# Microservices Oriented Architecture (MSA)

by
Dr.Neila BEN LAKHAL

(PhD @ Tokyo Institute of Technology)
Neila.benlakhal@enicarthage.rnu.tn

# Microservices resources guide (1/2)

- Jones, T (2017). “So You’re Thinking of Decomposing Your Monolith into Microservices.”
- Richardson, C (2014). “Microservices: Decomposing Applications for Deployability and Scalability.”
- Montesi, F.; Weber, J. (2016). “Circuit Breakers, Discovery, and API Gateways in Microservices.”
- M Fowler and J Lewis. Microservices. ThoughtWorks, 2014
- https://martinfowler.com/microservices/
- https://microservices.io/
- https://www.programmableweb.com/api-university/microservices-101-understanding-and-leveraging-microservices
- https://blog.codeship.com/so-youre-thinking-of-decomposing-your-monolith-into-microservices/#disqus_thread
- https://www.infoq.com/articles/microservices-intro
- https://www.nginx.com/blog/introduction-to-microservices/

# Microservices resources guide (2/2)


- Building Microservices, designing Fine-Grained Systems, By Sam Newman, Published by O'Reilly Media February 2015.
- Daya, S. et al. (2015). "Microservices from Theory to Practice." IBM Redbook (SG24-8275-00).
- Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-reconfiguring microservices. In Theory and Practice of Formal Methods, pages 194–210. Springer, 2016.
- Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. CoRR, abs/1606.04036, 2016.
- Montesi, Fabrizio and Janine Weber. "Circuit Breakers, Discovery, and API Gateways in Microservices." *CoRR* abs/1609.05830 (2016)
- Kalske M., Mäkitalo N., Mikkonen T. (2018) Challenges When Moving from Monolith to Microservice Architecture. In: Garrigós I., Wimmer M. (eds) Current Trends in Web Engineering. ICWE 2017. Lecture Notes in Computer Science, vol 10544. Springer, Cham.
- IBM microservices courses (2018).

# XWhy so many companies already said Yes to Microservices Architecture (MSA) ?

# Companies using Microservices

- Though Microservices is a newly coined concept, Nowadays, hundreds of systems are already built using microservices .To name but a few :









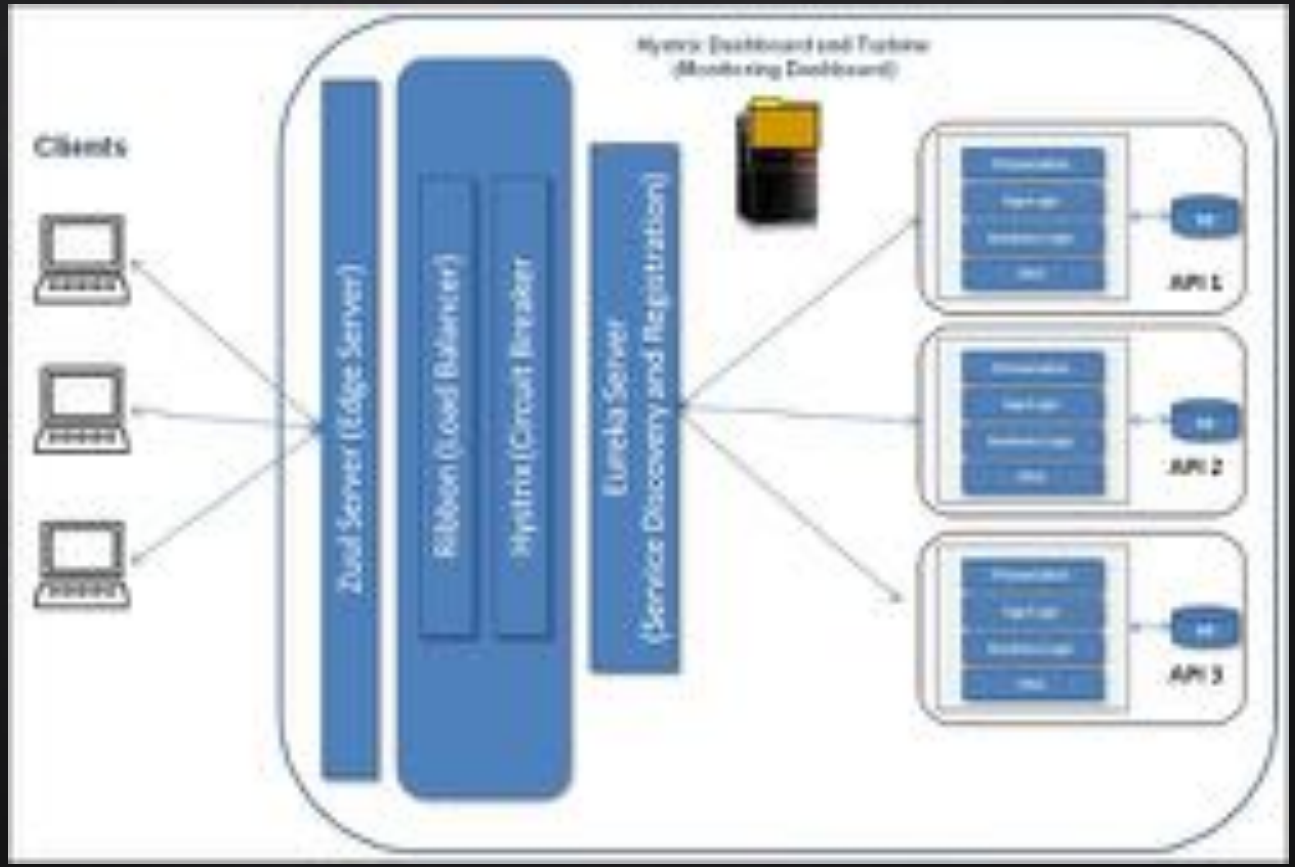
# Netflix

- We can't talk about microservices without evoking Netflix.
- Netflix are pioneers in microservice architecture.
- They were the first to successfully implement MSA on a large scale.
- Why ?
- In 2008, when the Netflix application was **monolithic**, the entire Netflix website was brought down because a single missing semicolon led to a major database corruption[1].

- Netflix Solution : Netflix Open Source Software Center (OSS)
- **What is a monolithic application ?**

neila.benlakhal@enicarthage.rnu.tn

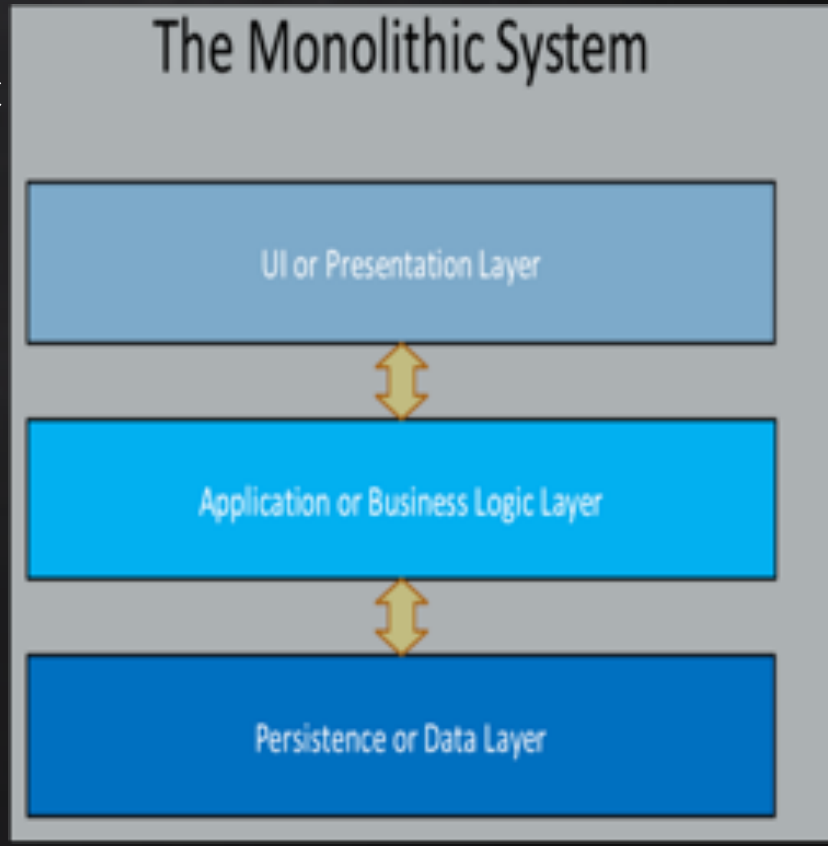
# Microservices architecture using Netflix OSS



# What is a monolithic application ?

- to understand what is a monolithic application, we need to have a closer look at how systems are built:
  - We Have Data
  - We write code to access that data
  - We then provide interfaces to view or edit this data

  => 3-tiers architecture.



# Monolithic application ?

- A monolithic application is an application where **all of the logic runs in a single app server**.
- Typical monolithic applications are **large** and built by **multiple teams**, requiring careful orchestration of deployment for **every** change.
- The single monolith would encompass **all** the business activities for a single application.
- **All components are tightly-coupled and interdependent.**

# When to use monolithic application ?

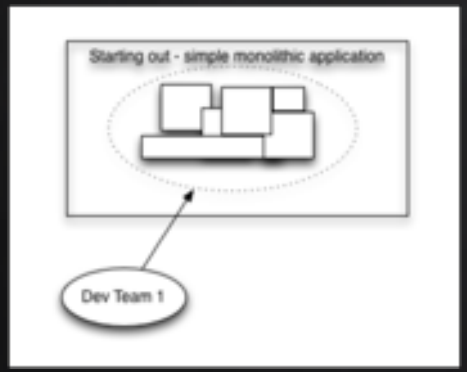
- For **simple** applications, it is always good to go with monolithic since they are:
  - Simple to develop,
  - Simple to deploy and
  - Simple to scale.

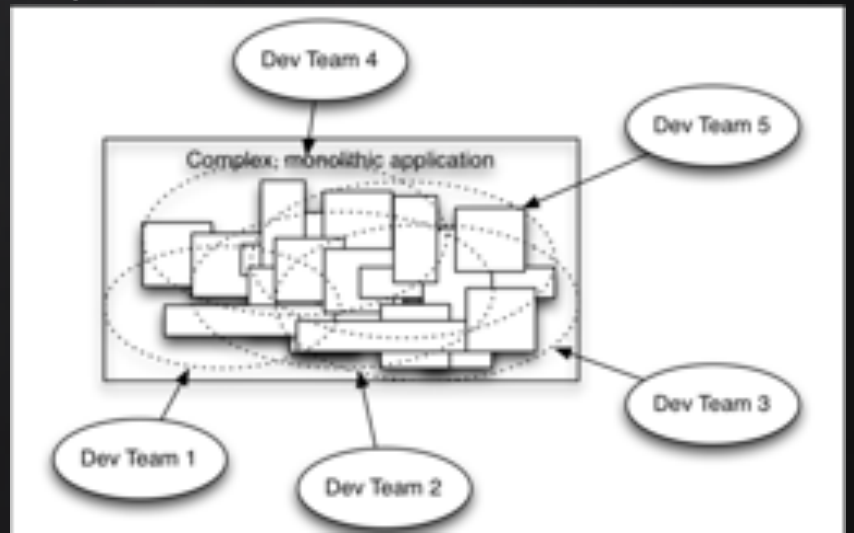
However, for **complex** applications, that need to scale rapidly, requiring frequent and easy releases, microservices-based architecture is a better choice.

neila.benlakhal@enicarthage.rnu.tn

# How a monolithic application grows

- Your once-simple application has become large, complex, and intertwined.
- Multiple independent development teams work on your application **simultaneously** to add new features.
- These presumably independent development teams are actually not independent at all, since they are working on the **same** code base and changing the **same** sections of code.



