POL00337644

CTO

CTO Principles, Guidelines and Patterns

Microservices

Version:1.0

Dated: 09-OCT-2018

CTO

POL00337644

Document Control

Abstract	Principles, Guidelines and Patterns for Microservices based Systems
Owner	CTO
Current Status	Draft
CTO Document Type	Principles

Change History

Version	Date	Author	Status	Comment
0.1	03/09/2018	Adrian Eales	Draft	Initial Draft
0.2	02/10/2018	Adrian Eales	Draft	Revised following review comments
0.3	03/10/2018	Adrian Eales	Draft	Revised following subsequent clarifications
1.0	09/10/2018	Adrian Eales	Initial Issue	Following review and approval at EAG

Reviewers & Approvers

Name & Organisation	Role	Date
Enterprise Architecture Group	A	09/10/2018
Michael Austin	R	18/09/2018
CTO Team	R	07/09/2018

POL00337644

Table of contents

POL00337644

CTO

Preface

Purpose of This Document

This document describes the principles and associated guidance that should be followed when developing Microservice based systems with the Post Office.
Following the principles and guidance within this document will lead to a more standardised development and deployment approach for systems within the Post Office.

Intended Audience

This document is intended to be read by Architects, and Lead Developers involved in the Architecture and Design of Microservice based systems.

References

#	Artefact	URL	Description
1	Post Office Architecture Principles	https://poluk.sharepoint.com/sites/Extranet/Strategy/AR/SitePages/Architecture-Principles.aspx?web=1	Post Office's top level Architecture Principles
2	Microservices viewpoints	https://martinfowler.com/articles/microservices.html	Martin Fowler's Microservices Website and comments/definitions

POL00337644

1. Scope of Document

1.1 General

The principles set out in the following document will replace all other previous Microservices Principles at the Post Office.

Principles are a fundamental part of our Enterprise Architecture. They are used to guide project investment and technology decisions and drive the Post Office forward to its desired future state. The principles are not technical standards they are overarching and can be applied to any technology based project.

The Governance Stack

- Principles: Provide direction and guide the decision making process
- Policies: Provide firm rules and frameworks to be followed
- Standards: Control specific technical implementations
- Patterns: Provide guidance on preferred methods of constructing solutions

This document is subordinate to the overarching Architectural Principles (Ref #1). In particular this document builds upon the following principles:

1.0 Architect the Big Picture
2.0 Plan for Service Life
3.0 Agility through Standardisation
4.0 Re-use, Buy, Build
6.0 Value for Money

This document sets out the Principles, Guidelines and Patterns that will be used in the creation of Microservices based systems which have been developed by, or on behalf of, the Post Office.

This document is intended to be read by Architects, and Lead Developers involved in the Architecture and Design of Microservice based systems.

1.2 Exclusions

Areas not currently covered in this document:

- Security implications.
- Implications of Access control – Authentication and Authorisation / Role Based API access.
- Implications and differences between providing Microservices for internal and external *Consumers*.
- Method of Microservice provisioning and execution method (e.g. containerisation, VMs etc).
- Serverless Functions.
- *Co-ordination*/managing change in a Microservices eco-system.

POL00337644

2. Terminology

Note, where the following terms occur in this document, they are *italicized*.

