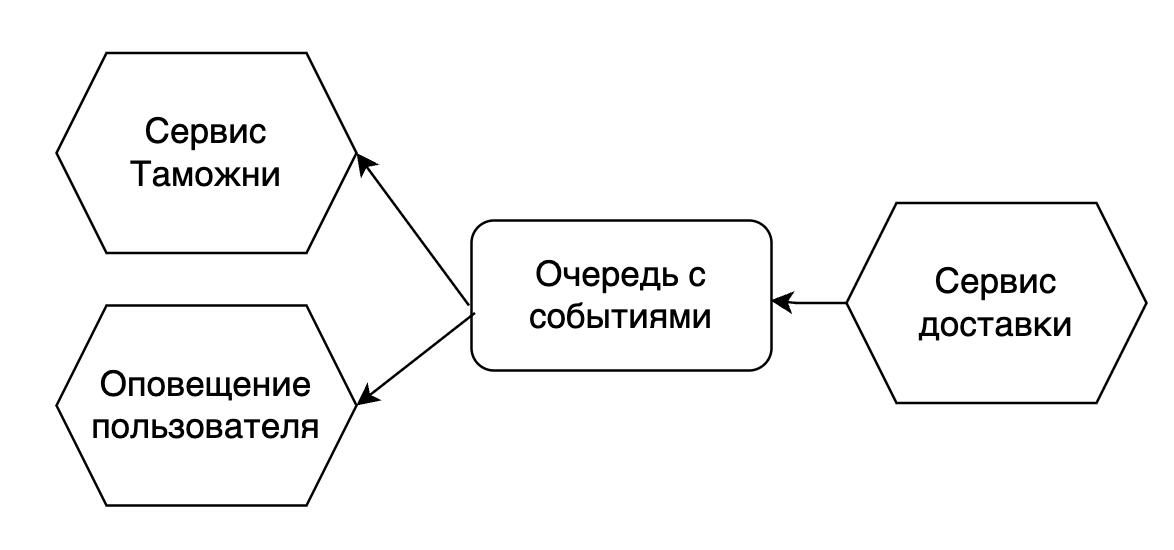


Рис. 2.13 – пример коммуникации через события

При реализации такого метода необходимо учитывать два ключевых момента – как микросервис будет публиковать событие и как другие микросервисы будут узнавать о том, что появились новые события для обработки. Для решения обоих проблем можно воспользоваться одним из уже готовых брокеров сообщений, таких как RabbitMQ. Данная технология содержит API как для сервиса генерирующего события, так и для сервиса, потребляющего их. Однако стоит понимать, что добавление дополнительного сервиса, который нужно развернуть и поддерживать для разработки и тестирования приложения, привносит дополнительную сложность.

Стоит также отметить, что существуют ситуации при которых данный подход может показывать себя не так хорошо. Например, при передаче больших могут возникнуть проблемы с отправкой сообщения. Так популярный брокер Kafka имеет лимит в размере 1 МБ, а последняя версия RabbitMQ теоретически может отправлять события размером до 512 МБ. Другой опасностью является то, что как только события начинают быть использованы другими сервисами, сервис, создающий события, при изменении контракта не может быть уверен, что данные необходимые другому микросервису не будут удалены.

Существуют также и другие сложности при использование событийно ориентированного общения, которая заключается в том, что по своей натуре оно носит асинхронный характер, а это всегда увеличивает сложность. Программные продукты всегда содержат недоработки и в случае возникновения бага, приложению также нужно иметь стратегию, как реагировать на исключительные ситуации. В общем и целом, можно выделить две важные рекомендации, которым следует придерживаться при использовании событий:

* В случае, если при обработке произошла ошибка и сообщение было оставлено очереди для дальнейшей ошибки может привести к тому, что оно будет бесконечно вызывать ошибки. В таком случае желательно поставить лимит на то, сколько раз сообщение может попытаться выполниться повторно.
* Следующая рекомендация связана с предыдущий и относится к тому, что необходимо делать в случае, если событие превысило количество попыток на обработку. Популярным подходом является создание дополнительной очереди, куда можно будет поместить событий, вызывающее ошибку.

