Тонкости асинхронного программирования в .net core

Плохие и хорошие примеры на практике

Меня зовут Александр! Я работаю в компании Европлан Fullstack разработчиком. Занимаюсь разработкой на.net более 8 лет. Долгое время разрабатывал решения на winforms/wpf/wcf/aspmvc, в последние годы ушел в веб разработку на angular(react)/.net core/postgre. В свободное время исследую кросплатформеные решения, например за capacitor от команды ionic, пушу на github в opensource для yandex карт.

async/await

- C# 5.0 2012 год
- ECMA2017 после 2015
- Swift 4.2. 2018 год.
- ……

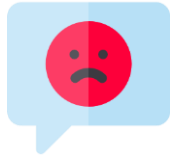
Делайте стек вызовов асинхронным

```
public int DoSomethingAsync()
{
  var result = CallAsync().Result;
   return result + 1;
}
```

Делайте стек вызовов асинхронным

```
public async Task<int> DoAsync()
{
    var result = await CallAsync();
    return result + 1;
}
```

Async void



```
public IActionResult Post() {
  OperationAsync();
  return Accepted();
}
public async void OperationAsync(){
  var result = await CallDependencyAsync();
  DoSomething(result);
}
```

Async void

```
public IActionResult Post() {
        _ = BackgroundOperationAsync();
        return Accepted();
}

public async Task BackgroundOperationAsync(){
    var result = await CallDependencyAsync();
     DoSomething(result);
}
```

Async void для Winforms

```
async void button1_Click(object sender, EventArgs e)
{
    this.button1.Enabled = false;
    var someTask = Task<int>.Factory.StartNew(() => slowFunc(1, 2));
    await someTask;
    this.label1.Text = "Result: " + someTask.Result.ToString();
    this.button1.Enabled = true;
}
```

Task.FromResult для простых операций

```
public class MyLibrary {

   public Task<int> AddAsync(int a, int b){

      return Task.Run(() => a + b);

   }

}
```

Task.FromResult синхронных операций возвращающих Task

```
public class MyLibrary {
    public Task<int> AddAsync(int a, int b) {
        return Task.FromResult(a + b);
    }
}
```

Избегайте Task.Run для долгих операций

```
private readonly BlockingCollection<Message> _msgQueue;

public void StartProcessing()
{
      Task.Run(ProcessQueue);
}
```

Используете выделенный поток для долгих операций

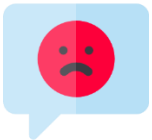
```
public class QueueProcessor
{
    public void StartProcessing()
      {
        var thread = new Thread(ProcessQueue)
        {
            IsBackground = true
        };
        thread.Start();
       }

}
```

Task.Run для долгих операций с TaskCreationOptions.LongRunning

```
public void StartProcessing()
{
    Task.Factory.StartNew(ProcessQueue,TaskCreationOptions.LongRunning);
}
```

Создавайте TaskCompletionSource<T> с TaskCreationOptions.RunContinuationsAsynchronously

```
public Task<int> DoSomethingAsync() {
    var tcs = new TaskCompletionSource<int>();
    var operation = new LegacyAsyncOperation();
    operation.Completed += result => {
        tcs.SetResult(result);
    };
    return tcs.Task;
}
```



Создавайте TaskCompletionSource<T> с TaskCreationOptions.RunContinuatiuonsAsynchronously

```
public Task<int> DoSomethingAsync(){

    tcs = new TaskCompletionSource<int>(TaskCreationOptions...);

    var operation = new LegacyAsyncOperation();

        operation.Completed += result => {tcs.SetResult(result);

    };

    return tcs.Task;

}
```

TaskCreationOptions.RunContinuationsAsynchronously

- Код выполнится в новом потоке
- Асинхронно дожидаемся событие
- Дожидаемся завершение Task

Используйте Dispose CancellationTokenSource с таймаутами

```
Task<Stream> HttpClientAsyncWithCancellationBad(){
    var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));
    using (var client = _httpClientFactory.CreateClient())
    {
        var response = await client.GetAsync("http…", cts.Token);
        return await response.Content.ReadAsStreamAsync();
    }
}
```

