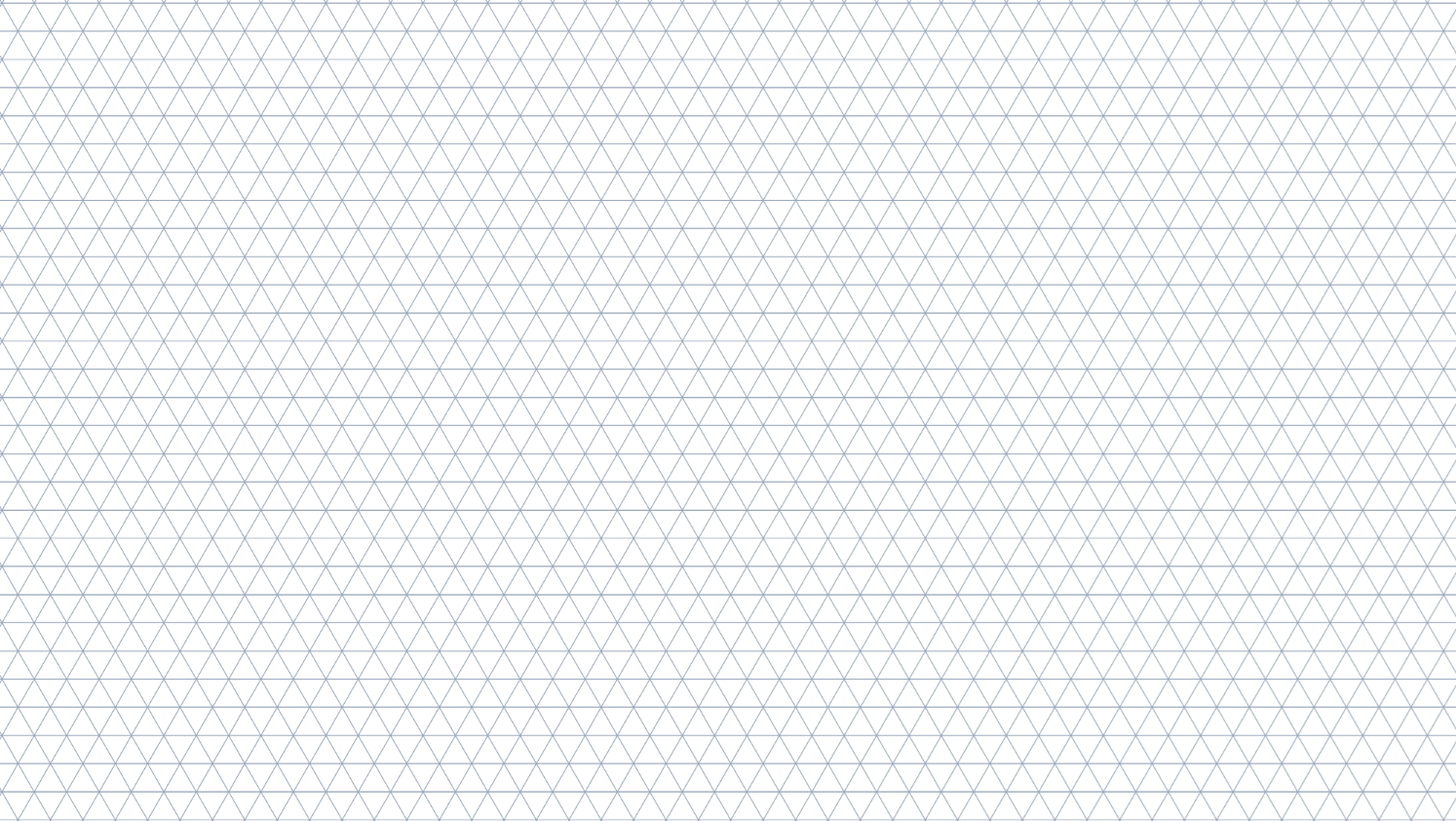
https://www.banamex.com/citibanamex/img/citibanamex.png



CITIBANAMEX API AND MICROSERVICES ARCHITECTURE

Reference Document – DRAFT V2.0

May 2017

**Table of Contents**

[1. Major organizations are undergoing a journey of transformation from monolithic to microservices architecture 3](#_Toc481660519)

[1.1 API and Microservice definitions 3](#_Toc481660520)

[1.2 Advantages of a Microservice-driven development approach 4](#_Toc481660521)

[1.3 CitiBanamex Microservices Architecture Guiding Principles 6](#_Toc481660522)

[1.4 High Level Logical Architecture 8](#_Toc481660523)

[1.5 Current State 9](#_Toc481660524)

[1.6 Strategic Architecture – Phase Migration from Monolithic to Microservices Architecture 10](#_Toc481660525)

[1.6.1 Phase 1 – Begin by building pass through services 10](#_Toc481660526)

[1.6.2 Phase 2 – Introduce datastores to microservices 10](#_Toc481660527)

[1.6.3 Phase 3 – Migrate ESB functionality and sync data source to backends 11](#_Toc481660528)

[1.6.4 Phase 4 – Make microservice source of truth, sunset backend 11](#_Toc481660529)

[1.7 Detailed Architecture View 13](#_Toc481660530)

[1.8 Macro-architecture considerations for microservice runtime 14](#_Toc481660531)

[1.9 Micro-architecture considerations for microservice development 15](#_Toc481660532)

[1.10 No SQL databases - a key considerations as it drastically enables ability to develop web-native application 18](#_Toc481660533)

[1.11 Evaluating NoSQL Database – there are variety of dimensions to consider in choosing the right NoSQL solution 19](#_Toc481660534)

[2 DevOps Pipeline 20](#_Toc481660535)

[3 API Design Cookbook 23](#_Toc481660536)

[3.1 12-Factor Apps Design Principles for Microservices on PCF: 23](#_Toc481660537)

[3.1.1 Codebase — one codebase tracked in revision control, many deploys 23](#_Toc481660538)

[3.1.2 Dependencies — explicitly declare and isolate dependencies 23](#_Toc481660539)

[3.1.3 Config — Store config in the environment 24](#_Toc481660540)

[3.1.4 Backing Services — Treat backing services as attached resources 24](#_Toc481660541)

[3.1.5 Build, release, run — strictly separate build and run stages 24](#_Toc481660542)

[3.1.6 Processes — execute the app as one or more stateless processes 25](#_Toc481660543)

[3.1.7 Port binding — Export services via port binding 25](#_Toc481660544)

[3.1.8 Concurrency — Scale out via the process model 25](#_Toc481660545)

[3.1.9 Disposability — Maximize robustness with fast startup and graceful shutdown 26](#_Toc481660546)

[3.1.10 Dev/prod parity — Keep development, staging, and production as similar as possible 26](#_Toc481660547)

[3.1.11 Logs — Treat logs as event streams 26](#_Toc481660548)

[3.1.12 Admin processes — Run admin/management tasks as one-off processes 27](#_Toc481660549)

[3.2 Microservices Design Patterns — Functional decomposition 27](#_Toc481660550)

[3.2.1 Aggregator Pattern: 28](#_Toc481660551)

[3.2.2 Proxy Pattern 29](#_Toc481660552)

[3.2.3 Chained microservice design pattern 29](#_Toc481660553)

[3.2.4 Branch microservice design pattern 30](#_Toc481660554)

[3.2.5 Shared Data Design Pattern 31](#_Toc481660555)

[3.2.6 Asynchronous Messaging Design Pattern 31](#_Toc481660556)

[3.3 Agile Sprints - Definition of Done (DoD) 32](#_Toc481660557)

[4 API Security Considerations 35](#_Toc481660558)

[4.1 Microservices Security - tools 36](#_Toc481660559)

[4.2 API Gateway Security 39](#_Toc481660560)

[5 High Availability on Cloud Foundry 42](#_Toc481660561)

[5.1 High Availability and Scalability in the platform 42](#_Toc481660562)

[5.1.1 Availability Zones 42](#_Toc481660563)

[5.1.2 Application instances 42](#_Toc481660564)

[5.1.3 Example Scenario 42](#_Toc481660565)

[5.2 Best practices for high availability and scalability 44](#_Toc481660566)

[6 Platform-As-a-Service key concepts 47](#_Toc481660567)

[6.1 Provisioning of API/Microservices on PCF (cloud/on prem) 47](#_Toc481660568)

[7 Main API Policies 49](#_Toc481660569)

[7.1 Access to input properties code snippet 50](#_Toc481660570)

[7.2 Access to runtime context code snippet 50](#_Toc481660571)

[7.3 Access to input payload in XSLT code snippet 51](#_Toc481660572)

[7.4 Access to HTTP headers code snippet 52](#_Toc481660573)

[7.5 Modify the payload code snippet 53](#_Toc481660574)

[7.6 Configure error information code snippet 53](#_Toc481660575)

[7.7 Set variables code snippet 54](#_Toc481660576)

[8 API Naming Standards and Guidelines 55](#_Toc481660577)

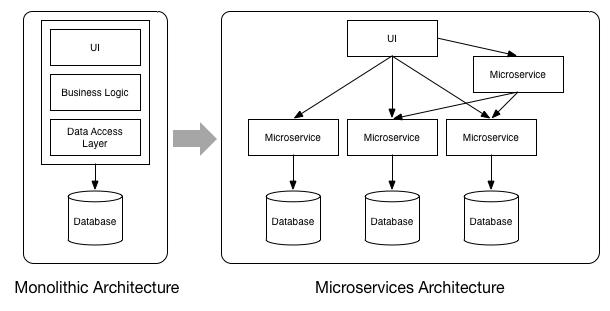
# Major organizations are undergoing a journey of transformation from monolithic to microservices architecture

## API and Microservice definitions

|  |  |
| --- | --- |
| **Concept** | **Description** |
| Microservice | A microservice is an isolated, loosely coupled unit of development that works on a single concern. This definition must guide the solution design: a new microservice should be developed for every functionality following the Single responsibility Design Principle (<https://8thlight.com/blog/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>). Microservices are associated with a contract that define their operations, methods and behaviors. |
| API | An API (Application Programming Interface) is a specification for how two software components should interact. For CITI Banamex an API will be defined as a microservice providing a single contract with specific parameters. |

The Microservice architectural style is an approach to developing an application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

These services are built around business capabilities and independently deployable by fully automated deployment mechanism. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.



## Advantages of a Microservice-driven development approach

|  |  |  |
| --- | --- | --- |
| **Advantage** | **Description** | **Examples** |
| Strong modularization | Microservices are loosely coupled and carry out one specific functionality well. Communicate via interfaces (e.g. REST). | An ATM locator service will perform that specific function independent of other services, with ease of integration and providing greater availability and reliability than monolithic services |
| Easy replaceability | Can be easily replaced with interfaces staying the same. | Swap out services when newer technologies are to be used. E.g. a Java microservice can be moved to a Node.js implementation as long as the interface remains same, this is not possible in an ESB or Appserver container |
| Sustainable development | The above means that long term efficient development of the applications is sustainable. With monoliths productivity decreases over time as the architecture is eroded and issues / inefficiencies creep in. monoliths are efficient at small changes (everything is in one place) but microservices are efficient at scaling and availability and force de-coupling of components (improving pace of large changes). | Monoliths like ESB (TIBCO, Tuxedo) has to be completely replaced to deploy scalable services and APIs. In microservices architecture, services are autonomous and independent so can be replaced without affecting larger ecosystem of services |
| Time to market | Time to market is quicker as different teams in parallel with independent development timelines, independent deployment schedules, independent scaling and maintenance, can build services. | In an ESB, all services have to be deployed in one single container and interdependencies mean releases are slower. (e.g. quarterly release window). Microservices deployment units and release are independent of each other |
| Free choice of technologies | Do not need to build applications in the same technology. Services can be built using different technologies. New technologies can be experimented with brought in safely and specific services in the application can be swapped out separately | There is no lock in of technology. Each API and microservice can use a different technology since they are independent (e.g. mixture of Java, Node.js, python), as long as RESTful interfaces are adhered to |
| Independent scaling | Each microservice can be scaled independently of other services avoiding cost of scaling components unnecessarily | Scaling a monolith is costly and entire container with all services in it has to be scaled horizontally and/or vertically. Microservices can be deployed with elastic scaling meaning busier ones will spawn off more instances and then release resource once traffic slows down |
| Continuous delivery | Services can be deployed independently of each other and changes can be made much more frequently with lower risk | CI/CD automation means services can be deployed even several times in a day, a far cry from monolithic deployments (e.g. quarterly) |
| Stateless | Migration from stateful services of existing core banking systems, developed from highly process oriented terminals, to stateless, multi channel architecture | many of the mainframe and backend process entail stateful interactions (e.g. load data, make change, store data) that can be migrated to independent stateless microservices orchestrated via RESTful interactions |

## CitiBanamex Microservices Architecture Guiding Principles

|  |  |  |
| --- | --- | --- |
| **Dimension** | **Impact** | **Example** |
| Cost | Create agile & cost efficient IT architecture which responds quicker to changing customer needs | Automation and agile delivery means features can be introduced at speed adapting to customer needs and choices |
| Standardization | Use leverage effects (e.g. shared components) and align to CitiBanamex enterprise standards and Digital Transformation strategies; Standardization of approach by adopting common principles and guidelines | API Governance ensures adherence to standards so APIs are easily consumed and also provides visibility to wider organization on existing APIs so projects can make decisions whether they can reuse an API or if there is a need to place a request for a new API. |
| Development Speed | Create/refactor applications for speed of change by using agile development methods and innovative technologies | Release cycles and time to market to introduce a new feature currently is 3+ months which can be reduced to 1-2 weeks |
| Deployment frequency | Embrace DevOps culture and automate software delivery pipeline | Automation through Devops help speed release cycles and reduce time-to-market |
| Lightweight | Create cloud native applications deployed in lightweight runtime containers | Monolithic architectures like ESBs cannot scale up to the demand of digital use cases like serving mobile channels in a responsive manner or catering to sensor data streams from IoT devices etc. having microservices deployed in lightweight containers can help them scale to meet the availability and performance needs of channel apps and devices. |
| Open | Leverage Open Source and innovative Cloud technologies; Design lightweight and reusable APIs based on open standards and modern technologies for an API-based ecosystem | Architecture is flexible enough so that components can be swapped and replaced as needed. Open source software and use of cloud reduced over cost of ownership and operations of software and infrastructure. No vendor lock in. |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Developer Notes:**

**Cost**: Use of open source technologies and components for cost effectiveness, agility and speed.

Standardization: Get design approved by Governance body to ensure standardization of design. Governance council will also have view to enterprise APIs and can suggest if an API already exists or a new one to be built.

