## ## Building Microservices ##

Foundation
what, how, split, Communicate

Implementation
comm, workflow, build, deploy, test, observe, secure, resilience, scale

People
UI, org, architect

---

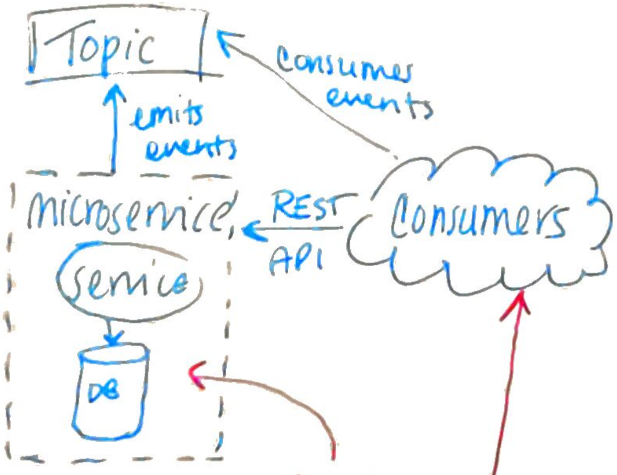
Chapte 1, what are microservices?

Microservices are independently releasable services that are modeled around a business domain

+ technology agnostic
+ type of service-oriented architecture
+ independent deployability is key
+ a black box to consumers
  only access through a REST interface

information hiding: hiding as much information as possible inside a component + exposing as little as possible through external interfaces.

↓ coupling ↑ cohesion



changes here don't affect as long as the interface doesn't break backwards compatibility.

Hexagonal architecture pattern:
- Alistair Cockburn
- importance of keeping internal implementation separate from external interfaces
- you may want use the same fxnality over different types of interfaces

# Building microservices ch1 #

Key Concepts

① Independent Deployability: can make a change to a ~~µservice~~ µservice, deploy & release it w/o having to deploy any other µservices. This is actually how you do it.

* get into the habit of deploying & releasing changes to a single microservice into prod w/o having to deploy anything else
↳ keystone habit!
We need explicit, well-defined & stable contracts b/w services.

② Modeled Around a Business Domain: use Domain Driven Design concepts to define service boundaries. makes it easier to roll out new functionality & recombine µservices in different ways.

want to make cross-service changes as infrequently as possible

services are end-to-end slices of business functionality

* with µservices we have made a decision to prioritize ↑ cohesion of business functionality over high cohesion of technical functionality.

③ Owning their own state: give the microservice the ability to ~~sta~~ decide what is shared vs. hidden.
If you need information from another service, ask. That way we can control what can change frequently vs. infrequently.

* don't share databases!

End-End slice of business functionality that (where appropriate) encapsulates UI, business logic & data

④ Size, perfect size to fit in your head: Don't worry about size too much, ~~Consise~~ consider these Q's first: ① how many microservices can you handle? ② how are your microservice boundaries defined?

⑤ Flexibility: µservices buy you options, they have a cost & we have to decide if the options are worth the cost.

Adopting µservices is like turning a dial. ↑µservices ↑flexibility ↑pain

"By turning up the dial gradually, you are better able to assess the impact as you go & stop if required"

⑥ Alignment of Architecture & Organization: Conway's law, structure stream-aligned teams to reflect the slices of business functionalit in the µservice

Monolith: when all functionality in a system must be deployed together

Single process monolith: all code deployed in a single process

modular monolith: 1 process - ∞ modules, modules can be changed independently but all need to be deployed together

distributed monolith: system w/ multiple service that must be deployed together

+ simpler deployments, troubleshooting, monitoring & testing

+ Simplified code reuse, simpler choices

→ look for problems as you scale & then for technology that can help

→ previous distinctions b/w logical & physical architecture can be problematic we'll need to understand both worlds

##Building microservices##

### Tools

#### Log Aggregation + Distributed Tracing

+ Correlation ID: single ID used for a series of related service calls

★ investigate Lightstep + Honeycomb

#### Containers + Kubernetes

- Containers provide isolation for our services
- With ∞ Containers, they'll need to be orchestrated (K8s)

#### Streaming

- share data w/o monolithic databases
- organizations are moving towards real time feedback
- ↑ Apache Kafka bc. of message permanence, compaction + scalability

↳ stream processing w/ KSQLDB, can also use Apache Flink or Debezium

#### Public Cloud + Serverless

- Google Cloud, MSFT Azure, AWS
- (serverless) message brokers, storage solutions + DBs
- FaaS: Function as a Service

### Advantages of µservices:

- more opinionated in the way service boundaries are defined
- information hiding + DDD + distributed system = ↑ gains

#### Technology Heterogeneity

- multiple collaborating µservices can decide to use different technologies inside each one
- can pick the right tool for each service
- you can embrace the technology that makes sense
- can adopt new technology + advancements quicker, can limit the risk of trying something new

↳ you can choose to limit ex. Netflix/Twitter restricted to JVM langs.

- easier upgrades, less risk

##Building microservices##

Advantages, cont'd:

### Robustness

- making sure failures don't cascade
- Service boundaries become obvious bulkheads
  ↳ note that new failures will need to be handled
  ex. Networks & machines can & will fail, need to handle this reality to ensure robustness holds

### Scaling

- scale the services that need to be scaled

### Ease of Deployment

- ↓risk + ↓fear = ↑deployments & ↓changesets

### Organizational Alignment

- can align your organization w/ your architecture and have smaller, more productive teams
- can change ownership easily

### Composability

- functionality can be consumed in different ways for different purposes
- need architectures that can keep up with holistic needs of customer engagement
- our APIs are flexible seams that can be opened up

Pain Points

### Developer Experience

↑services can lead to ↓dev experience

- JVM can limit the # of µservices that can run on a single machine
- what do you do when you can't run the whole system on one machine?
  ↳ can ↓scope or ↓ability to develop locally
  ↳ could be a problem if any dev should/would work on any part of system

### Technology Overload

- need to balance breadth & complexity of tech against the costs that a diverse array of technology can bring
- manage data consistency, latency, service modeling, etc. introduce tools as you need them

Distributed Systems
Observability
- Cindy Sridharan

##Building Microservices##

Pain points contd.

Cost

- ↑processes, ↑computers, ↑network, ↑storage, ↑software
- learning slowdown while people learn the tools

Reporting

- ↑ difficulty gathering holistic info & reports because data is scattered across ∞ databases
- Can either stream data or centralize your data

Monitoring & Troubleshooting

- do we understand what would happen if a single service is down?
- how do we know when it's important to wake someone up?

Security

- more work is being done over a network & therefore is more vulnerable

Testing

- need to balance

  the more you test, the more confident you are

  but

  ↑scope, harder to setup test data & fixtures, longer it takes to run, more difficult to work out what went wrong
- as microservices grow there is a diminishing return on end to end tests, won't give same confidence

  ↳ lead us to contract-driven testing or testing in production & progressive delivery techniques like parallel runs or canary releases

Latency

- make a small change & measure the impact
- have an understanding of what acceptible latency is for a given action

Data Consistency

- may need to move from using transactions to using

  ↳ sagas & eventual consistency
- requires a fundamental change in how we think about data

Domain Driven Design
Eric Evans

##Building Microservices##

Who should use microservices?

*get your architectural + organizational boundaries right!

- Stable domain (kind of)
- large enough team to handle complexity
- handle their own deployment & management of their own software
- ↑the # of people working on the same system at the same time
- SaaS, 24/7 systems
- want to provide services to customers over many channels