HODL off your axe

don’t start choppin’ up that poor monolith just yet

Máté Láng

HWSW microservices meetup, 2021 October

$ whoami

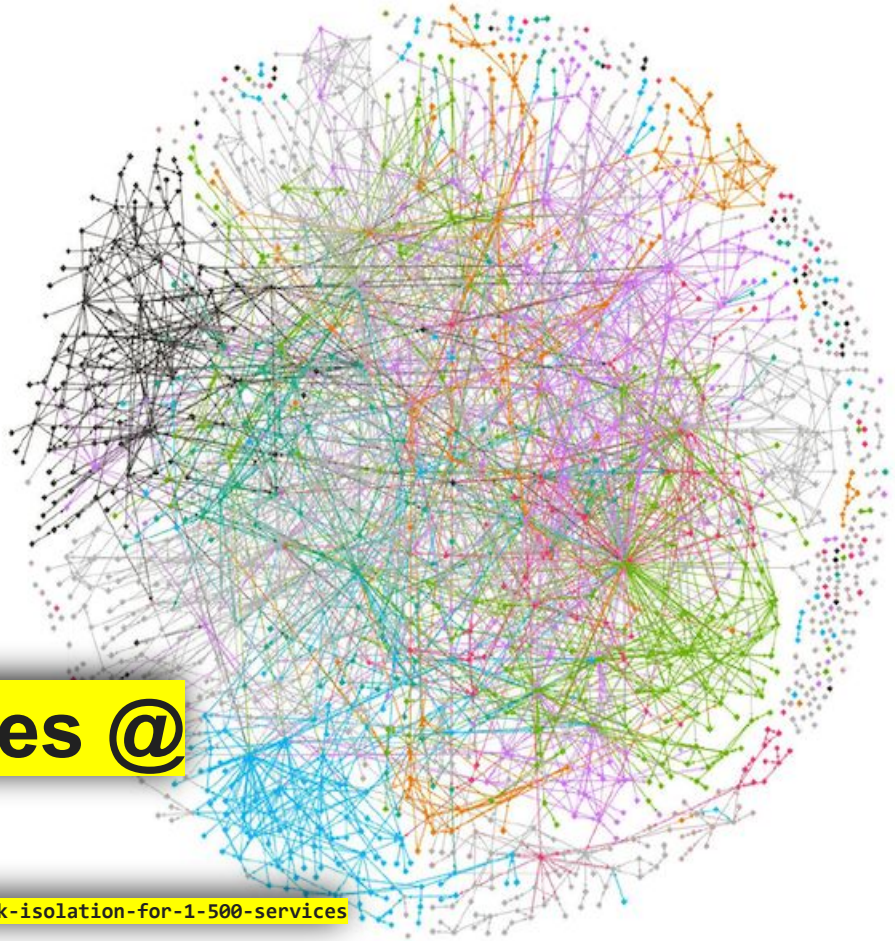
- Consultant, binhatch.com
- Previously
  - Cloud Infrastructure Team Lead @ Connatix
  - CTO @ SmartUp
  - Development Lead @ Endava
- Experience: Java, Go, AWS, Kubernetes, Terraform

- Previously @ HWSW:
  - HWSW mobile! 2018, 2019
  - IaC HWSW! meetup, March 2021
  - HWSW Terraform IaC Course Trainer, 2021

Born with passion for tech...

