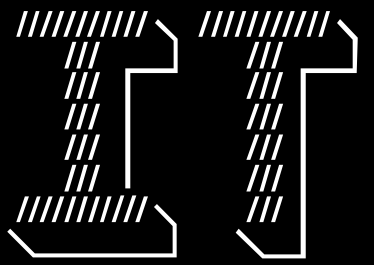
From: C# Nullability

To: Functional programming approach (Unit, Tagged Union, Optional, Try/Result)

Andrei Sergeev, Pavel Moskovoy

Райффайзен



Спикер

Andrei Sergeev

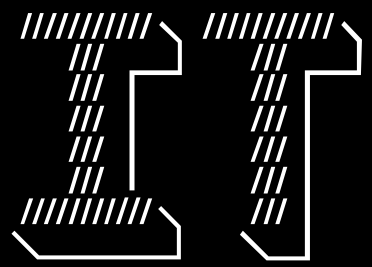
E-mail: a.sergeev.official@gmail.com

Занимается проектированием архитектуры и разработкой программного обеспечения масштабируемого микросервисного back-end.

В качестве основной платформы разработки использует .NET — начав работать с .NET Framework 1.1 и продолжая работать с актуальными .NET Core 3.1 и C# 8.

Использует функциональный подход в разработке, используя при этом и лучшие практики смежных и «конкурирующих» принципов, языков и платформ — F#, Java, Kotlin, Ruby, etc.

Райффайзен

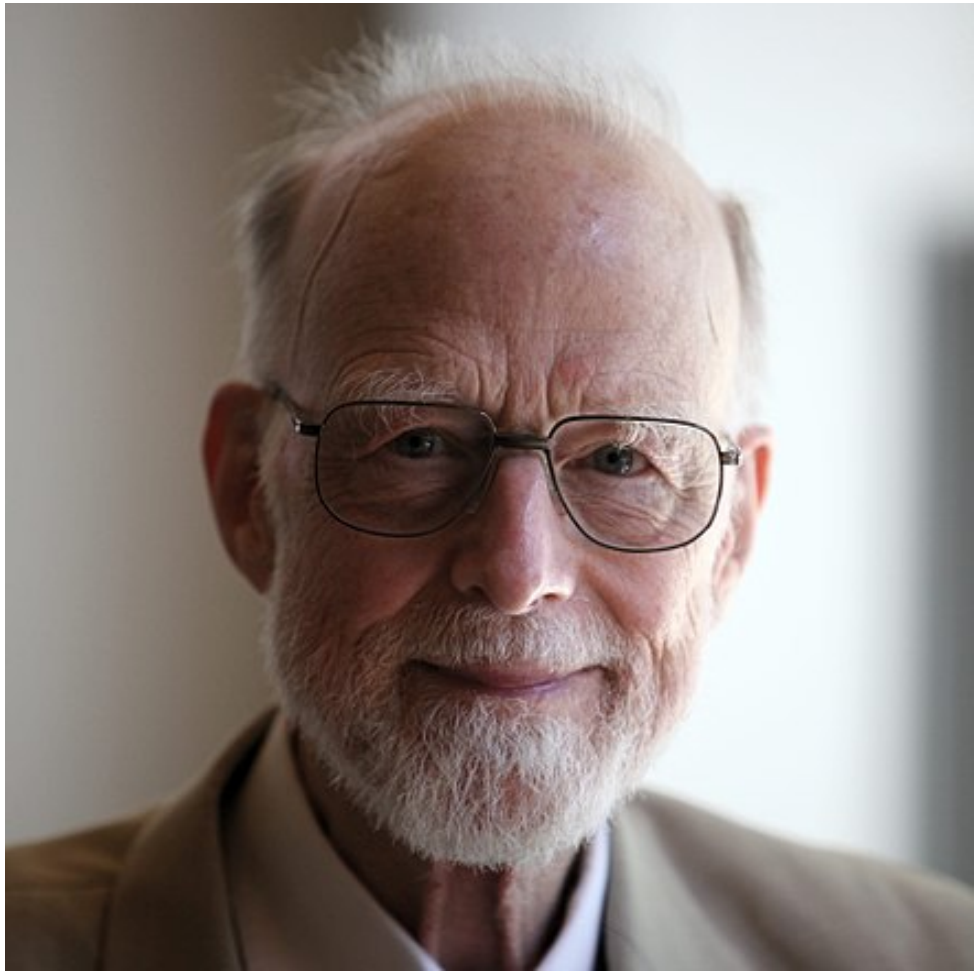


The purposes

Райффайзен

- Минимизация NullReferenceException и других технических исключений в доменном коде на C#
- Презентация подлинно функционального подхода при разработке на современных C# и .NET Core
- Предложение сообществу компактного фреймворка для применения ФП
- Повлиять на развитие библиотеки .NET Standard

