**SOA 2.0**

**Microservices Design Guidelines**

Document Owner : SOA CoE, HKIT

Last Review : 11th, December, 2017

Version : 0.2

**Version History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date (DD/MM/YYYY)** | **Description of Change** | **Reviewed By** | **Approved By** |
| 0.1 | 03/10/2017 | Initial Draft | Chandi Prasad Ojha | Angus Chan |
| 0.2 | 06/12/2017 | Added section on Security, Error Handling, Logging, Naming Standards and Anti-Corruption Layer | Chandi Prasad Ojha |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Table of Contents**

[1. Introduction 4](#_Toc501008725)

[2. Acronyms & Definitions 4](#_Toc501008726)

[4. Key Characteristics 5](#_Toc501008727)

[5. Key Design Principles 6](#_Toc501008728)

[6. Reference Architecture 7](#_Toc501008729)

[9. Security Considerations 9](#_Toc501008730)

[10. Service Error Handling and HTTP Response Mechanism 11](#_Toc501008731)

[11. Logging & Monitoring 12](#_Toc501008732)

[12. Design Patterns Summary 13](#_Toc501008733)

[13. Design Patterns Details 15](#_Toc501008734)

[14. Naming Standards & Conventions 30](#_Toc501008735)

[15. Anti-Corruption Layer – Interfacing with Manulife on Premise IT Systems 31](#_Toc501008736)

[16. Implementation Best Practices & Guidelines 32](#_Toc501008737)

# Introduction

Purpose of this document is to establish and record the key design principles, guidelines and patterns for microservices implementation. This document will help SOA 2.0 factory teams to enforce best design practices while working on different microservices projects.

The intended audiences for this document are –

* Solution Architects
* Microservice Designers
* DevOps Leads
* Developers
* Testers.

# Acronyms & Definitions

|  |  |
| --- | --- |
| **Acronym** | **Definitions** |
| MSA | Microservice Architecture |
| SOA | Service Oriented Architecture |
| API | Application Programming Interface |
| PCF | Pivotal Cloud Foundry |
| SSO | Single Sign On |
| NFR | Non Functional Requirements |
| PCC | Pivotal Cloud Cache |
| PaaS | Platform As A Service |
| IaaS | Infrastructure As A Service |

# Key Characteristics

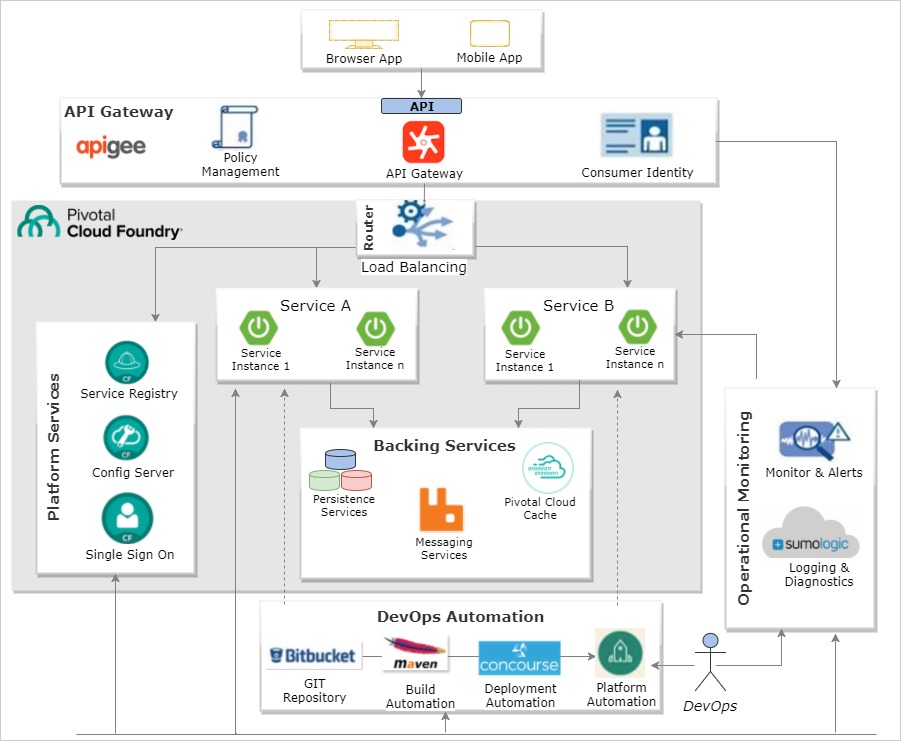
|  |  |
| --- | --- |
| **Characteristics** | **Description** |
| Build around business capabilities | Microservices are designed to represent business capabilities instead of technology layers. |
| Componentization as a Service | Microservices are independently deployable components. Changes made in one service will not propagate to whole application / product |
| Elasticity | A microservice must be able to scale up OR down independent of other services in the same application / product. |
| Resilient | A microservice must fail without impacting other services. Failure of a single service instance should not have cascading impact on application / product. |
| Smart Endpoints & Web Scale Pipes | While interfacing with other services, a microservice owns domain & process flow logic. Integration / messaging platform will provide communication channels at web scale. |
| Composable | A microservices must offer granular interface that is uniform and is designed to support service composition |
| Evolutionary Design | Microservices are designed around bounded context which will allow service evolution at more granular level thus making it highly cohesive. |
| Decentralized Data Management | Microservices persist and manage their own data. Every service can use different data storage mechanism |

# Key Design Principles

|  |  |
| --- | --- |
| **Design Principles** | **Description** |
| Consumer Focus | Always consider consumer’s experience while making design and implementation decisions. The consumer includes end customer as well as internal users & systems. |
| Start Simple and then iterate to address complexity | Start with breaking your existing systems into mini – monoliths. In every iteration, ensure to have a working functional code and then iterate further to improve it by splitting to next level of granularity. |
| Follow API Frist Design | Follow top – down approach to identify candidate APIs for each Business Unit. Also perform bottom – up analysis to discover candidate APIs from existing systems. Discuss with Business and IT stakeholders to finalize and prioritize the APIs implementation.  API design is very critical as it directly impacts the customer experience. Hence perform API design as the first design activity. Make use of Swagger / Open API standards for API design specifications. |
| Keep Evolving | Build a microservices platform which enables to introduce new features without taking down the service. Start with setup and configuration of basic support functions like – Deployment environment setup, Config Server, Service Registry, Log consolidation etc.  As next step, configure PCF platform to provide backing services like – API Gateway, DB, Data Stores, message broker, In-Memory cache etc.  Finally, perform needful steps on PCF platform to enable operational analytics and dashboard functions. |
| Automation is the Key | Follow DevOps culture to automate service implementation lifecycle. This will reduce human error thus improving reliability and consistency.  Start with automation of basic DevOps activities like packaging and build generation, code quality checks and deployment in DEV region to perform smoke testing.  As next step, setup and configure Continuous Integration pipeline automating all CI activities.  As next step, setup and configure Continuous Delivery pipeline automating the delivery / deployment of service artefacts on PCF. |

