Microservice Architecture Blueprint

“The Introduction” Episode 1

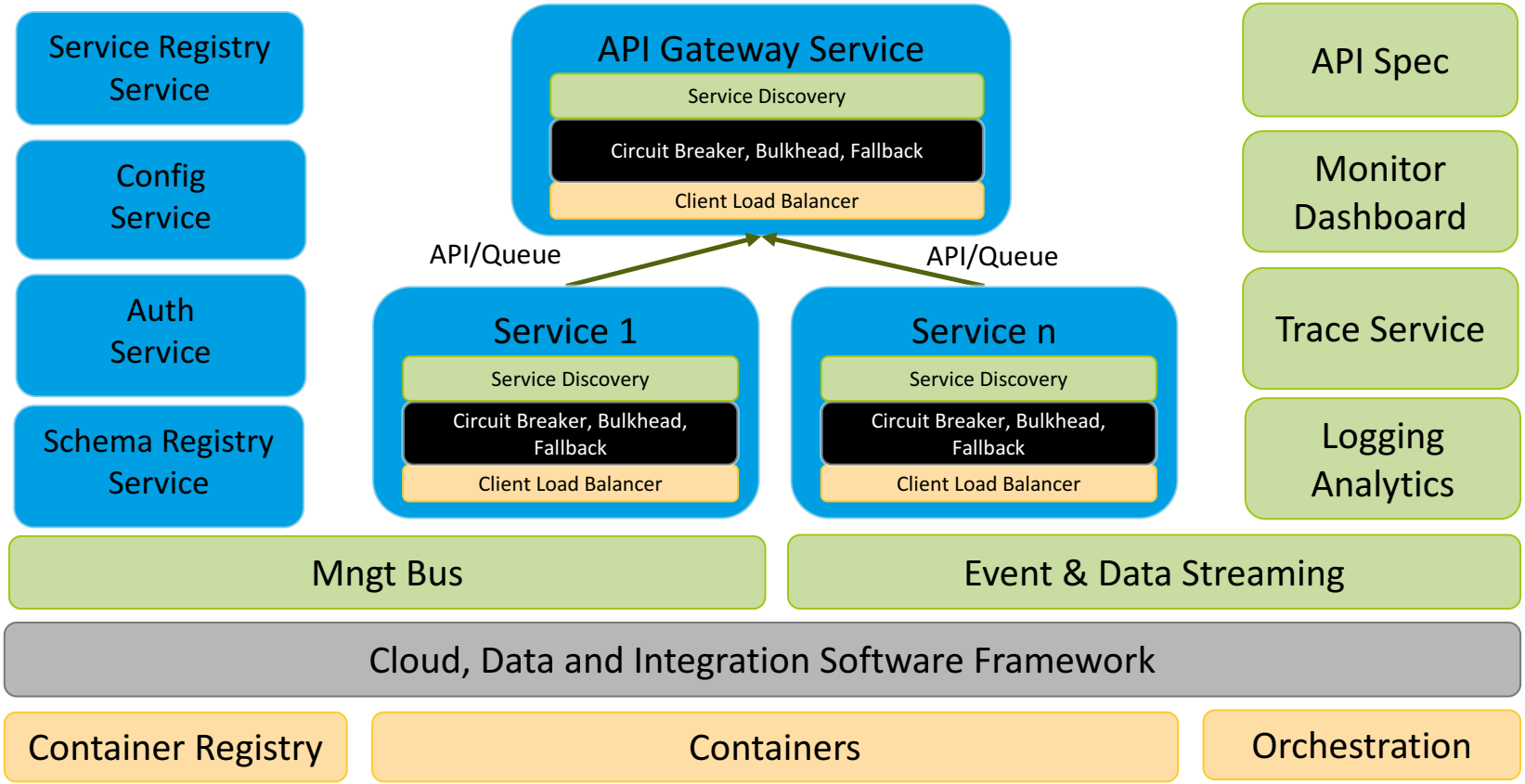
Kim Horn

Version 1.14 **2 August 2017**

Agenda

- Background;
- Microservice Definition - Introduction to Micro Services;
- Decomposition;
- The Problem Space;
- Microservice Patterns;
- Blueprint we are providing to help this implementation;
- Next Steps.
  - Episode 2…

Distributed Cloud Blueprint– NFRs - use what you need



Blueprint Principles

- Based on and Demonstrates the use of Documented Cloud Patterns;
- Provides Implemented Deployed Framework based on working code;
- Based on Trusted, well documented Open Source re-usable components that implement the Cloud patterns;
- Deployable anywhere: data center, AWS, Azure, Google, My Laptop;
- Employs best practices proved by best of breed organisations;
- Integrated and Cohesive Framework of components, that can be used individually or as a set.
- Guidelines to help apply the right patterns and components to meet NFRs.


NETFLIX ZUUL
github.com/Netflix

Microservices Definitions

Lightweight Communications

Implicit in the statement that processes are “communicating with lightweight mechanisms” are the ideals:

- Synchronous connections are direct, socket to socket, using light weight protocols, e.g. HTTP.
  - There are no ESBs or mediators in between;
  - There are no load balancers and routers ‘in between’;
  - Except Boundaries, e.g. organisation, bounded context, where a gateway provides isolation;
  - Extra client work is required to provide reliability;
- Asynchronous connections are event and stream based, using direct and lightweight mechanisms, such as stream pipelines and brokers e.g. Kafka;
  - Provide Reliability, High Scalability and very low coupling;
  - Keep asynchronous communications internal.
- NFRs decide between Synchronous and Asynchronous;

What is a Microservice Architecture ?

A microservices architecture is: “a *service-oriented architecture composed of loosely coupled elements that have bounded contexts*.”

Adrian Cockcroft
Director of Web Engineering Netflix, and Cloud Architect.

“Small Autonomous services that work together, modelled around a business domain.”

Sam Newman

‘Microservices’ as Distributed Systems

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.”

Martin Fowler in March 2014

Microservice Characteristics*

- A microservice is responsible for one **single capability;**
  - Single Responsibility Principle;
- A microservice is **individually deployable;**
  - *makes testing, management and maintenance easier;*
- A microservice consists of **one or more processes;**
  - *must run in a separate process; to do with isolation;*
- A microservice owns its **own data store;**
- A **small team** can maintain a handful of microservices;
- A microservice **is replaceable:**
  - *must be able to be rewritten from scratch within a reasonable time frame;*
  - *has its own code base;*
  - replaceable - both build and run time (fail fast and replace).

* from the Book “Microservices in .Net”

What Amazon said

Amazon doesn't claim to do 'microservices' but rather 'service oriented'. They said:

"For us service orientation means encapsulating the data with the business logic that operates on the data, with the only access through a published service interface. No direct database access is allowed from outside the service, and there's no data sharing among the services."

Werner Vogels

Single Microservices are not Interesting

*"'**Microservice architecture**' is a better term than 'microservices'. The latter suggests that a single microservice is somehow interesting."*

Chris Richardson Oct 2016

“The microservice style of architecture is not so much about building individual services so much as it is making the *interactions between* services reliable and failure-tolerant.

