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](#_Toc103329792)

[Введение 5](#_Toc103329793)

[1. Сервис–Ориентированная Архитектура. 6](#_Toc103329794)

[2. Микросервисная Архитектура. 9](#_Toc103329795)

[3. Определение требований к проекту. 19](#_Toc103329796)

[4. Архитектура проекта. 29](#_Toc103329797)

[5. Реализация проекта. 33](#_Toc103329798)

[Заключение 46](#_Toc103329799)

[Литература и документация: 47](#_Toc103329800)

[Приложения 48](#_Toc103329801)

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.

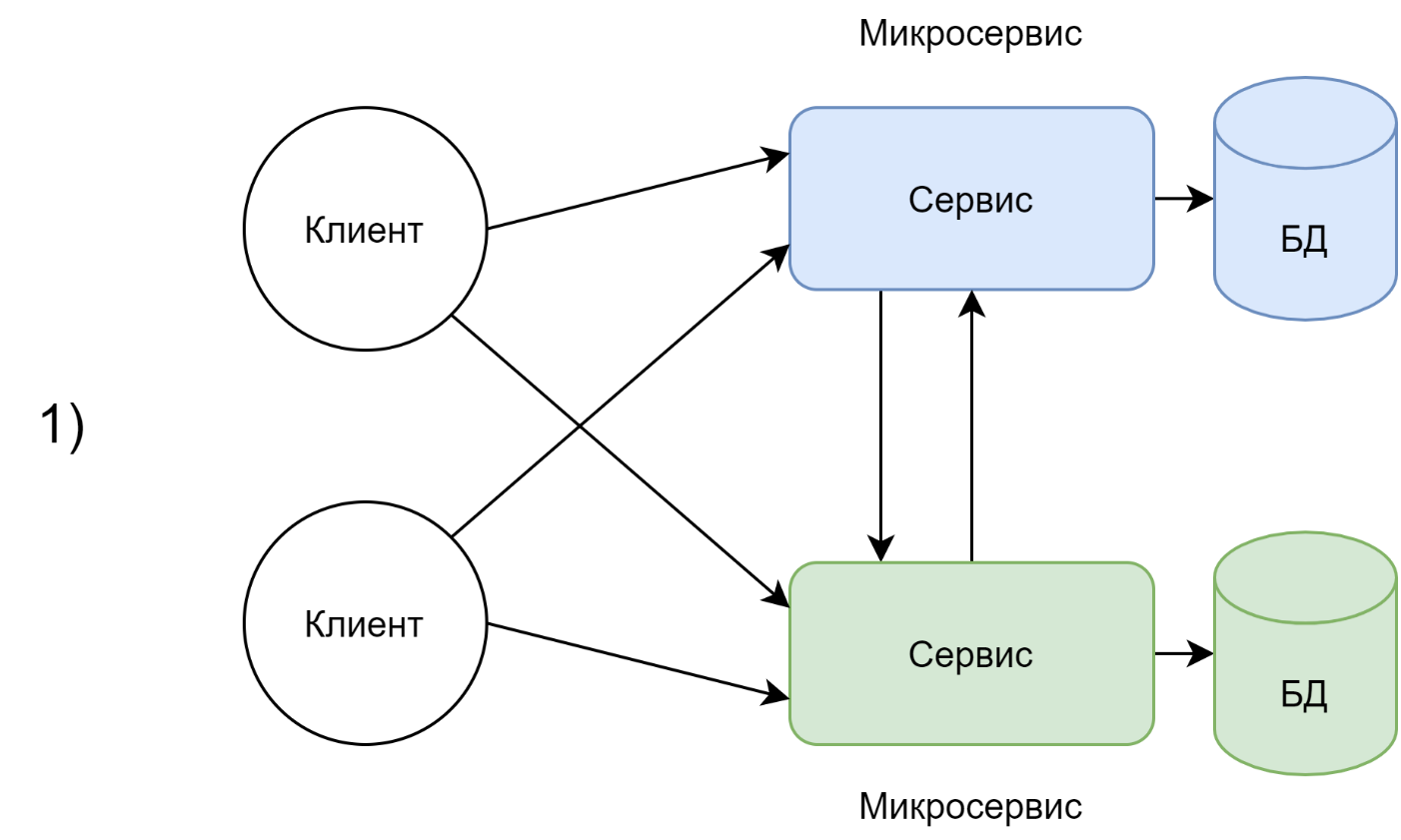
Преимущества Микросервисной Архитектуры:

* **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.

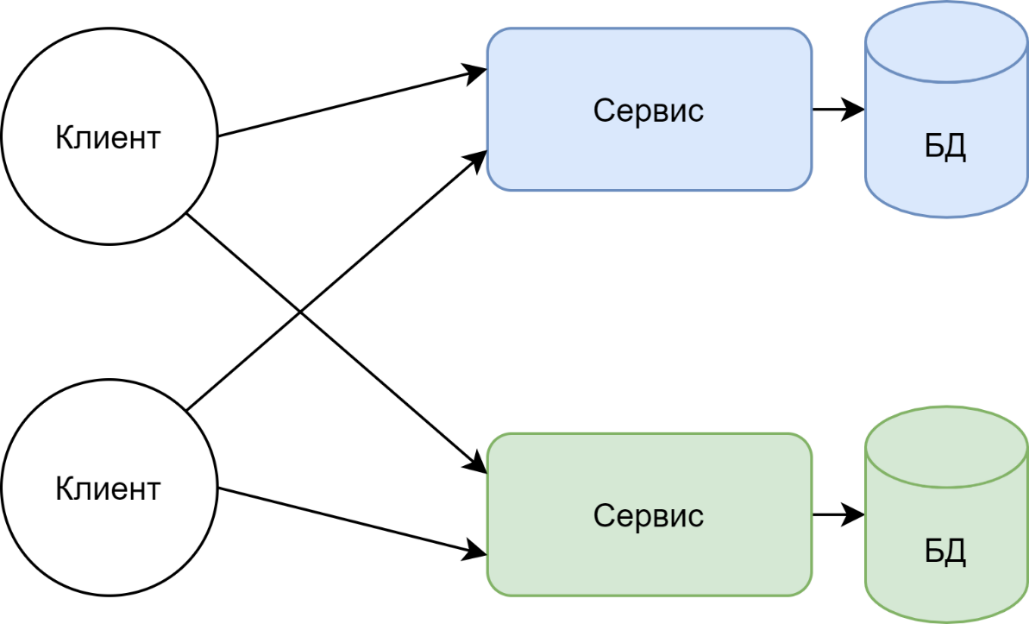
Недостатки Микросервисной Архитектуры:

* 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.

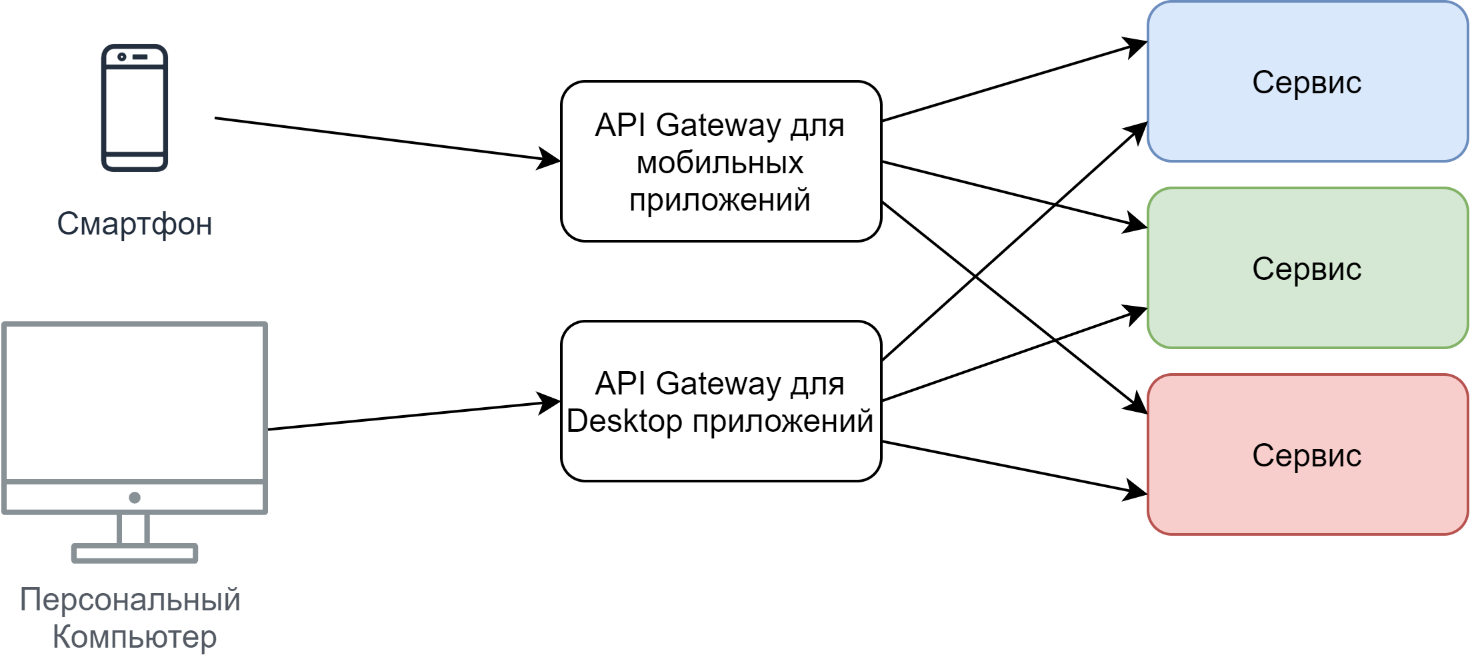
Шаблон проектирования **API Gateway (Шлюз)**, базирубшийся на паттерне **Фасад**, создан для решения данных проблем, путём введения промежуточного Сервиса, инкапсулирующего все остальные:



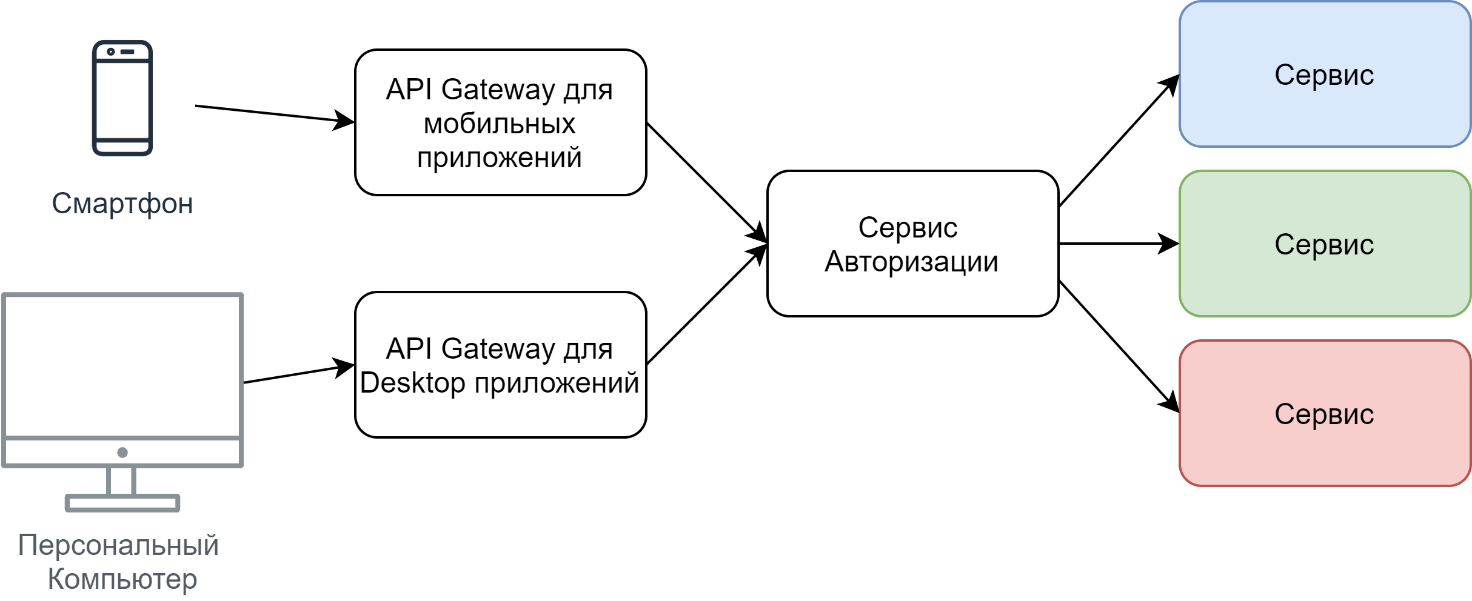
Необходимо уточнить, что API Gateway – это входная точка для **группы** Сервисов, но не для всех. Это значит, что при наличии большой Системы, необходимо создать **несколько** Шлюзов, группирующих Сервисы по таким критериям как пользовательский опыт.

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:

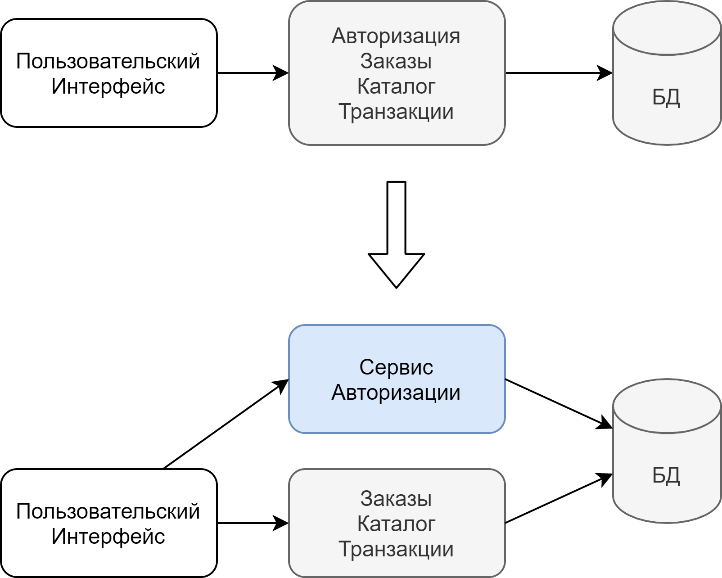


**Переход от Монолита к Микросервисам**

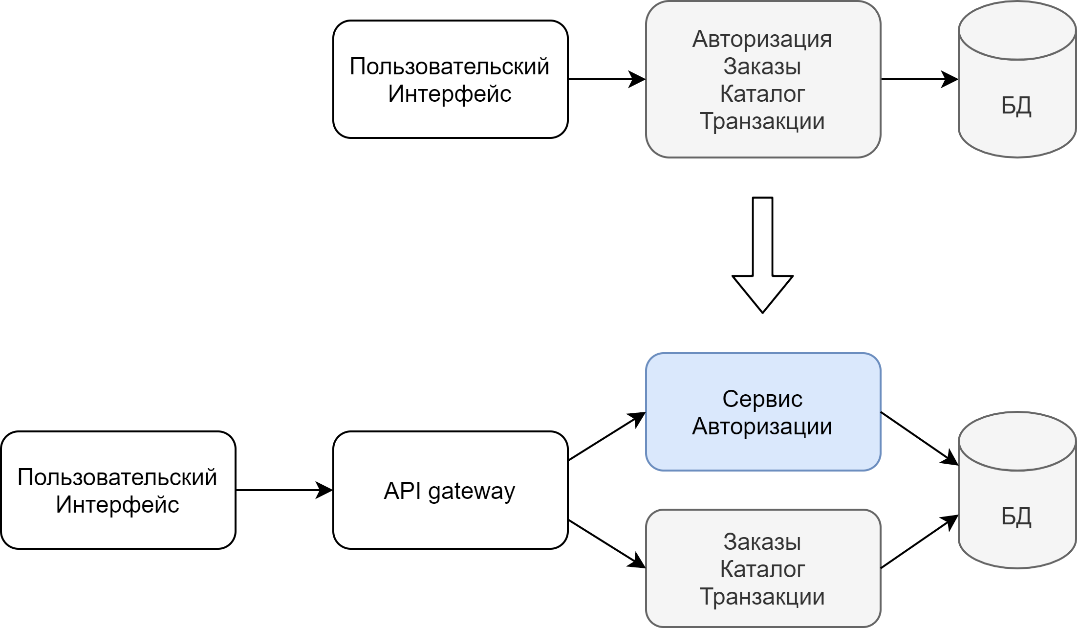
Переход от Монолита к Микросервисам является достаточно сложным процессом. Единственный способ – это итеративное разделение Монолита на Сервисы:



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.



# Определение требований к проекту.

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

Требования к проекту:

* Кросплатформенность
* Доступность из любой точки мира
* Отзывчивость приложения
* 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.
* Невозможность выполнить действия от лица другого игрока.
* 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**

Полный стек выбранных технологий:

**Front-End**: Angular 13.3

**Back-End**: .Net 6, C# 9.0

**СУБД**: SQL Server 2019

**Message Queue Service**: RabbitMQ

* 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.

Для Front-End разработки был выбран Angular - один из современных фреймворков.

Кросплатформенность Серверной части (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.

Для Back-End разработки была выбрана платформа .Net 6, которая является развитием .Net Core, и является кросплатформенной.

* Отзывчивость Веб-Приложения можно достичь, создавая Single Page Application (SPA) с использованием одного из современных Front-End фреймворков: Vue, React, Angular и другие.

Angular содержит возможности по ленивой загрузке (lazy loading) модулей, что позволяет снизить обьём отправляемых браузеру данных. Кроме того, он содержит возможности для кастомизации, и содержит компонентный подход.

* Система Авторизации пользователей использует технологию JWT –

Json Web Token, благодаря которойданыне пользователя не сохраняются после его аутентификации. Вместо этого, при всех последующих операциях используется ранее полученный токен авторизации.

* При регистрации, пароль пользователя не сохраняется в сыром виде. Вместо этого, он проходит этап хеширования с солью. Результат сохраняется в базу данных.

Во время аутентификации, происходит хеширование введённого пароля, и дальнейшее сравнение полученного хеша с другим, уже имеющимся в базе данных.

* Веб-Сервисы, созданные на .Net 6 возможно запускать вместе с технологиями Docker и Kubernetes, в том числе и в Linux контейнерах. Таким образом, данный проект очень просто использовать в одном из доступных облачных сервисов.
* Безопасное внесение изменений в проект – одна из самых больших проблем программирования в целом, для решения которой были созданы принципы SOLID, множество шаблонов проектирования, а также разные архитектуры. Данные направления до сих пор находятся в процессе развития.

Принципы SOLID и шаблоны проектирования должен использовать любой компетентный разработчик ПО.

На крупных проектах существует необходимость в наличии Архитектора, который специализируется в данном направлении.

Для данного проекта была выбрана **Vertical Slice** Архитектура, так как она позволяет разрабатывать каждый Endpoint индивидуально и **независимо** от других. Следовательно, любой из них с лёгкостью можно заменить или удалить в любой момент времени, или даже переместить в другой Сервис.

**Выбор Архитектуры Сервиса**

* **N-Уровневая Архитектура (N-Tier Architecture)**

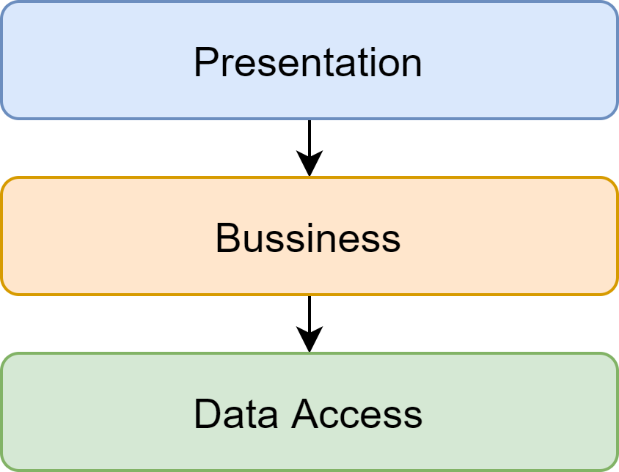
Простейшая Архитектура, призванная разделить приложение на модули в соответствии с определённой структурой – это **N-Уровневая** **(N-Tier)** Архитектура. Она делит приложение на **слои** и **уровни**.

**Уровень (Tier)** – это обозначение части системы, отделённой физически, на разных рабочих машинах.

**Слой (Layer)** – это логическое разделение частей приложения на отвественности.

Данная Архитектура подразумевает, что каждый Слой или Уровень **не может** ссылаться или использовать предыдущий. Более верхний слой использует один из более нижних.

Нередко данная архитектура используется в конкретном частном случае - с наличием только трёх слоёв, называя её **Трёхслойной Архитектурой**:



Наиболее верхний слой называется слоем **Презентации (Presentation)** или **Пользовательского Интерфейса (UI)** и отвечает за отображение информации на экране, и взаимодействие с пользователем.

