GeekBrains

Message-driven architecture

Принципы асинхронных и синхронных взаимодействий

На этом уроке

1. Вспомним понятия синхронных и асинхронных коммуникаций;
2. Создадим небольшое приложение демонстрирующее разницу;
3. Сравним эти два стиля взаимодействия;
4. Поймем в каких случаях необходимо использовать одно из них.

Оглавление

Синхронное и асинхронное взаимодействия

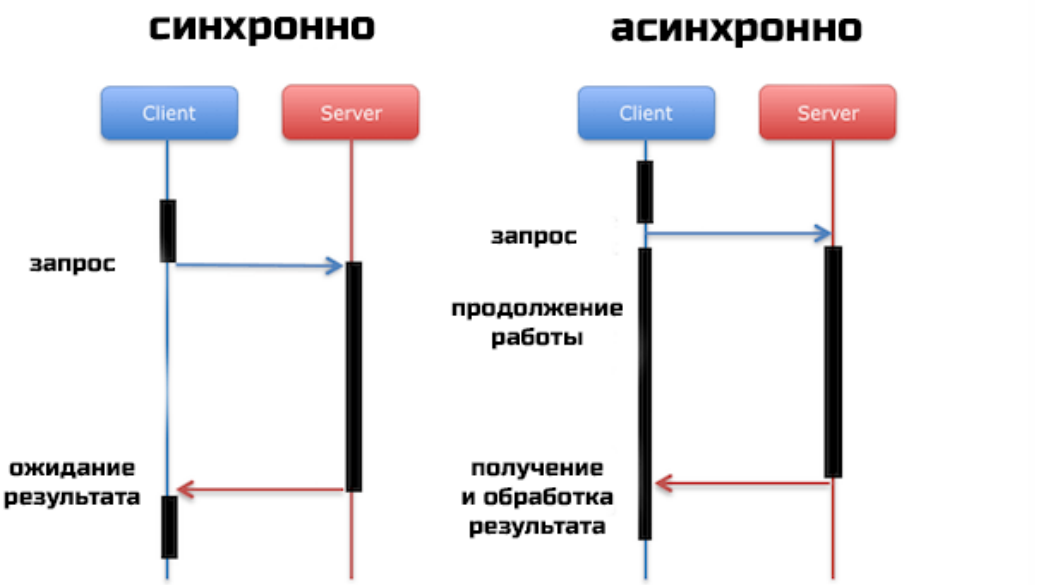
Основной характеристикой способа взаимодействия является его синхронность или асинхронность. Скорее всего, на этапе обучения основам языка и платформы вы уже сталкивались с понятиями синхронности и асинхронности, например при изучении async/await, TAP и т.д.. Сейчас эти понятия выходят на новый уровень своей значимости, так как приложения построенные для обеспечения бесперебойной работы под высокой нагрузкой в современных корпорациях опираются на два этих подхода.

Для осуществления коммуникаций у всех компонент ПО есть свое API ввода-вывода как раз к нему синхронность и асинхронность имеют прямое отношение. Для того чтобы начать какой-то процесс его надо инициировать, например, нажать на кнопку (ввод). Затем, мы хотим увидеть какой-то результат, понять что программа выполнила свою работу (вывод). Далее рассмотрим как приложение ведет себя в двух разных вариантах реализации API.

Представьте перед собой любую офисную программу, например, Excel. Вам срочно нужно посчитать несколько сложных формул, вы ввели все необходимые данные, ввели формулу в нужное поле и запустили расчет. Вы уже готовы начинать вводить следующие данные и получать результат, но Excel еще не справился с первой формулой! Кнопки недоступны, не кликаются, сделать ничего нельзя и приходится ждать окончания первого расчета чтобы продолжить следующий. Ситуация выглядит не очень приятной, но такова реальность если API ввода-вывода приложения реализовано синхронно. Что это значит более формально? Если выполнение программы синхронизируется с процессами ввода и вывода, то есть мы нажали кнопку, программа выполнила все необходимые действия, мы получили вывод и в промежуток времени от первого клика до вывода мы не могли совершать других действий - это синхронный (или блокирующий) вызов. Синхронные вызовы реализуют формат общения в стиле запрос/ответ. Например, когда вы говорите по телефону, навряд ли вы можете говорить с кем-то ещё, ведь вы сосредоточены на одном собеседнике и не готовы отвлекаться, а если и готовы, то вам придется прервать текущий разговор не получив полного “вывода”.