**Development Speed and deployment frequency**: Build, test and deploy functionality incrementally and often and publish to test env. frequently. Devops tools will ensure regression testing and reporting so nothing is broken when pushing incremental functionality ahead.

**Lightweight**: Build APIs for performance and deployment on lightweight containers. Analyse use cases and code to determine resource needs for the functionality being coded and configure container manifest files with optimal values. Using generic values in manifests for every API may result in unnecessary wastage of memory or compute resources.

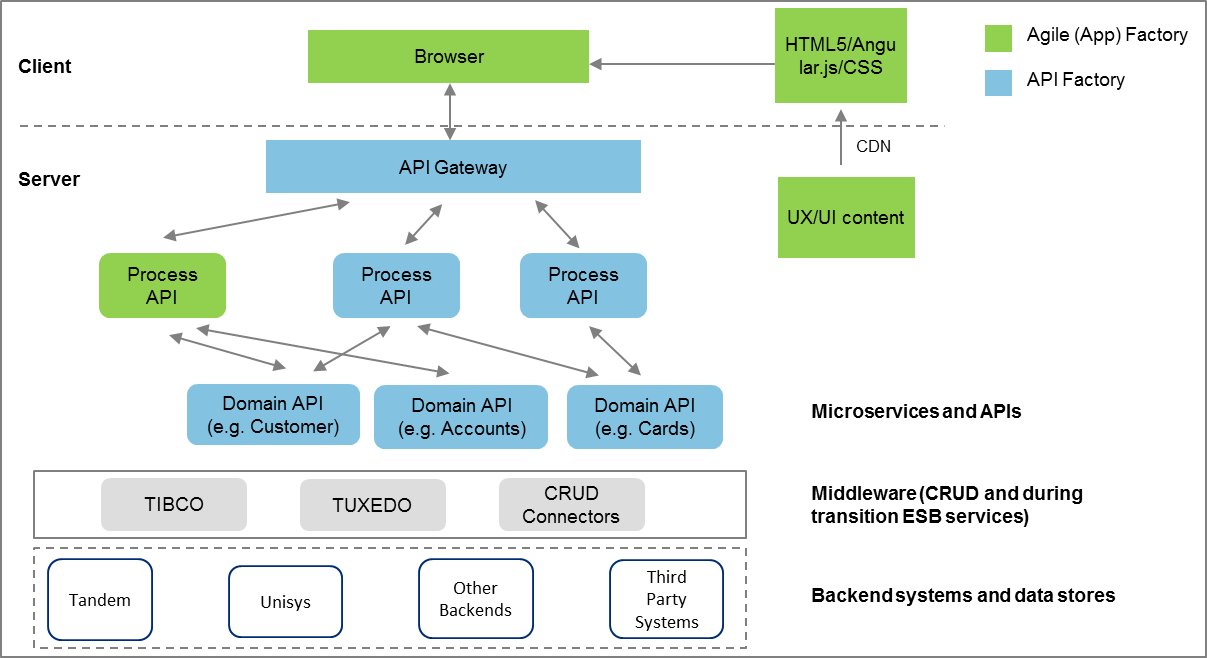
**Open**: Stick to open standards (e.g. REST/JSON) so architecture is independent of its components which can be swapped if needed to accommodate new innovation, higher performance etc. These changes can be effected easily if open standards are followed.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## High Level Logical Architecture

The following diagram depicts the end-to-end high level logical architecture

**Logical Architecture**



**Definitions**

**Process APIs**: term used to describe more coarse grained APIs that orchestrates multiple fine grained APIs to implement a business use case. Orchestration requirements will often originate from projects delivering specific functionality.

**Domain APIs**: term used to describe atomic/fine grained APIs that operate mainly on domain object models and microservices datastores performing CRUD operations. These APIs have high reusability and will be used by multiple process APIs.

**Description**

The layers allude to the different work streams and skills needed to build the end-to-end user journeys. The UI/UX layer would involve building mobile and web apps conformant to MVC principles and might leverage JavaScript frameworks like Angular.js, React.js, JQuery etc.

The middle layer consisting of the API Gateway and Microservices layer will be the responsibility of API factory team. The development of Microservices will involve the domain based data modeling, resource modeling, coding business logic, deploying in CloudFoundry environment through Devops pipeline, build, automated unit tests and deployment onto target environment.

There are multiple connection technologies to backend systems – CRUD Connectors, TIBCO and Tuxedo are the main near term targets. The Microservices built by the API Factory will abstract all of these connectors and present a rationalized and easy to use "resource model" view of all back-end systems via a RESTful design (not just RPC-over-HTTP).

Any API development should be preceded by an analysis and design session with participation of backend SMEs to identify systems in context and interface needs with ongoing collaboration through the development cycle.

## Current State

The current state conceptual model is shown in the diagram below. The current state is highly complex interaction of monolithic components and legacy, making the model inflexible, costly and difficult to maintain, high cost of new changes due to cascading effects of interface changes or data model changes, difficult and costly to scale, costly and complex to enhance.

## 

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Developer Notes:** A few monolithic containers that are constrained to scale are highlighted in the red boxes. While ESBs (TIBCO) and Appsevers (Weblogic) can achieve functional decoupling of services, they still get deployed to one big container and issues with one service like memory hogs will affect another service in the same container thus making the services in effect inter-dependant from resource sharing and performance point of view.

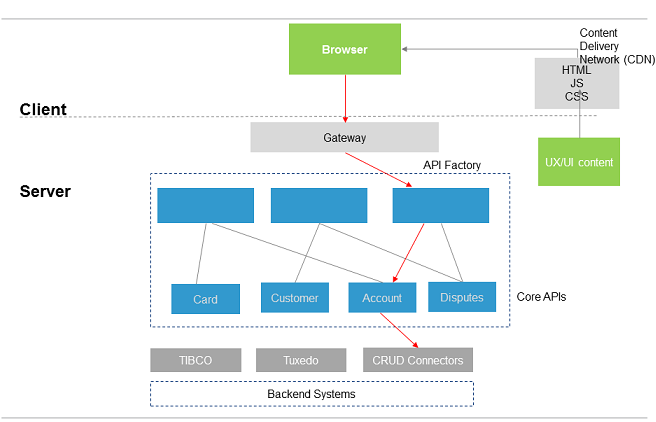
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## Strategic Architecture – Phase Migration from Monolithic to Microservices Architecture

The overall strategic architecture for CitiBanamex is to move to API and Microservices based model, that can scale to the needs of new age Digital use cases including IoT and AI with services having to support data streams often than discreet API calls. The monolithic architecture will be decomposed in phases and transformed to a Microservices architecture. The 4 main phases this will the accomplished are highlighted here:

### Phase 1 – Begin by building pass through services

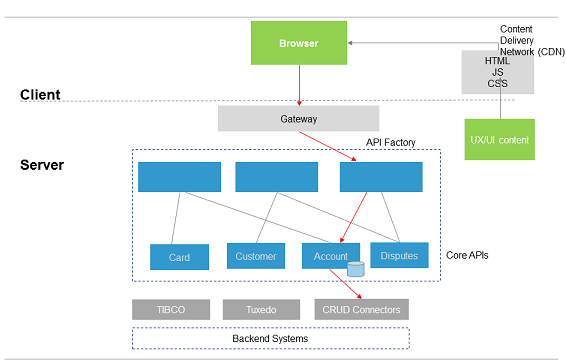
Phase 1 will build out Microservices deployed on a PaaS platform that has the advantages of autoscale. However, they will integrate to existing backend for CRUD operations and business logic and not store any data locally. However, high speed caching components can be used for certain use cases. The following diagram shows a pass through flow:



### 

### Phase 2 – Introduce datastores to microservices

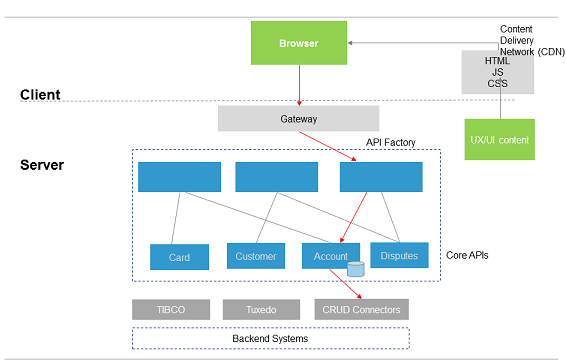
Phase 2 will build new Microservices and convert some of the existing ones talking to the backend to use local datastore (ODS) to store data, mainly for Read operations. Backend continues to be the system of records and all Update operations still invoke backend services via TIBCO/TUXEDO/CRUDConnectors. Domain specific data modeled and stored in Microservices store. Any backend updates to same domain model needs to be synched to MS store



Datastore added for microservices to store data locally and sync with backend book of records. Mainly read operations on DB in this phase.

### Phase 3 – Migrate ESB functionality and sync data source to backends

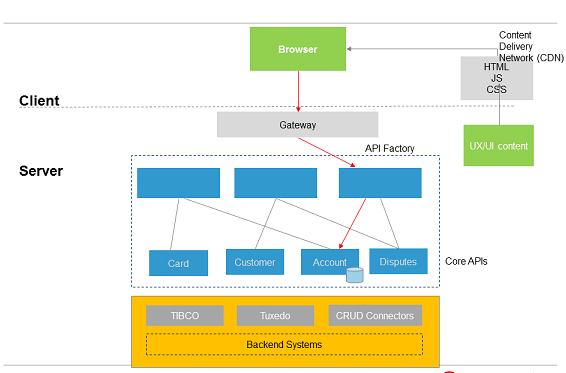
Phase 3 will look to migrate more and more existing business logic in backend and ESB services to Microservices. Data models will be defined based on domains like accounts, cards, customers, transactions etc and domain specific Microservices developed talking to corresponding storage. All clients/apps that "write" data to the back-ends need to be migrated to call the microservices. At the end of this phase, the microservice becomes the source of truth, and legacy back-ends would be read-only.



Both read and updates handled by microservices in this phase. More and more ESB services migrated to MS layer and microservices domain models extended.

### Phase 4 – Make microservice source of truth, sunset backend

Phase 4 will look to assure complete migration of ESB services and backend functionality to Microservices layer. At this point domain Microservices will work with own domain specific storage and all cross-domain APIs will have been mapped and defined as well. Backend is planned for sunset after ensuring any residual processing is addressed.