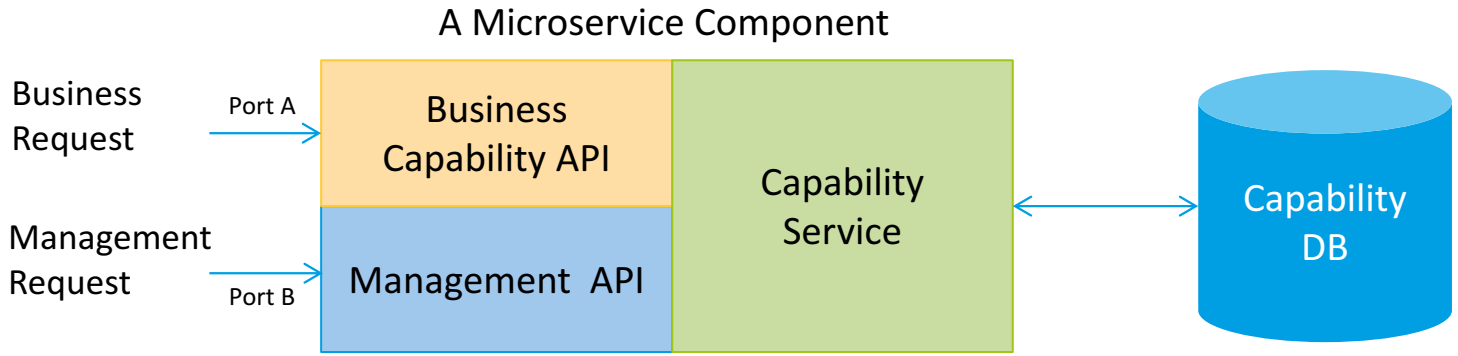
While the focus on these interactions is new, the need for that focus is not. We’ve long known that services don’t operate in a vacuum.

Even before cloud economics, we knew that - in a practical world - clients should be designed to be immune to service outages. The cloud makes it easy to think of capacity as ephemeral, fluid.

The burden is on the client to manage this intrinsic complexity.”

APIs and APIs

A microservice architecture components each exposes a single capability via an API. Each components may have its own DB, dealing with data only relevant to that capability.



- A microservice allows a small team to build, manage and operate their service autonomously.
- The team has minimal dependencies with the rest of the organisation and has the freedom to do the job at hand.
- The team supports a business capability not a technology; so crosses all technology;
- The API hides the implementation details from the consumers of the API.

Microservice Architecture Style

It's a bit more complex than these definitions suggest …..

- **Microservices are not about creating single services**, but are a Style of Architecting Applications; using a set of interacting capabilities; that together provides quality of service.
- **Microservices are not about re-use**, as was Enterprise SOA, its about decomposing a single application. Need to ley go of these 'old' practices;
- Microservice are also not about providing external APIs. In essence they are internal components of an application isolated in its domain.
- They involve a collection of styles including: Event Driven Architecture, REST, Data Streaming and Pipes and Filters.
- Today microservices can be supported by infrastructure components, that are themselves microservices, each focusing on a particular platform infrastructure concern.

Decomposition

Decomposition

"Architecture gives us intellectual control over the very complex by allowing us to substitute the complex with a set of interacting pieces, each one of which is substantially simpler than the whole.

The prudent partitioning of a whole into parts is what allows groups of people— often groups of groups of people separated by organizational, geographical, and even temporal boundaries— to work cooperatively and productively together to solve a much larger problem than any of them would be capable of individually."

"**Software Architecture Documentation in Practice:"**

Bachmann, Bass et al 2000

Modules

Parnas' "On the Criteria To Be Used in Decomposing Systems into Modules" said:

"We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others."

- The point of modularity is not one of reuse, but one of concealment, or abstraction
  - Hiding assumptions from the rest of the program.
- You can ask, how easily an implementation could be grown, deleted, rewritten, or swapped with a different system altogether, without changing the rest of the system.

Modularity is the basis of microservices, with modules that are distributed and communicating over a network.

Separation of Concerns

“We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained —on the contrary!— by tackling these various aspects simultaneously. It is what I sometimes have called **"the separation of concerns"**, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one-and multiple-track minded simultaneously.”

Edsger Wybe Dijkstra

Small Decoupled Services

Unbundling applications into small decoupled services, means we can build systems that are:

- **Flexible -** can be composed and rearranged to quickly deliver new functionality. The smaller the unit of code, the less complicated it is to change and the less time it takes to test and deploy.
- **Resilient -** an application is no longer a single “ball of mud” where a degradation in one part of the application causes the whole application to fail. Failures can be localized and contained, to a small part of the application, and the application can degrade gracefully. in case of an unrecoverable error.
- **Scalable -** easily distributed horizontally, making it possible to scale specific features and services appropriately which is much more cost effective. With a monolithic application where all the logic for the application is intertwined, the entire application needs to scale even if only a small part of the application is the bottleneck.

Jesse Robins - Definition

“Operations at Web Scale is the ability to consistently create and deploy *reliable software* to an *unreliable platform* that scales horizontally.”

Horizontal Scaling = Distributed System

Reliability in distributed systems is determined by the weakest component.

The number one hot spot for serious system failure is network communication and making any assumptions leads to danger.

- Faults will happen;
- Need to stop these cascading.

The Blueprint provides a base set of Patterns that provides system reliability to mitigate these component issues, and allow scaling;

Peter Deutsch and others at Sun listed these eight fallacies of distributed systems:

- The network is reliable;
- Latency is zero;
- Bandwidth is infinite;
- The network is secure;
- Topology doesn't change;
- There is one administrator;
- Transport cost is zero;
- The network is homogeneous.

Isolation, Fault Tolerance and Scalability

If things are unreliable then the most important concern to focus on is Isolation.

- Components are decomposed by not just how they expose functionality but the impact of failure; failure needs to be contained and isolated;
- So when a component fails it does not propagate failure throughout the system:
  - Components should **fail fast**, not linger around in a state of death;
  - Components should **fail early**, communicate their failure, so they can be dealt with quickly and not consume resources;
- When components are isolated they can be run on parallel;
- **Components that are isolated and running in parallel provide scalability.**
- **Architecting for Failure means Recovering Gracefully:**
  - Need to be able to **Monitor health** of components; know early;
  - Need to be able to **Isolate unhealthy things** automatically;
  - Need to be able to **Manage Components**: Start, Stop, Restart, Fix, Replace, Upgrade…

Re-Composition Architecture

After decomposing need to re-compose into a runnable ‘system’.