Используйте Dispose CancellationTokenSource с таймаутами

```
Task<Stream> HttpClientAsyncWithCancellationGood()
{
    using (var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10))
    {
        using (var client = _httpClientFactory.CreateClient())
        {
            var response = await client.GetAsync("http…", cts.Token);
            return await response.Content.ReadAsStreamAsync();
        }
    }
```

Используйте CancellationToken(s) с APIs принимающими CancellationToken

```
Task<string> DoAsyncThing(CancellationToken cancellationToken = default)
{
        byte[] buffer = new byte[1024];

        // We forgot to pass flow cancellationToken to ReadAsync
        int read = await _stream.ReadAsync(buffer, 0, buffer.Length);

        return Encoding.UTF8.GetString(buffer, 0, read);
}
```

Используйте CancellationToken(s) с APIs принимающие CancellationToken

```
Task<string> DoAsyncThing(CancellationToken token = default)

{

    byte[] buffer = new byte[1024];

    int r = await _stream.ReadAsync(buffer,0,buffer.Length, token);

    return Encoding.UTF8.GetString(buffer, 0, read);

}
```

Всегда вызывайте FlushAsync при StreamWriter(s) при Dispose

```
app.Run(async context =>
{
    using (var streamWriter = new
      StreamWriter(context.Response.Body))
    {
        await streamWriter.WriteAsync("Hello World");
    }
});
```

Всегда вызывайте FlushAsync при StreamWriter(s) при Dispose

```
app.Run(async context =>
{
    using (var streamWriter = new
      StreamWriter(context.Response.Body))
    {
        await streamWriter.WriteAsync("Hello World");
        // Force an asynchronous flush
        await streamWriter.FlushAsync();
    }
});
```

Знайте разницу с await и без

```
public Task<int> DoSomethingAsync()
{
    return CallDependencyAsync();
}
```

Знайте разницу с await и без

```
public async Task<int> DoSomethingAsync()
{
    return await CallDependencyAsync();
}
```

Знайте разницу с await и без

- Оверхед по StateMachine, проброс Task.
- Места создания exception в разных местах

ConcurrentDictionary.GetOrAdd

```
public class PersonController : Controller{

 public IActionResult Get(int id){
     var person = _cache.GetOrAdd(id, (key) =>
                    db.People.FindAsync(key).Result);
     return Ok(person);
 }
}
```

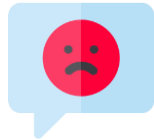
ConcurrentDictionary.GetOrAdd

```
public class PersonController : Controller{
    private AppDbContext _db;

    public async Task<IActionResult> Get(int id){
        var person = await _cache.GetOrAdd(id, (key) =>
                        db.People.FindAsync(key));
        return Ok(person);
    }
}
```

ConcurrentDictionary.GetOrAdd

- Структура внутри лочит на время добавления записи лочит на время операции
- Таск положить быстрей чем результат

Конструкторы синхронны



```
public class Service : IService
{
    private readonly IRemoteConnection _connection;

    public Service(IRemoteConnectionFactory connectionFactory)
    {
        _connection = connectionFactory.ConnectAsync().Result;
    }
}
```

Конструкторы синхронны

```
private readonly IRemoteConnection _connection;


static async Task<Service> CreateAsync(IRemoteConnectionFactory
conFactory)
    {
        return new Service(await conFactory.ConnectAsync());
    }
}
```

Конструкторы синхронны

- Все минусы синхронного кода в конструкторе применимы
- Уходим от синхронной асинхронности

ASP.NET Core

All IO in ASP.NET Core is asynchronous.