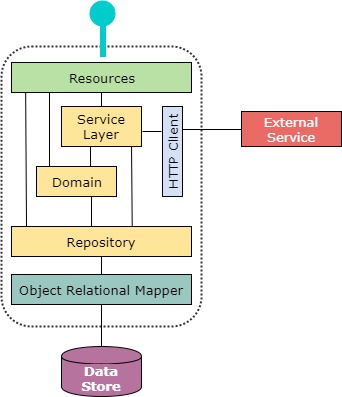
# Reference Architecture

Outer Architecture View



* API Requests from System of Engagements will be received by **Pivotal GoRouter**
* GoRouter delegates the API request to **APIGEE Micro-gateway**
* APIGEE Micro-gateway polls **APIGEE Edge cloud** to fetch policies and API proxy configurations
* APIGEE Micro-gateway passes through the call to **Go Router**
* Go Router sends the request to **Feign Client** which looks up **Service Registry** to fetch Microservice endpoint
* **Microservice App endpoint** gets executed and performs needful business function using **backing services**.
* Microservice App sends the **response back to Consumer** through Go Router.
* Micro-gateway shares the **operational analytics** with APIGEE Edge cloud in asynchronous mode.
* **DevOps CI / CD pipelines** ensure the continuous delivery of new versions of microservices.

Inner Architecture View



* A **resource** is an object which acts as mapper between the service and data representing the domain.
* Service logic resides in **domain model (Service layer, Domain, Repository)** representing the business domain
* Once the **resource** receives the request, it validates it and then call the domain to process it
* If multi domain activities are involved in a transaction, the request will be delegated to **service layer**
* The **repositories** acts on domain entities and is always backed by a data store
* The **repositories** utilize **ORM** frameworks to perform CRUD operations on data store
* In order to communicate with other services, **HTTP Client** is utilized to handle the request and response
* When one service needs to communicate with other remote service, the **Gateway component** will encapsulate the message marshalling and transformation (of messages to and from domain objects) logic.

# Security Considerations

The very much dynamic nature of microservice architectures introduces much more complexity while handling security. There is also a need to consider security based on application or service boundaries. The main difference between monolith applications and micro service architecture is the more number of moving part in microservice based architecture. Hence some considerations needs to thought out on securing rapidly changing infrastructure which still allows the individual services to change and grow without centralized management.

Following are the some of the key topics which needs to be considered with respect to security

**Data Security**

A different kind of data requires different level of protection and this would have an influence on how the data is accessed, transmitted and stored. The below are some of the considerations which needs to be taken into account with respect to Data Security

* Never transmit passwords as plain text
* Store password using salted hashes which provides more security
* Use standard encryption methodologies
* Sensitive data should be encrypted as early as possible and should be decrypted as late as possible. This helps prevent accidental exposure in logs
* Always protect the private keys

**Credentials Management**

Establishing identity in a highly distributed and dynamic environment would be very difficult. It is always required to establish and maintain the identity of users without introducing extra latency and contention with frequent calls to a centralized service. End-to-end SSL can bring some benefit in that bytes are not flowing around in plain text, but it does not establish a trusted environment on its own and requires key management. The below are some of the considerations which needs to be taken into account with respect to Credentials Management

* Credentials, certificates, and keys must be stored at some common place for automation scripts / tools to pick them
* Never store credentials alongside applications
* Never store credentials in a public repository
* Always store only the encrypted values. Spring Cloud Config uses Git to back up their configuration. It provides APIs to automatically encrypt configuration values and decrypt when required.

**Managed Services**

Connections to backing services are additional sources of network traffic that may need to be secured. In many cloud environments, backing services are provided by the platform, relying on multi-tenancy and API keys or access tokens to provide data isolation. It is important to understand the characteristics of backing services, especially the services which store data (MySQL / MongoDB etc.), to understand how the data is stored at rest to ensure regulatory requirements are satisfied

**Data Logging**

Log data is mostly a trade-off between what is required to analyze an issue and what should be protected for regulatory and privacy reasons. When writing to logs, it is always good to take full advantage of log levels to control how much data is written to logs. For user-provided data, consider whether it belongs in logs at all if not required do not log the values unnecessarily

**Authentication and authorization**

With monolithic applications, it is common to have fine grained roles or role associated groups in a central user repository. With the emphasis on independent lifecycles for microservices this may not work. Development of an independent microservice is then constrained by and coupled with updates to the centralized resource. It is common to have authentication (establishing the user’s identity) performed by a dedicated, centralized service or API gateway. This central service can then further delegate user authentication to a third party. OAuth provides an open framework for delegating authorization to a third party. Although the framework does define interaction patterns for authorization, it is not a strictly defined programming interface and there exists some variations between OAuth providers

When working with authorization (establishing user authority or permission to access secured resource), in a microservices environment, it is good to keep group or role definitions coarse grained in common, cross cutting services. Also allow individual services to maintain their own fine grained controls. The most important factor is independence. A balance must be found between what can be defined in common authorization service to meet requirements for the application level and at service level.

# Service Error Handling and HTTP Response Mechanism

REST is basically an architecture style and it is always good to leverage the standard behavior of the protocol underlying it. Hence when creating REST API on HTTP(S), it would be logical to follow HTTP protocol standards strictly (to maximum extent). Conforming to the standards makes it easy for the consumers of the API to understand it easily.

While working on REST style approach, it is very important to segregate the API as logical resources. Once arrived at the resources, then operations can be performed on these resources with HTTP Request methods (GET, POST, PUT, DELETE etc.) where each of these methods have specific meaning.

When it comes to API Response, it is always recommended to utilize the appropriate HTTP Status Codes. The response and status codes may vary across each and every API due to the type of data each API’s deal with, the way the consumer uses the API to interact and the server side processing impact. To start with the selection of status codes for the API response, identify the most possible outcomes (Success, Validation Failed, External dependencies (another service, database) not available, application exception, invalid request, application down etc.) for the API and accordingly decide the HTTP status codes. The below table summarizes the more frequently used HTTP status code along with the scenario’s when these status codes has to be utilized

|  |  |  |
| --- | --- | --- |
| **Code** | **Description** | **Comment** |
| 200 | Ok | Request processed successfully |
| 201 | Created | Resource created, response contains reference to new resource |
| 204 | No Content | Request processed successfully but has not resulted in new resource creation |
| 304 | Not Modified | Resource has not been modified since the version specified by the request headers |
| 400 | Bad Request | Inputs not as per expectation / data validations failed |
| 401 | Unauthorized | Similar to 403 Forbidden, but specifically for use when authentication is required and has failed |
| 403 | Forbidden | Need account or proper permission to access resource |
| 404 | Not Found | Requested resource not found |
| 405 | Method Not Allowed | Request method not supported |
| 409 | Conflict | Edit conflict between multiple simultaneous updates |
| 410 | Gone | Requested resource no longer available and will not be available again |
| 415 | Unsupported Media Type | Request format not supported |
| 500 | Internal Server Error | Backend Service / SQL / Queue / Cache / Unknown Application Error |
| 503 | Service Unavailable | Service down or not available |

The application / service exception handling strategy depends on the possible API usage scenarios and the corresponding API response. Once the scenarios and responses have been identified, it would be easy to arrive at a generic exception handling strategy for the service.