… but here, also infected with COVID (did not know back then) 🦠🦠🦠

HWSW IaC meetup! (online) 2021 March



Microservices @ Monzo

https://monzo.com/blog/we-built-network-isolation-for-1-500-services


The five
weaknesses of
engineers

Weakness #1

Attraction to composable & modular design

Weakness #2

Attraction to elegant complexity

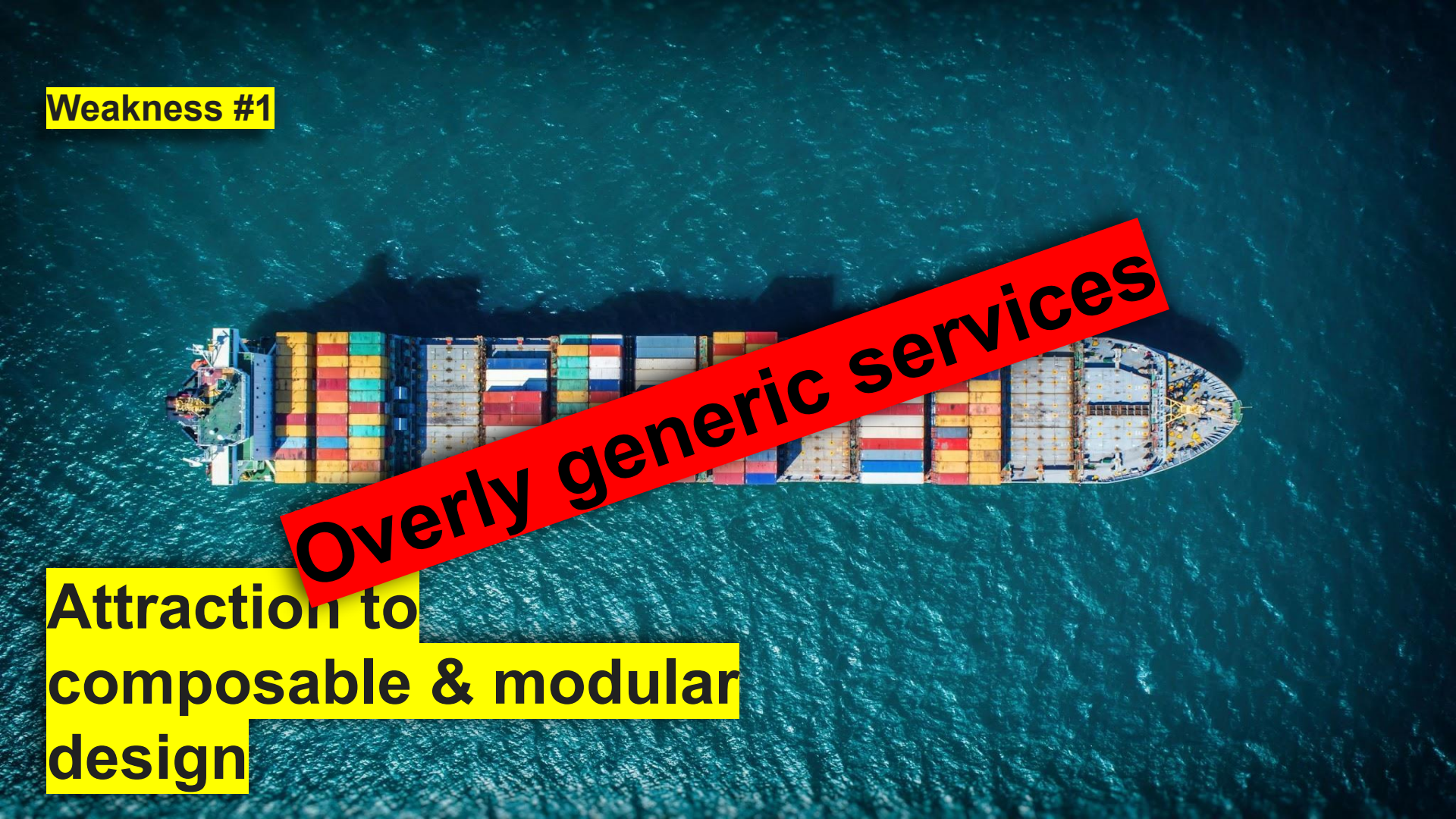

Weakness #3
Attraction to strategic thinking and control

Weakness #4

Attraction to our beloved tooling


Weakness #5
Attraction to prestigious peers

Weakness #1



Attraction to composable & modular design


Weakness #2
Overly complex implementation
Attraction to elegant complexity
SEVENTEEN
17 JEWELS
OMEGA
WATCH Co
Ω
321

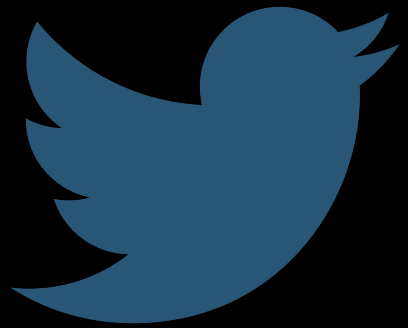

Weakness #3
Rigidity instead of agility
Attraction to strategic thinking and control


Weakness #4
Law of the hammer
Attraction to our beloved tooling


Weakness #5
Solution in search of a problem
Attracti
prestigious peers

What if we extracted functionality gradually when appropriate?

Shouldn’t be that hard. Why is it?