Term	Description
Choreography	Choreography is a loosely coupled, distributed, process for integrating two or more Microservices and/or External Systems together to automate a process, or synchronize data in real-time. For more information please see § 3.2.1.5.
CI/CD Pipeline	Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early. Continuous Delivery (CD) is the natural extension of Continuous Integration - an approach in which teams ensure that every change to the system is releasable, and that we can release any version at the push of a button. Continuous Delivery aims to make releases boring, so we can deliver frequently and get fast feedback on what users care about. A Ci/CD Pipeline is an automated process (and toolsets) that combine CI and CD into a single linear activity.
Contract	The collection of *Services* that are offered by a Microservice, and the *Objects* that can be operated upon.
Consumer	Application or system that makes a (RESTful) *Request* to an API Gateway or Microservice
Co-ordination	Co-ordination is a non-directive approach for sharing state information between Microservices or other systems. For more information please see § 3.2.1.5.
Entity	A self-contained set of related data elements e.g. user, catalogue,
Object	An instantiation of an *Entity*.
Operation	A transactional *request*, e.g. Authorise User. An operation tends to be a single transaction upon an *Object/Entity*.
Orchestration	Orchestration is the process of integrating two or more Microservices and/or External Systems together to automate a process, or synchronize data in real-time. For more information please see § 3.2.1.5. Orchestration is normally considered to be implemented as a centralized capability.
Request	A RESTful API GET, PUT etc submitted to a Microservice
Service	The *Operation*(s) that can be performed upon the supported *Objects* via a *Request* to a Microservice.
Service Mesh	A Service Mesh is a configurable infrastructure layer for a Microservices *System*. It makes communication between service instances flexible, reliable, and fast. The mesh provides service discovery, load balancing, encryption, authentication and authorisation, and other capabilities.
Single Responsibility	This also links to the *Object* orientated programming SOLID Principles in which the Single Responsibility Principle (SRP) is defined as *"Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class."*
State	In computer science, the state of a program, or application, is defined as its condition regarding stored inputs.
Stateless	Stateless means that an application has no record of previous *Requests* or interactions, and each *Request* has to be handled based entirely on information that comes with it.
System	An Application or Eco-System performing a specific business process or *Operation*.

POL00337644

3. Microservices Overview

3.1 Microservices Basics

Microservices are an architecture style, in which large complex software *Systems* are composed of one or more independently deployable services. To provide these services, Microservice are created as software applications which exist as a self-contained blocks of functionality, each exposing a defined interface for consumption by a *Consumer*.

The term "Microservice" is attributed to Martin Fowler (see Ref 2) whose description is:

> *The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.*

A Microservices is considered to focus on completing one task only and carries out that task efficiently. In all cases, that one task represents a single self-contained business capability.

An instance of a Microservice is the smallest unit (excluding predefined Cloud Functions) of developed code in a deployable, self-contained infrastructure that can be automatically scaled up or down, depending on usage requirements.

For the purposes of this document, a Microservice:

- IS independent of all other Microservices.
- IS developed, released, versioned, and managed, in isolation of other *Systems*/Microservices.
- IS a collection of one or more Actions that relate to a single *Entity*.
- IS testable in isolation.
- IS effectively a black box, from the perspective of *Consumers*.
- DOES NOT replicate functionality that already exists within the *System*.
- DOES NOT require a specific software development language or underlying infrastructure.
- IS NOT a scheduled activity, or background process that is called through an event, queue or pub/sub mechanism.
- IS NOT an endpoint or URL, though one or more endpoints or URLs may refer to a single Microservice.
- IS normally developed to support multiple *Consumers*.
- IS developed to provide a specific function or *Service*, not to support a single project.
- IS deployed behind an API Gateway. Therefore *Consumers* should NOT connect directly to a Microservice, and should pass through an API Gateway first.

In a Post Office context, there is a strategic desire to create a cross-company, reusable, library of Microservices representing the breath of technical and business services.

These Microservices may be created by multiple donor programmes, however each Microservice should provide a unique set of functionality without there being significant overlap or duplication with existing Microservices.

This will drive the formation of an API driven eco-system within the Post Office.

It should be noted that the deployment of Microservices come with potential challenges. Whilst Microservices simplify the implementation (the "inner architecture") of each *Service*, their distributed nature implies a more complex operational environment (the "outer architecture").

Business-critical use of Microservices requires a degree of maturity in continuous delivery and DevOps practices before committing to business-critical use of Microservices.

Commented [AM1]: I do not believe this to be true. This constraint may force services into being too granular. It depends on what you mean by Entity.