Избегайте использования синхронных перегрузок

```
public class MyController : Controller
{
    [HttpGet("/pokemon")]
    public ActionResult<PokemonData> Get()
    {
        var json = new StreamReader(Request.Body).ReadToEnd();

        return JsonConvert.DeserializeObject<PokemonData>(json);
    }
}
```

Использование асинхронных перегрузок чтения / записи.

```
public class MyController : Controller
{
     [HttpGet("/pokemon")]
     public async Task<ActionResult<PokemonData>> Get()
     {
         var json = await new StreamReader(Request.Body).ReadToEndAsync();

         return JsonConvert.DeserializeObject<PokemonData>(json);
     }
}
```

Использование синхронных перегрузок HttpRequest.ReadAsForm

```
public class MyController : Controller
{
    [HttpGet("/form-body")]
    public IActionResult Post()
    {
        var form = HttpRequest.Form;
        Process(form["id"], form["name"]);
        return Accepted();
    }
}
```

HttpRequest.ReadAsFormAsync()

```
public class MyController : Controller
{
    [HttpGet("/form-body")]
    public async Task<IActionResult> Post()
    {
        var form = await HttpRequest.ReadAsFormAsync();
         Process(form["id"], form["name"]);
        return Accepted();
    }
}
```

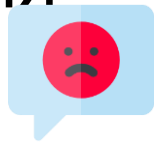
Не работайте с непотокобезопасным кодом в нескольких потоках, например HttpContext.

```
public async Task<SearchResults> Get(string query)
{
        var query1 = SearchAsync(SearchEngine.Google, query);
        var query2 = SearchAsync(SearchEngine.Bing, query);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query);
        await Task.WhenAll(query1, query2, query3);
        var results1 = query1.Result;
        return SearchResults.Combine(results1, results2, results3);
 }
SearchAsync()  {  ... HttpContext.Request.Path;}
```

Не работайте с непотокобезопасным кодом в нескольких потоках, например HttpContext.

```
public async Task<SearchResults> Get(string query)
{
    string path = HttpContext.Request.Path;
    var query1 = SearchAsync(SearchEngine.Google, query, path);
    var query2 = SearchAsync(SearchEngine.Bing, query, path);
    await Task.WhenAll(query1, query2);
   var results1 = query1.Result;
    var results2 = query2.Result;
    return SearchResults.Combine(results1, results2, results3);
}
```

Не используйте HttpContext после завершения запроса



```
public class AsyncVoidController : Controller
{
    [HttpGet("/async")]
    public async void Get()
    {
        await Task.Delay(1000);
        await Response.WriteAsync("Hello World");
    }
}
```

Не используйте HttpContext после завершения запроса

```
public class AsyncController : Controller
{
    [HttpGet("/async")]
    public async Task Get()
    {
        await Task.Delay(1000);
        await Response.WriteAsync("Hello World");
    }
}
```

Не захватывайте сервисы в фоновых потоках

```
public IActionResult
FireAndForget1([FromServices]PokemonDbContext context)
{
    Task.Run(() =>
    {
        await Task.Delay(1000);
        context.Pokemon.Add(new Pokemon());
        await context.SaveChangesAsync();
    });
}
```

Не захватывайте сервисы в фоновых потоках

```
public IActionResult FireAndForget3(){
    Task.Run(async () =>{
        await Task.Delay(1000);
        using(var ctx = ConnectionFactory.CreateDbContext()) {
            ctx.Pokemon.Add(new Pokemon());
            await ctx.SaveChangesAsync();
        }
    });
    return Accepted();
}
```

Не захватывайте сервисы в фоновых потоках

- ObjectDisposedException

Не добавляйте headers после запуска HttpResponse

```
app.Use(async (next, context) =>
{
    await context.Response.WriteAsync("Hello ");
    await next();
    // This may fail if next() already wrote to the response
    context.Response.Headers["test"] = "value";
});
```

