Microservices

Principles, Patterns and Implementation considerations

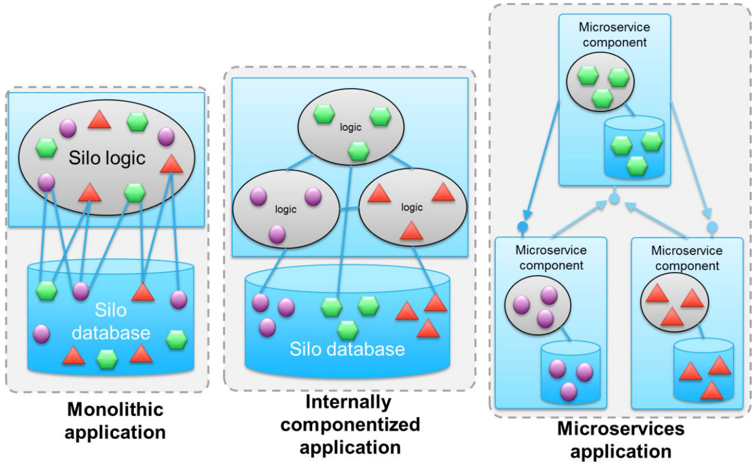
IBM **Developer**

WW Developer Advocacy Team

Microservices and its Impact on CIO’s Agenda

“…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. …”

Martin Fowler
http://martinfowler.com/articles/microservices.html



Agility and productivity

- The team that is developing the microservice can completely understand the codebase
- They can build, deploy, and test it independently of other components in much faster iteration cycles

Scalability

- The Microservices development team can scale the component at run time independently of other microservice components
- These Microservices take advantage of the elastic capabilities of cloud-native environments that have cost-effective access to enormous resources

Resilience

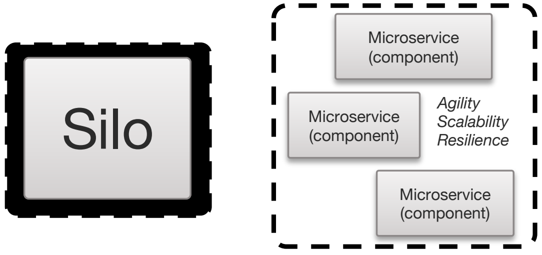
- The separate run time immediately provides resilience that is independent of failures in other components
- Technologies such as containers enable microservice components to fail quickly and independently, instead of taking down whole areas of unrelated functionality

Microservices – Fundamentals

Microservices Architecture

Simplistically, microservices architecture is about breaking down large silo applications into more manageable fully decoupled pieces

SOA done well



Monolithic application

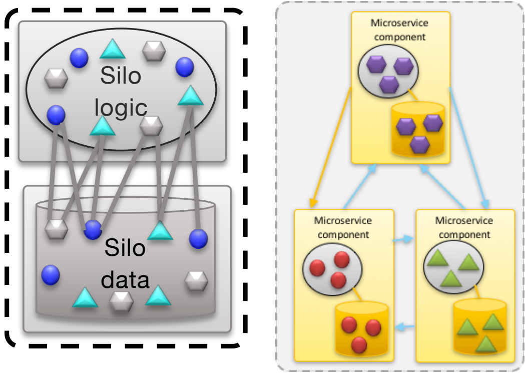
Microservices application

A *microservice* is a granular decoupled component within a broader application

Encapsulation is key

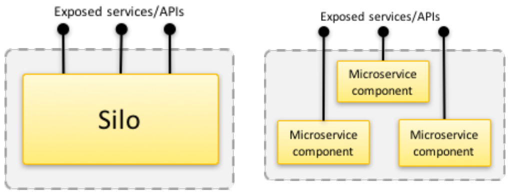
Related logic and data should remain together, and which means drawing strong boundaries between Microservices.

Business Centric



Microservice as API enabler

The “service” in “microservice” refers to the granularity of the **components**, not the exposed interfaces



An application split into Microservices may well expose the same APIs as its monolithic equivalent

Key takeaways

✓ Business Oriented

✓ Microservices need to focus on a unit of work, and as such they are small. There are no rules on how small a microservice must be, but making them too small can cause latency.

