Молдавский государственный университет

Факультет математики и информатики

Департамент информатики

**Чобану Артём**

**Разработка Веб - приложений на основе**

**Микросервисной Архитектуры**

***0613.1 Tehnologia informației***

Дипломная работа

Директор департамента: **\_\_\_\_\_\_\_\_\_\_\_\_\_\_** Капчеля Титу, др. конф.

Научный руководитель: **\_\_\_\_\_\_\_\_\_\_\_\_**\_\_ Латул Георгий, преподаватель.

Автор: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ ***Чобану Артём***

Кишинэу – 2022

Содержание

[Ключевые слова и сокращения 4](#_Toc101376300)

[Введение 5](#_Toc101376301)

[1. Сервис–Ориентированная Архитектура. 6](#_Toc101376302)

[2. Микросервисная Архитектура. 9](#_Toc101376303)

[3. Проектирование архитектуры приложения. 19](#_Toc101376304)

[4. Реализация проекта. 20](#_Toc101376305)

[Заключение 21](#_Toc101376306)

[Литература и документация: 22](#_Toc101376307)

[Приложения 23](#_Toc101376308)

А Н Н О Т А Ц И Я

На дипломную работу

„**Разработка Веб-приложений на основе Микросервисной Архитектуры**”

**Ключевые слова:** Сервис, Веб-Сервис, Сервис-Ориентированная Архитектура, Микросервис, Микросервисная Архитектура, Монолит.

Целью работы является изучение Микросервисной Архитектуры выявление её преимуществ и недостатков.

В дипломной работе проводится сравнение Микросервисной и Монолотной Архитектур, приводятся примеры шаблонов проектированя, лучших практик.

Для примера был разработан проект чата, состоящего из двух микросервисов. Проект разработан на платформе ASP.NET 6.0 с использованием языка программирования C# 10.0, что означает, что проект является кросплатформенным.

В работе будет представлена архитектура, части кода, а также скриншоты проекта, примеры его работы.

Дипломная работа состоит из введения, \_ глав и приложения. В первой главе рассматриваются \_\_\_\_. Во второй главе рассмотрен \_\_\_. Третья и четвертая главы посвящены проектированию и разработке приложения \_\_\_.

Дипломная работа состоит из \_\_ страниц и содержит \_\_ таблицу, \_\_ рисунков, \_\_ схему, \_\_ страниц приложений.

# Ключевые слова и сокращения

Веб-сервис, Сервис-Ориентированная Архитектура, Микросервис, Микросервисная Архитектура, Vertical Slice Архитектура, Message Queuing Service (Служба очереди сообщений), API Gateway (API Шлюз), Монолит.

CQRS – Command Query Responsibility Segregation

MQS – Message Queueing Service

SOA – Service Oriented Architecture

API – Application Programming Interface

# Введение

Современные веб-приложения становятся всё сложнее, требования растут, и касаются множества аспектов, базирующихся на предыдущем опыте, и необходимости для конкретного проекта. Ниже приведены основные требования к комплексным веб-приложениям:

1. **Отказоустойчивость** – проблемы в работе многих крупных и важных сервисов систем могут привести к огромным проблемам у пользователей по всему миру. Например, отказ крупной системы оплаты приведёт к невозможности совершать транзакции по всему миру.
2. **Простота развёртывания** – необходимое требование для избежания ошибок, а также возможности сделать развёртывание незаметным. Таким образом, новые версии приложения могут развёртываться без прекращения работы старых версий.
3. **Масштабирумость** – возможность повысить производительность пропорционально добавлением вычислительных ресурсов. Данный пункт особенно актуален с появлением облачных технологий.
4. **Заменяемость** – одно из важнейших требований к современным приложениям – это необходимость в возможности беспрепятственного изменения любых его компонентов.
5. **Простота тестирования** – повышает шанс обнаружения ошибок на ранних стадиях разработки приложения.
6. **Количество зависимостей** – нередко любое изменение в проекте приводит к неверной работе множества компонентов системы, что является огромной проблемой для её пользователей. Это происходит из-за огромного количества зависимостей компонентов друг от друга, что является признаком неверной архитектуры системы, или вовсе её остуствия.
7. **Безопасность** – сохранность как самого приложения, так и данных пользователя.

С развитием сферы Информационных Технологий появлись, и продолжают появляться различные способы соблюзти данные требования.

На данный момент, **Микросервисная Архитектура** является наиболее современным способом создания крупных и сложных информационных систем, позволяющим соблюзти все требования, приведённые выше.

# Сервис–Ориентированная Архитектура.

**Сервис-Ориентированная Архитектура** (англ. Service Oriented Architecture - SOA) – это архитектурный стиль, ориентированный на использование несвязанных между собой компонентов, называемых **Cервисами**. Часто каждый из них является полностью отдельным приложением.

В большинстве случаев в Сервис-Ориентированной Архитектуре используются **Веб-Сервисы**, являясь основной частью информационной системы.

