Котлов В. Н., Фирсова С. А. Исследование микросервисного подхода к разработке архитектуры программных систем // Международный журнал информационных технологий и энергоэффективности. – 2023. –
Т. 8 № 8(34) с. 37–47

Международный журнал информационных технологий и энергоэффективности

Сайт журнала:
http://www.openaccessscience.ru/index.php/ijcse/

УДК 004.9

ИССЛЕДОВАНИЕ МИКРОСЕРВИСНОГО ПОДХОДА К РАЗРАБОТКЕ АРХИТЕКТУРЫ ПРОГРАММНЫХ СИСТЕМ

Котлов В. Н., [1] Фирсова С. А.
*ФГБОУ ВО "НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ МОРДОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.П. ОГАРЁВА", Саранск Россия (430005, Республика Мордовия, город Саранск, Большевистская ул., д.68)***,** *e-mail: [1]karpushkinasa@yandex.ru*

В статье описывается микросервисный подход к разработке программных систем. Указываются преимущества и ограничения данного подхода в сравнении с монолитной архитектурой. Приводится пример решения, основанного на применении микросервисной архитектуры, в котором демонстрируется настройка взаимодействия между двумя независимыми сервисами на основе протокола gRPC.

Ключевые слова: Разработка программных систем, монолитная архитектура, микросервисная архитектура, протокол удаленного вызова процедур, язык программирования C#.

RESEARCH OF MICROSERVICE APPROACH TO THE DEVELOPMENT OF SOFTWARE SYSTEMS ARCHITECTURE

Kotlov V. N., [1]Firsova S. A.,
"NATIONAL RESEARCH MORDOVIA STATE UNIVERSITY. N.P. OGAREVA", Saransk Russia (430005, Republic of Mordovia, Saransk city, Bolshevistskaya street, 68), e-mail: [1]karpushkinasa@yandex.ru

The article describes a microservice approach to the development of software systems. The advantages and limitations of this approach in comparison with monolithic architecture are indicated. An example of a solution based on the use of a microservice architecture is given, which demonstrates the configuration of interaction between two independent services based on the gRPC protocol.

Keywords: Software systems development, monolithic architecture, microservice architecture, remote procedure call protocol, C# programming language.

В современном информационном обществе, где технологические требования постоянно эволюционируют, разработка программных систем стала неотъемлемой составляющей бизнеса и технического прогресса. Однако с ростом сложности приложений и необходимостью удовлетворять различным потребностям пользователей, традиционные архитектуры программных систем сталкиваются с вызовами, которые оказывают ограничивающее влияние на их эффективность.

Монолитная архитектура является стандартной моделью разработки программного обеспечения, в которой разрабатываемая программная система представляет собой единую, крупную и автономную систему, функционирующую независимо от других приложений [1].

При данном подходе все бизнес-задачи объединяются в одной большой вычислительной сети с единой базой кода. Например, на рисунке 1 показано приложение, построенное на основе монолитной архитектуры, включающее пользовательский интерфейс (User Interface), уровень доступа к данным (Data Access Layer), бизнес-логику приложения (Business Logic), которые функционируют в одной вычислительной сети. Данное приложение взаимодействует с базой данных, расположенной, возможно, в другой вычислительной сети.

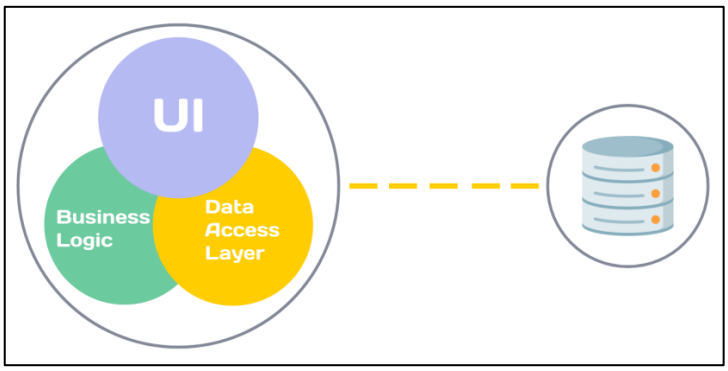

Рисунок 1 – Монолитная архитектура программной системы
Источник: Авторский материал

Монолитную архитектуру удобно применять на ранних этапах работы над программными проектами, чтобы облегчить развертывание программной системы и избежать излишних трудозатрат при управлении кодом [2]. Она позволяет сразу представить весь функционал приложения в одной монолитной сущности.