The Billion Dollar Mistake



Charles Antony Richard Hoare

Изобрел Null reference при разработке системы типов языка ALGOL

«I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.»

https://en.wikipedia.org/wiki/Tony_Hoare

Null reference: How that happens

Райффайзен IT

Технические исключения при разработке, тестировании и в Production.
Не то, чего хочется при разработке доменной логики.

Ticket-666: «Возник NullReferenceException при оформлении заказа продукта»

Description: **«Пользователь не смог оформить заказ»**

Null reference: Why that happens

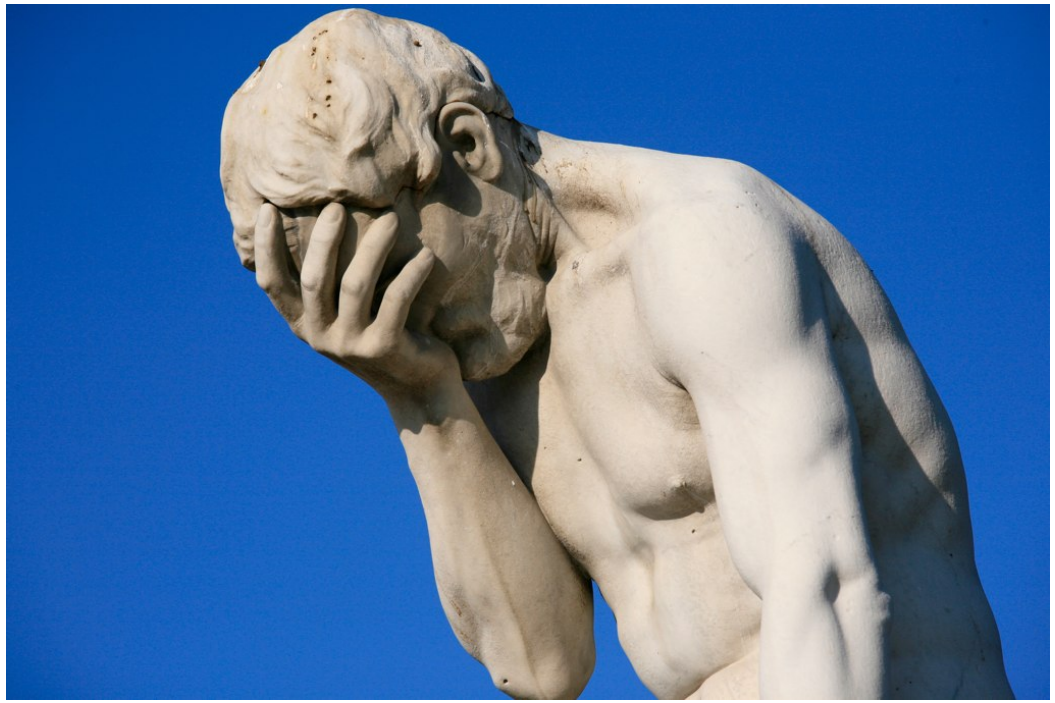

```
public sealed class Order
{
    public Guid Id { get; }
    public long Number { get; }
    public Guid CustomerId { get; }
    public Order(in Guid id, in long number, in Guid customerId)
    {
        Id = id;
        Number = number;
        CustomerId = customerId;
    }
}
```

Null reference: Why that happens - 1


```
// 1. Нарушение статической (static) типизации: превращение типа в тип-сумму
// 2. Слабая (Weak) типизация: присвоение без приведения типа
Order order = null;

// 3. Требование явного присвоения типа (Сильная, Strong-типизация)
// могло бы снизить число ошибок
Order orderOk = (Order)null;

// 4. Отсутствие контроля компилятора при обращении к экземпляру типа-суммы
var orderNumber = orderOk.Number;

// 5. Должно было быть так:
if (orderOk is not null)
{
    var orderNumberOk = orderOk.Number;
}
```

Null reference: Why that happens - 2

Приходится думать не о доменной логике, а технических вещах — ссылочная переменная или нет??