А теперь представьте, что вы провели те же действия, но в этот момент ничего

не зависло и вы можете продолжать свои расчеты, не затрачивая время на ожидание первого результата. Здорово, не правда ли ? Это и есть асинхронный вызов (или неблокирующий) - нам не нужно ждать ответа, “синхронизироваться” с ним, мы получим его ровно тогда, когда он будет готов. А до тех пор можем спокойно использовать любую другую интересующую нас функциональность. Примером из реальной жизни может быть простой тостер. Ведь когда вы нажимаете на кнопку, вы не стоите возле него поддерживая кнопку (если он, конечно, исправен), а заняты своим делом ? В нужный момент он “сообщит” характерным звуком о готовности и вы сможете насладиться продуктами его работы. Также хорошим примером может служить любой мессенджер или email, но тут я уже предлагаю вам самим поразмышлять над синхронными и асинхронными процессами в окружающей жизни.



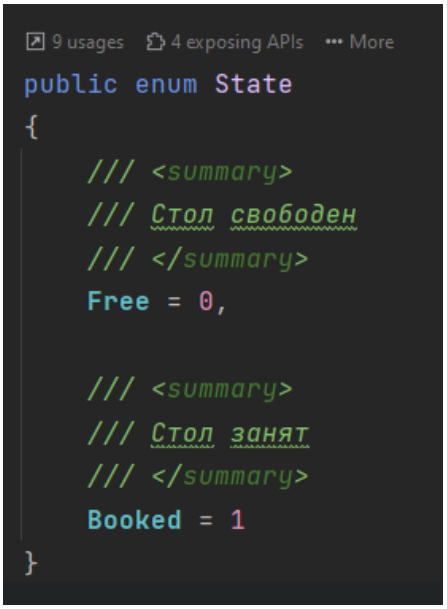
Создание синхронного и асинхронного API

Сейчас мы попробуем создать небольшое приложение, которое реализует оба примера чтобы посмотреть реальную разницу на практике.

Познакомимся с процессом на основе реализации небольшого количества логики, связанного с бронированием столика в ресторане. Разберем 2 случая: мы бронируем столик по телефону и бронируем столик через сайт/мессенджер. В процессе разработки будут некоторые допущения и “плохой” код (у него обязательно будет пометка!). В качестве API ввода-вывода будем использовать консоль.

Итак, мы хотим бронировать столики в ресторане. Нам понадобятся: столы, нерасторопные менеджеры (чтобы мы успели заметить их работу) и сам ресторан.

Начнем со стола. У стола будет идентификатор, количество мест и его статус - забронирован или не забронирован. Создадим сначала перечисление с двумя элементами: стол свободен и стол занят.



Теперь опишем сам стол:

```
5 usages  1 exposing API
public class Table
{
    6 usages
    public State State { get; private set; }
    3 usages
    public int SeatsCount { get; }
    4 usages
    public int Id { get; }

    1 usage
    public Table(int id)
    {
        Id = id; //в учебном примере просто присвоим id при вызове
        State = State.Free; // новый стол всегда свободен
        SeatsCount = Random.Next(2, 5); //пусть количество мест за каждым столом будет случайным, от 2х до 5ти
    }

    3 usages
    public bool SetState(State state)
    {
        if (state == State)
                return false;

        State = state;
        return true;
    }
```

Осуществлять бронь столов мы будем по звонку в ресторан и для этого нам понадобится… сам ресторан! В нем должно быть какое-то количество столов и возможность их бронировать. Создадим класс Restaurant и поместим в него несколько столов.

```
1 usage
public class Restaurant
{
    private readonly List<Table> _tables = new ();

    1 usage
    public Restaurant()
    {
        for (ushort i = 1; i <= 10; i++)
        {
            _tables.Add( item: new Table(i));
        }
    }
```