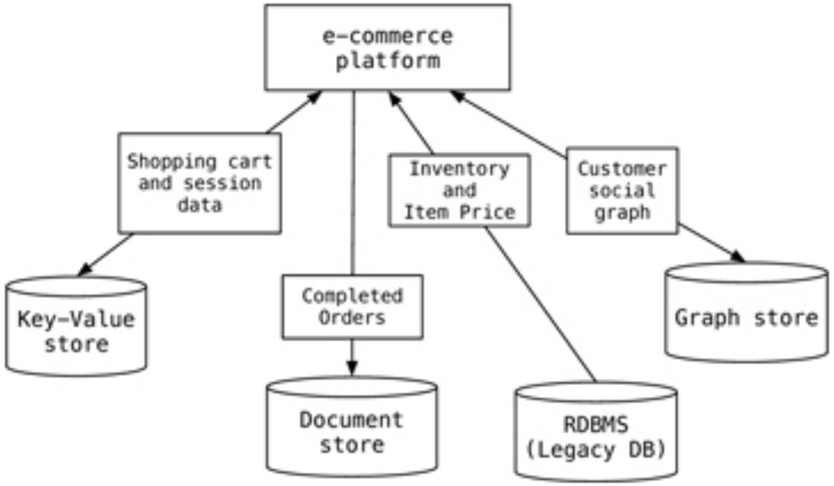
✓ A microservice needs to be treated like an application or a product. It should have its own source code management repository, and its own delivery pipeline for builds and deployment.

✓ Microservices-powered applications provide lightweight communication mechanism through the use of technology agnostic APIs.

✓ Microservices should have independent storage, should be independently changeable, should be independently deployable, should support distributed transactions.

✓ Reuse isn’t the only business motivation for microservices. There are others, such as localized optimizations to improve user interface (UI) responsiveness and to be able to respond to customer needs more rapidly.

Polyglot Persistence



Ideally, you want every microservice to manage its own database. This enables polyglot persistence, using different databases (for example, Cloudant versus MongoDB, both of which are NoSQL) and different types of data stores (such as Structured Query Language (SQL), NoSQL, graph).

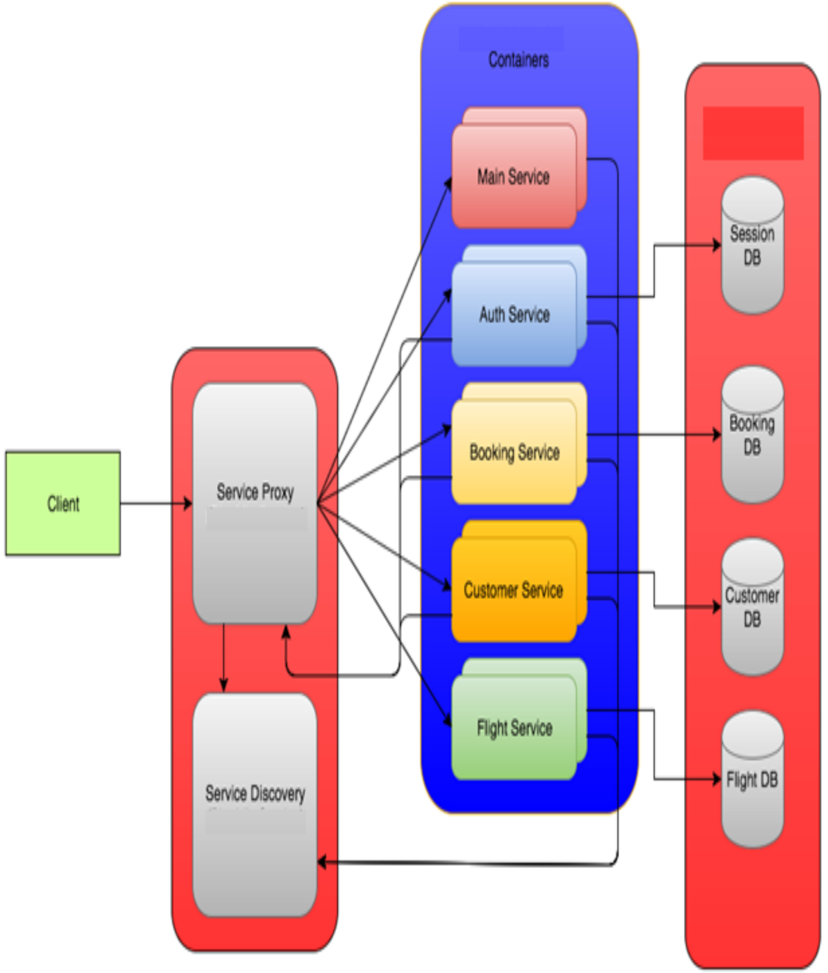
However, we might need to have more than one microservice use the same database for one of many reasons, for example, to preserve the ACID nature of a transaction that would otherwise be distributed across microservices and databases.

Database Type	Examples	When best used
Key-Value Databases	Compose for Redis, Memcached, Amazon DynamoDB	Storing interaction data, user preferences, simple shopping carts. Not great at set operations.
Document Databases	Cloudant, Compose for MongoDB	Event logging, content management, analytics. Not great at complex queries.
Column-Family Databases	Cassandra, Db2 BLU components for DB2 as a Service	Event logging, counters, blogs. Not compatible with ACID txns.
Graph Databases	Compose for Janus, Neo4J, OrientDB	Social Networks, Location-based Services. Not optimal for certain types of bulk updates.