Данный способ коммуникации часто напрашивается сам собой, когда данные просятся быть оповещением, которые другие части системы смогут использовать, а также в случае, если необходимо упростить один из микросервисов, так как с помощью событий, перегруженный микросервис больше не будет отвечать за логику отправки и обработки запросов. Это в свою очередь помогает стать системе менее связанной и легче поддаваемой изменениям.

### **2.5.6 Обработка ошибок**

При обычном написании кода в случае возникновения ошибки разработчик получает исключение, содержащее информацию о месте возникновения ошибки, а также в большинстве случаев указывает и на причину возникновения. В мире микросервисов нет аналогичной функциональности, встроенной на уровне языка программирования, поэтому ясность и простота обработки исключений напрямую зависит от решений принятыми разработчиками.

В своей книге “Распределенные системы”, Эндрю Таненбаум и Мартин Стэн сформулировали основные случаи исключений возникающий при коммуникации микросервисов [10]:

* Фатальная ошибка – сервер упал и недоступен, решить проблему возможно только перезагрузкой
* Запрос был отправлен, но ответ не был получен
* Таймаут – ответ был отправлен слишком поздно
* Ответ был получен, но он содержит ошибку

Многие из встречающихся ошибок не критичны и в каком-то проценте всегда будут периодически случаться. Например, сервис был недоступен пять секунд, но после этого второй запрос уже прошел успешно. Для того, чтобы легче было понимать, когда требуется вмешательство разработчика, а когда нет, необходимо правильно использовать код или статус ошибки. Например, в протоколе HTTP существует статусы ответов от сервера, где 400-е и 500-е ошибки указывают на ошибку. Четырехсотые ошибки указывают на то, что причина ошибки заключается в клиенте и сервер корректно обработал исключение. В таком случае скорее всего не стоит привлекать разработчика, так как данная ошибка могла произойти из-за неполадки сети. Пятисотые же ошибки указывают, что сервер не смог обработать запрос и случилась исключительная ситуация, возможно требующая анализа со стороны разработчика.

# **­­­­­­III. МИГРАЦИЯ ПРИЛОЖЕНИЯ ''МЕНЕДЖЕР ЗАДАЧ'' НА МИКРОСЕРВИСНУЮ АРХИТЕКТУРУ**

## **3.1 Обзор изначального приложения**

Перед миграцией приложения на микро сервисную архитектуру, рассмотрим функционал изначального приложения “Менеджер задач”. Оно представляет собой копию известного приложения “Jira”, используемое для трекинга задач во многих IT организациях.

Приложение представляет собой монолит, написанный с использованием фреймворка Spring, и использует базу данных PostgreSQL. Бэкенд приложения предоставляет пользователю возможность регистрировать новых пользователей в системе, распределять задачи между пользователями, создавать спринты, изменять статус задач. Данный функционал распределен между тремя основными контроллерами – пользователь, задача, спринт и имеет следующую спецификацию:

ProfileController – контроллер для управления профилем:

* POST - /profile – создание нового профиля
* GET - /profile – получение списка профилей
* GET - /profile/{profileId} – получение профиля с помощью ID
* PATCH - /profile/{profileId} – обновление профиля
* DELETE - /profile/{profileId} – удаление профиля по ID

TaskController – контроллер для управления задачами:

* POST - /task – создание новой задачи
* GET - /task – получение списка задач
* GET - /task/{taskId}– получение задачи с помощью ID
* PUT - /task/{taskId}/profile/{profileId} – закрепить задачу за определенным пользователем
* PUT - /task/unasign/{taskId}/ - убрать пользователя с текущей задачи
* PATCH - /task/{taskId} – обновление задачи
* DELETE - /task/{taskId} – удаление задачи
* PUT - /task/{taskId}/todo – установка статуса задачи на TODO
* PUT - /task/{taskId}/progress – установка статуса задачи на IN PROGRESS
* PUT - /task/{taskId}/done – установка статуса задачи на DONE

SprintController – контроллер для управление спринтами:

* POST - /sprint – создание нового спинта
* GET - /sprint – получение списка спринтов
* GET - /sprint/{sprintId} – получение спринта с помощью ID
* PUT - /sprint/{sprintId}/add/{taskId} – добавление задачи к спринту
* DELETE - /sprint/{sprintId}/remove/{taskId} – удаление задачи из спринта
* PATCH - /sprint/{sprintId} – обновление информации о спринте
* DELETE - /sprint/{sprintId} – удаление спринта
* POST - /sprint/{sprintId} – закрытие спринта

Как видно из спецификации API несмотря на то, что приложение представляет собой монолит, тем не менее оно так же разбито на сущности и может быть представлено следующей схемой (рис. 3.1).

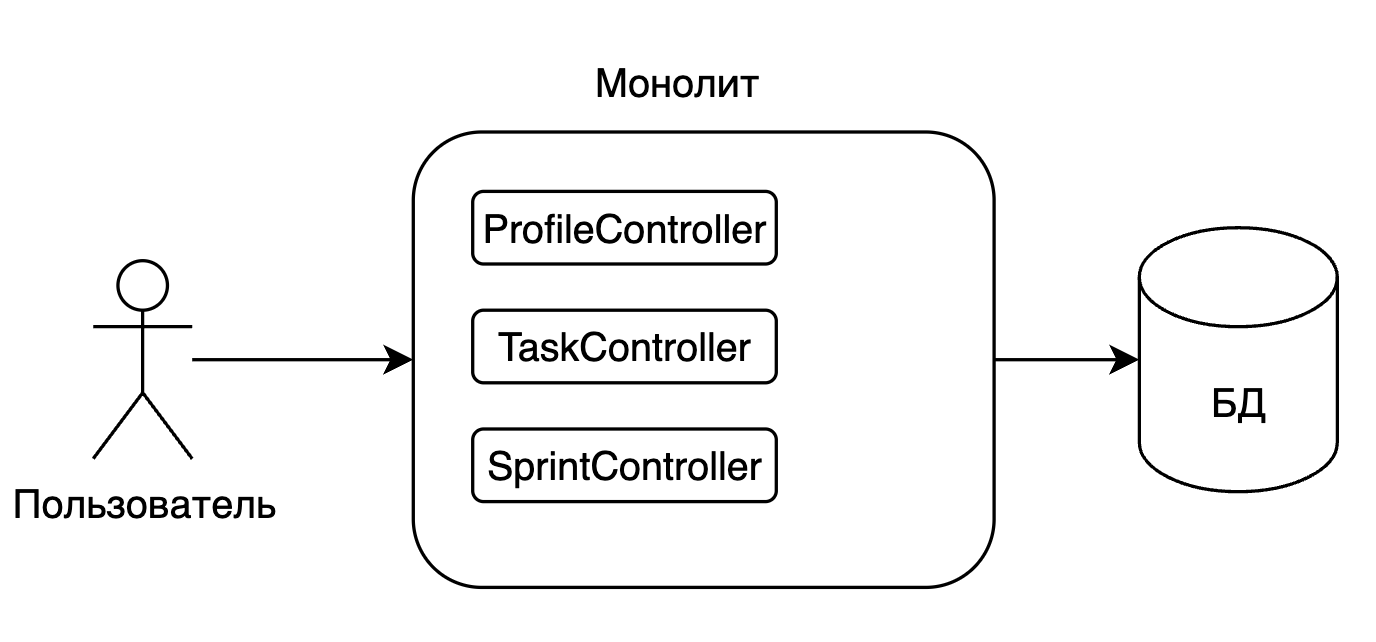
****

Рис. 3.1 - Схема устройства монолитного приложения

В дальнейшем данное разбиение будет очень удобно использовать при выделении функционала из монолита в отдельные микросервисы, каждый из которых будет ответственен за свою часть логики. Таким образом разбиение по доменам произойдет легко и без трудностей по идентификации границ обязанностей каждого из модулей. В качестве дополнения к существующей системе также будет добавлен новый модуль, вводящий новую сущность – Проект, которая предоставит пользователю возможность группировать спринты и задачи в зависимости от проекта, а также контролировать доступ к ним.