**Веб-Сервис** (Веб-Служба) – это веб-приложение, имеющее открытый (стандартизированный) для клиента интерфейс (**API**), через который она предоставляет собственный функционал. На данный момент, самыми распространёнными являются Веб-Сервисы, следующие архитектурному стилю REST, и называющимся RESTful.

Согласно принципам Сервис-Ориентированной Архитектуры, любой Сервис должен соответствовать следующим условиям:

1. Самодостаточность и независимость от других.
2. Сокрытие внутренней реализации от пользователя.
3. Предоставление конкретного функционала доменной логики, подробно описанного и предсказуемого, т.е. предоставление интерфейса для взаимодействия с ним.
4. Каждый Сервис может состоять из нескольких сервисов.

В основе Сервис-Ориентированной Архитектуры лежит принцип **независимости** веб-сервисов, дающий следующие преимущества:

* **Независимая разработка каждого из Сервисов:** возможность разработки каждого Сервиса отдельной командой, с использованием совершенно разных технологий.
* **Простота тестирования**: каждый Сервис является отдельным приложением, что позволяет тестировать их индивидуально, без внимания к другим компонентам информационной системы.
* **Заменяемость**: существует возможность заменить любой Веб-Сервис на другой, имеющий аналогичный функционал.
* **Более простое внесение изменений**: возможность независимого изменения функционала любого из веб-сервисов, что абсолютно не затрагивает другие Веб-Сервисы.

Кроме Веб-Сервисов, возможно использовать различные технологии для реализации Сервисов. Например, компонентом информационной системы может быть Desktop приложение, общающийся с Веб-Сервисом для получения или изменения необходимых данных.

Сервис-Ориентированная Архитектура является набором **абстрактных** принципов, которым должны следовать Сервисы, она не зависит от используемых технологий, и допускает любые из них. Благодаря этому, возможно использование совершенно различных технологий для создания Сервисов, созданных таким образом, чтобы следовать принципам Сервис-Ориентированной Архитектуры.





На данных рисунках представлено 2 случая Систем Сервис-Ориентированной Архитектуры, включающих:

1. Один Сервис.
2. Два Сервиса.

В обоих случаях оба Сервиса доступны множеству клиентов, данные Сервисы являются независимыми друг от друга, и взаимодействуют с одной общей базой данных.

Не смотря на то, что в процессе работы встречаются совершенно разные задачи, существуют общие принципы Сервисно-Ориентированной Архитектуры, многие из которых схожи:

* **Абстракция:** внутреннее устройство Сервиса должно быть скрыто от пользователей. Взаимодействие производится через открытый интерфейс.
* **Независимость:** каждый Сервис должен быть самодостаточным и самостоятельно управлять функциональностью.
* **Нормализация:** в целях уменьшения избыточности функционала, Сервис необходимо разделить на несколько более мелких компонентов. Некоторый функционал может быть выделен в полностью отдельный Сервис.
* **Отсутствие состояния:** Сервис должен выполнять операции тогда, когда это необходимо, и не хранить состояние постоянно, уменьшая количество потенциальных ошибок, и потребляя меньше ресурсов. Таким образом, Сервисы часто могут выполнять **Запросы** или **Команды** пользователя, что следует принципу **CQS** (Command-Query Separation).
* **Открытый интерфейс:** каждый Сервис имеет открытую спецификацию интерфейса (API), через который происходит взаимодействие с клиентом.
* **Повторное использование:** разделение логики на компоненты позволит заново их использовать, без необходимости создания новой имплементации похошего функционала.
* **Долголетие:** старые функции Сервиса должны быть доступны наряду с новыми, не заставляя клиента обновлять ПО.
* **Доступность:** возможность использования Сервиса вне зависимости от местанахождения в сети.

На данном рисунке изображён пример Информационной Системы магазина, который использует Сервис-Ориентированную Архитектуру.



# Микросервисная Архитектура.

**Микросервисная Архитектура** – это развитие Сервис-Ориентированной Архитектуры, ориентированный на разделение Сервиса на несколько **слабо связанных** друг с другом Сервисов.

Согласно принципам Микросервисной Архитектуры, любой Микросервис должен сответствовать следующим условиям:

* **Наличие собственного хранилища данных**: два Микросервиса не должны иметь общую базу данных.
* **Разделение ответсвенностей:** каждый Микросервис имеет ограниченный спектр задач и отвечает за одну область.
* **Взаимодействие с другими Микросервисами,** но технически каждый из них является отдельным Сервисом.
* **Отсутствие излишнего взаимодействия:** Микросервисы являются частью одной системы, но не должны быть тесно переплетены между собой**.**

Последний принцип является одним из главных для Микросервисной Архитектуры. При разработке Системы, необходимо добиться **максимально возможной независимости** каждого из Микросервиса. Таким образом, при прекращении работы одного из Микросервисов, остальные будут **продолжать свою работу**, хоть и не полностью.