Microservice Solution Overview

Here is a hypothetical Airline solution broken up into th following standalone services.

•Main app - presentation layer
•Auth Service - manages sessions (login, logout, validatior
•Customer Service - manages customer Info (get, update, validate password)
•Flight Service - manages flight data
•Booking Service - manages booking data
•Support Service - a simple dummy websocket chatroom (may add a cognitive service here)

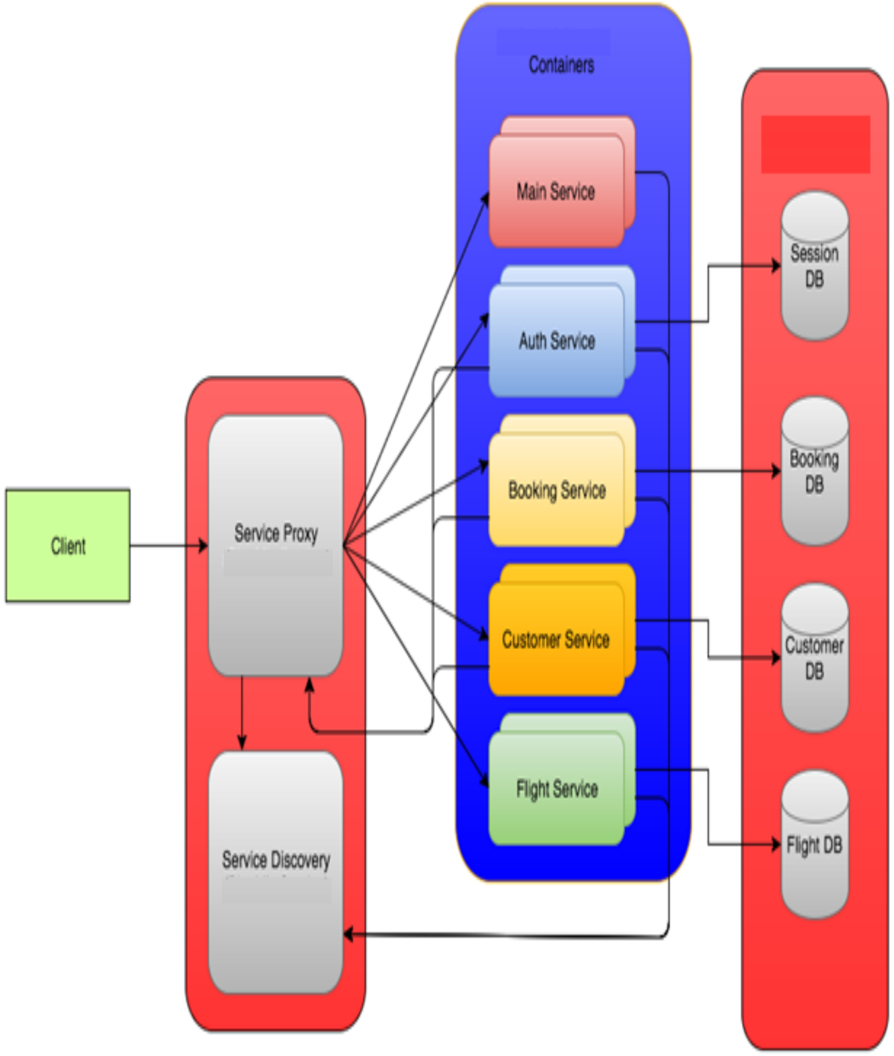


Microservice Solution Overview

•Multiple instances of each service can be created.
•A service proxy is setup in front of the services to route/
balance them.

- The Auth, Booking, and Customer services also need to know how to reach the proxy because the call other services.
- Service Discovery manages the micro-services.

•The databases can be separate instances (recommende
or one big instance.



