Microservices

Definition

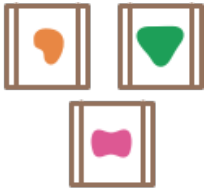
> **Microservice** architecture – a variant of the service-oriented architecture (SOA) structural style – arranges an application as a collection of loosely coupled services. In a microservices architecture,
>
> - services are fine-grained and
> - the protocols are lightweight.

source: https://en.wikipedia.org/wiki/Microservices

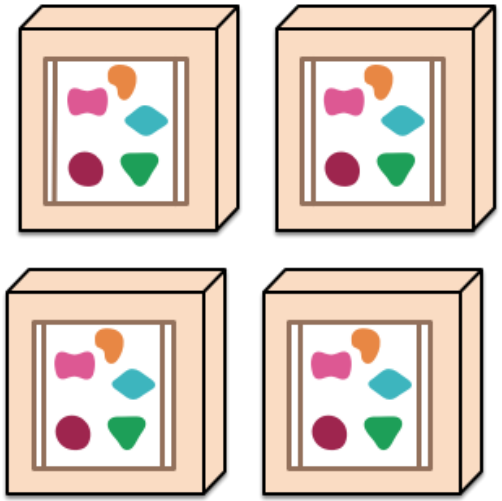
Monoliths and Microservices

A monolithic application puts all its functionality into a single process...

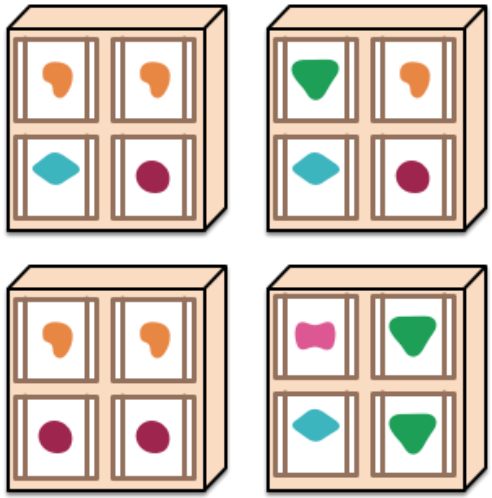
A microservices architecture puts each element of functionality into a separate service...



... and scales by replicating the monolith on multiple servers



... and scales by distributing these services across servers, replicating as needed.



Service-Oriented Architecture

Service-oriented architecture (**SOA**) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. A SOA service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online. SOA is also intended to be independent of vendors, products and technologies.

A service has four properties according to one of many definitions of SOA:

1. It logically represents a business activity with a specified outcome.
2. It is self-contained.
3. It is a black box for its consumers, meaning the consumer does not have to be aware of the service's inner workings.
4. It may consist of other underlying services.

SOA vs microservices

- Heavyweight vs. lightweight technologies
  - SOA tends to rely strongly on heavyweight middleware (e.g., ESB), while microservices rely on lightweight technologies
- Protocol families
  - SOA is often associated with web services protocols (SOAP, WSDL, and WS-* family of standards)
  - Microservices typically rely on REST and HTTP
- Views
  - SOA mostly viewed as integration solution
  - Microservices are typically applied to build individual software applications

Microservices: benefits

- Increase software agility
  - Microservice: independent unit of development, deployment, operations, versioning, and scaling
  - Exploit container-based virtualization: popular approach is “microservice instance per container”
- Faster delivery
- Improved scalability
- Greater autonomy

Microservices and containers

- Microservices as ideal complementation of container-based virtualization
  - Package each service as a container image and deploy each service instance as a container
  - Manage each container at runtime (scaling and or migrating it)
- Pros:
  - Service instance scaling out/in by changing the number of container instances
  - Service instance isolation
  - Resource limits on service instance
  - Build and start rapidly
- Cons:
  - Need container orchestration to manage multi-container app

Application decomposition

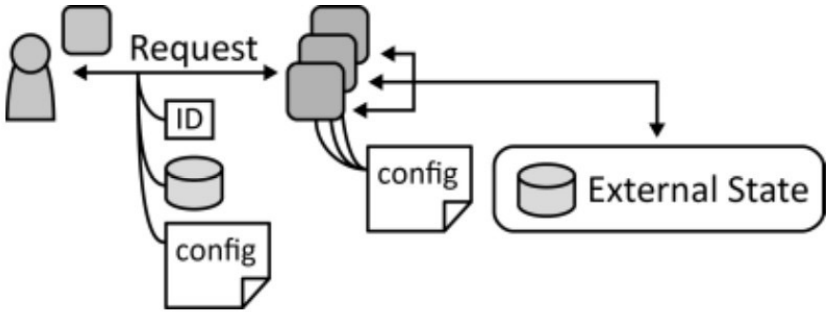
- How to decompose the application into microservices?
- Mostly an art, no winner strategy but rather a number of strategies:
  - Decompose by business capability and define services corresponding to business capabilities. E.g., Order Management is responsible for orders
  - Decompose by domain-driven design (DDD) subdomain. E.g., Order Management, Inventory, Product Catalogue, Delivery
  - Decompose by use case and define services that are responsible for particular actions. E.g., Shipping Service is responsible for shipping complete orders
  - Decompose by nouns or resources and define a service that is responsible for all operations on entities/resources of a given type. E.g., Account Service is responsible for managing user accounts