Spring provide several different options for handling exceptions and the most flexible one would be to utilize the @RestControllerAdvice to handle all the identified exceptions at one class with the corresponding HTTP status code. Spring readily provides several exception classes (Method Argument Not Valid, Http Message Not Readable, No Handler Found, Http Media Type Not Supported, No Handler Found, Http Request Method Not Supported etc.) that are readily available to handle which can also be easily mapped to the above status codes. Combining this with service specific exception classes (Customer Not Found, Policy Not Found, SQL Exception, Backend Service Exception etc.) it would be easy to build a more effective and robust exception handling mechanism.

# Logging & Monitoring

Logging is a very important aspect for any application to understand and track the applications internal information. As applications evolve and becomes more complex, having clear [visibility of metric and event data](https://logmatic.io/blog/why-monitoring-4-hard-facts-to-swallow-for-modern-app-perf/) to improve monitoring, troubleshooting and understanding becomes even more crucial. Following are some of the best practices and guidelines with respect to logging, monitoring and analyzing logs

|  |  |
| --- | --- |
| **No** | **Logging & Monitoring - Best Practices & Guidelines** |
| 1 | Utilize simple and consistent log structure |
| 2 | Standardize the log format (JSON) so that it is easy to read and parse |
| 3 | Use a standard logging framework (SL4J) |
| 4 | Ensure logs are written asynchronously so that the application performance does not suffer due to logging |
| 5 | Arrive at a standard logging configuration and utilize the same across the teams |
| 6 | Make sure sensitive data is not logged |
| 7 | Log as much as you can since this would help in getting the right information during analysis. At the same time ensure correct data with proper information is logged |
| 8 | Customize the logging level for the frameworks (spring, hibernate etc.) and application specific packages (Service, Dao etc.) so that we can get more insights |
| 9 | Ensure the usage of appropriate frameworks (Spring Sleuth) to facilitate analyzing of logs which cut across several services to complete a particular transaction |
| 10 | Analyze the provisions to change the log level at runtime if required |
| 11 | Encrypt the log data if it is being sent over HTTP |
| 12 | Ensure notification mechanism has been properly configured so that the system / product can respond to event whenever it occurs |
| 13 | Arrive at standard template for different scenarios which helps make analysis fast and easy |
| 14 | Setup separate log dashboards based on the distinct events which would provide lots of insights on the applications / services etc. |

# Design Patterns Summary

|  |  |
| --- | --- |
| **Design Pattern** | **Usage Scenarios** |
| Anti – Corruption Layer | This pattern can be used to ensure that a microservice design is not limited by its dependency on legacy systems. |
| Strangler | This pattern can be used for incrementally migrating a legacy system by gradually replacing specific business functions with new apps and microservices. |
| API Gateway | This pattern can be used to provide an additional layer of security, traffic management and to perform composition of experience APIs from domain / business APIs. |
| Back-end For Front-end | This pattern can be used in cases where you want to avoid customizing a single backend / domain service for multiple front–end interfaces. |
| Polyglot Persistence | This pattern can be used when different microservices need to persist different data types like – relational data, key-value pair, graph data etc. |
| Circuit Breaker | This pattern can be used for managing failure scenarios when interfacing with a remote service or resource. |
| CQRS | This is a domain driven design pattern which segregates the query / read operations from command / transactional operations. Both operation types will have exclusive data models with transactional updates getting synced with read data model. |
| Client Side Discovery | This pattern is used when we have multiple service instances configured with load balancing. When the consumer is able to discover the available service instance, it is referred as client side discovery. |
| Server Side Discovery | This pattern is used when we have multiple service instances configured with load balancing. When the consumer delegates the service instance discovery to a 3rd party, it is referred as server side discovery. |
| Near Cache | Frequently accessed data can be cached near the services using an external cache store as backing service. This will help microservice avoid repeated calls and improve the overall user experience. The cache implementation can follow Inline Caching OR Look-aside caching patterns  **Look Aside Cache:** This caching pattern provides more control to apps / data services as they will be responsible for cache management activities. Cache server / component will not have any code deployed.  **Inline Cache:** This caching pattern is an alternate to Look Aside Cache in which the data population is cache and cache data management is performed by the cache server / component. |
| Adapter | This pattern exposes the current SOAP / JMS services as REST APIs which then makes it compatible to be used within micro services architecture |
| External Configuration | This pattern utilizes an exclusive Configuration server backed by a source repository / file server in order to externalize the parameters that are expected to change across environments. This will ensure that the microservice application has access to needful configurations across all environments (Dev, Test, Prod etc.). |
| Distributed Tracing | This pattern implements distributed tracing mechanism to debug and analyze the issue in real-time using the trace-id and span-id. |
| UI Composition | This pattern enables full page UI to be decomposed into multiple UI fragments, each being processed using specific microservice. The UI page structure would contain the position and location where data from a specific microservice needs to be displayed. Individual micro service team would be responsible for developing the UI component / widget as per the specification provided |
| Blue – Green Deployment | This pattern is used to release the application in a predictable manner, with less down time. Also this helps rolling back to older version of the application in case of any issues in the new version |
|  |  |

# Design Patterns Details

* 1. Anti – Corruption Layer

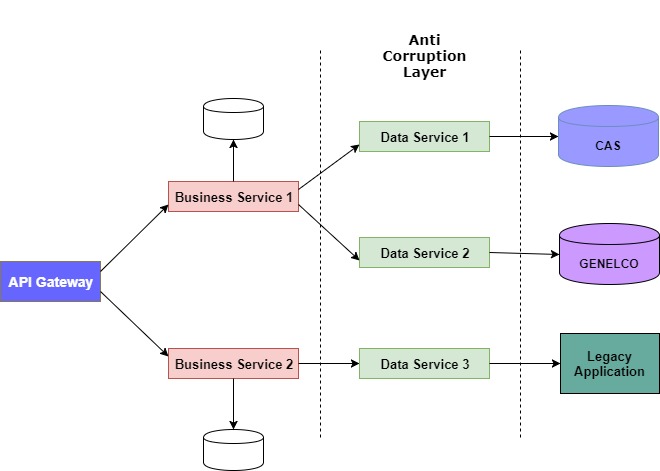
**Problem Statement**

*Micro services and Legacy systems (SoRs) have different semantics and migration / replacement of the core system might happen gradually over a period of time. Till then the integration between the new and core legacy systems needs to be maintained.*

**Recommended Solution**

Utilize the anti-corruption layer pattern to translate the communication to isolate the business micro services with the legacy systems ,so that the legacy system remains while the business micro services can avoid compromising its design / approach.