Противоречие данного принципа состоит в том, что полная независимость Сервиса **не включает** его в Микросервисную Архитектуру приложения. Это значит, что Микросервисы **должны взаимодействовать** друг с другом, **но настолько мало**, насколько это возможно.

Как и в случае с Сервис-Ориентированной Архиитектурой, разработчик **оставляет за собой** решения о конкретной имплементации архитектуры разрабатываемой им Системы. Это значит, что некоторые принципы **могут нарушаться** в пользу некоторых преимуществ, таких как уменьшения сложности или повышение производительности.

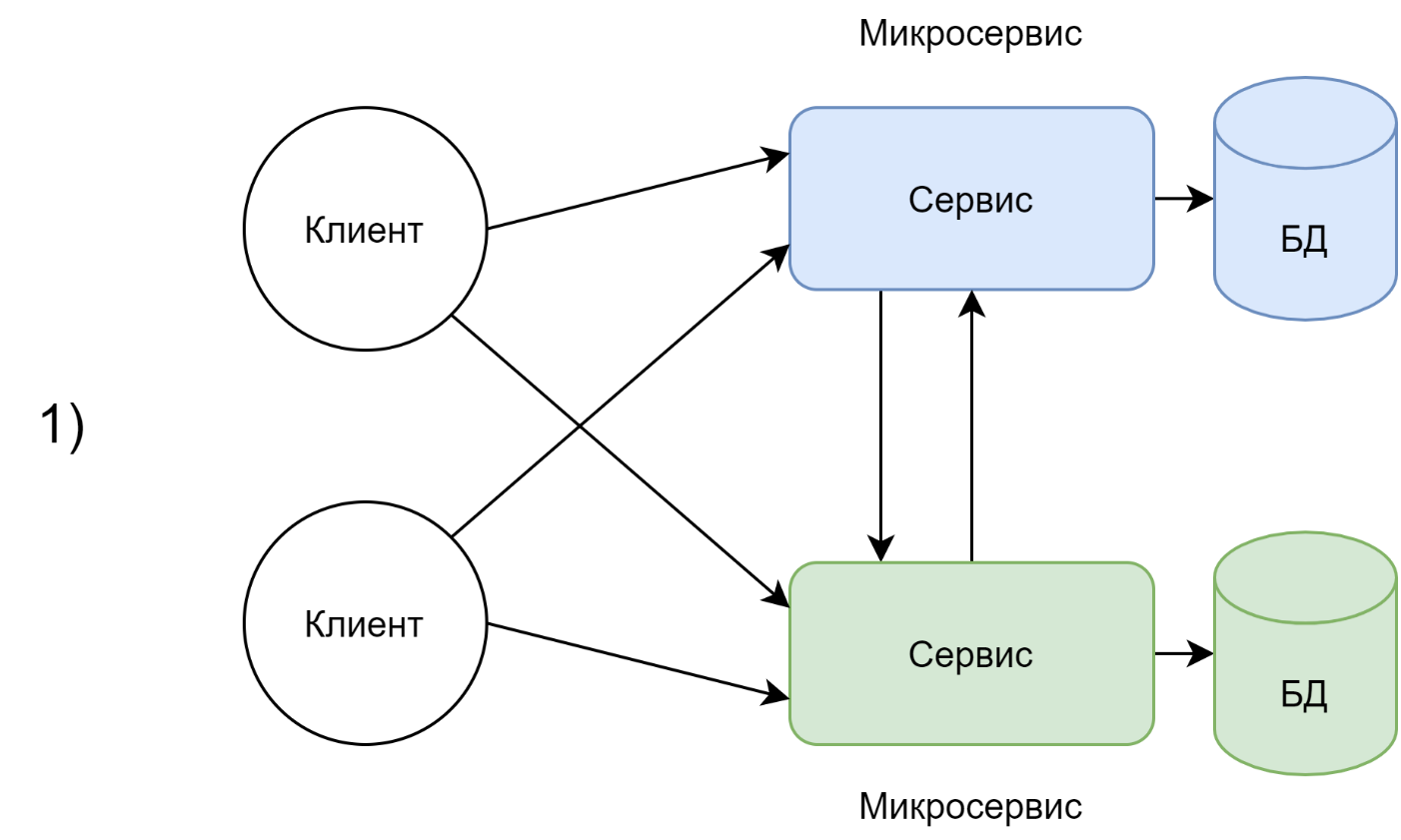
Преимущества Микросервисной Архитектуры:

* **Масштабируемость**: возможно отдельное масштабирование каждого Микросервиса.
* **Модульность**: понимание приложения становится гораздо проще, ввиду разделения приложения на множество небольших Сервисов.
* **Разнообразие технологий**: микросервисы могут быть разработаны с использованием разных технологий.
* **Параллельная разработка**: каждый микросервис может разрабатываться разной командой.
* **Изоляция данных**: каждое хранилище данных используется только одним Микросервисом.