Commented [AM2]: I do not believe this to be true. Especially when you consider event driven architectures or architectures that rely on domain events.

Commented [AM3]: How do we tie these into the enterprise architecture to ensure it is coherent and the delivery teams have a clear mandate?

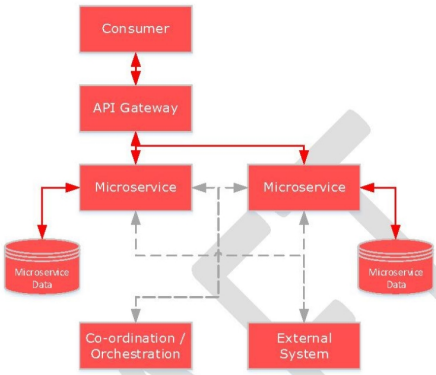
Commented [AM4]: How do we implement business transactions across services?

POL00337644

Adopting Microservices also requires a well-thought out set of non-functional and operational requirements and a delivery environment (the "Outer Architecture") accordingly.

3.2 The Microservices Eco System

The logical elements (depicted in the diagram below) are considered to exist in a Microservices based *System*:



3.2.1.1 Consumer

The *Consumer* application or *System* that makes a (RESTful) *Request* to the API Gateway or to a Microservice. Note, that where one Microservice makes a *Request* to another Microservice, then the *Consumer* may also be another Microservice.

3.2.1.2 API Gateway

This function may be made up from multiple deployed elements, with functionality potentially separated across design time/developer support, and run time activities.

Whilst the standard/expected pattern is for this logical element to always be deployed, in very simple, or low maturity systems (e.g. Proof of concepts) this element may be absent and the functionality below may be in part implemented as part of the Microservices themselves. If this is the case, then clear plans should be in place for the future implementation of this element. Note - this approach is considered an anti-pattern for mature systems.

The functionality offered by this logical element is:

- A single point of entry to the *System*
- API end point / API Proxy for Microservices
- Dual/mutual endpoints – ie providing multiple APIs, with each API tailored to a specific *Consumer*, against a single Microservice (ie supporting *n* APIs : 1 Microservice)
- Protocol translation
- Load/traffic shaping
- Border security
- Authentication & authorization, including inspecting, routing based upon, and passing on authorization/session tokens
- Data in transit encryption
- *Request* routing
- Logging *Requests*

Additionally API Gateways can also be used as a point of *Request* aggregation (e.g. taking responses from multiple Microservices to provide a response to a single API *Request*). It is however

Commented [AM5]: An API gateway is absolutely not a fundamental requirement for a Microservice architecture. An API gateway is primarily responsible for managing API's that are externally facing. They control versioning, throttling, access via keys etc. etc.

Gateways/Ingress controllers usually are – but this is also not required if for instance you have client side load balancing.

My suggestion would be to create blueprints for the different types of architectures we will support and reference those blueprints from here.

POL00337644

recommended that where possible this type of activity is handled by a *Co-ordination* or *Orchestration* element (Ref: § 3.2.1.5).

3.2.1.3 Microservice

Microservices are the unit of reusable currency, and are expected to conform to the Principles in this document.
There are varying types of Microservice patterns – please see § 5.2 for details of these patterns.
Functionality must be deployed in Microservices. Business processes, complex processing rules, or the like must not be built into the API Gateway.

3.2.1.4 Microservice Data

Each Microservice can have a private data store logically linked to itself. This data store should not be shared with other Microservices.
The data store is used for configuration, *State*, and any other data that is shared between all instances of a specific Microservice.

Commented [AM6]: It is important to state that if a service requires access to data in a different domain or service there are a couple of strategies;

-Copy the data into its domain. You are then bound to ensuring the data is consist (usually by subscribing to saga's or domain events)
-Get the data via an exposed API on the target service

3.2.1.5 Co-ordination / Orchestration

The *Co-ordination* or *Orchestration* element(s) provide linking of processes and data across Microservices

Co-ordination tends to be achieved via the use of products such as Zookeeper, and is often seen as providing a passive control of activities. This approach is generally used to provide e.g. cross-Microservice *System* state management.