**Solution Design**



**Implementation Guidelines**

Implement the anti-corruption layer as an independent micro service. Ensure the anti-corruption layer contains all the necessary logic to translate the communication between the business micro services and the legacy systems

**Points to Consider**

* The new anti-corruption layer may add to latency
* This adds additional services that needs to be managed and maintained
* There may arise need to further decompose the functionality and hence the anti-corruption layer needs to be further partitioned
  1. Strangler

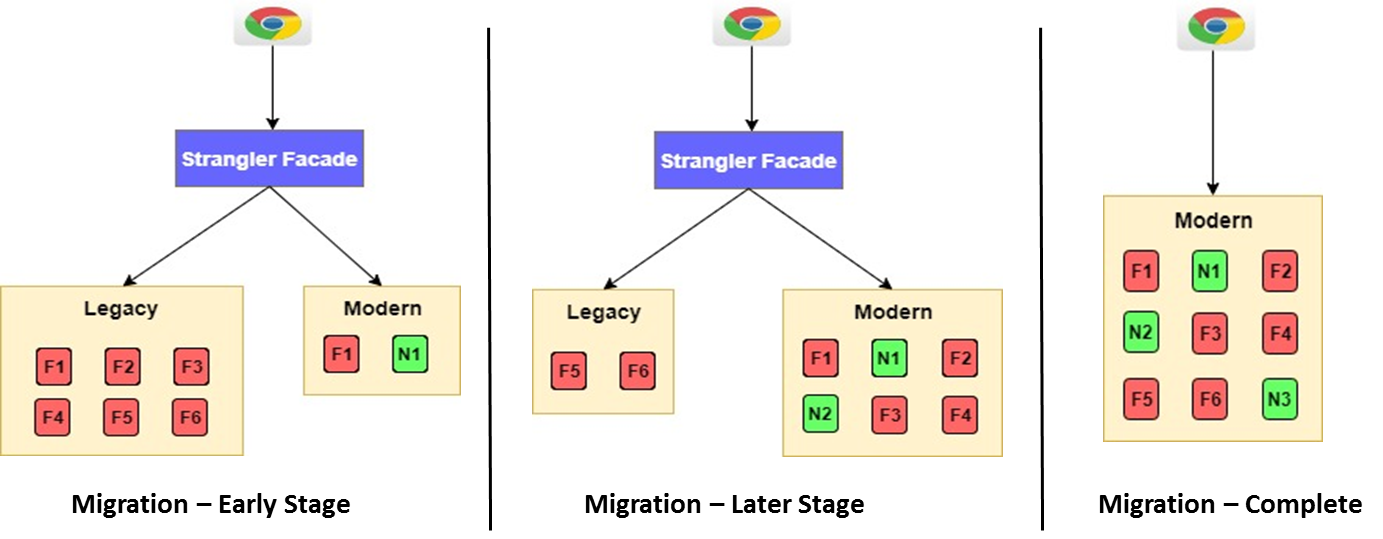
**Problem Statement**

*Have a big complex legacy system which needs to be replaced can become a huge exercise and might be riskier if planned to be done at one shot.*

**Recommended Solution**

Use the strangler pattern approach to incrementally replace functionalities one by one, thereby making the migration to new system gradually, consumers unaware of any migration and also minimize the risk.

**Solution Design**



**Implementation Guidelines**

Implement a façade which intercepts all requests going to the legacy system and route the request to legacy system or the new application. With the façade routing the users to correct application based on the functionalities, start migrating existing functionalities or add new functionalities

**Points to Consider**

* Ensure the data services can be accessed side-by-side by both legacy and new application
* Might introduce single point of failure or might create performance issues which needs to be properly managed
  1. API Gateway

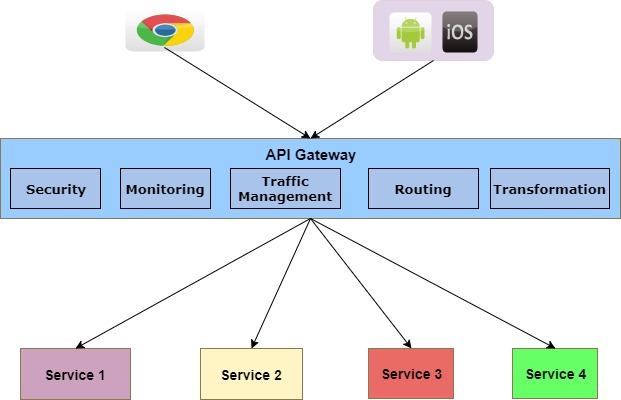
**Problem Statement**

*We may face scenario’s like, consumer needs to invoke multiple API’s to perform a single task, API interface and implementation getting changed overtime, different application clients (mobile / web) with different data requirements, several cross cutting concerns (Authentication, Security, Logging, API traffic management etc.) that needs to be handled in a centralized fashion and difficulties managing frequent deployments, release of API's.*

**Recommended Solution**

Use the API gateway pattern to simply proxy the request or perform orchestration, provide client specific API’s, perform transformation as required, apply security, metering API requests, easily switch traffic from old version to new version of API.

**Solution Design**



**Implementation Guidelines**

Use this pattern to reduce the chattiness between the UI and services, provide client specific APIs, perform centralized authentication, traffic management, Routing, API monitoring etc.

**Points to Consider**

* The API gateway should be located near the backend services to reduce latency
* The API gateway service may introduce a single point of failure.
* The API gateway requires separate effort for setup, configure, deploy and manage
  1. Back-End For Front-End

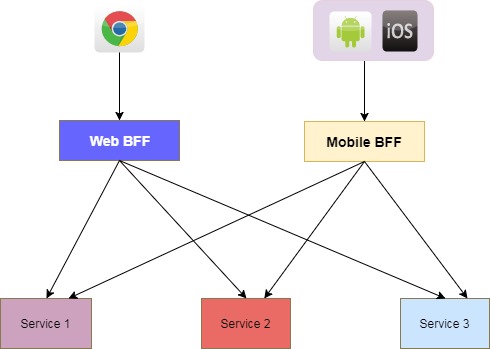
**Problem Statement**

*With different types of consumers (Web, Mobile, 3rd party applications), each consumers may have different data & processing requirement. Therefore it is difficult to provide one API which satisfies the requirement for different type of consumer*

**Recommended Solution**

Use the Backend for Frontend pattern which is a variant of API gateway pattern. It provides different types of clients with their own customized data based on their requirement.

**Solution Design**



**Implementation Guidelines**

Provide a separate BFF service for each type of consumer to provide API specific to that particular client (Mobile, Web, 3rd Party App etc.)

**Points to Consider**

* Exclusive BFF service for each client type requires effort for setup, configure, deploy and manage them.
* The BFF service may introduce a single point of failure.
  1. Polyglot Persistence (Decentralized Data Management)

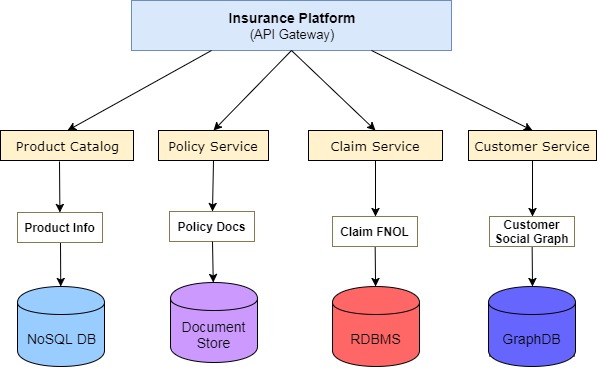
**Problem Statement**

*Different services will have different data storage needs. Some services need a relational database, other services might need a NoSQL DB for storing complex, unstructured data or Graph DB (like Neo4J) to efficiently store and query graph data.*

**Recommended Solution**

Each microservice should own and manage their data. Any external access to microservice owned data should only be allowed through provided interfaces.