Теперь у нас ресторан на 10 столов. Отлично, мы почти готовы к открытию. Для упрощения не будем вдаваться в детали, кто и как у нас бронирует столики. Ведь клиенту нет особой разницы, кто для него это сделает, главное чтобы в результате он пришел к назначенному времени и все было в порядке. Предположим, что у нас есть два способа бронирования: через смс/мессенджер и по звонку.

```
1 usage
public void BookFreeTable(int countOfPersons)
{
    Console.WriteLine("Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, оставайтесь на линии");

    var table = _tables.FirstOrDefault(t:Table => t.SeatsCount > countOfPersons
                                        && t.State == State.Free);
    Thread.Sleep(millisecondsTimeout:1000 * 5); //у нас нерасторопные менеджеры, 5 секунд они находятся в поисках стола

    Console.WriteLine(table is null
        ? $"К сожалению, сейчас все столики заняты"
        : $"Готово! Ваш столик номер {table.Id}");
}
```

И версия по смс/мессенджер

```
1 usage
public void BookFreeTableAsync(int countOfPersons)
{
    Console.WriteLine("Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, Вам придет уведомление");
    Task.Run(async () =>
    {
        var table = _tables.FirstOrDefault(t:Table => t.SeatsCount > countOfPersons
                                            && t.State == State.Free);
        await Task.Delay(1000 * 5); //у нас нерасторопные менеджеры, 5 секунд они находятся в поисках стола
        table?.SetState(State.Booked);

        Console.WriteLine(table is null
            ? $"УВЕДОМЛЕНИЕ: К сожалению, сейчас все столики заняты"
            : $"УВЕДОМЛЕНИЕ: Готово! Ваш столик номер {table.Id}");
    });
}
```

Отлично, наш ресторан готов к работе! Пора приглашать первых клиентов. Для этого добавим в класс Program.cs в метод Main следующий код:

```
Console.OutputEncoding = System.Text.Encoding.UTF8;
var rest = new Restaurant();
while (true)
{
    Console.WriteLine("Привет! Желаете забронировать столик?\n1 - мы уведомим Вас по смс (асинхронно)" +
                      "\n2 - подождите на линии, мы Вас оповестим (синхроннно)"); //приглашаем ко вводу
    if (!int.TryParse(s: Console.ReadLine(), out var choice:int ) && choice is not (1 or 2))
    {
        Console.WriteLine("Введите, пожалуйста 1 или 2"); //всегда нужно защититься от невалидного ввода
        continue;
    }

    var stopWatch = new Stopwatch();
    stopWatch.Start(); //замерим потраченное нами время на бронирование, ведь наше время - самое дорогое что у нас есть
    if (choice == 1)
    {
        rest.BookFreeTableAsync( countOfPersons: 1); //забронирцем с ответом по смс
    }
    else
    {
        rest.BookFreeTable( countOfPersons: 1); //забронирцем по звонку
    }

    Console.WriteLine("Спасибо за Ваше обращение!"); //клиента всегда нужно порадовать благодарностью
    stopWatch.Stop();
    var ts:TimeSpan = stopWatch.Elapsed;
    Console.WriteLine($"{ts.Seconds:00}:{ts.Milliseconds:00}"); //выведем потраченное нами время
}
```

Пора запускать и пробовать. Синхронный вызов вернул нам следующее

```
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
2 - подождите на линии, мы Вас оповестим (синхроннно)
2
Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, оставайтесь на линии
Готово! Ваш столик номер 1
Спасибо за Ваше обращение!
05:39
```

Отлично! Все сработало, только мы были на линии 5 секунд и не могли в это время делать ничего другого. Как мы, так и наш менеджер на телефоне. Зато у нас есть результат и мы его подождали.

