# Chapter 1 Microservices

## What are microservies?

Small, autonomous services that work together.

Small enough, no smaller. Focused on doing one thing well.

Autonomous: golden rule: can you make a change to a service and deploy it without changing anything else?

## Benefits

1. Technology Heterogeneity: Use different technologies inside each one. If you can rewrite your microservice in two weeks, you may mitigate the risks of embracing new technology.
2. Resilience: Failure doesn’t cascade
3. Scaling: Scale those services that need scaling
4. Ease of deployment: Make a change to a single service and deploy it independently. Easy to roll back it.
5. Organizational Alignment: Align architecture to organization. Minimize the number of people working on any one codebase
6. Composability: Reuse of functionaility.
7. Optimizing for replaceability: Avoid a big legacy system that no one wants to touch

## SOA

Think Microservices as a specific approach for SOA in the same way that XP and Scrum are specific approaches for Agile.

## Other Decompositional Techniques

Shared libraries

Modules

## Drawbacks

1. Distributed transactions and CAP
2. Deployment and maintenance

# Chapter 2 The Evolutionary Architect

## Inaccurate Comparisons between software architect and building architect

## Evolutionary vision for the architect

1. Software will continue to evolve. Architects need to shift their thinking away from creating the perfect end product, and instead focus on helping create a framework in which the right systems can emerge, and continue to grow as we learn more.
2. Zoning

## Principled approach

1. Strategic Goals
2. Principles
3. Practices
4. Combines the above 3

## Required Standard

1. Monitoring
   1. Matrics: Graphite
   2. Health: Nagios
   3. Logging
2. Interfaces

A small number of defined interface technologies.

1. Architectural safety

Services should shield themselves from unhealthy, downstream calls.

1. Governance Through Code
   1. Exemplars
   2. Tailored Service Template
2. Tech Debt
3. Exceptional Handling

# Chapter 3. How to Model Services

## What makes a good service

1. Loose coupling: a change to one service should not require a change to another. A loosely coupled service knows as little as it needs to about the services with which it collaborates.
2. High cohesion: Related behavior sit together, and unrelated behavior to sit elsewhere. If we want to change behavior, we want to be able to change it in one place instead of lots of places.

## Bounded context

From DDD. Any domain consists of multiple bounded contexts, and residing within each are things that do not need to be communicated outside. Each bounded context has an explicit interface, where it decides what models to share with others.

1. Shared and hidden models
2. Modules and services: start from modules in a monolithic system and then move to a separate service, unless you are very proficient.
3. Premature decomposition
4. Business capabilities: Think bounded context not in terms of data that is shared, but about the capabilities those contexts provide the rest of the domain.
5. Nested bounded context
6. Communication in terms of business concepts: Think of the communication between Microservices in terms of the same business concepts.

# Chapter 4 Integration

## Ideal integration technology

1. Avoid breaking changes
2. Keep your apis technology-agnostic: avoid integration technology that dictates what technology stacks we can use
3. Shared database：fastest to start with. Drawbacks:
   1. Internal implementation exposed
   2. Tied to specific technology.
   3. The same logic applied in many places

## Synchronous VS Asynchronous

1. Orchestration VS choreography
2. Synchronous
   1. RPC
   2. REST
3. Asynchronous event-based collaboration
   1. Message Queue. ATOM and feed
   2. Complexity: Example: catastrophic failover
4. Service as state machines: When a consumer wants to change a customer, it sends an appropriate request to the customer service. The customer service, based on its logic, gets to decide if it accepts that request or not. Aggregated root.

# Chapter 5. Splitting the Monolith

## It’s all about seams

## Reasons to split the monolith

1. Pace of change
2. Team structure
3. Security
4. Technology

## Tangled dependencies

## The Database

1. Breaking foreign key relationship
2. Shared static data: file, service, static code
3. Shared data: wrap it with a service
4. Staging the break
5. Transaction boundaries
   1. Try again later
   2. Abort the entire operation
   3. Distributed transactions

## Reporting

1. Reporting database
2. Data retrieval via service calls
3. Data Pumps
4. Event data pump
5. Backup data pump

# Chapter 7 Testing

## Types of tests

## Test Scope

Test Pyramid

1. Unit Tests
2. Service Tests : Stubbed collaborator
3. End-to-end tests

Mocking or stubbing

Tricky end to end tests

1. Flaky and Brittle Tests
2. Who writes these tests: Shared codebase, with joint ownership
3. How long? Run in parallel, remove tests no longer needed
4. Small numbers of core journeys to test the whole system, not stories
5. Consumer-driven contract (CDC) PACT

Testing after production

1. Separating deployment from release:
   1. Blue/green deployment
   2. Canary deployment
2. Mean time to repair over mean time between failure

# Chapter 8 Monitoring

## Single Service, Single Server

Nagios, New Relic

Logrotate

## Single Service, Multiple Servers

Nogios group our hosts, track host-level matrics like CPU, memory, networking

## Multiple Services, Multiple Servers

Collection and central aggregration of as much as we can get our hands on, from logs to application metrics.