**Solution Design**



**Implementation Guidelines**

Analyze the data requirement for the service and accordingly decide on the data storage technology (RDBMS / NoSQL / Graph DB etc.). Ensure data is accessible only via provided interfaces.

**Points to Consider**

* Implementing queries that need to join data belonging to different services will become challenging.
* Maintaining multiple SQL / NoSQL data stores will increase maintenance effort.
  1. Circuit Breaker

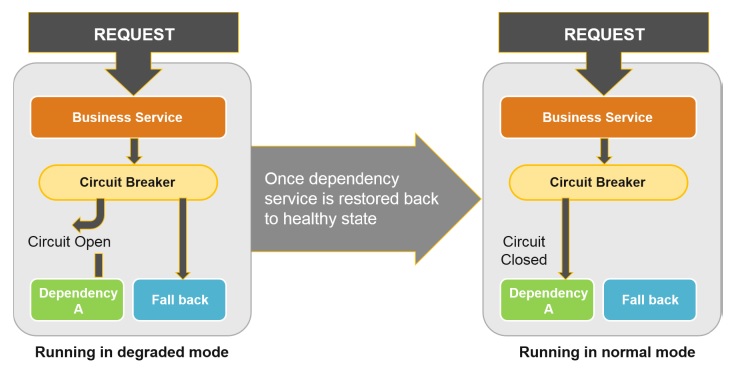
**Problem Statement**

*Micro services can be dependent on one another. When dependent services are not available or is not responding due to high load, the requests would start to wait and finally result in resource exhaustion. This results in cascading effect bringing down the whole system.*

**Recommended Solution**

To overcome the above application stability related concerns, always use a circuit breaker pattern when calling other services. This helps to handle service failures due to service unavailability or network issues.

**Solution Design**



**Implementation Guidelines**

Enable circuit breaker for the client code which calls other service. Implement and configure the fallback mechanism. Configure an optimal timeout.

**Points to Consider**

* Deciding about the optimal timeout could be challenging.
  1. CQRS (Command Query Responsibility Segregation)

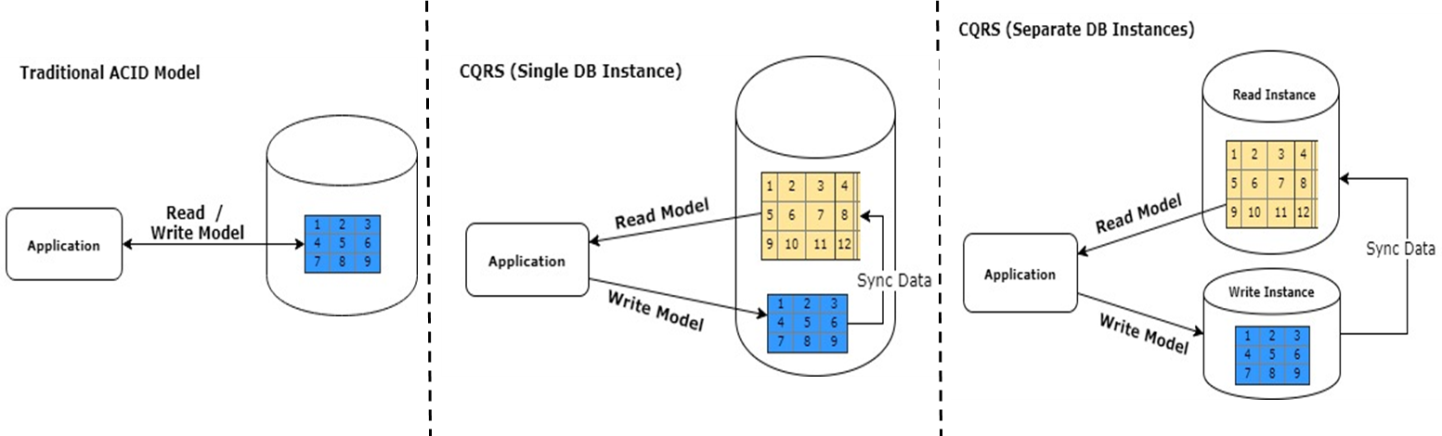
**Problem Statement**

*Traditional CRUD approach works well when limited logic is applied on data operations. But risk increases as the complexity and throughput of the system of the system grows and can have negative impact on performance due to load on data store, managing security becomes more complex and might also result in data contention when records gets locked when multiple users or actors work on same set of data.*

**Recommended Solution**

By utilizing the Command and Query Responsibility Segregation, we would be able to segregate the operations that read data (Query) and operations that write data (Command) with separate interfaces. With separate interfaces the data models can be made different for read and write thereby simplifying the design and implementation. Use multiple read only replica of read data store to greatly increase query performance and UI responsiveness.

**Solution Design**



**Implementation Guidelines**

Define separate interfaces and data models for read and write operations. Use same physical data store and use SQL views or generate projections on the fly. Further separate the data store for read with different structure and de-normalize the data

**Points to Consider**

* Consider CQRS at limited places where it is most required and valuable
* The read model store must be in sync with write model store else, user might further issue update to write model store based on stale data in read model store due to eventual consistency
  1. Client Side Discovery

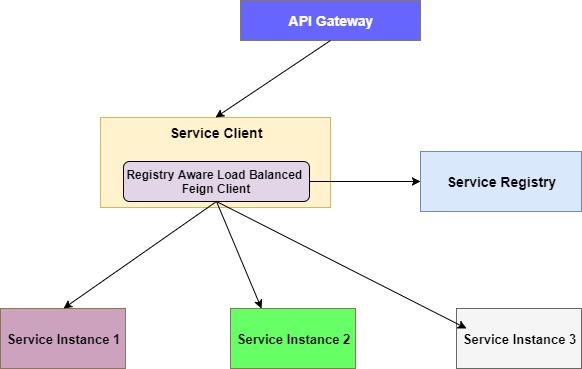
**Problem Statement**

*Micro services runs in a virtual / container based environments and the number of service instances change dynamically as new instances are created shutdown. Hence it would be difficult to make requests to dynamically changing services instances.*

**Recommended Solution**

Implement the Client-side Service Discovery pattern by making the client obtain the location of the service instance by querying the service registry. The service client will have the component (like Feign client) which is responsible for querying the service registry.