Microservices and scalability

- How to achieve service scalability?
  - Use multiple instances of same service and load balance request across multiple instances
- How to improve service scalability?
  - State is complex to manage and scale
  - Stateless services scale better and faster than stateful services
- We also need service discovery
  - Service instances have dynamically assigned network locations and their set changes dynamically because of autoscaling, failures, and upgrades: we need a service discovery

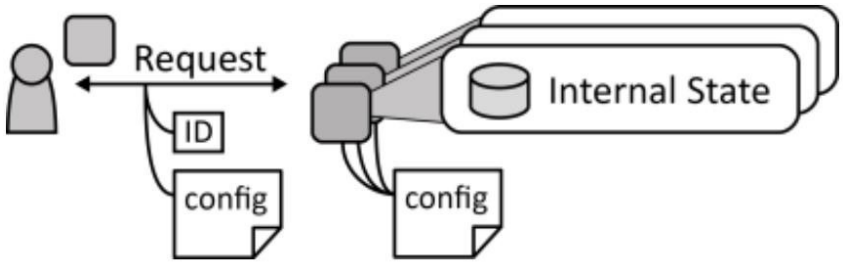
Stateless vs. stateful service

Stateless service: state is handled external of service to ease its scaling out and to make the application more tolerant to service failures



Source: “Cloud Computing Patterns”, https://bit.ly/2SekKo6

Stateful service: multiple instances of a scaled-out service need to synchronize their internal state to provide a unified behavior



Source: “Cloud Computing Patterns”, https://bit.ly/2yyxHkg

Characteristics of a Microservice Architecture

from a software engineering perspective

see https://martinfowler.com/articles/microservices.html

Characteristics of a Microservice Architecture

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

Componentization via Services

A component is a unit of software that is independently replaceable and upgradeable

Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services.

- **Libraries** as components that are linked into a program and called using in-memory function calls
- **Services** are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