Автор: Alex E. Proimos - https://www.flickr.com/photos/proimos/4199675334/, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=22535544

Null reference: Why that happens - 3


```
// КАК нам обработать extraOrder??
public void ProcessOrder(in Order order, in Order extraOrder)
{
    if (order is null)
    {
        throw new ArgumentNullException(nameof(order));
    }

    //if (extraOrder is null) // ТАК??
    //{
    //    throw new ArgumentNullException(nameof(order));
    //}

    var number = order.Number;

    if (extraOrder is not null) // или ТАК??
    {
        var extraNumber = extraOrder.Number;
    }
}
```

Null reference: Why that happens - 4

Райффайзен

Смешение технического и доменного уровней абстракции:

Использование null как признака отсутствия значения.

==>> Ошибки и неподдерживаемый legacy-код

C# Nullability: The Beginnings

Райффайзен

2005: public struct Nullable<T> where T : struct

```
int? i = 0;
int? j = null;
int? k = default;
int notNullable = default;

if (i is null)
{
    // i is null
}

if (i.HasValue is false)
{
    // i does not have a value
}
```

C# Nullability: The Beginnings — For what?

public struct Nullable<T> where T : struct

```
int? i = 0;
int? j = null;
```

- Для чего?: Удобное чтение из БД nullable-чисел
- А что делает?: Закрепляет путаницу между null и отсутствующим значением; закладывает несовместимость nullabulity/absentability классов и структур в будущем
- Решает проблему «Absentability» кардинально?: Нет

C# Nullability: Nowadays, 2019

В C# 8 стало можно вот в такое для классов:

```
public void ProcessOrder(in Order order, in Order? extraOrder)
{
    if (order is null)
    {
        throw new ArgumentNullException(nameof(order));
    }

    // do something with Order
    var number = order.Number;

    // do something with extra Order
    if (extraOrder is object)
    {
        var extraNumber = extraOrder.Number;
    }
}
```

C# Nullability: Nowadays, 2019 - 2

И вот в такое:

```
public Order? FindOrder(in Guid orderId)
{
    // TODO: Find order
    return null;
}
```

C# Nullability: Nowadays, 2019 - 3

Райффайзен

Как это работает?:

- Поддержка только на уровне метаданных, компилятора и IDE, с сохранением тех же nullable ссылочных типов
- Несовместимо с Nullable<T>-структурами
- Отсутствие единой «коробки» для классов и структур вида Optional<T> или Absentable<T>

C# Nullability: Nowadays, 2019 - 4

Какие плюсы?:

- Nullability для классов – в любом случае хорошо, помогает бороться с NullReferenceException
- На минимальном уровне, но введена совместимость nullability классов и структур:

```
public interface IMessageQueue<TMessage> where TMessage : notnull
{
    public TMessage GetMessage();
}
```

Где TMessage может быть и классом, и структурой

C# Nullability: Nowadays, 2019 - 5

Какие минусы?:

Не получится вот в такое, и «теряется весь смысл»:

```
public interface IMessageQueue<TMessage>
where TMessage : notnull
{
    public TMessage? GetMessage();
}
```

C# Nullability: Nowadays, 2019 - 6

Какие еще минусы?:

Null может придти с помощью null propagation operator (!), и контракты публичного API придется проверять.

```
var customer = new Customer(firstName: null!, lastName: null!, middleName: null);

public sealed class Customer
{
    public string FirstName { get; }
    public string LastName { get; }
    public string? MiddleName { get; }
    public Customer(in string firstName, in string lastName, in string? middleName)
    {
        FirstName = firstName ?? throw new ArgumentNullException(nameof(firstName));
        LastName = lastName ?? throw new ArgumentNullException(nameof(lastName));
        MiddleName = middleName;
    }
}
```

C# Nullability: Nowadays, 2019 - 7

Райффайзен

А еще минусы?:

Внешние библиотеки могут быть разработаны и собраны с выключенным режимом nullability (#nullable disable).

```
User user = dbContext.Find<User>(id); // EF Core DbContext.Find Method

Console.WriteLine(user.Name);
```

==>> внезапно, NullReferenceException

Возврат null методом Find является частью контракта («не найдено»), но компилятор не выдаст предупреждение, что в переменную, объявленную как «User user» (а не «User? user»), может быть присвоен null.