## **3.2 Подготовка инфраструктуры**

Как говорилось ранее в теоретическом разделе работы вместе с преимуществами, которые привносят микросервисы, данный подход также увеличивает сложность системы и инфраструктуры. В случае с микросервисной архитектурой перед началом миграции сервиса, будет необходимо создать инфраструктуру, которая впоследствии облегчит работу, а именно сконфигурировать микросервис, который будет ответственен за регистрацию других микросервисов и упростит коммуникацию между ними. Другим важным составляющим является API gateway, через который в свою очередь будут проходить все запросы к микросервисам, и именно он будет ответственен за аутентификацию.

### **3.2.1 Service Registry**

Service Registry – это популярный паттерн используемый в микросервисной архитектуре, который упрощает коммуникацию между микросервисами. Проводя аналогию с реальным миром, он представляет собой справочную книгу, у которой можно запросить адрес и порт другого микросервиса. Это возможность критически необходима при использовании микросервисной архитектуры, так как микросервисы часто могут менять свои IP адреса.

На практике Service Registry это еще один микросервис. При запуске другие микросервисы будут отправлять запрос на регистрацию и далее периодически подтверждать свою доступность. Адаптируя пример к случаю с микросервисами приложения “Менеджер задач”, схематично это будет выглядеть следующим образом (рис. 3.2).

**Diagram

Description automatically generated**

Рис. 3.2 - Схема работы регистрации микросервисов.

Как видно из схемы (рис. 3.2.1), каждый из микросервисов регистрирует себя и далее становится доступным для коммуникации. Эти данные могут быть использованы, API gateway для направления запроса к сервису, так и для коммуникации между сервисами. Такой сервис может быть реализован с помощью многих фреймворков или написан вручную. Тем не менее для сохранения времени, затрачиваемое для разработки, будет использован фреймворк Spring, а именно модуль из семейства Cloud - spring-cloud-starter-netflix-eureka-server. С его помощью может быть получен работающий микросервис без написания кода, а только заданием правильной конфигурации, как в примере ниже:

eureka:  
 client:  
 register-with-eureka: false  
 fetch-registry: false  
server:  
 port: 8761

Следующим шагом необходимо также сконфигурировать остальные микросервисы для отправки данных микросервису регистратору. Это может быть сделано также исключительно через конфигурацию. Далее будет приведен пример конфигурации сервиса Sprint Service, где указывается название сервиса, по которому к нему можно будет в будущем обращаться, а также адрес регистрационного микросервиса:

spring:  
 application:  
 name: SPRINT-SERVICE  
  
eureka:  
 client:  
 service-url:  
 defaultZone : <http://localhost:8761/eureka/>

При запуске всей системы также имеется возможность зайти на страницу микросервиса регистратора и посмотреть список известных микросервисов. Пример, такого табло представлен на рисунке 3.3.

Table

Description automatically generated

Рис. 3.3 - Пример табло сервиса Service Registry

### **3.2.2 API gateway**

API gateway - является не менее важным типом микросервиса при построении веб приложений, который является входной точкой (точкой доступа) в систему через которую проходят все запросы. В контексте рассматриваемого приложения он будет ответственен за аутентификацию входящих запросов (блок 1 рис. 3.4), а также заниматься перенаправлением запросом к другим микросервисам (блок 2 рис. 3.4).

Diagram

Description automatically generated

Рис. 3.4 - схематичное представление работы API gateway

При реализации такого микросервиса, так же, как и ранее будет использован один из модулей Spring фреймворка, а именно модуль spring-cloud-starter-gateway. С его помощью реализация будет достигнута с наименьшим количеством написанных строк. В конфигурации данного микросервиса необходимо будет указать символически имена микросервисов и пути к ним, а также указать какие из путей должны быть защищены, а какие нет. В случае приложением менеджера задач незащищенным должны будут быть пути, отвечающие за регистрацию и аутентификацию пользователя.

spring:  
 cloud:  
 gateway:  
 routes:  
 - id: **project-service** uri: **lb://PROJECT-SERVICE** predicates:  
 - **Path=/project/\*\*** filters:  
 - AuthenticationFilter  
  