Microservices Design principles

High Cohesion

Loose Coupling

Business centric

Resilience

Observable

Automated

High Cohesion

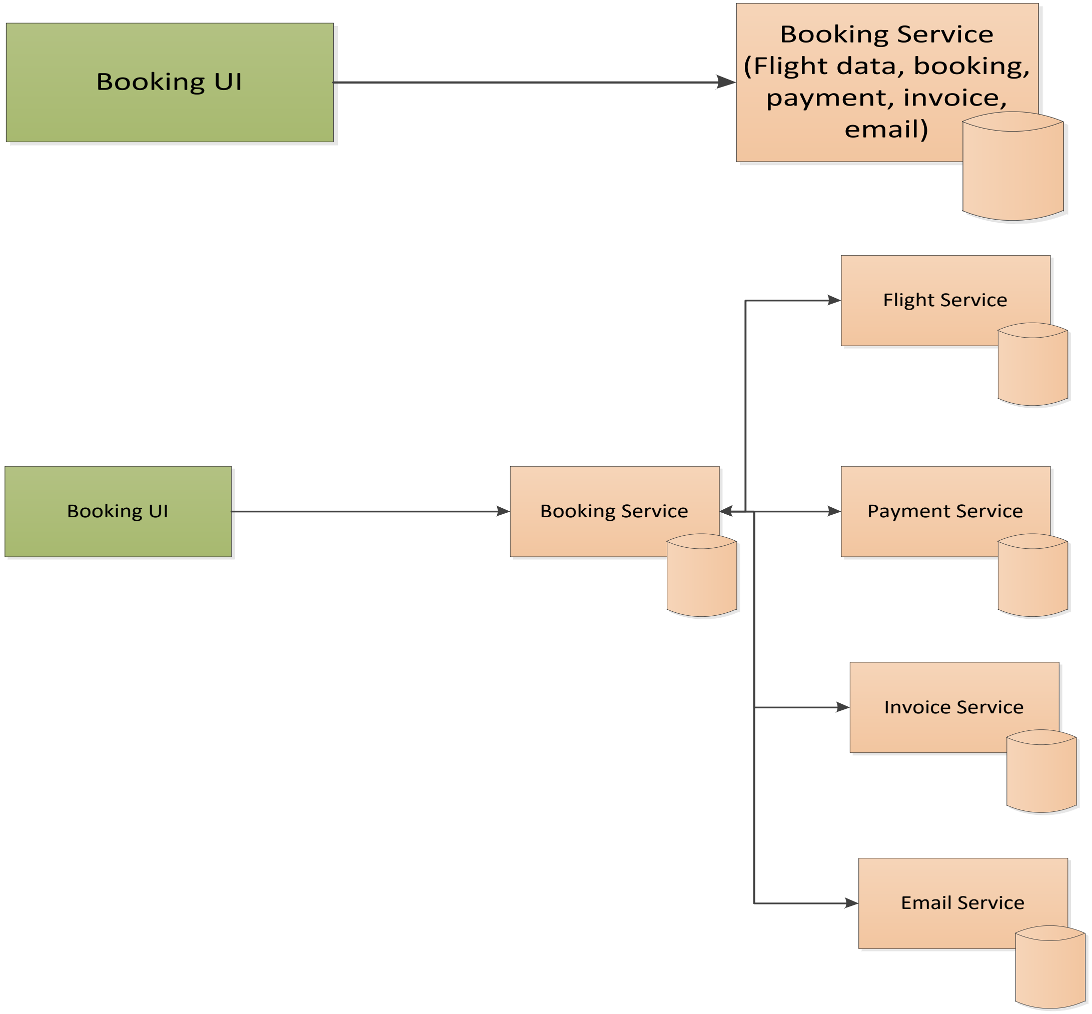
Each microservice should have a single function or focus. example domain centric.

Revisit our flight data example:

•Booking Service – browse flights (flight service), book tickets, payment service, generate invoice, send emails etc.

In its simplest form, the booking UI and service are tightly coupled and hence “less” cohesive.

With high cohesion:
•divide this functionality into multiple smaller, independent and focused micro services.
•manage each of these functionalities independently



Loose Coupling

Each microservice should be loosely coupled with other microservices

Decouple microservices over the network

- Use Open communication protocols
- HTTP/REST is standard being technology agonistic
- Avoid Client libraries resulting in RMI/EJB kind of communication

Standard interface for communication

- Have agreed interface for microservice communication
- Decouple internal representation of data within a microservice from how data is exchanged between two services.

Develop versioning strategy

- New changes should not break existing contract
- Should be backward compatible
- If breaking changes are unavoidable, then have parallel versions in play.
- Use semantic versioning to relay compatibility – Major, Minor, Patch
- Provide a migration strategy to consumers when new versions (not backward compatible) are available.