См. https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext.find?view=efcore-3.1#Microsoft_EntityFrameworkCore_DbContext_Find__1_System_Object___

C# Nullability: Nowadays, 2019 - 8

Райффайзен

И… ещё минусы?:

С выходом C# 8, для однозначной трактовки/компилируемости кода необходима определенность, для какого режима nullability написан код

==>> потенциальные ошибки в доменной логике после сборки

==>> зависимость от глобальных настроек проекта, либо необходимость указывать режим директивы nullable в исходных файлах

Это заслуживает темы отдельного доклада, но что мы предлагаем сейчас? ==>>

C# Nullability: Nowadays, 2019 — Advice

- Для однозначной интерпретации кода, с выходом C# 8, мы предлагаем указывать в каждом файле исходного кода режим nullability (#nullable enable/disable)

- Также предлагаем устанавливать опцию TreatWarningsAsErrors для предупреждений компилятора для nullability feature

- В своих проектах мы во всех файлах исходных кодов указываем #nullable enable, т. к. полагаем, что лучше уже сейчас ориентироваться на эту технологию

Functional programming approach

Райффайзен

- Начали с Nullability и исследования Nullable ReferenceTypes для классов, а закончили разработкой и внедрением компактного фреймворка для разработки на C# в функциональном стиле

- Включает типы Optional<T> (единый для классов и структур), Result<TValue, TError>, Tagged Union, Unit, асинхронный конвейер (AsyncPipeline) и методологию использования

- Работает в Production и в ближайшее время ожидается поступление в OpenSource-репозитирий:

  https://github.com/Raiffeisen-DGTL/ViennaNET

Functional programming — Optional (1)

1. Известный в функциональном программировании тип Optional<T> для хранения значение либо признака его отсутствия.
2. Может трактоваться как частный случай типа-суммы (Tagged Union).
3. Предназначен для использования в качестве результата функции.
4. Содержит набор монад для обработки результата.

Functional programming — Optional (2)

Особенности реализации

В качестве основного образца для портирования на C# взят известный Optional из Java 8 и последующих версий.

При реализации использовались возможности современного C#, включая nullability, например:

```
public readonly partial struct Optional<T> : IEquatable<Optional<T>> where T : notnull
  {
      public static readonly Optional<T> Absent;
      private readonly ImmutableWrapper<T>? valueWrapper;
      public bool IsPresent => valueWrapper is object;
      public bool IsAbsent => valueWrapper is null;
      public T Value => valueWrapper ??
          throw new InvalidOperationException("The optional instance does not have a value.");
      public Optional(in T value)
          =>
          valueWrapper = value ?? throw new ArgumentNullException(nameof(value));
```

Functional programming — Optional (3)

Райффайзен

Примеры методов

```
public T OrElse(in T otherValue) => IsPresent ? Value : otherValue;
public T OrElseGet(in Func<T> otherValueFactory) => IsPresent ? Value : otherValueFactory();
public Task<T> OrElseGetAsync(in Func<Task<T>> otherValueFactoryAsync) => IsPresent ?
Task.FromResult(Value) : otherValueFactoryAsync();
public T OrElseThrow(in Func<Exception> exceptionFactory) => IsPresent ? Value : throw
exceptionFactory();
public Optional<TResult> Map<TResult>(in Func<T, TResult> mapper) where TResult : notnull
    =>
    IsPresent switch { true => mapper(Value), _ => Optional<TResult>.Absent };
public Optional<TResult> FlatMap<TResult>(in Func<T, Optional<TResult>> mapper)
    where TResult : notnull
    => IsPresent ? mapper(Value) : Optional<TResult>.Absent;

public Task<Optional<TResult>> FlatMapAsync<TResult>(in Func<T, Task<Optional<TResult>>> mapperAsync)
    where TResult : notnull
    => IsPresent ? mapperAsync(Value) : Task.FromResult(Optional<TResult>.Absent);
public Optional<T> Filter(in Predicate<T> match) => IsPresent && match(Value) ? this : Absent;
```

Functional programming — Optional (4)

Единообразие Optional<T> для классов и структур, с использованием «под капотом» ссылочного тип для упаковки:

```
public sealed class ImmutableWrapper<T>
      where T : notnull
{
      public T Value { get; }

      public ImmutableWrapper(in T value)
          => Value = value ?? throw new ArgumentNullException(nameof(value));
}
```

Functional programming — Optional (5)

Примеры использования Optional<T> при проектировании API:

Optional<TEntity> Find<TEntity>(Guid id) { … }

Optional<TSource> FirstOrAbsent(this IEnumerable<TSource> source, in Func<TSource, bool> predicate)

Functional programming — Result (1)

Случай типа-суммы для предоставления значения в случае успеха операции, либо известной ошибки (подлежащей обработке).