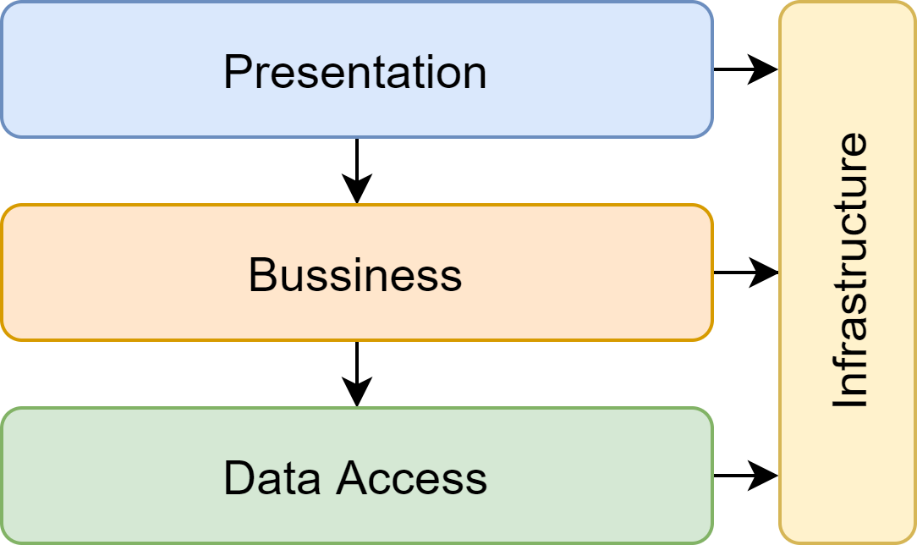
Средний слой – называется **Бизнес** слоем и отвечает за логические функции приложения, связанные с процессами, происходящими на организации, для которой разрабатывается приложение.

**Слой Доступа к Данным (Data Access)** – наиболее нижний слой, отвечающий за способ взаимодействия с хранилищем данных.

Таким образом, Data Access слой предоставляет возможности по использованию базы данных, Business слой использует эти их в связи с правилами работы приложения, а Presentation слой вызывает процессы из Business слоя в соответствии с командой пользователя.

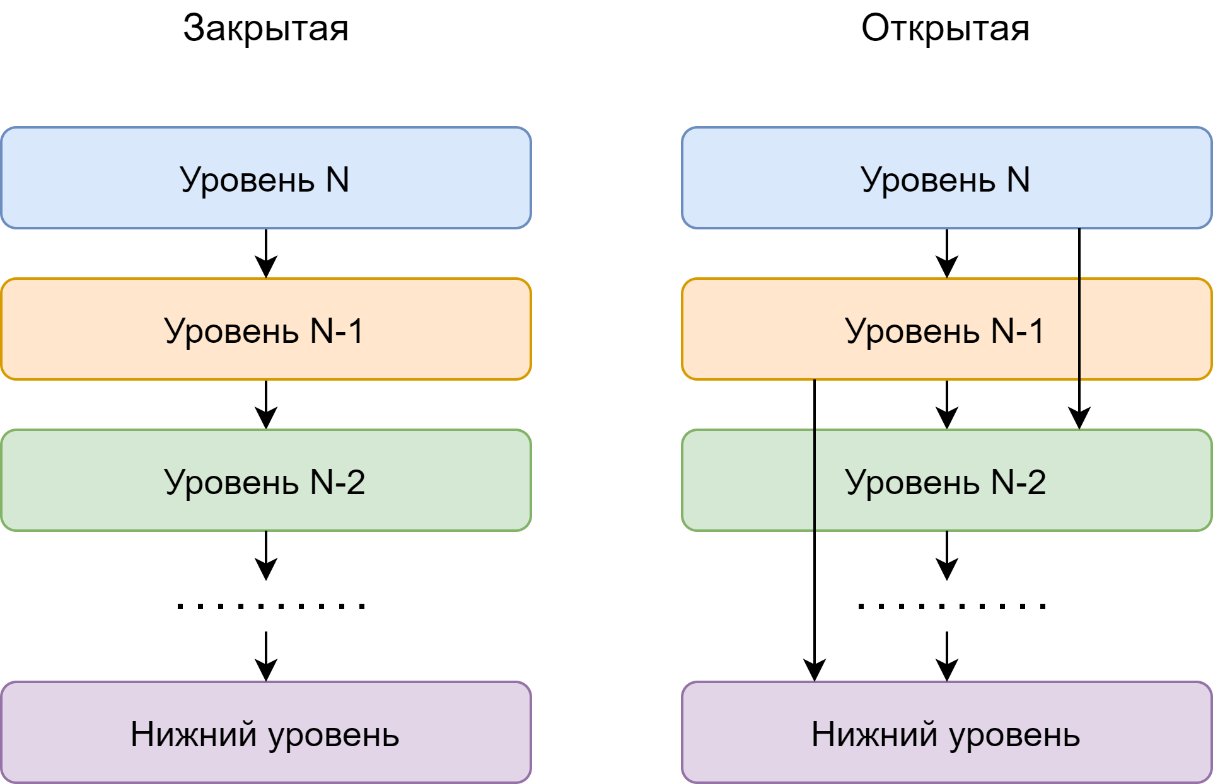
В качестве улучшения данной архитектуры добавляют слой **Инфраструктуры**, содержащего инструменты приложения, с помощью которых оно работает, любые технические решения, не содержащие бизнес логику и не связанные ни с одним из остальных слоёв приложения. Например

Инфраструктура может использоваться **любым** из имеющихся слоёв.



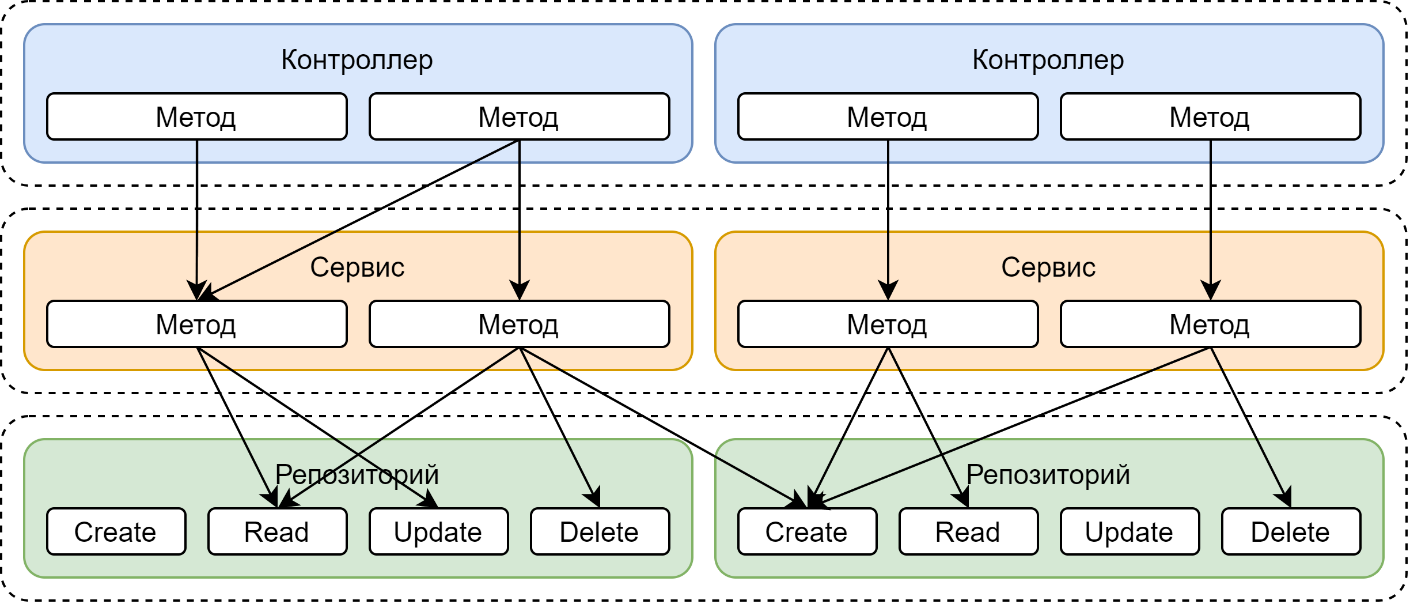
Существует два вида N-Уровневой Архитектуры:

* **Закрытая** – каждый слой может ссылаться только на следующий.
* **Открытая** – каждый более верхний слой может ссылаться на любой из более нижних слоёв.



Главным **преимуществом** данной архитектуры явлется её **простота**, вследствие чего она подходит для разработки **небольших** приложений.

Её **недостатком** является **громоздкость** слоёв в случае больших приложений, а также огромное количество зависимостей верхних слоёв от одних и тех же компонентов нижних слоёв. Это повышает шанс возникновения проблем при необходимости внесения изменений в нижние слои. Заменить один из нижних слоёв зачастую вовсе не представляется возможным без полного рефакторинга.



* **Чистая Архитектура (Clean Architecture)**

**Чистая Архитектура** – частный случай открытой N-Уровневой Архитектуры, в котором более внутренние слои содержат **логику и правила работы** приложения, а более внешние – их **инструментами**, которые зависят от внутренних слоёв.

Чаще всего, данную архитектуру изображают с применением **четырёх** слоёв:

* + **Domain** – содержит обьекты, представляющие собой правила работы области применения (Бизнес правила), такие как: сущности, перечисления.
  + **Application** – представляет собой логику конкретного приложения, использующей Domain обьекты. Он также содержит необходимые ему интерфейсы (например, для доступа к данным).
  + **Infrastructure** – содержит имеплементации интерфейсов из слоя Application.
  + **WebUI** – точка входа в приложение, использующая классы из Application, **передавая им имплементации** необходимым им интерфейсов из слоя инфраструктуры.

Графическое изображение данной архитектуры от Роберта Мартина:



Признаки данной архитектуры:

* + Более внешний слой зависит от более внутреннего слоя. Внутренние слои представляют собой **правила**, а внешние – **инструменты**.
  + Бизнес слой состоит из **двух** слоёв: Domain и Application.
  + Наличие **адаптеров** – обьектов, передаваемых в более внутренние слои с помощью механизма внедрения зависимостей, являющихся имплементациями интерфейсов из более внутренних слоёв.
  + Внешний слой отвечает за отображение информации.

Преимущества данной архитектуры:

* **Изолированность и независимость** слоя приложения и бизнес логики от способов доступа к данным, общения с другими сервисами, отображения информации и других механизмов.
* Простая **замена** одного фреймворка на другой: возможность передачи другой имплементации уже существующего интерфейса, и даже выбор имплементации прямо во время работы приложения.

Например, с использованием репозиториев одна ORM с лёгкостью меняется на другую.

* **Простота тестирования**: благодаря инверсии зависимостей, возможно создание моков, использующихся в модульном тестировании.

Недостатки данной архитектуры:

* Во многих проектах возможность замены может не быть полезной. К примеру, существуют проекты, которые никогда не меняют способ доступа к данным.
* Необходимо использовать фреймворки и библиотеки, строго реализуя интерфейсы приложения, что зачастую может быть проблематично из-за принципиальной разницы во фреймворках.

В таком случае, возможны изменения и в более внутренних слоях приложения.

* Возможность использования обьектов из более внутренних слоёв. В более конкретном случае, обьект, предназначенный для доступа к данным имеет возможность использования бизнес-обьекта из слоя Application.

Чистая Архитектура подходит проектам с необходимостью **частой замены** адаптеров (например, способов доступа к данным), а также со **сложной** бизнес логикой, вокруг которой необходимо сконцентрировать всё приложение.

