# Overview

Microservices are small be **independently deployable** services that work together, modelled around a business domain

* Considered service-oriented architecture (SOA)
* Make independently deployable your focus, other things will fall into place
* Carefully define the boundaries of your business domains
* Do not model around technical tools

## Defining Size

Signs a module is too big

* New members cannot get onboard quickly
* Team members cannot individually work on different components
* High cost to change or implement new features
* Slow builds

Size is always relative to the context

The amount of services will rise with lower costs to add a new one, and decrease if this cost is high

There is always a tradeoff

* Larger services are more complex, but there are less of them
* Smaller services are more granular, but there are more resources required to support them

When starting off, get a few services in production and evaluate how to approach creating more services

## Downsides of Microservices

Become concerned with communication between services

More complexity in finding things

Troubleshooting is more difficult - monitoring and diagnostics are more complicated

* *“We replaced our monolith with microservices so that every outage could be more like a murder mystery”*

Data consistency can be an issue

Latency increases

Deploying the entire system is more difficult

End-to-end integration testing is more difficult

Security is more complex

## Advantages of Microservices

Ability to scale things independently

Should be able to deploy quicker and more often

More focused codebases

Developers are not in each other’s way as often, so you can “throw” more people on the problem

Testing individual services is easier

Adding enhanced security on specific services is allowed

Code reusability is increased

**Most of these can be achieved without microservices. If your goal is to ship more quickly, try these things independently before switching to microservices.**

## When NOT to implement microservices

1. Greenfield projects – you don’t know what you are doing. If you get service boundaries wrong, you drastically increase cost of change. You don’t have a thorough understanding of the business domain. Startups should focus on finding a market fit because that is more important than great architecture to scale something the market may not even want.
2. Software stored and managed by your customer - you’ve increased the complexity required for the customer.
3. You have no idea why you are using microservices - your customer does not care about your architecture. You must have a specific goal in mind or you are flying blind. You must be able to answer, “what benefit is your customer going to see in result of this architecture?”.

## Architecture guidelines

There is no such thing as an anti-pattern, patterns add value based on context

Different architecture styles are optimized for different purposes

Microservices can vary in implementation due to business needs and optimizing certain needs

## Deciding how to structure the team

Conway’s law – team tend to form in the same manner as their business. Organizational structure is reflected in the system it produces.

Martin Fowler’s First Law of Distributed object design – Don’t just distribute your objects

**3-Tired architecture team structure – thin, horizontal slices**

Popular because it matches the organization of many businesses

* Presentation Tier
  + Front end developers, Javascript heavy
* Business Tier
  + Developers, middle-tier languages
* Data Access
  + DBAs, SQL

**Feature-based team – thick, vertical slices**

Team focused on the end-to-end delivery of features

Team consists of everyone needed to deliver the feature

Reduces handoffs, but adds contention

**Microservices team – thin, vertical slices**

Focused on end-to-end delivery of sub-products

# Defining module boundaries

Services exist to provide capability to other services

Consumer focused mindset - boundaries are driven by a public view of what consumers of your service will see

Encapsulation is important

Only expose what you need to in your service interfaces

Service contracts should be small as possible

Serializer should have to ability to hide certain transferrable fields

## Domain-driven design

Bounded context – breaking things into smaller pieces

## Event storming

technique for creating boundaries, a collaborative exercise involving technical and non-technical stakeholders

1. Identify events
2. Identify commands that trigger those events, example is a decision made by a user
3. Group commands and events into aggregates (i.e. pieces of a state machine)
4. Group aggregates into bounded contexts

Event – something that happens in the system

Command – something that triggers the event

Aggregate – group of commands and events

* Find groupings first, then name the aggregate
* Some events span aggregates

Bounded contexts – groups of aggregates

# Refactoring Monoliths

## Strangler pattern

1. Asset capture – identity the functionality to move to a new microservice
2. Redirect calls – intercept calls to old functionality, and redirect to the new service
   1. Might be copy and paste
   2. More likely a total or partial rewrite

## Data

Separate schemas, same database engine

* Can host multiple schemas on the same db instance
* Littler or no extra infrastructure required
* Potential single point of failure

Solution? Stop adding to the problem – new data need to go into it’s own db

# Communication between services

## Performance

Calls across the network is expensive

Per-call service overhead is very high compared to just calling a method

Data moved by marshalling, rather than just being passed by reference

Tailor the service to call them the least amount of times

## Request types

Synchronous – block and wait, simple and straight forward

Asynchronous – fire and maybe forget, great for long-running jobs, lower latency

## Collaboration Style

Request/ Response – initiate request, expect a response, synchronous or Asynchronous

Event-based – things happen, things react, Asynchronous

Choose the one that fits your architecture



# Presentation Resources

https://bit.ly/ms-workshop-2018

<https://www.dropbox.com/sh/n0h3vjf2mhudz2d/AAByTaiTWlCaBQhwDR3fMkkIa?dl=0>

Books:

Building Microservices by Sam Newman

Domain-Driven Design by Eric Evans

Domain-Driven Design Distilled by Vaughn Vernon