Three ways to do this:

1. **Build Time Binding - Example ‘Monolith Web Portal’**
   - Create independent ‘small’ service components (Packages) in a library;
   - Combine (integrate) at build into one big blob (WAR, EAR);
   - Deploy the blob; a monolith, into one process;
2. **Run Time Binding**
   - Independently deploy ‘small’ components, as distributed services, in their own processes;
   - Clients discovers location of services dynamically; physical location is hidden;
   - These are Choreographed as an Application.
3. **Combination of the above**

Modularity

Modularity can be boiled down into three guiding principles:

- **Strong encapsulation:** hide implementation details inside components, leading to low coupling between different parts. Teams can work in isolation on decoupled parts of the system.
- **Well-defined interfaces:** you can't hide everything (or else your system won't do anything meaningful), so well-defined and stable APIs between components are a must. A component can be replaced by any implementation that conforms to the interface specification.
- **Explicit dependencies:** having a modular system means distinct components must work together. You'd better have a good way of expressing (and verifying) their relationships.

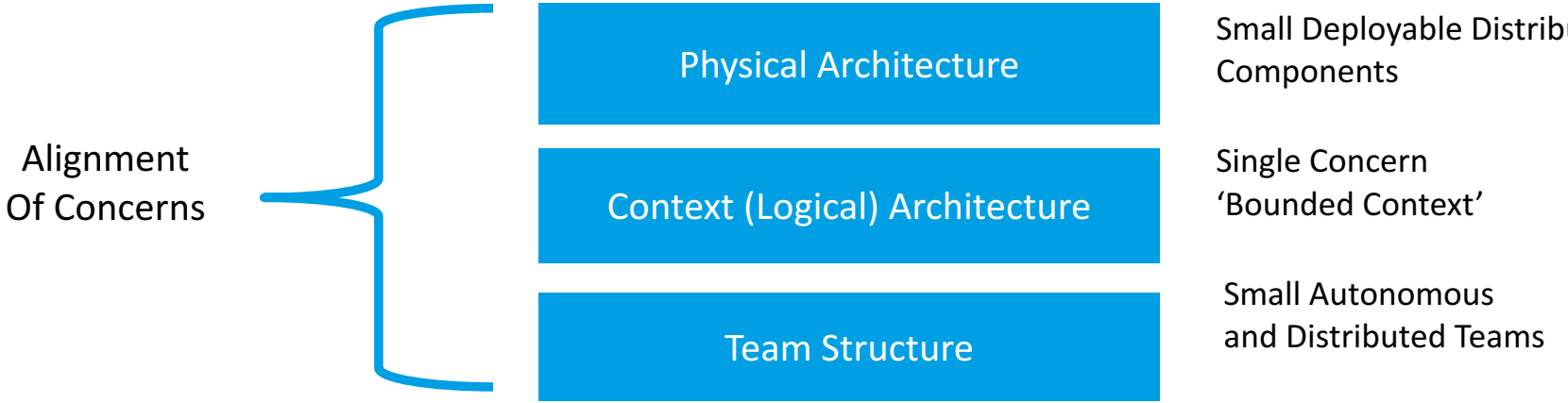
Modules to Microservices

Microservices realise important modularity principles, leading to tangible benefits:

- They are small and focused, reducing complexity.
- Services can be internally changed or replaced without global impact.
- Teams can work and scale independently.
- Modules are natural units for code-ownership.
  - Teams can be responsible for one or more modules in the system.
  - The only thing shared with other teams is the public API of their modules.

Inverse Conway's Law

Netflix designed the org structure for the system architecture it wanted.

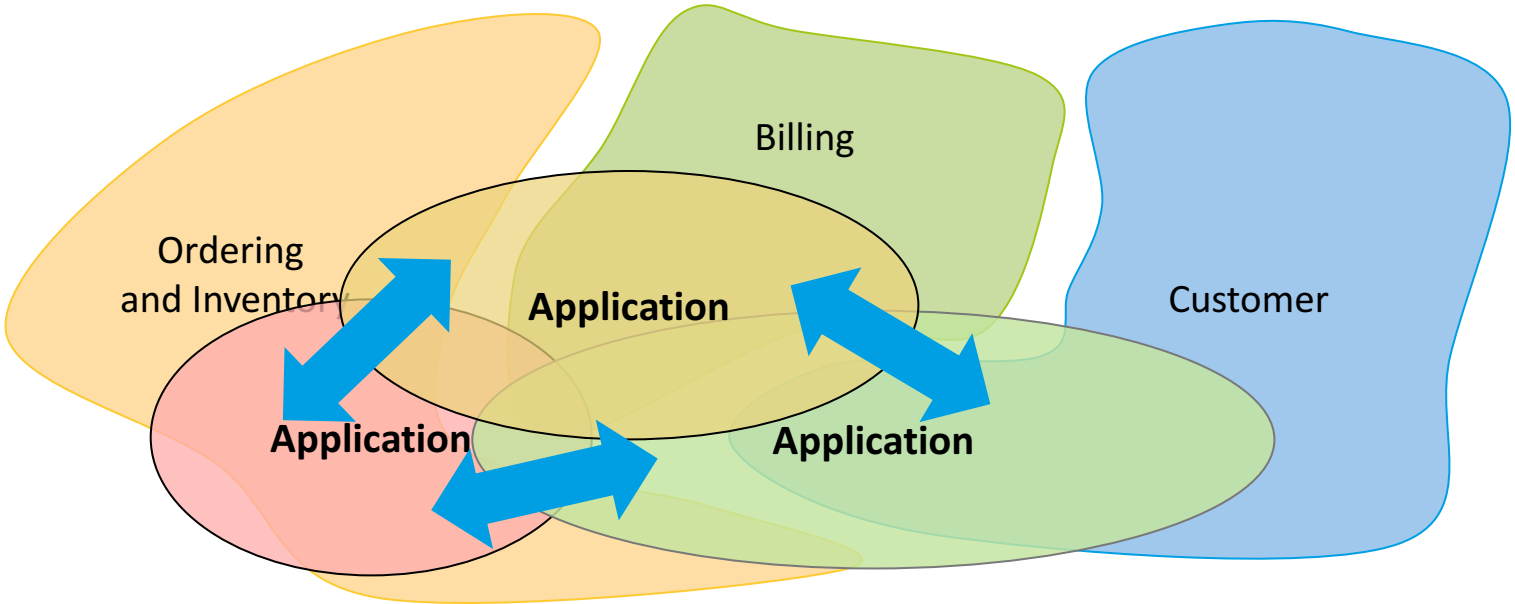


Evolutionary Architecture – incremental change and decoupling means it is easy to replace or rewrite a service by a small team.