ESB services moved to microservices layer and all CRUD happens to domain models in microservices store. Backend disconnected and phased for decommission

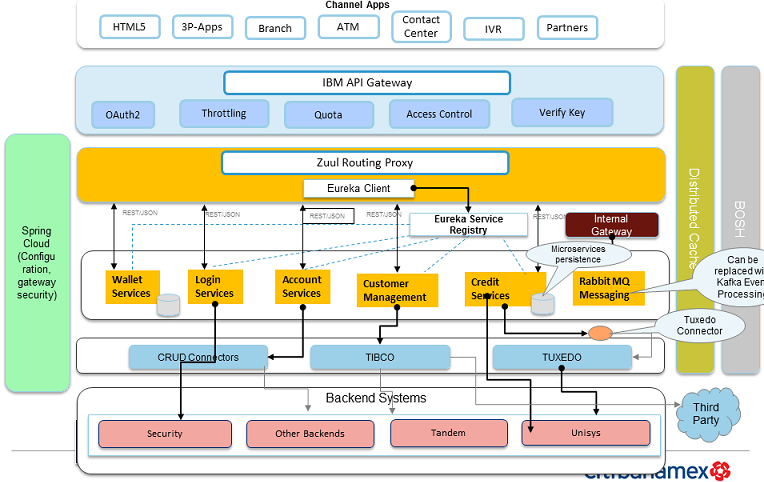
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Developer Notes:** Phase 4 is the strategic architecture vision. At this stage we have achieved truly autonomous microservices based architecture that can scale enormously and provide a platform for innovation and customer experiences based on big data analytics, AI, machine learning etc and provides the platform with computing prowess and efficiency for the same. API Factory design should inculcate the strategic vision and follow these rules in the previous phases:

1. Analyse tactical vs strategic components in service design. Spend more time and effort on strategic components and less on tactical/throw away code.
2. Programs/projects already using some legacy interfaces may prefer to keep it that way. Resist the urge to deviate from strategic architecture e.g. a service maybe available on TIBCO providing integration to an existing backend system. However, the use case being served will be constrained in terms of scale and performance due to the constraints of the weakest scaling component in the chain.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## Detailed Architecture View



The detailed architecture elaborates on the high level layered architecture with component product mappings depicted. Microservice would make use of caching services like Redis for certain use cases specially ones dealing with more static data. Kafka is a high throughput message bus from Apache that can handle event streams and event driven patterns extremely efficiently. In the current state Rabbit MQ will be used for asynchronous messaging and event driven patterns. As indicated in the transition phases, the endeavour over time would be to move away from TIBCO and TUXEDO interfaces and build CRUD connectors integrating to legacy so monolithic services can be migrated gradually into microservices. Zuul is an edge service that provides dynamic routing, Static Response handling, resiliency, security and insights. Eureka is a REST based service that is primarily used in cloud for locating services for the purpose of load balancing and failover of middle-tier servers.  Eureka is provided in spring cloud for microservices to register and be discovered.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Developer Notes:**

* Overall architecture roadmap is to transform from monolithic containers to microservices architecture deployed on elastically scalable lightweight containers for advantages of scale, speed of delivery, autonomous services and technology independence.
* Take advantage of Polyglot persistence in design by choosing microservices datastores appropriate to the use case being implemented.
* Make use of caching (Redis) especially for non-volatile data reads.
* **Circuit Breakers:** It is a library that implements the circuit breaker pattern isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve the system’s overall resiliency.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## Macro-architecture considerations for microservice runtime

Agile Microservice development with quick feedback cycles requires a dedicated runtime environment. On the macro (or “outer”) architecture level, the following aspects/concerns must be addressed in a uniform way, compliant with existing and/or emerging enterprise standards:

|  |  |
| --- | --- |
| **Consideration** | **Technology** |
| Quick and simple deployment of Microservices via containers, no installation of software components on target systems, container must provide everything required to run the service. | Pivotal CloudFoundry PaaS  Image result for pivotal cloud foundry |
| Clustered container execution environment that allows transparent distribution of containers between cluster nodes and elastic auto-scaling (start-up and shutdown of container instances) depending on load requirements | Pivotal CloudFoundry PaaS  Image result for pivotal cloud foundry |
| Mechanism for service registration, service discovery and service configuration | Eureka  Image result for eureka netflix |
| Mechanism for synchronous interaction between services: RESTful HTTP w/ JSON payloads. Internal communication between Microservices theoretically could use low-latency RPC (Thrift, protobuf, etc) | JSON, low latency RPC  Image result for JSON |
| Mechanism for asynchronous interaction between services, domain event propagation, publish-subscribe: near term choice AMQP-based message broker, e.g. RabbitMQ, future choices might be Kafka (better messaging than RabbitMQ) and Akka (actor model helps develop async services) | RabbitMQ, Akka  Image result for rabbitmq |
| Centralized collection, storage and analysis of log data: Splunk/ELK/Logstash | Splunk, ELK, Logstash  Image result for splunk |
| Real-time monitoring of both technical and business metrics | Graphite, Grafana  Image result for grafana |
| Notifications/alerts: existing eDelivery services can be reused with RESTful wrapper | eDelivery project |

## 

## Micro-architecture considerations for microservice development

Developers of business Microservices should be aware of, but largely freed from having to deal with the complexities of a distributed architecture

On the micro (or “inner”) architecture level, it might thus prove useful to agree on a number of common frameworks and patterns for the implementation of business Microservices:

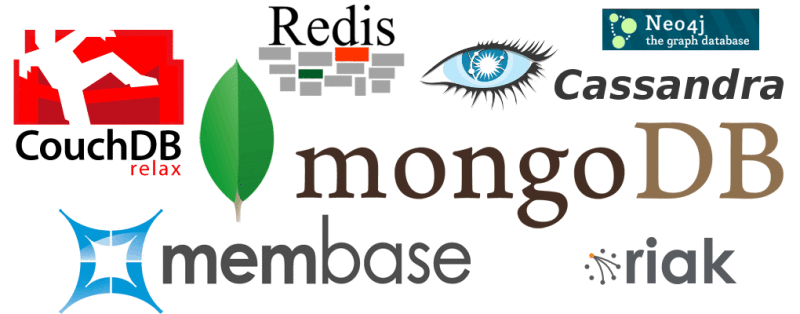
|  |  |
| --- | --- |
| **Consideration** | **Proposed Technology** |
| Service development in accordance with the macro architecture principles and interfacing with the common infrastructure components should be easy and painless | Spring Boot and Spring Cloud for Microservice development in Java  Image result for images:spring boot |
| Stability of the system as a whole must not be compromised if parts of the system fail, so well-proven resilience patterns should be applied whenever a service has to rely on other downstream services to fulfil its contract | Netflix Hystrix  Image result for netflix hystrix |
| Service developers should be encouraged to expose rich business metrics from their business service implementation, allowing for end-to-end monitoring of overall system performance and creation of meaningful monitoring dashboards | Dropwizard Metrics library  Image result for dropwizard |
| A choice of supported persistence mechanisms should be provided to accommodate different service implementations’ persistence requirements | K/V store, RDBMS, GraphDB  Image result for database Screen Clipping |

In addition to the above, the following toolsets have been identified for use in CitiBanamex:

|  |  |  |
| --- | --- | --- |
| **Function** | **Tool** | |
| Agile Lifecycle – tracking & monitoring | Jira | Image result for jira logo |
| Development IDE | Spring Tool Suite (STS) for Eclipse | Image result for Spring Tool Suite (STS) logo |
| Version Control | Bitbucket, Git | Image result for Bitbucket logoGit |
| Swagger Generation | http://swagger.io/swagger-editor/ |  |
| Code Quality Check | Sonar | Image result for sonar code quality check logo |
| DevOps | Jenkins | Image result for Jenkins logo |
| Testing | Stubs, maven profiles + bash scripts + pipeline, MockMVC | http://maven.apache.org/images/maven-logo-black-on-white.png  Image result for bash scripts logo |
| Environments | Dev, SIT, UAT & PROD |  |
| Logs collection, storage, analysis | Splunk, Logstash, ElasticSearch | Image result for splunk logo  Screen Clipping  Image result for ElasticSearch logo |
| Real time monitoring of APIs for both technical and business metrics | Spring actuators |  |
| System monitoring | PCF Monitoring | Image result for pivotal cloud foundry |

Microservices may persist domain specific data into their own private datastores accessible only through APIs. Different strategies are used to determine the optimal datastore needs for a microservice depending on the type of data its handling. NoSQL datastores are a popular choice. A NoSQL database environment is a non-relational and largely distributed database system that enables rapid, ad-hoc organization and analysis of extremely high-volume, disparate data types. Different types of NoSQL datastores and their selection criteria are highlighted here:

## No SQL databases - a key considerations as it drastically enables ability to develop web-native application



What are NoSQL databases?

A storage and retrieval system that is not based on a relational database structure. NoSQL databases enable next generation mobile functionality as it is simple to design, better for horizontal scaling, and generally more flexible.

|  |  |  |
| --- | --- | --- |
| Types | Description | When to use |
| Key-Value Store | Designed for storing data in a schema-less way. Examples of this type of database include Cassandra, DyanmoDB, Azure Table Storage (ATS), Riak, BerkeleyDB | Scenarios include where there is a need for ease of maintenance (no matter how big your data gets), flexible parsing/wide column requirements, no multiple secondary indexes, and massive scale |
| Column Store | Also known as wide-column stores. Instead of storing data in rows, these databases are designed for storing data tables as sections of columns of data, rather than as rows of data.  Offer very high performance and a highly scalable architecture | HBase, BigTable, Druid, HyperTable. E.g. scenarios like real time analytics on big data cluster, scenarios that need to handle data of the order or petabytes, not suitable for transactional storage |
| Document Database | Expands on the basic idea of key-value stores where “documents” are more complex, in that they contain data and each document is assigned a unique key, which is used to retrieve the document. These are designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data. Examples include MongoDB and CouchDB | Ideal scenarios are where there is a need for real-time analytics, high-speed logging, caching, and high scalability. However, it’s not as good if your app requires a highly transactional system |
| Graph Database | Based on [graph theory](http://en.wikipedia.org/wiki/Graph_theory), these databases are designed for data whose relations are well represented as a graph and has elements which are interconnected, with an undetermined number of relations between them. Examples include Neo4J and Polyglot | In scenarios like fraud detection, real time recommendation engines, network & IT operations, identity and access management |

## Evaluating NoSQL Database – there are variety of dimensions to consider in choosing the right NoSQL solution

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data model** | **Performance** | **Scalability** | **Flexibility** | **Complexity** | **Functionality** |
| Key-value store | High | High | High | None | Variable (None) |
| Column Store | High | High | Moderate | Low | Minimal |
| Document Store | High | Variable (High) | High | Low | Variable (Low) |
| Graph Database | Variable | Variable | High | High | Graph Theory |

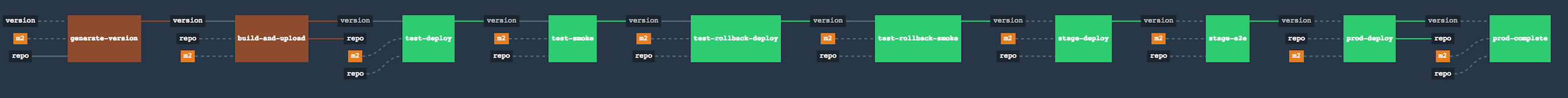
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

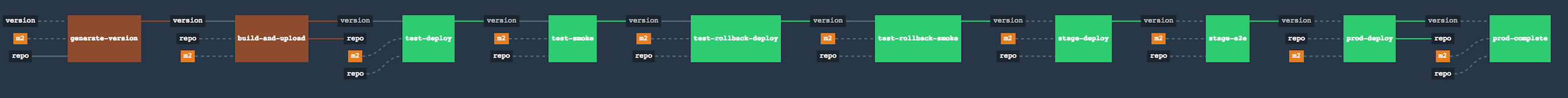
**Developer Notes**: Polyglot persistence is a powerful feature available to microservices designers and developers. Make use of suitable persistence choices based on the above criteria specific to use cases being catered to.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# DevOps Pipeline

Citibanamex DevOps pipeline is shown in the flow diagram below.





Description of pipeline steps:

1. **generate-version**

In this step, we are generating the version of the application/pipeline. This step will checkout the latest version from SCM, version will be release version of the build. In this step, we are also tagging the repository with release tag (naming convention will be followed).

That way in each subsequent step of the pipeline we’re able to retrieve the tagged version. Also, we know exactly which version of the pipeline corresponds to which Git hash.

Tools – Git, Bitbucket, sonarqube

Test Run:

* Requirement validation
* Code Review - Trigger code analysis with Sonarqube

1. **build-and-upload**

The process of turning the code into a bundle of scripts, assets and binaries that run the code is the build. Binaries will be uploaded to Nexus repository.

During this phase, we are executing a Maven build using Maven Wrapper, with unit and integration tests. To do this, just execute mvn clean deploy either from command line or Jenkins job.

Tools – Maven, Jenkins, Nexus

Test Run:

* Continuous Integration using Jenkins
* Maven profiles, validate deploy task in pom.xml
* check settings.xml for maven deploy
* Validate below checks in pom.xml:
  + groupId task to retrieve group id
  + artifactId task to retrieve artifact id
  + version task to retrieve the current version

1. **test-deploy**

In this step, we will first take the backup of the current version deployed on the test env and rename it. Then fetch the latest build and deploy to the test environment.