Не добавляйте headers после запуска HttpResponse

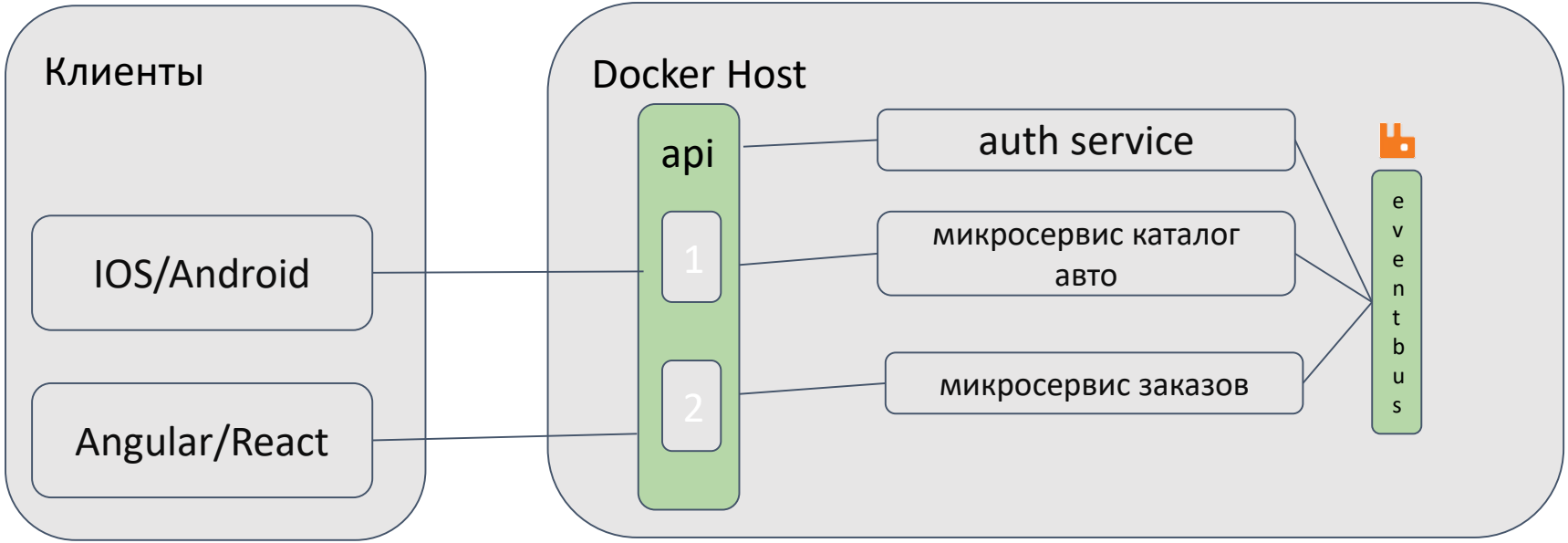
```
app.Use(async (next, context) =>
{
    await context.Response.WriteAsync("Hello ");
    await next();
    if (!context.Response.HasStarted)
    {
        context.Response.Headers["test"] = "value";
    }
});
```