* **Архитектура Вертикального Среда (Vertical Slice Architecture)**

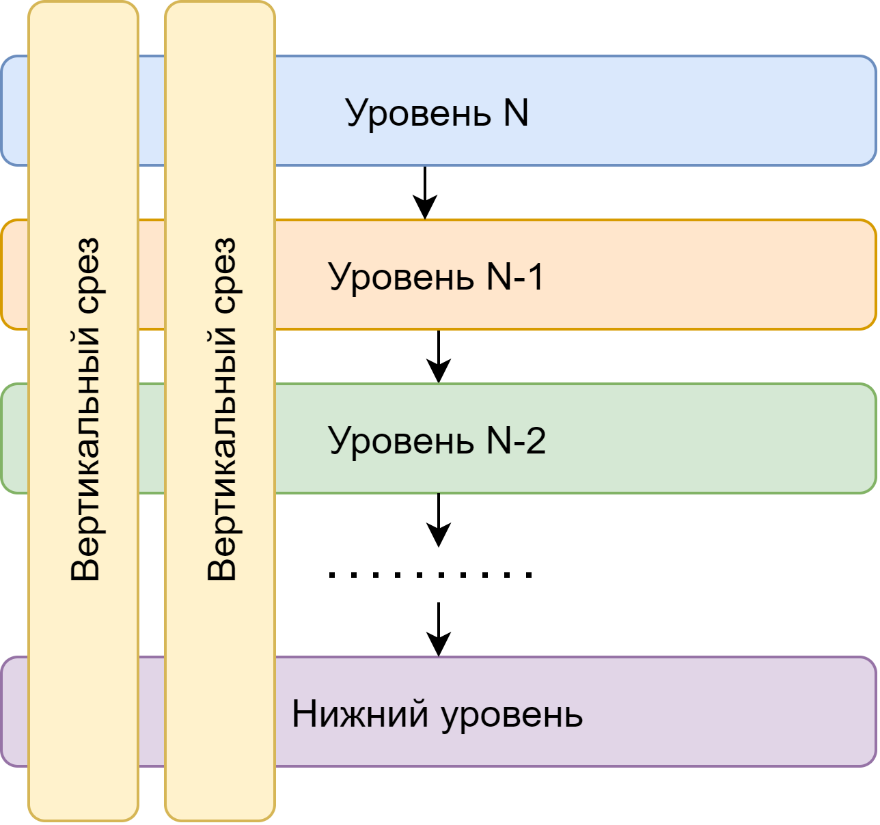
Организация приложения с помощью слоёв является **монолитным** решением.

Практика выявляет следующие недостатки ранее описанных архитектур:

* Привязка каждой функции приложения к следующему слою.
* Необходимость выявления **всех ссылок** на те компоненты, в которых проводятся изменения. Проверку работоспособности всех данных компонентов необходимо проводить даже при изменении одного из них.
* Во время внесения изменений, часто затрагиваются все слои приложения.
* Излишняя Абстрактизация тех компонентов, где в этом нет необходимости:
* Многие проекты **никогда не меняют** ORM, для инкапсуляции которой используется шаблон проектирования **Репозиторий**.
* Абстрактные компоненты используются **только в одной** из частей приложения, что делает бессмысленным её обобщение.

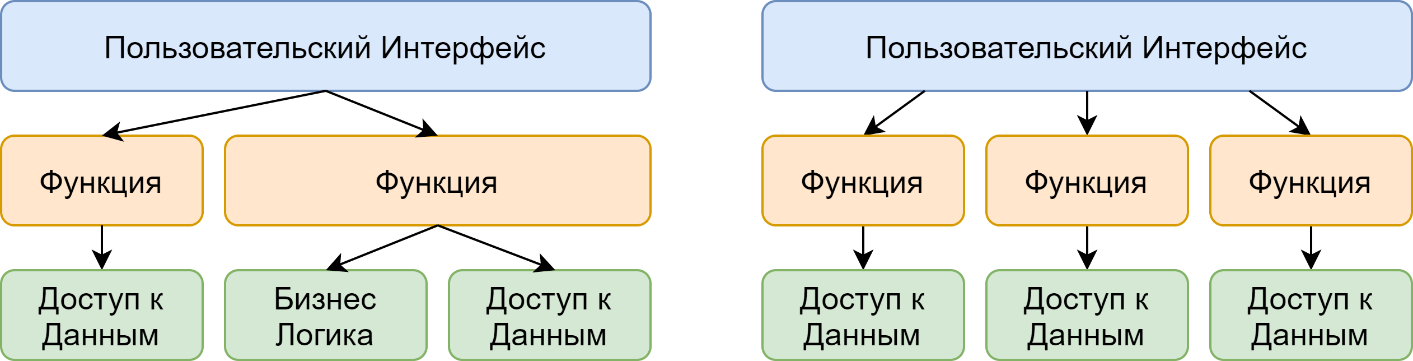
Архитектура Вертикального Среза устраняет данные проблемы следующим образом:

* Определение набора функций приложения, каждая из которых далее будет разрабатываться **индивидуально**, отдельно от остальны.
* Устранение абстрактизации между слоями в пределах каждой функции, и прямая привязка слоёв.



Каждый из вертикальных срезов является одной из функций приложения, которая состоит из всех необходимых ей компонентов.

При необходимости, вертикальный срез также можно делить на слои, комбинируя идеи данной архитектуры с принципами предыдущих:



Так как данная архитектура рассматривает приложение как набор отдельных, независимых функций, которые являются либо **запросом**, либо **командой**, принцип **CQRS (Command Query Responsibility Segregation)** будет лежать в основе разработки приложения.

Преимущества данной архитектуры:

* Устранение излишней абстрактизации, что упрощает разработку и поддержку каждой из функций приложения.
* **Невозможность** влияния изменений в одной функции на работоспособность других частей приложения.
* Потенциальная возможность выделения функций в **отдельные приложения**, что актуально в Микросервисной Архитектуре.
* Простота тестирования: каждый запрос тестируется индивидуально.

Недостатки данной архитектуры:

* Невозможность модульного тестирования из-за устранения абстрактизации между слоями. Для этого можно ввести абстрактизацию, но в другом виде, отличной от реализации в слоях.

Также, данная проблема частично решается интеграционных тестов.

* Усложнённая структура приложения.
* Каждый вертикальный срез может стать громоздким из-за совмещения в ней всех слоёв, в следствие чего необходимо продумывать архитектуру каждого среза в отдельности.

Архитектура Вертикального среза подходит для больших проектов, в которые часто вносят изменения, и с потенциальной необходимостью разделить их на несколько отдельных проектов.

# Архитектура проекта.

Проведя анализ функционала приложения, можно выделить два типа задач:

* Задачи, которые будут выполняться один раз, и для которых требуется выполнение HTTP метода (GET, POST и т.д.), **без наличия** постоянного подключения.

Это задачи, которые выполняются перед входом началом постоянного обмена сообщениями. Такими задачами являются:

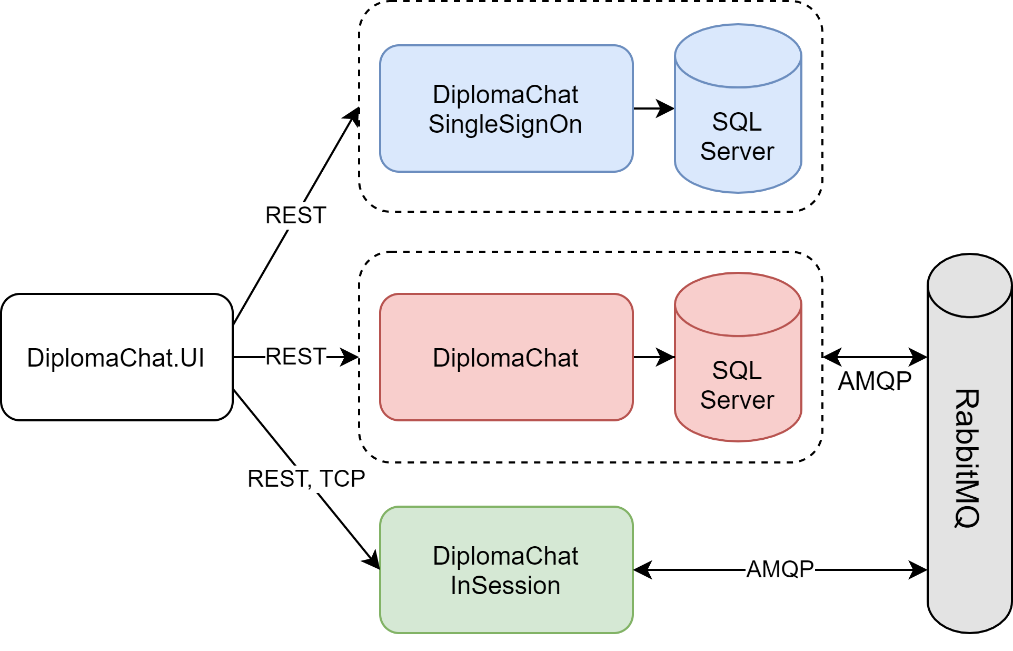
* + Создание чата.
  + Присоединение к чату.
  + Выход из чата.
* Задачи, для которых требуется постоянное подключение, для **беспрерывного обмена данными**, а также вызова событий.

Примерами таких задач являются:

* + Обмен сообщениями между пользователями.
  + Уведомление о присоединении пользователя.
  + Уведомление о выходе пользователя из чата.

Таким образом, приложение будет разделено на **два Микросервиса**: один - для операций, предшествующих началу обмена сообщениями, и другой – для действий внутри сессии. Они будут называться **DiplomaChat** и **DiplomaChat.InSession**.

При выполнении одной из операций микросервисом **DiplomaChat**, будет производится отправка сообщения в одной из очередей сообщений сервиса RabbitMQ. **DiplomaChat.InSession** является подписчиком на данные очереди. Таким образом, второй микросервис будет получать уведомление об одном из действий пользователя перед подключением к нему.



**DiplomaChat.UI** – Front-End часть системы, созданная на Angular.

**DiplomaChat.SingleSignOn** – Веб-Сервис, созданный исключительно для регистрации и авторизации пользователей, и хранящий всех пользователей в своей базе данных. Данный Веб-Сервис является примером **Сервиса Единого Входа (Single Sign On).**

**API Спецификация:**

* Ресурс Account
  + POST /account/register
  + POST /account/login
  + GET /account/details [Необходима Авторизация]

**DiplomaChat –** хранит функционал по созданию, присоединению, выходу из чата, а также получения информации об уже существующих чатах. Имеет собственную базу данных, где хранит свои данные. Регистрация пользователя существует и в этом микросервисе, но только с целью установки псевдонима.

**API Спецификация:**

* Rooms
  + PUT /rooms/create
  + GET / rooms/join/{roomId}
  + GET /account/leave
* Users
  + POST /user/register
  + GET /user/profile
  + GET /user/{userId}/profile

**DiplomaChat.InSession –** хранит функционал по обмену сообщениями между пользователями, для которого будет использована библиотека SignalR.