We are downloading the built artifact from Nexus and we are uploading it to Cloud Foundry, this will deploy the new version on the env. If the application is using a database then it gets upgraded at this step.

Tools – Jenkins, CloudFoundry

Test Run:

* Make sure that the instance is up and running with the new version
* Log collections, analysis, monitoring (splunk)

1. **test-smoke**

Trigger test, run sanity and regression tests on the environment.

Tests that are executed on a deployed application. The concept of these tests is to check the crucial parts of your application are working properly. If you have 100 features in your application but you gain most money from e.g. 5 features then you could write smoke tests for those 5 features.

Running smoke tests on a deployed app via the smoke Maven profile

Tools - Stubs, maven profiles + bash scripts + pipeline, MockMVC

Test Run:

* Bash scripts to run unit test cases
* Log collections, analysis, monitoring (splunk)

1. **test-rollback-deploy**

Rollback the build to the previous stage, rollback is required in case there are release issues.

Test the backwards compatibility of the application in order to roll it back if necessary.

In this step, we will rename the new version since it is not working or broken after the deployment and then restore the previous version on the env.

Test Run:

* Make sure that the instance is up and running with the previous version
* Log collections, analysis, monitoring (splunk)

1. **test-rollback-smoke**

Run sanity and regression tests on the target environment once rollback to previous version.

Test the backwards compatibility of the application in order to roll it back if necessary.

As we did the rollback in previous step, we need to execute test cases to ensure that rollback is successful and env is running with the previous state/version.

Test Run:

* Bash scripts to run unit test cases
* Log collections, analysis, monitoring (splunk)

1. **stage-deploy**

After getting sign off from the test environment, deploy the build to the staging environment. Stage is an environment that does resemble production. Most likely applications are deployed there in versions that correspond to those deployed to production.

Test Run:

* Continuous Deployment using Jenkins
* Log collections, analysis, monitoring (splunk)

1. **prod-deploy**

Deploy the build to the production environment. Promote the same build to production which was deployed in staging and pre-prod environments. Once the application is deployed to production we are tagging it with prod version.

In this step, we are deploying our tested application to the prod env for customers.

We are taking the backup of the old version before proceeding with the new deployment. Jar will be downloaded from the Nexus repository on prod server. Deploy new version and run the application.

Test Run:

* Speed up the time required to deploy a feature to production
* Make sure that the instance is up and running with the new version
* Performance testing
* Log collections, analysis, monitoring (splunk)

1. **prod-complete**

Trigger email notification to all stakeholders.

Publish Release Notes to the customer after release is completed.

# API Design Cookbook

## 12-Factor Apps Design Principles for Microservices on PCF:

|  |  |  |
| --- | --- | --- |
| **Design principle** | **Criticality** | **Description** |
| Codebase | Mandatory | All application code for a service lives in one repository |
| Dependencies | Mandatory | Explicitly declare and isolate dependencies |
| Config | Mandatory | Store config in the environment |
| Backing Services | Mandatory | Treat backing services as attached resources |
| Build, realease, run | Recommended | Strictly separate build and run stages |
| Processes | Mandatory | Execute the app as one or more stateless processes |
| Port binding | Mandatory | Export services via port binding |
| Concurrency | Recommended | Scale out via the process model |
| Disposability | Recommended | Maximize robustness with fast startup and graceful shutdown |
| Dev/prod parity | Recommended | Keep development, staging, and production as similar as possible |
| Logs | Mandatory | Treat logs as event streams |
| Admin processes | Recommended | Run admin/management tasks as one-off processes |

### [Codebase](http://12factor.net/codebase) — one codebase tracked in revision control, many deploys

Criticality: Mandatory

All application code for a service lives in one repository. A codebase is run by developers on their local machines, and deployed to any number of other environments, like a set of testing machines, and the live production servers.

### [Dependencies](http://12factor.net/dependencies) — explicitly declare and isolate dependencies

Criticality: Mandatory

All the environments code runs in need to have some dependencies, like a database, or an image-processing library, or a command-line tool. Applications must not assume those things will be in place on a given machine. Most languages and frameworks provide a natural way to do this. List all the versions of all the libraries and when the code is deployed, a command is run to download all the right versions and put them in place.

### [Config](http://12factor.net/config) — Store config in the environment

Criticality: Mandatory

The twelve-factor compliant app stores config in environment variables. Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

### [Backing Services](http://12factor.net/backing-services) — Treat backing services as attached resources

Criticality: Mandatory

A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as [MySQL](http://dev.mysql.com/) or [CouchDB](http://couchdb.apache.org/)), messaging/queueing systems (such as [RabbitMQ](http://www.rabbitmq.com/) or [Beanstalkd](http://kr.github.com/beanstalkd/)), SMTP services for outbound email (such as [Postfix](http://www.postfix.org/)), and caching systems (such as [Memcached](http://memcached.org/)).

Backing services like the database are traditionally managed by the same systems administrators as the app’s runtime deploy. In addition to these locally-managed services, the app may also have services provided and managed by third parties. Examples include SMTP services (such as [Postmark](http://postmarkapp.com/)), metrics-gathering services (such as [New Relic](http://newrelic.com/) or [Loggly](http://www.loggly.com/)), binary asset services (such as [Amazon S3](http://aws.amazon.com/s3/)), and even API-accessible consumer services (such as [Twitter](http://dev.twitter.com/), [Google Maps](https://developers.google.com/maps/), or [Last.fm](http://www.last.fm/api)).

The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the [config](https://12factor.net/config). A [deploy](https://12factor.net/codebase) of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as [Amazon RDS](http://aws.amazon.com/rds/)) without any changes to the app’s code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change. Each distinct backing service is a *resource*.

This is another case where defining dependencies cleanly keeps the system flexible and each part is abstracted from the complexities of the other, a core tenet of good architecture.

Resources can be attached and detached to deploys at will. For example, if the app’s database is misbehaving due to a hardware issue, the app’s administrator might spin up a new database server restored from a recent backup. The current production database could be detached, and the new database attached – all without any code changes.

### [Build, release, run](http://12factor.net/build-release-run) — strictly separate build and run stages

Criticality: Recommended

The process of turning the code into a bundle of scripts, assets and binaries that run the code is the build. The release sends that code to a server in a fresh package together with the nicely separate config files for that environment. Then the code is run so the application is available on those servers.

The idea here is that the build stage does a lot of heavy lifting, and developers manage it. The run stage should be simple and bullet-proof so that the team can be confident knowing that the application is running well, and that if a machine gets restarted (say, a power failure happens) that the app will start up again on launch without the need for human intervention.

### [Processes](http://12factor.net/processes) — execute the app as one or more stateless processes

Criticality: Mandatory

It is likely that an application will be running on many servers, because that makes it more fault tolerant, and support more traffic. As a rule, each of those instances of running code should be stateless. In other words, the state of the system is completely defined by the databases and shared storage, and not by each individual running application instance.

For example, a signup workflow, where a user has to enter 3 screens of information to create their profile. One (wrong) model would be to store each intermediate state in the running code, and direct the user back to the same server until the signup process is complete. The right approach is to store intermediate data in a database or persistent key-value store such as Gemfire, so even if the web server goes down in the middle of the user’s signup, another web server can handle the traffic, and the system is none-the-wiser.

### [Port binding](http://12factor.net/port-binding) — Export services via port binding

Criticality: Mandatory

The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.

In a local development environment, the developer visits a service URL like http://localhost:5000/ to access the service exported by their app. In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.

This is typically implemented by using [dependency declaration](https://12factor.net/dependencies) to add a webserver library to the app, such as [Tornado](http://www.tornadoweb.org/) for Python, [Thin](http://code.macournoyer.com/thin/) for Ruby, or [Jetty](http://www.eclipse.org/jetty/) for Java and other JVM-based languages, Express for Node.js. This happens entirely in user space, that is, within the app’s code. The contract with the execution environment is binding to a port to serve requests.

### [Concurrency](http://12factor.net/concurrency) — Scale out via the process model

Criticality: Recommended

When running the code, the idea is that many little processes are handling specific needs. Therefore, there might be dozens of handlers ready to process web requests, and another dozen to handle API calls for enterprise users. Still another half-dozen processing background welcome-emails going to new users, or sending tweets for users sharing things on a social media service.

By keeping all these small parts working independently, and running them as separate processes (in a low-level technical sense), the application will scale better. More stuff is done concurrently, by smoothly adding additional servers or additional CPU/RAM and taking full advantage of it with more of these small, independent processes.

### [Disposability](http://12factor.net/disposability) — Maximize robustness with fast startup and graceful shutdown

Criticality: Recommended

When new code is deployed, ideally the new version should launch right away and start to handle traffic. If an application has to do 20 seconds of work (say, loading giant mapping files into RAM) before it is ready to handle real traffic, makes it harder to rapidly release code, and more churn is introduced on the system to stop/start independent processes.

With the proliferation of so many third party libraries in today’s software systems, sub–1-second startup times are less and less common. However, beyond loading code, an application should have everything it needs waiting in high-speed databases or caches, so it can start up snappily and be ready to serve requests.

Further, the application should be robust against crashing. Meaning, if it does crash, it should always be able to start back up cleanly. One should never do any mandatory “cleanup” tasks when the app shuts down that might cause problems if they failed to run in a crash scenario.

### [Dev/prod parity](http://12factor.net/dev-prod-parity) — Keep development, staging, and production as similar as possible

Criticality: Recommended

It has become in vogue in recent years to have a much more rapid cycle between developing a change to an app and deploying that change into production. For many companies, this happens in a matter of hours. In order to facilitate that shorter cycle, and the risk that something breaks when entering production, it is desirable to keep a developer’s local environment as similar as possible to production.

This means using the same backing services, the same configuration management techniques, the same versions of software libraries, and so on.

This is often accomplished by letting developers use a tool like [Vagrant](http://www.vagrantup.com/) to manage their own personal virtual server that is configured just like production servers.

### [Logs](http://12factor.net/logs) — Treat logs as event streams

Criticality: Mandatory

A twelve-factor app should not concern itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behavior.

In staging or production deploys, each process’ stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival. These archival destinations are not visible to or configurable by the app, and instead are completely managed by the execution environment. Open-source log routers (such as [Logplex](https://github.com/heroku/logplex) and [Fluent](https://github.com/fluent/fluentd)) are available for this purpose.

The event stream for an app can be routed to a file, or watched via realtime tail in a terminal. Most significantly, the stream can be sent to a log indexing and analysis system such as [Splunk](http://www.splunk.com/), or a general-purpose data warehousing system such as [Hadoop/Hive](http://hive.apache.org/). These systems allow for great power and flexibility for introspecting an app’s behavior over time, including:

* Finding specific events in the past.
* Large-scale graphing of trends (such as requests per minute).
* Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

### [Admin processes](http://12factor.net/admin-processes) — Run admin/management tasks as one-off processes

Criticality: Recommended

There may be need of lots of one-off administrative tasks once an app is live. For example, doing data cleanup on bad data discovered, running analytics, or turning on and off features for A/B testing.

Usually a developer will run these tasks, and when they do, they should be doing it from a machine in the production environment that is running the latest version of the production code. In other words, run one-off admin tasks from an identical environment as production. Do not run updates directly against a database; do not run them from a local terminal window.

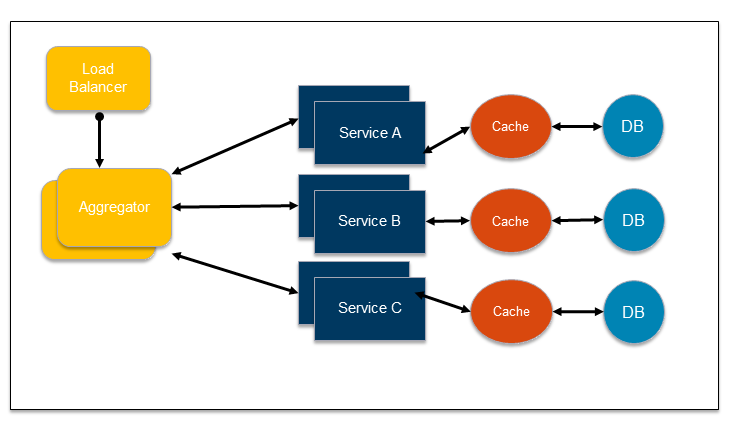
## Microservices Design Patterns — Functional decomposition

The main characteristics of a microservices-based application are functional decomposition or domain-driven design, well-defined interfaces, explicitly published interface, single responsibility principle, and potentially polyglot. Each service is fully autonomous.Thus changing a service implementation has no impact to other services as they communicate using well-defined interfaces. Functional decomposition of an application is the key to building a successful microservices architecture. This allows loose coupling (REST interfaces) and high cohesion (multiple services can compose with each other to define higher level services or application).