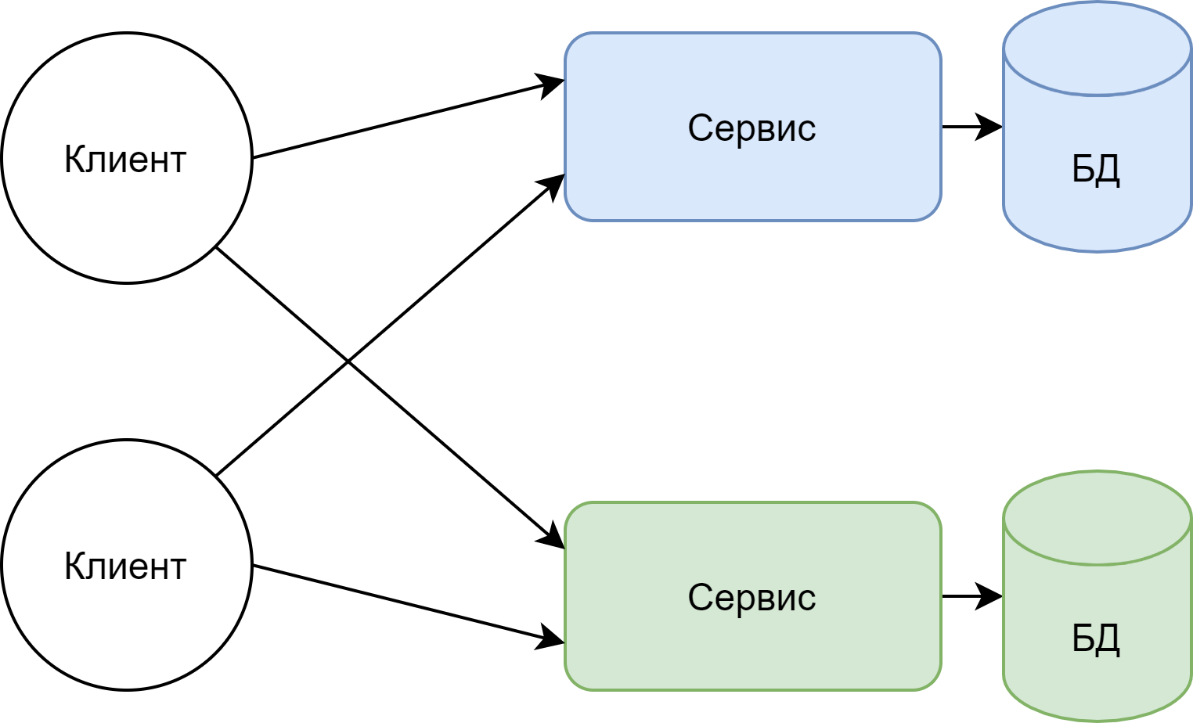
Недостатки Микросервисной Архитектуры:

* Сложность разработки приложения с самого начала.
* Сложность в обеспечении целостности данных, ввиду наличия множества хранилищ.
* Необходимость в специалистах в данной области, таких как Архитекторы.
* Необходимость создания способов взаимодействия, а также связи между Микросервисами.
* Сетевые задержки, вызванные большим количеством обменов сообщениями.
* Обновления могут вызвать проблемы с совместимостью Микросервисов.
* Непредстказуемость: ввиду разнообразия задач на каждом проекте, возникновение совершенно разных и неповторяющихся проблем является постоянным явлением.

Система с Микросервисной Архитектой, в которой участвуют два Микросервиса:



Как было упомянуто ранее, Система, в которой отсутствует взаимодействие между Сервисами, не является Микросервисной:



**Микросервисная** **Архитектура** противопоставляется **Монолитной** **Архитектуре**, подразумевающей разработку Сервиса как единого приложения.

**Сравнение Микросервисной и Монолитной Архитектур**

|  |  |  |
| --- | --- | --- |
|  | **Монолит** | **Микросервисы** |
| **Развёртывание** | Необходимость развёртывания только одной системы. | Необходимость развёртывания всех микросервисов, входящих в систему. |
| **Масштабирование** | Сложное масштабирование | Масштабирование каждого из микросервисов по-отдельности, а также развёртывание нескольких экземпляров. |
| **Внесение изменений** | Сложное внесение изменений и добавление новых функций. | Каждый микросервис может быть изменён независимо от других. |
| **Устойчивость к ошибкам в изменениях** | Любое изменение может затронуть остальной функционал всего приложения. | Проблема в одном микросервисе не затрагивает другие. |
| **Отказоустойчивость** | Отказ одной части может привести к отказу всей системы. | Отказ одного микросервиса не прекращает работу других. |
| **Тестирование** | Необходимость полного тестирования приложения  (End-to-End testing) | Индивидуальное тестирование каждого микросервиса в отдельности. |
| **Безопасность** | Проще организовать безопасность внутри одной системы. | Сложная организация безопасности, с использованием **Шлюза** (Gateway) |
| **Разработка** | Члены команды должны иметь опыт работы со всей системой и огромной базой данных. | Каждый микросервис разрабатывается отдельной командой. |
| **Выбор технологий** | Вся система разработана с использованием одной конкретной технологии. | Каждый микросервис может использовать собственную технологию. |
| **Обновление технологий** | Сложное обновление до новых версий платформ разработки ПО. | Индивидуальное обновление каждого микросервиса, либо создание полной замены старому. |
| **Сложность разработки** | Низкая | Высокая |

