ZASTRPAY STUDIOS

F# in the real world

Sharing experience about
how F# simplifies software (development)

Deyan Petrov, Konstantin Ivanov, Andriy Horen

F# Vienna Meetups:

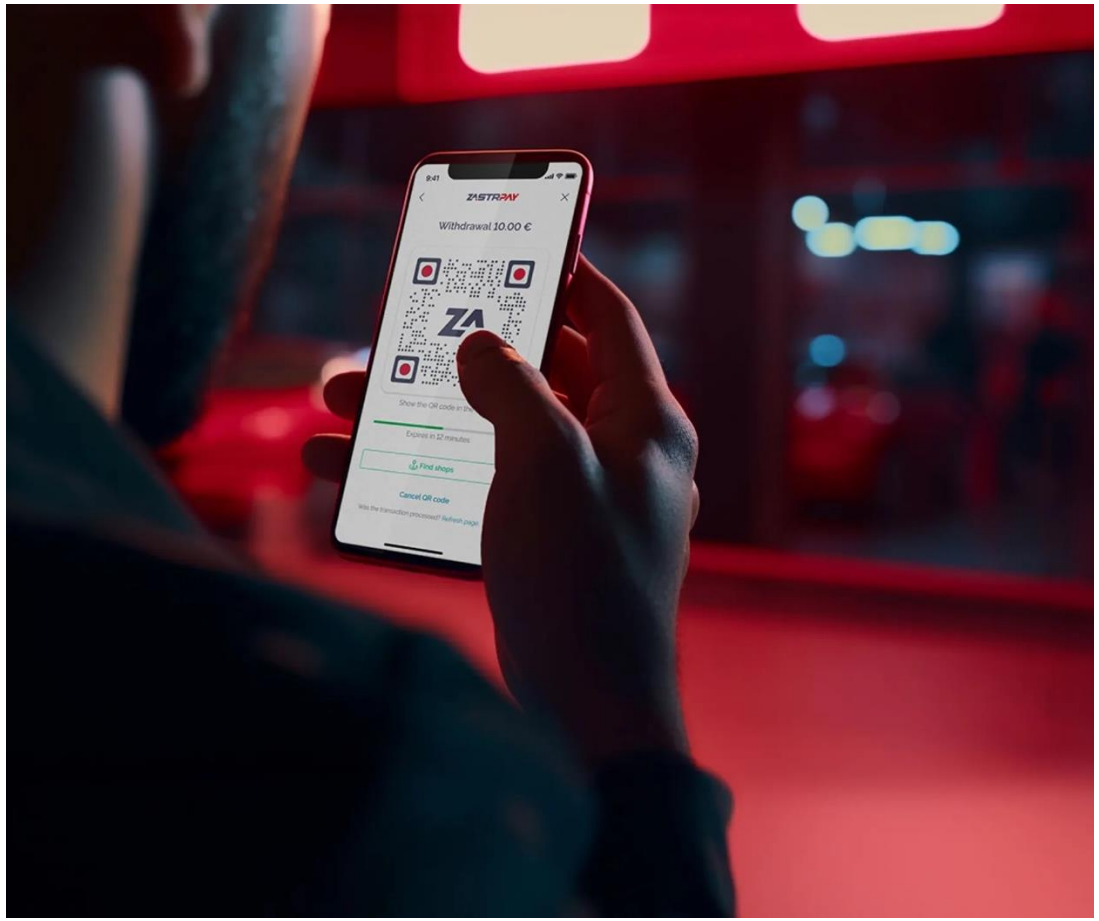
- Part 1+2: 27 Jan 2025
- Part 3+4: 04 Mar 2025

Introduction

- **Zastrpay Studios** (started in 2019 in Vienna, originally 5G Pay)

- We are the technical provider of the Zastrpay cash digital payment product

  www.zastrpay.com



How it started with F# ;)

- Choice of Tech Stack was one of the conditions for joining the company as "CTO" ... guess what was chosen ;)
- Question to **F# Vienna Meetup** organizers upon starting this project:

March 26th, 2019

Hi [redacted] I am thinking of starting a new project (Azure or AWS cloud-based financial system) in F# and wondering if I would be able to find developers on the local market (Vienna) willing to do F#. What is your opinion on that? Thanks and best regards, Deyan Petrov

15:35

F#?

r/csharp • 13 days ago
ruben_vanwyk

Would you take an F# job?

Hi everyone. Was wondering if any senior engineers would consider taking a job in F# if presented, especially since it's similar to LINQ in functionality and the whole dotNET is available to you?

alien3d • 13d ago

never saw one 🤣 . if good why not .

Glum_Past_1934 • 13d ago

Never saw one

xabrol • 13d ago

F# is better than c# imo, so yeah. It's like typescript but it compiles and typescript is my favorite.

WomenRepulsor • 13d ago

I never knew something was actually coded in F#

Yelmak • 13d ago

I would if there were any going, functional programming is cool and I've never had a good excuse to really use it

Rogermcfarley • 13d ago

There are F# jobs? When did this happen?

darnold992000 • 13d ago

In a heartbeat.

Team and Organizational Principles

- Small dev team which grew from 1 to 8+ developers in the past 5 years
  - All former C# developers, re-trained in F# upon onboarding
- End-to-end responsibility
- No sys admins, no testers, no project managers, no business analysts, product managers as individual contributors, etc.
- Strong focus on technology
- A list of additional principles can be found at https://5gpay.com/manifest

Zastrpay Studios Tech Stack

Frontend

- React
- TypeScript
- React Native
- Expo
- nx

Backend

- F#
- .NET

Azure DevOps

- Backlogs
- Pipelines
- Repos

Infrastructure

- AKS
- Azure Container Registry
- træfik
- MongoDB
- Azure Data Explorer
- Application Insights
- Azure Event Hubs
- Azure Storage Accounts
- Azure Key Vault
- Azure CDN
- Azure SignalR
- Linux VMs

AKS for low-cost and scalable hosting with very high pod density, high availability. Traefik as ingress proxy

MongoDB Atlas for a highly available OLTP database with little maintenance effort

Azure Data Explorer as very fast (column store) and cheap data warehouse/reporting/read model database

Azure Event Hubs (low-cost, scalable) as a pub-sub message bus

High test automation with BDD and CDCT

Multiple low-risk deployments per day (Azure DevOps pipelines)

Microservice, event-driven architecture for simplicity and ease of deployment

Infrastructure / Operation costs about 1 sysadmin salary

Azure Cloud hosting for fast setup

Backend is 100% F#

.sln files **36**

.fsproj files **333**

.fs files **4294**

.fsx files **189**

Total Lines **464k**

Lines CODE **381k**

Lines AVG per file **108**

Part 1: How do we use F# in detail?

1. No OOP, using only the functional aspects of F#
2. File and module ordering is great (bottom-up)
3. Pipelines are awesome
4. Separation pure from impure code is important
5. Async, Result and AsyncResult CEs only
6. Railway-Oriented Programming
7. Function composition using map and bind
8. DI with partial application
9. Wrap .NET BCL and SDK in helper infrastructure functions
10. Few type aliases, raw function signatures
11. Measures with UMX
12. Dto mapping, serialization, exhaustive mapping
13. Anonymous Records for strongly-typed DB queries
14. Function names can be English sentences
15. F# Tooling

No OOP, using only the functional aspects of F#

- DUs, Record Types and Functions – that is all one needs ;)

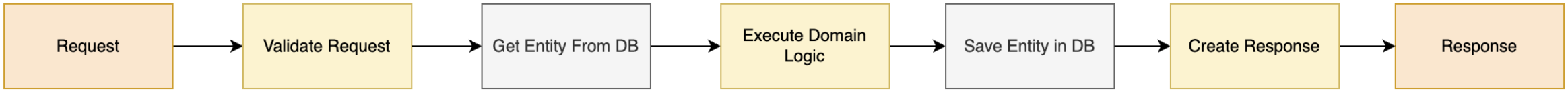
```fsharp
open System

type Order(orderId: Guid) =
    let mutable status: string = "Pending"
    let createdOn: DateTime = DateTime.Now

    member this.OrderId: Guid = orderId
    member this.CreatedOn: DateTime = createdOn
    member this.Status: string
        with get() = status
        and set(value: string) = status <- value

    member this.Create(): unit =
        printfn $"Order {this.OrderId} created on {this.CreatedOn}."

    member this.Pay(): unit =
        this.Status <- "Paid"
        printfn $"Order {this.OrderId} has been paid."

let order: Order = Order(Guid.NewGuid())
order.Create()
order.Pay()
```

```fsharp
open System

type Order = {
    OrderId: Guid
    CreatedOn: DateTime
    Status: string
}

module Order =
    let create (orderId: Guid) : Order =
        printfn $"Order {orderId} created on {DateTime.UtcNow}"
        {
            OrderId = orderId
            CreatedOn = DateTime.UtcNow
            Status = "Pending"
        }

    let pay (order: Order) : Order =
        printfn $"Order {order.OrderId} has been paid."
        {
            order with Status = "Paid"
        }

let order: Order = Order.create (Guid.NewGuid())
let paidOrder: Order = Order.pay order
```

For inter

No OOP, using only the functional aspects of F#

- Processing of an HTTP request consist of a bunch of functions chained together/composed into a pipeline (no need for OOP)



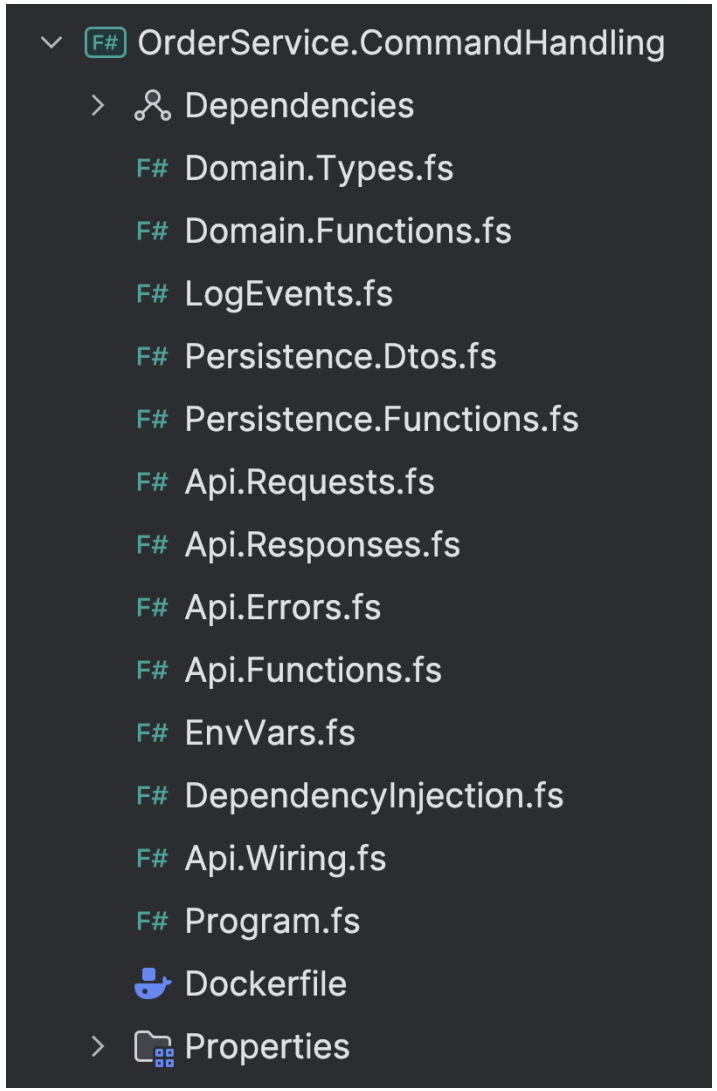
- Clear separation between data and functions
  - no need to wonder where to put some logic/operations – in a separate "Manager" class, or directly in the Domain/Entity class ;)
  - Functions related to a type are grouped into a module with the same name
  - High-order functions, "functions for all design patterns in FP"*
- No type extensions, simple modules with functions instead

```
type DateTime with
    member xs.ToStringIso() : string =
        xs.ToString("yyyy-MM-ddTHH:mm:ss.fffZ")
```

```
module DateTime =
    let toStringIso (dt: DateTime) : string =
        dt.ToString("yyyy-MM-ddTHH:mm:ss.fffZ")
```

File and module ordering is great (bottom-up)

- Pure Domain types and functions on the top, no dependency on nothing
- Persistence depends on Domain
- Orchestration (Api.Functions) depends on Domain & Persistence, etc.
- Caveat: could still depend on a Project Reference (outside of the project) …
- Inside a file, you know that the stuff on the bottom depends on the stuff on the top
- Code reviews are greatly helped by this strict order!