Verb (e.g. Checkout) or Nouns (Product) of the application are one of the effective ways to achieve decomposition of an existing application. For example, product, catalog, and checkout can be three separate microservices and then work with each other to provide a complete shopping cart experience.

|  |  |
| --- | --- |
| **Design pattern** | **Advantages** |
| Aggregator | Each individual service can evolve independently |
| Proxy | No aggregation happens in the client side |
| Chained microservice | Produce a single consolidated response to a request |
| Branch microservice | Simultaneous response processing from two, likely mutually exclusive, chains of microservices |
| Shared data design | Full-stack and has control of all the components |
| Asynchronous Messaging | Connects synchronous and asynchronous services |

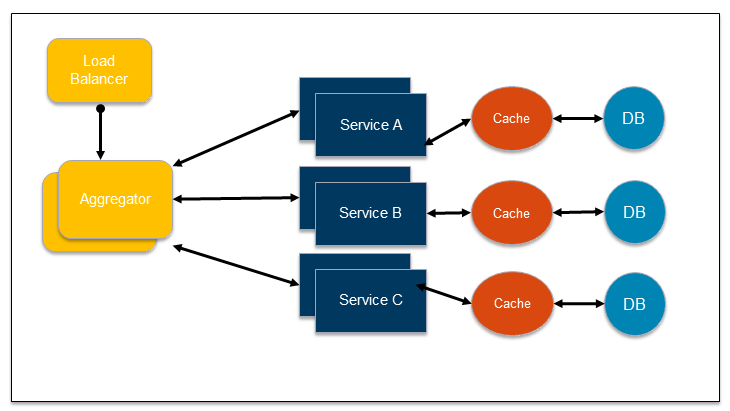
### Aggregator Pattern:



This design pattern follows the DRY principle. If there are multiple services that need to access Service A, B, and C, then its recommended to abstract that logic into a composite microservice and aggregate that logic into one service. An advantage of abstracting at this level is that the individual services, i.e. Service A, B, and C, and can evolve independently and the business need is still provided by the composite microservice.

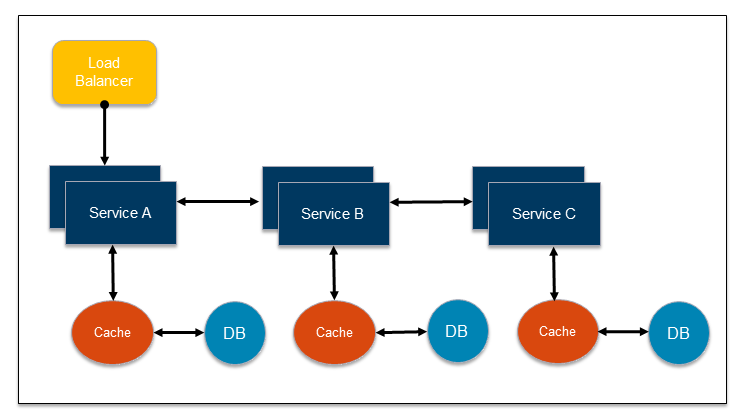
Note that each individual microservice has its own (optional) caching and database. If Aggregator is a composite microservice, then it may have its own caching and database layer as well.

### Proxy Pattern



Proxy microservice design pattern is a variation of Aggregator. In this case, no aggregation needs to happen on the client but a different microservice may be invoked based upon the business need. The proxy may be a passthrough proxy in which case it just delegates the request to one of the services. Alternatively, it may be a smart proxy where some data transformation is applied before the response is served to the client. A good example of this would be where the presentation layer to different devices can be encapsulated in the smart proxy.

### Chained microservice design pattern

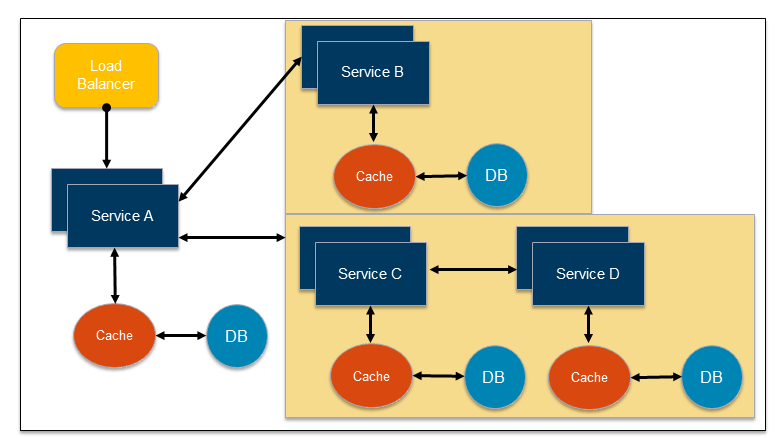


Chained microservice design pattern produce a single consolidated response to the request. In this case, the request from the client is received by Service A, which is then communicating with Service B, which in turn may be communicating with Service C. All the services are likely using a synchronous HTTP request/response messaging. The key part to remember is that the client is blocked until the complete chain of request/response, i.e. Service <-> Service B and Service B <-> Service C, is completed. The request from Service B to Service C may look completely different as the request from Service A to Service B. Similarly, response from Service B to Service A may look completely different from Service C to Service B. Different services are adding their business value.

An important aspect to understand here is to not make the chain too long. This is important because the synchronous nature of the chain will appear like a long wait at the client side, especially if its a web page that is waiting for the response to be shown. There are workarounds to this blocking request/response and are discussed in a subsequent design pattern.

A chain with a single microservice is called singleton chain. This may allow the chain to be expanded at a later point.

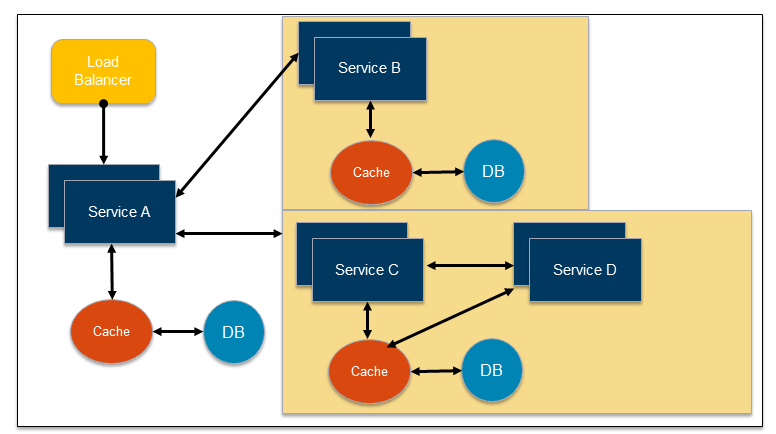
### Branch microservice design pattern



Branch microservice design pattern extends Aggregator design pattern and allows simultaneous response processing from two, likely mutually exclusive, chains of microservices. This pattern can also be used to call different chains, or a single chain, based upon the business needs. Service A, either a web page or a composite microservice, can invoke two different chains concurrently in which case this will resemble the Aggregator design pattern. Alternatively, Service A can invoke only one chain based upon the request received from the client.

This may be configured using routing of JAX-RS or Camel endpoints, and would need to be dynamically configurable.

### Shared Data Design Pattern



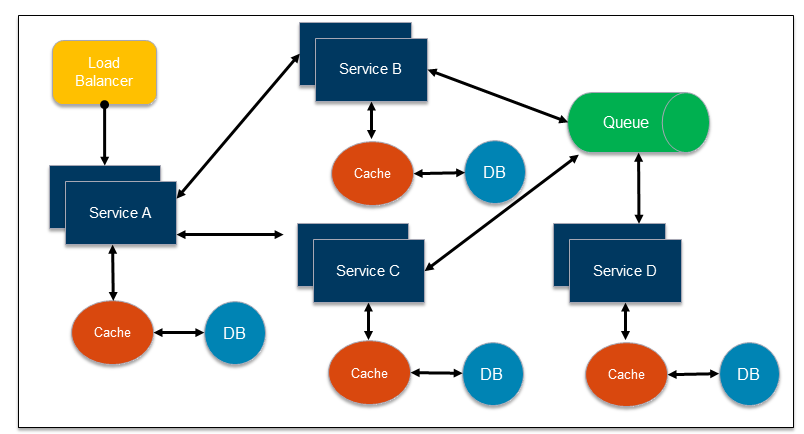
One of the design principles of microservice is autonomy. That means the service is full-stack and has control of all the components – UI, middleware, persistence, transaction. This allows the service to be polyglot, and use the right tool for the right job. For example, if a NoSQL data store can be used, it may be more appropriate than developing a complex SQL database.

However a typical problem, especially when refactoring from an existing monolithic application, is database normalization such that each microservice has the right amount of data – nothing less and nothing more. Even if only a SQL database is used in the monolithic application, denormalizing the database would lead to duplication of data, and possibly inconsistency. In a transition phase, some applications may benefit from a shared data microservice design pattern.

In this design pattern, some microservices, likely in a chain, may share caching and database stores. This would only make sense if there is a strong coupling between the two services. Some might consider this an anti-pattern but business needs might require in some cases to follow this. This would certainly be an anti-pattern for greenfield applications that are designed on microservices from the beginning.

### Asynchronous Messaging Design Pattern

While REST design pattern is quite prevalent, and well understood, but it has the limitation of being synchronous, and thus blocking. Asynchronous behaviour can be achieved for non-blocking responsive use case needs. Some microservice architectures may elect to use message queues instead of REST request/response for this reason.



In this design pattern, Service A may call Service C synchronously which is then communicating with Service B and D asynchronously using a shared message queue. Service A -> Service C communication may be asynchronous, possibly using WebSockets, to achieve the desired scalability.

A combination of REST request/response and pub/sub messaging may be used to accomplish the business need.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Developer Notes:** Make full use of asynchronous design pattern wherever possible for maximum responsiveness, reliability, robustness and enabling improved user experience on channel or client app side.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## Agile Sprints - Definition of Done (DoD)

Teams typically start with a simple definition of done and slowly evolves as they have a stable sustained velocity. For a sprint deliverable to be considered complete, the following acceptance criteria is listed as a checklist to help developers:

|  |  |  |
| --- | --- | --- |
|  | **Definition of Done** | **Description** |
| i | Unit test cases written | Unit test cases should be conceived and written with complete coverage from negative testing and non functional point of view |
| ii | Performance tests automated & passing | Microservices developed should be tested for performance to identify any coding/design issues and bottlenecks. If needed a different design pattern may be switched to for performance optimization |
| iii | Deployed in Integration environment | APIs and Microservices tested in dev environment using stubs and simulators should be deployed in Integrated test environments with appropriate configs |
| iv | Functional tests automated & passing | Microservces should withstand and pass automated functional tests and regression test for each build and deploy |
| v | Accepted by PO | Test reports generated through pipeline tools to be reviewed and accepted by PO |
| vi | Code reviewed and refactored | Third party review of code (manual and using tooling) would be a good practice and any necessary refactoring to be done |
| vii | Meets all acceptance criteria | Component to be checked for conformance against all defined acceptance criteria both functional and non-functional requirements (NFR) |
| viii | Checked in | Code must be checked in and available in repos in curated and purified form at all stages. In some cases developers can create private branches for any research or experiments. Merging of code should be done carefully to exclude experimental additional and not breaking existing functionality |
| ix | Unit tests automated and passing | Final components much pass all unit and automated tests for promotion to subsequent environments |

The definition of done ensure that the working software produced at the end of the sprint is of high quality and potentially shippable.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Developer Notes:**

* Refer to the 12-factor app principles often and see conformance of critical factors.
* Follow the DOD checklist for components to verify their completeness
* Follow TDD principles to catch bugs early in development and speed up delivery through reduced bugs count in automated testing in Devops pipeline

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# API Security Considerations

Security is a concern addressed at both API gateway level and by underlying Microservices. An API gateway establishes a single entry point for all requests coming from all clients. It subsequently knows how to route the request to the appropriate microservices through Banamex’s PSG security component. By using this technique, Microservices stay secured behind a firewall, allowing the API gateway to handle external requests and then talk to the microservices behind the firewall. The security techniques that each layer uses are described in the following table:

|  |  |
| --- | --- |
| **Layer** | **Security techniques** |
| API Gateway | * E2E encryption of sensitive data elements * Client credentials (client-id and client-secret ) used to validate and generate Authorization tokens |
| PSG | * OAuth Implementation – 2-legged for Citi Apps |
| API | * OWASP security controls * Sequence protection * Role based access control |

The following table describes the actual communication mechanisms between layers that will ensure the microservices are not accessible without authorization.

|  |  |  |
| --- | --- | --- |
| **From** | **To** | **Authentication/Authorization Implementation** |
| Services Client | API Gateway | Client credentials validation (ClientID as request header) |
| API Gateway | PSG | Server-server authentication via 2-way SSL |
| PSG | API | Server-server authentication via 2-way SSL |
| API | API via API Gateway | Access Token issued during startup using valid client credentials |
| API, PSG | GemFire Cache | Server-server authentication via 2-way SSL  UserID/password based authentication with role based authorization |

Additional business rules:

* The microservice components are divided in three sub-layers: process, core and system connectors. The three sub-layers are all located within the API layer, behind the API Gateway and PSG.
* Different channels will have different security requirements. Example: A Bancanet mobile app session has a TTL of 15 minutes and requires user credentials and token validation; while an ATM operation doesn’t require a TTL and doesn’t require any user credentials. The channels should each have a dedicated micro service with its own security rules, and are generally process APIs.