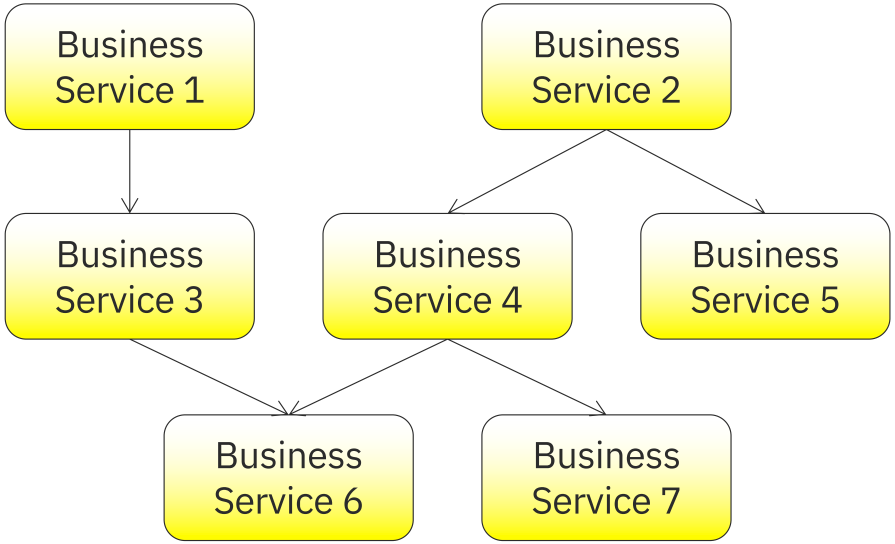
Business Domain Centric

Each microservice should map to a business domain or business function

Microservices can be defined iteratively on the basis of business domain or business functions. These domains can initially be defined in a coarse manner and then further split into possible business functions.

Remember high cohesion and loose coupling principles when splitting microservices. This also helps us align and group services better with the right teams within a large organization.

Business services can delegate to other business services. Avoid circular dependencies. Careful that each service still implements a complete task. Not separate layers



Business Service 1

Business Service 2

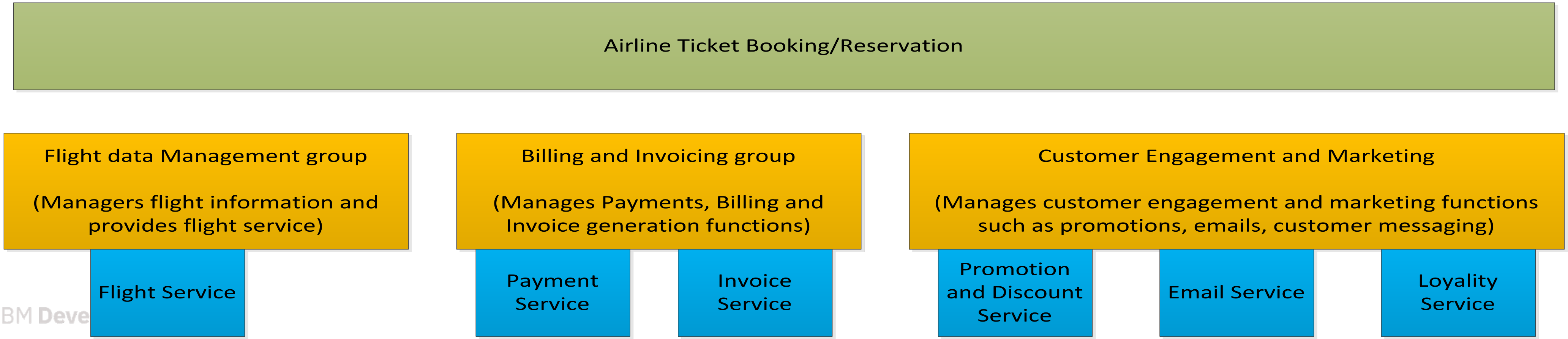
Business Service 3

Business Service 4

Business Service 5

Business Service 6

Business Service 7



Airline Ticket Booking/Reservation

Flight data Management group

(Managers flight information and provides flight service)

Flight Service

Billing and Invoicing group

(Manages Payments, Billing and Invoice generation functions)

Payment Service

Invoice Service

Customer Engagement and Marketing

(Manages customer engagement and marketing functions such as promotions, emails, customer messaging)

Promotion and Discount Service

Email Service

Loyality Service

Resilience

Build microservices for resiliency

Design for failures for down stream service.
When happens on failures

- Degrade functionality
- Default functionality

Design for fail fast

- Consider the usage/configuration of retries and timeouts. A long hanging transaction will not lead to a good user experience.
- Retries and Timeouts also help with network failures along with service failures

Observable – Monitoring and Logging

Centralized Monitoring

Use Real time monitoring

Monitor for operational indicator
- CPU, memory etc

Have services expose metrics which can be centrally logged and monitored
- Response times, Exceptions, Time-outs, Retries

Aggregate monitoring data

Use tools to help allow data aggregation, visualization to help see patterns and alert triggers.

Critical in context of Microservices with multiple microservices having multiple instances.

Critical to understand the lifecycle of how a request was processed in a distributed manner.

Include data such as host name, correlation id to help enable tracing a transaction across multiple services.

The log format should be consistent to help allow manage and query these logs later.