 - id: **identity-service** uri: **lb://IDENTITY-SERVICE** predicates:  
 - **Path=/auth/\*\***

В примере выше приведен пример конфигурации путей для микросервиса отвечающего за регистрацию и аутентификацию пользователя, а также сервиса Project Service, который предоставляет API для работы с проектами. Стоит отметить, что сервис Project Service указывает пользовательский класс AuthentificationFilter, который в свою очередь будет содержать в себе логику, отвечающую за аутентификацию запросов. В случае, если запрос был аутентифицирован, микросервис так же извлечет имя пользователя из запроса и добавит его в качестве хедера к дальнейшему запросу к микросервису. В дальнейшем эта информация будет использоваться для определения кто именно из пользователей пытается совершить ту или иную операцию, а также имеет ли он на это право.

Другим моментом, который стоит отметить это то, что сервис не указывает точный IP адрес микросервиса, а лишь его имя, которое будет зарегистрировано в микросервисе регистраторе. Таким же образом, в дальнейшем другие микросервисы могут пользоваться именем другого микросервиса вместо его IP адреса.

Результатом использования данного паттерна проектирования было достигнуто множество преимуществ, а самое главное были сняты некоторые из обязанностей с микросервисов. Теперь им не понадобится заниматься аутентификацией запросов пользователя, а в случае, если запрос не аутентифицирован он даже не дойдет до микросервиса, тем самым не расходую ресурсы. Другим преимуществом является то, что любые фильтры или дополнительная логика, относящаяся к каждому запросу, может быть также размещена в данном микросервисе. В случае же инкрементальной миграции монолитного приложения на новую архитектуру API gateway предоставляет дополнительную гибкость в маршрутизации запросов.

## **3.3 Миграция доменов на микросервисную архитектуру**

В связи с отсутствием некоторых критически важных функций в изначальном приложении, таких аутентификация или авторизация пользователей, а также возможность создать и группировать работу по проектам, в ходе миграции будут добавлены микросервисы отвечающие за эти функции.

### **3.3.1 Миграция профиля пользователя в Identity/Profile микросервис**

В прошлом в обязанности данного домена входило только удержание информации о пользователе и не включало в себя такие функции как аутентификация пользователя, а также содержит некоторую информацию, которая будет перенесена в другой домен.

Первоначальная структура объекта представляет собой набор полей, хранящих информацию о имени пользователя, электронного адреса, роли на проекте, и статусе аккаунта. Во-первых, при добавлении аутентификации необходимо будет добавить новое поле для сохранения пароля. Далее будет необходимо добавить возможность создавать пользователя в системе, генерировать токен для аутентификации и его последующей валидации. Спецификация для контроллера, отвечающего за аутентификацию, выглядит следующим образом:

* POST /auth/register – регистрация нового пользователя в системе
* POST /auth/token – запрос на получение токена
* POST /auth/validate – запрос на валидацию токена.

Во-вторых, с появлением в будущем нового домена, отвечающего за менеджмент проектов, атрибут пользователя роль должна будет переместиться в другой микросервис, так как в новой версии приложения на различных проектах у пользователя могут быть различные роли, а в каком-то из случаев и вовсе не быть доступа к проекту. API спецификация для получения информации от пользователя будет выглядеть следующим образом:

* GET /user/{username} – получение информации о пользователе по имени.
* GET /user/{userId} – получение информации о пользователе по идентификатору.

Также из приложения была удалена возможность удаления аккаунта, для того чтобы не нарушать целостность данных, так как идентификатор пользователя может находиться во многих таблицах в базах данных за пределами данного микросервиса.

### **3.3.2 Создание микросервиса Project Service**

Как говорилось ранее, в прошлым проект не поддерживал создание и менеджмент множества проектах и все задачи находились как будто на одном проекте, были видны всем пользователям системы. При миграции приложения было решено добавить возможность группировать задачи и спринты по проектам, а также контролировать доступ к ним. Таким образом пользователь, не являющийся участником проекта, не сможет не только изменять статус задач, но также даже не будет знать о существовании другого проекта.