Преимущества монолитной архитектуры приведены в Таблице 1:

Таблица 1 – Преимущества монолитной архитектуры

Название	**Описание**
Простое развертывание	Использование единого исполняемого файла или каталога упрощает процесс развертывания приложения
Упрощенная разработка	Приложение проще разрабатывать, когда оно создано с использованием единой базы кода
Высокая производительность	Централизованная база кода и репозиторий позволяют одному интерфейсу API выполнять функции, которые в микросервисах могут выполнять многочисленные API
Упрощенное тестирование	Монолитное приложение представляет собой единую централизованную систему, что упрощает проведение сквозного тестирования по сравнению с распределенными приложениями
Удобная отладка	Весь код находится в одном месте, что упрощает поиск проблем и проведение запросов

Источник: Авторский материал

Тем не менее, монолитная архитектура также имеет свои недостатки и ограничения,

которые представлены в Таблице 2:

Таблица 2 – Недостатки монолитной архитектуры

Название	**Описание**
Снижение скорости разработки	Большое монолитное приложение усложняет и замедляет процесс разработки, особенно при работе в команде
Тяжелая масштабируемость	Невозможно масштабировать отдельные компоненты монолитной архитектуры, что может стать проблемой при обработке больших объемов данных и нагрузок на систему
Слабая отказоустойчивость	Ошибка в одном модуле может повлиять на доступность всего приложения, что делает монолит менее отказоустойчивым
Сложность внедрения новых технологий	Любые изменения в инфраструктуре или языке разработки сказываются на приложении целиком, что может привести к дополнительным затратам при внесении изменений в код
Ограниченная гибкость	Возможности монолитных приложений ограничены используемыми технологиями, что может ограничить применение инновационных технологий
Необходимость полного развертывания	Внесение даже небольших изменений потребует повторного развертывания всего монолитного приложения, что может замедлить процесс его обновления

Источник: Авторский материал

В поисках более эффективных решений и преодоления ограничений монолитной архитектуры, разработчики все чаще обращают взор к микросервисной архитектуре. Микросервисная архитектура представляет собой современный метод организации архитектуры программного обеспечения, который строится на основе независимо развертываемых служб, называемых микросервисами [3]. Каждый микросервис выполняет конкретную функцию и может работать автономно, имея свою базу данных и собственную бизнес-логику (Рисунок 2). Это означает, что каждая служба в микросервисной архитектуре может разрабатываться, обновляться, тестироваться и масштабироваться независимо от других служб. Взаимодействие между микросервисами происходит посредством четко определенных интерфейсов, часто использующих легковесные протоколы, такие как REST API или сообщений на основе очередей.