Automation

Devops principles for developing microservices

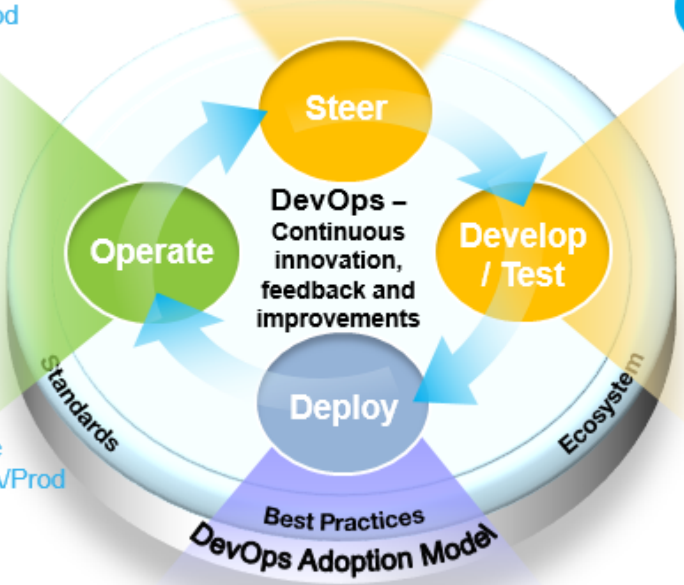
Continuous User Feedback and Optimization

Gain insights across Dev/QA/Prod and respond rapidly to the marketplace

Continuous Business Planning

Collaborative Development and Continuous Integration

Delivers High Quality Outcomes and Experiences with Speed



Continuous Monitoring

Monitor E2E across the Software development life cycle in Dev/QA/Prod

Continuous Testing

Highly automated approaches to dramatically reduce the time *and* effort in delivering quality results

Continuous Release and Deployment

On-Demand self deployments to Dev/QA/Prod supports continuous integration and delivery

Microservices Implementation Considerations

Communication Patterns

Hosting

Registration & Discovery

Monitoring

Logging

Caching

Scaling

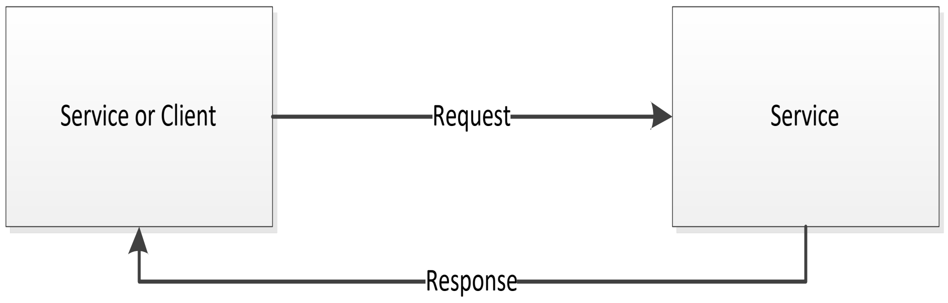
APIs

Automation

Communication patterns

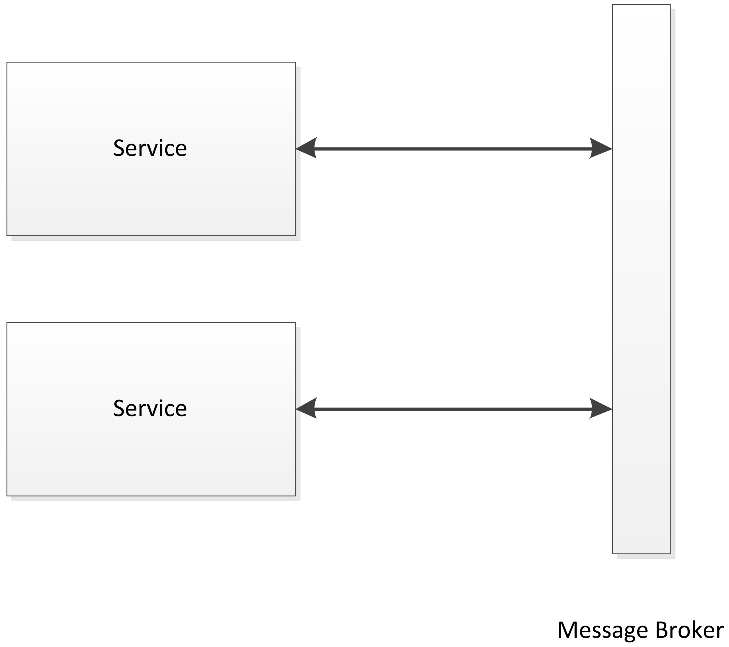
Synchronous Communication

- HTTP/REST – Technology agnostic
- Have to wait until a response is received
- Design for failures – use timeouts or retries



Asynchronous Communication

- service publishes an event which can be consumed by another service to be able to process the request.
- No need to wait for the response
- Use technologies such as Message Queuing



Hosting considerations

Virtualization

Containerization

Self Hosting

A common practice is to run one microservice per container.

Registration and Discovery

Service Registry Database

What to register?