Pipelines are awesome

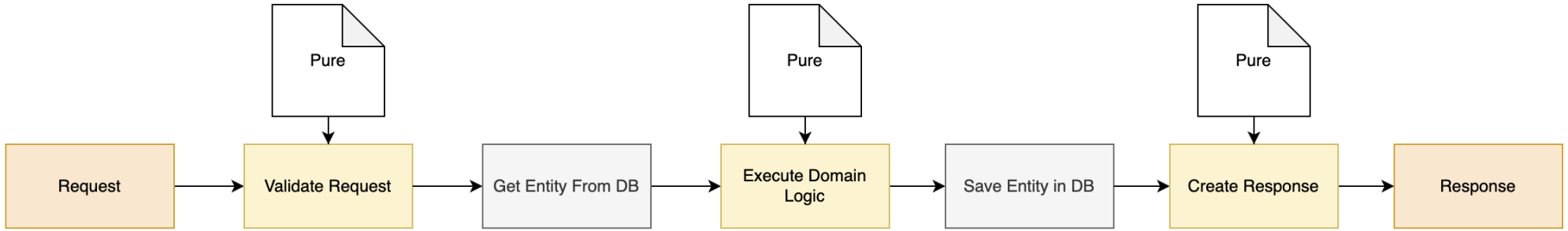
- Not much to add ;)

```
asyncResult {
    // parse and validate command
    let! cmd =
        req : MappedHttpRequest
        |> CreateOrderCommandDto.fromHttp : Result<CreateOrderCommandDto,DeserializationError>
        |> Result.mapError CreateOrderCommandError.RequestBodyDeserializationFailed : Result<
        |> Result.bind (
            CreateOrderCommandDto.toDomain req
            >> Result.mapError CreateOrderCommandError.RequestValidationError
        ) : Result<CreateOrderCommand,CreateOrderCommandError>
        |> AsyncResult.ofResult : Async<Result<CreateOrderCommand,CreateOrder
```

```
do!
    changes : ChangeStreamDocument<TransactionDto> list
    ▷ Seq.collect _.FullDocument.Entries : AccountingEntryDto seq
    ▷ Seq.filter (fun accEntry → accEntry.DelayedBooking ▷ Option.defaultValue false) : AccountingEntryDto seq
    ▷ Seq.groupBy (fun accEntry → accEntry.AccountId, accEntry.Currency) : ((Guid * CurrencyDto) * AccountingEntryDto seq) seq
    ▷ Seq.map (fun ((accountId, currency), accEntries) →
        (accountId, currency),
        accEntries : AccountingEntryDto seq
        ▷ Seq.sumBy (fun e → if e.Type = AccountingEntryTypeDto.Debit then -1M * e.Amount else e.Amount) : decimal )
    ▷ Seq.map (fun ((accountId, currency), deltaAmount) →
        doFlushAccountBalanceChange
            log
            updateAccountBalance
            findAccountById // propagate function dependencies to private helper
            accountId
            deltaAmount
            currency
            changeTimestamp
            trxId) : Async<unit> seq
    ▷ Array.ofSeq : Async<unit> array
    ▷ Async.Parallel : Async<unit array>
    ▷ Async.Ignore : Async<unit>  // it's just an array of units
```

Separation pure from impure code is important

- Pure/Impure/Pure/Impure/Pure sandwich



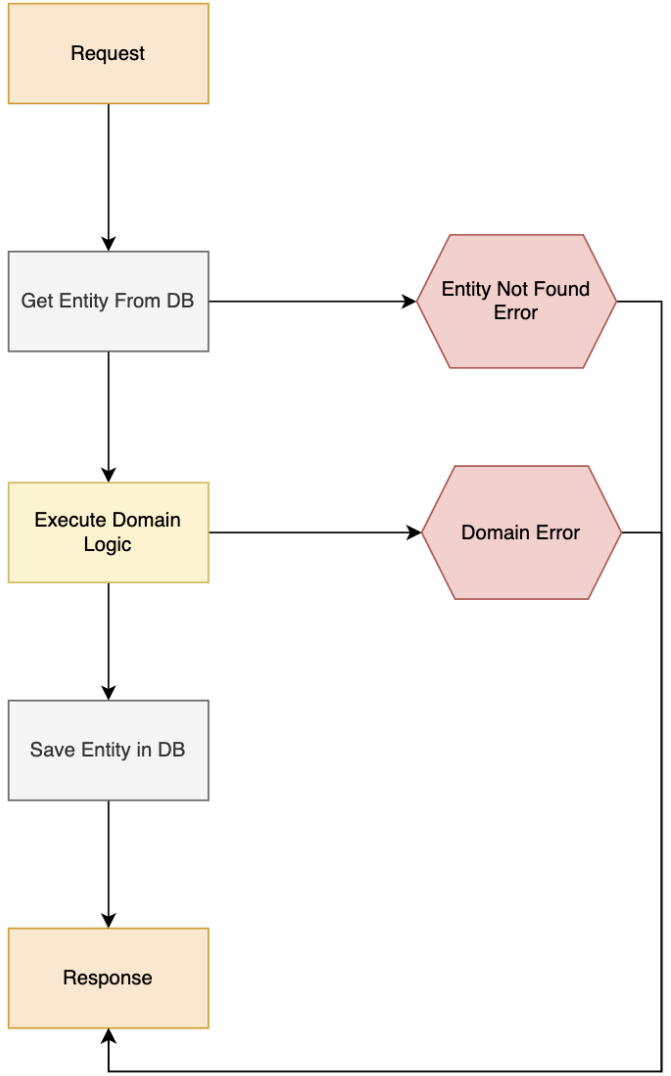
- The more deeply nested function call – the more problems separating pure from impure code
  - and vice versa – flat orchestration code helps immensely!
- It does occur though that pure code is included in an impure (e.g. returning Async) function, instead of being extracted
  - Functional thinking seems to be an afterthought

Async, Result and AsyncResult CEs only

- CEs are required to bring code to the left
  - vs. going more and more to the right (“Pyramid of Doom")
  - or having a very long pipeline with Async.bind/map, then AsyncResult.bind/map till the end …
- Only 3 CEs are really required
  - OK, you caught me, there is a 4th one – ResultA ;)
- The central orchestration code usually uses asyncResult
  - Everything else inside gets "elevated" to asyncResult as the common denominator
- Pure functions use at most Result/ResultA
- If a function uses async or asyncResult then it is impure

Railway-Oriented Programming

- **Exception** = something really unexpected, e.g. DB down, network error, OOM, missing or not parseable configuration (under our control), etc.
  - Usually results in 500 http status code (unless there is some retry mechanism)
- **Error** = something that is expected, can still happen if all infra is working, for example user input validation errors, business rules/invariants
  - Usually results in 400 http status code
  - Also in line with "Do not use exceptions for control flow" recommendation
- The decision is context-specific
  - If microservice1 calls microservice2 and gets 400, then this can be still treated as an exception in the context of microservice1 (i.e. from its perspective this should never happen!)
- Extensions like bindError, bindExn to switch from error to happy path
- Bottom-up **aggregation of errors** from inner to calling functions
  - every inner/simple function only returns error code relevant to it without knowing the caller!
- FsToolkit.ErrorHandling used
  - but for 3+ years initially used directly Scott Wlaschins source code



Function composition using map and bind

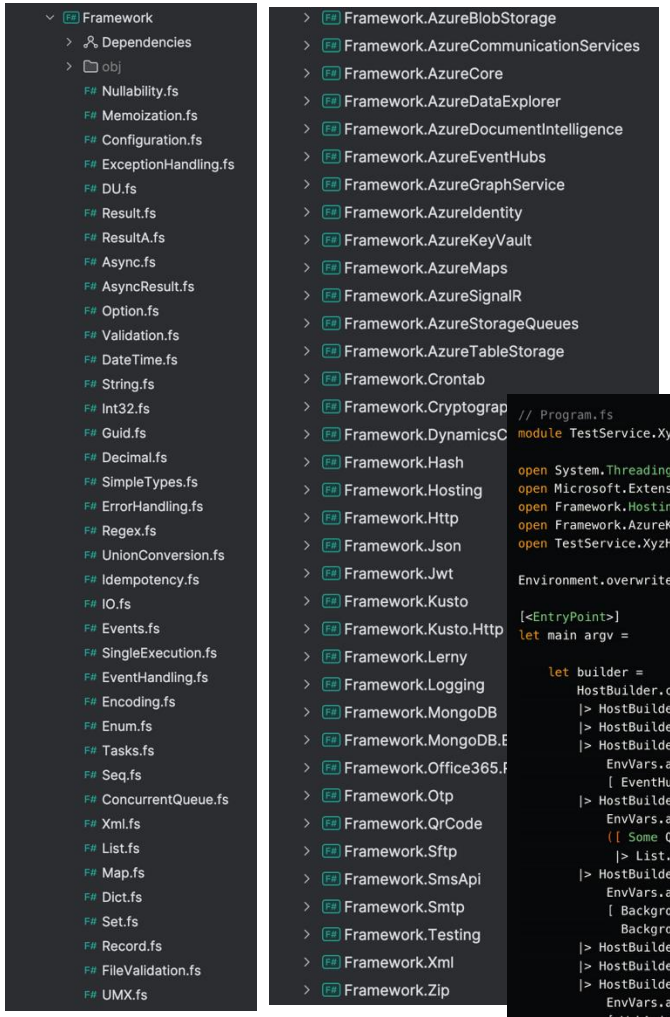
- Functors, applicatives and monads (as part of the view "functional design patterns") understanding is not critical for writing code
- What needs to be fundamentally understood though is:
  - It is all about function composition/chaining, how to make function signatures compatible
  - Writing simple functions, with the simplest possible input/output arguments
- In reality map and bind work on functions - return functions with elevated input/output
- Sometimes map is wrongly used, and you need to call flatten => replace with bind

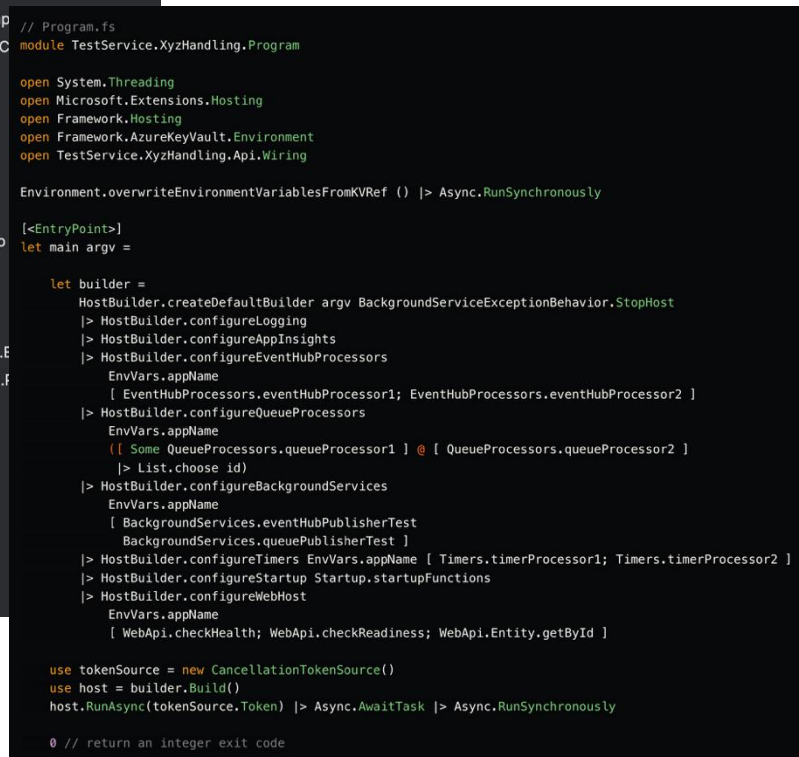
DI with partial application

- Every function has its own dependencies
  - No shared dependencies record across different functions ~ ctor injection in C#
- Dependencies = functions
  - can be easily mocked if the function signature is the same
- DependencyInjection.fs is responsible for exposing functions (from Api.Function.fs) with already partially applied arguments
- The caller is oblivious to the above and supplies only the remaining arguments – e.g. a http message
- Top-level composition root applies dependencies also to "inner" functions
  - bottom-up, participating functions do not know each other's dependencies
  - See https://fsharpforfunandprofit.com/posts/dependency-injection-1/#passing-dependencies-to-inner-functions for a detailed explanation
  - Very important to have flat orchestration functions - then only 1 function must be dependency-injected, no inner