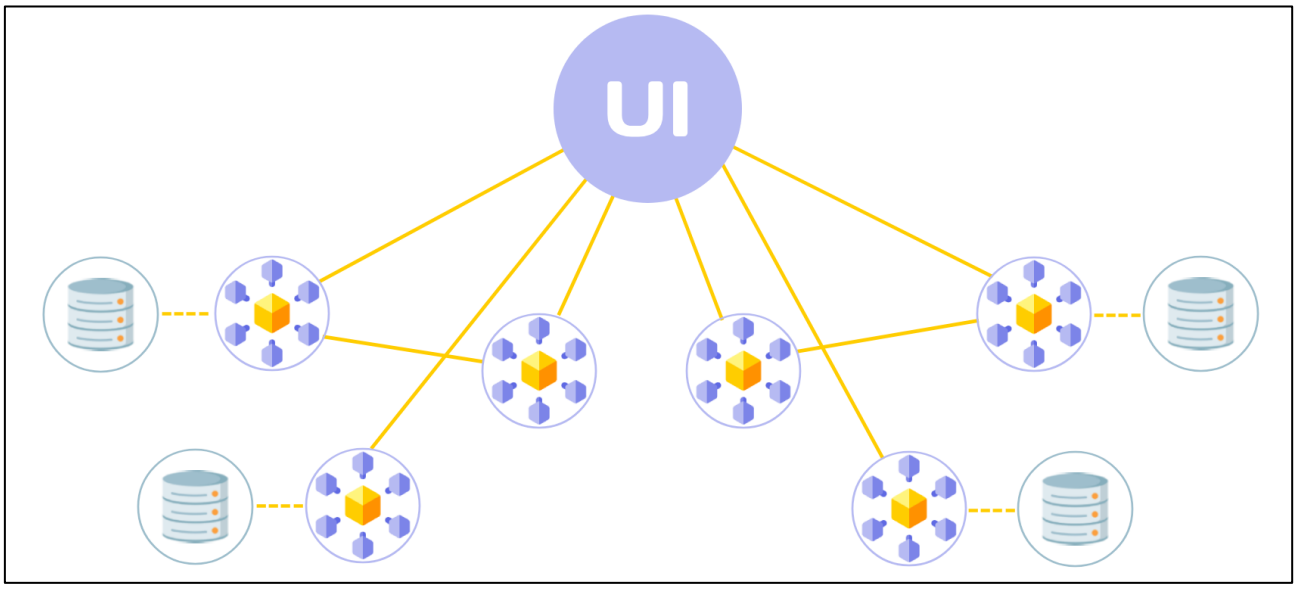

Рисунок 2 – Микросервисная архитектура программной системы

Источник: Авторский материал

Преимущества микросервисной архитектуры включают (Таблица 3):

Таблица 3 – Преимущества микросервисной архитектуры

Название	**Описание**
Гибкость разработки	Каждая служба может быть разработана независимо от других служб, что упрощает и ускоряет процесс разработки
Легкость масштабирования	При увеличении нагрузки на конкретную службу, ее можно масштабировать отдельно от других служб, что позволяет оптимизировать всевозможные ресурсы, затрачиваемые на масштабирование
Улучшенная отказоустойчивость	При отказе одной из служб, другие службы продолжат функционировать нормально, что повышает надежность всей системы
Легкость внедрения новых технологий	Каждая служба может использовать собственные технологии, что упрощает внедрение новых технологий в систему
Улучшенная организация команды	Команды разработчиков могут быть специализированы на определенных службах, что способствует повышению производительности и способности быстро реагировать на изменения

Источник: Авторский материал

При использовании микросервисной архитектуры следует учитывать следующие особенности её применения (Таблица 4):

Таблица 4 – Особенности применения микросервисной архитектуры

Название	Описание
Управление сложностью	Разделение сложных систем на множество микросервисов (служб) требует более тщательного управления и координации между ними
Сетевое взаимодействие	Взаимодействие между службами может вызывать задержки и проблемы с сетью передачи данных
Мониторинг и отладка	Необходимо обеспечить эффективный мониторинг и отладку каждой службы для обнаружения возможных проблем
Согласованность данных	При наличии нескольких баз данных в разных службах, обеспечение согласованности данных становится более сложной задачей
Управление версиями	Необходимо тщательно управлять версиями каждой из служб

Источник: Авторский материал

В целом, микросервисная архитектура является мощным инструментом для построения гибких и масштабируемых программных систем. Однако ее успешная реализация требует тщательного планирования, анализа и учета всех ограничений и особенностей разрабатываемого программного продукта. Выбор между монолитной и микросервисной архитектурой должен основываться на конкретных требованиях проекта и его потенциальных перспективах. Важно принимать во внимание, что микросервисы лучше подходят для крупных и сложных систем, где разделение функционала на независимые компоненты обеспечивает более эффективную разработку и масштабируемость. Небольшие проекты или проекты, в которых достижение высокой производительности не является первостепенной задачей, могут извлекать больше пользы из монолитной архитектуры.

Рассмотрим методы коммуникации между службами микросервисной архитектуры.