Из сравнения **Монолитной Архитектуры** с **Микросервисной** видно, что обе архитектуры имеют как преимущества, так и недостатки, а значит ни одна из них не является лучшей во всех ситуациях. Можно выделить определённые критерии выбора архитектуры в зависимости от проекта.

* **Микросервисная Архитектура** подходит для **крупных** проектов, состоящих из множества компонентов, и над которыми должно работать множество команд, состоящих из 6-12 человек. Подобную Информационную Систему почти невозможно разработать в виде Монолита.

В данном случае, выбор Микросервисной Архитектуры позволит снизить трудозатраты каждой команды в разработке и тестировании, организации самого процесса разработки, развёртывания каждого из них. Кроме того, это снижает риски, связанные с дефектами системы, а полный отказ является практически невозможным.

* **Монолитная Архитектура** подходит для более **мелких** проектов, разделение которых на микросервисы является избыточным.

Это добавит излишней сложности, так как микросервисам необходимо постоянно обмениваться данными.

Кроме того, ни одно из преимуществ использования микросервисов не проявит себя. Например, внесение изменений в мелкий проект является простой операцией.

* Также может подойти выбор разработки Монолита на **начальном этапе** разработки приложения, с **последующим его разделением** на несколько Микросервисов.

Данное решение позволит снизить сложность и увеличить скорость начальной разработки системы – тогда, когда нет прямой необходимости в Микросервисах. Выбор именно такого сценария развития проекта зависит от потребностей разработчика пректа.

Например, начальная разработка Монолита является полезным при необходимости **очень быстрой** разработки, а также при отсутствии уверенности в длительности проекта. Все преимущества Микеросервисной Архитектуры проявляют себя, в первую очередь, в **долгосрочной** перспективе.

Пример архитектуры магазина, использующий Микросервисную Архитектуру:



**Использование Сервисов Очереди Сообщений**

**Очередь Сообщений (Message Queue)** – это форма общения между Сервисами с помощью помещения набора сообщений в очередь(или несколько очередей) с последующей их отправкой **подписчику** на одну из очередей сообщений. Обычно, каждая очередь имеет собственный идентификатор.

Каждое сообщение может быть полученно только **одним** из подписчиков и только **единожды**. После этого, происходит удаление сообщений.

Очередь сообщений используется, в основном, для общения микросервисов.



**Сервис Очереди Сообщений (Message Queue Service)** – это Сервис, имеющий функционал по созданию очередей сообщений, добавлению сообщений в очередь, подписки на очереди, а также отправки сообщений их подписчикам.

Существует множество Сервисов Очереди Сообщений от разных компаний, имеющие разный функционал. Многие расширяют приведённое выше определение, добавляя возможности обработки бизнес логики, инициации сообщений через определённый период времени, и так далее.

Побольшей части, данные Сервисы используются для отправки **событий**. Таким образом поддерживается целостность данных, распределённых на несколько микросервисов. Примером такого события может быть обновление цены на товар, или создание заказа.

**Выбор Сервиса Очереди Сообщений**

Примером одного из наиболее популярных Сервисов Очереди Сообщений является **RabbitMQ**. Один из Сервисов, насыщенных разнообразным функционалом является **BizTalk**.

Кроме того, некоторые облачные сервисы предоставляют собственный встроенный MQ сервис. Одним из них является **Amazon SQS (Simple Queue Service).**

Выбор Message Queuing Сервиса зависит от потребностей в его функционале. Сервисы, имеющие большее количество возможностей, как правило, обладают более низкой производительностью. Таким образом, RabbitMQ является более легковестным решением, по сравнению с BizTalk, однако не имеет возможности самостоятельно инициировать отправку сообщений.

Мноие из данных функций являются отдельными службами облачных Сервисов. Например, Microsoft Azure имеет сервис Web Job, предназначенный для повторной инициации событий в течение определённого времени.