The Problem Space

Current State - Concepts and Code Leak

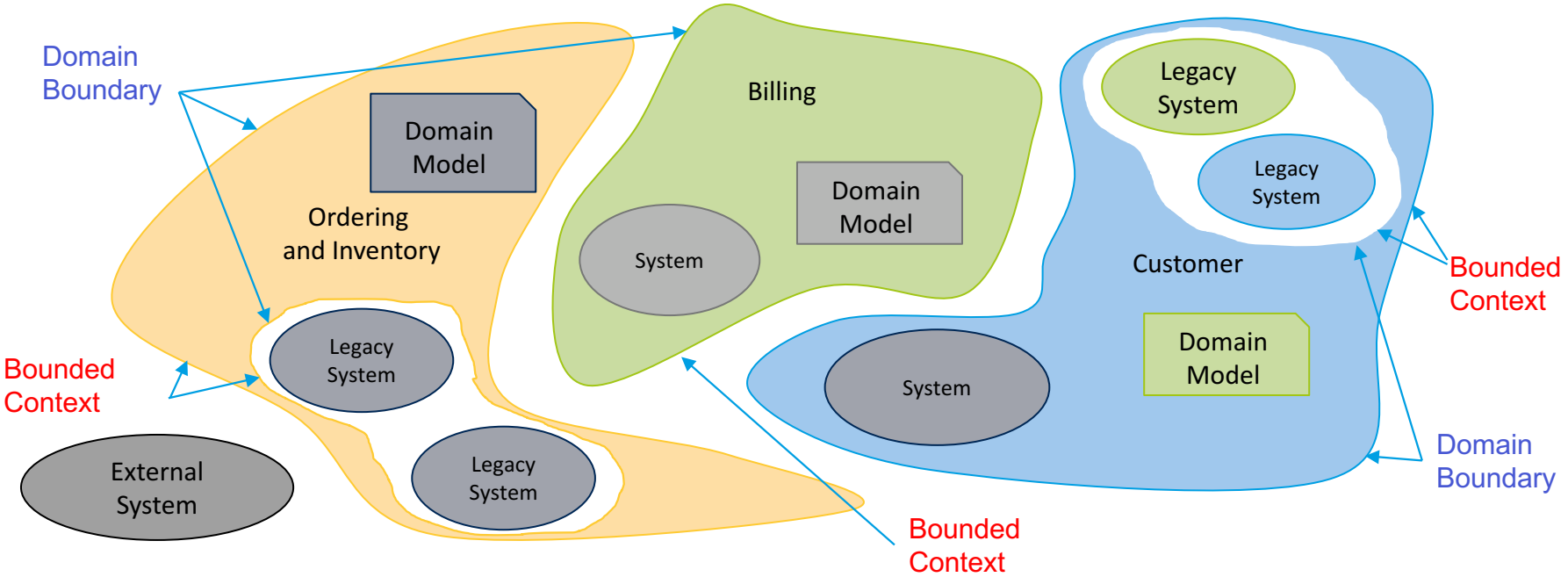
- Applications leak into each other making them hard to manage and maintain;
- *Conceptually the 'things' in the domains get confusing: Billing has an 'account', so does Customer, so does ordering, as does Web, as does 'Security', and Email, HR, bank payment, etc…..*



Recommended Approach – DDD* Isolation

Acknowledge the fact that **Ubiquitous Languages and Unique Domain models exist.**

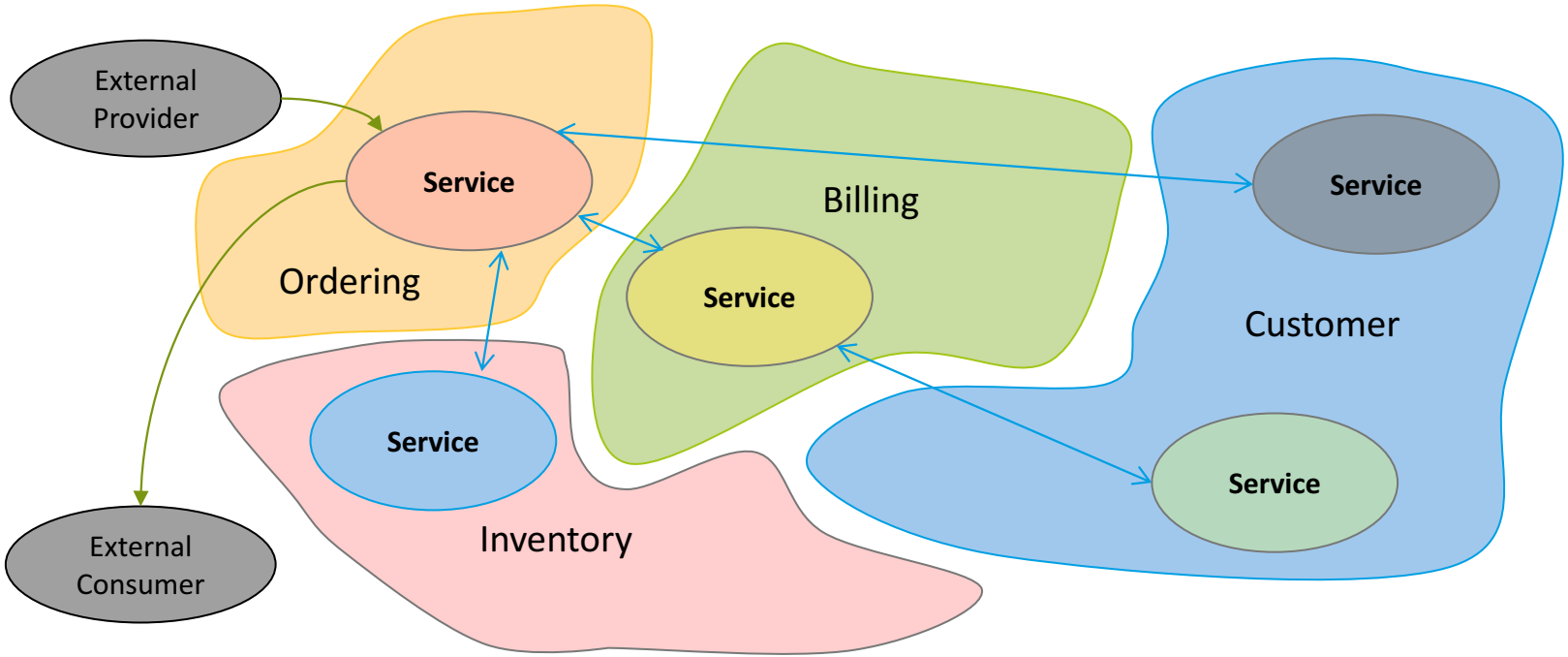
- Preserve the domains and use microservices as the way forward to provide autonomy;
- Acknowledge Legacy system exist and provide approaches o accommodate them.



* DDD = Domain Driven Design

Service Oriented Organisation

- Each Business's Capability is offered as a 'Service'.
- Further decompose the domain into subdomain.
- Services can be exposed externally and internally.



What is a Microservice Architecture ?

A microservices architecture is: “a *service-oriented architecture composed of loosely coupled elements that have bounded contexts*.”