RESTful API (Representational State Transfer) **–** один из наиболее популярных способов взаимодействия между микросервисами. RESTful API основан на стандартных HTTP методах, таких как **GET** (получение данных), **POST** (создание новых данных), **PUT** (обновление данных) и **DELETE** (удаление данных). Каждый микросервис предоставляет свои конечные точки (endpoints), через которые другие службы могут обращаться для получения или обновления информации с помощью Json-объектов (рисунок 3):

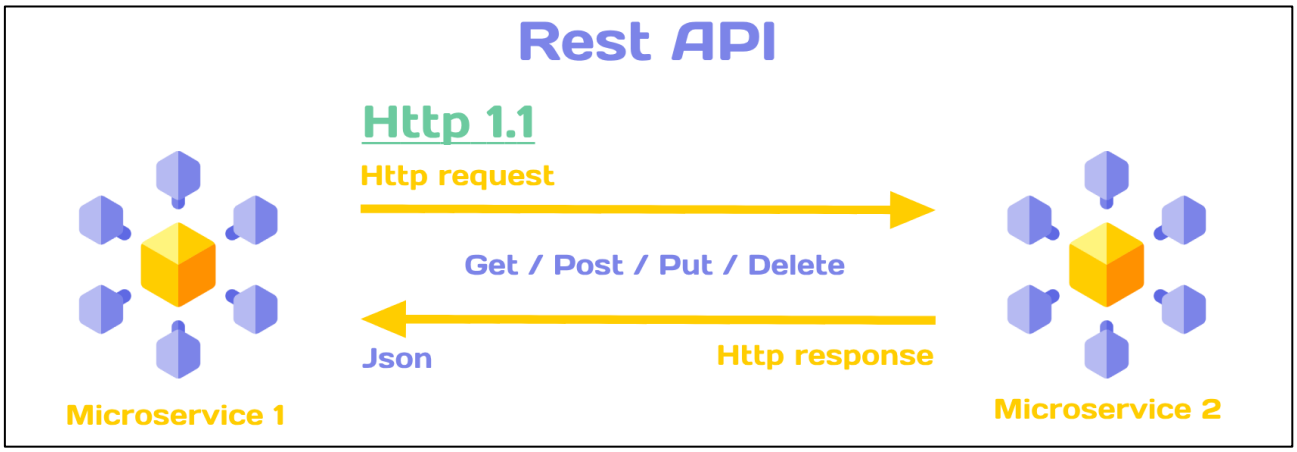

Рисунок 3 – Общение между микросервисами с помощью Rest API
Источник: Авторский материал

gRPC – современный протокол удаленного вызова процедур, разработанный компанией Google. Он основан на Protocol Buffers (protobufs) и предоставляет высокопроизводительный и эффективный способ взаимодействия между микросервисами. gRPC обеспечивает поддержку множества языков программирования и позволяет определить много различных типов данных. В отличие от REST API, протокол gRPC не предоставляет базовых методов, таких как GET, POST, PUT и DELETE. Вместо этого он использует механизмы RPC (Remote Procedure Call – удаленный вызов процедур), который позволяет клиентам вызывать методы, расположенные на удаленных серверах, так будто они вызывают локальные функции. Данные методы описываются в proto-файле и могут иметь различные типы (Рисунок 4): Unary (одинарный), Server Stream (серверный стриминг), Client Stream (клиентский стриминг) и Bi-directional stream (двунаправленный стриминг).