Попробуем асинхронный запуск

```
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
2 - подождите на линии, мы Вас оповестим (синхроннно)
1
Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, Вам придет уведомление
Спасибо за Ваше обращение!
00:04
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
2 - подождите на линии, мы Вас оповестим (синхроннно)
1
Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, Вам придет уведомление
Спасибо за Ваше обращение!
00:00
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
2 - подождите на линии, мы Вас оповестим (синхроннно)
УВЕДОМЛЕНИЕ: Готово! Ваш столик номер 1
УВЕДОМЛЕНИЕ: Готово! Ваш столик номер 1
```

Теперь наши бронирования не блокируются и мы можем сразу же бронировать столики не тратя свое время. Напомню, что формат времени в данном случае секунды:милисекунды. Правда вышла одна накладка - теперь наша бронь не синхронизирована относительно друг друга и мы два раза забронировали один и тот же столик. С этим стоит разобраться и это будет в домашнем задании.

По результатам работы мы видим, что доступность приложения реализующего асинхронное API выше, чем с синхронными API. Но с этим возрастает и сложность, и количество подводных камней. С ними подробнее будем разбираться на следующих занятиях.

Сравнение синхронного и асинхронного API

Могло показаться, что асинхронные взаимодействия превосходят своей удобностью для конечного пользователя синхронные и это действительно так в каких-то случаях. Но в каких-то необходимо применять и синхронные взаимодействия. Здесь мы разберемся когда применять тот или иной подход и какие у них плюсы и минусы.

	Плюсы	Минусы
Синхронное	- простота - удобство отладки - легкость тестирования	- снижение отказоустойчивости приложения - увеличение времени ответа клиенту - высокая связанность.
Асинхронное	- высокая доступность приложения - отсутствие синхронного времени ожидания ответа - слабая связанность	- API сложнее в реализации, тестировании, отладке. - потенциальное дублирование данных

Как же выбрать нужный нам вариант ? Есть следующая схема выбора разных стилей коммуникации. Их можно разделить на 2 уровня. Первый уровень определяет выбор между отношениями “один к одному” и “один ко многим”:

- “Один к одному” - каждый клиентский запрос обрабатывается ровно одним обработчиком;
- “Один ко многим” - каждый запрос обрабатывается несколькими обработчиками.

Второй уровень определяет выбор между синхронным и асинхронным взаимодействием:

- синхронное - клиент рассчитывает на своевременный ответ и может даже заблокироваться на время ожидания;

- асинхронное - клиент не блокируется, а ответ, если таковой придет, может быть отправлен не сразу.

Есть несколько способов взаимодействия “один ко многим”:

- Запрос / ответ: клиент инициирует запрос и ждет ответа. Клиент ожидает, что этот ответ придет немедленно. В приложении на основе потоков ожидающий процесс может вызвать блокировку потока (наш случай).

- Уведомление (также известное как односторонний запрос): клиентский запрос отправляется на сервер, но от сервера не ожидается ответа.

- Запрос / асинхронный ответ: клиент отправляет запрос на сервер, и сервер отвечает на запрос асинхронно. Клиент не блокирует и спроектирован таким образом, что ответ по умолчанию не приходит сразу.

Есть несколько способов взаимодействия “один-ко-многим”:

- Режим публикации / подписки: клиент публикует сообщения с уведомлением и использует ноль или более заинтересованных компонент системы (это почти как наши уведомления клиента).

- Режим публикации / асинхронного ответа: клиент публикует сообщение с запросом, а затем ожидает ответа от интересующей службы (а это если бы еще хотели добавить в нашу логику депозит).

Каждое приложение представляет собой комбинацию этих режимов. Для некоторых служб достаточно одного механизма коммуникация; для других служб

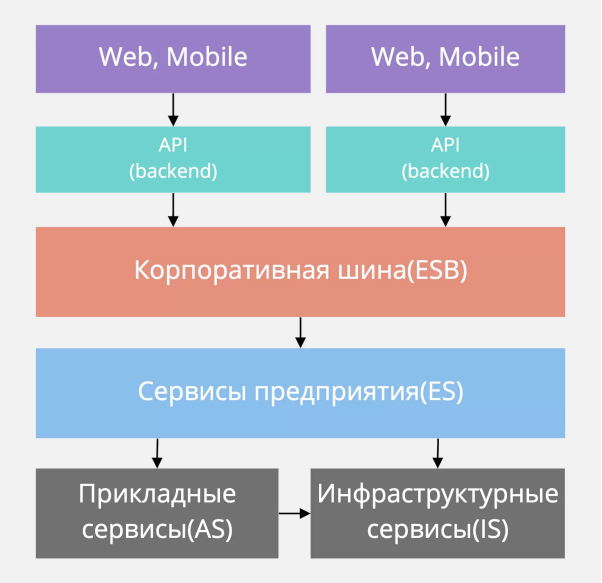
требуется комбинация нескольких механизмов. Здесь необходимо отталкиваться от бизнес потребностей в каждой конкретной задаче.