Для достижения данной цели необходимо создать подобие CRUD операций для создания и редактирования проектов, а также разработать API для менеджмента доступа к проекту. Первая задача тривиальна, и спецификация для нее будет выглядеть следующим образом:

* POST /project – создание нового проекта (текущий пользователь будет назначен создателем проекта и автоматически зачислен на проект)
* GET /project/all – получение всех проектах релевантных текущему пользователю.
* PUT /project/{projectId}- обновление данных о проекте (название, описание)

Второй обязанностью данного микросервиса будет контроль доступа к проектам. Для этого помимо таблицы для хранения данных о проекте будет создана еще одна таблицы для хранения доступа пользователей к различным проектам. Спецификация для этой части выглядит следующим образом:

* POST /project/management/{projectId}/{username} – добавление пользователя
* GET /project/management/{projectId}/{username} – получение информации, зачислен ли пользователь на проект
* PATCH /project/management/{projectId}/{username}?role={newRole} – изменение роли пользователя на проекте
* DELETE /project/management/{projectId}/{username} – удаление пользователя с проекта
* PATCH /project/management/{projectId}/close – закрытие проекта
* PATCH /project/management/{projectId}/reopen – открытие проекта

В дальнейшем данный микросервис будет часто использоваться другими микросервисами для авторизации действий пользователя, то есть избежания ситуаций, когда пользователь может манипулировать объектами зная их уникальные идентификаторами, но на самом деле не являясь частью проекта.

### **3.3.3 Миграция микросервиса Sprint Service**

В связи с тем, что приложение теперь поддерживает также разбиение по проектам, остальным микросервисам при миграции будет необходимо содержать в себе дополнительную логику для проверки авторизован ли пользователь выполнять то или иное действие. В случае с миграцией сервиса, отвечающего за менеджмент спринтов, необходимо будет изменить структуру сущности спринт.

Во-первых, ранее контроллер, отвечающий за данную сущность, отвечал за добавление или удаление задач из спринта. Это было возможно и удобно, так как ранее таблицы хранящие в себе данные о спринтах и задачах находились в одной базе данных и имели связь один ко многим. В случае микросервисной архитектуры данные больше не находятся в одной базе данных, поэтому перенос данной функции в домен задач выглядит логичным.

Во-вторых, прошлая версия содержала довольно сжатые и неполные данные, предоставляющие лишь базовую информацию. В ходе миграции были также добавлены дополнительный поля, которые могут хранить информацию о целях спринта, дате начала, дате окончания спринта, статусе спринта, а также метаинформация о фактическом времени начала и окончания спринта.

Спецификация данного микросервиса после миграции будет выглядеть следующим образом:

* POST /sprint – создание нового спринта
* GET /sprint/{sprintId} – получение информации о спринте по уникальному идентификатору.
* GET /sprint/project/{projectId} – получение списка доступных спринтов для определенного проекта.
* PUT /sprint/{sprintId} – обновление информации о проекте.
* POST /sprint/{sprintId}/start – начало спринта
* POST /sprint/{sprintId}/close – закрытие спринта

### **3.3.4 Миграция микросервиса Item Service**

Последним микросервисом требующий миграции является Item Service, который будет предоставлять возможность создавать и в дальнейшем манипулировать задачами на проекте. По сравнению с другими микросервисами при миграции данной функциональности не потребуется больших изменений.

Единственный момент, который требует тщательного внимания это авторизация действий пользователя. Данному микросервису будет необходимо выполнять запросы в другие микросервисы для проверки принадлежности задачи к спринту и проекту, а также имеет ли пользователь достаточно прав для создания или модификацию задач.

Спецификация API будет выглядеть следующим набором доступных операций:

* POST /item – создание новой задачи с указанием проекта и пользователя создателя
* GET /item/{itemId} – получение задачи по уникальному идентификатору.
* PUT /item/{itemId} – изменение информации о задаче.
* GET /item/project/{projectId} – получение задач принадлежащих указанному проекту.
* GET /item/sprint/{sprintId} – получение задач принадлежащих указанному спринту.