Orchestration is seen as providing a more controlling/directive approach, and as such drives a business process or workflow across a number of Microservices. An example of this is where one Microservice, in order to fulfill a *Request*, requires responses from multiple other Microservices or External Systems in a specific order. Where an *Orchestration* pattern is used, the Microservices are ignorant of the business process that links them, and can be developed as simple functional blocks.

The *Orchestration* element can be achieved via the use of products such as: API Gateways (not a recommended pattern), Enterprise Service Buses, BPM Tools, Apache Camel etc.

Commented [AM7]: Enforcing orchestration via enterprise tools is an anti-pattern in my opinion. It can introduce deployment dependencies and scatters business logic across components / services.

Most requirements can be satisfied by either orchestrating within the relevant service (at point of request origin e.g. add item to basket) or as you mention below via eventing.

Another alternative (borrowed from SOA principles) is to use specific services that are compositions of operations across services e.g. CustomerOrderProcessService. The point of entry would be the Customer domain, hence this service would sit within the CustomerManagement capability. Although for most requirements this would be overkill.

There is also another pattern for Microservice deployment called *Choreography*. In this a group of Microservices generally receive a *Request* (e.g. by subscribing to events) and then perform their functionality in response to that *Request*. This pattern requires monitoring to ensure an end to end consistency and completion of tasks. This pattern is considered to be easier to implement for simple *Systems*, but for complex *Systems* - understanding interactions can be very difficult.

Whilst *Service Mesh* type implementations are growing in popularity, this document will not yet cover them given their lack of maturity.

3.2.1.6 External Systems

External Systems are the back end or 3rd party systems that Microservices need to connect with in order to complete a *Request*. In effect these are any system, which is deployed separately from the API Manager and Microservices.

The External System include one of the following patterns:

- The External System exposes RESTful APIs that are consumed by Microservice(s);
- The External System exposes data *Objects* that are consumed by Microservice(s);
- The External System exposes functionality via other protocols/methods.

Examples of these are:
POL CFS;
CWC Transtrak;
EBay APIs;
Service Now APIs.

POL00337644

POL00337644

4. Microservices Principles

4.1 Split of responsibilities

4.1.1 Statement

The elements shown in § 3.2 shall exist as follows:

- There shall be one or more *Consumers*;
- There shall be one or more API Gateways;
- There shall be several Microservices;
- There should be a Microservice Data store aligned to each Microservice;
- There may be a *Co-ordination* or *Orchestration* element;
- There may be External Systems.

Functionality shall be separated across these elements as described in § 3.2,

4.1.2 Rationale

To provide standardization across multiple Microservice based *Systems*, it is important to follow a standard and consistent separation of functionality across the logical elements in the *System*.

Microservices are considered the element of reusable currency, and may be proxied by more than one API Gateway.

There are competing API Gateway technologies.

Future moves to a *Service Mesh* type environment require functionality to be in External Systems and Microservices, not in API Gateways.

Supports Architecture Principles:
1.0 Architect the Big Picture
2.0 Plan for Service Life
3.0 Agility through Standardisation

4.1.3 Implications

- An API Gateway element should be deployed.
- Complex rules shall not be deployed upon an API Gateway.
- Routing of Microservice to Microservice should be direct, or via a *Co-ordination/Orchestration* element, rather than routing via an API Gateway.
- Functionality must be deployed in Microservices. Business processes, complex processing rules, or the like must not be built into the API Gateway.
- *Request* Authentication should take place within the API Gateway, not within each Microservice. However the management of fine-grained access *Objects* and *Operations* can be managed by Microservices.

4.2 Single Responsibility / Atomic

4.2.1 Statement

Each Microservice must be responsible for a specific feature or a functionality or aggregation of cohesive functionality – known as a *Single Responsibility.* A guideline to applying this principle is: "*Gather those things which change for the same reason, Separate those things which change for the different reason*".