Некоторые из терминов могут показаться новыми, например, слабая и высокая связанность, какие проблемы отказоустойчивости могут быть и какие что это за загадочное “сложнее”. Со всем этим мы столкнемся в следующих шагах, подробно разбирая темы необходимые для освоения message-based architecture.

Service oriented architecture (SOA) и Microservice oriented architecture (MSA)

Ранее мы рассмотрели коммуникации на уровне одного приложения - монолита. Простые вызовы в рамках одного процесса, которые позволяют организовать логику более оптимизировано. Монолитные приложения занимают свою нишу в разработке современного ПО, но большие компании, например, OZON, Google, Raiffeisenbank, Beeline и др. Уже не могу существовать в рамках одного приложения. Становится слишком мало места, разработчики мешают друг другу, скорость доставки до продуктива увеличивается и в связи с этим начинают себя проявлять иные подходы, в которых асинхронная коммуникация является одним из основополагающих факторов. Сейчас мы познакомимся с ними.

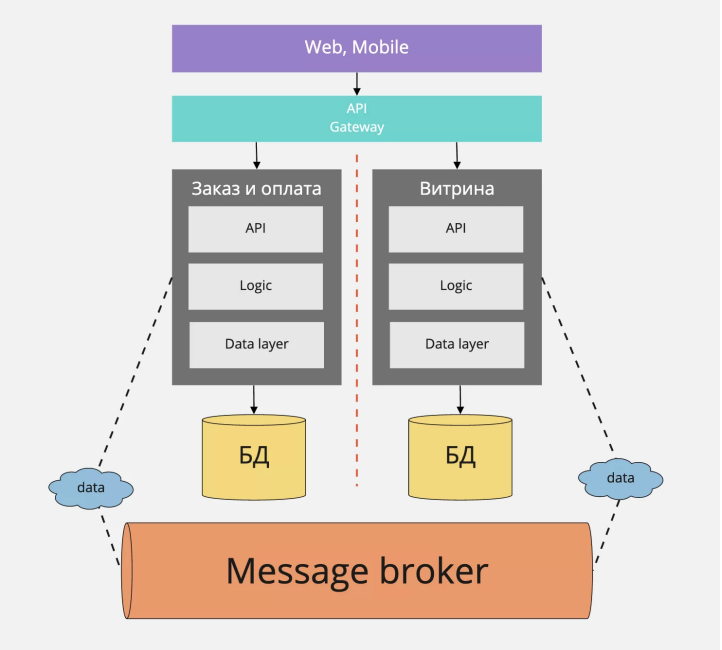
SOA - в основе подхода лежат несколько основных идей – переиспользование сервисов и корпоративная шина. Разработчики стремятся разбить систему на сервисы таким образом, чтобы их можно было использовать повторно. Взаимодействие и маршрутизация осуществляется через корпоративную шину ESB. Типичная SOA архитектура показана на рисунке ниже.



Как вы могли заметить, проектируя архитектуру в данном стиле, мы имеем очень большое количество слоев и как следствие команд, которые ими владеют. Любой запрос пронизывает все слои системы, в большей степени напоминая

монолитную архитектуру. Но данный тип архитектуры намного сложнее, потому что является распределенной архитектурой.

MSA - микросервисы в отличие от SOA, наоборот, избегают повторного использования, применяя философию - предпочтительнее дублирование, а не зависимость от других сервисов. Повторное использование предполагает связанность, а архитектура микросервисов в значительной степени старается ее избегать. Это достигается за счет разбиения системы на сервисы по ограниченным контекстам (бизнес областям). Типичная MSA архитектура показана на рисунке ниже.