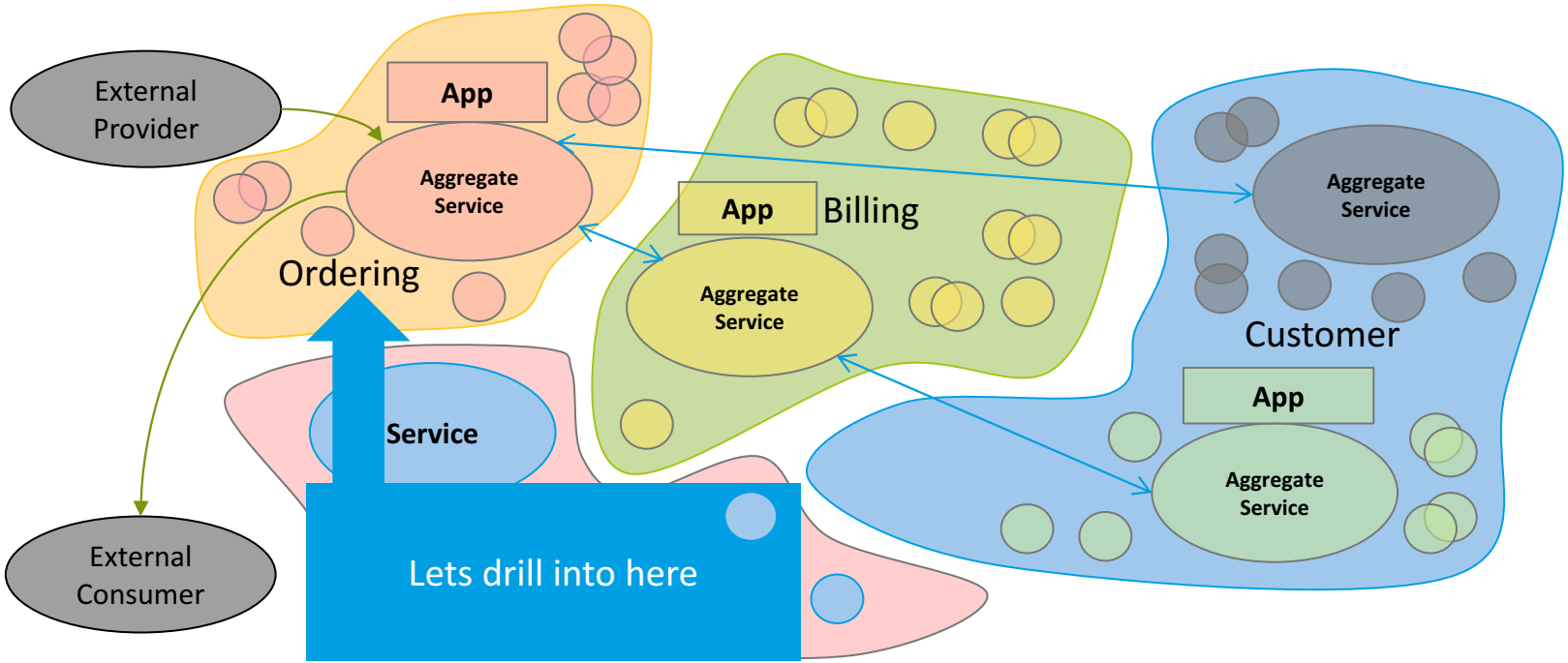
Adrian Cockcroft
Director of Web Engineering Netflix, and Cloud Architect.

“Small Autonomous services that work together, modelled around a business domain.”

Sam Newman

In reality its more complex..

- Applications may be made up of 100's of Services;
- Need to provide an encompassing aggregate of these for consumers;
- Need to provide a way for consumers to find and use services.

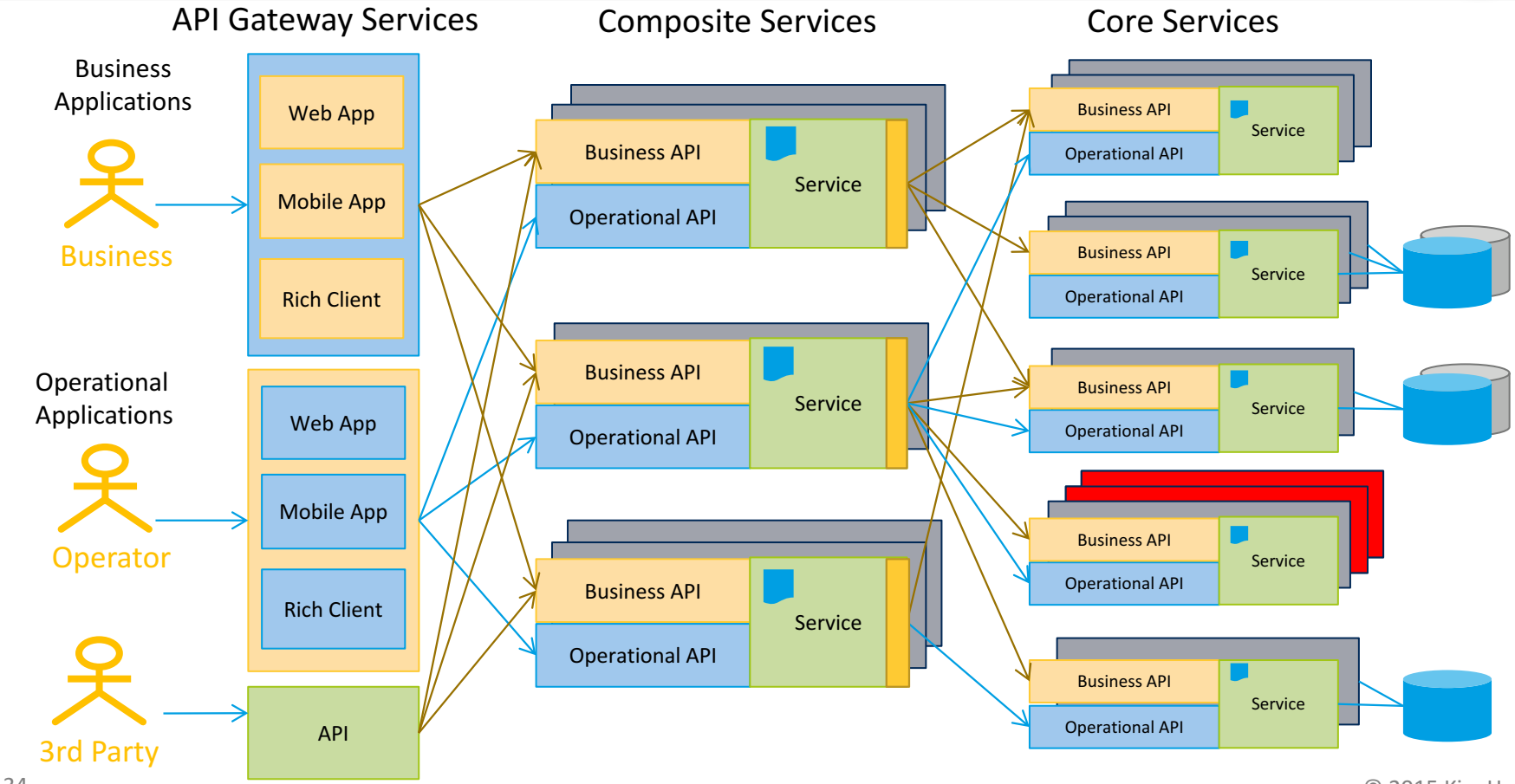


Partitions Services Across Layers

- **Core services** - Handle persistence of business data and applying business rules and other logic
- **Composite services** - can either orchestrate a number of core services to perform a common task or aggregating information from a number of core services.
- **API Gateway Services** - expose functionality externally allowing, for example, third parties to invent creative applications that use the underlying functionality in the system landscape.