**API Спецификация:**

* **Chats**
  + GET chats/{roomId}/members

**SignalR события:**

* Вызываемые пользователем:
  + Connect
  + Disconnect
  + SendMessage
* Получаемые пользователем:
  + UserConected
  + UserDisconnected
  + ReceiveMessage

Преимущества, полученные благодаря данной архитектуре

* **Возможность расширения системы**: DiplomaChat.SingleSignOn напрямую не относится к архитектуре проекта, так как не взаимодействует с остальными Сервисами, и не сожержит функционала, относящегося к чату.

При возникновении нового проекта, требующего наличие механизма аутентификации и авторизации пользователей, Сервис Единого Входа **уже будет существовать**, что значительно **снизит затраты** на разработку нового проекта.

Кроме того, база данных пользователей **будет единой**, что позволит им авторизироваться в другие проекты без регистрации, а также использовать стандарт OpenID и протокол OAuth.

* **Отказоустойчивость**: При возникновении проблем в микросервисе DiplomaChat, перестанет работать Создание, присоединение и выход из чата.

Обмен сообщенями между пользователями, которые уже присоединились к чату, будет возможен и далее, ведь DiplomaChat.InSession будет продолжать работать.

Данное преимущества работает и наоборот – при отказе DiplomaChat.Insession, обмен сообщениями перестанет работать, в отличие от создания, присоединения и выходе из чата.

Недостатки:

* Необходимость дублирования некоторой части логики: создание, присоединение, а также выход из чата должен происходить в обоих микросервисах. Разница в том, что DiplomaChat хранит сессии в базе данных, а DiplomaChat.InSession – в оперативной памяти.
* Необходимость сохранять целостность данных между микросервисами, что может стать проблемой при отказе DiplomaChat.InSession – при повторном запуске необходимо получить список всех уже существующих чатов.

Порядок действий, выполняемого клиентом (DiplomaChat.UI):

* Регистрация пользователя (если он не существует).
* Авторизация, получение токена авторизации.
* Получение списка чатов.
* Создание нового чата.
* Присоединение к уже существующему чату (включая только что созданный).
* Установка постоянного TCP подключения и обмен сообщениями.
* Выход из чата.

В качестве способа реализации Архитектуры Вертикального Среза, была выбрана библиотека **MediatR**, реализующая шаблоны проектирования **CQRS** и **Посредник**.

Она содержит:

* Запрос (Request) – класс, реализующий интерфейс IRequest<T>.
* Обработчик (Handler) – класс, реализующий интерфейс IRequestHandler<TRequest,TResponse>, который получает TRequest (реализующий IRequest<T>), и возвращает TResponse. Он является главной частью бизнес-слоя приложения.
* Ответ (Response) – класс, возвращаемый Обработчик.
* Посредник (Mediator) – обьект, который, получив на вход запрос, находит соответствующий ему обработчик, и далее вызывает его.

Общий вид Vertical Slice архитектуры .NET Web API приложений с несколькими контроллерами, использующей библиотеку MediatR:

Diagram

Description automatically generated Diagram

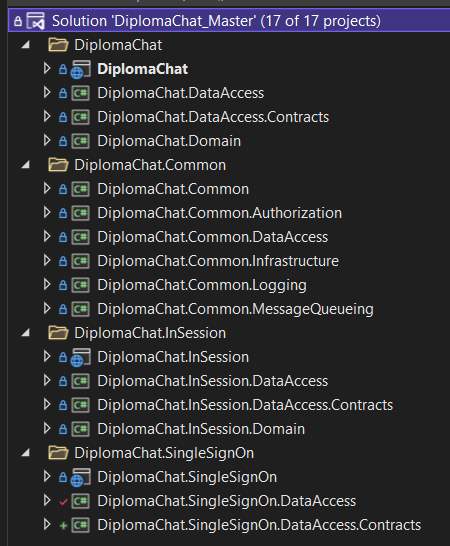
Description automatically generated

# Реализация проекта.

**Создание проектов**

Visual Studio создаёт проекты со следующей структурой: одно Решение (Solution) может содержать множество проектов (В данном случае .csproj файлы).

Структура всего Solution:



Структура названий стандартных Проектов в Решении:

* Главный проект, в котором и находится точка входа в приложение, носит название самого приложения. В данном случае это DiplomaChat, DiplomaChat.Common, DiplomaChat.InSession и DiplomaChat.SingleSignOn.

Название главного проекта присутсвует во всех остальных проектах решения.

* **DataAccess** – содержит имплементации классов, предназначенных для доступа к данным.
* **DataAccess.Contracts** – содержит интерфейсы, предназначенные для доступа к данным. Данные интерфейсы являются зависимостью для главного проекта, DataAccess содержит их имплементации.
* **Infrastructure** – Инструменты, с помощью которых работает приложение.

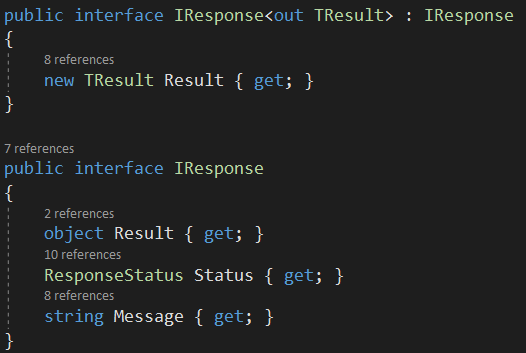
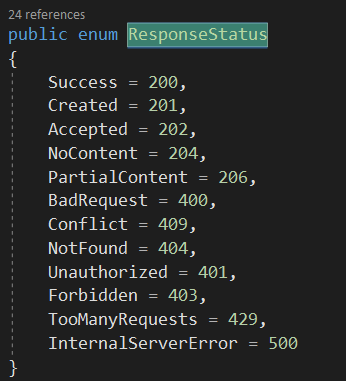
Существуют части инфраструктуры, выделенные в отдельный проект, но имеющие .Infrastructure в названии: Infrastructure.MessageQueueing, Infrastructure.Logging.

* **Domain** – проект, содержащий классы для работы с бизнес логикой, без привязки к логики конкретного приложения.
* **Application** – проект, в который можно выделить всю бизнес логику данного приложения. Он не обязателен, может быть полезен для структуризации проекта или контроллирования ссылок Бизнес слоя приложения.

Также, используется **Swagger UI** – инструмент для визуализации API Документации, имеющий возможность вызова методов Веб-Сервиса. Это упрощает разработку т.к. для вызова HTTP методов не нужны сторонние приложения.

**DiplomaChat.Common** – библиотека классов, содержащая общие компоненты всех Сервисов данного преокта. Её создание возможно из-за того, что все Сервисы являются Web API приложениями.

* **DiplomaChat.Common.Authorization** – содержит инструменты для JWT авторизации: создание токена, чтение информации из него, схемы безопасности.
* **DiplomaChat.Common.DataAccess** – Содержит классы и интерфейсы для абстрактизации ORM Entity Framework, базовые классы сущостей и их конфигурации.
* **DiplomaChat.Common.Infrastructure** – инструменты, с помощью которых работает приложение. Например, интерфейс IResponse, с помощью которого можно вернуть результат запроса, задать код состояния, а также задать сообщение в ответе.

* **DiplomaChat.Common.Infrastructure.Logging** – Журналирование с использованием библиотеки **Serilog**.
* **DiplomaChat.Common.Infrastructure.MessageQueueing** – классы для отправки сообщения в очередь, а также их получение. Имеется имплементация для RabbitMQ.

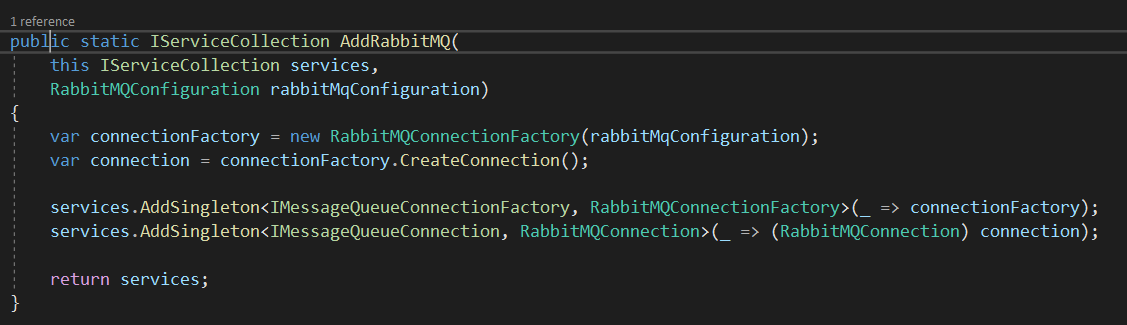
Кроме того, он содержит атрибуты и методы расширения, позволяющие просто создать класс-получатель сообщений из нескольких очередей. Это стало возможноым благодаря рефлексии.

**Пример** из DiplomaChat.InSession: данные методы будут автоматически вызваны при получении сообщения из соответствующей очереди, с предварительно сериализированным JSON из сообщения.

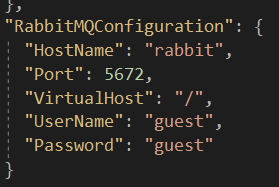
Text

Description automatically generated

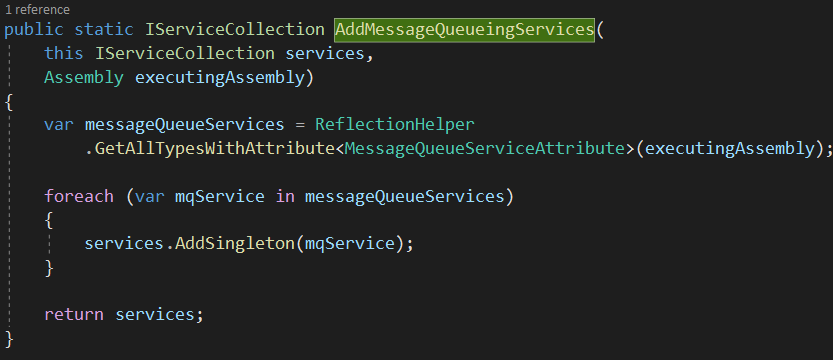
Метод расширения для добавления RabbitMQ:



Для его работы необходимо добавить соответствующую конфигурацию в appsettings.json:

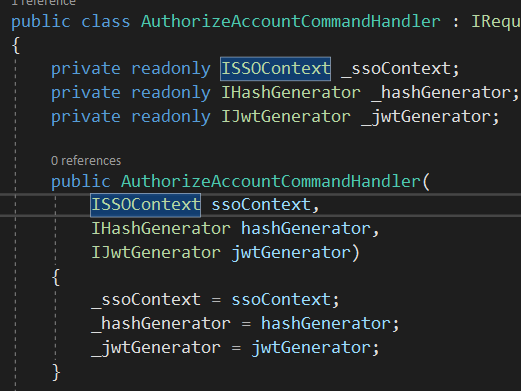
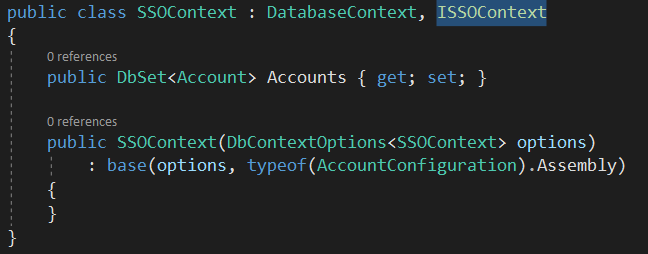