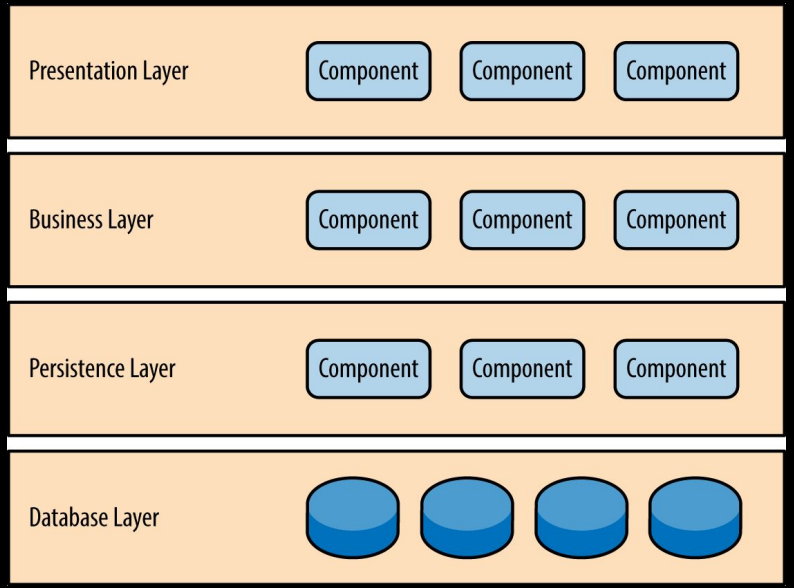


“It’s not hard if you design it to be an implementation detail. I mean, in the end, a micro service is just a function library called through a socket. Eliminating the socket shouldn’t be that hard.”

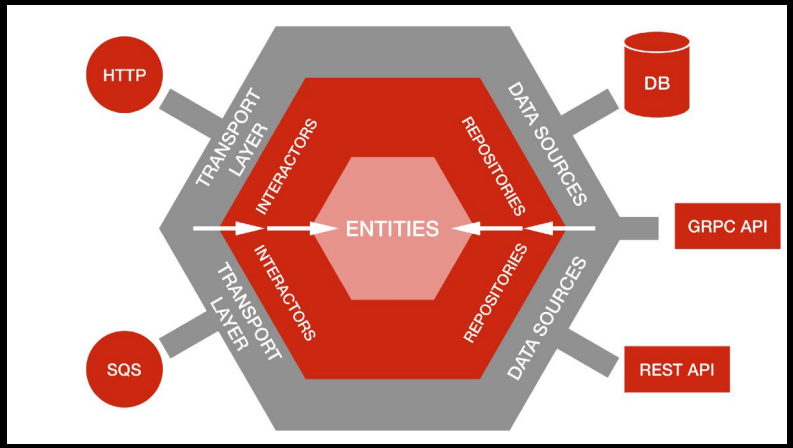
(Uncle Bob Martin, Twitter)

https://twitter.com/unclebobmartin/status/1442132963265818627

Tangled code is hard to extract



Layered architecture



Hexagonal architecture

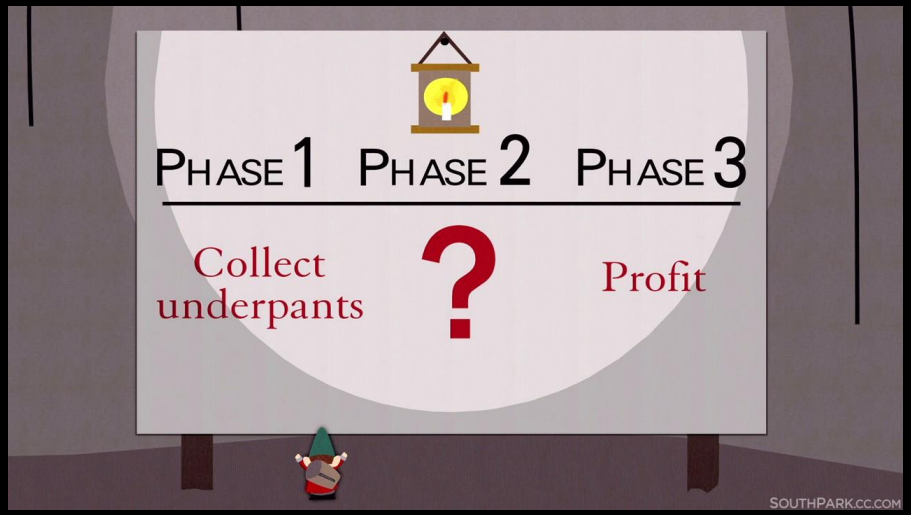
The hardest part. Your data.

- Splitting your problem domain into well defined bounded contexts is a hard exercise
- ACID gives us the impression that we have complete control, but the real world is not transactional and does not stop for a second
- Optimized data models help efficient manipulation, but are hard to maintain
- Distributed transactions are the root of all evil (embrace eventual consistency)
- In distributed systems things can go wrong, they most certainly will

developers.redhat.com/blog/2016/08/02/the-hardest-part-about-microservices-your-data

A balanced strategy

- Phase 1 - write **clean** code
- Phase 2 - write or extract to microservices when it brings advantage
  - well defined context
  - optimal data storage
  - operational (e.g scaling)
  - better suited programming language
  - organizational
- Phase 3 - rinse and repeat




A concrete case study from the buzzy world of blockchain

Building a distributed Tx Executor

- Blockchain Tx orders saved into Database
- Change Data Capture streams changes to data model
- CDC stream is sharded and competing consumers process events
- Changes describe a Finite State Machine
- At-least-once processing, any action can fail, and should be recoverable and idempotent
- There are no compensating actions on blockchain
- Needs to be horizontally scalable and support high number of Txs

What the ___ is a nonce?