В отличие от SOA каждый сервис обладает всеми необходимыми для функционирования частями – имеет свою собственную базу данных и существует как независимый процесс. Такая архитектура делает каждый сервис физически разделенным, самодостаточным, что ведет с технической точки зрения к архитектуре без разделения ресурсов.

Сервисы раскрываются для потребителей также через слой API, но его стараются проектировать с полным отсутствием какой-либо логики. Это фактически просто проксирование API сервисов во вне.

Взаимодействие между сервисами сводится к обмену данными, используя брокер сообщений. Именно к обмену данными, а не вызову методов из других сервисов.

SOA и MSA не являются основной темой этого курса, поэтому подробнее на них останавливаться не будем. Можно сказать, что они несколько больше приближены к решению конкретных бизнес-задач, мы же рассматриваем MDA, которая может помочь организовать грамотное взаимодействие в SOA и MSA обеспечив отличную масштабируемость и отказоустойчивость. MDA несет в себе определенную сложно настройки и организации, подводные камни и проблемы, знания о большинстве из которых нужно получать своим опытом и опытом коллег из книг, конференций и т.д. Основные идеи, которые помогут избежать большую часть проблем будут рассмотрены в дальнейших уроках.

Глоссарий

API - это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это». Если переводить на русский, это было бы слово «договор». Договор между двумя сторонами.

Синхронность - термин «синхронный» в контексте разработки означает, что вычисление происходит сразу, как серия последовательных шагов, и до его завершения больше ничего не происходит.

Асинхронность - логическая(!) оптимизация выполнения программы, которая может работать как в одном, так и во многих потоках. Мы отдаем некоторую команду и не ожидая ее результат продолжаем свою работу. В течение какого-то времени мы получим результат силами реализация различных механизмов асинхронности.

Практическое задание

- Создать возможность снять бронь синхронно и асинхронно, используя для этого номер забронированного стола;
- Выделить логику для отправки уведомления в отдельный класс, он будет отвечать за все вопросы связанные с коммуникациями с клиентами, добавить задержку (они будет имитировать создание сообщения) и сделать вызов уведомления асинхронным;
- Добавить автоматическое “снятие брони”. Например, раз в 20 секунд при наличии забронированных мест - бронь должна слетать. Асинхронно, независимо от ввода-вывода. Подсказка: можно использовать таймер.
- (*) Добавить синхронизацию бронирований для множественных асинхронных и синхронных вызовов, это значит, что бронируя столики не дожидаясь предыдущих ответов мы должны получать последовательный результат.
- (**) Добавить ограничение на количество мест за столом относительно количества гостей, например, если придет 5 человек, то сесть можно только за стол где больше 5 мест. Столы сдвигать можно. Должна быть синхронизация между разными одновременными бронированиями.

  Пример:

```
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
2 - подождите на линии, мы Вас оповестим (синхроннно)
1
Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, Вам придет уведомление
Спасибо за Ваше обращение!
00:05
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
2 - подождите на линии, мы Вас оповестим (синхроннно)
1
Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, Вам придет уведомление
Спасибо за Ваше обращение!
00:00
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
2 - подождите на линии, мы Вас оповестим (синхроннно)
2
Добрый день! Подождите секунду я подберу столик и подтвержу вашу бронь, оставайтесь на линии
УВЕДОМЛЕНИЕ: Готово! Ваш столик номер 1
УВЕДОМЛЕНИЕ: Готово! Ваш столик номер 2
Готово! Ваш столик номер 3
Спасибо за Ваше обращение!
15:30
Привет! Желаете забронировать столик?
1 - мы уведомим Вас по смс (асинхронно)
```

Используемые источники

1. Микросервисы: Паттерны разработки и рефакторинга - Крис Ричардсон - Питер - 2021 год - 544 страницы
2. https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/
3. https://microarch.ru/blog/soa-vs-msa
4. Для задания со звездочкой - https://docs.microsoft.com/ru-ru/dotnet/standard/threading/overview-of-synchronization-primitives