# Problems with monolith architecture 1/3

## 1-Frequent releases are not possible

- BP frequent changes requires frequent code changes
  - Enterprises need about 2 months to release one version, at best!
- For example, a change occurred in one component only but due to this change whole application will go down :
  - We need to deploy all the application since we have one .war file.
  - A lot of developers will be involved and testing efforts will be high to test whole application.
  - New features must be thoroughly tested before they can go live: Unit Tests, Integration Tests, User Acceptance Tests, etc. The testing itself may take a month or more to complete.

# Problems with monolith architecture 2/3

## 2-Complex and tightly coupled code base

- The application will have a tightly wired code which is very hard to maintain.
- Training for the new member will be complex and documentation will be so confusing.
- It is impossible to know precisely  who is working on what piece of the application at any point in time,
- Code-change collisions,
- Code quality—and hence application quality and availability—are likely to suffer,
- It is hard for individual development teams to make changes without having to deal with the impact of other teams,
- Conflicts with incompatible changes,
- Etc.

# Problems with monolith architecture 3/3

## 3-Difficult to manage the team

A team working on a change can block other teams.

## 4-Interoperability is not possible

Modules are tightly coupled, will not have the facility to change the technology stack of a specific module.

## 5-IDE overhead

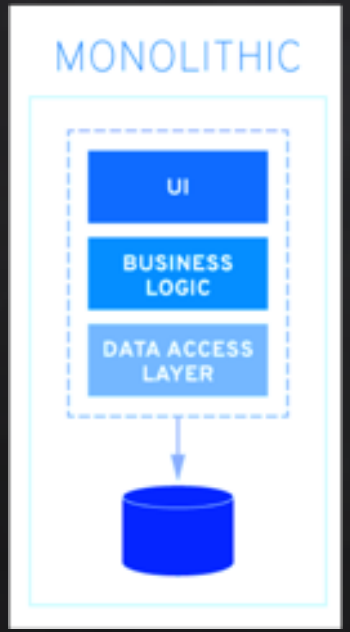
Each developer will have to load whole application code in the local workstation that will slow the performance of IDE and reduce the efficiency and productivity of an individual.

## 6-Scalability of application is expensive and difficult

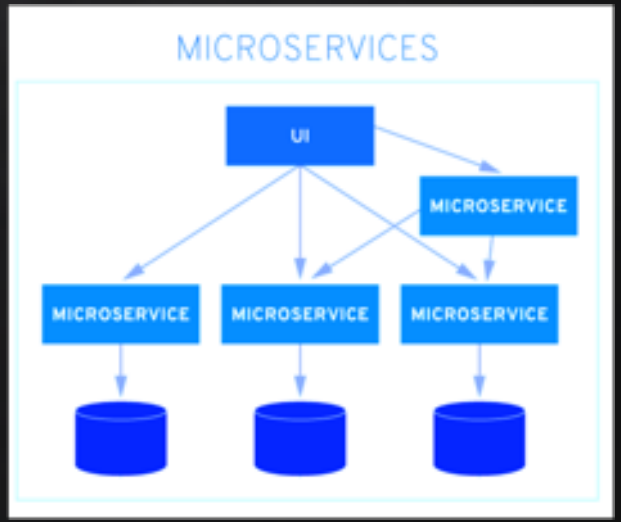
A monolithic application is very tightly coupled and has interdependent complex wirings so it would be very tedious to scale it.

# Solution: split the monolith vertically 1/2

- Instead of cutting the system from a **technical** perspective **horizontally**



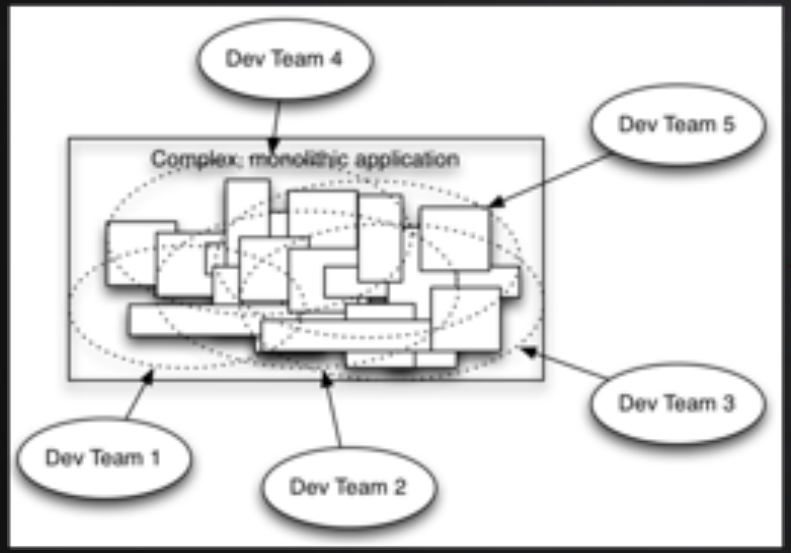

- Split the system **vertically**
- Each service is a complete coherent task (business logic)
- Each task is a microservice (MS)

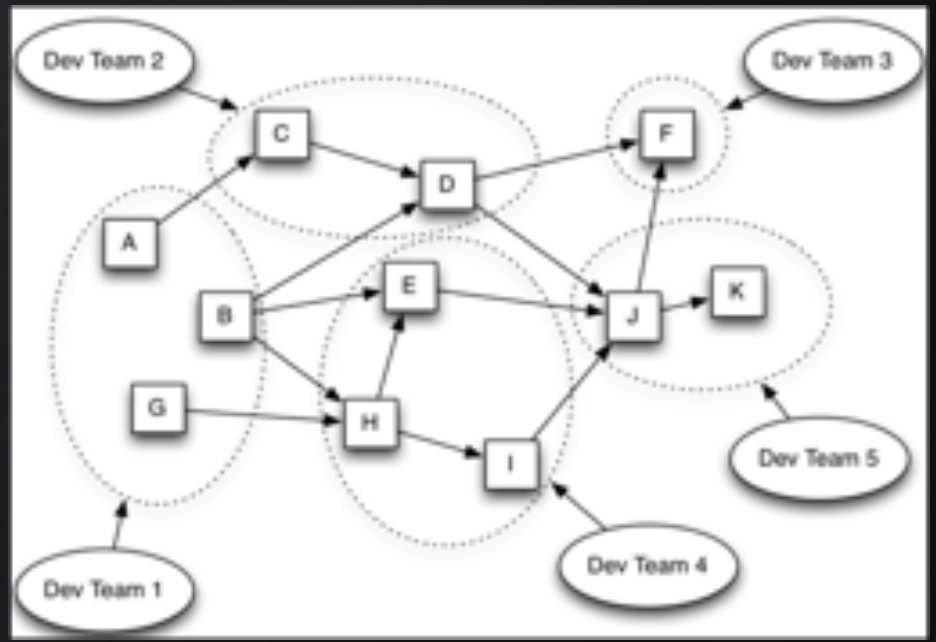


neila.benlakhal@enicarthage.rnu.tn

# Solution: split the monolith vertically 2/2

- The same application constructed as a series of **microservices**. Each microservice has a **clear team owner**, and each team has a clear, non-overlapping set of responsibilities.



## Data in Traditional approach

- Single monolithic database
- Tiers of specific technologies


## Data in Microservices approach

- Graph of interconnected microservices
- State typically scoped to the microservice
- Remote Storage for cold data

# Microservice definition

Microservices are small software components that are specialized in one task and work together to achieve a higher-level task.

Similar to Web services definition !

# Microservices are :

## Small, Autonomous services, that work together:

How small is small?

- There's no clear definition !
- Something that could be re-written in roughly two weeks time

## Capable to exist autonomously:

- Shall be independently deployable
- Ideally on a separate host
- All communications to outside world are via network calls
- Interface to communicate should ideally be technology independent

# What are microservices ? (more complete definition)

- In short, the microservice architectural style is an approach to developing a single application as a suite of **small services**, each running in its own process and communicating with lightweight mechanisms.

- These services are built around business capabilities and **independently** deployable by **fully automated** deployment machinery.

- 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.

-- James Lewis and Martin Fowler(2014)

# Another definition (IBM)

"A microservice is an engineering approach focused on **decomposing** applications into **single-function** modules with **well-defined interfaces** which are **independently deployed** and operated by **small teams** who own the **entire lifecycle** of the service."

--IBM

Microservices accelerate delivery by :

Minimizing communication and coordination between people while reducing the scope and risk of change.

# Team communication/scope of change

- Reduce cross-team communication
  - Members from the same team developing the same MS can communicate incessantly, however
  - Cross-team communication is minimized to the maximum
  - The only essential cross-team communication is to agree on the interfaces each service is going to have. e.g., ask the other team for string as output type.

- Reduce scope of change
  - Improve your service, fixe bugs, make new versions, without affecting the surrounding services.
  - The scope of change is within the team, the interface won't be modified.

# Microservices architecture (MSA)

- In MSA, large complex software applications are composed of one or more services.
- Each of these microservices focuses on completing one task only and does that one task really well.
- Each one task represents a small business capability.

# Définition microservice (Fr)

- « Un microservice est une approche adoptée dans l'ingénierie des SI qui repose sur la décomposition des applications en modules à fonction unique.

- Chaque MS est doté d'une interface bien défini, il est déployé et exploité indépendamment des autres MS, par une équipe à taille réduite, qui a la responsabilité de gérer tout le cycle de vie du MS en question.