Wrap .NET BCL and SDK in helper infrastructure functions

- Including mapping "expected" Exceptions to Results
- Allowing for partial application of static "clients" which are instantiated upon application startup
- Allowing for passing in mocked inner functions – e.g. make an http call
- Such wrappers are usually small – 100 LOCs
- F# App Stub for AKS hosting - wrappers around BackgroundService providing easy spawning of multiple jobs





```fsharp
// Program.fs
module TestService.XyzHandling.Program

open System.Threading
open Microsoft.Extensions.Hosting
open Framework.Hosting
open Framework.AzureKeyVault.Environment
open TestService.XyzHandling.Api.Wiring

Environment.overwriteEnvironmentVariablesFromKVRef () |> Async.RunSynchronously

[<EntryPoint>]
let main argv =

    let builder =
        HostBuilder.createDefaultBuilder argv BackgroundServiceExceptionBehavior.StopHost
        |> HostBuilder.configureLogging
        |> HostBuilder.configureAppInsights
        |> HostBuilder.configureEventHubProcessors
            EnvVars.appName
            [ EventHubProcessors.eventHubProcessor1; EventHubProcessors.eventHubProcessor2 ]
        |> HostBuilder.configureQueueProcessors
            EnvVars.appName
            ([ Some QueueProcessors.queueProcessor1 ] @ [ QueueProcessors.queueProcessor2 ]
             |> List.choose id)
        |> HostBuilder.configureBackgroundServices
            EnvVars.appName
            [ BackgroundServices.eventHubPublisherTest
              BackgroundServices.queuePublisherTest ]
        |> HostBuilder.configureTimers EnvVars.appName [ Timers.timerProcessor1; Timers.timerProcessor2 ]
        |> HostBuilder.configureStartup Startup.startupFunctions
        |> HostBuilder.configureWebHost
            EnvVars.appName
            [ WebApi.checkHealth; WebApi.checkReadiness; WebApi.Entity.getById ]

    use tokenSource = new CancellationTokenSource()
    use host = builder.Build()
    host.RunAsync(tokenSource.Token) |> Async.AwaitTask |> Async.RunSynchronously

    0 // return an integer exit code
```

Few type aliases, raw function signatures

- Scott Wlaschin was suggesting to define type aliases for functions, however using the original function signatures seems to be eliminate 1 lookup

```
type LogInfo = string -> unit
type LogError = string -> unit

type DbGetEmail = UserId -> EmailAddress
type DbUpdateProfile = UserId * UserName * EmailAddress -> unit
type Notify = EmailAddress * EmailAddress -> unit
```

```
module CustomerUpdater =

  let updateCustomerProfile
    (logInfo:LogInfo)
    (logError:LogError)
    (getEmail:DbGetEmail)
    (updateProfile:DbUpdateProfile)
    (notify:Notify)
    (json: string) =
    try
      let request = parseRequest(json)
      let currentEmail = getEmail(request.UserId)
      updateProfile(request.UserId, request.Name, request.EmailAddr
      logInfo("Updated Profile")
    // etc
```

```
Yevgen Cherkasenko +1
module GenericEntity =
    Yevgen Cherkasenko +1
    let create
        (insertGenericEntity: GenericEntity -> AsyncResult<GenericEntity, InsertEntityError>) //function dependency
        (findGenericEntityById: Guid -> Async<GenericEntity>) //function dependency
        (req: MappedHttpRequest) // input
        : AsyncResult<MappedHttpResponse, MappedHttpErrorWithLogInfo<_>>
        =
        asyncResult {
            // parse and validate command
```

Measures with UMX

- FSharp.UMX is an amazing little library which **extends** F# measures to other data types (besides numerics) like Guid, string, DateTime, etc.
- It allows us to make some primitive/general types more specific
  - Prevents errors mixing up different guids or strings!
  - Makes function signatures more readable

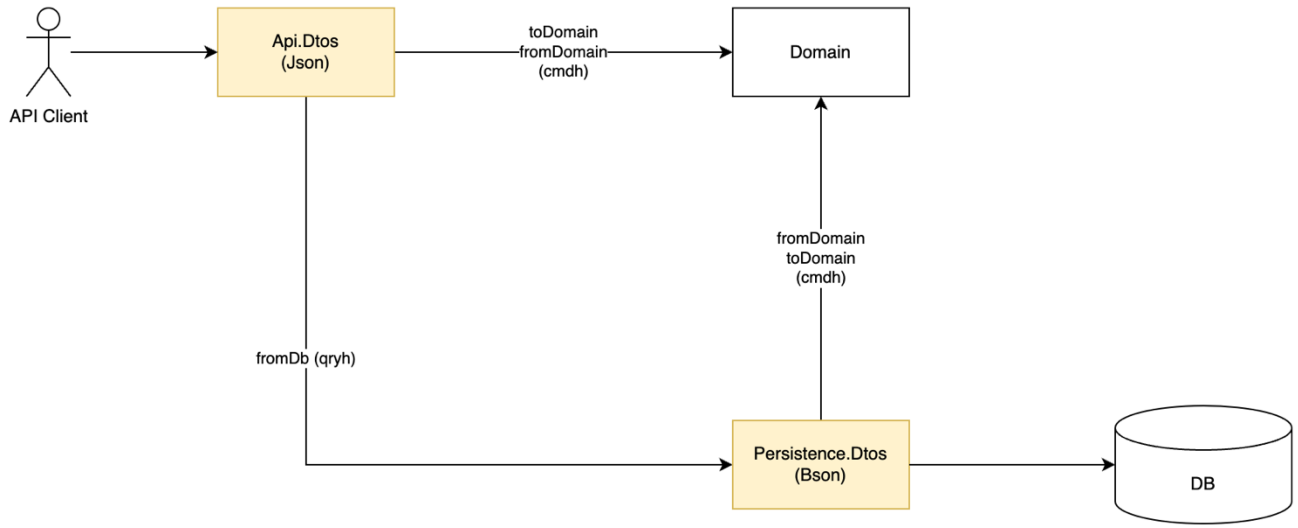
```
[<Measure>]
Yevgen Cherkasenko
type GenericEntityEventId
```

```
Yevgen Cherkasenko
type GenericEntity =
    {
        Id: Guid<GenericEntityId>

        Type: GenericEntityType
        State: GenericEntityState

        CreatedOn: DateTime<CreatedOn>
        CreatedBy: TriggerSource
        LastModifiedOn: DateTime<LastModifiedOn>
        LastModifiedBy: TriggerSource
        LastEvent: GenericEntityEvent
        LastCorrelationId: Guid<LastCorrelationId>
    }
```

```
Yevgen Cherkasenko +1
let activate
    (findGenericEntityById: Guid<GenericEntityId> -> Async<Result<GenericEntity, EntityNotFoundByIdError>>) //function dependency
    (replaceGenericEntity: GenericEntity -> AsyncResult<GenericEntity, UpdateEntityError>) //function dependency
    (req: MappedHttpRequest) // input
    : AsyncResult<MappedHttpResponse, MappedHttpErrorWithLogInfo<_>>
    =
    asyncResult {
```

Dto mapping, serialization, exhaustive mapping

- Custom Serializers for System.Text.Json (Json <> Api.Dtos)
- Custom Serializers for MongoDB C# driver (Bson <> Persistence.Dtos)
- DUs + exhaustive matches are used in both Api.Dtos and Persistence.Dtos!
- No AutoMapper or similar used, hand-written mapping only ;)



Anonymous Records for strongly-typed db queries

- Anonymous Records are a perfect fit for strongly-typed MongoDB queries
  - Parameter values are directly supplied upon initialization
  - SQL Injection not possible

```
async {
    let filter =
        {|
            _id = accountId
            ``$or`` =
                [
                    {| lastChgTs = null |} :> obj
                    {| lastChgTs = {| ``$lt`` = changeTimestamp |} |}
                ]
        |} : {| $or: obj list; _id: Guid |}
        |> FilterDefinition.create : FilterDefinition<'a>

    let update =
        {|
            ``$inc`` = {| bal = amount |}
            ``$set`` =
                {|
                    modOn = DateTime.UtcNow
                    lastChgTs = changeTimestamp
                    lastTrxId = trxId
                |}
        |} : {| $inc: {| bal: decimal |}; $set: {| lastChgTs: BsonTimestamp; lastTrxId: Guid; modOn: DateTime |} |}
        |> UpdateDefinition.create : UpdateDefinition<'b>

    return! findOneAndUpdate filter None update |> AsyncResult.ignore
}
```

Function names can be English sentences

- Ideal for BDD-style integration tests

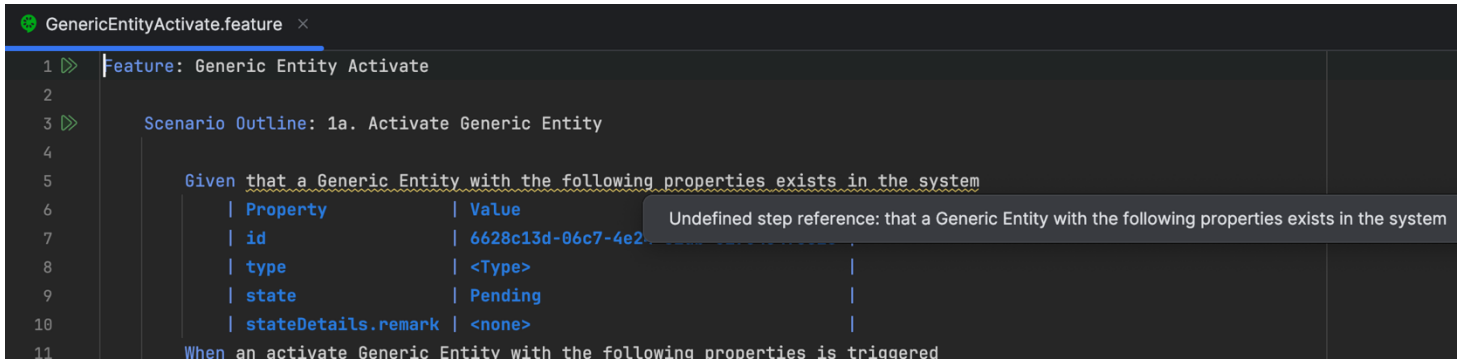
```
[<When>]
Yevgen Cherkasenko
let ``a new GenericEntity with following properties is created`` (properties: Table) : Context =
    let ctx = Context.create ()

    // start the watcher
    let cancelTokenSource =
        TestServer.start
            "GenericEntityWatcher"
            (DependencyInjection.GenericEntityWatcher.watchCollectionAndHandleChangesWithHandleNotifyItem
                (Assert.handleNotifyItem ctx.ChangeHandledEvent)
                None)
```

```
[<Then>]
Yevgen Cherkasenko +1
let ``a GenericEntity audit trail entry with the following properties is published``
    (properties: Table)
    (ctx: Context)
    : unit =
    let properties =
        properties.Rows : string array array
        |> Table.replacePlaceholders (
```

F# Tooling

- Fantomas for automatic code formatting
  - Fixed version in .config/dotnet-tools.json
- TickSpec for BDD style tests
  - Gherkin syntax highlighting IDE plugin works
  - But no IDE support for jumping between *.feature and *.fs files
- Debugging Step Into goes to Nirvana
  - Especially when in CEs ..
  - Disable "Enable external source debug" in Rider for example

```
# Fantomas properties
[*.{fs,fsx}]
fsharp_align_function_signature_to_indentation = true
fsharp_max_array_or_list_width = 80
fsharp_max_function_binding_width = 80
fsharp_max_if_then_else_short_width = 80
fsharp_max_infix_operator_expression = 80
fsharp_max_record_width = 40
fsharp_max_value_binding_width = 80
fsharp_multiline_block_brackets_on_same_column = true
fsharp_alternative_long_member_definitions = true
fsharp_record_multiline_formatter = number_of_items
fsharp_max_record_number_of_items = 2
```



Part 2: What F# challenges have we been facing?

