##Building Microservices Ch2##

How to model microservices

- need to change microservices independently
  - ↳ like modular decomposition but w/networked interaction b/w models
  - ↳ can rely on other modular decomposition tips to set boundaries around microservices

① Information Hiding

desire to hide as many details as possible behind a boundary/module

+ improved dev time
+ comprehensibility
+ Flexibility

"The connections b/w modules are the assumptions which the models make about each other" David Parnas

So keep your assumptions small!

② Cohesion

"the code that changes together, stays together."

aiming ~~to~~ for strong cohesion

- want to find boundaries w/in our problem domain that ensure related behavior is in one place & communicate w/other boundaries as loosely as possible

③ Coupling

want ↓ coupling

* knows as little as it needs about the services it uses
* ↓ # calls b/w services

Coupling + Cohesion

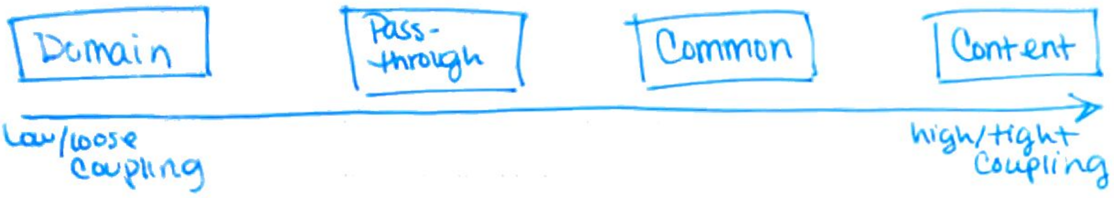
"A structure is stable if cohesion is strong + coupling is low" Larry Constantine

Cohesion applies to relationship b/w things inside a boundary (microservice)

Coupling describes the relationship b/w things across a boundary

Types of Coupling

Some coupling is unavoidable but we can minimize it



Domain Coupling (˙ᵕ˙)

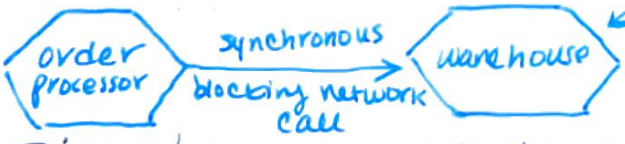
when one μservice needs to interact w/another b/c it needs the other μservice's functionality

- this is unavoidable b/c μservices need to collaborate & work together
- if you see a μservice w/too many outgoing connections it may imply that a μservice is doing too much & too much logic is centralized

* Share only what you have to & absolute minimum data you need

temporal coupling is when concepts are bundled together purely because they happen at the same time

* in μservices it's when one μservice needs another to do something at the same time for the operation to complete



if warehouse is unreachable the call fails.
This is a form of temporal coupling

It's not bad, but need to stay aware & only make blocking calls when you need to. Otherwise, utilize async messaging w/ message brokers.

Pass-Through Coupling (˙-˙)

when one μservice passes data to another μservice b/c the data is needed by some other service further downstream.

- if that data needs to change, now 3 μservices need to change too
- you could move the logic into intermediary, talk directly to downstream μservice, or let the intermediary ignore the data

Common Coupling (·∩·)

when 2(+) μservices use a common set of data

- Could be shared database, memory, filesystem, etc
  - if the data is static it's not a terrible idea, but frequently changing is no good
  - * changes to the structure of the data can impact multiple μservices
- You can make a finite state machine w/ enforced rules on what transitions are allowed & what transitions aren't allowed
  - ↳ or establish an owner of the state changes that can reject outside requests.

* if you have μservice that looks like a thin CRUD wrapper it's a sign that you have ↓ cohesion & ↑ coupling because a service should be able to handle its own data
  * can also be a source of resource contention & is a bad sign.

Content Coupling (`∩´) avoid

when an upstream μservice reaches into the internals of a downstream μservice & changes its internal state.