- Les MS accélère la livraison des logiciels en minimisant la communication et la coordination requises entre équipe de devs et en réduisant l'étendu des modifications potentiel. »

# Example : Flight reservation application

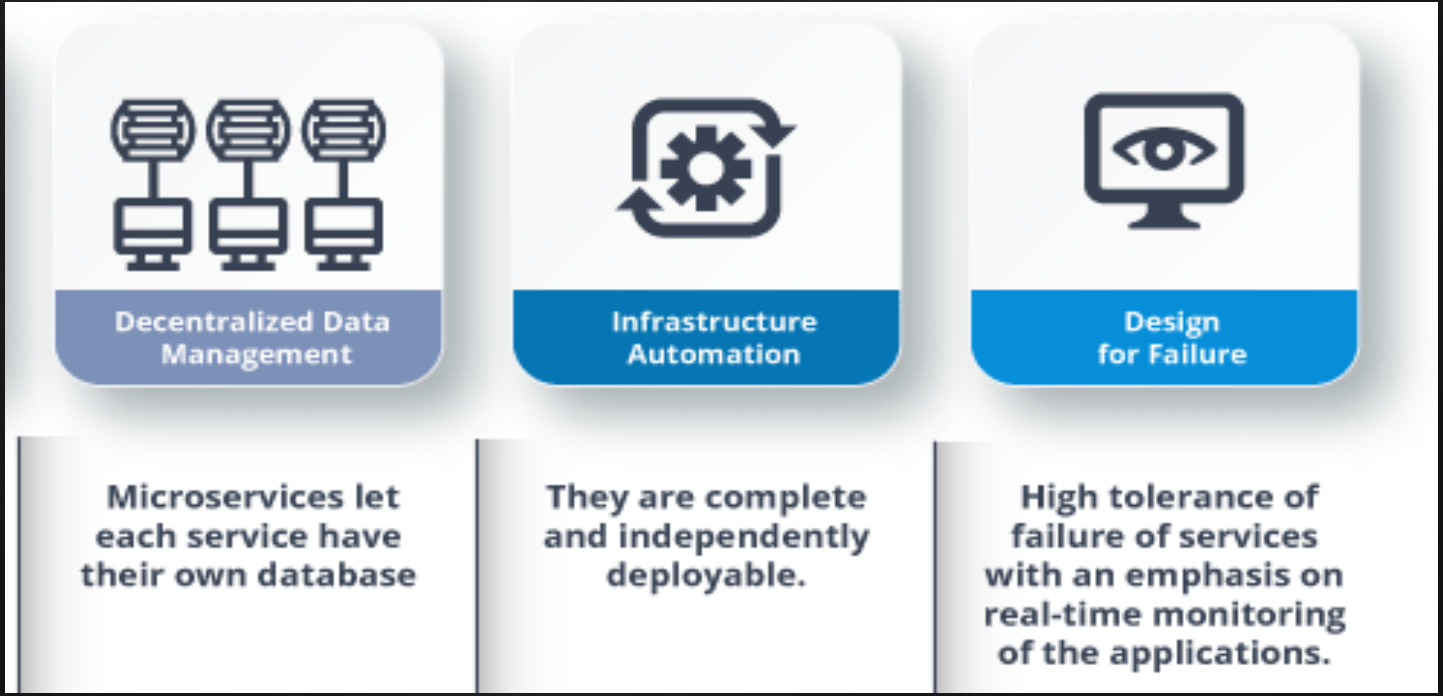
- Application deployed as a set of microservices
- They are independently deployed, invoked, managed.
- Number of version/instance run from each MS is different
  - Many invocation of the `bookFlights` microservice, but not all of them require the invocation of the `RewardsProgram` microservice.

- Deploy the application in cloud is essential!

neila.benlakhal@enicarthage.rnu.tn

# Microservices characteristics 1/2

# Microservices characteristics 2/2



# MS characteristics (more details) 1/6

## 1. Small and focused

- MS need to focus on a specific unit of work, and as such they are small.
  *There are no rules on how small a MS must be but it must be small enough so that you can rewrite and maintain the entire MS easily within a small team.*
- To guaranty rapidity of service delivery, each MS should have its own source code management repository.
- Each MS has its own *presentation+ business + data layer*
- Each MS must be deployed separately (VM or container)
- Size of the team building the MS? If you cannot feed the team building a microservice with 2 pizzas, your microservice is too big!
- Reuse is not the main issue when deciding about service granularity, rapidity/service responsiveness/service delivery are more important !

# MS characteristics (more details) 2/6

## 2. Team-owned

- Each microservice has its own team, responsible for the whole Lifecyle of the MS.
- The team is small in size
- Must be able to maintain, front-end, back-end and data layer (web developer, database administrator, etc.)

# MS characteristics (more details) 3/6

## 2. Loosely coupled

- You need to be able to deploy a single MS on its own.
- There must be zero coordination necessary for the deployment with other MS.

➔Enables frequent and rapid deployments.

# MS characteristics (more details) 4/6

## 3. Language-neutral

- Microservices need to be built using the programming language and technology that makes the most sense for the task at hand.
- Communication with microservices is through language-neutral APIs, e.g., HTTP.

You should standardize on the integration and not on the platform used by the microservice !

# MS characteristics (more details) 5/6

## 4. Bounded Context

- *A bounded context refers to the coupling of a component and its data as a single unit with minimal dependencies.*
- Each microservice must **not** "know" anything about underlying implementation of other microservices surrounding it.

# MS characteristics (more details) 6/6

## 5-Scalability

*"Scalability is the ability of the whole system to cope with the demand as the company grows bigger".*

- Example : Amazon black Friday.

In monolithic application, if a service across the application performs significantly slower than the rest, it will cause the entire application to fail

- In MSA, every microservice should be scaled-up without interfering with the rest of the system
- Since MS are isolated, possibility to run various number of instances.

# Others characteristics

## Technology Heterogeneity

Since all the smaller systems are autonomous, they can be built using separate technologies.

## Resilience

Microservices are isolated from each other
If one service is down, the others may still keep going

Fault isolation and repair becomes easier

## Ease of Deployment

In monolithic systems, changes are not easy :Every change may involve a complete re-build, and then a re-deployment.

Since all microservices are deployed independently

Changes to one can be deployed without affecting others (changes which are not breaking changes)

If the new release doesn't work, a quick restore can be made to previous version

## Organizational Alignment

Teams can be built around business contexts rather than technology

One microservice can be owned by one team

No external involvement, teams can be responsible for the full lifecycle of the service, including operation and maintenance

## Optimized for Replaceability

Microservices are optimized for replacement
Since they are small, it is easier to write them from the scratch and replace

# MS vs Big Web services

- Microservices are:
  - Developed in any programming language.
  - They communicate with each other using language-neutral application programming interfaces (APIs).
  - They don’t need to know anything about underlying implementation or architecture of other.

- ***These are also properties of Big Web services in SOA !***

# Microservices disadvantages

## MS are message communication-based over a network

- **Network Calls are slower** : Out-of-process call, like HTTP request is always slower than in-process call like method invocation
- **Networks can fail** : Difficult to figure out if a MS is down, or is out-of-reach
- **Networks are insecure**

## More hosts to manage

- Typically each microservice runs on a different host
- A host could either be a physical machine, a virtual machine or even a docker container
- More microservices mean more hosts to manage

## Distributed Transactions

- With one database or process, transactions are easier to implement
- A rollback in case of failure is easy to achieve
- With multiple processes and database instances, each keeping track of some part of an overall business transaction, it is difficult to revert changes in case of failures

## Distributed Monitoring and Logging

- Logging and monitoring one host is significantly simpler than a cluster of hosts
- Without automation, things can quickly become unmanageable

# Monolithic architecture VS microservices architecture

| aspect | Monolithic | Microservices |
|---|---|---|
| codebase | There is a single and often very large codebase for the entire application. | consists of multiple, smaller sized codebases; each service powering the application has its own codebase. |
| Database | Use a single logical database, often relational, for an entire application or set of applications. Monoliths allow multiple pieces of data to be updated together in a single transaction | each service has its own unique database system or share the same database system but have different instances. The persistent data for each microservice is kept private and is only accessible via its API. |
| Development Teams | Teams are larger compared to teams of MS | Microservices teams tend to be smaller in size compared to the teams of a monolithic architecture. Each team is in charge of the whole lifecycle of MS. |
| Programming languages | One or at most small set of programming languages. Difficult to use other language, all the application must be rewritten. | Each service can be developed with a different language, framework, and/or library. |
| Testing | Application is built, deployed, and scaled as a single unit making it much easier to test than a distributed application<br>End-to-end testing can be easily implemented for a monolithic application. | MS is more complex to test: each MS that needs to be tested :If one of the microservices needs to be tested, it would have to be launched along with all of the other services that it depends on. |

# How to go from a monolith to MSA?

# Principes a ne pas quitter des yeux lors de la mise en place dune MSA

Lorsque vous êtes sur le point de migrer vers/construire une MSA, il faut veiller au respect des principes suivants :

## Principe 1:

- Chaque service a son propre processus: appliquer la règle un «container » par service.
- Les services sont optimisés pour effectuer une seule et unique fonctionnalité.

## Principe 2:

- « One business function per service » principe de la fonction unique: l'unique raison pour que un service change est lorsque sa fonction change.

# Principes a ne pas quitter des yeux lors de la mise en place dune MSA

Principe 3:

- La communication doit se faire directement via REST/SOAP API ou via un bus de message.

Principe 4:

- Chaque service doit obéir à un cycle d'intégration et de déploiement propre à lui, les services évoluent à des rythmes différents.

Principe 5:

- La haute disponibilité par service: le niveau de scalabilité requis pour un service n'est pas automatiquement le même pour les autres : augmenter/réduire le nombre d'instance par service.

# Approaches de mise en place d'une MSA

## Approche 1: « Build MSA from scratch »

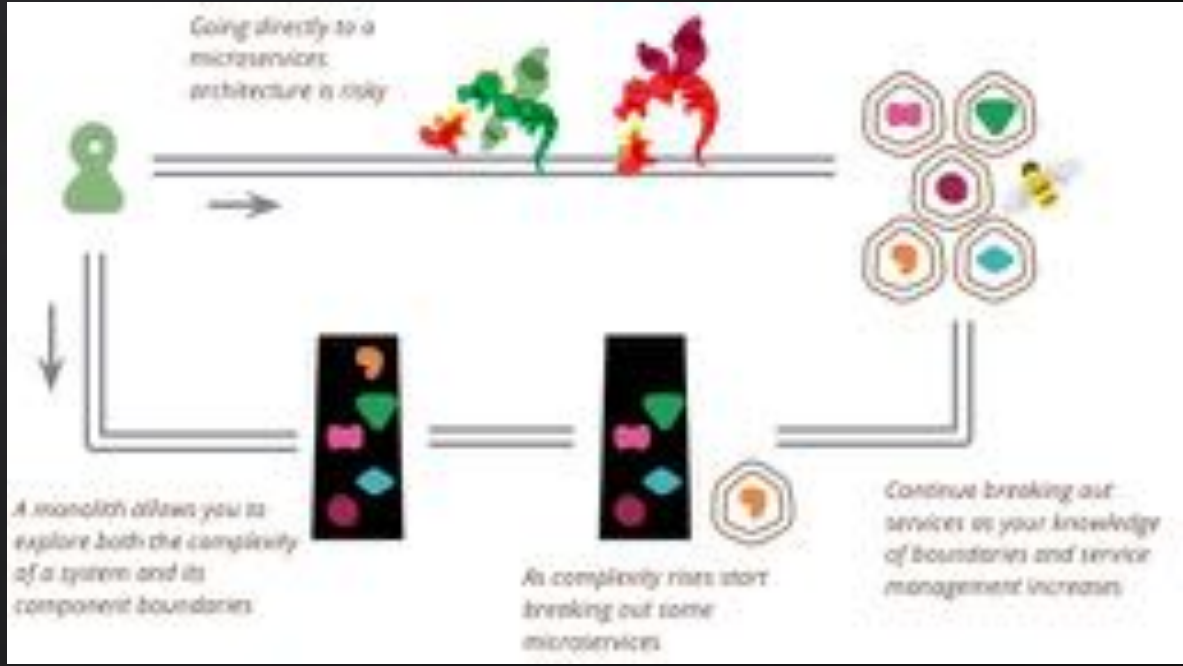
- Se lancer directement dans l'implémentation de MS.