С помощью вышеописанных операций пользователь сможет манипулировать задачами на проекте в том числе перемещать их между спринтами и проектами по необходимости. Данный микросервис является последним в списке на миграцию изначального приложения и на данном этапе мигрированное приложение будет готово к тестированию, что будет продемонстрировано в следующем разделе.

## **3.4 Демонстрация работы приложения**

В данном разделе будет представлен пример работы с приложением, прошедшим миграцию с монолитной архитектуры на микросервисную архитектуру с добавлением новых функций. Пример использования будет содержать сценарий, в котором будет продемонстрирован основные функции приложения от регистрации нового пользователя и создания проекта до создания задач и их группирования по спринтам. Для создания и отправки запросов к микросервисам будет использован инструмент Postman.

1. Перед началом работы создадим нового пользователя в системе с именем ‘TestUser’ и паролем ‘pass’ (рис. 3.5)

Graphical user interface, text, application, email

Description automatically generated

Рис. 3.5 - регистрация нового пользователя

1. Далее получим токен для дальнейшей успешной аутентификации в системе используя ранее созданного пользователя (рис. 3.6)

Graphical user interface, text, application, email

Description automatically generated

Рис.3.6 - Получение токена

1. Далее используя токен, создадим новый проект (рис. 3.7)

Graphical user interface, text, application, email, website

Description automatically generated

Рис. 3.7 - Создание нового проекта

1. Далее добавим на ранее созданный проект другого пользователя ‘Vanila’ (рис. 3.8)

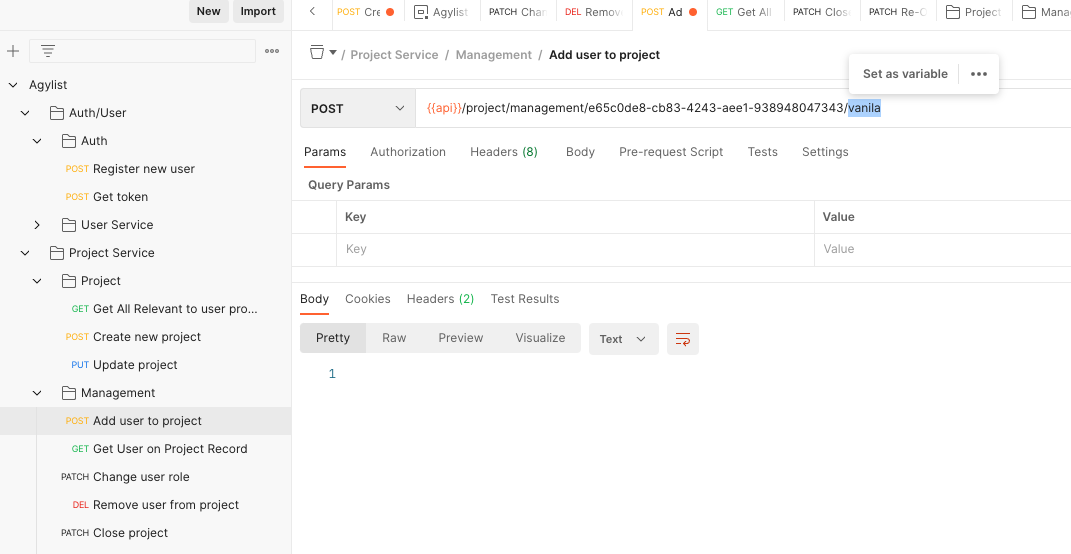
****

Рис. 3.8 - Добавление нового пользователя на проект

1. Изменим роль ранее добавленного пользователя на проект (рис. 3.9)

Graphical user interface

Description automatically generated with medium confidence

Рис. 3.9 - изменение роли пользователя

1. Создадим первый спринт на проекте (рис. 3.10)

Graphical user interface, text, application, email

Description automatically generated

Рис. 3.1 -1 создание нового спринта

1. Далее создадим новую задачу в ранее созданный спринт (рис. 3.12)

Graphical user interface, text, application, email

Description automatically generated

Рис. 3.12 - создание истории

1. В качестве финального шага начнем ранее созданный спринт (рис. 3.13)

Graphical user interface, text, application

Description automatically generated

Рис. 3.13 - старт спринта

## **3.5 Обзор структуры полученного приложения**

В результате миграции было получено приложение, состоящее из шести микросервисов, два из которых в большей степени являются инфраструктурными и помогают всей системе функционировать. Основной же монолит был разбит на четыре главных домена, которые в последствии при миграции и были использованы для определения границ микросервисов. Код разработанного приложения доступен по ссылке - <https://github.com/evgheni-panenco/agylist-microservices>.