Добавление MQ Сервисов, имеющих атрибут MessageQueueService:



Сервис **DiplomaChat.SingleSignOn** состоит из 3 проектов:

* **DiplomaChat.SingleSignOn** – точка входа в приложение.
* **DiplomaChat.SingleSignOn.DataAccess** – содержит специфические для данного Сервиса классы доступа к данным: сущность Account, её конфигурацию, а также контекст базы данных.
* **DiplomaChat.SingleSignOn.DataAccess.Contracts** – Содержит интерфейс для доступа к базе данных **ISSOContext**, реализуемый классом **SSOContext** из DataAccess проекта, и используемый в Бизнес слое проекта (в одном из обработчиков):



**Архитектура Сервиса DiplomaChat.SingleSignOn**

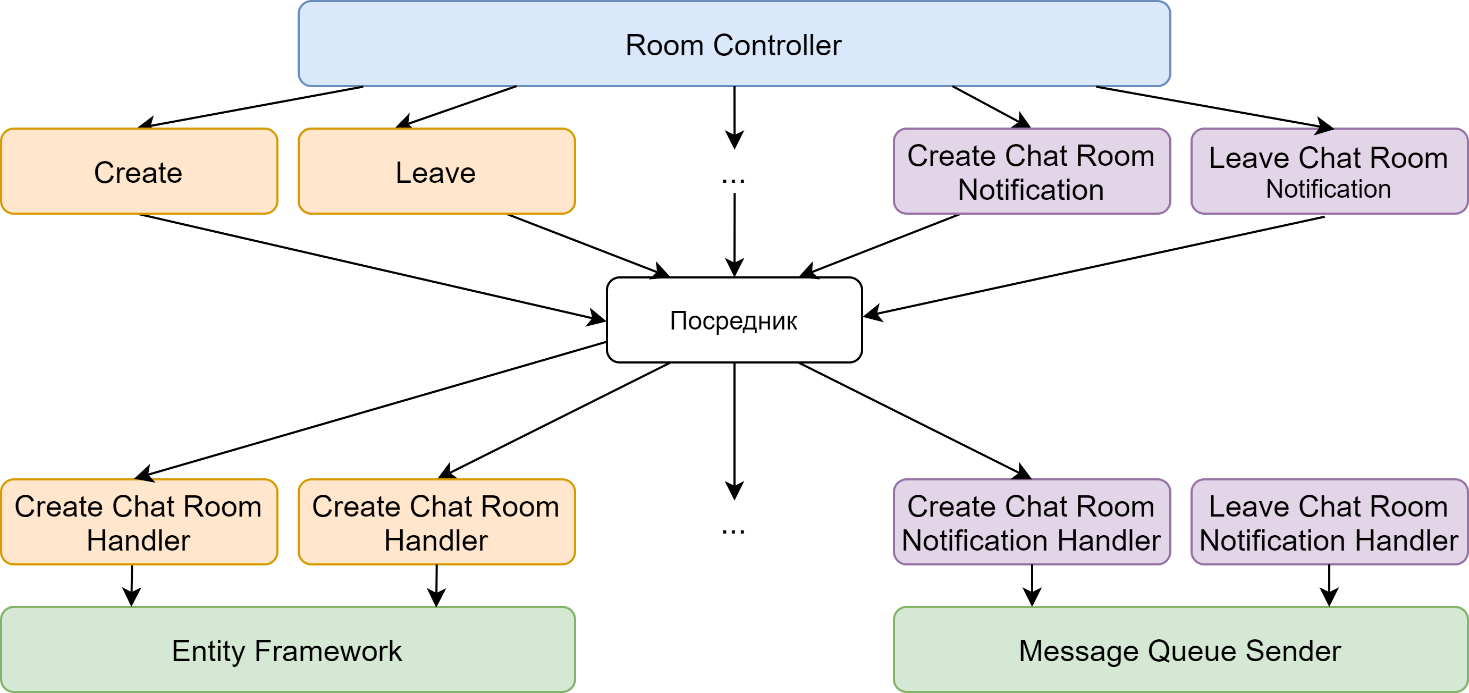
Diagram

Description automatically generated

Микросервис **DiplomaChat** также состоит из стандартных проектов**:**

* **DiplomaChat**
* **DiplomaChat.DataAccess**
* **DiplomaChat.DataAccess.Contracts**
* **DiplomaChat.Domain**

**Архитектура Микросервиса DiplomaChat**

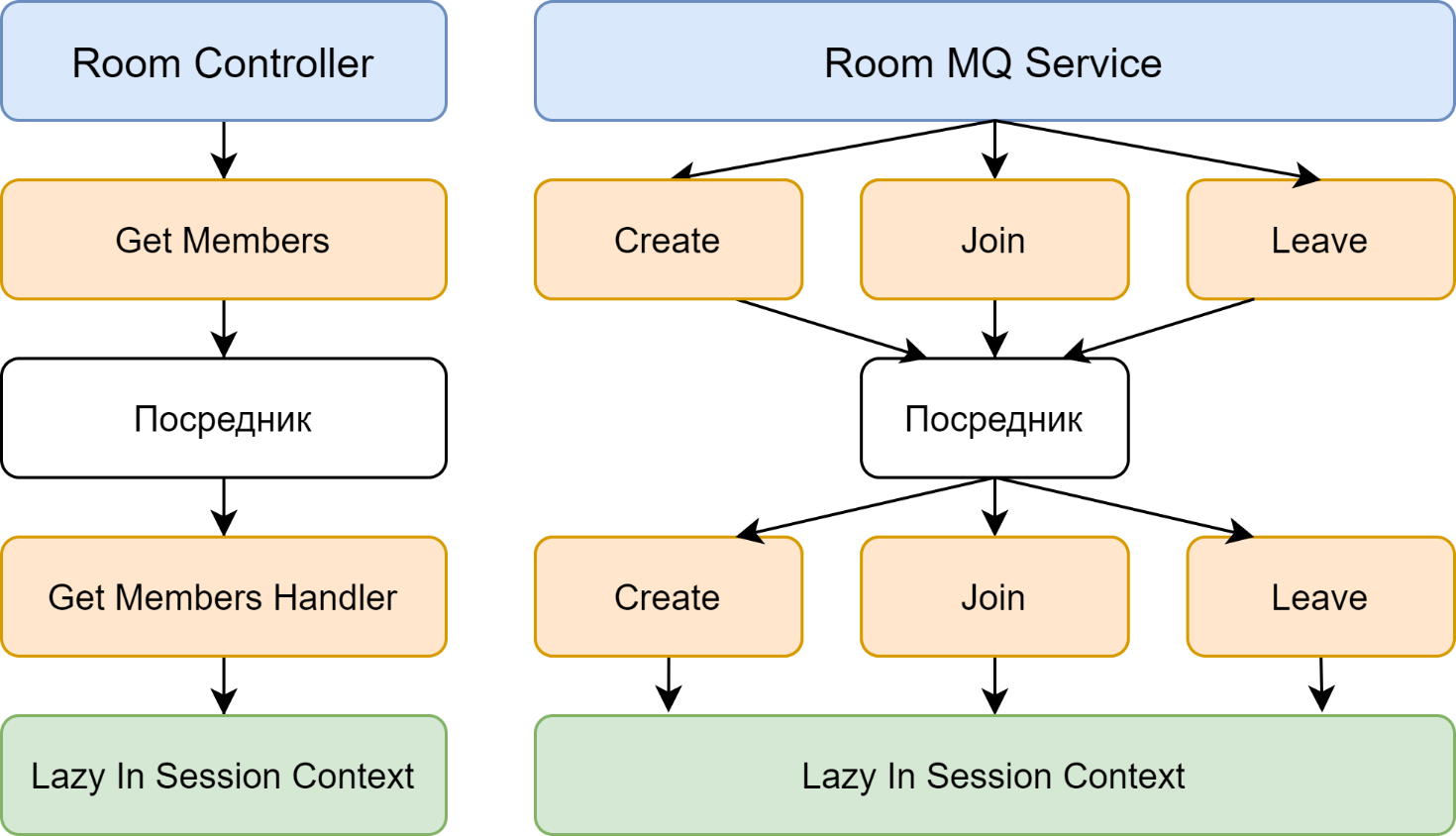


|  |  |
| --- | --- |
|  | На скриншоте отображена структура папок проекта, которая также указывает на использование Vertical Slice архитектуры. В папке **Features** находятся классы всех HTTP методов проекта, сгруппированные по ресурсам (Rooms и Users). Каждая папка содержит 3 класса, являющихся **запросом**, **ответом** или **обработчиком** соответствующего метода. В ней же находится контроллер.  Папка **Notifications** содержит подобные 3 класса, но там находятся “уведомления”, отправляющие сообщения на MQ Сервис.  Как было сказано ранее, при необходимости вся папка Features может быть вынесена в отдельный проект.  Структура папок остальных Сервисов аналогична. |

Микросервис **DiplomaChat.InSession** содержит аналогичные проекты:

* **DiplomaChat.InSession**
* **DiplomaChat.InSession.DataAccess**
* **DiplomaChat.InSession.DataAccess.Contracts**
* **DiplomaChat.InSession.Domain**

**Архитектура Микросервиса DiplomaChat.InSession**

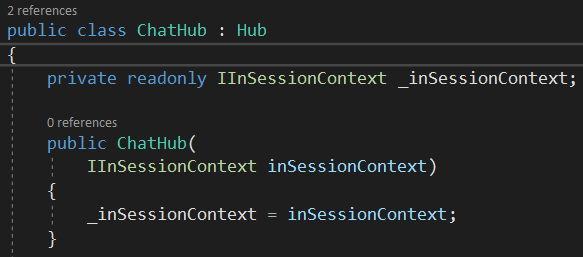


**Обмен сообщениями в чате**

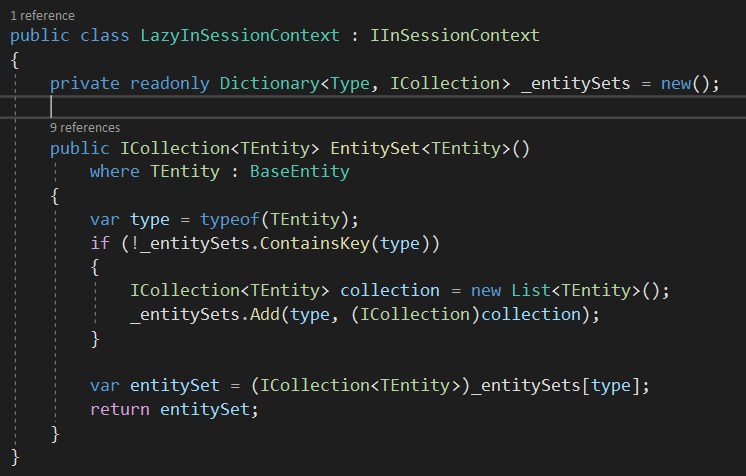
Библиотека **SignalR** предоставляет инструменты для обмена сообщениями в реальном времени по установленному TCP подключению. Класс, содержащий события SignalR имеет название **Hub**.