- Host, port, version

When to register

- On Startup

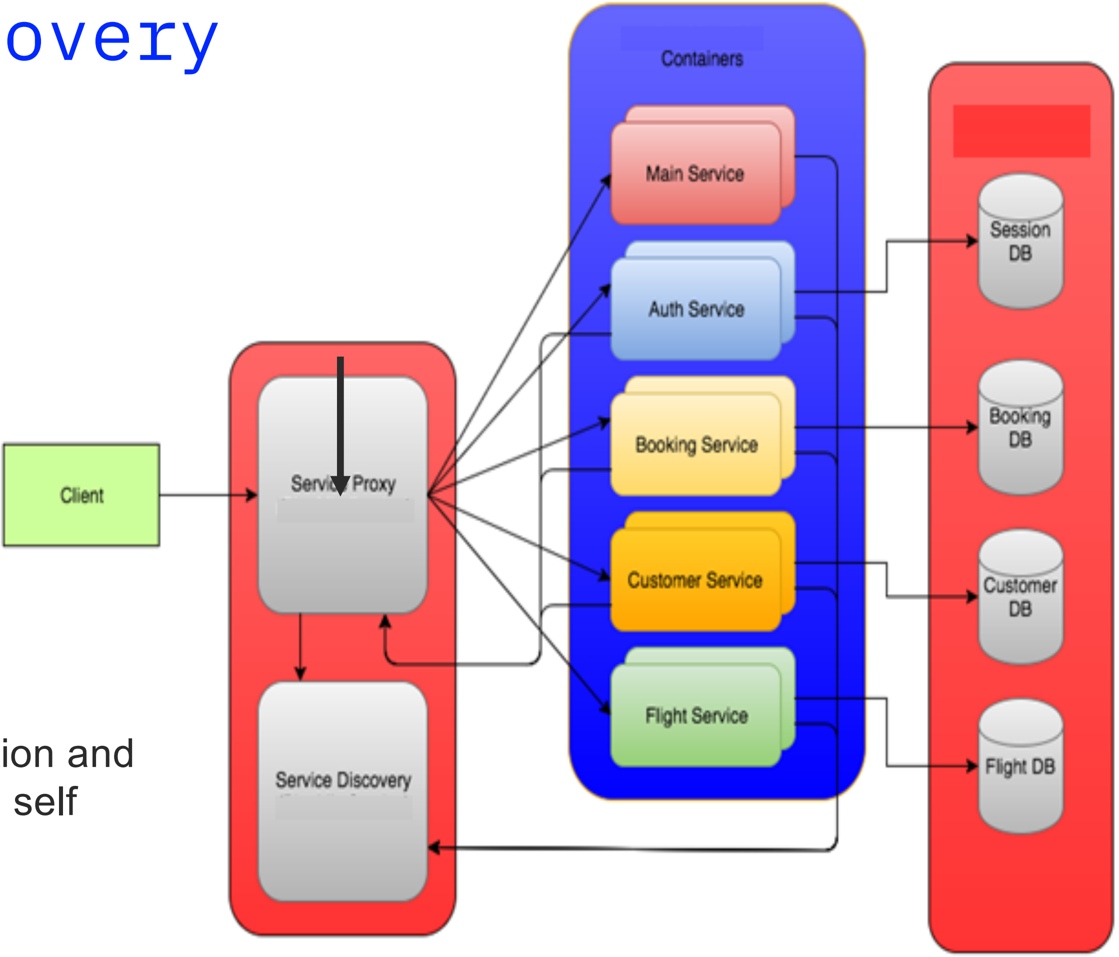
When to deregister

- On failures

Cloud platforms makes it easy to manage registration and discovery. This has to be self managed mostly in a self hosted scenario.

Service lookup patterns

- Client connects to service registry – Client side discovery
- Client connects to Gateway/Load Balancer to further connect to service registry – Server side discovery



Monitoring and Logging Technology

Centralized Monitoring tools

Example Nagios or New Relic

- Collect Metrics
- Aggregate
- Visualize data
- Send test transactions and monitor test transactions
- Monitor network
- Send alerts when required

Centralized Logging

Example Logstash, Elastic log, splunk

The logs needs to be pushed to these tools which can then be stored in a database. Capabilities provided are:

- Push logs
- Query log
- Aggregate log from multiple servers
- Provide logging libraries for client side
- Visualize data
- Supports standardized logging

Performance Considerations

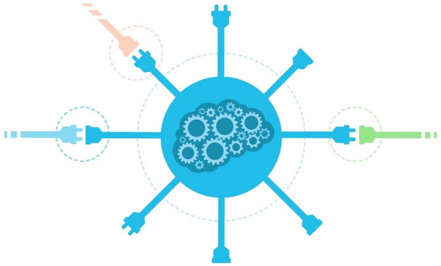
Scaling

- Horizontal Scaling is preferred and readily supported through cloud capabilities.
- Scaling can be automated or made on-demand
- Automated

API Gateway

- Load balancing and Caching
- Centralized Security
- Centralized Analytics
- Centralized service discovery
- Centralized entry points