В контексте базы данных изначального приложения она также была разделена на четыре отдельные базы данных, по базе данных на каждый из доменов. Структура различных объектов в созданных базах данных представлена ниже на рисунке 3.14.

Graphical user interface

Description automatically generated

Рис. 3.14 - схема баз данных распределенной по доменам

Схема для всей системы целиком представлена на рисунке 3.15.

Diagram

Description automatically generated

Рис. 3.15 - общая структуры приложения после миграции

# **ЗАКЛЮЧЕНИЕ**

В ходе выполнения дипломной работы на тему “Миграция монолитного приложения на микросервисную архитектуру” были рассмотрены история развития архитектуры ПО, преимущества микросервисной архитектуры, основные составляющие на которые следует обращать внимание при построение качественной архитектуры, такие как типы связей, разделение функциональности по доменам, типам коммуникаций в микросервисной архитектуре.

В практической части работы была произведена миграция монолитного приложения “Менеджер задач – Agylist” на микросервисную архитектуру. В ходе миграции изначальное приложение было разбито на домены, которые впоследствии использовались для определения границ будущих микросервисов. Помимо существующих доменов, так же был добавлен новый домен, ответственный за менеджмент проектов, а существующий домен пользователя был расширен для поддержки регистрации и аутентификации пользователей. Для облегчения работы всей системы были добавлены дополнительных два микросервиса – Service Registry и API gateway. Изменения также затронули и изначальную базу данных, которая после миграции разделилась на четыре самостоятельные базы данных.

В заключении можно сказать, что архитектура программного обеспечения развивается точно так же, как и любая другая технология в мире информационных технологий. С каждым десятилетием появляются новые проблемы, бросающие вызовы текущим системам и создание новых подходов для построения информационных систем неизбежно. Микросервисная архитектура на данный момент является наиболее популярным выбором не только при создании новых продуктов, но также и при миграции старых проектов на новую архитектуру, что и послужило практической частью данной работы.

# **БИБЛИОГРАФИЯ**

1. [Bragaru-2006] T.Bragaru, Gh.Căpățână. Teze de master, de licență și anuale. Indicații metodice. – Chișinău, CEP USM, 2006.
2. Liliana ROTARU, Maria HĂMURARU - GHID METODIC PENTRU PERFECTAREA TEZELOR DE LICENŢĂ/MASTER, - Chisinau 2012
3. Saw Newman – BULDING MICROSERVICES 2nd edition, O’Reilly Media Inc., August 2021
4. Эволюция архитектуры ПО - <https://www.linkedin.com/pulse/evolution-software-architectures-amit-sukhija-psm-pspo>
5. Domain Driven Design Wiki - <https://en.wikipedia.org/wiki/Domain-driven_design>
6. DDD Aggregates - <https://www.jamesmichaelhickey.com/domain-driven-design-aggregates/>
7. DDD Bounded Context - <https://martinfowler.com/bliki/BoundedContext.html>
8. Saw Newman – MONOLITH TO MICROSERVICES, O’Reilly Media Inc., November 2019
9. Strangler fig wiki - <https://en.wikipedia.org/wiki/Strangler_fig>
10. Типичные типы ошибок при коммуникации - Maarten van Steen and Andrew S. Tanenbaum, Distributed Systems, 3rd ed. (Scotts Valley, CA: CreateSpace Independent Publishing Platform, 2017)
11. Spring Cloud, Service Registry - <https://spring.io/guides/gs/service-registration-and-discovery/>
12. Spring Cloud, API gateway - <https://cloud.spring.io/spring-cloud-gateway/reference/html/>