Customers can’t have more than one session open in the API with the same credentials. Example: A user can’t log from the Bancanet mobile app and the web portal at the same time; they must close one session before opening the other.

## Microservices Security - tools

Microservices on CloudFoundry can be secured using OAuth2 Bearer tokens or JSON Web Tokens built on top of OAuth2 protocol. In OAuth2, a client application, often web application, acts on behalf of a user, but with the user's approval as Authorization Server. Common examples of Authorization Servers and Resource Servers on the internet: Facebook, Graph API Google, Google APIs

Cloud Foundry provides OAuth2 authorization services as well through Cloud Controller. OAuth2 Bearer tokens centralizes account management and permissions. It's a lightweight protocol and easy to experiment with simple tools such as curl or wget.

A sample curl request to a OAuth2 secured microservice would look like the following:

$ curl -H "Authorization: Bearer $TOKEN" https://myhost /resource,

Where https://myhost is the resource server and $TOKEN is the Bearer token returned from the Authorization server.

CloudFoundry UAA also supports JSON Web Tokens (JWT) signed and encoded. JWT is a token format, which defines a compact and self-contained mechanism for transmitting data between parties in a way that can be verified and trusted because it is digitally signed. Additionally, the encoding rules of a JWT also make these tokens very easy to use within the context of HTTP.

Being self-contained, (the actual token contains information about a given subject) they are also a good choice for implementing stateless authentication mechanisms. When going this route and the only thing a party must present to be granted access to a protected resource is the token itself, the token in question can be called a bearer token.

It is recommended passing access token within the Authorization header instead of going with custom headers following the rules of RFC 6750. Authorization headers are recognized and specially treated by HTTP proxies and servers. Thus, the usage of such headers for sending access tokens to resource servers reduces the likelihood of leakage or unintended storage of authenticated requests in general, and especially Authorization headers.

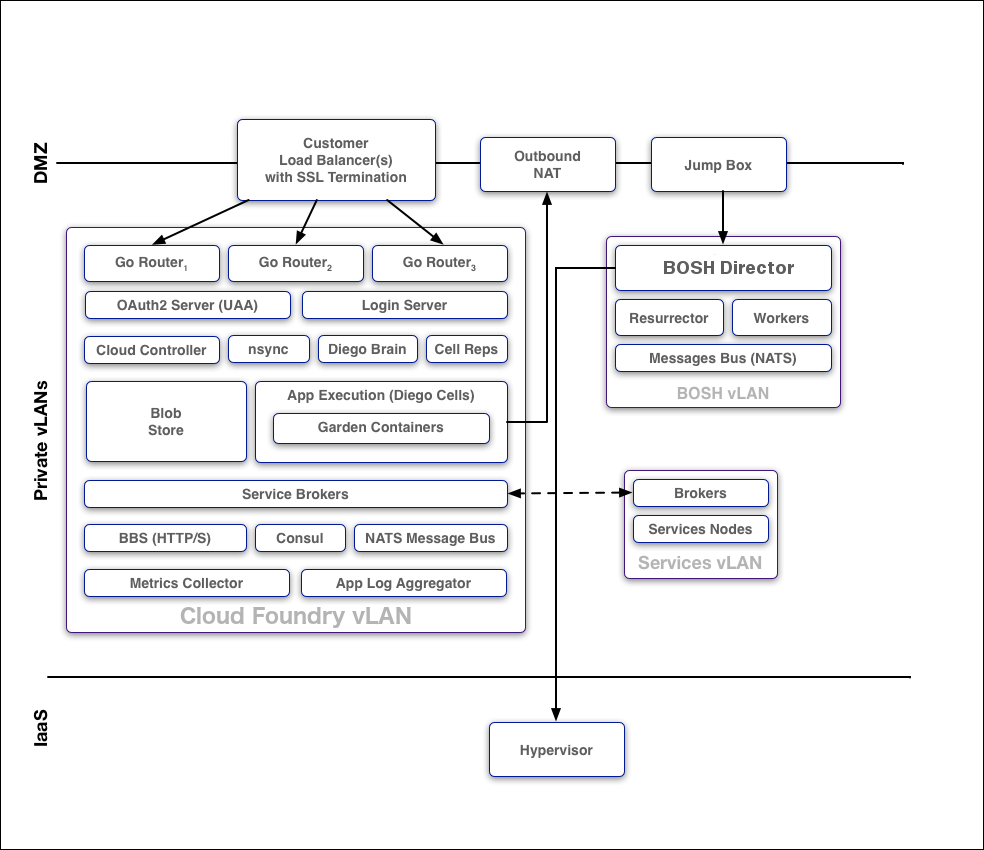
Cloud Foundry implements the following measures to mitigate against security threats:

* Minimizes network surface area
* Isolates customer applications and data in containers
* Encrypts connections
* Uses role-based access controls, applying and enforcing roles and permissions to ensure that users can only view and affect the spaces for which they have been granted access
* Ensures security of application bits in a multi-tenant environment
* Prevents possible denial of service attacks through resource starvation

CitiBanamex has an internal security product aka PSG that works as a proxy for PCF microservices. Alternately PCF interceptors can be used as a proxy.

System Boundaries and Access

As the image below shows, in a typical deployment of Cloud Foundry, the components run on virtual machines (VMs) that exist within a VLAN. In this configuration, the only access points visible on a public network are a load balancer that maps to one or more Cloud Foundry routers and, optionally, a NAT VM and a jumpbox. Because of the limited number of contact points with the public internet, the surface area for possible security vulnerabilities is minimized. Cloud Foundry recommends that NAT VM is installed for outbound requests and a Jumpbox to access the BOSH Director, though these access points are optional depending on network configuration. The diagram below assumes PCF 1.8.



Protocols

All traffic from the public internet to the Cloud Controller and UAA happens over HTTPS. Inside the boundary of the system, components communicate over a publish-subscribe (pub-sub) message bus [NATS](http://nats.io/), HTTP, and SSL/TLS.

BOSH

Operators deploy Cloud Foundry with BOSH. The BOSH Director is the core orchestrating component in BOSH: it controls VM creation and deployment, as well as other software and service lifecycle events. HTTPS is used to ensure secure communication to the BOSH Director.

BOSH includes the following functionality for security:

* Communicates with the VMs it launches over NATS. Because NATS cannot be accessed from outside Cloud Foundry, this ensures that published messages can only originate from a component within the deployment.
* Provides an audit trail through the bosh tasks command. This command shows all actions that an operator has taken with BOSH.
* Allows set up of individual login accounts for each operator. BOSH operators have root access.

Identity Management, Authentication and Authorization

[User Account and Authentication](https://docs.cloudfoundry.org/concepts/architecture/uaa.html) (UAA) is the central identity management service for Cloud Foundry and its various components.

UAA acts as an [OAuth2](https://oauth.io/) Authorization Server and issues access tokens for applications that request platform resources. The tokens are based on the [JSON Web Token](http://jwt.io/) (JWT) and are digitally signed by UAA.

Operators can configure the identity store in UAA. If users register an account with the Cloud Foundry platform, UAA acts as the user store and stores user passwords in the UAA database using [bcrypt](http://en.wikipedia.org/wiki/Bcrypt). UAA also supports connecting to external user stores through LDAP and SAML. Once an operator has configured the external user store, such as a corporate Microsoft Active Directory, users can use their LDAP credentials to gain access to the Cloud Foundry platform instead of registering a separate account. Alternatively, operators can use SAML to connect to an external user store and enable single sign-on for users into the Cloud Foundry platform.

Managing User Access with Role-Based Access Control

Applications that users deploy to Cloud Foundry exist within a space. Spaces exist within orgs. To view and access an org or a space, a user must be a member of it. Cloud Foundry uses role-based access control (RBAC), with each role granted permissions to either an org or a specified space.

Security for Service Broker Integration

The Cloud Controller authenticates every request with the Service Broker API using HTTP or HTTPS, depending on which protocol that is specified during broker registration. The Cloud Controller rejects any broker registration that does not contain a username and password.

Service instances bound to an app contain credential data. Users specify the binding credentials for [user-provided service instances](https://docs.cloudfoundry.org/devguide/services/user-provided.html), while third-party brokers specify the binding credentials for managed service instances. The VCAP\_SERVICES environment variable contains credential information for any service bound to an app. Cloud Foundry constructs this value from encrypted data that it stores in the Cloud Controller Database (CCDB).

A third-party broker might offer a dashboard client in its catalog. Dashboard clients require a text string defined as a client\_secret. Cloud Foundry does not store this secret in the CCDB. Instead, Cloud Foundry passes the secret to the UAA component for verification using HTTP or HTTPS.

## API Gateway Security

The Gateway security features and scenarios for selecting them are discussed below.

IBM API manager can provide security services for underlying microservices, by authenticating callers and controlling access privileges to APIs, resources and data. Various design principles can be applied depending on use case scenarios. Some security features and considerations are highlighted here:

Authenticating by using enterprise user registry

IBM API Management supports a variety of user registry types for authenticating users and securing APIs. Enterprise user registry can be used for authentication in IBM API Management if it is of one of the following types:

LDAP directory

If the user registry uses Lightweight Directory Access Protocol (LDAP), it can be used in IBM API Management for both user authentication and API security.

Authentication URL

A non-LDAP user registry can be configured by using an authentication URL. An authentication URL enables integration with third-party authentication providers, for example, and can be used in IBM API Management for both user authentication and API security.

SCIM

IBM API Management can authenticate with a user registry by using the System for Cross-domain Identity Management (SCIM) standard. Consider using SCIM for a custom user registry; implement a SCIM "bridge" to enable IBM API Management to connect to the registry. SCIM can be used for user authentication but not for API security.

Local User Registry

Authenticate users with a local user registry. A local user registry is an internal registry stored within API Management.

If IBM API Management on Cloud (the SaaS offering) is used, any LDAP registry that is used must be visible on the internet; it must not be accessible only from within the corporate intranet.

IBM API Management Local User Registry- Password lockout criteria

A user can be locked out of his/her account if attempts to log in and fail consecutively. Account lock out applies only for local user registries. The length of time of lockout increases as the number of consecutive failed attempts increases.

For example, a user can be locked out for 15 seconds if he/she has five consecutive failed attempts, or 32 minutes for 12 consecutive failed attempts.

External user registries, such as LDAP, might enforce their own lockout criteria.

SSL Profiles

In IBM API Manager, SSL profiles are used to secure transmission of data through websites. SSL certificates guarantee that information one submits will not be stolen or tampered with.

API Management supports the use of SSL certificates but does not itself produce strong encryption keys or manage enterprise encryption keys. Encryption keys should be created and managed according to enterprise set procedures. When publishing an API, SSL certificate is transferred with an API to the Gateway. During the transferal, certificates have SSL encryption and password protection using the DataPower administrator account's credentials. After being transferred to the Gateway, the keys and certificates are stored in DataPower's cert: folder, which is an encrypted file system. Files in the cert: folder can be extracted only when running a DataPower secure backup, during which all information from the DataPower appliance is double encrypted using asymmetric keys and can be restored only to another DataPower appliance.

If IBM API Management on Cloud (the SaaS offering) is used, one can use SSL profiles configured in the API Manager user interface to protect access to back-end services. However, one cannot use them to protect access on the front-end, such as connecting to the API Management user interfaces or invoking API calls, because the IBM Operations team on behalf of all customers controls the front-end capability.

Setting application identification requirements

Applications that call the operations can be required to provide their client ID, or their client ID and a secret.

**Client ID:** When configuring security in IBM API Management V4.0.2 and later choose ‘Client ID” option if it’s needed for the application developer to include only the client ID in the API calls. This ID identifies applications that are making API calls so that application-specific quotas and statistics can be monitored, displayed, and enforced. This option is selected by default.

**Client ID and Client Secret:** This option is chosen when the developer is to include both the Client ID and the Client Secret in the API calls. This option is similar to providing a user name and password.