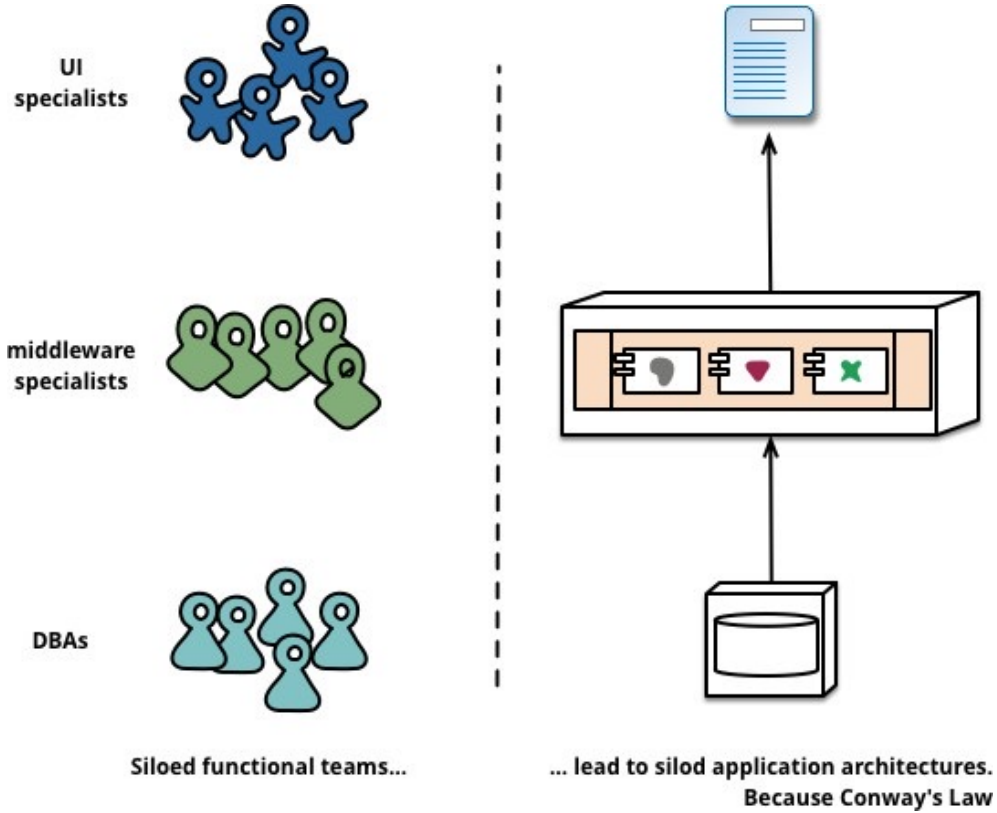
Organized around Business Capabilities

Conway's Law

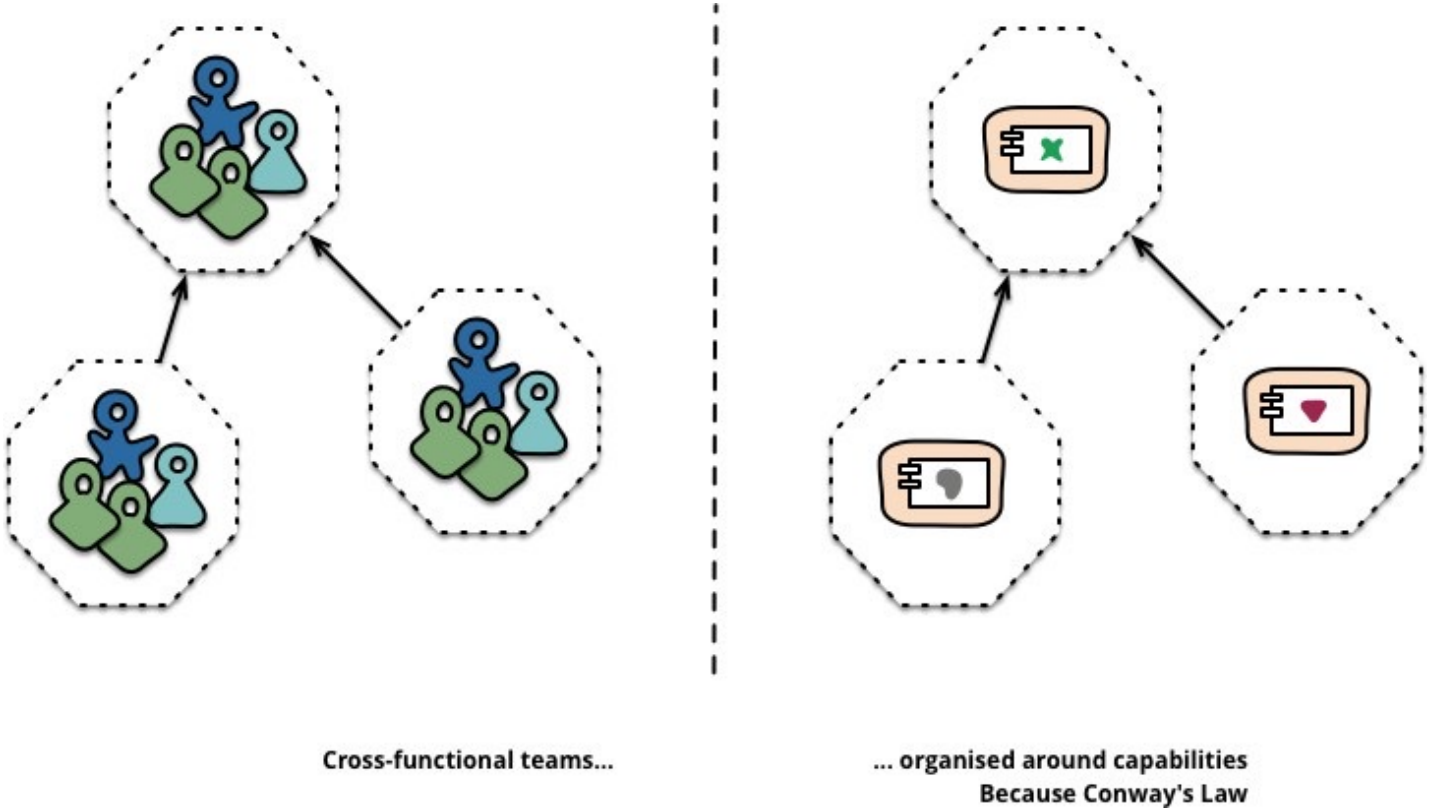
Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

-- Melvin Conway, 1968

Organized around Business Capabilities



Organized around Business Capabilities



Cross-functional teams...

... organised around capabilities
Because Conway's Law

Products not Projects

Most application development efforts use a project model: where the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.

Microservice proponents tend to avoid this model, preferring instead the notion that a team should own a product over its full lifetime.