- **Domain Partitioning** - horizontally we can apply some domain driven partitioning

Apps, Services, Instances, Environments – Across 3 Layers



Goals of the Blueprint

- Provide a proven and **robust open source alternate** to expensive and limiting commercial products;
- Provide an E2E **integrated and cohesive stack of plug and play components** combined into a lightweight framework. There are many commercial products in this space but they do not integrate or work seamlessly together.
- **Stop guessing capacity needs**, decisions to fix before will be wrong, either resources sitting idle or limited, causing failure. Provide elasticity; use as much or as little capacity as you need, and scale up and down automatically.
- Provide mature components that exhibit **proven scalability, availability, security, reusabilty, conceptual integrity, manageability and buildability.**
- Provide **distributed communications synchronously and asynchronously**, where applicable, providing streams for business events and data.
- Provide a **Cloud enabled** set of components that run anywhere.
- Choose components with **quality documentation** and readily available examples; to nurture our distributed systems capabilities.
- Capitalise on the vast experience of **best of breed organisations**;

Further Goals

- Accept that both **networks and platforms are unreliable** and provide graceful mechanisms to prevent faults from cascading into complete system failure.
- Distributed processes will come and go, change their locations, e.g. with faults.
  - Provide a way to **discover the locations of the healthy services**;
  - Provide mechanism to **detect failure** and allow components to fail fast;
- Provide **decomposition heuristics** that help determine the right services with right level of granularity (coarse to fine);
- **Automate Change:** Supports best and creative practices for build with fully automated deploy;
- Provide mechanisms to integrate non-java, and non-framework java components;
- Provide services to **consistently and reliably distribute configuration, live, in real time,** to many services running in many environments;;
- Provide approaches to maintain **visibility into the composite behavior** of a system that emerges from the evolving topology of maybe 1000s of autonomous services, so that the system can be monitored, managed and operated;

MicroServices Patterns

Catalog of Patterns – Main Types:

DP

Development Patterns - deals with the basics of building quality microservices.

RP

Routing Patterns – deal with 1000s of services: how to find services, elasticity, node failures, health and dynamic replacement;

CP

Client Resiliency Patterns – stop failures bringing down whole system and impacting consumers;

BP

Build/Deploy Patterns – build and deploy instances that have known and stable configuration, and configure infrastructure.

SP

Security Patterns – Authorisation, Authentication, Credential management and propagation;

MP

Monitoring and Logging Patterns – Log aggregation, correlation, monitoring and tracing.

TP

Testing Patterns – A/B, Canarying and Squeeze.

Development Patterns

DP

- **Service Granularity** - How do you decompose a business domain down into microservices, so they have the right level of responsibility; what is 'Micro'?
- **Communication Protocols** - How do consumers communicate with services; Sync Vs Async; XML / JSON / REST / etc
- **Interface Design (API)** – What is the best way to design the actual service interfaces ? How to version them ? Are they intuitive ?
- **Configuration Management of Service** - How do you manage the configuration of your microservice so that, as it moves between different environments, and the cloud, you don't have to change the code ?
  - **Runtime Reconfiguration** – apps that can be reconfigured without redeployment;
- **Event Driven Architecture** - How do you decouple your microservices using events ?
  - **Event Sourcing** – recreate current state from event history;
- **Data Base Per Service** – data bases are decoupled;
- **Eventual Consistency** – BASE vs ACID, Brewer's Theorem, CAP.

- **Autonomous teams** – an organisatonal pattern where small autonomous team own microservice end to end, top top bottom, bounded by their context (API).

Routing Patterns

RP

Service Discovery – How do we dynamically find services, without having to know their physical location and having it hard coded ? How do we ensure misbehaving services are removed from the pool ?

Service Routing – How do provide a single ‘aggregated’ entry point for all the services, and their instances, so that that policy, cross cutting concerns, routing rules are uniformly applied (as a PEP)?

API Gateway

Backend for FrontEnds

Creative and Surgical Routing – Tactical routing designed to meet a specific need, debig code, test a new feature, debug a specific customer.

Client Resiliency (Stability) Patterns

CP

Prototypes and GUIDs – request but no response or useful response;

Timeouts – kill requests that go past a set time;

Retries – try requests again that failed before; solution to Busy Signal;

Circuit Breakers Pattern – know when to stop retries, helps fail fast;

Fallback Pattern – when requests keep failing, provide a "plug-in" mechanism that allows client to carry out the work through some alternatives means;

Bulkhead Pattern - limit the impact of a fault and contain it;

Client-side load balancing – cache the locations of service instances and balance the load to the healthy ones;

Gatekeeper – protect services with a dedicated node that intermediates;

Load Shedding – Throttling control the consumption of resources used by a service, tenant, or an application; to protect it. Alternative to Auto-Scaling.

HandShaking – a server communicates with the client in order to control its own workload and or provide health updates to load balancer.

Busy Signal – tell client I am too busy

Horizontal Scaling / Auto-Scaling

Build/Deployment Patterns

BP

Build and Deployment Pipeline – create a repeatable, one-button, build and deploy process to any environment.

Infrastructure as Code – treat the provisioning of your services as code that can be executed and managed under source control.

Immutable Servers – once a microservice image is created and deployed make sure it never changes; no configuration creep.

Phoenix Servers – the longer a service runs the greater the opportunity for configuration drift. Ensure services are torn down regularly and recreated off an immutable image;

Red/Green Deployments – deploy old and new services in parralel;

Canarying – deploy slowly, trickling traffic at first to test waters;

Squeeze – when canarying succeeds gradually increase traffic.

Example - Service Registry Pattern

RP

Context

- Clients of a service need to discover and determine the location of a service instance to which to send requests.

Problem:

- How do clients of a service and/or routers know about the available instances of a service?

Forces:

- Each instance of a service exposes a remote API such as HTTP/REST, or Thrift etc. at a particular location (host and port);
- The number of services instances and their locations changes dynamically. Virtual machines and containers are usually assigned a dynamic IP address. An EC2 Autoscaling Group, for example, adjusts the number of instances based on load.

Solution:

- Implement a database of services, their instances and their locations;
- Service instances are registered on startup and deregistered on shutdown;
- Client of the service and/or routers query the service registry to find the available instances of a service;
- A service registry might invoke a service instance's health check API to verify that it is able to handle requests. Unhealthy instances are flagged.