## Logs

ELK, Splunk

## Metrics Tracking Across Multiple Services

Graphite, aggregrate across samples

## Service Metrics

1. Basic metrics like response times and error rates
2. Number of times customers view their past orders
3. Codahale’s Metrics (Metrics) library allows you to store metrics such as counters, timers… Support sending data to graphite

## Synthetic Monitoring

1. Use fake event to ensure the system is behaving semantically. Semantic monitoring
2. Use service or system end to end test to implement semantic monitoring

## Correlation IDs

Generate correlation IDs at the first call and passed along to all subsequent calls

## The Cascade

Each service instance should track and expose the health of its downstream dependencies, from the database to other collaborating services.

## Standardization

Monitoring is one area where standardization is important. Write your logs in a standard format. Have Metrics in one place and have a list of standard names.

## Consider the audience

Supporting team, capacity planning, boss

## Handle operational and business metrics in the same way

Riemann, Suro

# Chapter 9 Security

## Authentication and Authorization

1. SAML (Okta is a SAML based identity provider. LDAP, Active Directory are directory service)
2. OpenID Connect (OAuth 2.0)

## Single Sign-On Gateway

## Fine-grained authorization

Each microservice in question apply the authorization rule

## Service to service authentication and authorization

1. Allow everything inside the perimeter
2. HTTP(S) Basic Authentication
3. Use SAML or OpenID Connect
4. Client Certificates
5. HMAC over HTTP
6. API Keys
7. The deputy problem

## Securing data at rest

1. Go with the well known
2. Pick your targets

## Defense in Depth

1. Firewalls
2. Logging: Detect and recover from bad things happening
3. Intrusion Detection System(IDS): Monitor for suspicious behavior
4. Network Segregation
5. Operation system:
   1. Patch software
   2. Security modules

# Chapter 10 Conway’s law and system design

**Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization’s communication structure**

## Loose and tightly coupled organizations

## Netflix and amazon

Two-pizza teams: small teams owning the whole lifecycle of their services

## Adapting to communication pathways

If the organization building the system is more loosely coupled, the systems being built tend toward the more modular, and therefore hopefully less coupled

## Service ownership

The team owning a service is responsible for making changes to that service

## Drivers for shared services

1. Too hard to split
2. Feature teams
3. Delivery bottleneck

## Internal open source

1. Roles of the custodians
2. Maturity

## Bounded context and team structures

## The orphaned service

## Conway’s law in reverse

## People

Give your developers time to adjust to the new practices

# Chapter 11 Microservices at Scale

## Failure is everywhere

We can do our best to limit the causes of failure, but at certain scale, failure becomes inevitable

## How much is too much

Cross-functional requirements: knowing how much failure you can tolerate, how fast your system needs to be.

1. Response time
2. Availability
3. Durability of data

## Degrading Functionality

Understand the impact of each outage, and work out how to properly degrade functionality. If the shopping cart service is unavailable, we could still show the webpage with the list. Hide the shopping cart or replace it with an icon saying “Be Back Soon”

## Architectural safety measures

An example:

1. Timeout of connection pool not correctly set
2. Share a single HTTP connection pool for all outbound requests

## The antifragile organization

Netflix's Simian Army of failure bots

* Chaos Monkey
* Chaos Gorilla

Handle the failure in the system

* Timeouts
* Circuit Breakers: all further requests fail fast while the circuit breaker is in its blown state
* Bulkheads: Example: isolated connections pools for different services. If one pool get exhausted, the others aren’t impacted.

## Idempotency

## Scaling Applications

1. Go bigger: larger machines
2. Splitting workloads: As microservices are independent processes, it should be easy them onto their own hosts to improve the throughtput and scaling. None critical service doesn’t need to be as resilient as the core service
3. Spreading the risk:
   1. Different services on different hosts
   2. disaster recovery
4. Load balance
5. Worker based system
6. Starting again
   1. Don’t build system for massive scale at the beginning as the traffic may never come
   2. Rapidly experiment and understand what capabilities we need to build
   3. The need to change our system to deal with scale isn’t a sign of failure, but a sign of success

## Scaling databases

1. Availability of Service VS Durability of Data
2. Scaling for reads: Copy data from a primary node to one or more replicas. Use replicas to distribute our reads
3. Scaling for writes:
   1. Sharding: A-M to one instance, N-Z to another
      1. Problems: Query
      2. Adding nodes
      3. Not improve resiliency
   2. Alternative scaling models of Cassandra, Mongo or Riak
4. CQRS

## Caching

1. Client, Proxy and server side caching
2. Caching in HTTP
3. Caching for writes
4. Caching for Resilience
5. Hiding the origin
6. Keep it simple

## Autoscaling

1. Predictive: Scaling by well-known trends
2. Reactive: bring up additional instances when you see an increase in load or an instance failure

## Service Discovery

1. DNS
2. Dynamic Service Registries
   1. Zookeeper
   2. Consul
   3. Eureka
   4. Reports and dashboards for humans

## Documenting Service

1. Swagger
2. HAL and the HAL Browser