## Approche 2: « Monolithic-First »

- Commencer à partir d'une application monolithique. 3 stratégies possibles :
  - Stratégie 1: Construire une application monolithique tout en gardant dans l'esprit qu'on passe vers une MSA.
  - Stratégie 2: « Peel away MS from the edges of an existing monolithic application » :Commencer par les parties extrêmes/sur le bord de l'application monolithique et les transformer en MS. Laisser le cœur de l'application monolithique.
  - Stratégie 3**:** « iterative strategy » : commencer par découper en des services de granularité forte « coarse-grained services »; laisser les équipes s'habituer au développement de MS, et les découper en service plus fin « fine-grained services».

# Quelle approche est mieux ?

- Il n'y a pas de solution sûre vu que l'architecture MSA est une nouvelle technologie non mature.
- Recommandations :
  - Ne pas se lancer dans l'implémentation d'une MSA directement tant que les équipes n'ont pas assez d'expérience dans le développement de MS.
  - Eviter au mieux la première approche : très couteuse.
  - *L'utilisation des MS n'est pas un plus dans tous les SI (M.Fowler).*



# How to decompose a monolith in MS ? 7 Recommendations 1/2

Suivre les recommandations suivantes:

1. Dans une MSA, les MS sont tous réunis autour de l'aboutissement d'un objectif commun, il faudrait donc commencer par décomposer le monolithe d'un point de vu **métier « business logic »** et non pas d'un point de vu **technique « technical perspective »** comme il est le cas pour les application monolithique.
2. Ne pas considérer l'application dans sa totalité, mais partie par partie, dans la limite du possible.
3. Commencer par identifier les fonctionnalités qui vous semblent plus candidate à être séparée du monolithe :
    1. fonctionnalités qui sont logiquement séparées; ex., pour un BP réservation d'un vol, le conversion de monnaie est une tâche annexe.
    2. Le code qui peut être directement séparés, ex., écrit dans un langage diffèrent, manipule une BD indépendante, etc.
4. Pour chaque fonctionnalité métier identifiée comme indépendante, voir si elle dispose déjà dune interface; est-elle faiblement couplée à d'autres codes?

# How to decompose a monolith in MS ? 7 Recommendations 2/2

5. L'aspect « scalability » : identifier les fonctionnalités métiers/les codes qui ont tendance à causer des « bottlenecks » vu qu'ils sont très fréquemment invoqués et commencer par ces codes.
6. L'aspect « team size »: Considérer l'organisation des équipes : partager le code en MS selon les équipes qui se sont déjà formées. Séparer le code des équipes géographiquement délocalisées.
7. L'aspect « Fault-tolerance »: identifier les sections qui ont un taux d'échec important; les décomposer facilitera le monitoring, permettra de les isoler, de prévoir des instances en plus pour eux et des traitements d'erreurs juste pour eux.

Identifier le minimum de MS qui vérifient les propriétés des MS et qui vérifient ces 7 recommandations. Commencer même avec un seul. Gagner de l'expérience avec puis passer au développement d'autres MS.

# Refactoring an existing code base of a monolith, how to ?

- Si le code de l'application est non modulaire, comment faire ?
  - Les APIs: Séparer le code de chaque SOAP API|REST API|EJB à part
  - Les BDs:
    - Identifier les tables interrogées par un seul code, les séparer dans une nouvelle BD a part
    - Identifier les tables selon la fréquence de leur interrogation; séparer-les dans une BD a part
  - Les packages: faire le re-packaging des applications (les .war) par fonctionnalité et non pas par application.

# Communication in MSA

- Une fois le « refactoring » fait, les MS identifiés, il faut garder a l'esprit que une MSA est distribuée par nature et doit être déployer dans le cloud pour garantir l'aspect scalabilité.
- L'aspect **communication** est très critique.
- Les méthodes de communication adoptes dans les applications monolithiques ne sont pas suffisantes :
  - La couche **métier** est distribuée,
  - La couche **donnée** est distribuée.

## Data in Traditional approach

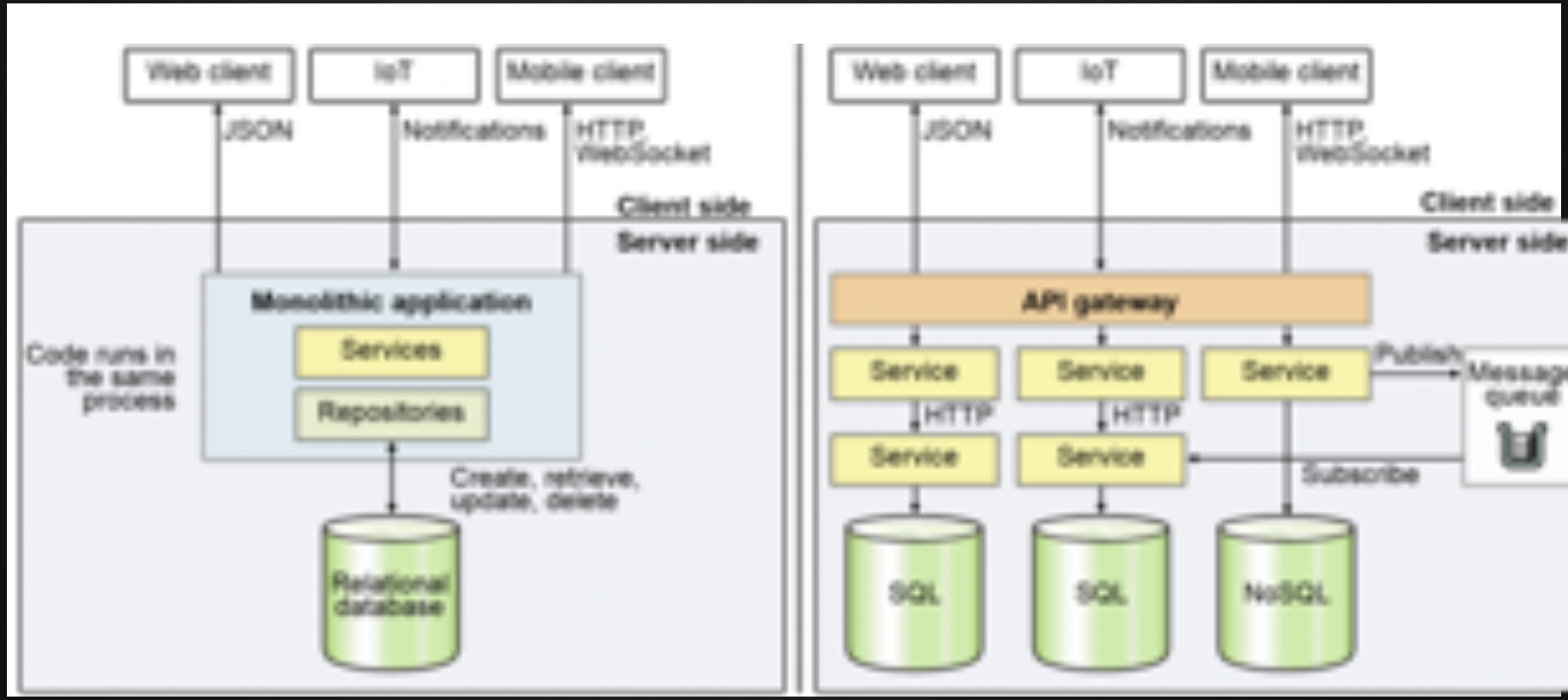
- Single monolithic database
- Tiers of specific technologies


## Data in Microservices approach

- Graph of interconnected microservices
- State typically scoped to the microservice
- Remote Storage for cold data