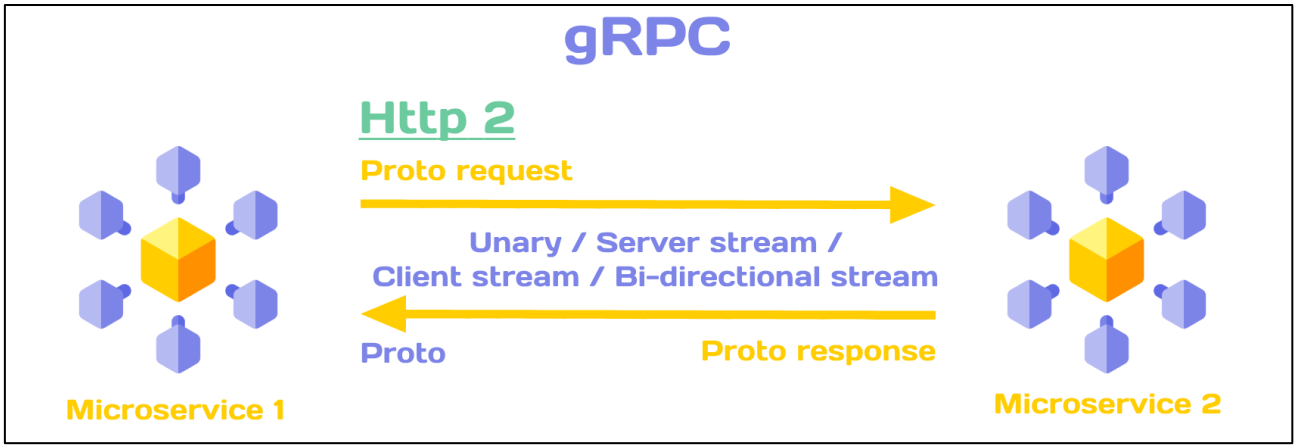

Рисунок 4 – Общение между микросервисами с помощью gRPC
Источник: Авторский материал

GraphQL – язык запросов для API, позволяющий клиентам запрашивать только те данные, которые им нужны, а не все данные, предоставляемые сервером. GraphQL позволяет определить точные требования клиентов к данным, что способствует оптимизации и

эффективности выполнения запросов. GraphQl передаёт данные с помощью объектов Json для выполнения запросов используются два типа методов: Query – запрос без изменения данных и Mutation – запрос с изменением данных (Рисунок 5).

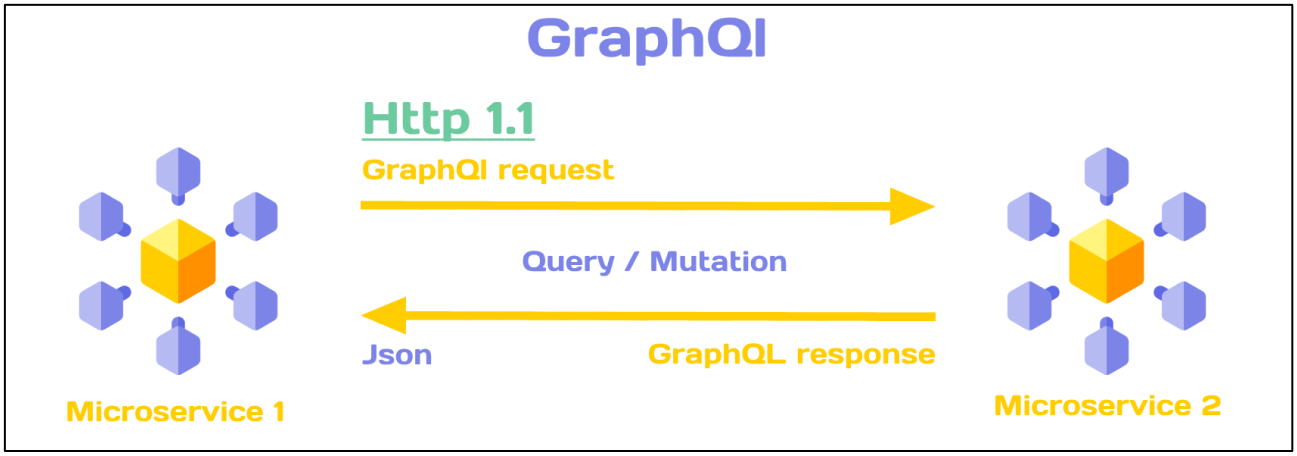

Рисунок 5 – Общение между микросервисами с помощью GraphQL
Источник: Авторский материал

Все перечисленные выше методы представляют синхронный способ обмена сообщениями. Данный способ применяется тогда, когда нужно получить ответ от сервера максимально быстро. Если скорость ответа или даже сам ответ от сервера нас не интересует, можно прибегнуть к асинхронному варианту взаимодействия микросервисов.