В качестве основного образца взят Result из Kotlin. Также есть реализации Try/Result для Java и Scala.

```
public partial interface IResult<TValue, TError> where TValue : notnull where TError : notnull
{
    bool IsSuccess { get; }
    bool IsFailure { get; }
    protected TValue Value { get; }
    protected TError Error { get; }
}

partial interface IResult<TValue, TError>
{
    TResult Fold<TResult>(in Func<TValue, TResult> onSuccess, in Func<TError, TResult> onFailure)
        where TResult : notnull
        =>
        IsSuccess ? onSuccess(Value) : onFailure(Error);

    Task<TResult> FoldAsync<TResult>(in Func<TValue, Task<TResult>> onSuccessAsync, in Func<TError, Task<TResult>> onFailureAsync)
        where TResult : notnull
        =>
        IsSuccess ? onSuccessAsync(Value) : onFailureAsync(Error);
}
```

Functional programming — Result (2)

Методы конвейера:

```
partial interface IResult<TValue, TError>
{
IResult<TResultValue, TError> PipeResult<TResultValue>(in Func<TValue, IResult<TResultValue, TError>> func)
    where TResultValue : notnull
    =>
    IsSuccess ? func(Value) : Result.Failure(Error).Build<TResultValue>();
Task<IResult<TResultValue, TError>> PipeResultAsync<TResultValue>(in Func<TValue, Task<IResult<TResultValue, TError>>> funcAsync)
    where TResultValue : notnull
    =>
    IsSuccess ? funcAsync(Value) : Task.FromResult(Result.Failure(Error).Build<TResultValue>());
IResult<TResultValue, TResultError> PipeResult<TResultValue, TResultError>(in Func<TValue, IResult<TResultValue, TResultError>> func, in Func<TError, TResultError>
errorMapper)
    where TResultValue : notnull where TResultError : notnull
    =>
    IsSuccess ? func(Value) : Result.Failure(errorMapper(Error)).Build<TResultValue>();
Task<IResult<TResultValue, TResultError>> PipeResultAsync<TResultValue, TResultError>(in Func<TValue, Task<IResult<TResultValue, TResultError>>> funcAsync, in
Func<TError, TResultError> errorMapper)
    where TResultValue : notnull where TResultError : notnull
    =>
    IsSuccess ? funcAsync(Value) : Task.FromResult(Result.Failure(errorMapper(Error)).Build<TResultValue>());
IResult<TResultValue, TResultError> PipeResult<TResultValue, TResultError>(in Func<TValue, IResult<TResultValue, TResultError>> func, in Func<TResultError>
errorFactory)
    where TResultValue : notnull where TResultError : notnull
    =>
    IsSuccess ? func(Value) : Result.Failure(errorFactory()).Build<TResultValue>();
Task<IResult<TResultValue, TResultError>> PipeResultAsync<TResultValue, TResultError>(in Func<TValue, Task<IResult<TResultValue, TResultError>>> funcAsync,
    in Func<TResultError> errorFactory)
    where TResultValue : notnull where TResultError : notnull
    =>
    IsSuccess ? funcAsync(Value) : Task.FromResult(Result.Failure(errorFactory()).Build<TResultValue>());
}
```

Functional programming — Tagged Union

Вспомогательный тип-сумма общего назначения:

```
public sealed class DiscriminatedUnion<TBase, TFirst, TSecond> : IDiscriminatedUnion<TBase> where TBase : notnull where TFirst : notnull, TBase where TSecond :
notnull, TBase
{
    public readonly Optional<TFirst> First;
    private DiscriminatedUnion(in TFirst first)
    {
        First = first;
        presentIndex = 0;
    }
    public static DiscriminatedUnion<TBase, TFirst, TSecond> CreateFirst(in TFirst first)
        =>
        new DiscriminatedUnion<TBase, TFirst, TSecond>(first: first ?? throw new ArgumentNullException(nameof(first)));
    public readonly Optional<TSecond> Second;
    private DiscriminatedUnion(in TSecond second)
    {
        Second = second;
        presentIndex = 1;
    }
    public static DiscriminatedUnion<TBase, TFirst, TSecond> CreateSecond(in TSecond second)
        =>
        new DiscriminatedUnion<TBase, TFirst, TSecond>(second: second ?? throw new ArgumentNullException(nameof(second)));
    int IDiscriminatedUnion<TBase>.Cardinality => 2;
    private readonly int presentIndex;
    int IDiscriminatedUnion<TBase>.PresentIndex => presentIndex;
    Optional<TBase> IDiscriminatedUnion<TBase>.this[in int index] => index switch
    {
        0 => First.Map(value => (TBase)value),
        1 => Second.Map(value => (TBase)value),
        _ => throw new ArgumentOutOfRangeException(nameof(index), index, "The index is out of union cardinality."),
    };
}
```