В собственном приложении используются наследники данного класса. В моём случае это **ChatHub**. Необходимо учитывать, что при каждом получении сообщения или вызова события, **создаётся новый** обьект класса-наследника.

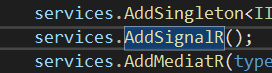
С архитектурной точки зрения, Hub находится в Бизнес слое приложения. Он использует обьект доступа к данным через интерфейс **IInSessionContext**.

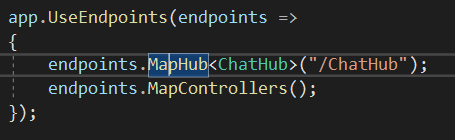


Класс **InMemoryInSessionContext**, хранящий данные в оперативной памяти, используется качестве имплементации IInSessionContext. Для этого он использует **Dictionary** (Словарь - реализует интерфейс Ассоциативного списка), где ключ – тип сущности, хранящейся в коллекции, а значение – сама коллекция.



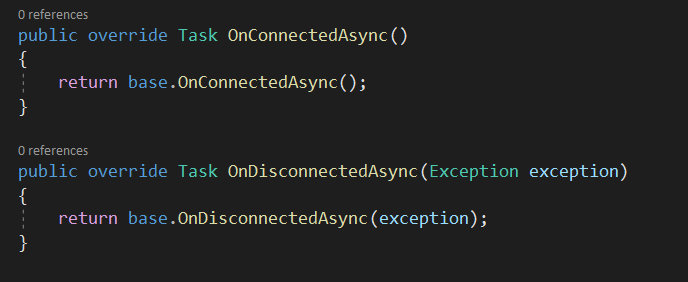
Созданный Hub необходимо, добавить в **DI** контейнер, передав путь, с помощью которого клиент сможет подключиться к нему:





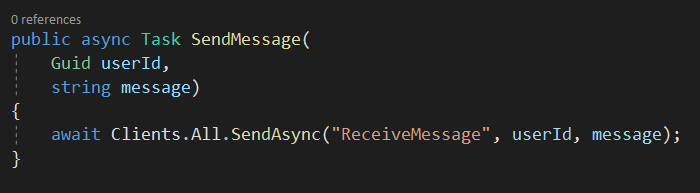
**SignalR** содержит два стандартных события:

* **OnConnected** – вызывается после установления подключения с клиентом
* **OnDisconnected** – вызывается при разрыве соединения с клиентом

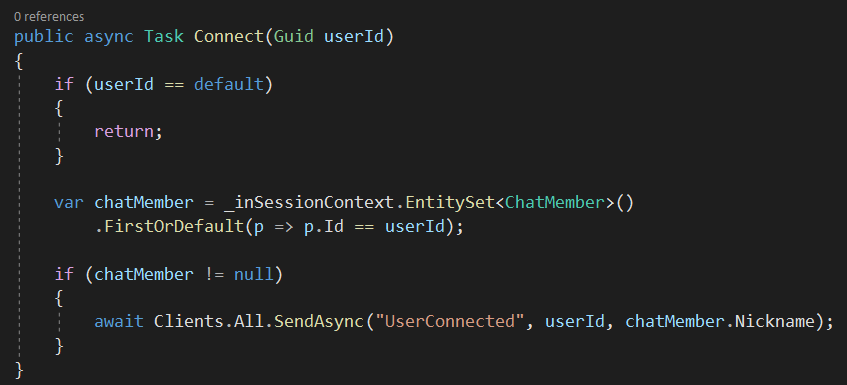


В данном случае, они не используются из-за невозможности передачи параметров. Данные методы должны содержать логику **инициализации**. Бизнес логика должна отсутствовать.

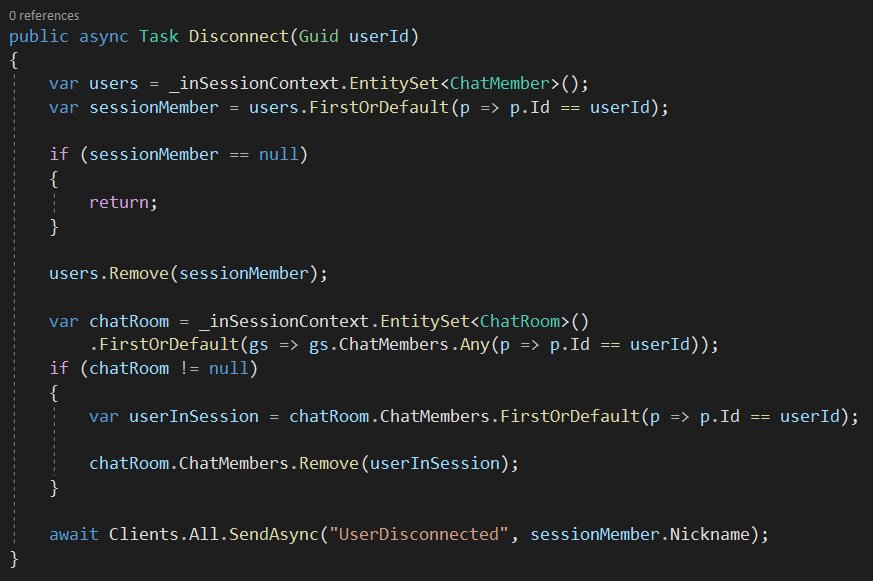
В качестве собственных событий выступают методы данного класса. Например, событие отправки сообщения:



Событие подключения пользователя к чату:



Событие отключения пользователя от чата:

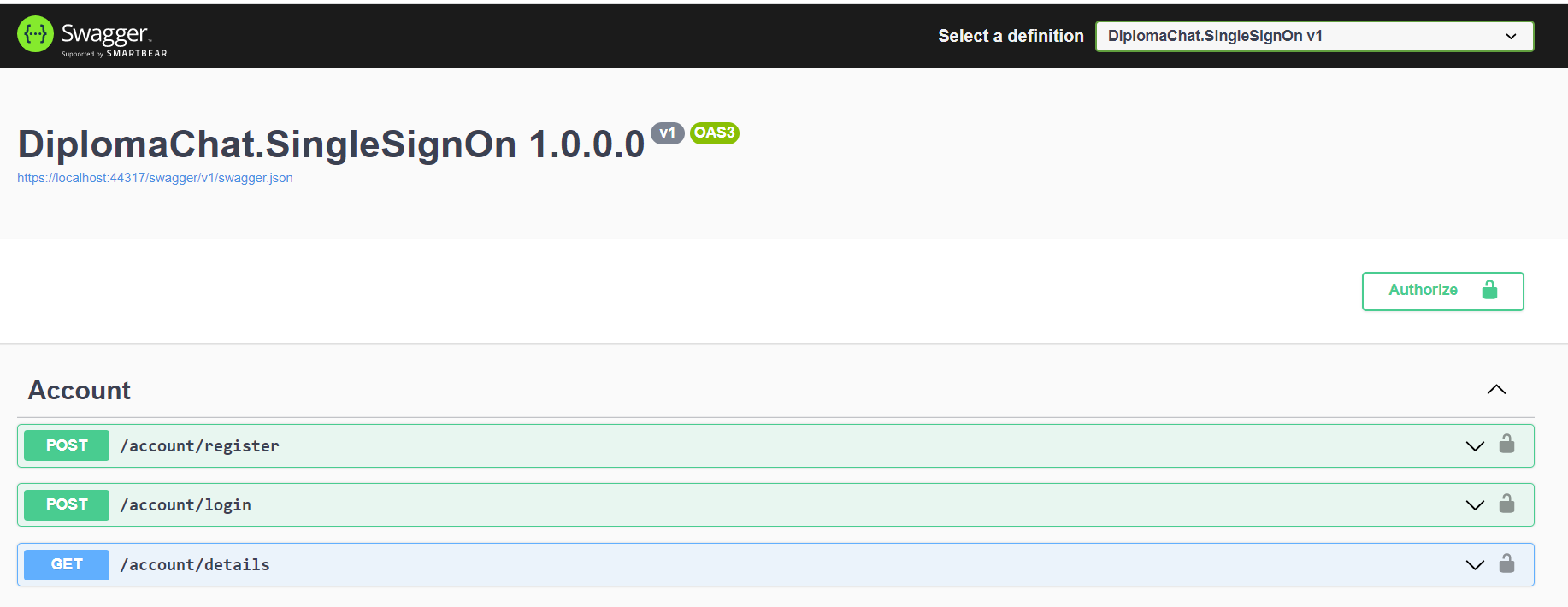


Как видно из скриншотов, метод **SendAsync** вызывает событие у клиентов. Здесь также присутствует возможность передачи параметров.

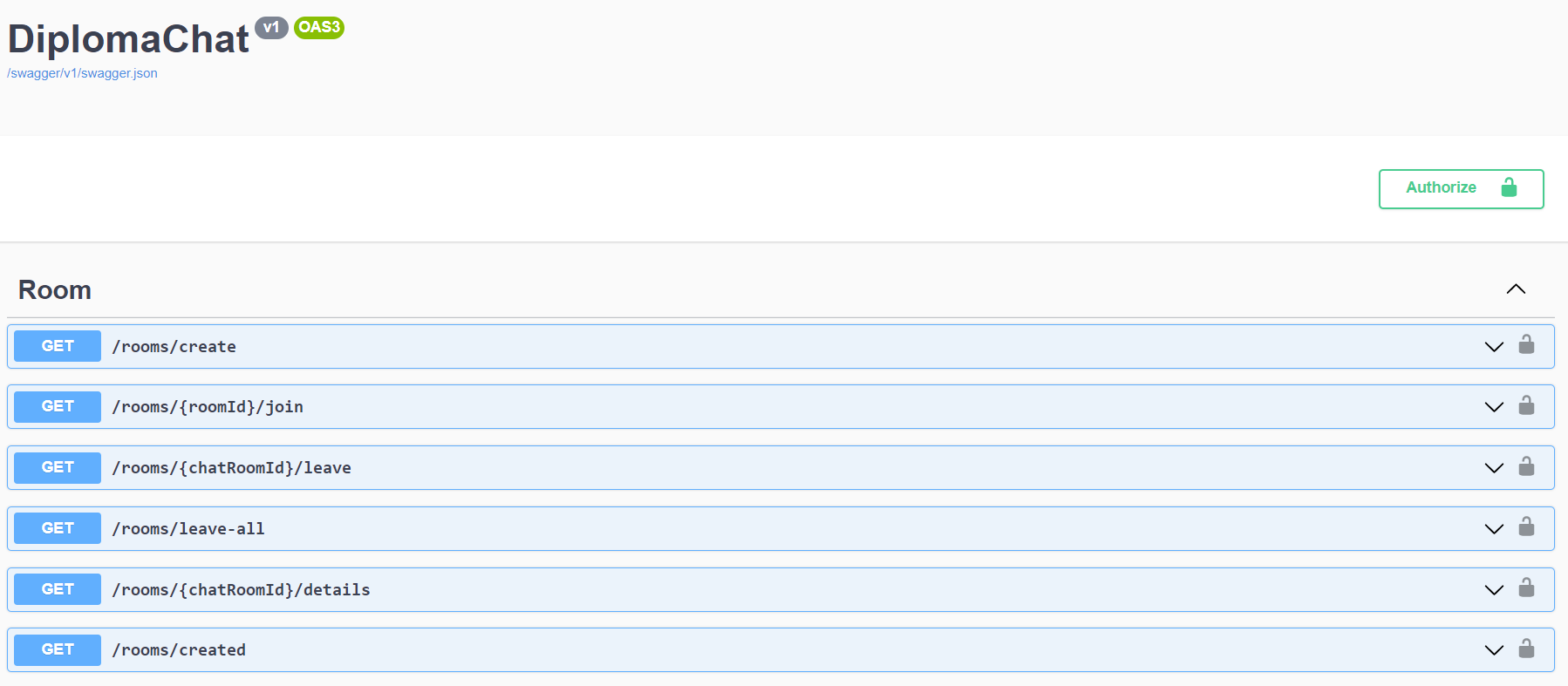
Для подключения, создаётся класс HubConnectionBuilder, в который необходимо указать Url, а также способ подключения. Далее, необходимо указать методы, которые будут вызваны при вызове события, и непосредственно начать установку подключения.