**None:** This option is chose if no Client ID or Client ID and secret is required. If this option is selected, the following restrictions apply:

* Cannot apply Rate Limits to a Plan.
* There is no way to identify the caller of the API so this means that no analytics can be tracked against the application, developer organization, or Plan. However, analytics can still be tracked against the API or operation.

Basic authentication:

This common pattern is used to authenticate users through an LDAP user directory or an authentication URL. Basic authentication requires API users to provide a valid user name and password to access selected operations. The application developer must also provide an HTTP authorization header in requests that are sent to operations that require basic authentication.

If an authentication URL is used, the user credentials that are provided in the Authorization header are validated by the endpoint specified in the URL. If the user is authenticated, IBM API Management expects an authentication URL to return an HTTP 200 OK response status code. All other HTTP response status codes result in an authentication failure and access is denied.

OAuth authentication:

OAuth is a token-based authorization protocol that allows third-party websites or applications to access user data without requiring the user to share personal information. Each token grants access to a specific site for specific resources for a defined duration. By using an OAuth token, a user can grant a third-party site access to their information, which is stored with another service provider, without sharing their personal credentials. If the user changes their mind and decides that they do not want a third-party site to continue to have access to their information, the user can revoke the token access. If the token revocation URL is specified, the token revocation list is always checked before access is granted to the user information. Currently, OAuth 2.0 authorization is supported including the ability to select between Public or Confidential client types and the following grant types:

|  |  |  |
| --- | --- | --- |
| **Grant Type** | **Public Client** | **Confidential Client** |
| Authorization Code | Yes | Yes |
| Implicit | Yes | No |
| Resource Owner Password Credentials | Yes | Yes |
| Client Credentials | No | Yes |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Developer Notes:**

* Use of API Gateway pattern and authenticate APIs using client id and secret. This enables identification of the caller for metering, throttling, quota etc.
* Secure microservices using auth tokens passed in HTTPS headers e.g. JWT tokens
* Ensure sensitive data is encrypted/masked e.g credit card numbers
* Keep auth params in PCF env VCAP variables e.g. userid/password for DB login, avoid hardcoding

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# High Availability on Cloud Foundry

High availability and scalability in a microservices architecture is accomplished by combining the capabilities of the platform and following best practices for high availability application design.

## High Availability and Scalability in the platform

High availability is accomplished in Cloud Foundry first by scaling components VMs and locating them in multiple Availability Zones (AZ) so that their redundancy and distribution minimizes downtime during ongoing operation, product updates, and platform upgrades.

Scaling component VMs means changing the number of VM instances dedicated to running a functional component of the system. Scaling usually means increasing this number, while scaling down or scaling back means decreasing it.

### Availability Zones

Deploying Cloud Foundry across three or more AZs and assigning multiple component instances to different AZ locations lets a deployment operate uninterrupted when entire AZs become unavailable.

Banamex has 6 Availability Zones for its Cloud Foundry platform. Cloud Foundry maintains its availability as long as a majority of the AZs remain accessible. For example, a three-AZ deployment stays up when one entire AZ goes down, and a five-AZ deployment can withstand an outage of up to two AZs with no impact on uptime. When using more than 3 AZs, it’s recommended that the number of AZs is odd.

### Application instances

CF supports deploying applications instances across multiple AZs. CF balances the applications deployed across the AZs. If an AZ goes down, other application instances will still be running in the remaining AZs.

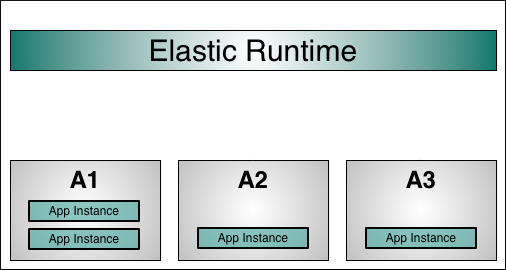
If application instances for any reason (application fault or AZ fault), CF restarts new instances to maintain capacity.

CF monitors the processes in the VMs, if it detects a failure, it restarts the process and generates notifications to the administrators. The VMs send a heartbeat every 60 seconds to the monitor. If the monitor stops receiving the heartbeat from the VM, a request is generated to have a new VM instance created to replace the one that failed. This requests should be processed by the Iaas provider.

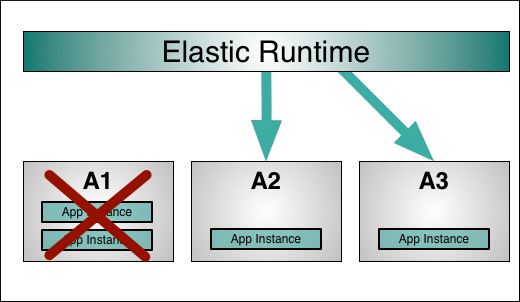
### Example Scenario

Test

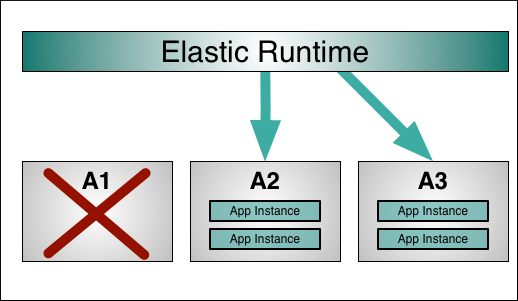
In this example, an operator scales an application to four instances in an Elastic Runtime environment distributed across three availability zones: A1, A2, and A3.



If A1 experiences a power outage or hardware failure, the two application instances running in A1 terminate while the application instances in zones A2 and A3 continue to run:



If A1 remains unavailable, Elastic Runtime balances new instances of the application across the remaining availability zones:



## Best practices for high availability and scalability

|  |  |  |
| --- | --- | --- |
| **Best practice** | **Rationale** | **Implementation** |
| Use client-side service discovery | Any server in the cloud may become unavailable at any time. In that case, clients should still be able to locate services in the cloud.  The number of service instances and their locations change dynamically. The CF VMs have dynamically assigned IP addresses. | Implement Netflix Eureka registry servers for failover and have the services register on it.  **Developer notes:**  This solution does not allow for sticky sessions; session data should be stored in the GemFire distributed cache. |
| Fail fast before a service gets overloaded  Use short timeouts  Auto retry failed requests with random retry intervals  Use circuit breakers when a dependency has failed | Failure of an internal service should not propagate to the external API and reach the clients and users.  The clients of a service shouldn’t be kept waiting for a timeout to occur.  When a service fails fast, it makes sense to automatically retry a request before breaking the circuit. | Decorate all service calls with Netflix Hystrix to provide fallback mechanisms in case a dependency fails.  **Developer notes:**  The fallback mechanism can be implemented using one of three approaches:   * Custom fallback – using stubbed data, cached data, or queue a write request to generate a fallback response. * Fail silent – The fallback method simply returns null or empty values. * Fail fast – When the response data is required or there’s no good fallback, the client gets a 5xx response.   Ideally, all service dependencies would have custom fallbacks. The “fail silent” and “fail fast” approaches are reasonable alternatives when it’s not possible to provide fallback coverage for all services. |
| Maintain services loosely-coupled  Independently develop, test and deploy services  Avoid sharing dependencies and resources between services  Have services developed and owned by independent teams | A microservice is an isolated component that works on a single concern.  Groups of micro services should not need to be deployed together; changing the codebase for one service should not force others to be rebuilt; services should not depend on the same resource (like a database); services should not block each other. | It is preferred that microservices do not communicate between them at all. If communication must occur, it should not happen synchronously, but using RabbitMQ, Kafka or Akka.  **Developer notes:**  If changes to a shared library require all services be updated simultaneously, then there’s a point of tight coupling across services. Do not share common dependencies by configuring them on the host. Upgrading the library or SDK might end up breaking a service. Each service should be treated entirely independent of others.  In some scenarios, the commonly used libraries and SDKs can be moved to a dedicated service that can be managed independently, making the service immutable. |
| Clearly separate stateless and stateful microservices.  Push the data from stateful microservices into external services: GemFire or a NoSQL database | Microservices should be easy to scale and relocate. | Microservices scale better when they are stateless. Within seconds, multiple containers can be launched across multiple hosts. Instances of a stateless service don't store any information related to previous requests and don’t acknowledge the presence of other services. An incoming request can be sent to any instance of the service. |
| Design service calls to be idempotent | Transient failures during service calls can be overcome by retrying the same operation after a delay, the result of the operation should not be affected by the number or retries. | The challenge with automatically retrying service calls is that it is not known if the failing job completed its work before it failed or not. To keep things operationally simple, the services must be idempotent. The end result should be the same whether the job ran once or more than once.  **Developer notes:**  Service calls should not be expressed as “deltas”, for example, “increment X by 20”; it is preferred to send the actual value, for example, “set X to 40”. In cases where this is not an option, it’s possible to track unique identifiers in the client requests and identify those that have been processed successfully in order to send a cached response, or ignore it. |
| Use Splunk for centralized logging  Use correlation IDs for business transactions  Use IDs for services | When a service fails, it should be possible to quickly identify the root cause of the problem. A centralized logging system with adequate labelling facilitates debugging and tracing. | In a system of microservices, a single business transaction can span multiple microservices. Frequently, more than one instance of a service will be running to provide high availability. The services may be implemented in different languages and technologies. Splunk can gather logs from all those services, indexing them for easy searching.  **Developer notes:**   * Logs should use clear key-value pairs: key1=value1, key2=value2, key3=value3 . . . * Log messages should be human-readable and developer friendly, for example, log data in JSON format. * Log messages should contain unique identifiers for the business transaction and the component performing the logging. * Categorize the log messages according to severity: LOG, WARN, INFO, DEBUG and ERROR.   Log to local text files and use Splunk forwarders. |

# Platform-As-a-Service key concepts

* Cloud Foundry is an open source [cloud computing](https://en.wikipedia.org/wiki/Cloud_computing) [platform as a service](https://en.wikipedia.org/wiki/Platform_as_a_service) (PaaS) originally developed by VMware and now owned by [Pivotal Software](https://en.wikipedia.org/wiki/Pivotal_Software) - a joint venture by [EMC](https://en.wikipedia.org/wiki/EMC_Corporation), [VMware](https://en.wikipedia.org/wiki/VMware) and [General Electric](https://en.wikipedia.org/wiki/General_Electric). Cloud Foundry was designed and developed by a small team from [Google](https://en.wikipedia.org/wiki/Google)
* When an application is deployed to Cloud Foundry, an image is created for it and stored internally. The image is then deployed to a Garden container (in the new Dieogo architecture) to run in. For multiple instances, multiple images are started on multiple containers.
* Supported runtimes and frameworks include Java (Spring Framework), Ruby (Rails and Sinatra), Node.js, Scala, Python, PHP, Go .

Project source and corresponding buildfiles can be uploaded to CloudFoundry platform using CLI commands. For Java based Microservices using Maven build, a .pom file is packaged along with source file and pushed to CloudFoundry using CLI tools. When CloudFoundry sees a .pom file, it invokes the correct buildpack for the source to build it, deploy onto containers, and run it.

**Developer Notes:**

* Get acquainted with PCF platform and CF CLI. Some of the most common commands and usages are described in the next section
* Build scripts to perform deployment from Dev IDEs or command prompt to test deployment of services on PCF
* Get familiar with accessing logs in PCF environment, debugging system and runtime issues. Refer to the CLI commands listed in this section as most frequently used ones
* Check deployment parameters in manifest like memory allocation for a service, minimum and maximum number of instances to be spawned and optimize.

## Provisioning of API/Microservices on PCF (cloud/on prem)

Microsrvices developed using Java/Spring boot should follow the 12-factor app principles described above for being ready to be deployed in CloudFoundry environment. Although most of the CF deployment automation is taken care by Devops tools and scripts developers should familiarize themselves with basic principles of CF and CLI commands to be able to login, view logs, debug issues and push code if needed.

Developer Notes:

The cf utility provides many options, but for deployment cf push is all that is required. It accepts arguments to specify the name of the application, where to load it from and the URL that should be used to access it. For example:

cf push spring-music -i 2 -m 512M -n spring-music-v1 -p build/libs/spring-music.war

pushes the Java web application spring-music. Two instances are deployed (this is a Java web-application so by default this is two [Tomcat](https://en.wikipedia.org/wiki/Apache_Tomcat) instances), each with 512M of memory. The URL starts with spring-music-v1 and the web-archive (application) can be found at build/libs/spring-music.war. Every Cloud Foundry instance manages one or more domains. For example, all Pivotal Web Services (PWS) applications are accessed via the cfapps.io domain, so if this Spring Music application had been deployed to PWS, its URL would have been spring-music-v1.cfapps.io.[[dubious](https://en.wikipedia.org/wiki/Wikipedia:Accuracy_dispute#Disputed_statement) – [discuss](https://en.wikipedia.org/wiki/Talk:Cloud_Foundry#How_is_this_a_URL.3F)]

When an application is deployed to Cloud Foundry, an image is created for it and stored internally. The image is then deployed to a Garden container to run in. For multiple instances, multiple images are started on multiple containers. This is where [BOSH](https://en.wikipedia.org/wiki/BOSH_(bosh_outer_shell)) comes in - Cloud Foundry's internal Controller uses BOSH to get the underlying infrastructure to spin up virtual machines to run the Garden containers on. When an application is deleted, all of its containers are destroyed and their resources are freed for other applications to use. If the application instance crashes, its container is killed and a new Garden container is started automatically. A container only ever runs one application ensuring isolation, security and resilience.