1. Pipelines are overused
2. Long fs files
3. Some .NET BCL/C# features hard to map to F#
4. Functions with many parameters due to DI
5. No real early return
6. Debugging experience
7. Async vs. Task CEs

Pipelines are overused

- Option.map + Option.defaultValue vs. if-else/match with
- Very long pipeline vs splitting the code into several small pipelines/code blocks (by using CEs + do!/let!-ing)
  - Additionally async { ... and return! neutralize themselves

```
new *
let someOtherFunctionReturningOption () : int option = Some 1
new *
let someOtherFunction x : 'a : Async<unit> = async { return () }

new *
let f1a () : Async<unit> =
    async {
        let x : int option = someOtherFunctionReturningOption ()
        // ...
        do!
            x : int option
            |> Option.map (fun x -> someOtherFunction x) : Async<unit> option
            |> Option.defaultValue (async { return () }) : Async<unit>
    }

new *
let f1b () : Async<unit> =
    async {
        let x : int option = someOtherFunctionReturningOption ()
        // ...
        match x with
        | Some x -> do! someOtherFunction x
        | None -> return ()
    }
```

```
let someOtherFunction1a x : 'a : Async<'a> = async { return x }
let someOtherFunction1b x : 'a : Async<'a> = async { return x }
let someOtherFunction2a x : 'a : 'a = x
let someOtherFunction2b x : 'a : 'a = x

let f1a x : 'a : Async<'a> =
    async {
        return!
            x : 'a
            |> someOtherFunction1a : Async<'a>
            |> Async.bind someOtherFunction1b : Async<'a>
            |> Async.map someOtherFunction2a : Async<'a>
            |> Async.map someOtherFunction2b : Async<'a>
    }

let f1b x : unit : Async<unit> =
    async {
        let! intermediateResult =
            x
            x : unit
            |> someOtherFunction1a : Async<unit>
            |> Async.bind someOtherFunction1b : Async<unit>

        let intermediateResult2 =
            intermediateResult : unit
            |> someOtherFunction2a : unit
            |> someOtherFunction2b : unit

        return intermediateResult2
    }
```

```
let f1a2 x : 'a : Async<'a> =
    x : 'a
    |> someOtherFunction1a : Async<'a>
    |> Async.bind someOtherFunction1b : Async<'a>
    |> Async.map someOtherFunction2a : Async<'a>
    |> Async.map someOtherFunction2b : Async<'a>
```

Long fs files

- Multiple types, modules, functions per file
- Quite a few files are close to or even above 2k LOCs
  - Usually Api.Functions.fs, containing orchestration code
- Culture shock for most C# developers ;)
- The long files should be split …
  - but we are a bit lazy – longer files also work fine …
  - and having only a few files (<10) per microservice also simplifies the understanding

Some .NET BCL/C# features hard to map to F#

- E.g. https://github.com/fsharp/fslang-suggestions/issues/255
  - F# tries to eagerly infer types, but that gets it into trouble with generics and when clauses narrowing the type …
- IAsyncEnumerable is not in FSharp.Core, but in FSharp.Control.AsyncSeq (or TaskSeq)
  - We still have some very old code like this, where at the beginning we were wondering how to iterate asynchronously in F# …

```
|> Async.RunSynchronously) : 'dtoT list option  // SMELL, is there some form of IAsyncEnumerator ????
```

Functions with many parameters due to DI

- Affects mainly orchestration functions (Api.Functions.fs) and usually not too bad
- But it can get very long for some orchestration functions
- Also problem with "Go to declaration" for function dependencies
- Perhaps: Group function dependencies in anonymous record?
  - (+) All function dependencies get applied together/at the same time anyway
  - (-) Nesting + PascalCasing and dot notation upon invoking

```
Yevgen Cherkasenko +1
let activate
    (findGenericEntityById: Guid<GenericEntityId> -> Async<Result<GenericEntity, EntityNotFoundByIdError>>) //function dependency
    (replaceGenericEntity: GenericEntity -> AsyncResult<GenericEntity, UpdateEntityError>) //function dependency
    (req: MappedHttpRequest) // input
    : AsyncResult<MappedHttpResponse, MappedHttpErrorWithLogInfo<_>>
    =
    asyncResult {
        // parse and validate command
```

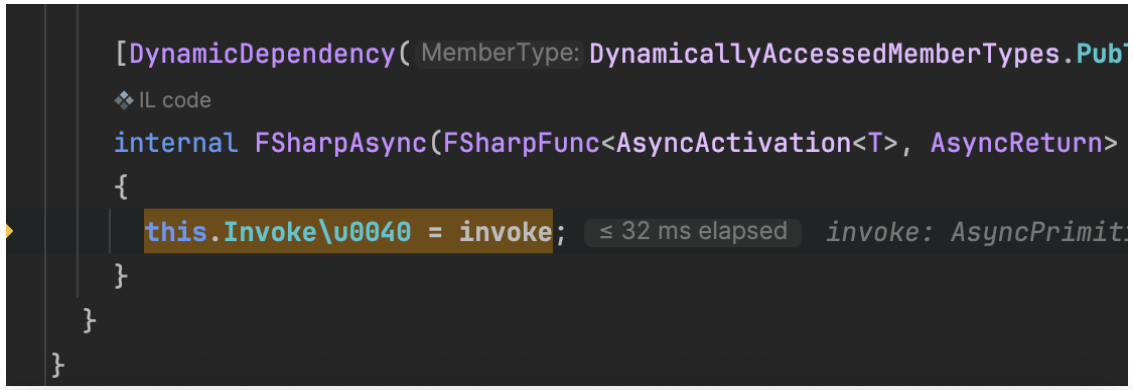
```
Yevgen Cherkasenko +4
let registerDocumentMetadata
    (insertDocument: Document -> Async<Result<Document, InsertEntityError>>) // function dependency
    (findDocumentById: Guid -> Async<Document>) // function dependency
    (replaceDocument: Document -> Async<Result<Document, UpdateEntityError>>) // function dependency
    (getDocumentUploadLink:
        Guid<DocumentId>
            -> ReferenceEntityType
            -> ReferenceEntityId option
            -> FileName
            -> LimitedString
            -> TriggerSource
            -> Async<BlobLink>)
    (req: MappedHttpRequest)
    : Async<Result<MappedHttpResponse,MappedHttpErrorWithLogInfo<ValidationErrorReasons>>> =
    asyncResult {
        let! cmd =
```

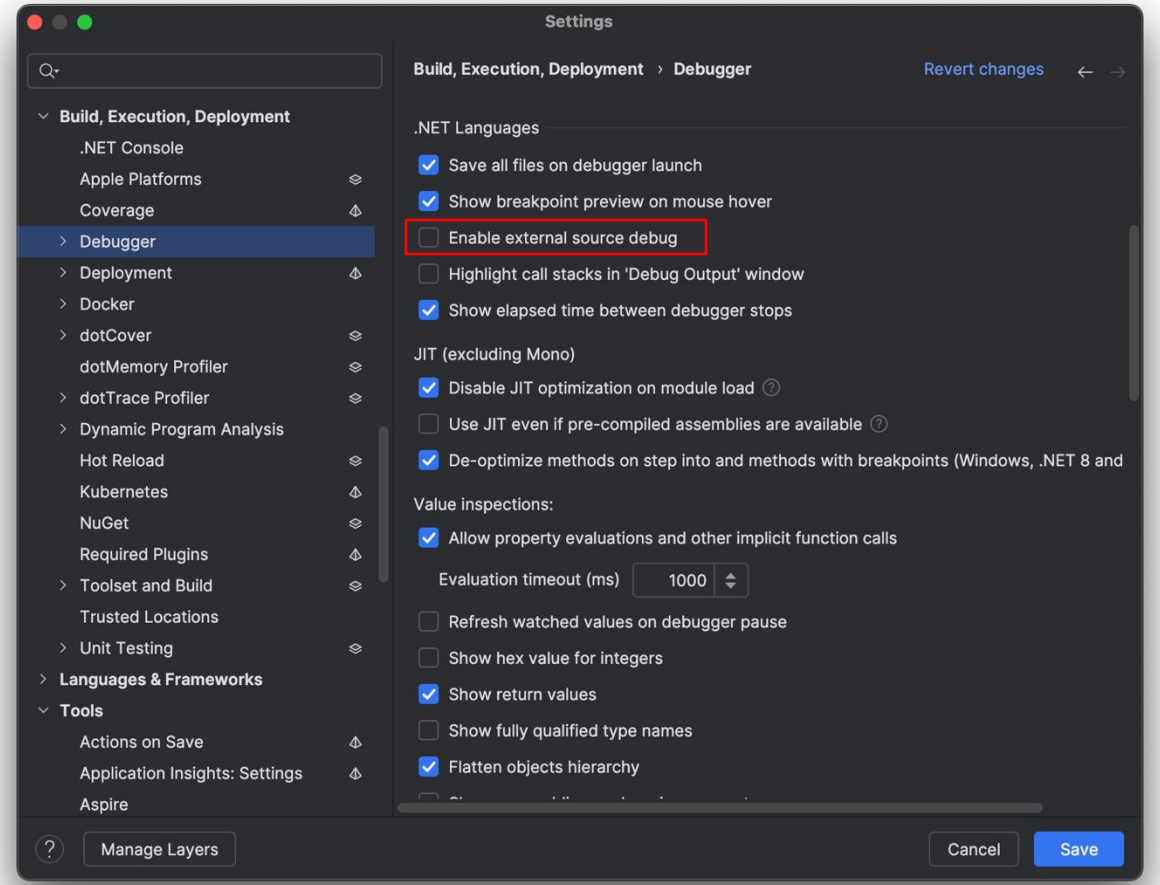
No real early return

- F# is expression-based (which is a good thing!) so shortcuts and early returns are not possible
- Alternative: extract branches into sub-functions, and call these in an orchestration function
- Do not get mislead by special unit-returning functions or CEs
  - "unit however is special"
  - Else can be skipped if return type is unit
  - Shortcutting (e.g. with return! Error) only works in some computation expressions (e.g. asyncResult)
  - Both of the above are unreliable and sometimes misleading
- Repl/EarlyReturnTests.fsx

Debugging Experience

- Stepping into, especially in computation expressions, leads you quickly to "no mans land"
- Disable external source debugging to remain sane ;)
- But on the positive side: **fsx are debuggable since recently!!!**





Async vs. Task CEs

- Tasks / Task CEs have been re-vamped and optimized in F# 6.0
  - F# RFC FS-1087 - Resumable code and resumable state machines
  - Async / Async CEs have not been optimized
- Performance benefit of task vs. async CE is likely overstated #34856

- What now
  - Task CEs everywhere?
    - Prefer cold/not-started Async CEs …
  - Async CE everywhere, Task CE only for Interop?
- When will Async CEs be optimized as well?
  - "Anyway, the approximate reimplementation appears to run as fast as TaskBuilder for sync cases, and as fast as tasks for async cases. That makes it like 10-20x faster than the current F# async implementation. Stack traces etc. would be greatly improved to."

Bonus: F# vNext?

- "If I had 3 wishes ..."
  1. Anonymous Discriminated Unions
     - for the case where every function has a specific very narrowly defined set of possible errors (not shared across functions)
     - Currently we have to define a DU, e.g. ProcessCommand1Error with cases Error1 and Error2
     - If anonymous DUs were available we could directly write in the signature of the function that it returns **Result<_, Error1 | Error2)**
  2. Optimize Async CEs in the same way as Task CEs have been optimized
  3. No 3rd wish, F# is perfect ;)

```
type Error1 = {
    SomeValue1: int
    SomeValue2: int
}

type Error2 = {
    SomeValue1: int
    SomeValue2: int
}

let orchestrate someArg1 : int someArg2 : int : Result<unit, Error1 | Error2> =
    if someArg1 = 1 then
        Ok ()
    elif someArg2 = 1 then
        Error1 { SomeValue1 = 1; SomeValue2 = 2 }
    else
        Error2 { SomeValue1 = "a"; SomeValue2 = "b" }
```