Commented [AM8]: Change to Gateway or Ingress

Commented [AM9]: Not if you refer to an enterprise component. This should be done within the capability services – REST or Message Bus.

Commented [AM10]: To be more specific – cross context, inter service comms should be via the presentation services.

Commented [AM11]: Simpler to state that a service is fully responsible for all of its business, deployment, Security & functional concerns.

Commented [AM12]: This is nonsensical. Services should rely on HTTPS and certs. Services should also ensure all requests are AuthZ.

POL00337644

There are no formal rules on the size that a Microservice should be, however a typically referenced guideline is the Two-Pizza Team rule, which states if you cannot feed the team building a Microservice with two pizzas[1], your Microservice is too big.

4.2.2 Rationale

To support an Agile methodology, the scope (and therefore size) of Microservices must be constrained.

Large sized Microservices will drive increased complexity, and increase testing requirements.

Supports Architecture Principles:
1.0 Architect the Big Picture
2.0 Plan for Service Life
3.0 Agility through Standardisation
4.0 Re-use, Buy, Build
6.0 Value for Money

4.2.3 Implications

- Each Microservice shall consume less than a whole Scrum team during a Sprint. Under no circumstances should the development of a Microservice be spread across multiple teams.
- A Microservice shall perform *Operations* upon one *Object* (data or *Service*) only.
- A Microservice should NOT be designed to allow multiple services to be selectable based upon query parameters (e.g. a single Microservice supporting both operation=login, and operation=commitbasket).
- Microservice documentation (and *Contracts*) must clearly specify a unique *Single Responsibility* (Ref: § 7).
- Microservice should be designed in the first instance as globally consumable resource i.e. consumable/reusable by multiple applications/*Systems*.

4.3 Loosely coupled

4.3.1 Statement

It is essential that a Microservice is loosely coupled to other Microservices and *Systems*.
It must be possible to deploy a single Microservice on its own. There must be zero coordination necessary for the deployment with other Microservices.

4.3.2 Rationale

Loose coupling enables frequent and rapid deployments, reducing the time to market for new features and capabilities.
Testing of new deployments, and modifications, can be limited to the new/changed Microservice.

Supports Architecture Principles:
3.0 Agility through Standardisation
4.0 Re-use, Buy, Build
6.0 Value for Money

4.3.3 Implications

- The initial deployment of *Systems*, which consist of multiple new business capabilities (and therefore multiple new Microservices), are excluded from this Principle.
- Interactions with other Microservices shall be via published end-points only.

Commented [AM13]: The 'size' of a service is not readily quantifiable.

The first thing to focus on is ensuring the core capabilities have been defined, the services must then be aligned with the capability to ensure that you are delivering business value.

Commented [AM14]: We cannot quantify the size of a service therefore how can we quantify the size of a team that implements said service.

This will drive out nano services that are far too granular.

Commented [AM15]: Not sure I agree with this. Using DDD, you can have multiple entities under the same service. They may not be exposed to clients but they will still exist under the hood.

Commented [AM16]: Agreed on this if the query parameters invoke an operation on different entities.

[1] Remember that developers tend to be hungry and can eat a lot of Pizza - so often one Pizza will only feed one developer ;)

POL00337644

- No interaction is permitted between Microservices via data directly shared in the data store layer. If necessary data should be accessed publically via another Microservice (see also § 4.4 Bounded Context).
- A Microservice must be capable of deployment as an individual element, without *Co-ordination*/deployment of other Microservices.
- If the deployment of a Microservice (Microservice A) requires an extension to another Microservice (Microservice B), then the changes to Microservice B must be scope additions, and Microservice B must continue to support all existing interfaces (ie backwards compatibility).