Асинхронный обмен сообщениями — это способ обмена сообщениями между микросервисами, в котором микросервисы общаются через через брокеры сообщений, такие как RabbitMQ или Apache Kafka. Это позволяет реализовать более слабую связь между компонентами системы и улучшить масштабируемость. Сервис, производящий сообщения, называется Producer, с помощью метода Produce передаёт сообщения в брокер сообщений. Сервис, принимающий сообщения, называется Consumer, с помощью команды Consume эти сообщения получает. Между сервисами и брокером сообщения передаются с помощью Http 1.1 в виде Json-объектов (Рисунок 6).

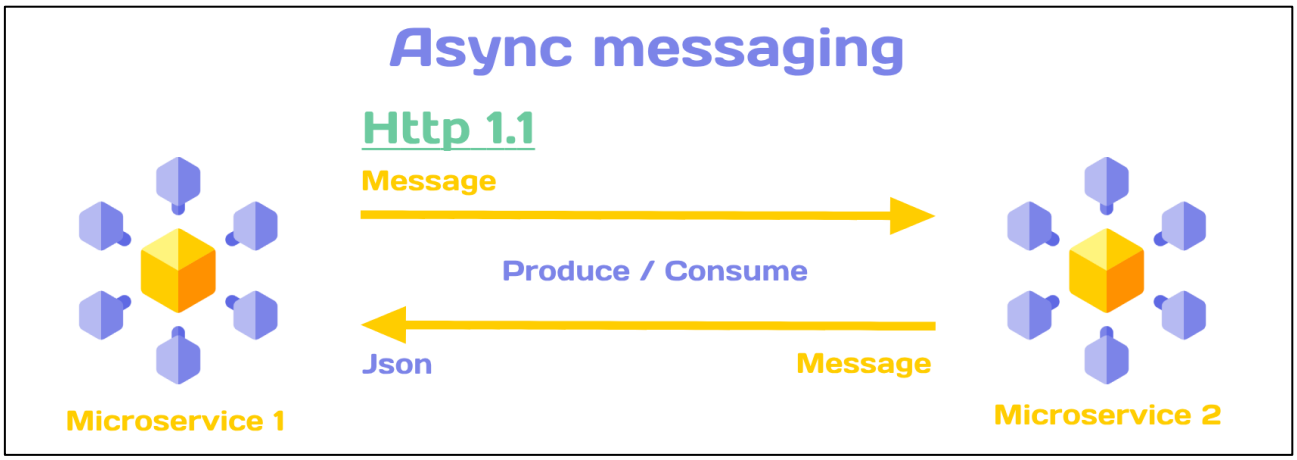

Рисунок 6 – Общение между микросервисами с помощью брокеров сообщений

Источник: Авторский материал

Важно отметить, что каждый способ коммуникации имеет свои преимущества и подходит для различных сценариев использования. Выбор способа коммуникации зависит от требований к разрабатываемой системе, типа передаваемых и получаемых данных, а также предпочтений и опыта команды разработчиков.

Приведем пример реализации микросервисной архитектуры на языке программирования C#. Для этого реализуем два микросервиса: первый будет отвечать за хранение информации о товарах, а второй – запрашивать эту информацию для последующей обработки и вывода полученных данных в консоль. Микросервисы будут взаимодействовать между собой через протокол gRPC.

Создадим новое решение с именем **MicroservicesExample.sln**, содержащее два отдельных проекта: **GoodsService,cs**, отвечающий за хранение данных о товарах, и **DisplayService.cs**, который будет запрашивать необходимую информацию о товарах у **GoodsService**, обрабатывать полученные данные и выводить их в консоль.

Перейдем к созданию proto-файла в проекте **GoodsService**, который будет описывать API для взаимодействия с этим сервисом. proto-файл представляет собой документ, содержащий описание всех доступных методов обмена сообщениями в рамках gRPC для данного сервиса. Это позволяет установить более эффективное взаимодействие между клиентскими и серверными приложениями, упрощая создание и обработку сообщений.

Кроме того в proto-файле определим методы обмена сообщениями с помощью протокола Buffers (protobuf), который является языком для определения структуры данных и сервисов взаимодействия с ними.