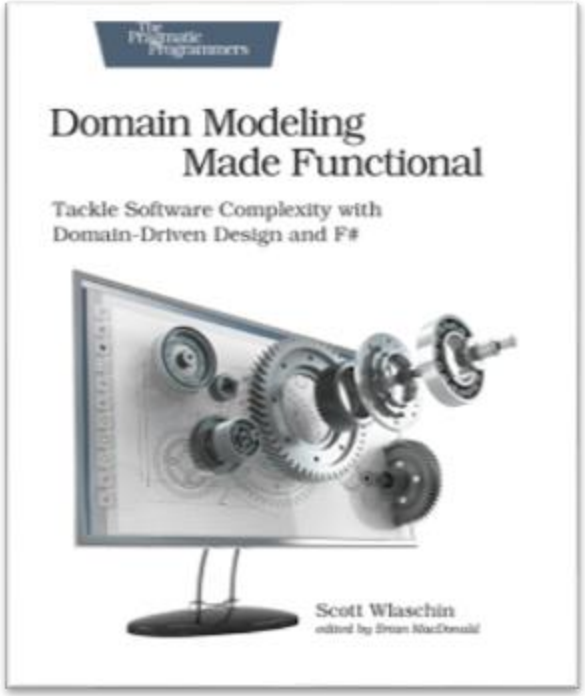
Bonus: Recommended Books about FP

Grokking Simplicity
by Eric Normand

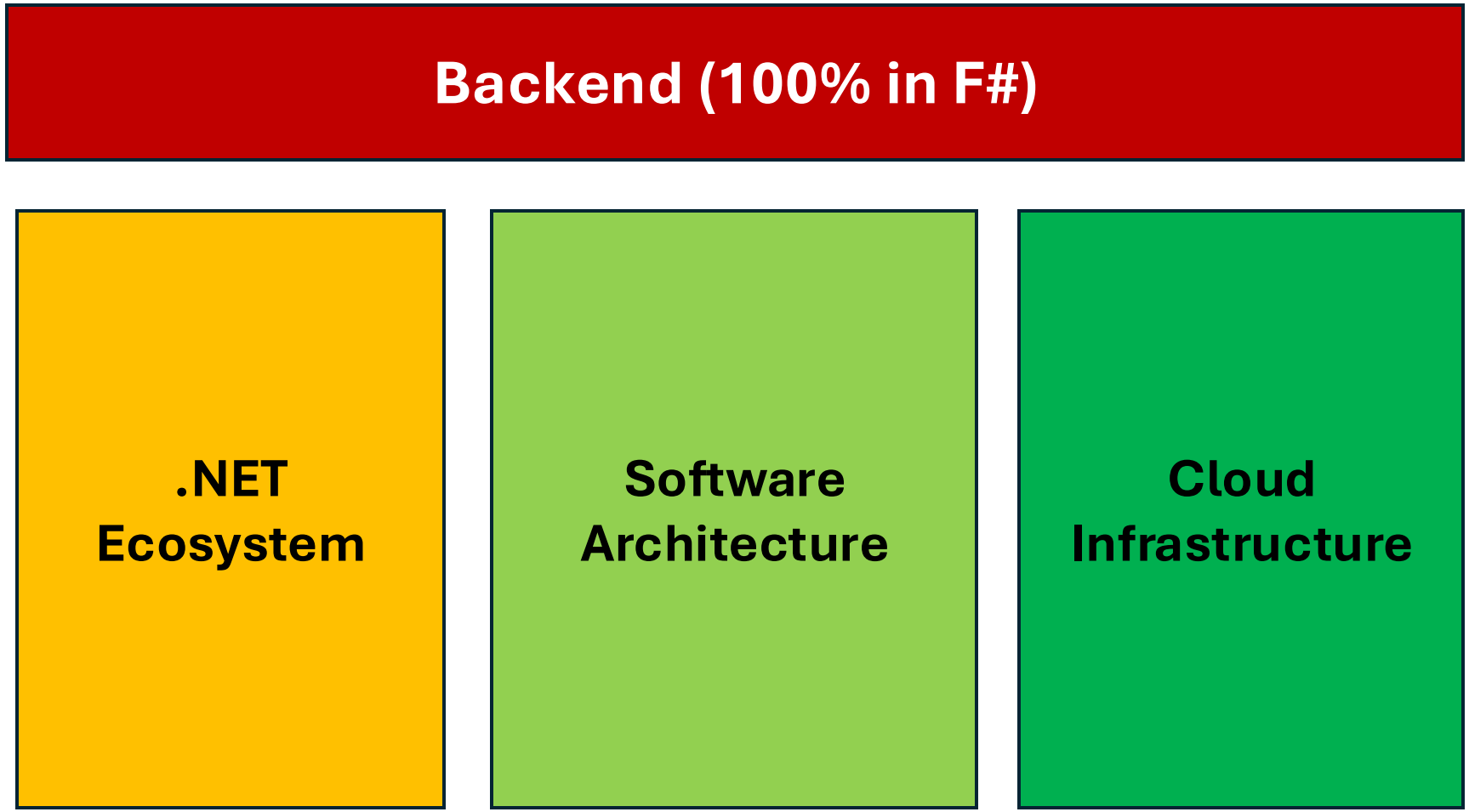
Explains what functional thinking is, and why you should prefer data before calculation (pure functions) before actions (impure code), etc.

Domain Modeling Made Functional
by Scott Wlaschin

Explains that an application is nothing more but a chain (pipeline) of functions, etc



… but F# is only the top of the iceberg …



.NET Ecosystem

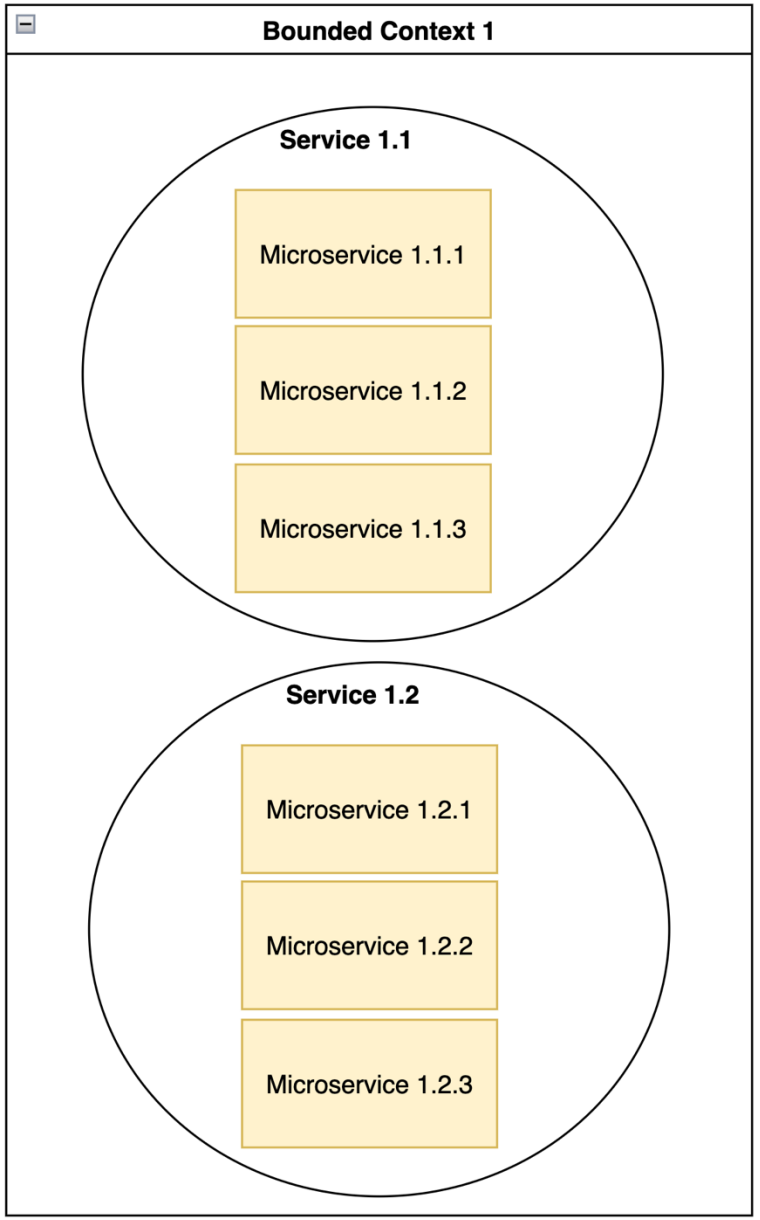
- Very Stable Foundation
  - Backed up by MS, will not die
  - F# might be a small dot on the map, but .NET is a huge continent!
  - Bigger ecosystem than the ones hosting other functional languages
- Constant Improvements with every new .NET version
  - Not only LTS, enjoying currently .NET 9 ;)
- Huge amount of BCL functionality, C# SDKs for everything
  - AWS, Azure, GCP SDKs, all covered
  - Database SDKs – e.g. MongoDB
  - SMTP, SFTP, etc.
- Not using Entity Framework though
  - No need, DDD + Document/NoSQL db
- Need to be careful with some tech from MS
  - Blazor, MAUI

Part 3: Some Architectural Principles

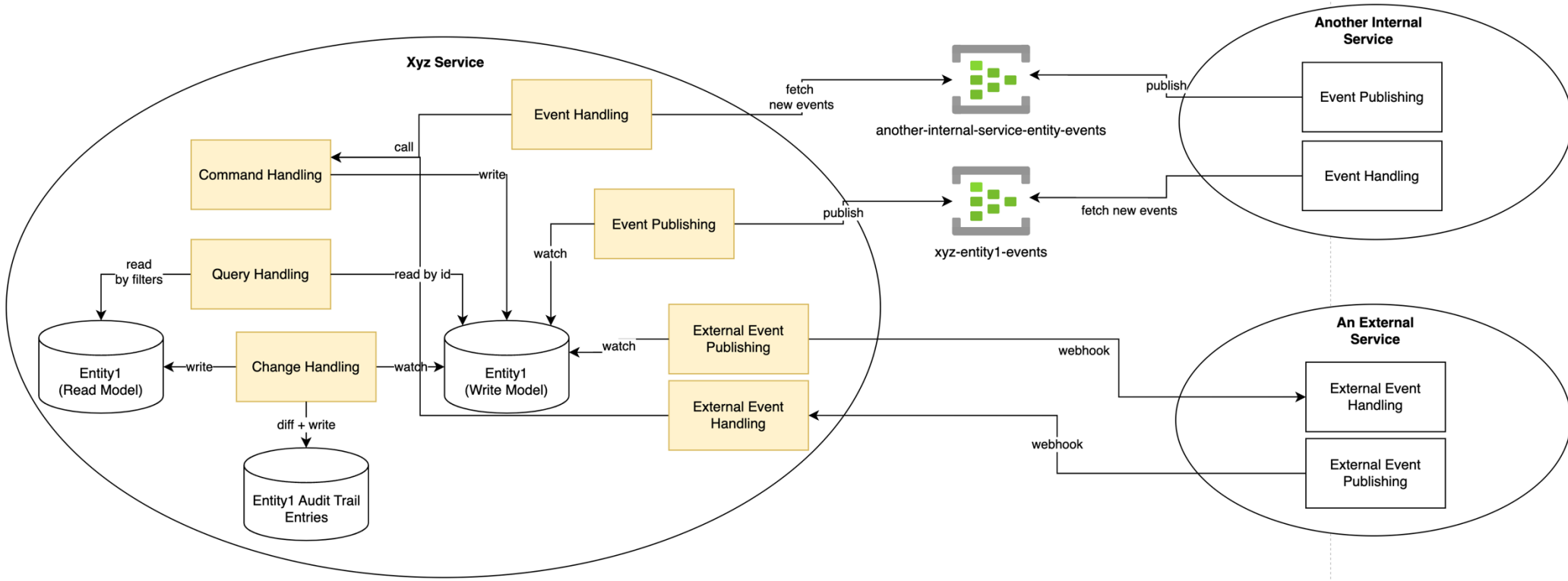
1. Microservices (but with monorepo)
2. CQRS+
3. Change Streams / CDC / Change Feed etc.
4. Current State + Last Event (no Event Sourcing)
5. Idempotency
6. Application Layers, "Project Layers"
7. Contract-first REST API Design
8. No ORMs and no database transactions, store entities 1:1 in a document database
9. 2-Phase Commit
10. BDD (Gherkin)
11. Inverted Testing Pyramid
12. Consumer Driven Contract Testing (CDCT)
13. Mock only at the lowest level

Microservices

- A lot of microservice backlash lately
  - You should always start with a monolith
  - Microservices = Distributed Monolith (worse than "normal" Monolith)
- Services/Microservices = divide and conquer
  - Service = Simpler/smaller "application worlds" or domains with 1-3 entities each
  - Microservice = part of a Service, broken down by technical aspect (see CQRS+ on next slide)
- Microservices are easy to deploy, individually
- Microservices are easy to scale and monitor
  - CPU/RAM consumption, K8s requests/limits
- Each Service has a clear API contract
  - Clear boundaries! (vs monolith)
  - The API serves as the testing surface
- Each Microservice has its own health check
  - Invoked upon startup and regularly afterwards (e.g. every x minutes)