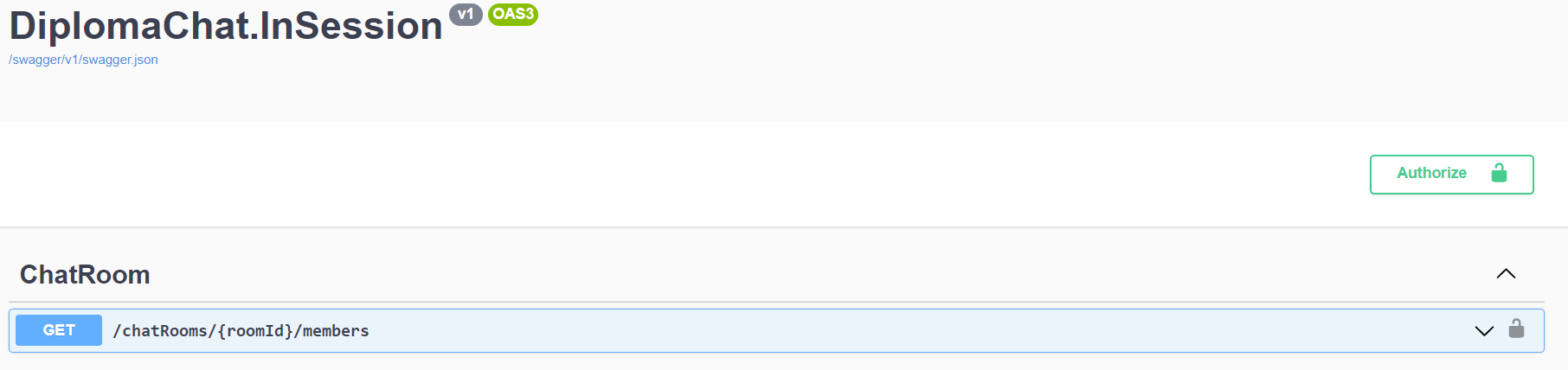
**Swagger UI для SingleSignOn**



**Swagger UI для DiplomaChat**

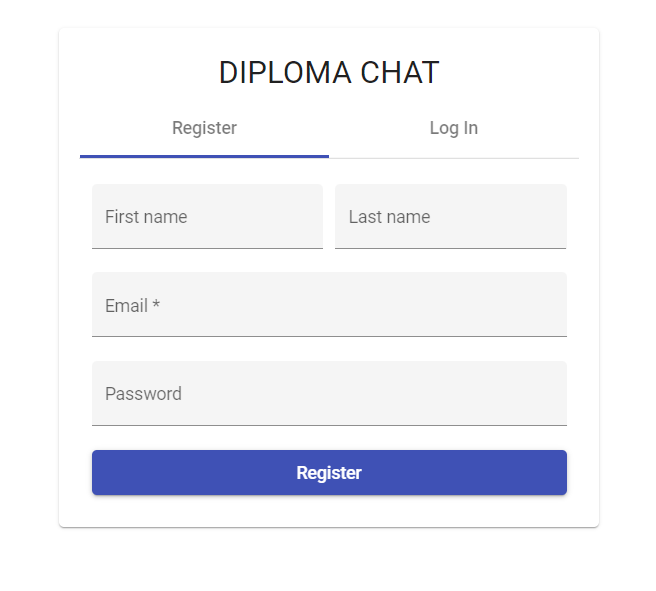


**Swagger UI для DiplomaChat.InSession**

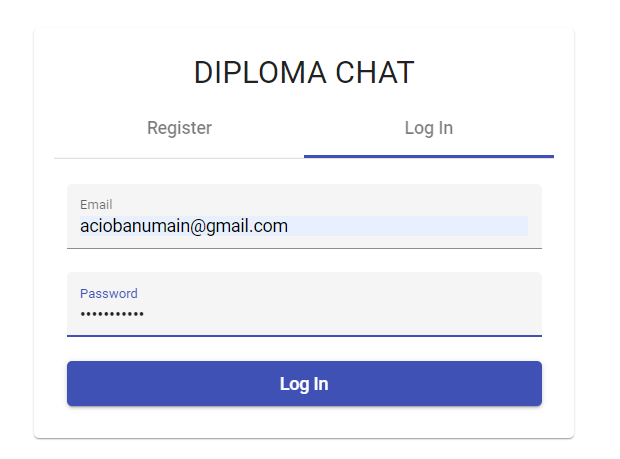


**Описание интерфейса приложения**

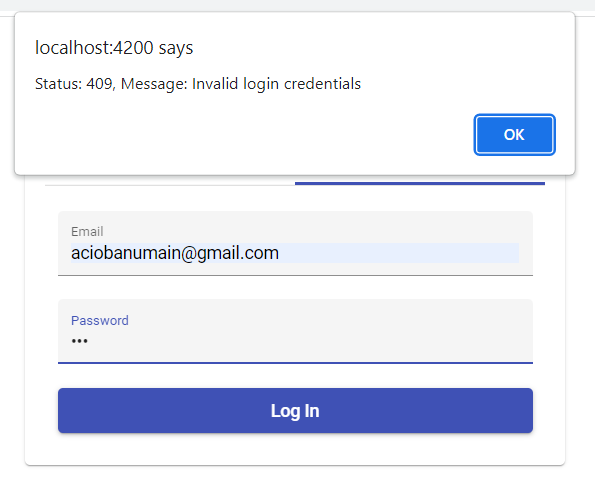
Страница регистрации



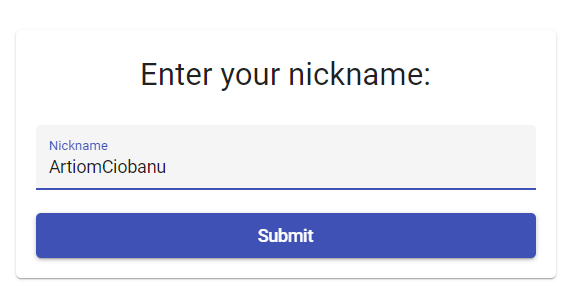
Страница Авторизации



Сообщение о неправильном логине или пароле



Страница ввода псевдонима, если он не был введён ранее

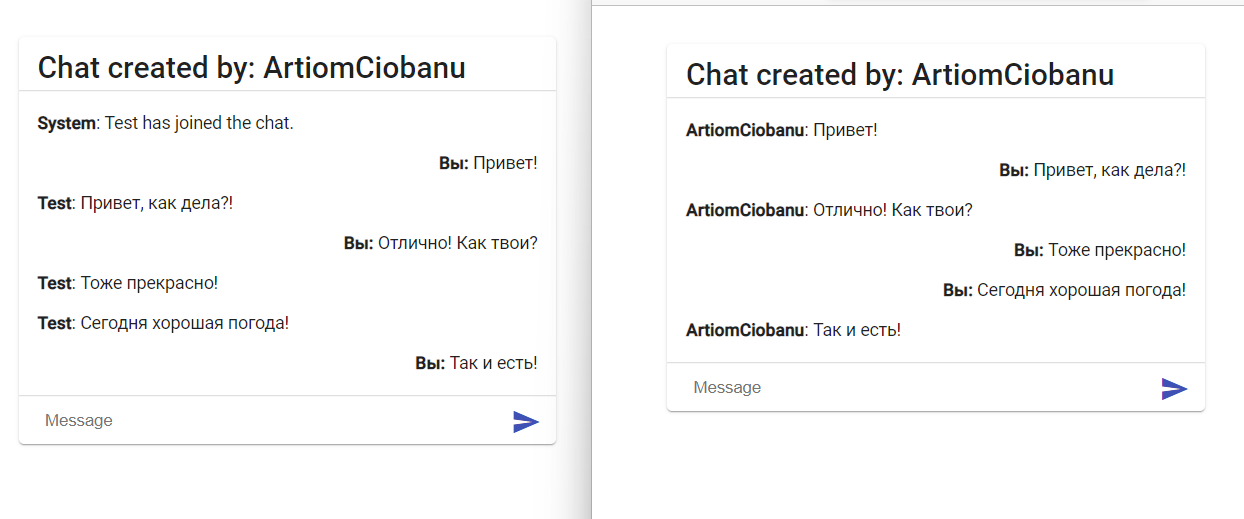


Страница списка чатов, на которой также присутствуют псевдоним пользователя, кнопка выхода, а также кнопка создания нового чата

Graphical user interface, application

Description automatically generated

Страница чата, в котором присутствует два пользователя



# Заключение

**Микросервисная Архитектура** – самый актуальный способ достижения таких преимуществ как масштабирование, упрощённое тестирование и более простое внесение изменений в проект.

В свою очередь, приложения с микросервисами являются более сложными и требуют компетентных специалистов, таких как **Архитекторы**.

Таким образом, Микросервисная Архитектура подходит для **крупных проектов**. Монолитное приложение подойдёт для **небольших систем**, так как в них преимущества от микросервисов не будут реализованы. При этом, они добавят излишней сложности в разработку и поддержку проекта.

Кроме того, существует возможность разработки монолита на **ранней стадии** с последующим переходом на микросервисы, когда это станет необходимо.

Использование **Сервис-Ориентированной Архитектуры** для определённых проектов может быть более подходящим. В таком случае, будут созданы несколько независимых монолитных приложений.

# Литература и документация:

Ссылки

<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>

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

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Книги

**Чистая Архитектура** – Роберт К.Мартин

**Microservices patterns** – Крис Ричардсон

**Microservices: From Design to Deployment** – Крис Ричардсон и Флойд Смит

# Приложения

..