Microservices

- Distribution
- Scalability
- Agility

N-Tier monolithic
Service oriented arch (SOA)
Microservices

Benefits
Risks
Compromises for any choice

Services
Comms
Distribution
DB / transactions
API Layer

asynch comms
tracing / logging
CD
Hybrid arch.

design considerations
managing trade offs
Edge services
devops culture

Basics
- Remote n/w calls over Http
- Frameworks / patters for RESTful services

Microservices

- Agile architecture
- Breaking Bigger problems into smaller ones
- no size constraints
- uses REST endpoints
- communication is important
- can be implement with open source
- agility and distributability

Resources for testing / deployment for monoliths

SOA

- more recoverable errors than just OK / Error
- aggregation layers @ SOA Bus
- Strong contract WSDL
- Envelope of SOA messages
- wiring using SOA Bus become complex

Costs of using microservices

- complexity
- time / money as you increase deployed artifacts
- determining where the code resides impacts costs
- its important to trim complicated processes and increase automation
- ★★★ distribution tax (nwk latency / congestion / thread blocking)
- reduction of reliability (evaluation of core services / whether they can withstand unreliability)

Fitment

- Single code

★ course

Base
- Self contained
- file system (eg. S3, etc)
- VM to cloud native movement

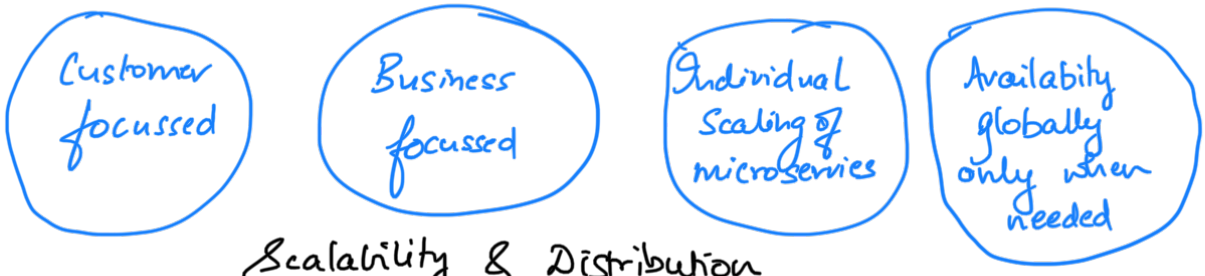
Cloud native
- 12 factor app development
- cloud infra / public - private - hybrid
- globally distributed (scalability / increased uptime)

Microservices
- fit nicely into cloud native paradigm

MICROSERVICES ATTRIBUTES & CRITERIA FOR DESIGN

- Communication is REST over HTTP
- developing common service documentation is critical
- size is not as critical as operations
- service operates on one specific domain ⇒ not on data or business objects
- low level data focussed services ⇒ expose CRUD ops across multiple data objs
- services handling multiple business processes across domains
- Sizing should ensure minimization of distribution tax ☆☆☆ (latency of calls will hurt)
- criteria for size : faster builds / fewer tests / deploy-startup faster
- Interservices communication specifics
- Protocol aware heterogeneous interoperability (bound to protocol)
- Polyglot style of development
- Passivity of APIs and versioning strategy is important to not cause issues with the communicating components

- analogy to OOP
- classes do one thing and provide operation to access data & functionality



Scalability & Distribution

- Elastic scalability

★ Gridlocks ★

- Latency & failures due to multiple latency issues
- circular calls

i) circuit breakers with default workflow
ii) timeout behaviors

Designing ms Systems

- [illegible] ⇒ determine domains ⇒ break up Services

Investigate → determine domains → break up services
- achieving the right amount of granularity
- bounded context for domains / find with data
- deeper analysis to build for latency betterment
- stronger contracts between boundaries

Data Layers / domains

* Cannot avoid DB transactions
* Leveraging distributed transactions is difficult

- how do we decompose a monolithic DB?

Breaking monolithic DBs into smaller ones & associate with MS	- making mistakes in design - cost is high - handling this with live systems - migrating data with API Layer is even more difficult.
Build MS that connect to monolithic DBs	- allows to see if domains are well defined - decompose DBs only after properly ascertaining transactional boundaries

Building good data domains are very important

Monolithic model

ACID model (traditional)

- atomic (fails or succeeds)
- consistent (all data model rules are enforced)
- isolated (visibility rules well defined - cant read data that is not in the correct state)
- Durable (persistence guarantee after operation completes)

Services model

BASE model (ms)

- eventual consistency across HA distributed platform
- no isolated consistency expectation
- as long as data isn't modified again we will eventually achieve end state in all nodes of the

Micros

distributed datastore. ✳

API Layer

Very important consideration

- dont do transformative ops within this layer
- aggregated proxy of all your MS
- exposes service endpoints
- pure proxy / isolates clients from knowing the IP address / ports of the services being called
- isolation from changes (versioning)

- Use versioning with API layers
- Still supports older parameters or legacy versions

Async comms

- To reduce latency
- Event driven - microservices
- architecting these comms are important
- Stream data platforms / message brokers
- Sync to async ops (load reduction)
- error state handling and recovery
- queue monitoring / data correctness checks

Logging

- plan for logging strategy across entire platform
- need for quality logging
- different VMs / sessions can confuse investigations
- distributed systems need convergence
- logging aggregators / aggregate logs
- Log trace with tokens (trace Id)

Continuous delivery (for agility)

- Building a CD model

- Microservices have lots of moving parts
- Building committed code base in automated way
- Sandbox, containers, non prod env
- Security pen testing / integration testing
- Start small and expand

Hybrid architectures

- n-tier arch based modeling
- data / business process / edge / gateway
- adjusting logic and flow

Service based arch

- single underlying DB
- doesn't modify datastores
- consider benefits and risks

Architectural choices / design considerations

- CI/CD pipelines
- logging / tracing flow / centralise / aggregate
- domain driven design / control latency
- consider non blocking code
- standardise stacks
- async first design
- less about code - more about ops / infra

Tradeoffs

- well defined module boundaries
- scalability
- deployment complexity
- too many moving parts
- polyglot development comes at a cost
- real need to plan.

Edge services

- outbound (expose client needs to outside world)
- inbound / translation (abstract from 3rd party dependencies
- edge service has abstraction
- vendor changes can be handled well.
- one more hop is introduced
- outbound can send multiple amounts of data depending on consumption needs

Dev-ops

- Engg culture
- Ops and dev in same sphere
- continuous monitoring and automated responses
- automate deployments (agility)
- microservices & devops go hand in hand