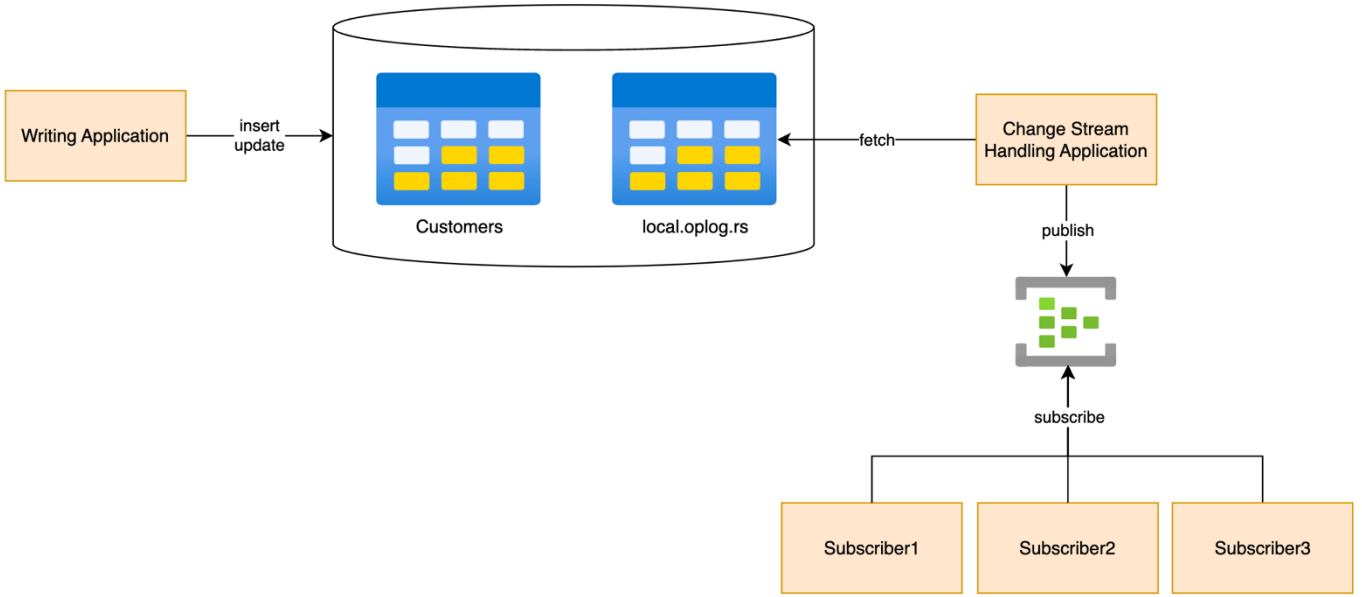


CQRS+



- CQRS = Write (Command Handling) + Read (Query Handling)
- CQRS+ = CQRS + a bunch of additional microservices
  - Change Handling
  - Event Publishing, External Event Publishing
  - Event Handling, External Event Handling
- Clear definition Service <> Microservice, and focus of each Microservice on a specific topic
  - Enables Microservice templates
- Domain Logic only in Command Handling microservice
- See https://dev.to/deyanp/cqrs-5276 for more details

Change Streams / CDC / Change Feed etc.



- A simple problem which is "haunting" 99% of business applications – write to db + write to message bus
- The best solution to it are Change Streams (and similar)
  - Alternative solution: Outbox Pattern
- Change Streams are basically database events which the application can process in a reliable way eventually
  - The (Command Handling) Application is solely responsible for writing into the db
  - A separate (Change Handling/Event Publishing) Application is solely responsible for processing the database events and publishing events to message bus, etc.
- See https://dev.to/deyanp/use-change-streams-instead-of-traditional-outbox-or-distributed-transactions-cdb for more details

Current State + Last Event

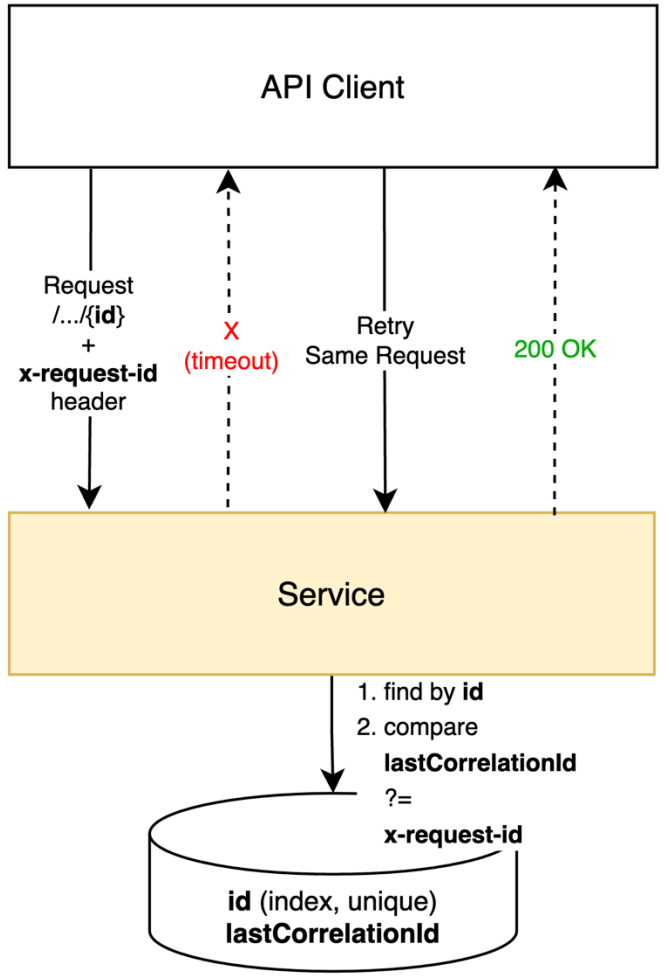
- Event Sourcing has its issues
- The "Current State" approach has some benefits
  - Easier to understand
  - Faster to materialize when processing commands
- How to make the "Current State" approach audit and EDA-friendly though?
- Answer: Current State + Last State
- See https://dev.to/deyanp/tbd-current-state-last-event-as-an-alternative-to-event-sourcing-5gm5 for more details

```
{
  "id": "ab607a9e-4662-11ea-b979-eb44b02db7b9",

  "createdOn": "2021-01-01T10:00:01.000Z",

  "someProp1": "value1",
  "someProp2": "value2",
  "someProp3": "value3",

  "lastModifiedOn": "2021-04-06T11:33:51.377Z",

  "lastEvent": {
    "type": "AddressChanged"
  }
}
```

```
type CustomerEvent =
| Registered
| Relocated

type Customer = {
    Id: CustomerId
    Address: Address
    LastEvent: CustomerEvent
}

module Customer =
    let relocate customer newAddress =
        // TODO: some validation here

        { customer with
            Address = newAddress
            LastEvent = CustomerEvent.Relocated // note this
        }
```

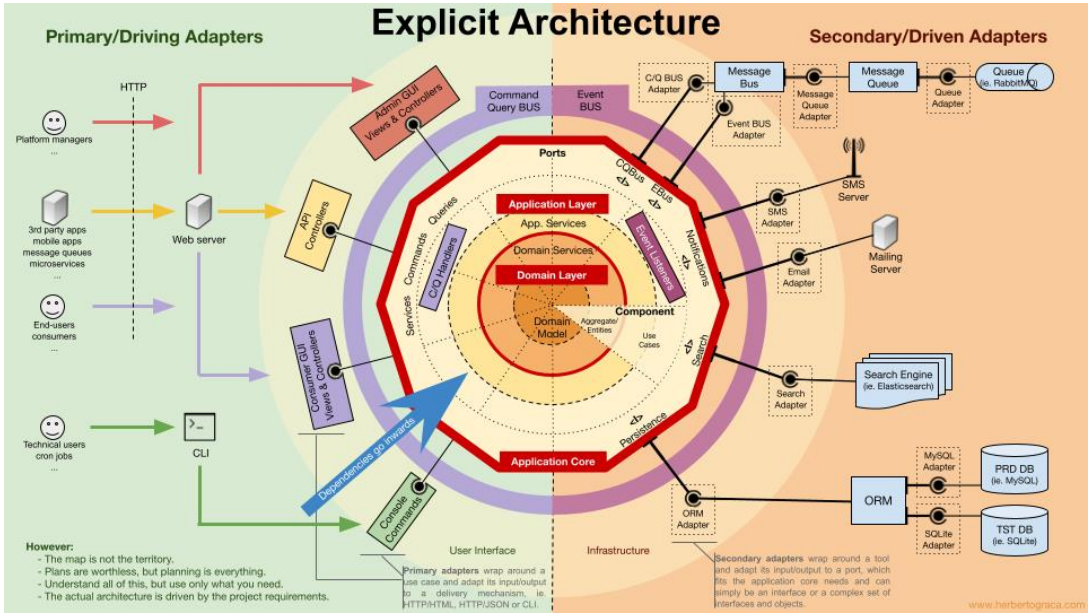
Idempotence

- Simplifies client and server code
  - Alternatives – a lot of ifs and "try fetch from the server"
- A must for all APIs which are not idempotent by default
  - Command Handling
- Client-side generated GUID/UUID entity **id**
  - Allows to quickly find the entity by id, regardless of database partitions, shards
- Client-side submitted GUID/UUID **x-request-id** header
  - Stored as attribute of every entity, and compared – if last state has this id, then this is a retry
- (+): No server-side caches (with time limitation) required with this approach
- (-): Application code required (vs. middleware)

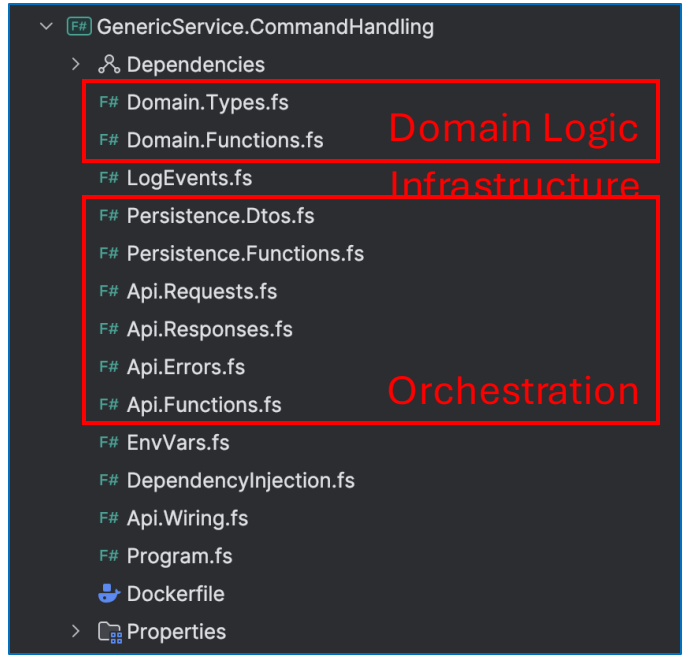


Application Layers

- Hexagonal Architecture by Alistair Cockburn, Onion Architecture by Jeffrey Palermo, Clean Architecture by Robert C. Martin:
  - Dependencies point from outside to the inside of the circle
  - Make business logic (at the center of the circle) unaware of infrastructure and external dependencies (incl. own exposed API)
- Data is remapped at every layer (1:1): API DTOs <> Domain Types <> Persistence DTOs
- Orchestration (Api.Functions) layer knows about everything
  - API
  - Domain
  - Persistence
- Persistence knows about Domain
- Domain knows about nothing