neila.benlakhal@enicarthage.rnu.tn

# Data is distributed/code is distributed



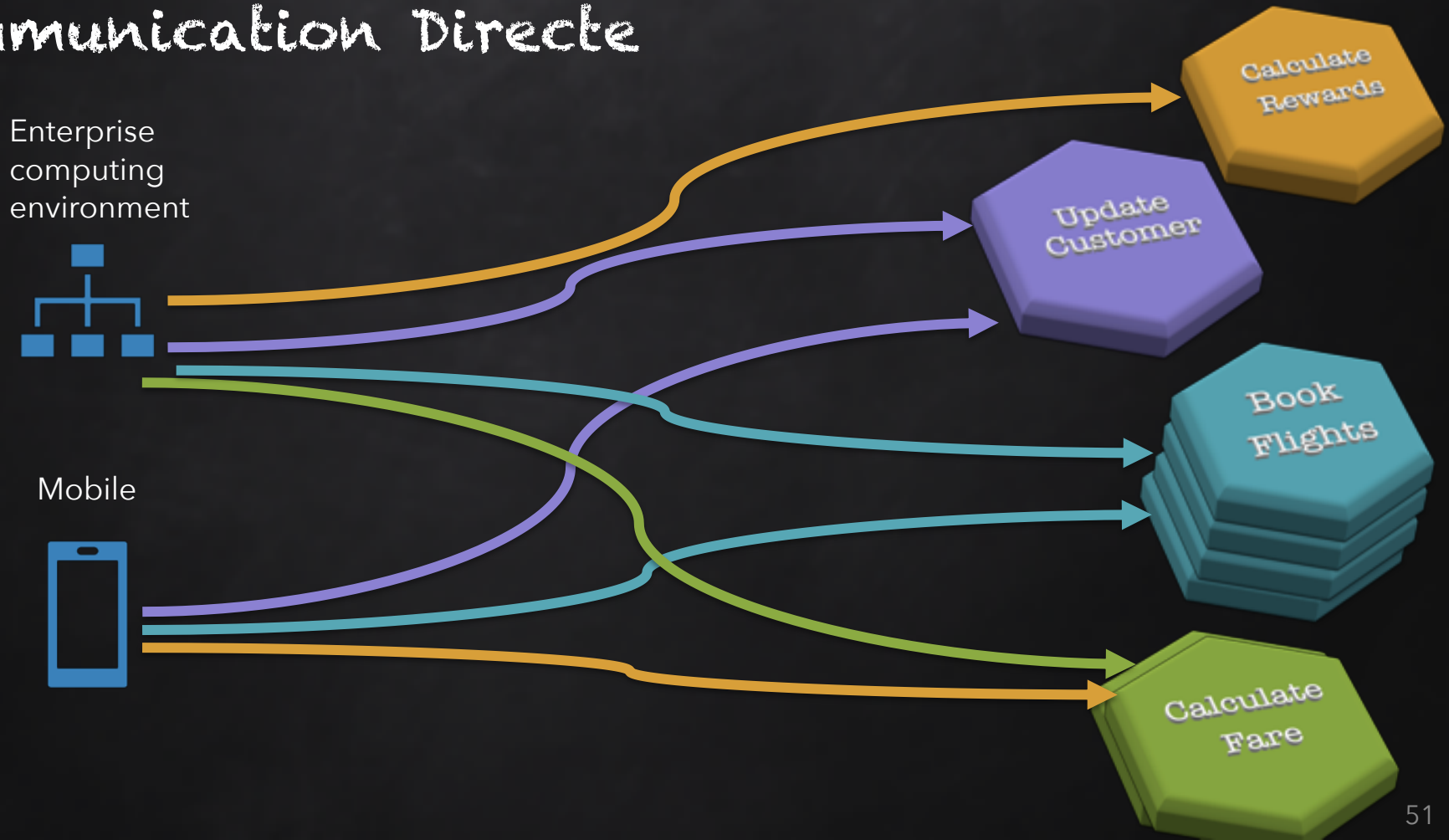
# Communication in MSA

- Microservices ont besoin de communiquer:
  - Avec l'environnement externe (between environment) : e.g., clients, servers, virtual machines, etc.
  - Avec l'environnement interne (within one environment) :e.g., microservices co-localisés, avec la couche Data, etc.
  - Modes de communications :
    1. Directe ou INdirecte.
    2. Synchrone ou Asynchrone.
- Chaque mode de communication va se faire avec des mécanismes et des technologies différentes.

# Communication between environments

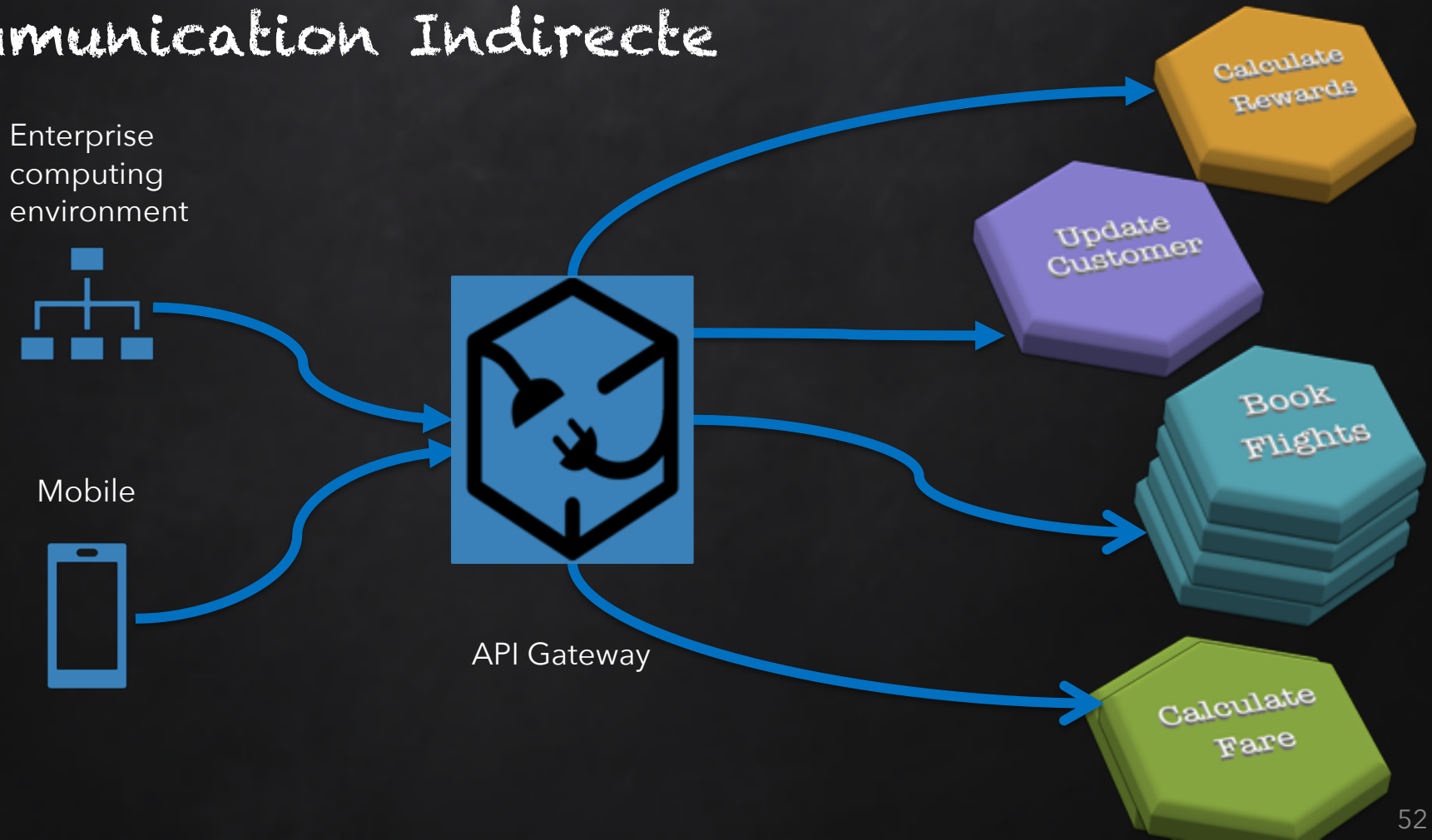
- Communication Microservice---Client:
- Client : Interface Mobile, Entreprise computing env. Etc.
- Problèmes potentiels:
  - Même si on assume que selon le contrat initial, le client doit disposer de l'@ du MS, les MS ne sont pas des instances uniques!
  - Comment reconnaitre l'instance déjà utilisée?
  - Les MS sont déployés dans le cloud, des instances sont crées et supprimés en continu.
  - Un service peut faire un Timeout ou prendre beaucoup de temps avant de répondre à un client (nombre important d'invocation).
- Solution : adopter le mode de communication Indirecte.

# Communication Directe



neila.benlakhal@enicarthage.rnu.tn

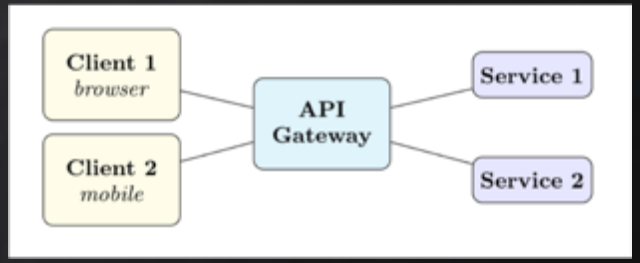
# Communication Indirecte



# API Gateway

- The API Gateway is a server that acts as an API front-end.
- Role:
  - It passes a request to the appropriate microservice on the back end, and then passes the response back to the requestor.
  - The requestor of a MS must keep the URL of the API Gateway static and constant.
  - The API gateway is responsible for identifying the address of the MS instance the requestor (or other co-located microservice) has invoked.
- Others roles:
  - Orchestrate and modify requests and responses in real time, apply security policies, etc.
  - Can be assimilated to an ESB in SOA.
- Advantage: It is only the API gateway that needs to know where the Microservice instance is situated.

# API Gateway 1/2




Fonctionnalités : (Facade Pattern)

1. Jouer le rôle d'un front-end(intermédiaire) entre MS et parties clientes.
2. Permettre l'interaction du MS avec des clients ayant des interfaces divergentes et manipulant des formats de données différentes.
3. Passer les appels aux MS concernés, réceptionner les réponses et les transférer aux clients.
4. Doit disposer d'une adresse **statique unique** connu par tout client.
5. Le client a juste besoin d'avoir l'adresse de l'API Gateway qui va lui fournir l'adresse de l'instance du MS qu'il a déjà utilisé.
6. En contre partie, c'est a l'API Gateway de répertorier les adresses des MS/instances.

# API Gateway 2/2

L’API Gateway constitue le point d’entrée à l'architecture MSA, il doit donc implémenter plusieurs autres mécanismes:

1. Service Discovery,
2. Load-balancing,
3. Security,
4. Monitoring,
5. Failure handling,
6. Authentication etc.

Exemples API Gateway : Zuul, AWS, etc.

# Synchronous/Asynchronous Communication

- Communication between a MS and its environment or between co-located MS can be synchronous or Asynchronous :
- Synchronous communication
  - The requester is waiting on the response.
  - Used protocol : HTTP/HTTPS : message formatted as REST/SOAP message.
- Asynchronous communication
  - Requires a **message broker** to hold the message until the receiver is ready to pick it up.
  - The holding point is called a **request queue** and the requester needs to know the address of the request queue rather than the service being requested.
  - The messages is stored and at some future moment it is pulled off the queue by the endpoint service.
  - The response follows a similar route : It is stored on a **response queue** then forwarded to the original requestor.
  - Used protocol: AMQP . (Advanced Message Queuing Protocol) is the protocol used by the **message broker** for messaging.
  - Example of message Broker : **RabbitMQ**

# Synchronous vs asynchronous communication

- Asynchronous delivery is preferred because there is more control over what is happening to the message.
  - For example, if the delivery fails, it can be attempted again without the requester having to resend it.
- Hybrid communication :

synchronous request + **A**synchronous response:

  - Message sent directly from requestor to MS,
  - Requestor makes a synchronous request but then looks for the response on a queue.

## Circuit breakers

La communication entre MS est essentiellement via message ce qui augmente les possibilités d’ échecs a l’ invocation et les Timeouts

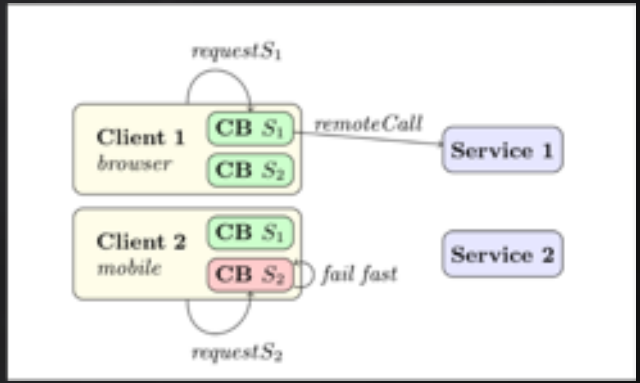
**Solution: Circuit Breaker**

# Circuit Breaker Pattern

- Definition :"prevent the failure of a single component to cascade beyond its boundaries, and thereby bring the entire system down with it".
- Rôles :
  - Un « circuit breaker » contribue à la résistance à l'erreur côté client et serveur :
    - **Coté client**: Evite aux clients de gaspiller leurs ressources en essayant d'accéder à un service indisponible
    - **Coté server**: Les services « overloaded » sont épargnés et ne sont plus invoqués. Ils utilisent leurs ressources pour traiter les invocations cumulées.
  - Evite les « cascading failures »: si un MS1 repose dans son exécution sur un MS2, et MS2 échoue, alors MS1 aussi échoue.
- Comment il fonctionne ?
- Il enveloppe les invocations des MS instances et fait leurs monitoring.
  - Lorsqu'un service est très souvent « unresponsive » ou il est trop lent à répondre, il sera considérer comme défaillant par le circuit breaker. Et a toute invocation un message « fault » sera envoyer au client.