→ the lines of ownership becomes less clear & its so hard to change a system

* it's important to have a clear separation b/w what can be changed freely & what cannot.
  - ↳ you need to know when you are changing functionality that is a part of your contract
  - → Some people refer to it at pathological coupling

Domain Driven Design - Eric Evans

Domain Driven Design (core concepts)

* useful for defining contracts + splitting services

- Ubiquitous language, map rich domain language of product owner to code
  Common language in code + in describing the domain
- Aggregate
  Collection of objects that are managed as a single entity
  Typically referring to real-world concepts w/a lifecycle.
- Bounded Context
  An explicit boundary w/in a business domain that provides functionality to the wider system but that also hides complexity

1 μservice $\xrightarrow{\text{manage}}$ 1 aggregate
↳ maybe manage multiple aggregates, but one aggregate shouldn't have > 1 μservice

Something that has: (aggregate)
state, identity, + life cycle managed by the system
typically refer to real-life concepts
the aggregate can say no to outside requests

* need a way to model aggregates that cross μservices
↳ can store URI's that are another μservice's endpoint to ~~explit~~ explicitly state this relationship
↳ can also construct your own reference than can be parsed to call another μservice if you're not using REST endpoints

Domain + Bounded Contexts

Our domain is everything we do at REACH, though we may not model that all in code

Hidden Models
the internal + external version of a model can be different.
the shared model can choose to hide unrelated information.

Shared Model
two μservices can have information about the same thing but from their own perspectives. They may still need to reference a shared global model.

EventStorming
Alberto Brandolini

Mapping Aggregates + Bounded Contexts to μservices

"The aggregate is a self-contained state machine that focuses on a single domain concept... with the bounded context representing a collection of associated aggregates ... with an explicit interface to the wider world"

- Both aggregates + bounded contexts can work as service boundaries
- Start w/ coarser-grained bounded contexts then decompose down to find the right seams

* if you decide to split a service later on, no one has to know. This decision is hidden from the outside world as an implementation decision that can change again later.
↳ this could also help w/ testing

Event Storming (Alberto Brandolini)

To help surface a domain model, bring together technical + nontechnical stakeholders to create a shared, joined-up view of the world
↳ You can use this model to construct an event driven model or a more request-response model

1. get everyone in the same room
   representatives of all parts of the domain
2. Find a way to make the activity engaging, put paper on walls, have it be dynamic, remove chairs but make it accessible. Have colored sticky notes
3. have participants identify the domain events (things that happen in the system that you care about. Use one color here.
4. identify commands that cause these events, this is a decision by a human to identify human interaction (blue)
5. identify potential aggregates (yellow)
6. cluster ~~aggressive~~ ~~aggregates are~~ events + commands around aggregate
7. group aggregates into bounded contexts (commonly follow the org. structure)

Vaugn Vernon
implementing DDD
DDD distilled

*Do whats right vs following dogma

Why is DDD helpful for μservices?

① bounded contexts are explicitly about hiding information
- presenting a clear boundary to the wider system while hiding internal complexity that can change w/o other impacts
- this is vital in helping to find stable μservice boundaries

② defining a common, ubiquitous language is vital for μservices + simplifying language in code
↑ understanding = ↑ empathy

DDD is just one technique.

Volatility-based decomposition

identify the parts of your system going through frequent change. Helpful in conjunction w/other techniques
* the goal determines the most appropriate mechanism

Data

* may want to separate code that handles PII vs PCI concerns, etc to minimize auditing concerns + protect the customer
* The in-scope zone needs to be inaccessible to μservices that don't require sensitive data

Technology

* if you need to use multiple runtimes, for example. but be careful w/this

Organizational

* how we organize our teams is how we'll organize our code so this should be taken into account
* shared ownership of a μservice is not a good idea, there needs to be a clear (and singular) owner team
* may need to adjust our organization to fit our architecture
* careful when splitting team + responsibilities across time zones. Make sure you're still slicing business functionality

Layering inside a μservice is ok if it helps w/ the code. But horizontal layers for μservice + ownership boundaries is no good