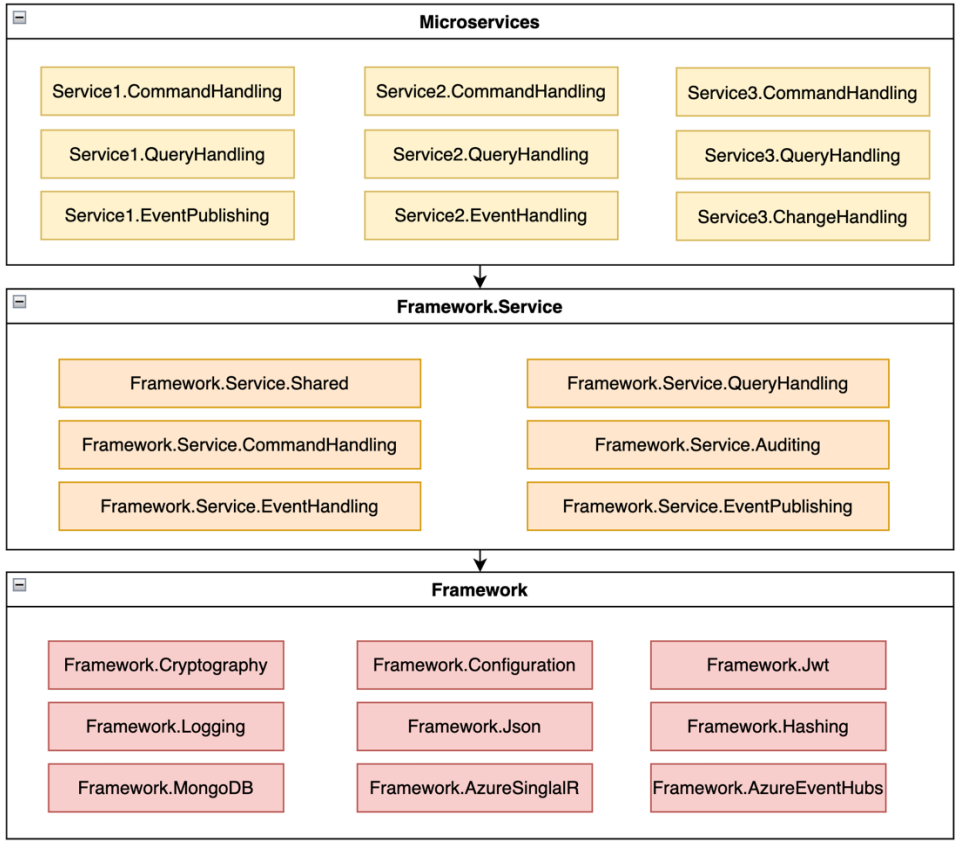


From DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together



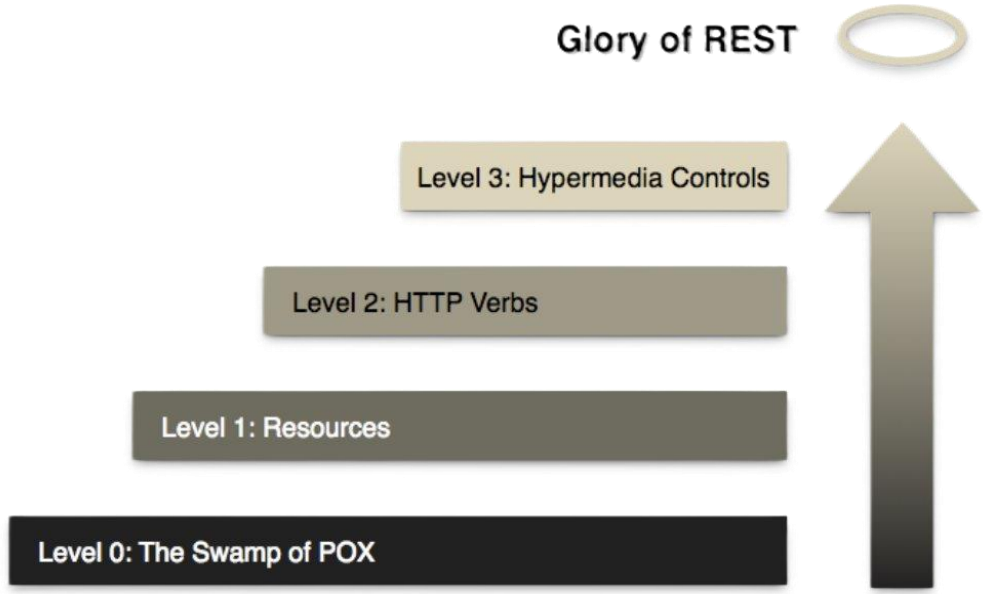
"Project Layers"

- Generic Infra/Helper code goes to Framework project
  - Microservice-agnostic
- Code specific to same class of microservices (e.g. *.EventPublishing) goes to Framework.Service.Xyz (e.g. Framework.Service.EventPublishing)
- Clear Project Layers (top: most specific, bottom: most generic)
  - Service1.CommandHandling
  - Framework.Service.CommandHandling
  - Framework.Xyz
- **Beware**: F# enforces bottom->up dependencies only within a project, **Project/Package References are not controlled and can be a source for wrong dependencies**!



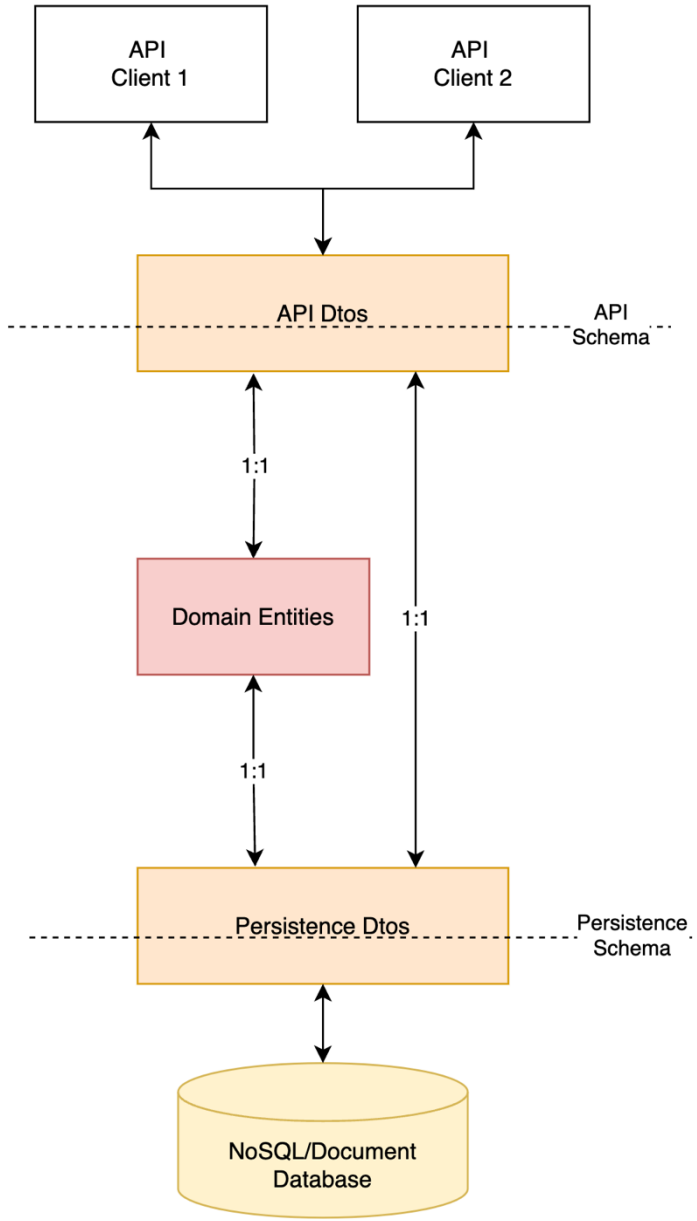
Contract-first REST API Design

- API Schema is very important
  - Handcrafted Swagger/Open API Spec yaml file ;)
- Api DTOs (simple F# record types) mapping 1:1 to the API schema
  - No code autogen, manually written
- At level 2 of the Richardson Maturity Model
  - no hypermedia (yet)
- Always backwards-compatible changes
  - We are still at v1 major version of all our APIs ;)
- Some **important peculiarities**:
  - PUT + Client-side generated GUIDs for entity ids for entity creation
  - Prefer POST + /.../verb-noun (e.g. /close, /reopen) instead of PATCH
  - Return Entity in the response to PUT/POST/PATCH calls instead of Location header + additional GET



No ORMs and no database transactions, store entities 1:1 in a document database

- 1 Entity = 1 document in the db = 1 atomic insert/update
  - No need for ORM
  - No need for db transactions
- Persistence DTOs layer containing the database schema
  - Similar to API DTOs layer containing the API schema
- 2 entities can be reliably written *together* in an eventually consistent way (Event Handling)
  - also in the same service
  - for sure in other services
- Alternative: execute 2 different entity insert/updates in an order, and take special care upon retry …
- Good domain modeling = correct Aggregate Roots and Transaction Boundaries



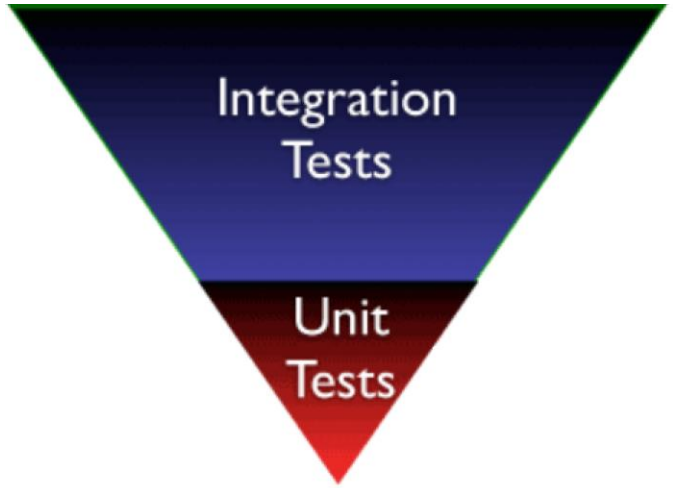
BDD (Gherkin)

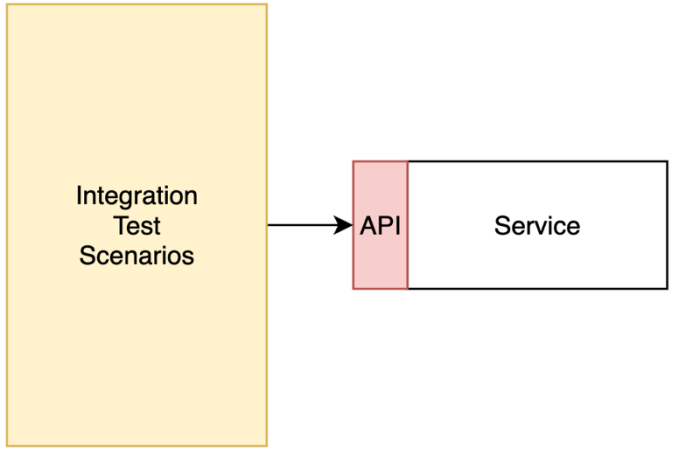
- Gherkin = Given – When – Then
  - = Arrange – Act – Assert
- Only 1 When!
  - Multiple Givens and Thens are OK
- Possibility to pass parameters as tables (readable)
- Readable tests (without code)
  - Also "Living Documentation", but not a priority for us
  - Feature Scenarios are allowed to be a bit more technical (but don't tell that to Gojko Adzic ;))
- Possibility to reuse of steps => write new test scenario with a single line of code!
- TickSpec is a very nice F# BDD Framework!

```
Feature: Generic Entity Activate

    Scenario Outline: 1a. Activate Generic Entity

        Given that a Generic Entity with the following properties exists in the system
            | Property            | Value                                |
            | id                  | 6628c13d-06c7-4e24-82db-e17e434f3528 |
            | type                | <Type>                               |
            | state               | Pending                              |
            | stateDetails.remark | <none>                               |
        When an activate Generic Entity with the following properties is triggered
            | Property        | Value                                |
            | genericEntityId | 6628c13d-06c7-4e24-82db-e17e434f3528 |
            | remark          | <Remark>                             |
        Then a response with http status code 200 is returned
        And the response has the following properties
            | Property            | Value                                |
            | id                  | 6628c13d-06c7-4e24-82db-e17e434f3528 |
            | type                | <Type>                               |
            | state               | Active                               |
            | stateDetails.remark | <Remark>                             |
        And the Generic Entity is stored with the following properties
            | Property       | Value                                |
            | _id            | 6628c13d-06c7-4e24-82db-e17e434f3528 |
            | type           | <TypeBson>                           |
            | state          | 1                                    |
            | stateDtls.rmrk | <Remark>                             |
            | lastEvent.type | 1                                    |

        Examples:
            | Type  | TypeBson | Remark      |
            | Type1 | 0        | <none>      |
            | Type2 | 1        | <none>      |
            | Type1 | 0        | Some Remark |
            | Type2 | 1        | Some Remark |
```

Inverted Testing Pyramid

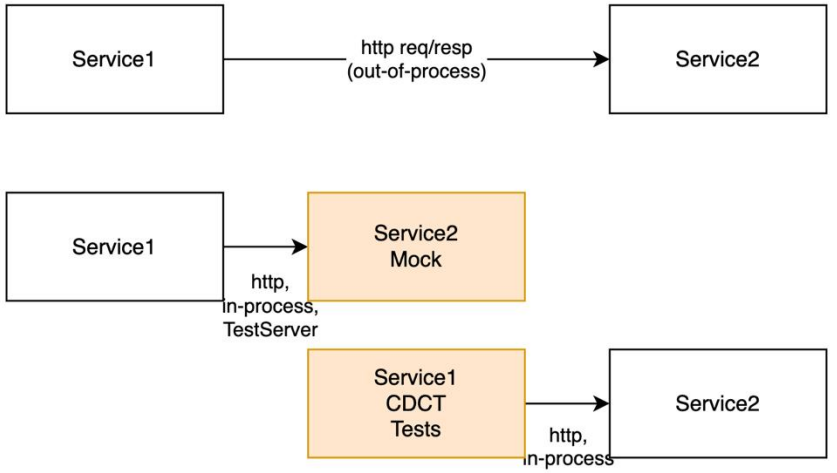
- Focus on API tests, including calls to the database or other critical infrastructure (blob storage, message bus)
  - database etc. can be mocked in some test scenarios, but is usually not mocked
  - database calls introduce some fragility, but every test can focus only on its data and clean it up or reuse it appropriately
  - database commands/queries are a source of bugs
- Benefits of API Integration Tests
  - survive internal refactoring much better
  - guarantee that your microservice really works (vs. only a very small part of it)
  - the right level of granularity for testing - APIs usually map better to actions
- Unit Tests are only required for special/complex business logic, algorithms (a few)
- Another heretical thing: when preparing existing data for integration tests (in Arrange/Given) we insert data directly into the database ;)