**Паттерн API Gateway (Шлюз)**

Предположим, что существует приложение, в архитектуру которого входит несколько Сервисов, с которыми клиенту необходимо взаимодействовать напрямую:



У подобной архитектуры существуют недостатки, связанные с прямым общением клиента с каждым из Сервисов:

* Необходимость разработки и конфигурации клиента таким образом, чтобы он мог обращаться к каждому Сервису в отдельности. Это может стать большой проблемой, если в состав ИС входит большое количество Сервисов.
* Клиент должен использовать Сервис с заранее известным API. При любом изменении любого из Сервисов, необходимо произвести соответствующие изменения в клиенте. При крупном рефакторинге, изменения в клиенте также будут достаточно крупными.
* Сервисы могут использовать протоколы, недоступные клиенту (например, если в качестве клиента выступает браузер), такие как AMQP или gRPC.
* Необходимость совершать множество запросов (обычно синхронных), что может снизить производительность.
* Необходимость создания системы безопасности в каждом из данных Сервисов, при любом изменении в которой необходимо изменять и развёртывать все Сервисы.

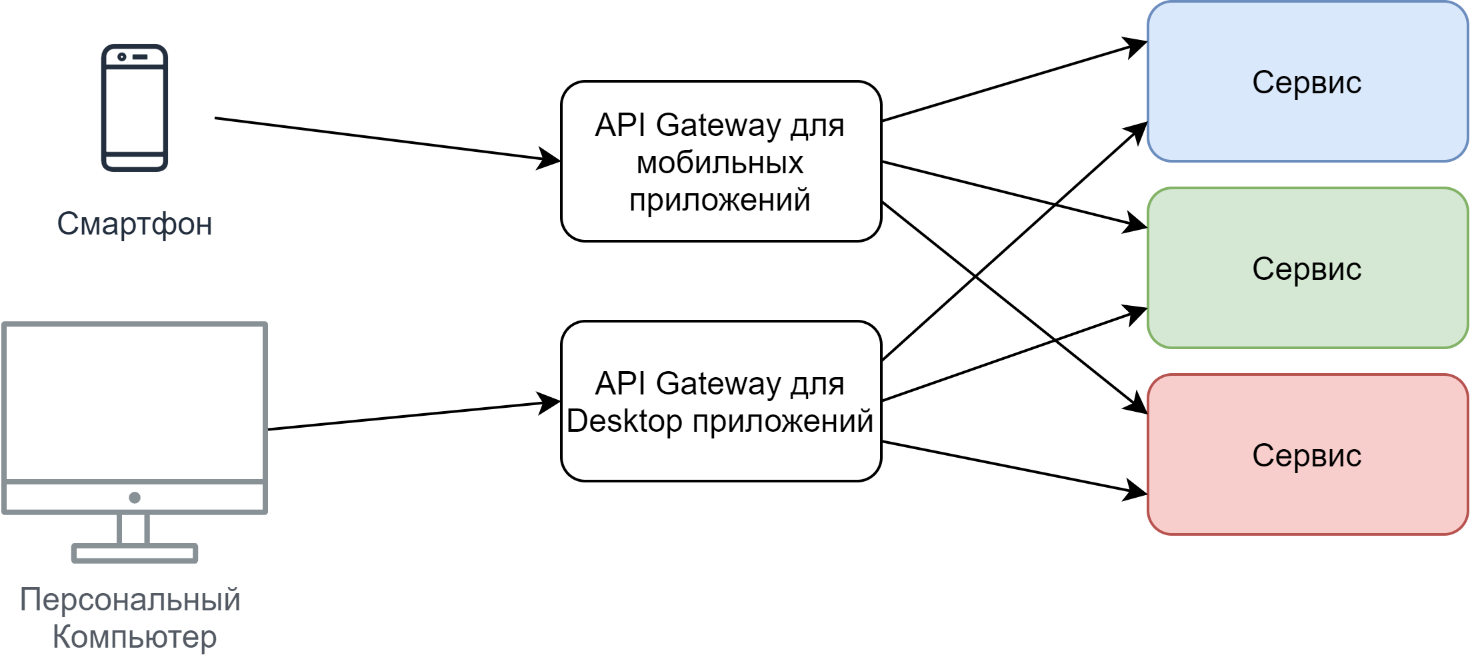
Шаблон проектирования **API Gateway (Шлюз)**, базирубшийся на паттерне **Фасад**, создан для решения данных проблем, путём введения промежуточного Сервиса, инкапсулирующего все остальные:



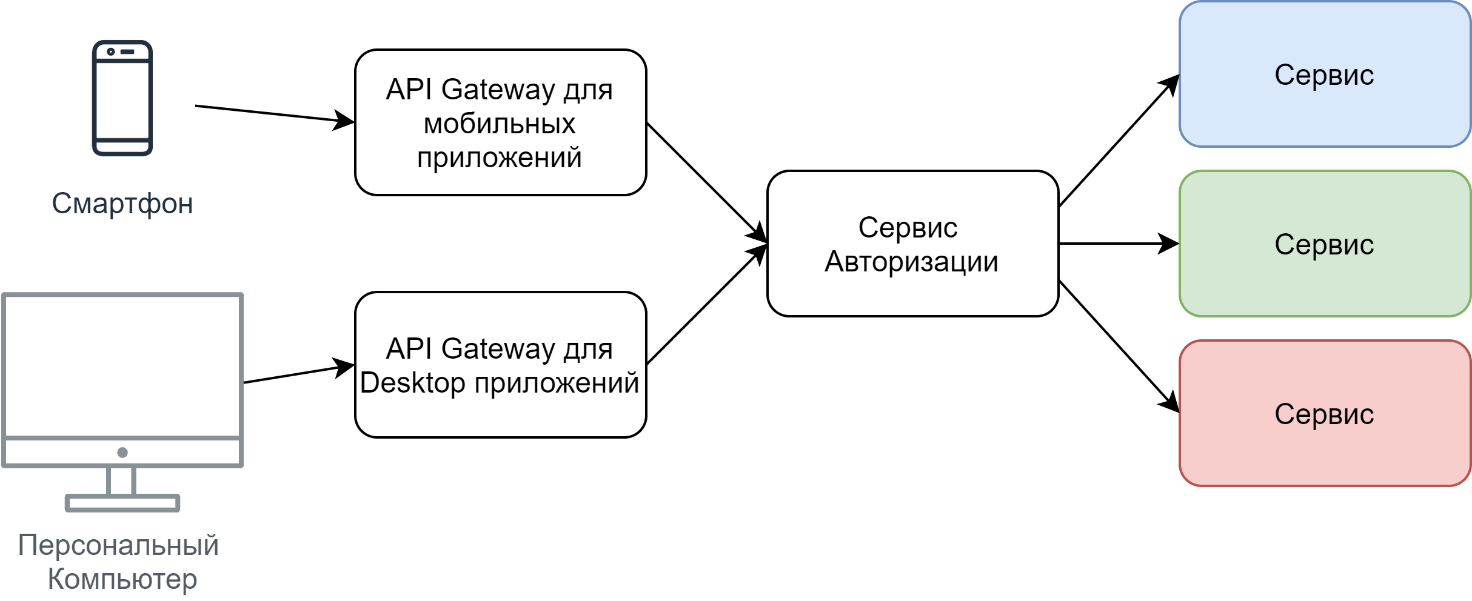
Необходимо уточнить, что API Gateway – это входная точка для **группы** Сервисов, но не для всех. Это значит, что при наличии большой Системы, необходимо создать **несколько** Шлюзов, группирующих Сервисы по таким критериям как пользовательский опыт.

На данный момент, принцип группировки Шлюзов не является стандартом, и должен выявляться в зависимости от задач.

Шаблон проектирования **Backends for Frontends** является одним возможных принципов группировки для кросплатформенных приложений. Возможности мобильных устройств отличаются от возможностей Персональных Компьютеров, из-за чего требования к ним отличаются. Данный шаблон проектирования позволяет обобщить логику, но при этом обслуживать приложения с разными требованиями.



Также, API Gateway может выступать в качестве посредника, предназначенного для авторизации:

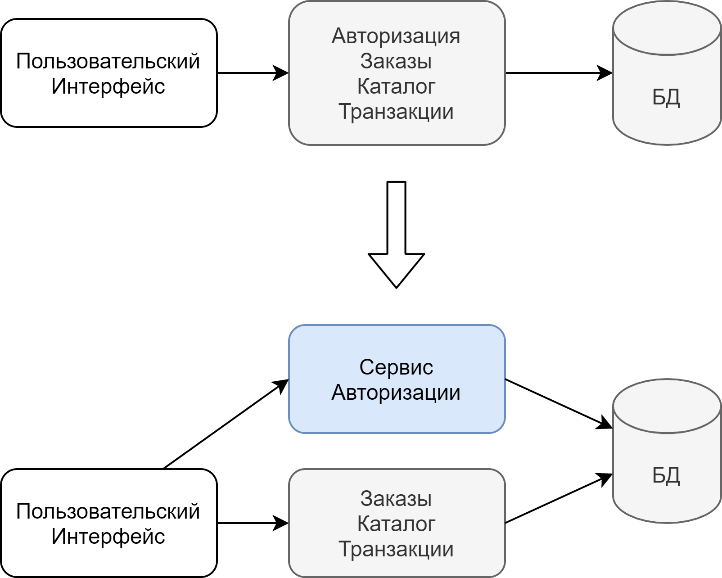


**Переход от Монолита к Микросервисам**

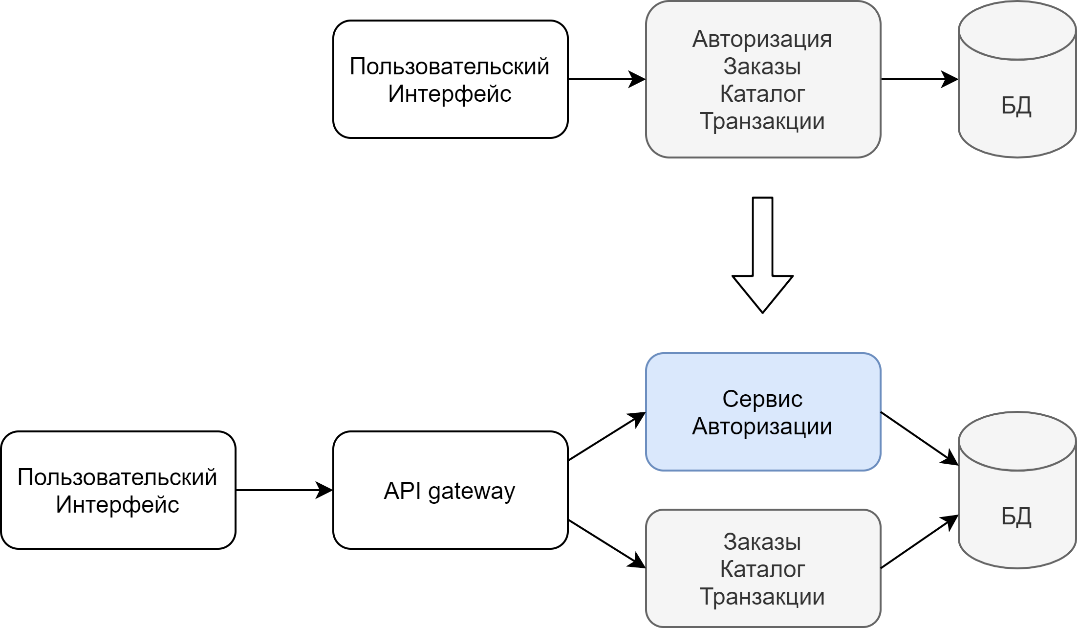
Переход от Монолита к Микросервисам является достаточно сложным процессом. Единственный способ – это итеративное разделение Монолита на Сервисы:



В процессе разделения, допускается использование одной и той же базы данных для нескольких Микросервисов. В таком случае, база данных должна быть разделена логически, но не физически:



Возможно и совместное использование одного из Микросервисов вместе с монолитом в целях тестирования, выделяя ему определённый процент нагузки. Подобную систему можно тестировать даже в Production версии.



# Проектирование архитектуры приложения.

Задача данной дипломной работы – разработка чата, использующего Микросервисную Архитектуру. Таким образом, станет возможно на практическом примере показать преимущества и недостатки использования Микросервисов.

Требования к проекту:

* Кросплатформенность
* Отзывчивость приложения
* Доступность из любой точки мира
* Наличие системы Регистрации и Авторизации пользователей
* Авторизация происходит один раз (нет необходимости авторизации при каждом запуске приложения)
* Для соблюдения GDPR, пароль пользователя не сохраняется ни в сессии, ни в базе данных.
* Возможность использования облачных сервисов.
* Возможность наиболее простого внесения любых изменений

Функционал проекта:

* Регистрация
* Авторизация
* Выход из аккаунта
* Просмотр списка чатов
* Создание чата
* Присоединение к выбранному чату
* Отправка сообщений в чат
* Получение сообщений от других пользователей в чате.

**Рассмотрение требований**

* Кросплатформенность **клиентской части (Front-End)** проще всего достичь при разработке Веб-приложений, доступных из браузера. Таким образом, приложение будет доступно с любой платформы, так как оно будет являться веб-сайтом, а не приложением, которое разрабатывается и устанавливается отдельно на каждое устройство.

Кросплатформенная Серверную часть (Back-End) достигается выбором платформы и языка программирования. Например, платформа .Net Framework не является кросплатформенной. Сервера, на которых работают приложения, созданные на данной платформе, могут иметь только Операционную Систему Windows.

Выбор технологий:

Описание требований, задач, функционала.

Описание двух микросервисов, общающихся через RabbitMQ.

Графическое представление архитектуры приложения

Архитектура каждого микросервиса в отдельности.

Vertical Slice Архитектура, CQRS, возможность разделения ответственностей на Read/Write.

# Реализация проекта.

Скриншоты, демонстрация функций.

# Заключение

Короткий повтор преимуществ и недостаткой, а также описание проектов, для которых лучше подходит микросервисная или монолитная архитектура.

# Литература и документация:

<https://netflixtechblog.com/optimizing-the-netflix-api-5c9ac715cf19>

<https://www.udemy.com/course/microservices-architecture-and-implementation-on-dotnet>

<https://aws.amazon.com/sqs/>

<https://www.rabbitmq.com/documentation.html>

<https://www.ibm.com/uk-en/cloud/learn/soa>

<https://microservices.io/patterns/>

<https://microservices.io/patterns/data/api-composition.html>

<https://microservices.io/patterns/decomposition/decompose-by-business-capability.html>

<https://jimmybogard.com/vertical-slice-architecture/>

<https://cloud.google.com/api-gateway/docs/authenticate-service-account>

<https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends>

<https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices>

<https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications>

<https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/multi-container-microservice-net-applications/rabbitmq-event-bus-development-test-environment>

# Приложения

Рисунок 1.3 – Сервис-Ориентированная Архитектура ИС магазина.