Европлан практика

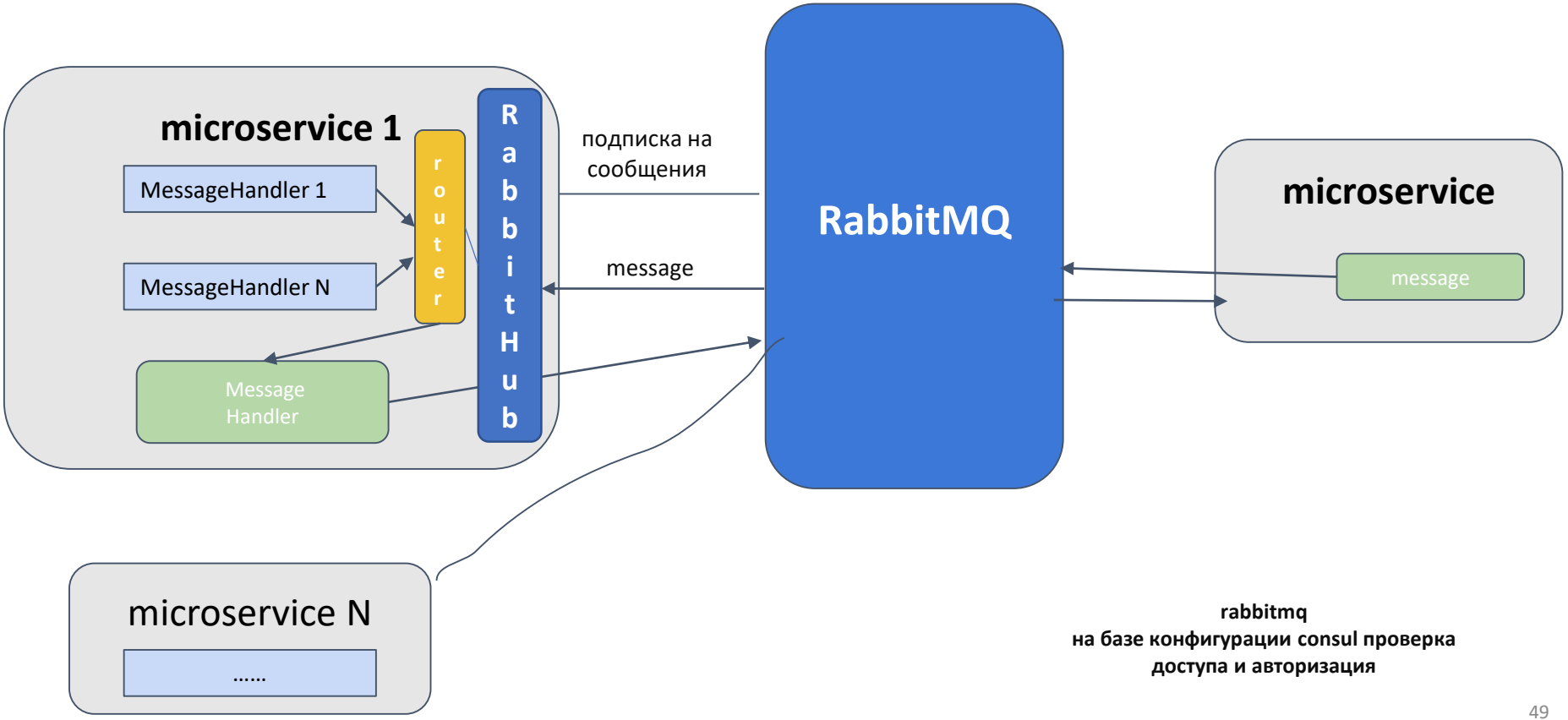
async await

микросервисы

Несколько команд работающие над разными частями приложения.
Новые члены включаются быстро в работу
Просто менять бизнес правила
CI CD
Новые языки и платформы.



Практика в микросервисах



rabbitmq
на базе конфигурации consul проверка доступа и авторизация

async await messageHandler

```
async Task<MessageProcessResult> MessageHandler(DeliveredMessage dm)
{
    var topic = dm.GetTopic();
    ...
    try
    {
           if (this.handlers.TryGetValue(topic, out var handler) ) {

               // могло возникнуть синхронное поведение через асинхронное
               result = await handler.Execute(dm).ForAwait();
            }
    …..
    }
    ...
}
```

async await messageHandler

```
async Task<MessageProcessResult> MessageHandler(DeliveredMessage
dm)
{
    var topic = dm.GetTopic();
    ...
    try
    {
           if (this.handlers.TryGetValue(topic, out var handler) )
{

        //  ThreadPool Starvation
        await Task.Run(() =>  handler.Execute(dm)).ForAwait();

    }
    …..
    }
    ...
```

async await messageHandler

```
async Task<MessageProcessResult> MessageHandler(DeliveredMessage dm)
{
    var topic = dm.GetTopic();
    ...
    try
    {
        if (handler.RunConcurrent)
        {
            result = await Task.Run(()=>handler.Execute(dm)).ForAwait();
        }
        else
        {
            result = await handler.Execute(dm).ForAwait();
        }
    }
    ...
}
```

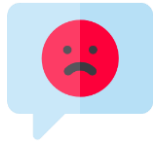
ConfigureAwait

handler.Execute(dm)).ForAwait();

ForAwait(this Task task) => task**.ConfigureAwait(false);**

- Избегаем deadlock в легаси.
- Снижаем SynchronizationContext расходы на сохр/восстановление.

Минусы



RunConcurrent = true

- ThreadPool конечен. Память.Pool starvation
- handler в отдельном потоке

Плюсы

Ставим **RunConcurrent = false**

- Экономим память
- Повышение масштабируемости
- MessageHandler работает асинхронно
- Пишем асинхронный callstack хендлера
- Экономим ThreadPool, один поток может обслужить N задач

Спасибо за внимание!