**Solution Design**



**Implementation Guidelines**

Register all the services with service registry. Make sure the client uses an load balanced instance of RestTemplate with Ribbon. Provide a timeout and fallback mechanism in case the actual service is not available or not responding

**Points to Consider**

* Since almost all services needs to be load balanced, it may result in creation of too many clients.
  1. Server Side Discovery

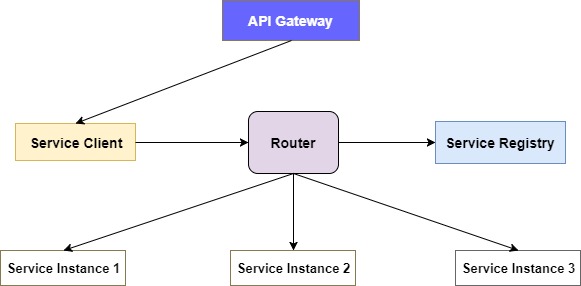
**Problem Statement**

*Micro services runs in a virtual / container based environments and the number of service instances change dynamically as new instances are created shutdown. Hence it would be difficult to make requests to dynamically changing services instances.*

**Recommended Solution**

Implement Server-side service discovery mechanism with service client making request with the help of a router which runs in a pre-defined location. The router then queries the service registry and forwards the request to service instances.

**Solution Design**



**Implementation Guidelines**

Register all the services with service registry. Ensure all requests go through the router. Configure timeout and fallback mechanism in router, case the actual service is not available or not responding

**Points to Consider**

* The router needs to be setup and configured separately if not available as part of the platform
* Extra hops required than client side discover
  1. Externalized Configurations

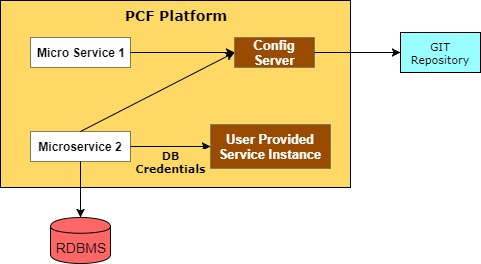
**Problem Statement**

*Micro services depend on several parameters that would change as the application moves on the deployment pipeline from dev, test and prod. As the application moves through different environments how do we manage the configurations between the environments without modifying the source code.*

**Recommended Solution**

To deploy same deployable unit across environments without change, utilize configuration server backed by a source repository / file server to externalize the parameters we expect to change across environments. This will ensure that application has everything to run properly across environments (Dev, Test, Prod etc.).

**Solution Design**



**Implementation Guidelines**

Configure the application to connect to Config Server. Maintain separate profiles for each environment and validate the configurations at startup to ensure the application has all the parameter to run properly

**Points to Consider**

* Use environment variables and user provided services instances in PCF to configure the application so that PCF can deliver these automatically at runtime
  1. Adaptor

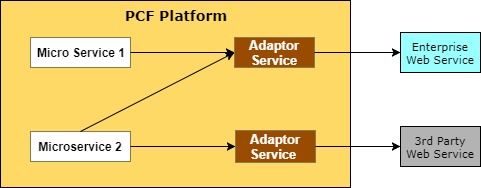
**Problem Statement**

*We have several enterprise web services (SOAP, JMS etc.) which is currently running without any issues and we want to reuse it as is.*

**Recommended Solution**

Utilize the adaptor pattern to expose the current services as REST APIs which then makes it compatible to the micro services architecture.

**Solution Design**



**Implementation Guidelines**

Convert the existing functional interfaces (verb based) to business oriented (nouns). Expose the existing SOAP operations as REST API’s with data format converted from XML to JSON.

**Points to Consider**

* Any changes to existing services would have an impact on the adaptor micro services
* Ensure circuit breaker is utilized to handle service unavailability
  1. Distributed Tracing

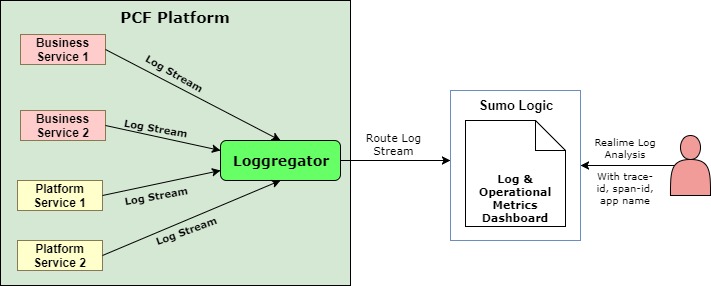
**Problem Statement**

*With micro service based architecture, services may handle a request by performing one or more operations or by invoking multiple services. Hence it might be difficult to troubleshoot or trace whenever an issue occurs.*

**Recommended Solution**

Implement distributed tracing mechanism using Spring Cloud Sleuth to debug and analyze the issue in real-time using the trace-id and span-id.

**Solution Design**



**Implementation Guidelines**

Add spring cloud sleuth to the project. Ensure each request from client has a unique request id sent as part of the header. Create custom queries based on different inputs and make it as templates for quick access and analysis

**Points to Consider**

* Ensure the traces are aggregated and sent to Sumo Logic
* Have predefined named queries to quickly fill the details based on some pre-conditions and inputs
* Utilize standard log format across micro services app
  1. Near Cache

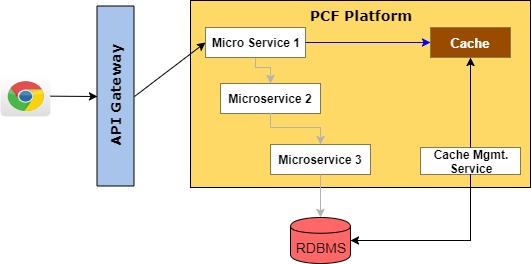
**Problem Statement**

*Due to the granularity of micro services there may be scenario’s where an application needs to make multiple service hops in order to retrieve data. This might result in performance latency and delay in request processing. Also there could be scenario where front end makes repeated service calls for same information.*

**Recommended Solution**

To address the above problem, data can be cached near the services using an external cache store as backing service. This will help microservice avoid repeated calls and improve the overall user experience.

**Solution Design**



**Implementation Guidelines**

Utilize the Cloud Cache component to cache the most frequently accessed data and other dependent data from the data stores. The cache implementation can follow Inline Caching OR Look-aside caching patterns.

**Points to Consider**

* The cache data must be invalidated at appropriate intervals
* Risk of users operating on stale data and further issue update to database
  1. Client Side UI Composition

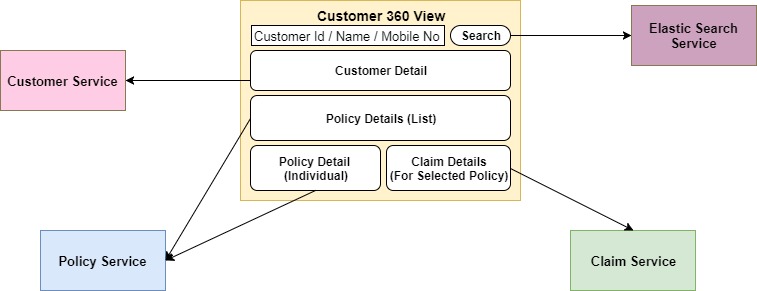
**Problem Statement**

*There could be scenario’s where a UI screen or page needs to show different sets of data which needs to be obtained from different micro services.*

**Recommended Solution**

Implement UI composition pattern which enables full page UI to be decomposed into multiple UI fragments, each being processed using specific microservice. The UI page structure would contain the position and location where data from a specific microservice needs to be displayed. Individual micro service team would be responsible for developing the UI component / widget as per the specification provided

**Solution Design**



**Implementation Guidelines**

The host application will include the front end of each service by conforming to standards. Utilize Client side identifiers to ensure all services are linked in some way. Invoke each service separately and display the data independently from client.