Example - Gateway Service Pattern

RP

Context:
Multiple clients communicating with many individual services to aggregate data at their end.
Problem:
- Minimize the number of requests to the backend, e.g. reduce chatty-ness.
- How to allow clients to communicate with individual services required.

Forces:
- Granularity of APIs provided by microservices is often different than what a client needs;
- Different clients need different data;
- Single PEP, e.g. security issues, CORS, logging, monitoring;
- Network performance is different for different types of clients;
- The number of service instances and their locations (host+port) changes dynamically;
- Partitioning into services can change over time and should be hidden from clients.

Solution:
- Implement a server-side aggregation endpoint or API gateway responsible for aggregating data or, in some cases, acting as a simple routing layer for appropriate services.

Variation:
- See ‘Backends for Frontends’ Pattern - Create multiple API gateways for the different types of clients or frontend platforms that your application needs to support.

Example - Circuit Breaker Pattern

CP

Context:
In a distributed environment where an application performs operations that access remote services, it is possible for these operations to fail due to transient faults.

Problem:

- Repeatedly retrying (**see Retry Pattern**) to access the service, that may just fail, wastes resource, and alternate fallback (**see Fallback Pattern**) solutions cannot be tried.
- Failures at this part of the system may lead to cascading failures that ultimately impact the consumer or consume critical resources.

Forces:

- Resources are wasted retrying when they wont succeed;
- Blocking or waiting for part of the system while retries happen;

Solution:

- A circuit breaker acts as a proxy for operations that may fail. The proxy should monitor the number of recent failures that have occurred, and then use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.
- Implementing the circuit breaker pattern adds stability and resiliency to a system

Other Cloud Patterns

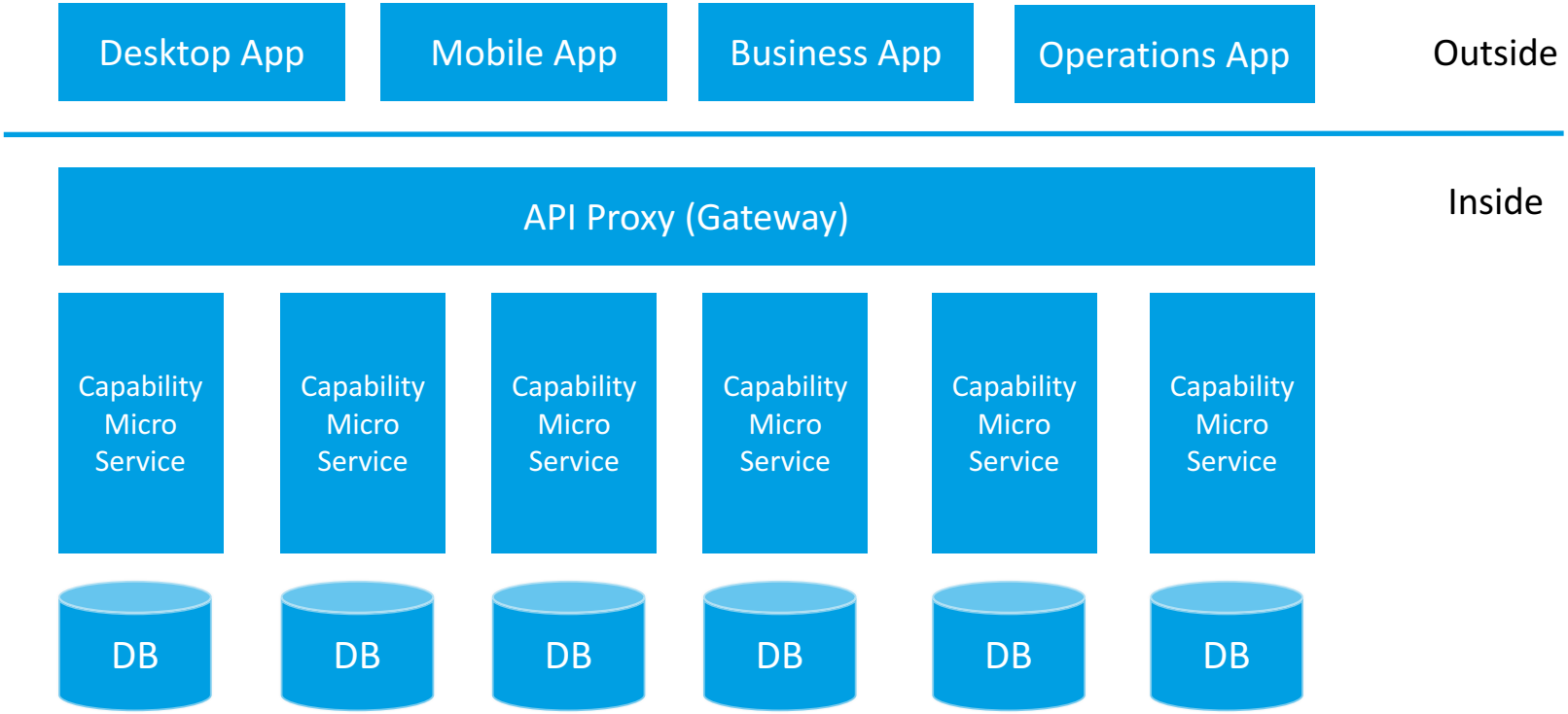
- Queue-Centric Workflow
- Map-Reduce
- DataBase Sharding – divide DB into horizontal partitions;
- Multi-tennancy
- Busy Signal
- Node Failure
- Network Latency
- Collocate
- Multi-Site Deployment
- Valet Key - Use a token or key that provides clients with restricted direct access to a specific resource or service in order to offload data transfer operations from the application code.
- Static Content Hosting Pattern / Content Delivery Network (CDN) – a type of cache, for web content external to organisation
- Priority Queue Pattern – service high priority queue elements first;
- Queue-Based Load Leveling – asynchronous queue levels variable load;
- Scheduler Agent Supervisor