# Circuit breaker(CB) pattern implementation 1/2

- Il peut être implémenter **Coté client :**
  - L'appel vers le MS passe par le CB
  - Quand le CB est ouvert, aucun appel n'est achemine au service.

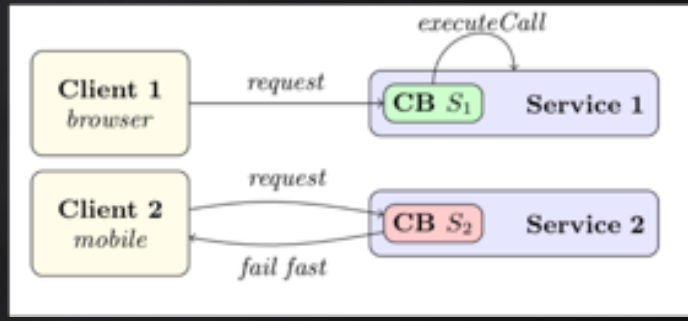



## Problèmes:

- Comment imposer au client de rajouter le code CB dans leur client qui peut être considéré comme malicieux?
- L'information sur la non-disponibilité du service et qui est utilisé coté client n'est pas général; elle a calculé juste en se basant sur ses propres invocations.

# Circuit breaker(CB) pattern implementation 2/2

- Il peut être implémenté coté service:
  - Directement implementable coté service.
  - Possibilité d'estimer la disponibilité du service de manière plus exacte en utilisant le log des invocations de plusieurs clients à la fois.




Problèmes:

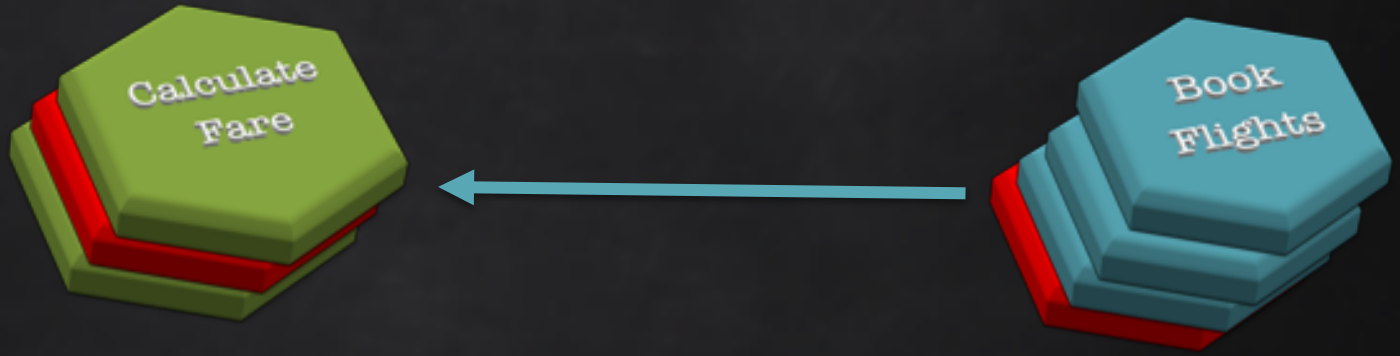
- Il est nécessaire de continuer de recevoir les appels même lorsque le CB est ouvert
- La règle de « single task per service » n'est plus respecté vu qu'il va avoir un autre code implémenté (CB).

## Microservice registration & discovery

Two Microservice instances might be located in the same place (e.g., container/cluster) but still there is this question, how do they know where to find each other?

Solution : service discovery & service registry

# Service registry ?




- How BookFlights can find one of these instances of CalculateFare and make sure that it is a valid one* ?

- Solution : service Registry (SOA)

*Failed Instances are in RED

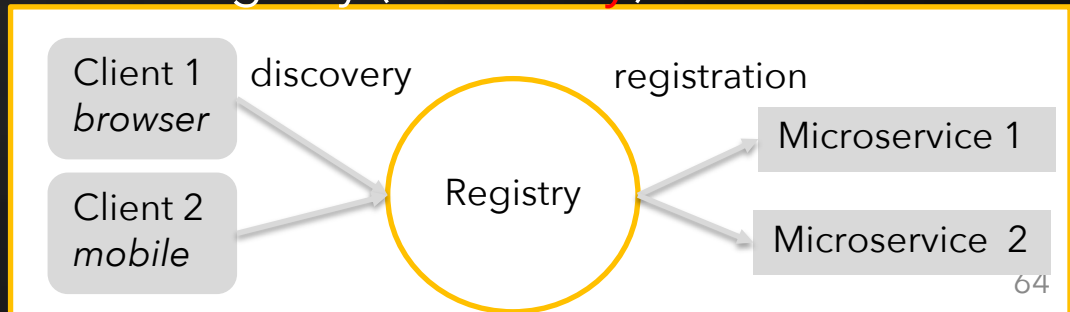
# Service registry ?

## Definitions:

- "A service registry is a service that can used by other components to retrieve binding information about other components."
- "A service registry is a database populated with information on how to dispatch requests to microservice instances."

## Interactions between the registry and other components:

- Interactions between microservices and the registry (registration) :
  1. Self-registration
  2. Third-party registration
- Interactions between clients and the registry (discovery):
  1. Client-side discovery
  2. Server-side discovery



# Service registration methods

- 1-Microservice self registration :
  - When a Microservice instance is created, it might register itself.
- 2-Third party registration :
  - A specific service that manages all the other services and records information about them in the registry.
  - Instances of the Microservice are tested by the registry :
    - **Failed instances** are flagged and their references are no longer handed to a service requestor.
    - **An instance flagged as being not faulty** is selected, and the reference handed back the requestor which can now send its request to that particular instance of the Microservice.

neila.benlakhal@enicarthage.rnu.tn

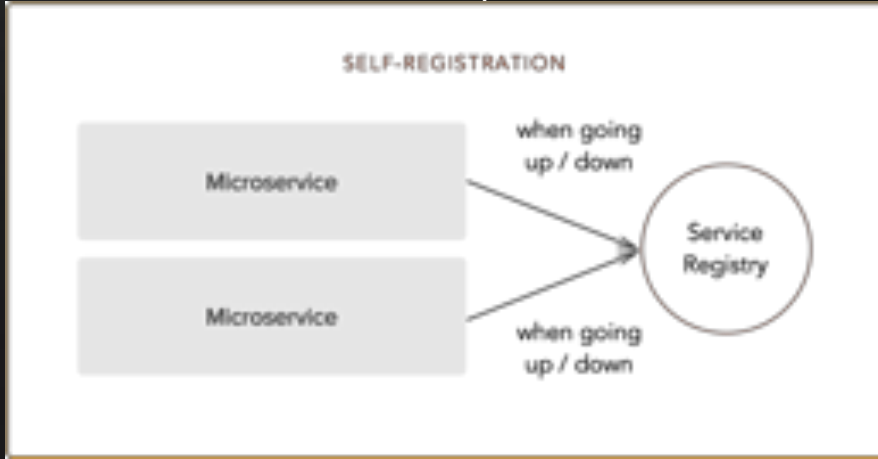
# 1-Microservice self registration

- Comment les instances s'enregistrent dans « Service Registry »?
  - Quand l'instance est crée, elle appelle le « service Registry »et y enregistre ses propres informations.

Problèmes :

- La règle de « single task » du service n'est plus respectée vu qu'il va y avoir un code en plus pour s'enregistrer et s'effacer du registre.
- Le même code redondant va être rajouter a plusieurs services.
- Les instances de MS qui échouent ne peuvent s'effacer du registre.

Solution : third-party registration.



neila.benlakhal@enicarthage.rnu.tn

# 2-Third-party registration

- Un processus « service Manager» aura pour tâche:
  - Création +Enregistrement +Des-enregistrement des instances
  - Passer en revu toutes les instances pour vérifier lesquels sont encours d' exécution, les défaillantes etc. et MAJ le "Service Registry".

Avantages :

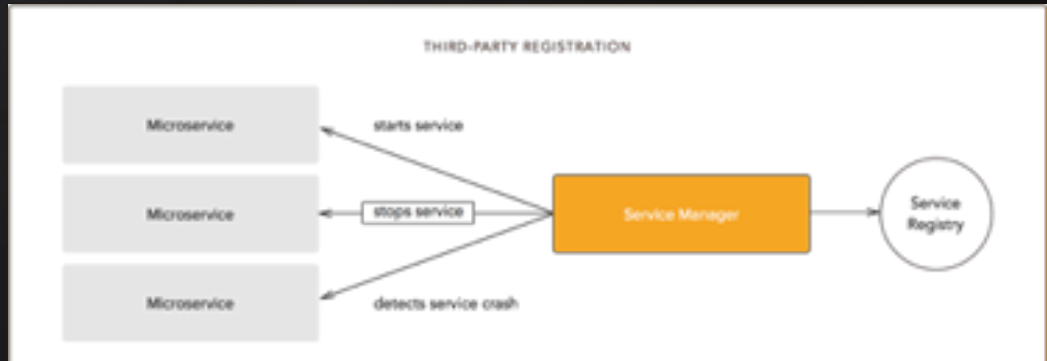
- Tâche centralisée non mise dans les MS.
- Possibilité de détecter et marquer les instances défaillantes.
- Possibilité de demander la création de nouvelles instances (Endpoint) si le service est surchargé.
- C'est la pratique utilisée actuellement.

- Exemples de «service manager» :

  Apache ZooKeeper

  Netflix Eureka

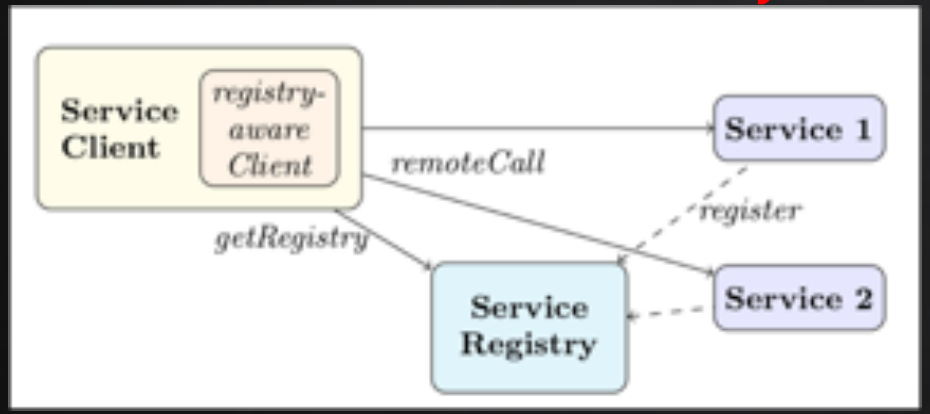
  Etc.