- Def: “how many transactions the signer account has sent previously”, in other words, a non-repeating ordinal from an ever growing sequence for a given account
- Determines order of txs
- Prevents replay attacks
- Troubles at scale:
  - Distributed or concurrent usage can mess up the sequence if uncoordinated
  - Gaps cause txs after unconfirmed nonce to hang until sequence is complete and continuous
  - Cap on max unconfirmed txs (currently 64 pending in tx pool) - sending more evicts the pool

See: https://medium.com/swlh/ethereum-series-understanding-nonce-3858194b39bf

Requirements for a distributed tracker

- Supports managing multiple parallel identities, called **lineages** to avoid theoretical cap of 64 txs
- Each lineage has its own sequence of **tickets**, with a well defined ordinal (nonce) which can be **leased** by a consumer
- Tolerates at-least-once consumers, by being **idempotent** for all operations
- Supports signalling the success of a tx (**close** a ticket)
- Supports signalling the non-retryable rejection of a tx (**release**)
  - In this case the released ticket has to be re-assigned ASAP to a subsequent tx to fill in potential gaps
- Throttles fast consumers to avoid filling tx pools (**backpressure**)
- Can be consumed from any programming language
- Cloud native

Why implement as a microservice?

- Very specific problem, clear context boundaries, no domain pollution
- Isolated blast radius in case of bugs
- Easily testable in isolation
- Truly reusable in a wide range of projects depending on the same semantics
- Foreseeable scaling need and opportunity (64 pending tx limit / account)

Let’s imagine a service interface

23 lines (20 sloc) | 910 Bytes

```go
package ticket

import (
        "errors"
        api "github.com/welthee/dinonce/v2/pkg/openapi/generated"
)

var (
        ErrNoSuchLineage             = errors.New("no such lineage")
        ErrNoSuchTicket              = errors.New("no such ticket")
        ErrInvalidRequest            = errors.New("invalid request")
        ErrTooManyLeasedTickets      = errors.New("too many leased tickets")
        ErrTooManyConcurrentRequests = errors.New("too many concurrent requests")
)

type Servicer interface {
        CreateLineage(request *api.LineageCreationRequest) (*api.LineageCreationResponse, error)
        GetLineage(extId string) (*api.LineageGetResponse, error)
        LeaseTicket(lineageId string, request *api.TicketLeaseRequest) (*api.TicketLeaseResponse, error)
        GetTicket(lineageId string, ticketExtId string) (*api.TicketLeaseResponse, error)
        ReleaseTicket(lineageId string, ticketExtId string) error
        CloseTicket(lineageId string, ticketExtId string) error
}
```

github.com/welthee/dinonce/blob/main/pkg/ticket/ticket_servicer.go


I see true perfection here

Meet “dinonce”

- OSS
- True “microservice”. Only 6 operations.
- GoLang
- Contract first OpenAPI 3.0
- Pluggable backend (currently PostgreSQL)
- Helm chart, deploys to k8s
- Independent component tests
- Terraform module to deploy to AWS EKS + RDS

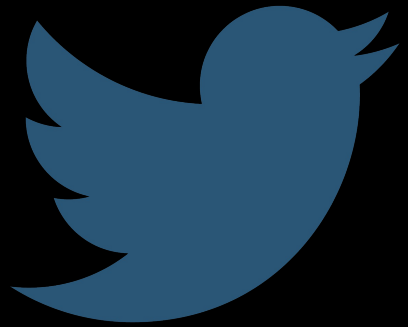
github.com/welthee/dinonce

What about the clients?

```
type TxContext struct {
  Reference   string
  AccountAddr string
}

type NonceProvider interface {
  GetNonce(ctx *TxContext) (*big.Int, error)
  ReleaseNonce(ctx *TxContext) error
  CloseNonce(ctx *TxContext) error
}
```

To sum it up...



The primary strategy of architecture is the drawing of hard boundaries between high level policy and low level detail such that the high level policy is entirely ignorant of the low level detail.

(Uncle Bob Martin, Twitter)

https://twitter.com/unclebobmartin/status/1443556534131245059


Máté Láng
github.com/matelang
linkedin.com/in/matelang
Thank you for your kind attention!

You should only do microservices if you...

- Know the problem domain very well
- Really, know the problem domain exceptionally well
- Are familiar with caveats of distributed computing and can embrace eventual consistency
- Are comfortable building multiple optimized data models for your use case
- Have a good amount of automation or are comfortable adding it
  - Testing
  - Scaffolding
  - Dev Environments
- Can handle the added operational complexity
- Have good enough observability and good insights