Functional programming — Unit

Известный в ФП тип Unit — замена System.Void, который недоступен в C# напрямую.
Необходим для превращения Action-ов в функции для их единообразного использования в функциональном стиле.

```
public readonly struct Unit : IEquatable<Unit>
{
    public static readonly Unit Value;
    public bool Equals(Unit other) => true;
    public override bool Equals(object obj) => obj is Unit;
    public override int GetHashCode() => 0;
    public override string ToString() => string.Empty;
#pragma warning disable IDE0060 // Remove unused parameter
    public static Unit FromResult<T>(in T result) => default;
#pragma warning restore IDE0060 // Remove unused parameter
}
```

Functional approach — errors handling

- Делим все ошибки на ожидаемые и неожидаемые.
- Ожидаемые ошибки инкапсулируются в типе Result<TValue, Terror> и «прогоняются» по конвейеру (с возможным маппингом) до получения окончательного результата.
- Неожидаемые ошибки (когда не представляется возможным восстановить состояние/флоу, и неизвестно, как из обрабатывать), являются исключениями — их достаточно залоггировать и позволить приложению «упасть» (в случае сервисной архитектуры еще нужно возвратить код HTTP 500, ASP.NET Core делает это автоматически).
- Таким образом, логика не строится на исключениях. А строится на «прогоне» состояния через конвейер без условных ветвлений.
- В конце конвейера полученное состояние (Value либо Error) схлопывается в значение единого типа (в случае сервисов — HTTP Status code) с помощью операции Fold.

Functional approach — errors reducing

Райффайзен IT

В результате применяемого подхода кардинально сокращается количество ошибок, благодаря тому, что все действия разбиваются на небольшие операции — чистые функции, с помощью которых состояние прогоняется во конвейеру всегда «дальше», в функциональном стиле.

В случае возникновения ошибки она легко локализуется и исправляется на этапе разработки.

Functional approach — flow/pipeline (1)

Пример использование Result, конвейера и Fold, конечная точка (end-point):

```
[HttpPut("updatealive")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status401Unauthorized)]
0 references | 0 changes | 0 authors, 0 changes
public async Task<IActionResult> UpdateAliveAsync([FromBody] UpdateAliveRequestMvcEntity updateAliveRequest)
    =>
    (await sessionManager.TryUpdateAliveAsync(TokenEntityMapper.Map(updateAliveRequest.Token)).ConfigureAwait())
    .Fold<IActionResult>(
        onSuccess: value => Ok(),

        onFailure: error => error.ErrorCode switch
        {
            SessionManagerErrorCode.TokenNotFound => UnauthorizedResult(error.ErrorMessage),

            SessionManagerErrorCode.SessionExpired => UnauthorizedResult(error.ErrorMessage),

            _ => throw new SessionManagerException(error), // Status500InternalServerError
        });
```

Functional approach — flow/pipeline (2)

Пример использование Result, конвейера и Fold, модель (model layer):

```
public Task<IResult<Unit, SessionManagerError>> TryUpdateAliveAsync(
    ITokenDto token, TimeSpan minRemainingLifeTime)
    =>
    Task.FromResult(
        InternalTryGetSession(token)
        .PipeResult(session => session.UpdateAlive(minRemainingLifeTime)));


private IResult<ISession, SessionManagerError> InternalTryGetSession(in ITokenDto token)
    =>
    Sessions.TryGetValue(TokenMapper.Map(token), out var session) ?
    Result.Success<ISession>(session).Build<SessionManagerError>() :
    Result.Failure(new SessionManagerError(SessionManagerErrorCode.TokenNotFound)).Build<ISession>();
```

Functional approach — AsyncPipeline

- Асинхронный конвейер будет представлен в одном из следующих докладов
- Представляет реализацию функционального конвейера на базе асинхронных методов .NET

The Results

Райффайзен

- Продукт на базе микросервисов с использованием разработанного функционального фреймворка работает в Production
- NullReferenceException оказались исключены на всех этапах — от тестирования до работы в Production. Минимизировано количество других «технических» исключений
- Разработанный фреймворк готов к размещению в OpenSource
  https://github.com/Raiffeisen-DGTL/ViennaNET

Спасибо за внимание

Райффайзен