More Cloud Patterns

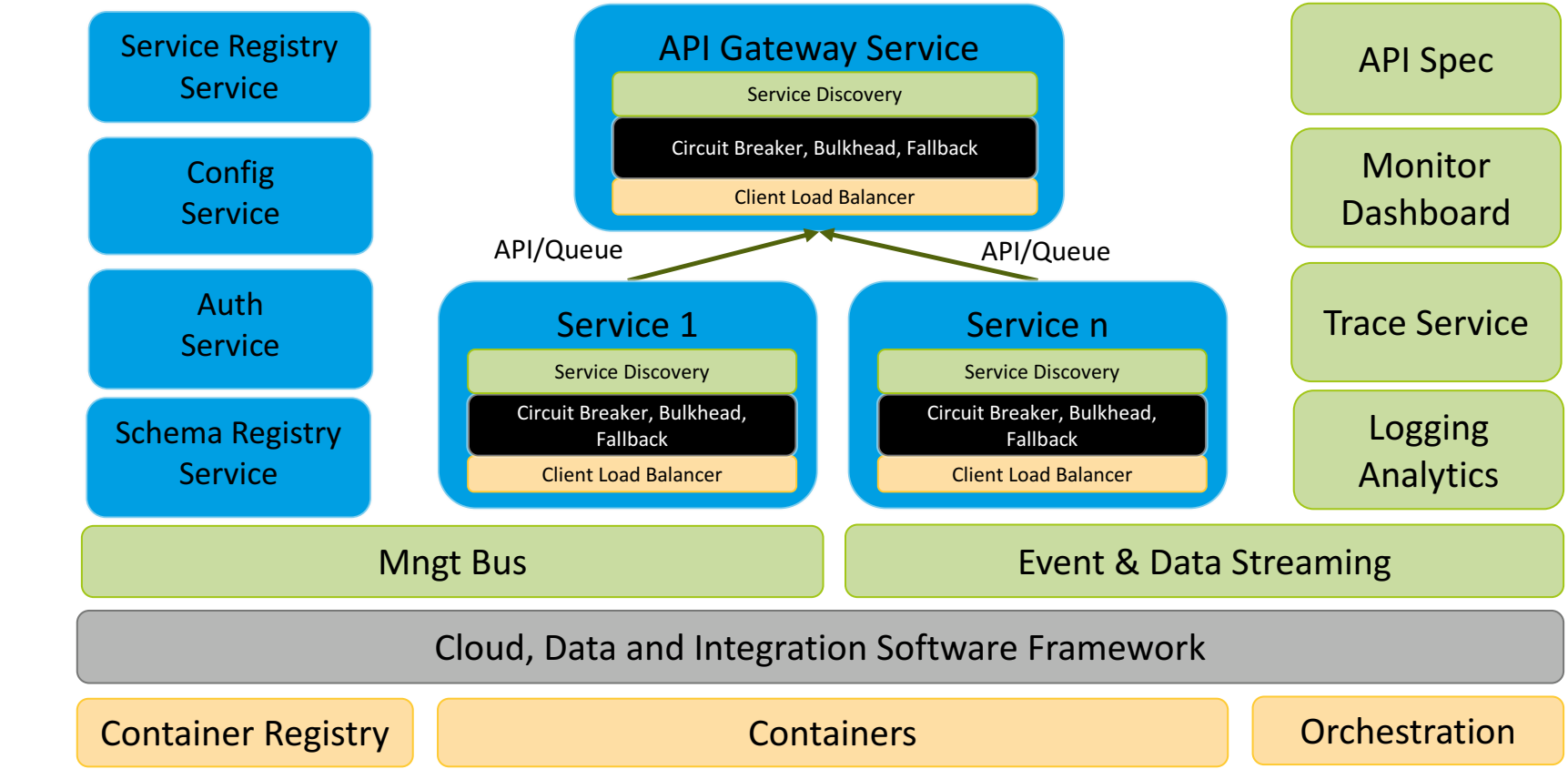
- Cache-Aside – cache data on demand;
- Compensating Transaction – undo a long running transaction;
- Competing Consumer – Use ‘Queue Centric Workflow’ with concurrent consumers;
- Compute Resource Consolidation – consolidate multiple tasks into one;
- CQRS – a Style that separates commands and queries;
- External Configuration Store- rather than hold/deploy config info locally at application, externalise it and provide API.
- Federated Identity – delegate authentication to external identity provider;
- Health Endpoint Monitoring – app exposes a service to return its status;
- Index Table – if secondary indexes not available emulate with separate table
- Leader Election – an app may have multiple tasks, running same code, concurrently solving problem, elect one to be leader to control other tasks.
- Materialized View – create prepopulated views to suite query when underlying format is not suitable; Essentially a cache that is updated when underlying data changes;
- Routing Slip – provide a list of processes to send a message too, a chain;
- Outbox – use an outbox to store messages before sending, good if queue is down.

Working Blueprint

Overall Framework



Distributed Cloud Blueprint– NFRs, use what you need



Common Tech For Patterns and Stack

Pattern / Capability	Tech
Gateway (reverse proxy)	Zuul 1&2 (Netflix) – wrapped by SpringCloud, Spring Cloud Gateway
Circuit Breaker, Bulkhead, Fallback	Hysterix (Netflix) – wrapped by SpringCloud
Client Side Load Balancer	Ribbon (Netflix) – wrapped by SpringCloud
Service Discovery Registry	Eureka (Netflix) – wrapped by SpringCloud / Consul
REST HTTP and Service Discovery Client	Feign (Netflix) – wrapped by SpringCloud
Configuration Service	Spring Cloud Configuration
Authorisation Service	Spring Cloud Security, SAML, OATH, OpenID
Event Driven Architecture, Mngt Bus	Spring Cloud Stream wraps Kafka (also supports Rabbit MQ), Spring Cloud Bus
Data Analysis and Flow Pipelines	Spring Cloud Data, Spring Cloud Data Flow, Sparks
Core Software Cloud Frameworks	Spring Boot, Spring Actuator, Spring Cloud, Spring Data, Spring Integration
IaaS, Container, Orchestration	Docker, Compose [Kubernites, Swarm]
Logging, Tracing & Analysis	SL4J, ELK[Elastisearch, Logstash, Kibana], Sprint Cloud Sleuth, Spring Cloud Zipkin.
In memory Cache	Spring Data Redis
Analytics, Monitoring Dashboard, Orchestration(bpm)	Atlas, Turbine, Visceral, Camuda, Conductor (lightweight)
API/ Service Specs, Schema Registry,Testing	Spring Hateoas (HAL), Spring Fox (Swagger), Avro, Protobuf, Spring Cloud Contract, WireMock

Microservice Architecture Principles

Microservices are:

1. Modelled around a bounded context; the business domain.
2. Responsible for a single capability;
3. Individually deployable, based on a culture of automation and continuous delivery;
4. The owner of its data;
5. Consumer Driven;
6. Not open for inspection; Encapsulates and hides all its detail;
7. Easily observed;
8. Easily built, operated, managed and replaced by a small autonomous team;
9. A good citizen within their ecosystem;
10. Based on Decentralisation and Isolation of failure;

Summary

Microservices are an Architectural Style:

- A mix of other styles is required for application architecture; this is not just ‘services’ but involves: distributed systems, Pipes and Filters, Reactive, and Event Driven;
- First style to support continuous integration and delivery;
- Provides a way to deliver robust and scalable distributed applications, in light of an unreliable world;
- No apriori bias for synchronous coms over asynchronous; base on NRFs;
- Support the goal of small autonomous teams and help structure them;
- Deal with the reality that business domains are contextual and trying to unify them just doesn’t work;
- To successfully implement distributed systems and microservices you need:
  - A blueprint of components and building blocks;
  - Well defines practices;
  - Well defined patterns covering: distributed systems, microservice and cloud;
  - Deployment approach based on Containers;
  - Integrated Continuous Delivery pipeline;

Next Episode

Describe

- Detailed Example
- Working Demo
- Decomposition and Bounded Context
- Containers
- REST and Not REST