На Рисунке 7 представлена структура proto-файла для микросервиса **GoodsService**, который содержит один метод **GetGood**. Данный метод принимает запрос типа **GetGoodRequest**, содержащий идентификатор товара, и возвращает ответ типа **GetGoodResponse**, содержащий объект "Good", в котором содержится информация о товаре:

```
syntax = "proto3";
package GrpcGoodsService;
service GrpcGoodsService
{ rpc GetGood (GetGoodRequest) returns (GetGoodResponse) {} }

message Good
{ string Title = 1; }

message GetGoodRequest
{ int64 id = 1; }

message GetGoodResponse
{ Good good = 1; }
```

Рисунок 7 – Proto-файл микросервиса GoodsService

Источник: Авторский материал

После компиляции данного proto-файла будет получен полноценный C#-сервис, который

можно унаследовать и переопределить его методы для добавления необходимой логики обработки запросов и ответов.

На Рисунке 8 показано наследование автосгенерированного из proto-файла сервиса **GoodsService**. Как видно из рисунка, у нас появился описанный в proto-файле метод **GetGoods**, который получает на вход **GetGoodsRequest**, а отдает **GetGoodResponse**, содержащий объект Good. Для примера был добавлен объект Goods, который будет содержать информацию о товарах. Метод **GetGood** будет возвращать один из его объектов или ошибку, если индекс объекта находится за пределами массива.

```
     public class GoodsService :
GrpcGoodsService.GrpcGoodsService.GrpcGoodsServiceBase
{
    private static readonly string[] Goods = { "Кружка", "Рюкзак", "Ноутбук",
"Тетрадь", "Проектор" };

         public override async Task<GetGoodResponse> GetGood(GetGoodRequest
request, ServerCallContext context)
    {
        if (request.Id < 0 || request.Id > Goods.Length - 1)
           {throw new ArgumentOutOfRangeException(nameof(request.Id));}
            return new GetGoodResponse
            {
             Good = new Good { Title = Goods[request.Id] }
        };
    }
}
```

Рисунок 8 – Наследование автосгенерированного класса
Источник: Авторский материал

Теперь можно приступить к написанию клиента. Для этого мы должны скопировать в **DisplayService** вышеприведенный proto-файл. После выполнения сборки проекта **DisplayService**, аналогично проекту **GoodsService**, появится класс **GrpcGoodsServiceClient**, с помощью которого можно вызывать методы из сервиса **GoodsService** (Рисунок 9).

```
     static async Task
HandleGrpc(GrpcGoodsService.GrpcGoodsService.GrpcGoodsServiceClient client)
{
         Console.WriteLine($"Запрос к серверу: Id = 1");
         var response = await client.GetGoodAsync(new GetGoodRequest { Id = 1
});
         Console.WriteLine($"Ответ сервера: {response.Good.Title}");
}
```

Рисунок 9 – Метод GetGoodsAsync класса GrpcGoodsServiceClient
Источник: Авторский материал

Вызвав метод GetGoodsAsync внутри функции Main (Рисунок 10), мы получим результат, показанный на рисунке 11. Также на этом рисунке представлена архитектура

созданного решения.

```
     public static async Task Main()
{
    var host = await Startup.CreateHost();
    var grpcClient = host.Services.GetRequiredService
            <GrpcGoodsService.GrpcGoodsService.GrpcGoodsServiceClient>();
         Console.WriteLine($"Запрос к серверу: Id = 1");
    var response = await grpcClient.GetGoodAsync(new GetGoodRequest { Id = 1
});
    Console.WriteLine($"Ответ сервера: {response.Good.Title}");
}
```

Рисунок 10 – Вызов метода GetGoodsAsync класса GrpcGoodsServiceClient
Источник: Авторский материал