Caching



- Helps improve performance
- Ideally implemented at API gateway layer
- Can also be done on client layer example for SPA applications
- Can also be done at service layer example for static data

References

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

http://www.redbooks.ibm.com/abstracts/sg248275.html?Open

https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

https://w3-03.sso.ibm.com/services/lighthouse/documents/D023183V03153V20

https://12factor.net/

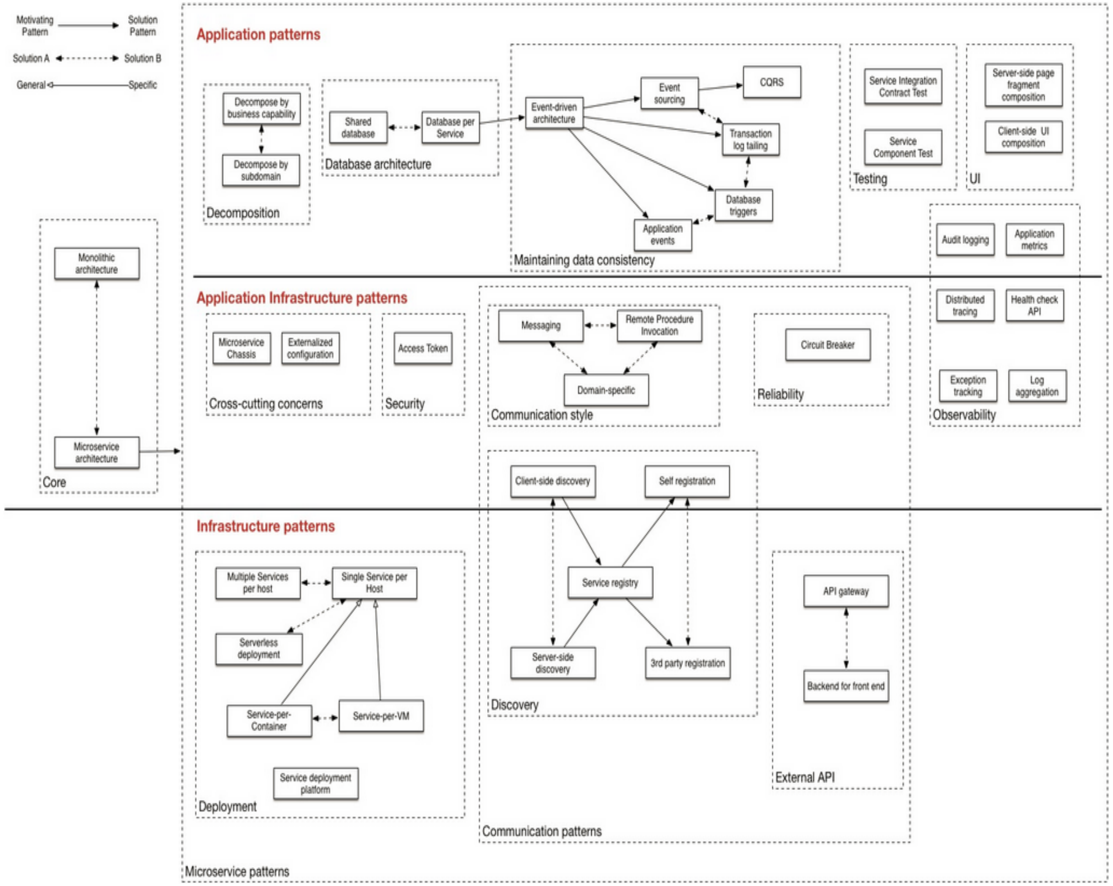
http://www.grahamlea.com/2016/08/distributed-transactions-microservices-icebergs/

http://microservices.io/patterns/index.html

Microservices Patterns

Microservices.io defines these patterns well

http://microservices.io/patterns/index.html



Core patterns
- Monolithic architecture
- Microservice architecture

Decomposition
- Decompose by business capability
- Decompose by subdomain

Deployment patterns
- Multiple service instances per host
- Service instance per host
- Service instance per VM
- Service instance per Container
- Serverless deployment
- Service deployment platform

Cross cutting concerns
- Microservice chassis
- Externalized configuration

Communication style
- Remote Procedure Invocation
- Messaging
- Domain-specific protocol

External API
- API gateway
- Backend for front-end

Security
- Access Token

Service discovery

Client-side discovery
Server-side discovery
Service registry
Self registration
3rd party registration

Data management
- Database per Service
- Shared database
- Event-driven architecture
- Event sourcing
- Transaction log tailing
- Database triggers
- Application events
- CQRS

Observability
- Log aggregation
- Application metrics
- Audit logging
- Distributed tracing
- Exception tracking
- Health check API

UI patterns
- Server-side page fragment composition
- Client-side UI composition

Testing
- Service Component Test
- Service Integration Contract Test

Reliability
- Circuit Breaker