**Points to Consider**

* Needs multiple requests to fetch content
* Since UI is developed by different teams need to ensure consistency across all UI components.
* Versioning the individual UI components can become cumbersome.
  1. Blue Green Deployment

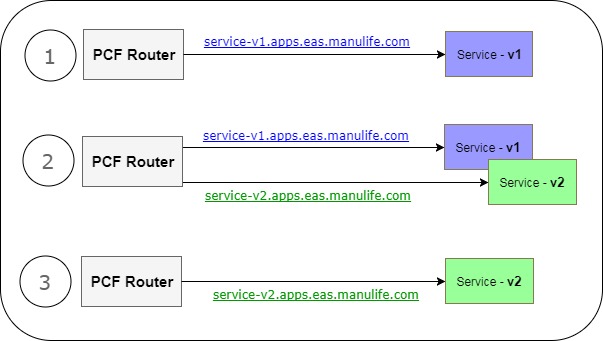
**Problem Statement**

*With more frequent releases and with almost zero downtime requirements, it would be difficult to release multiple application versions in a predictable manner. At the same time, in case of any issue in new release we should also be able to quickly rollback to the older version of the application.*

**Recommended Solution**

Utilize the Blue-Green deployment pattern to release the application in a predictable manner, with less down time. Also this helps rolling back to older version of the application in case of any issues in the new version.

**Solution Design**



**Implementation Guidelines**

Have two identical containers with blue hosting the current version and green hosting the new version of the application. Once the smoke testing of the new version is over, change the router to point to the green infrastructure.

**Points to Consider**

* Need to ensure that existing transactions in blue version completes successfully before the switch.
* Need additional infrastructure which might have an impact on cost.

# Naming Standards & Conventions

This may sound as very trivial aspect but can become a huge headache if not considered and done properly. If the names of things have to change throughout the project, every dependent system might get impacted and hence needs to be changed. When it comes to naming there are several areas which needs to be considered, right from

* **Pivotal Cloud Foundry Platform Setup**
* Org Naming
* Space Naming
* **APIGEE API Management & Gateway Platform**

Naming conventions for API’s needs to be decided upfront during the design phase and should be documented in the High / Low Level documentations. Following are the key areas which needs to be considered

* Artifact/API Naming
* Version Naming
* Scope Naming
* **Microservices Design**

Naming conventions for microservices needs to be decided upfront during the design phase and should be documented in the High / Low Level documentations. Following are the key areas which needs to be considered

* Project / Module Name
* Service Naming
* Group Naming
* Version Naming
* Managed Service Naming
* Route Naming
* **Microservices Development**

Naming conventions for Package, Class etc. can be decided upfront during the design phase and the names for methods and variables can be decided during the implementation if required.

* Package Naming
* Class Naming (Application, Controller, Service, Interface, Dao etc. based on layers)
* Method Naming
* Variable Naming
* Property Naming

When it comes to validating the conventions, we can utilize tools like Check Style to validate and enforce the standard naming conventions

* **SCM Configurations**

Naming conventions for Repository needs to be decided upfront during the design phase and for Branch, Feature and Release etc. should be planned during the Release planning and has to be created so that the team can start working and committing their changes.

* Repository Naming
* Branch / Feature / Release Naming
* **Application Packaging, Build & Deployment**

The naming conventions for build and deployment may not change frequently, but as and when new services are developed, these names should be decided during the design phase itself. At the same time the build and deployment scripts should be flexible enough so that with simple change in the configuration it is easy to build and deploy the services in respective environments without any support

* Build Pipeline Naming
* Release / Deploy Pipeline Naming

# Anti-Corruption Layer – Interfacing with Manulife HK Systems of Records

Domain Driven Design (DDD) stresses very much on the notion of bounded context. A bounded context can simply be understood as self-content component. Every domain model lives in precisely one Bounded Context (BC) and a Bounded Context contains only one domain model. It can interact with other components (BC) but it is coherent on its own. Cloud Native apps(Micro services) is always bounded to specific context, has its own domain model which belongs only to itself and should not leaked or be influenced by other micro services.

In Manulife context, the current scenario is not to do with 2 or more micro services but instead it’s with the interaction with on premise legacy systems. Mostly the cloud Native apps are dependent on legacy on premise systems. These on premise legacy systems have their own specific protocols and data models.

Cloud Native apps uses HTTP(S) as the protocol and may use altogether different model (Ex: OCDM - Manulife Standard Model comprises of ACCORD, Salesforce and Manulife specific models). The features and technologies used in on premise legacy systems vary widely from the features supported by micro services. In order to interoperate with these on premise legacy systems, sometimes the micro services design or technical approach may get compromised, forcing the micro services to adhere to protocols, API, data model etc. used by legacy application.

The solution is to utilize an Anti-Corruption layer to isolate the micro services and legacy on premise systems. Anti-corruption layer is used to enforce the core basic principle of “Every domain model lives in precisely one Bounded Context (BC)” and enabling interaction between 2 bounded context or legacy on premise application.

Key responsibilities of Anti-Corruption layer are -

* Translate communication messages across systems pertaining to different bounded contexts & context maps by mapping data attributes.
* Enable interoperability between 2 systems
* Host event driven data synchronization services for implementing eventual consistency.

Following are some of the key characteristics of Anti-Corruption layer

* A set of RESTful services will be designed as per micro services need to interface with SORs
* These services will implement Facade, Message Transformation and Adapter design patterns.
* These services will not contain any business logic.
* Another set of services will be implemented for addressing event driven data synchronization needs.

This layer will ensure that the legacy SORs will remain unchanged while the cloud native apps & microservices can have their own exclusive data models. Cloud native apps and services can be designed independently. This layer will enable the decoupling of cloud Native apps from On-Premise legacy system by avoiding any direct access / dependency

Key Points to consider when using Anti-Corruption layer

* There might be a slight increase in network latency which should be addressed by code optimization
* An additional set of REST services need to be managed and maintained
* Scaling of Anti-Corruption layer must be considered at par with Cloud hosted apps & services.
* Anti-Corruption layer will become part of overall solution stack along with other applications & services with respect to configuration, release, and monitoring processes.