Amazon's *"**you build, you run it**"* = development team takes full responsibility for the software in production

Smart endpoints and dumb pipes

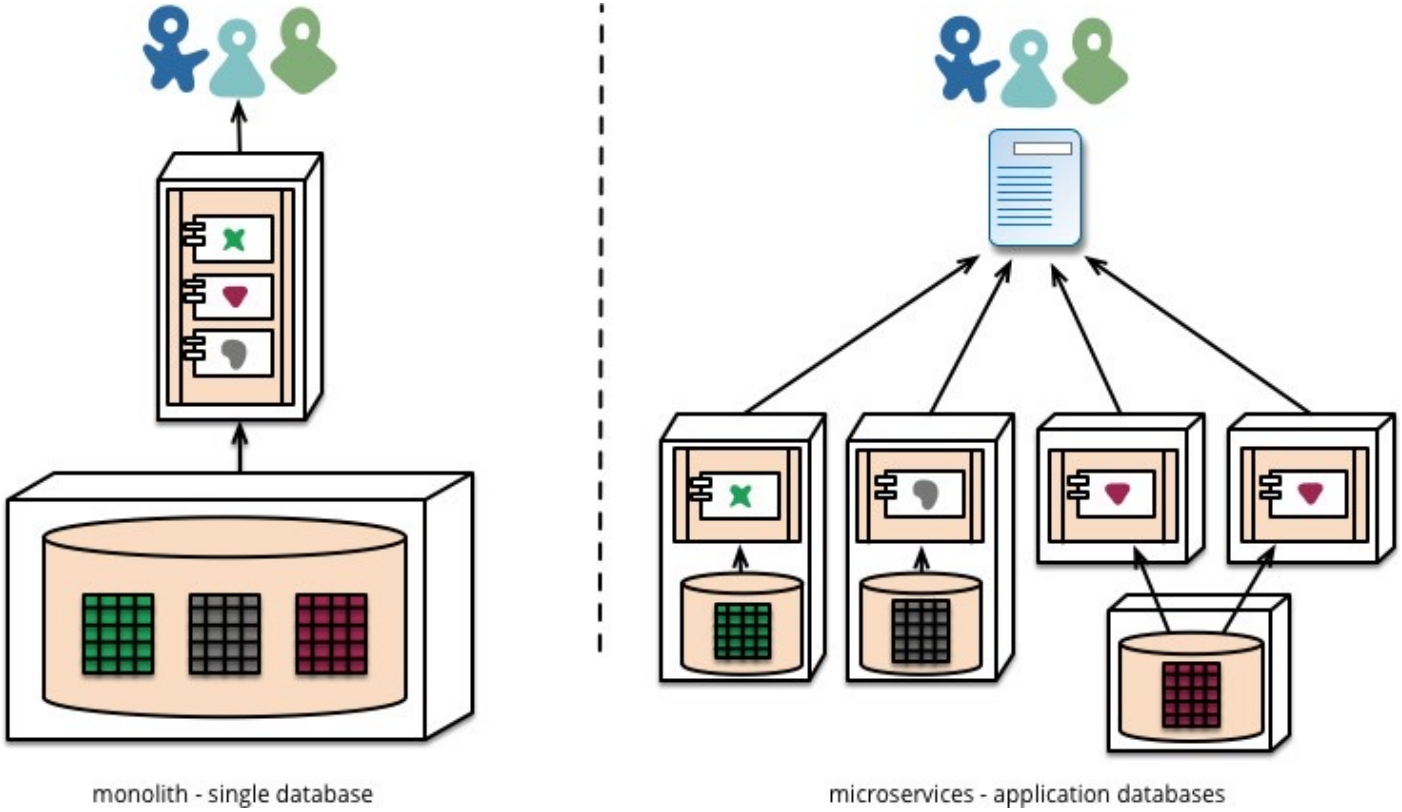
When building communication structures between different processes, we've seen many products and approaches that stress putting significant smarts into the communication mechanism itself.

The microservice community favors an alternative approach: *smart endpoints and dumb pipes*. Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response.

Decentralized Governance

- One of the consequences of centralized governance is the tendency to standardize on single technology platforms.
- Splitting the monolith's components out into services we have a choice when building each of them. You want to use Node.js to standup a simple reports page? Go for it. C++ for a particularly gnarly near-real-time component? Fine.

Decentralized Data Management



Design for failure

- A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services. Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it.
- Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service. Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received)

Evolutionary Design

- Microservice practitioners, usually have come from an evolutionary design background and see service decomposition as a further tool to enable application developers to control changes in their application without slowing down change.
- The key property of a component is the notion of independent replacement and upgradeability - which implies we look for points where we can imagine rewriting a component without affecting its collaborators.

Reading list

J. Lewis, M. Fowler, “Microservices” (2014)
https://martinfowler.com/articles/microservices.html

(documents available on Blackboard)

Questions, comments?