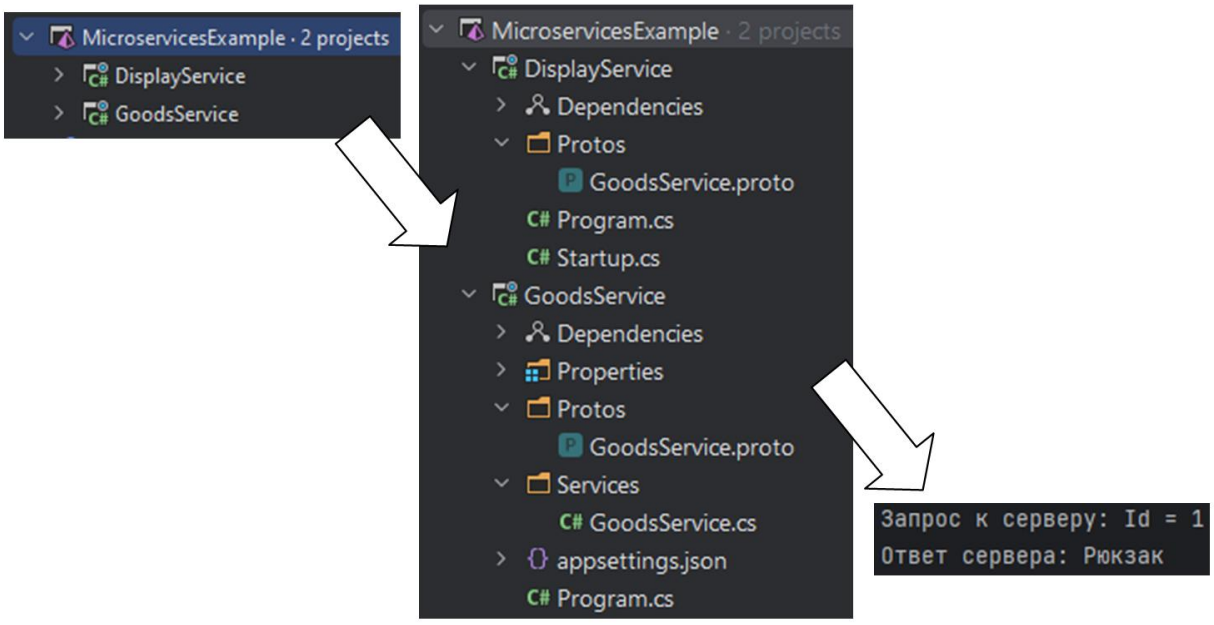

Рисунок 11 – Архитектура созданного решения и результат выполнения
Источник: Авторский материал

Таким образом, на данном примере было показано, как настроить взаимодействие между двумя независимыми сервисами на основе протокола gRPC.

Заключение.

Микросервисная архитектура представляет собой мощный и инновационный подход к разработке программного обеспечения, который стремительно завоевывает популярность в сфере информационных технологий. Её преимущества включают гибкость, масштабируемость, улучшенный процесс разработки и отказоустойчивость. Однако, её внедрение требует дополнительных усилий, хорошего понимания современных технологий и умения обеспечивать безопасность системы.

Список литературы

1. Никитин, И. В. Сравнение подходов монолитной архитектуры и микросервисной архитектуры при реализации серверной части веб-приложения / И. В. Никитин, Т. Ю. Гриценко // Дневник науки. – 2020. – № 3(39). – 22с.
2. Надейкина, Л. А. Распределенные системы, построенные на базе микросервисной архитектуры / Л. А. Надейкина, Н. И. Черкасова // Инновационные, информационные и коммуникационные технологии. – 2019. – № 1. – С. 300-304.
3. Караханова, А. А. Анализ микросервисной архитектуры, монолитных приложений, архитектуры SOA / А. А. Караханова // Синергия Наук. – 2020. – № 46. – С. 255-262.

References

1. Nikitin, I. V. Comparison of approaches of monolithic architecture and microservice architecture in the implementation of the server part of a web application / I. V. Nikitin, T. Y. Gritsenko // Diary of Science. – 2020. – № 3(39). – p. 22.
2. Nadeikina, L. A. Distributed systems based on microservice architecture / L. A. Nadeikina, N. I. Cherkasova // Innovative, information and communication technologies. – 2019. – No. 1. – pp. 300-304.
3. Karakhanova, A. A. Analysis of microservice architecture, monolithic applications, SOA architecture / A. A. Karakhanova // Synergy of Sciences. – 2020. – No. 46. – pp. 255-262.