# Implementation Best Practices & Guidelines

* 1. Generic Guidelines

|  |  |
| --- | --- |
| **No** | **Generic Guidelines** |
| 1 | “Micro” in the Microservice is about scope and not the size of codebase / deployable unit. |
| 2 | Microservices design should ensure agile / independent DEV and deployment activities. |
| 3 | Start with coarse grained services. Refactor them to smaller ones using an incremental approach |
| 4 | Microservices should be packaged independently including all possible dependencies. |
| 5 | Microservices implementation should follow smaller release cycles |
| 6 | Microservices should have no / minimal dependency on external environment & components |
| 7 | DevOps team should ensure consistency across different (DEV / QA / PROD) environments |
| 8 | Microservices implementation should follow CI best practices for producing high quality artifacts |
| 9 | DevOps team should implement fully automated Continuous Delivery pipeline for enabling frequent releases with zero downtime. |
| 10 | IaaS and PaaS platforms should enable automated provisioning and management of resources |

API Design Guidelines

|  |  |
| --- | --- |
| **No** | **API Design Guidelines** |
| 1 | Follow Swagger / Open API standards for defining API design specifications |
| 2 | Resources should be named using nouns, actions as HTTP verbs |
| 3 | The API URL paths should be as short as possible |
| 4 | Always version the API's as per the defined API versioning strategy |
| 5 | Ensure the API returns proper HTTP status codes and error messages |
| 6 | Ensure that the API always returns the latest version of data representation |
| 7 | Make use of different set of query parameters for different service consumers |
| 8 | Provide pretty print by default along with gzip or via query parameter |
| 9 | Implement Filter, Sort as query parameters based on predefined conventions |
| 10 | Make use of camel Case as naming convention for field names |

Service Design Guidelines

|  |  |
| --- | --- |
| **No** | **Service Design Guidelines** |
| 1 | Micro services should be scope around bounded contexts and should be aligned to domain driven design principles |
| 2 | Each service should be bounded to a single context and should include highly cohesive entities |
| 3 | Services design should follow no / minimal coupling. All dependencies should be self-contained |
| 4 | Services should always be discovered from service registry rather than direct invocation |
| 5 | Any external collaboration should follow asynchronous (message based) communication for optimal performance and load management |
| 6 | Services should be stateless. They should use external cache to persist session data if required |
| 7 | Data model should be designed specific to service scope enforcing decentralized data management |
| 8 | Services design should follow eventual consistency. This will help to scale and deliver optimal performance. |
| 9 | Service design should follow Circuit Breaker & Bulkhead patterns for better resiliency and fault management. |
| 10 | External resource requirement should be met by using cloud native backing service instances. |

Development Guidelines

|  |  |
| --- | --- |
| **No** | **Development Guidelines** |
| 1 | Service applications should have a single codebase. Every update in the codebase should be exclusively versioned |
| 2 | Make sure that API design specification is available before developing the micorservice |
| 3 | Automate application dependency management using tools like Maven |
| 4 | Define YAML files for externalizing app configurations. No configuration in source code files. |
| 5 | Define and enforce coding standards and best practices for respective technology & framework |
| 6 | Setup and configure CI pipeline for generating builds and automating code quality checks |
| 7 | Ensure that application can be started or stopped gracefully. Implement fallback mechanism for every service |
| 8 | Developers should perform a quick smoke test of developed service by hosting it in DEV region |
| 9 | Follow TDD / BDD based development approach. Unit testing should cover minimum 80% of the source code. |
| 10 | Setup and configure IDE with needful plugins for realizing DevOps best practices |

Integration Guidelines

|  |  |
| --- | --- |
| **No** | **Integration Guidelines** |
| 1 | Utilize REST for synchronous invocation of services and utilize AMQP / MQ for message based asynchronous communication across services |
| 2 | Utilize EIP (Enterprise Integration Patterns) libraries like – Apache Camel OR Spring Cloud Streams for implementing message based integrations. |
| 3 | Follow “Smart Endpoints and Dumb Pipes” design principle while using Message Brokers |
| 4 | Make use of JSON as standard format for exchanging messages across services |
| 5 | Make use of Cloud native event stores while implementing event based integration use cases |
| 6 | Accept all incoming attributes but validate only those as needed by the integration use case |
| 7 | Avoid volumetric data payload while interfacing with other services. |
| 8 | Define acceptable timeout for all kinds of interactions to avoid indefinite wait for response |
| 9 | Implement circuit breaker and define time limits to prevent timeout of consuming service |
| 10 | Implement Bulk Head pattern to continue functioning even when non-vital services are down |

Testing Guidelines

|  |  |
| --- | --- |
| **No** | **Testing Guidelines** |
| 1 | Follow TDD approach. Write unit test cases for validating microservices source code |
| 2 | Make use of code coverage tool to measure test coverage and ensure that it is always over 80% |
| 3 | Make use of mock objects when testing operations or external service integration |
| 4 | Make use of separate test data stores for request, mapping and validation specific data |
| 5 | Run security scans on source code to identify and fix security vulnerabilities |
| 6 | Perform consumer driven API testing to validate microservice provided interfaces |
| 7 | Perform component testing to validate microservice as one of the product component |
| 8 | Perform fault tolerance testing for deployed service instances using tools like Chaos Monkey. |
| 9 | Perform stress / Load testing to evaluate service ability to handle unexpected loads. Identify business critical microservices and conduct load testing in isolation. |
| 10 | Perform recoverability testing to assess service resiliency, fast startup and graceful shutdown |
| 11 | Execute Performance testing by grouping the microservices as per the business transactions and then benchmark the performance. |
| 12 | Utilize OAuth / OAuth 2 for authenticating user identity and validating access levels. |
| 13 | Perform vulnerability / security testing at application, database and network layers |
| 14 | Make use of security scanners for container based microservices deployment. |
| 15 | Perform testing of overall runtime ecosystem under varying fault / error conditions |

Deployment Guidelines

|  |  |
| --- | --- |
| **No** | **Deployment Guidelines** |
| 1 | Create immutable, self-contained artifacts to ensure deployment consistency across regions |
| 2 | All deployed service instances should utilize PCF managed backing services (DB, Cache, MQ etc.) |
| 3 | All deployed service instances should leverage PCF managed Config server and service registry |
| 4 | PCF and APIGEE platforms should be configured with appropriate level of security at application, data and network layers |
| 5 | Enterprise services (Email/ SMS, A&A etc.) should be available before deploying services |
| 6 | Register API proxies, products and policies in APIGEE Edge cloud platform |
| 7 | Setup and configure Sumo-Logic to extract and collate logs generated over PCF platform |
| 8 | Setup and configure CD pipeline to automate service deployment process |
| 9 | Ensure that all hosting environments (DEV / QA / PROD) over PCF have similar configurations |
| 10 | Finalize on Deployment Strategy. Per Container service deployment is recommended deployment strategy. Follow PCF guidelines for service deployment. |

Operational / Lifecycle Management Guidelines

|  |  |
| --- | --- |
| **No** | **Operational / Lifecycle Management Guidelines** |
| 1 | Follow policy driven approach for service monitoring (health condition, unhealthy criteria etc.) |
| 2 | Leverage PCF and APIGEE provided centralized monitoring and operational diagnostics features |
| 3 | Leverage PCF platform provided automated error management features |
| 4 | Leverage PCF Auto-Scaler for dynamic provisioning for service instances for peak load scenarios |
| 5 | Define optimal scaling configurations. Define policies to automate the scaling process |
| 6 | Define KPIs (key performance indicators) for monitoring deployed service instances |
| 7 | Define best practices for operations management over APIGEE & PCF platform |
| 8 | Implement automated notification of errors & operational issues to relevant stakeholders |
| 9 | Collect operational metrics and analyze them for identifying potential improvement areas |
| 10 | Ensure that PCF and APIGEE platform components provide zero downtime operations. Make use of blue-green deployment approach while releasing a new service version in production. |