4.4 Bounded Context

4.4.1 Statement

Bounded context indicates that a particular Microservice does not "know" anything about the underlying implementation of other Microservices surrounding it. "Knowledge" must be limited to the *Contract* offered by other Microservice(s).
If for whatever reason a Microservice needs to know anything about another Microservice (e.g. how it functions or specific on how it needs to be called), this is NOT a bounded context.

Commented [AM17]: Not the correct definition of a bounded context and therefore misleading.

A bounded context is focused only on domains, it does not care about services. Domains have entities and services.

4.4.2 Rationale

This is required to support Microservice Principles - § 4.2 Single Responsibility, and § 4.3 Loosely Coupled.

Supports Architecture Principles:
3.0 Agility through Standardisation
4.0 Re-use, Buy, Build
6.0 Value for Money

4.4.3 Implications

- Microservices shall offer a (documented) *Contract* of the services provided.
- Knowledge of other Microservices must be based solely upon the available, documented, *Contracts*.
- A Microservice may be bounded to it's personal data store, and to any directly connected External System (e.g. see § 5.2.2 for discussion of an Interface Microservice)

4.5 Stateless

4.5.1 Statement

Microservices must be *Stateless* and must respond to the *Request* without remembering the previous communications from the *Consumer*. Each *Request* to a Microservice should therefore be self-contained.

4.5.2 Rationale

This is necessary to enable, and support, efficient demand-based resourcing and scaling.

Supports Architecture Principles:
2.0 Plan for Service Life
6.0 Value for Money

4.5.3 Implications

- A Microservice shall process every *Request* based upon only the information contained within it.
- *State* within a *System* will generally be required, this shall be either managed by a *Co-ordination* or *Orchestration* component, an *External System*, or by appropriate data store in the data layer.

4.6 Independent

4.6.1 Statement

Microservices must process a *Request* independently, they may however collaborate with other Microservices.

POL00337644

For example, a Microservice that generates a unique report after requesting specific data from other Microservices is considered an independent Microservice.

4.6.2 Rationale

This is required to support Microservice Principles - § 4.2 Single Responsibility, § 4.3 Loosely Coupled, and § 4.4 Bounded Context.

Supports Architecture Principle:
1.0 Architect the Big Picture
2.0 Plan for Service Life
3.0 Agility through Standardisation
4.0 Re-use, Buy, Build
6.0 Value for Money

4.6.3 Implications

- See § 4.2.3, § 4.3.3 and § 4.4.3.
- A Microservice shall be developed, tested, deployed, operated and scaled independently from any other Microservice.
- A Microservice is responsible for managing its own data. A Microservice should manage its own data models and persistence to maintain encapsulation, and independence from other *Systems* whilst supporting cohesion i.e. code and data structures changing together.

4.7 Full-Stack Application

4.7.1 Statement

A full stack application is individually deployable. It has its own (virtual) hosting. A Microservice's business logic, data model and API interface must be deployable as a single logical system.

4.7.2 Rationale

This is necessary to support rapid deployment via a *CI/CD pipeline*.
This is necessary to support efficient demand based resourcing and scaling.

Supports Architecture Principle:
2.0 Plan for Service Life
3.0 Agility through Standardisation
6.0 Value for Money

4.7.3 Implications

- All necessary deployable resources must be included in a single package for deployment via the *CI/CD pipeline*.

POL00337644

5. Microservices Deployment Patterns

5.1 Basics

This section describes the patterns for:

The types of Microservices;
The patterns in which Microservices can be deployed.

Note, the choice of Microservice Type, and Microservice Deployment, Patterns should be agreed prior to development via normal Architectural Governance. Documentation expected as part of that process is given in § 7.

5.2 Microservice Type Patterns

The currently supported patterns for Microservices are listed in this section

5.2.1 Type 1 – Service Microservice

A Service Microservice is the standard pattern, in that the Microservice contains the functionality to support a series of *Operations* against an *Object*. A Service Microservice may reference other Microservices, or an External System in order to complete a *Request*.