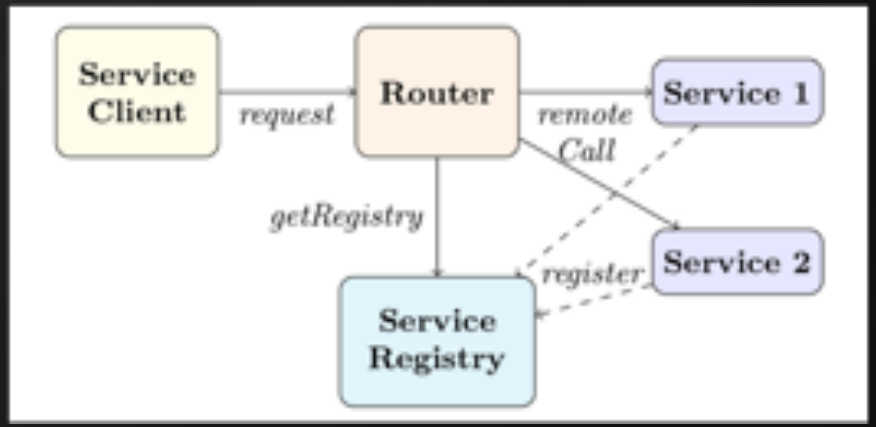
# Service discovery methods

- Le client a besoin de savoir où se trouve le service pour pouvoir l'invoquer et pouvoir récupérer les informations requise pour l'invocation.

## Client-side discovery




## Server-side discovery

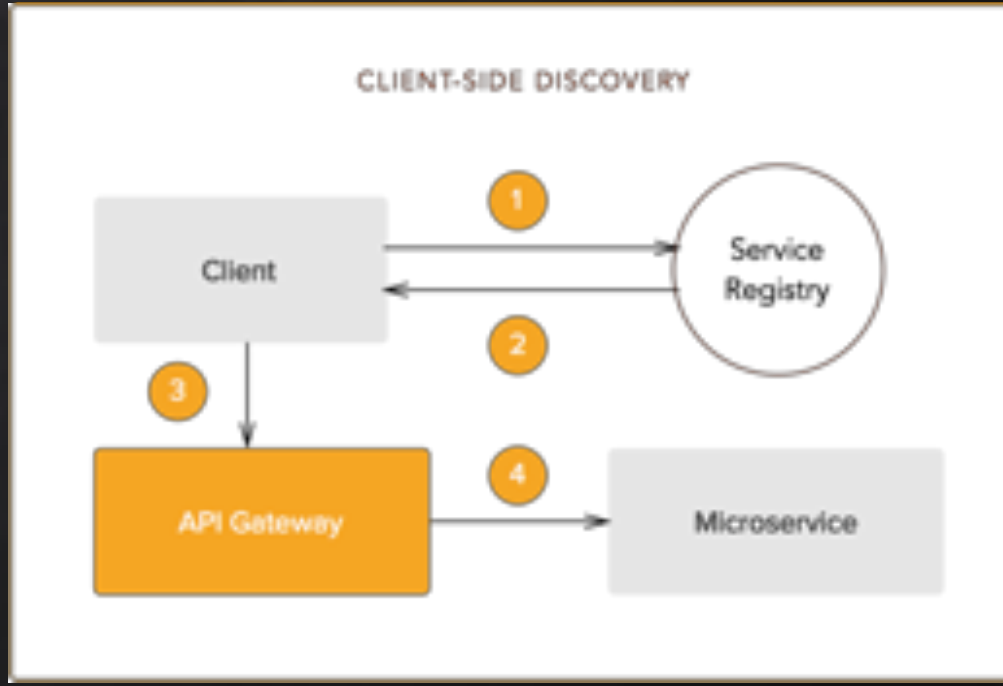


# 1 Client-side discovery

- Le client sait que le service a une location dynamique, alors il y a 2 appels émis : récupérer la référence + invoquer service.
- Le client doit connaitre le «Endpoint » du « service Registry »
- 2 possibilités :
  - Le service est situé derrière « API Gateway »
  - Pas de « API Gateway » : le client doit implémenter les mécanismes d'authentifcation, load-balancing etc.

Problème : nécessité d'implémenter « Service Discovery »par chaque partie cliente dans divers langages.

Exemple : Netflix OSS (Eureka+Ribbon)




1,2 Client récupère référence du à partir du « service Registry »

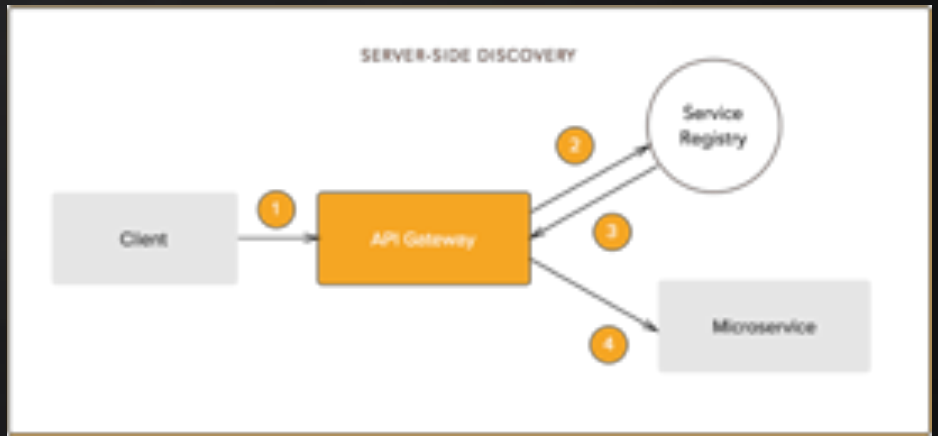
3,4 Client invoque service (via API Gateway)

# 2 Server-side discovery

- Déléguer la tache de "service Discovery" à un routeur externe « API Gateway ».
- Le client a juste besoin d'avoir le « Endpoint » statique du routeur en question.
- A la réception de l'invocation, le routeur se charge de la communiquer au service concerné après interrogation du « service Registry » pour récupérer une instance active du service.

Exemple :

AWS Elastic Load Balancer



# Required Pieces of an Effective Microservices Architecture

- Step1-Implement microservices
- Step2- Isolate microservices
- Step3-Communicate between/with microservices
- Step4-Deploy microservices
- Step5-Compose microservices
- Step6-Monitor microservices

# Step 1-Microservices implementation

- Microservices can be implemented in any programming language (java, PhP, JavaScript, python, etc.)
- Microservices need to deployed as services (REST/SOAP)
- The main technology choices are the way microservices communicate (synchronous, asynchronous, etc.) and which protocol they use (REST, messaging, SOAP)
- Any IDE of the developer's choice, like (Eclipse, IntelliJ, atom (open-source) or sublime text, and any of the version control systems from the client-server model (svn, perforce) or distributed model (Git, Visual Studio Team Service) can be used.
- Need for a microframework to automate repetitive tasks : build, deploy, publish, as service registry, service discovery etc.

# Examples of microframeworks:

- A **microframework** is a term used to refer to minimalistic web app frameworks. It is contrasted with full-stack frameworks.
  - Provided faster testing and deployment environment.
  - Used for repetitive tasks : Http request, response, service registry, etc.
- Examples of microframeworks :
  - Java : KumuluzEE, Spring boot, Dropwizard, Restlet,Spark , Akka Http, Javalin etc.
  - Node.JS : express.js, cote, Seneca, sail, etc.
  - Php : Lumen, silex, slip, slim, etc.

# Step2-Microservices isolation(Containerization)

- To effectively build microservices, containerization is essential.
- Containerization is an approach to software development in which an application or service, its dependencies, and its configuration are packaged together as a container image. The containerized application can be tested as a unit and deployed as a container image instance to the host operating system (OS).
- Just as shipping containers allow goods to be transported by ship, train, or truck regardless of the cargo inside, software containers act as a standard unit of software deployment that can contain different code and dependencies.
- Containerizing software this way enables developers and IT professionals to deploy them across environments with little or no modification.
- Containers also isolate applications from each other on a shared OS.
- Containerized applications run on top of a container host that in turn runs on the OS (Linux or Windows). Containers therefore have a significantly smaller footprint than virtual machine (VM) images.

# Step2-Microservices isolation(Containerization)

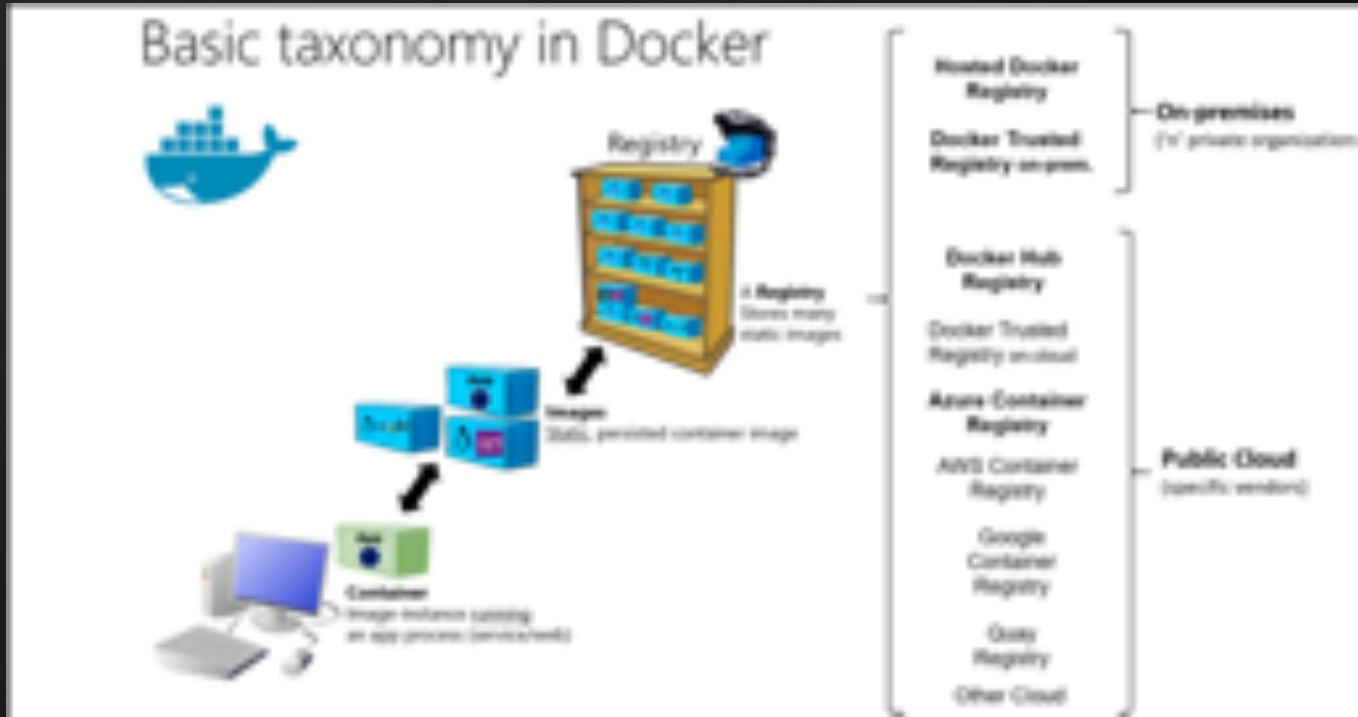
- Another benefit of containerization is scalability. You can scale out quickly by creating new containers for short-term tasks.
- From an application point of view, instantiating an image (creating a container) is similar to instantiating a process like a service or web app
- In short, containers offer the benefits of isolation, portability, agility, scalability, and control across the whole application lifecycle workflow.