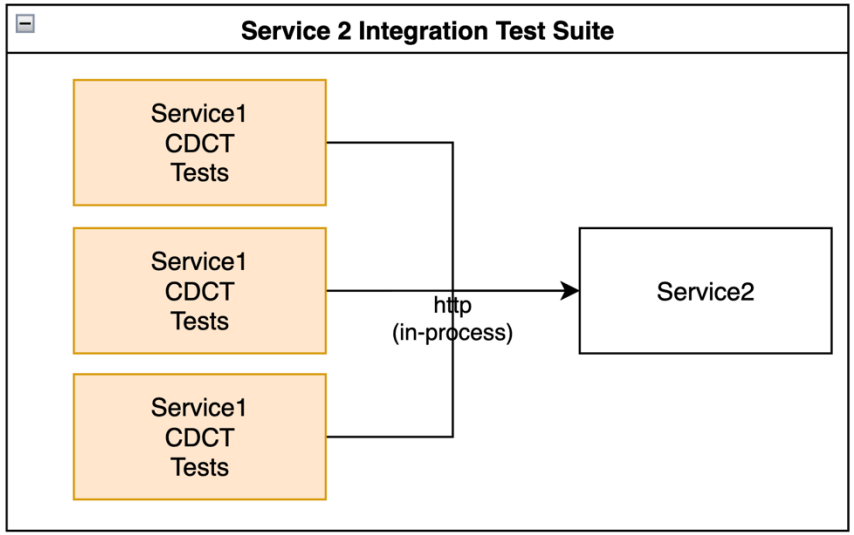




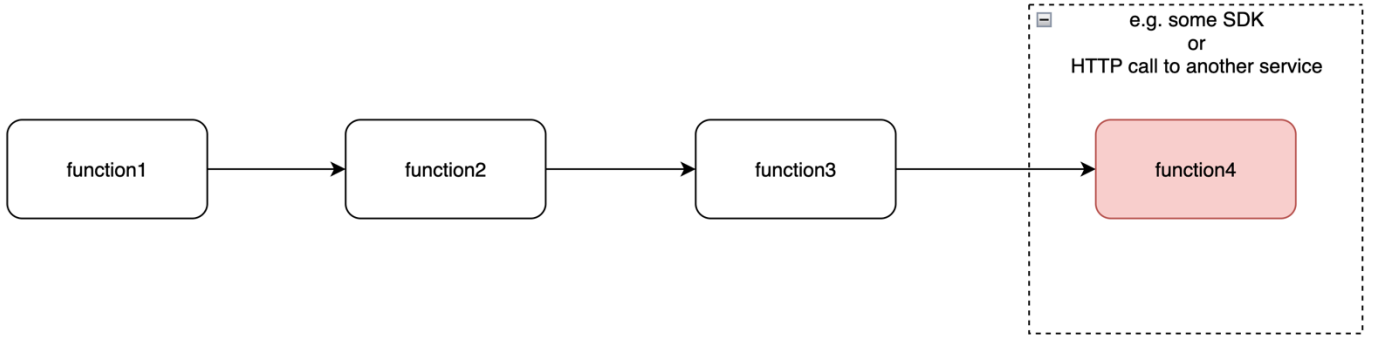
Consumer Driven Contract Testing (CDCT)

- How to have tests
  - running in-process in a build/deployment pipeline
  - Isolated per microservice (not spanning multiple microservices)
  - without maintaining a shared integration test environment?
- If there are no inter-microservice calls – no problem
- If there are A -> B microservice calls – we have a problem
  - We can mock the target microservice in the tests of the called microservice
  - But how do we make sure the mocks are correct, and always up-to-date?
- CDCT = take the mocks and add them to the integration test suite of microservice B
  - Automate them there, make sure they are always green
  - If A changes the mocks => update the corresponding tests in B!
  - B knows now what its clients expect, is more confident upon refactoring
  - Client can be another microservice, but also UI
- Pact.io is a well-known CDCT Framework
  - We do not use it, we carry over the tests manually
  - It is very easy to write consumer tests (90% generic code, HTTP request - response)

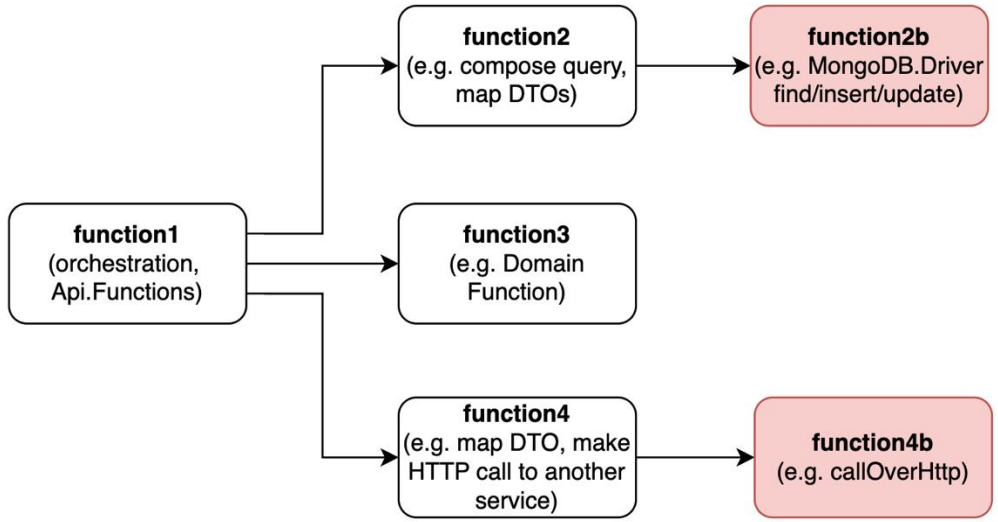




Mock Only at the Lowest Level



- Mock only function4(b)
  - In general, orchestration should be flat and we should not have so many levels of function calling though
  - Infrastructure functions are usually generic and used by many microservices
    - e.g. **callOverHttp**: HttpClient -> MappedHttpRequest -> Async<MappedHttpResponse>
- Goals
  - Maximize code under test
  - Reuse mocked implementations



2-Phase Commit

- Service1 -> Service2, Service3
- Without db transactions (as across services)
  - As per MongoDB official example from the time when it did not have db trxs
  - Later db trxs have been added, but not a good idea across service databases!
- First, create/commit
- Second, complete or alternatively cancel
- Using atomic db operations
  - $inc, $set, $push in MongoDB
- Write additional info (e.g. pendTrxs) upon commit, which can be used later for cancellation

```
let filter =
    {| _id = accountId; pendTrxs = {| ``$ne`` = trxId |} |} : {| _id: Guid; pendTrxs: {| $ne: Guid |} |}
    |> FilterDefinition.create : FilterDefinition<'a>

let update =
    {|
        ``$inc`` = {| bal = (changeBalanceAmount |> AccountingEntryAmount.value) |}
        ``$push`` = {| pendTrxs = trxId |}
        ``$set`` = {| modOn = DateTime.UtcNow |}
    |} : {| $inc: {| bal: decimal |}; $push: {| pendTrxs: Guid |}; $set: {| modOn: DateTime |} |}
    |> UpdateDefinition.create : UpdateDefinition<'b>

let! result = findOneAndUpdate filter None update
```

```
let filter = {| _id = accountId; pendTrxs = trxId |} |> FilterDefinition.create

let update =
    {| ``$pull`` = {| pendTrxs = trxId |}; ``$set`` = {| modOn = DateTime.UtcNow |} |} :
    |> UpdateDefinition.create : UpdateDefinition<'b>

return! findOneAndUpdate filter None update |> Async.Ignore
```

Part 4: Some Infrastructure Principles

1. Main Principle: No Sys Admins
2. Everything in the Cloud, nothing "on premise"
3. Script everything
4. Fully automated deployment pipelines
5. Configuration with simple environment variables only
6. No automation for database scripts
7. Local Dev Environment

No Sys Admins

- Developers own the infrastructure => end-to-end responsibility
  - Prioritizations/Tradeoffs are made considering also the actual running software
- Automation as much as possible
  - Similar to code testing – you do not want to repeat manual activities frequently
- Choose technologies which require low level of maintenance
  - e.g. MongoDB Atlas vs running MongoDB ourselves (e.g. in K8s)
- On Call Duty (~24x7) done by developers as well
  - Alerting (esp. false positives) is a neverending story :(

Everything in the Cloud, nothing "on premise"

- The cloud is like candy shop ;)
  - You can get everything you need, incl. more exotic stuff like SMS Sending, LLMs
- No waiting on IT Infra/Ops team!
  - A project can be done by 1 developer end-to-end, including provisioning of cloud service => cut dependencies/waiting times
- Security is a topic, requires external consultancy, e.g. directly by a Microsoft Architect (free service)
- Choosing the right cloud and the right service is a topic
  - Azure was selected due to our usage of MS technologies, however not the best choice
  - Azure Event Hubs cheaper than Azure Service Bus
- Logging costs are a problem in Azure
  - 50% of our cloud costs (excl. Database) are logging costs!
- Biggest incidents related to cloud provider
  - Azure AD B2C service

Script everything

- No clicky-clicky-clicky in Cloud Portals ;)
- Simple bash scripts
  - Using cloud cli
  - Using cloud provider templates (e.g. Azure Bicep)
- Imperative vs. Declarative automation is a topic
- Setting up a complete cloud environment from scratch is a problem
  - Stress-test environment setup in Azure, GCP and AWS done couple of years ago (see https://dev.to/deyanp/mongodb-atlas-azure-a-forced-marriage-169m), but not refreshed since then ..
  - DR-wise, setting up everything in another cloud region is a challenge
    - Also are we faster than the cloud provider at solving a major incident?

Fully automated deployment pipelines

- Application (= single microservice, or UI app) Rollout takes
  - 5 minutes for binaries, integration tests and config
  - 1-2 minutes if only config (env vars) have changed (keeping same binaries)
- Azure DevOps Pipelines (similar to Github Actions) with definition in yaml
  - Using dedicated VMs with fixed IPs/VNET to accommodate for network access restrictions
- All secrets are stored in a dedicated Integration Test Key Vault instance
- Integration Tests are running **in-process in the pipeline**
- Integration Tests are running against a dedicated Integration Test Database instance
- Kubernetes (AKS) takes care of redirecting traffic from old pod to new pod without downtime

Configuration with simple environment variables only

- No platform-specific json/xml files
- No 3rd party configuration services (e.g. Azure App Configuration) supporting runtime configuration changes
  - Rollout takes only 5 minutes for binaries + config, and 1-2 minutes if only env vars have changed (keeping same binaries)
- Feature flags as simple env vars as well

No automation for database scripts

- Using MongoDB (document database) with "less db schema"
  - Collections (tables) are created automatically upon first invocation, document schema is in the application (Persistence.Dtos.fs)
  - Indexes must be created manually (could be potentially automated)
- Data Migrations (DDL) are anyway very tricky to run automatically
  - Migration scripts (written in JS for MongoDB)

Local Dev Environment

- Use cloud services whenever not easily possible to run locally (emulators are a bit shaky ;))
  - Storage Account per developer
  - Partitioned per developer Event Hubs
  - Logical DWH database per developer in a shared cloud database
- k3d for running Kubernetes locally in Docker
- mirrord for debugging against a running Kubernetes cluster
  - steal API calls
  - duplicate API calls
- See Isolated Local Dev Env using k3d, multi-tenant Azure services, docker containers, and mirrord for more information

Bonus: Favorite Azure Technologies

- Azure Kubernetes Service
  - run applications with **high availability and high per-node density**
    - ~60 pods per node with 2 vCPUs and 16Gb RAM
  - has been very stable for us in the past 5 years
    - in contrast to some other Azure services …

- Azure Data Explorer
  - confusing name, not well-known **column-store database**
  - some vague similarity to ClickHouse
  - no data modification (or very difficult), append-only
  - pretty fast (for reading) and inexpensive (smallest instance 100 EUR/month)
  - very nice query language called KQL or Kusto uses pipes (like in F# ;)), SQL is also supported
  - some nice functions for
    - aggregating last state per entity
    - anomaly detection

- Azure Event Hubs
  - low-level Message Bus, similar to Kafka (append log)
  - relatively inexpensive (35 EUR/month for 10 topics)

ZASTRPAY STUDIOS

That was it!

- Have fun with technology!
- **F# is fun!**
- Let's share real world stories about F# and the context it is being used in, and learn from each other to get better!
- **F# (Vienna) FTW!**