5.2.2 Type 2 – Interface Microservice

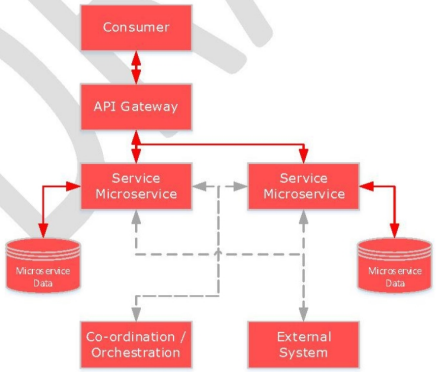
An Interface Microservice acts as broker for an External System that is referenced by a number of other Microservices. Generally this pattern is deployed where the method of communication with the External System is complex/not RESTful and cannot be referenced directly by a Service Microservice.

5.2.3 Type 3 – System Specific Microservice

An System Specific Microservice, is a derivation of a Type 1- Service Microservice. In this the scope of the Microservice is private to a specific *System*. Note, there shall not be *System* specific variants of Type 2 – Interface Microservices.

5.3 Deployment Patterns

5.3.1 Standard Deployment Pattern



This is the default pattern for deployment, and is expected to account for the majority of cases. In this pattern connected External Systems have supported RESTful APIs.

The method of operation is as follows:

Service Microservices are created and deployed to provide specific, unique, functionality.

Each Service Microservice delivers access to its functionality via a defined *Contract*.

Commented [AM18]: This needs to be expanded to cover the capability to service map.

Service Composition
1. Presentation Service : Responsible for presentation to the data to calling clients.
2.Business|Application Service : Encapsulation of all business logic.
3.System Service : Low level abstraction over system or infrastructure services e.g. DB/Messaging/Caching layers etc.
4.Integration Service | ACL : Anti corruption layer to broker comms between this service and other services or systems. Should translate data structures to internal versions to isolate impact of change.

These services would be deployed to provide the functionality for a capability e.g.

CustomerManagement

Is composed of:
CustomerPresentationService
CustomerMobilePresentationService
CustomerBusinessService
ExternalCustomerIntegrationService
CustomerNotificationSystemService

Note the example shows a specific Mobile only presentation service. Each of the items listed is an independently deployable service.

It should be noted that external callers into this capability or domain (customer) should always use the relevant presentation service.

POL00337644

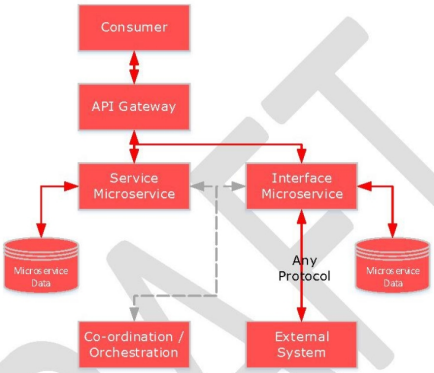
The Service Microservice functionality may be offered by:

i) A Service Microservice in isolation.
ii) A Service Microservice which collaborates with an External System via the API offered by that External System.
iii) A Service Microservice which collaborates with one or more other Service Microservices.

The API Gateway translates the services supported in the Microservice *Contracts* to one or more APIs.

The API *Consumer* accesses the offered APIs via the API Gateway.

5.3.2 Complex Backend Deployment Pattern



This patterns builds upon the previous pattern. Where *Services* related to an External Systems needs to be consumed, but the External System: does not provide a RESTful interface, needs a degree of *Orchestration*, and/or will be consumed by a number of Microservices.

In this instance an Interface Microservice is created to act as a dedicated interface to the External System.

The method of operation is as follows:

An Interface Microservice delivers access to its External System via a *Contract*.