A load-balancing router sits at the front of Cloud Foundry to route incoming requests to the correct application - essentially to one of the containers where the application is running.

Following is a link that can help in getting familiarized with CF CLI commands.

<https://docs.cloudfoundry.org/cf-cli/getting-started.html>

 For deploying Microservices on CloudFoundry platform, a POM file is used for triggering the java buildpack on CF. Developers typically go to the source folder for a microservice where all the Java code resides along with the manifest and push the entire folder to CF using CLI. This trigger build on CF platform and deployment of the code as a microservice. There are some checks to be done like no hardcoded Port numbers and they should be set using VCAP env variables, no local file system logging but logs based on event stream etc.

The most commonly used CF CLI commands are:

cf login <endpoint> - login to CF using user credentials

cf push – push code and trigger build and deploy on CF from developer machine command line

cf logs – check deployed app logs

cf scale – scale microservice instances initialized to run n instances at start (run n instances of the microservice at first, thereafter it is managed through autoscaling feature of CF).

# Main API Policies

Policies are configured on the API Gateway for having control security, routing of requests, data masking etc. Some of the most commonly used policies applied on APIs are:

Quota: defines a limit for number of calls a given client can make to an API in a day. For IBM API Gateway, quota is defined as part of the plan subscribed for.

Rate Limit or Throttling: restricts rate at which a client can invoke an API. Malicious scripts can start hitting APIs as 100s of times per seconds. API Management gateway can identify and filter such elements.

OAuth – Security policy to identify and authenticate API caller and control their access privileges

Spike arrest - there may be times of the year or day when certain APIs will experience very high traffic. If such an API is integrating to a backend, it can choke the backend services. This policy allows setting spike limits to ensure desired performance.

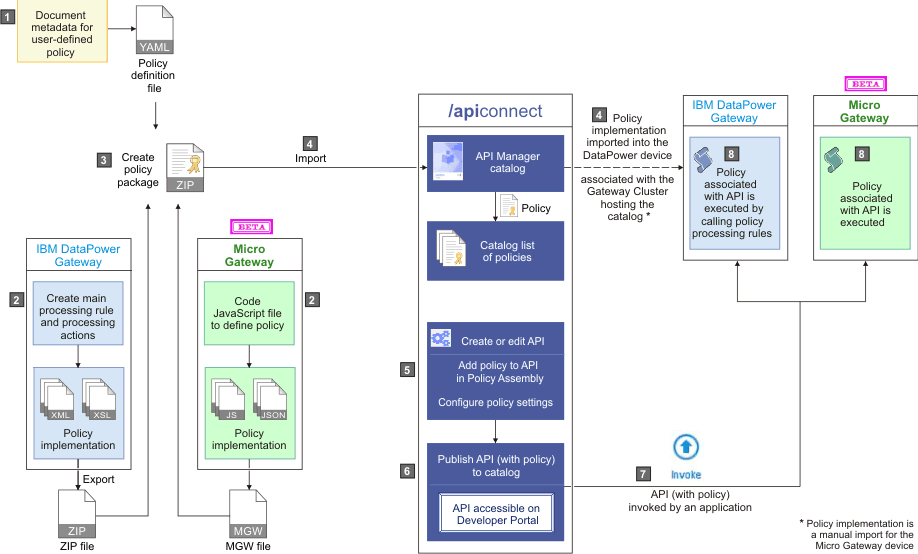
IBM API Connect Gateway also allows user defined policies to be executed. User defined policies are used to augment the activities that are performed by the Gateway. Scenarios when user defined policies may be used are:

* Implement own logic for dynamic routing of requests
* Enforce additional security constraints on APIs

A user defined policy is implemented on the API micro Gateway as a Node.js module. After a policy is imported into an IBM API Connect Catalog, the policy is available to be placed into an assembly flow of an enforced API. When the API is published, and invoked by an application, the API Gateway executes all policies that are associated with this API.

**Developer Notes**: Policies are an important way to control the usage of APIs, monitor and adjust for optimal performance. Policies are additional tools available to developers for pre-processing request and post processing responses from microservices. Some of the common policies and configuration snippets on IBM API Connect are provided in the subsequent section. In some scenarios, policy level controls can be used to respond to clients with required attributes and formats rather than loading more microservices for specific client requests. E.g. an address validation service may return wide dataset which is okay for some clients while others may want only specific fields in specific formats. Instead of building microservices conforming to individual client needs, policies can be applied at Gateway level for the request identification and pre/post processing and filtering.

The following diagram provides an overview of how to create and execute a user-defined policy in IBM API Connect.



Sample code snippets to help the creation of a user-defined policy implementation.

## Access to input properties code snippet

The following code block shows an example of how to access the input properties by using the policyProperties() function. The example defines a property that is named a\_property, which is declared as an integer value, but is retrieved in XSLT as text.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:dp="http://www.datapower.com/extensions"

xmlns:func="http://exslt.org/functions"

xmlns:apim="http://www.ibm.com/apimanagement" extension-element-prefixes="dp func apim">

<!-- Contains the APIM functions -->

<xsl:import href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<xsl:variable name="p" select="apim:policyProperties()" />

<xsl:message>

The value of my input property is

<xsl:value-of select="$p/a\_property" />

</xsl:message>

</xsl:template>

</xsl:stylesheet>

## Access to runtime context code snippet

The following code block shows an example of how to access the runtime context by using the getContext() function.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:dp="http://www.datapower.com/extensions"

xmlns:func="http://exslt.org/functions"

xmlns:apim="http://www.ibm.com/apimanagement" extension-element-prefixes="dp func apim">

<!-- Contains the APIM functions -->

<xsl:import href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<xsl:variable name="client-id" select="apim:getContext('client.app.id')" />

<xsl:message>

The calling application is

<xsl:value-of select="$client-id" />

</xsl:message>

</xsl:template>

</xsl:stylesheet>

## Access to input payload in XSLT code snippet

The following code block shows an example of how to access the input payload by using the payloadRead() function.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:dp="http://www.datapower.com/extensions"

xmlns:func="http://exslt.org/functions"

xmlns:apim="http://www.ibm.com/apimanagement" extension-element-prefixes="dp func apim">

<!-- Contains the APIM functions -->

<xsl:import href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<xsl:variable name="input" select="apim:payloadRead()" />

<xsl:message>

The input payload is

<xsl:copy-of select="$input" />

</xsl:message>

</xsl:template>

</xsl:stylesheet>

The payloadRead() function returns an XML node-set that contains the payload of the request. If the payload is in JSON format, a JSONx node-set is returned that can then be manipulated within an XSLT stylesheet. If the payload is not in JSON or XML format, the node-set that is returned is empty.

The following example shows how to use the payloadType() function to determine what type of payload (XML or JSONx) will be returned by payloadRead().

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:dp="http://www.datapower.com/extensions"

xmlns:func="http://exslt.org/functions"

xmlns:apim="http://www.ibm.com/apimanagement" extension-element-prefixes="dp func apim">

<!-- Contains the APIM functions -->

<xsl:import href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<xsl:variable name="payloadType" select="apim:payloadType()" />

<xsl:message>

<xsl:text>Payload type is [</xsl:text>

<xsl:value-of select="$payloadType" />

<xsl:text>]</xsl:text>

</xsl:message>

</xsl:template>

</xsl:stylesheet>

## Access to HTTP headers code snippet

The following code block shows an example of how to access the HTTP headers in XSLT by using the getContext() function.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:dp="http://www.datapower.com/extensions"

xmlns:func="http://exslt.org/functions"

xmlns:apim="http://www.ibm.com/apimanagement" extension-element-prefixes="dp func apim">

<!-- Contains the APIM functions -->

<xsl:import href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<xsl:variable name="content-type" select="apim:getContext('request.headers.content-type')" />

<xsl:message>

The request content type is

<xsl:value-of select="$content-type" />

</xsl:message>

</xsl:template>

</xsl:stylesheet>

Note:

Access or modification of HTTP headers by using DataPower extensions, such as dp:set-request-header, is not advisable, as such actions might yield unexpected results when the policy is combined with other policies and assembly steps.

## Modify the payload code snippet

The following code block shows an example of how to modify the payload in XSLT. The output must be an XML node-set, which represents either an XML or SOAP message, or a JSON message by using JSONx. To assist the API Gateway policy framework to accept the new or transformed message, call the apim-output template, as shown in the following example.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:apim="http://www.ibm.com/apimanagement"

xmlns:jsonx="http://www.ibm.com/xmlns/prod/2009/jsonx">

<!-- Contains the APIM functions -->

<xsl:include href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<!-- Creates a JSON document (empty object is for simplicity) -->

<jsonx:object>

</jsonx:object>

<!-- Indicates the media type of the output being produced -->

<xsl:call-template name="apim:output">

<xsl:with-param name="mediaType" select="'application/json'" />

</xsl:call-template>

</xsl:template>

</xsl:stylesheet>

where mediaType:

* 'application/json' is when the output is written in JSONx format.
* 'application/xml' is when the output is written in XML format.

Specifying the media type allows the next steps in the assembly flow to understand how to process the new payload.

## Configure error information code snippet

The following code block shows an example of how to configure the policy implementation to produce error information by calling the apim-error template.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:apim="http://www.ibm.com/apimanagement">

<!-- Contains the APIM functions -->

<xsl:include href="local:///isp/policy/apim.custom.xsl" />

<!-- Indicates this policy has a failure and provides

additional information for the client application -->

<xsl:template match="/">

<xsl:call-template name="apim:error">

<xsl:with-param name="httpCode" select="'401'" />

<xsl:with-param name="httpReasonPhrase" select="'Unauthorized'" />

<xsl:with-param name="errorMessage" select="'Please select a Plan'" />

</xsl:call-template>

</xsl:template>

</xsl:stylesheet>

where:

* httpCode is the code of the required error message.
* httpReasonPhrase is the reason for the error.
* errorMessage is the suggested action for the user.

## Set variables code snippet

The following code block shows an example of how to set a runtime variable to a specified string value by calling the setVariable template.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:dp="http://www.datapower.com/extensions"

xmlns:func="http://exslt.org/functions"

xmlns:apim="http://www.ibm.com/apimanagement" extension-element-prefixes="dp func apim">

<!-- Contains the APIM functions -->

<xsl:import href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<xsl:call-template name="apim:setVariable">

<xsl:with-param name="varName" select="'serviceEndpoint'" />

<xsl:with-param name="value" select="'https://endpoint.host.com/data'" />

</xsl:call-template>

<xsl:message>

<xsl:text>Variable [</xsl:text>

<xsl:value-of select="'serviceEndpoint'" />

<xsl:text>] set to [</xsl:text>

<xsl:value-of select="'https://endpoint.host.com/data'" />

<xsl:text>]</xsl:text>

</xsl:message>

</xsl:template>

</xsl:stylesheet>

where:

* varName is the name of the runtime variable where value is to be set.
* value is the string value to set the variable to. This can be a literal value, or another variable. For example, to set a named variable to the value of the Content-Type header in a request, one would specify the value as $(request.headers.content-type).

The following example shows how to retrieve the value of a runtime variable by using the getVariable() function.

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

xmlns:dp="http://www.datapower.com/extensions"

xmlns:func="http://exslt.org/functions"

xmlns:apim="http://www.ibm.com/apimanagement" extension-element-prefixes="dp func apim">

<!-- Contains the APIM functions -->

<xsl:import href="local:///isp/policy/apim.custom.xsl" />

<xsl:template match="/">

<xsl:variable name="varValue" select="apim:getVariable('serviceEndpoint')" />

<xsl:message>

<xsl:text>Variable [</xsl:text>

<xsl:value-of select="'serviceEndpoint'" />

<xsl:text>] = [</xsl:text>

<xsl:value-of select="$varValue" />

<xsl:text>]</xsl:text>

</xsl:message>

</xsl:template>

</xsl:stylesheet>

# API Naming Standards and Guidelines

CitiBanamex has defined API specification guidelines and naming standards. Please refer to the “citibanamex-microservices-specs.pdf” for complete API specs standards.