# Step2-Microservices isolation(Containerization)

- When it comes to services, containers simplify development, testing, deployments, and running in production.
- because containers require far fewer resources (for example, they don't need a full OS), they're easy to deploy and they start fast. This allows you to have higher density, meaning that it allows you to run more services on the same hardware unit, thereby reducing costs.
- Examples of containers: Dockers, the most widely used.
  - When using Docker, a developer creates an app or service and packages it and its dependencies into a container image. An image is a static representation of the app or service and its configuration and dependencies.

# Step2-Microservices isolation(Containerization)

- To run the app or service, the app's image is instantiated to create a container, which will be running on the Docker host. Containers are initially tested in a development environment or PC.
- Developers should store images in a registry, which acts as a library of images and is needed when deploying to production orchestrators.
- Docker maintains a public registry via Docker Hub; other vendors provide registries for different collections of images, including Azure Container Registry. Alternatively, enterprises can have a private registry on-premises for their own Docker images.



# Containers vs. VM

- Virtual machines include the application, the required libraries or binaries, and a full guest operating system. Full virtualization requires more resources than containerization.
- Containers include the application and all its dependencies. However, they share the OS kernel with other containers, running as isolated processes in user space on the host operating system. (Except in Hyper-V containers, where each container runs inside of a special virtual machine per container.)

# Step3- communicate with microservices

- 2 modes of communication:
  - Synchronous communication (REST/SOAP)
  - Asynchronous communication :Services should communicate in a universal language. A Message Queue can be used as the glue that sends messages to all subscribed clients.

Examples of Solutions based on AMQP (Advanced Message Queuing Protocol) :

- RabbitMQ (Pivotal) => most widely used
- ActiveMQ, Apollo (Apache)
- Qpid (Apache)
- HornetQ (Jboss) ,AWSMQ, ZeroMQ, etc.

# Step4-Deploy microservices 1/2

- Each service must be deployed as a set of service instances for throughput and availability.
- So How are services are packaged and deployed?

1. Run multiple instances of different services on a host (Physical or Virtual machine).

   Ex : Deploy each service instance as a JVM process. For example, a Tomcat or Jetty instances per service instance.

2. Deploy each single service instance on its own host
3. Package the service as a VM image and deploy each service instance as a separate VM.

   Ex : Netflix packages each service as an EC2 AMI and deploys each service instance as an EC2 instance.

# Step4-Deploy microservices 2/2

4. Service instance per Container: Package the service as a container image and deploy each service instance as a container

   Ex : There are several Docker clustering frameworks including: Kubernetes Amazon EC2

5. Serverless deployment: Use a deployment infrastructure that hides any concept of servers. The infrastructure takes your service's code and runs it. You are charged for each request based on the resources consumed.

   Ex: AWS Lambda, Google Cloud Functions, Azure Functions

6. Service deployment platform : Use a deployment platform, which is automated infrastructure for application deployment. It provides a service abstraction, which is a named, set of highly available (e.g. load balanced) service instances.

Ex: Docker orchestration frameworks including Docker swarm mode and Kubernetes, Serverless platforms such as AWS Lambda, PaaS including Cloud Foundry and AWS Elastic Beanstalk

# Step5-Compose Microservices

- Microservices can be composed as:
- Orchestration or Choregraphy
- *Orchestration :*
  - *Orchestration* engine runs in the cloud.
  - The engine orchestrates microservices based process flows . It Allows creating complex process / business flows in which individual task is implemented by a microservice.

- Example of orchestration engine: Conductor (Netflix)

# Step6-Monitor microservices

- How to understand the behavior of an application and troubleshoot problems?

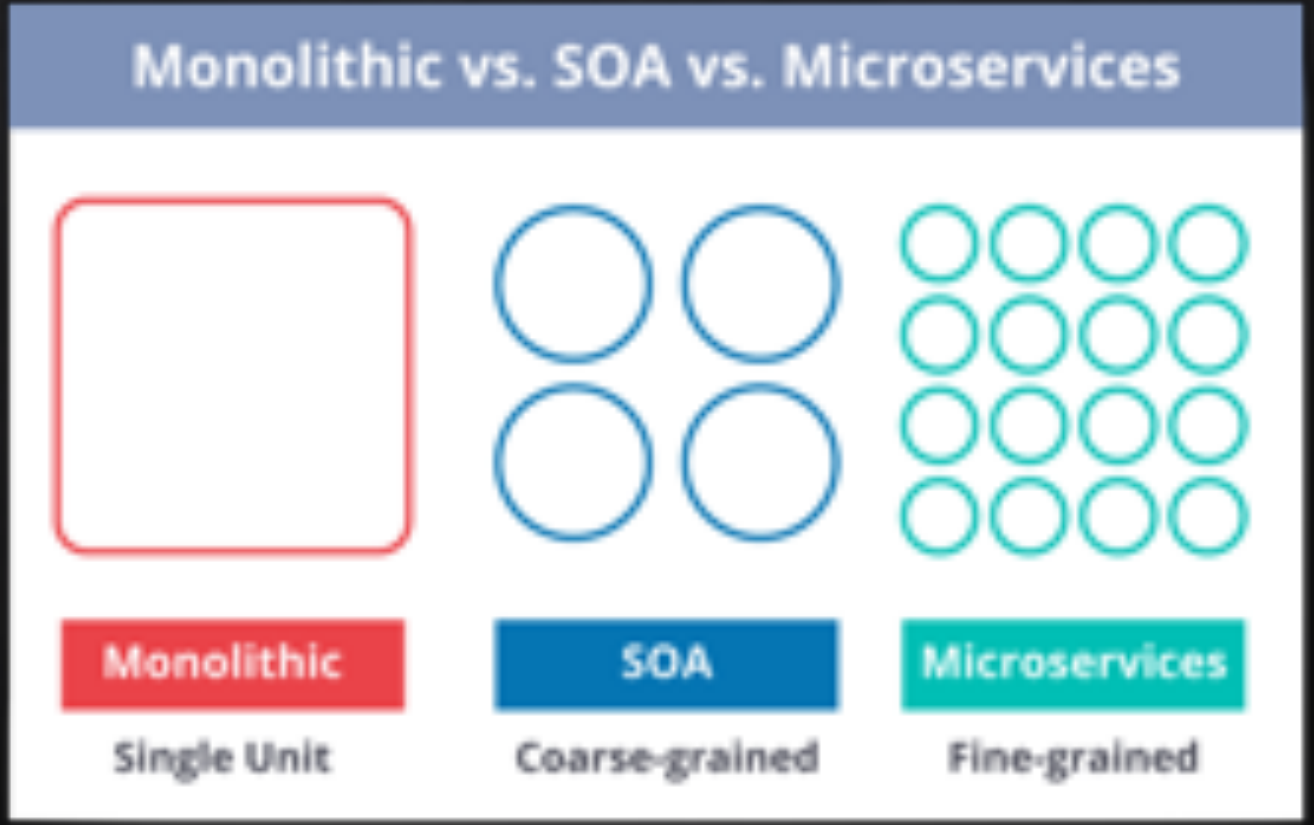
1. Instrument a service to gather statistics about individual operations.

   Ex: Java Metrics Library, Prometheus client librarie

2. Aggregate metrics in centralized metrics service, which provides reporting and alerting.

   Ex: Prometheus, AWS Cloud Watch

# MSA vs SOA


Monolithic vs. SOA vs. Microservices
Monolithic
SOA
Microservices
Single Unit
Coarse-grained
Fine-grained

# SOA vs MSA

- With SOA, the intent is a layered architecture of co-operating services where SOA focuses on describing the organization and coordination of the services.

- With MSA, the intent is to describe the nature of the services themselves and not quite so much the organization and coordination of them.

# MSA vs SOA

- SOA was designed with an ambition to solve very complex enterprise architecture problems, with the goal of facilitating a high level of reusability.

- In contrast, MSA was embraced by companies attempting to scale a single web property to web scale levels, enable continuous evolution, make engineering teams more efficient, and avoid technology lock-in.

# MSA vs SOA

- The focus of SOA is on reusability and discovery,
- The focus of MSA is on replacing a single monolithic application with a system that is easier to manage, scale and incrementally evolve.

# MSA vs SOA

- The key difference between the two approaches lies in granularity:
- Every MS in MSA is a unique function.
- A service in SOA can be a whole business process.

# MSA vs SOA : message passing

- Both Services and MS communicate via message passing.
- Even if services in SOA applications communicate via message passing, differently from MS, the internal components of each application are all part of a single executable artifact, called a monolith.

# MSA vs SOA: objective

- In SOA, services are used as an overlay meant to integrate and coordinate autonomous information systems.
  - This coordination is obtained via communications, which operate using standard protocols.
  - *It is more about interoperability between different information systems.*
- Microservices explore a different direction, i.e., that of using services as the **inner components** of an information system.
  - *It is more about re-engineering the inner components of an information system.*

# SOA vs MSA

- Although in both cases we are indeed talking about a set of services, the ambition of these services is different :
  - SOA attempts to put services forward to anybody who wants to use them.
  - Microservices, alternatively, are created with a much more focused and limited goal in mind, which is acting as a part of a single distributed system.
  - This distributed system is often created by breaking down a large monolithic application, and the intent is that a collection of microservices continue to work together as a single application. Typically, there is no ambition to serve multiple systems at the same time.
  - Unlike with SOA, microservices often exist implicitly. They are not discovered at run time, and do not require mediation. They are well known to the consumers, and therefore do not require service description. This does not imply that some kind of discovery is never present.

# Any questions?