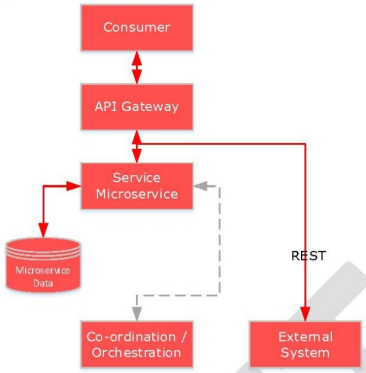
The *Contract* will be either consumed via a collaborating Service Microservice, or occasionally will be proxied directly by the API Gateway.

The API Gateway translates the *Services* supported in the Service Microservice *Contract* to one or more APIs.

The API *Consumer* accesses the offered APIs via the API Gateway.

POL00337644

5.3.3 Direct External System Deployment Pattern



Where an External System provides APIs that are: represented by a *Contract*; complete in breadth and depth; not subject to change; open/non-proprietary; and represent a specific, unique, *Service*.

Note, it is considered that most External Systems will be represented by one of the two previous patterns, and that this pattern will be used infrequently. However this pattern is presented for completeness.

It is recommended that this pattern should only be used where the External System is expected to be static for a fixed (long) term.

Consideration should also be made upon the commercial and operational implications of directly proxying an External System.

In this instance, the External System can be considered to be operate in the same manner as a Service Microservice. The method of operation is therefore the same as in § 5.3.1 Standard Deployment Pattern.

POL00337644

6. Microservices Naming

6.1 Basics

Microservices are a company resource, and therefore naming should be considered on the basis that a variety of different *Systems* (and teams) will be looking to consume them.

The naming should therefore try to be descriptive and self-explanatory.

System specific naming should only be used in specific instances, where the scope of a Microservice can be show to be limited to a single *System* (e.g. provision of a *System* specific UI). Otherwise a Microservice should be considered as globally consumable resource.

6.2 Service Microservices

The recommended format for Service Microservices is as follows:

<Object/Operation>_Service

Examples:

ITSMTickets_Service	For a Microservice that handles *Requests* against the "ITSMTickets" *Object.*
ProductCatalogue_Service	For a Microservice that handles *Requests* against the "ProductCatalogue" *Object.*
UserAuthentication_Service	For a Microservice that handles *Requests* against the "UserAuthentication" *Operation.*

6.3 Interface Microservices

The recommended format for Interface Microservices is as follows:

<External System Name>_<Object/Operation>_Interface

Examples:

BAL_Basket_Interface	For a Microservice that interfaces with the BAL, and handles *Requests* against the "Basket" *Object*
ForgeRock_UserAuthentication_Interface	For a Microservice that interfaces with the ForgeRock system, and handles *Requests* against the "UserAuthentication" *Operation.*

6.4 System Specific Microservices

The recommended format for System Specific Microservices is as follows:

<System Name>_<Object/Operation>_Local

Example:

AgentPortal_UIWidget2_Local	For a Microservice that is local to the system "Agent Portal" and handles *Requests* against the "UIWidget2" *Object.*

Commented [AM19]: This assumes an extremely low level of granularity. This needs to be reviewed a.s.a.p.

Operations should be defined in business terms, not in terms of systemic / front end components.

POL00337644

7. Microservices Documentation

The documentation that should be maintained depends upon the phase of a programme.

It is expected that teams should produce the following:

Document	Contents	Purpose
Architecture Overview	List of Microservices giving: • Name of Microservice • Microservice Type • Deployment Pattern • Responsibility – ie description of the *Service*, based upon *Object* and *Operations* • External Systems to be connected with (if applicable) • Method of communication with other *Systems* or Microservices • Compliance to Microservices Principles (ie this document)	• To ensure via Architectural Governance that uniqueness is observed. • To identify if required Microservices already exist, and can be reused, • To ensure compliance with Principles and Guidelines.
High Level Design	Catalogue of Microservices describing: • Name of Microservice • *Contract* specification • Interface specification (both provided and consumed form External Systems)	• To enable visibility and provide level of information necessary for reuse.

At present there is no centralised catalogue of Microservices. However such a system is planned, and once implemented this document will be updated to show how the above information should be structured and submitted.