



# Microservizi

Design pattern – Application modernization

Presented by

*Giorgio Dramis*

IBM Client Innovation Center - Bari




Dicembre 2023

IBM Client Innovation Center  
**Italy**

# Agenda

-  Riepilogo lezione precedente
-  Design Pattern – Parte 1
-  Demo time ed informazioni utili
-  Esercizi, domande e risposte

# Agenda

-  **Riepilogo lezione precedente**
-  **Design Pattern – Parte 1**
-  **Demo time ed informazioni utili**
-  **Esercizi, domande e risposte**

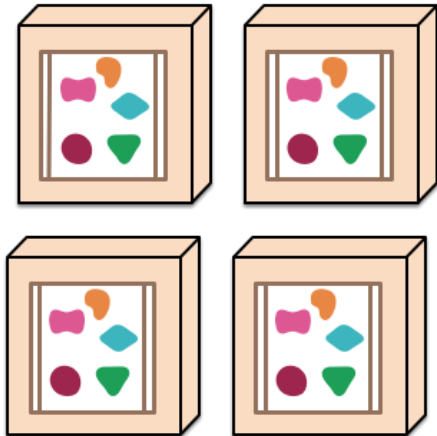
# Microservizi vs Monolite

L'architettura a microservizi si contrappone all'architettura monolitica.

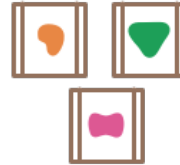
*A monolithic application puts all its functionality into a single process...*



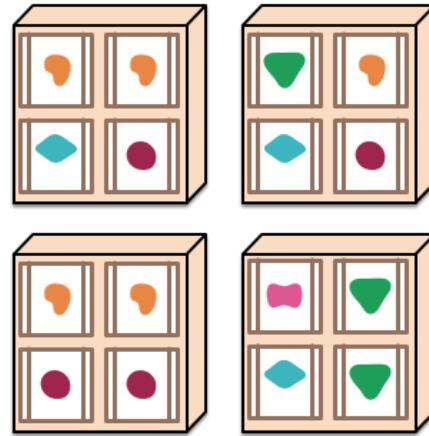
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*



- Un'applicazione monolitica inserisce tutte le sue funzionalità in un singolo processo e viene scalata replicando il monolite su più server.
- Un'architettura a microservizi inserisce ogni singola funzionalità in un servizio separato e viene scalata distribuendo questi servizi su vari server, replicandoli al bisogno.

# Microservizi: caratteristiche

- **Uno specifico obiettivo di business**, per garantire controllo delle scope, semplicità di monitoraggio ed automatizzazione delle fasi di test. Ciò fa sì che il cambiamento del singolo servizio non abbia impatto sugli altri.
- **Specializzato**, ossia progettato per una funzionalità o sulla risoluzione di un problema specifico. Se, nel tempo, si inserisce del codice aggiuntivo a un servizio rendendolo più complesso, il servizio può essere scomposto in servizi più piccoli.
- **Di dimensioni ridotte** per essere implementato da un piccolo team e per essere facilmente testato. E' bene precisare che man mano che la dimensione di un servizio decresce, aumentano i benefici relativi all'indipendenza tra le parti, ma cresce anche la complessità di gestire un numero elevato di parti.
- **Autonomo**, ossia sviluppato, distribuito, eseguito e ridimensionato senza influenzare il funzionamento degli altri. I servizi non devono condividere alcun codice o implementazione con gli altri.
- **Indipendenza nelle fasi di development, building e deploying**, in modo che resti svincolato dagli altri.
- **Logica stateless/sessionless**, in modo da massimizzare la scalabilità orizzontale.

# Microservizi: benefici introdotti

- **Scalabilità:** a differenza di un'architettura monolitica, un'architettura a microservizi permette di far scalare solo la parte dell'applicativo che ne ha necessità.
- **Resilienza:** suddividendo l'architettura in microservizi possiamo garantire che il malfunzionamento di una singola componente non pregiudichi il funzionamento dell'intero sistema (Circuit Breaker).
- **Velocità di deploy:** l'approccio a microservizi permette di poter rilasciare solo il servizio che è variato senza andare ad impattare l'intero dell'applicativo.
- **Migliore organizzazione del team:** l'approccio a microservizi permette di creare team cross-funzionali interscambiabili.
- **Poliglottismo:** possibilità di differenti implementazione utilizzando diverse tecnologie e piattaforme a seconda delle esigenze.
- **Ridurre il tempo di rilascio:** componenti indipendenti permettono la realizzazione di nuove funzionalità da rilasciare più velocemente in produzione fornendo una maggiore flessibilità per soluzioni pilota e prototipazione, riducendo il tempo di rilascio.

# Agenda

- 1 Riepilogo lezione precedente
- 2 **Design Pattern – Parte 1**
- 3 Demo time ed informazioni utili
- 4 Esercizi, domande e risposte

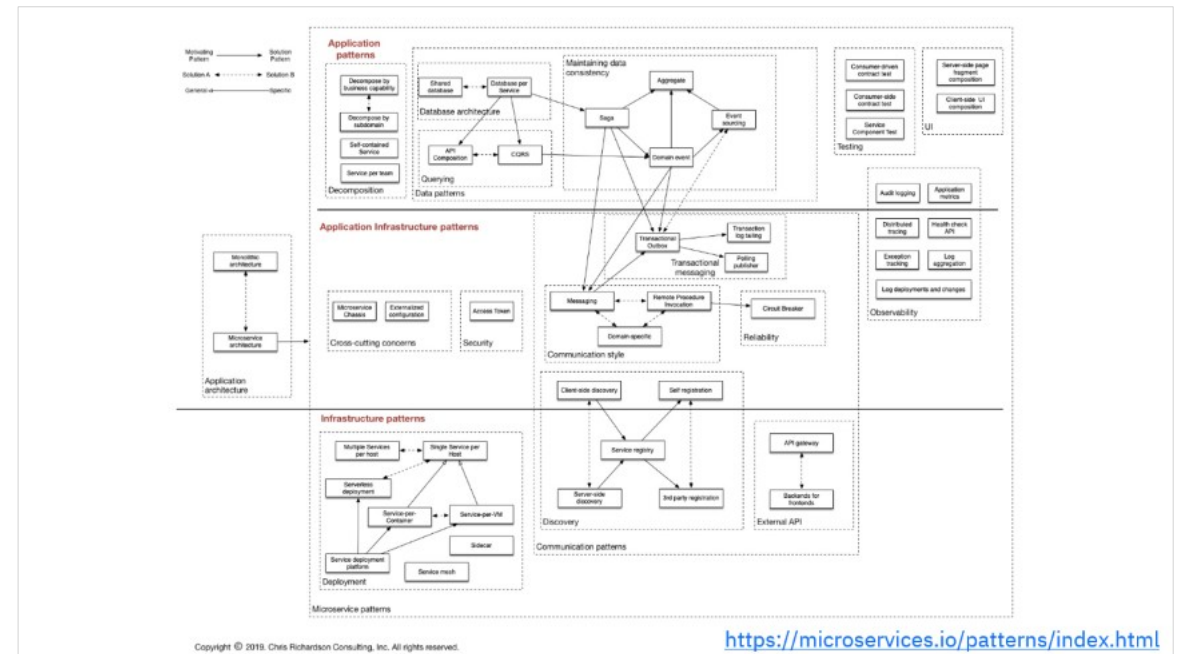
# Design Pattern

## Definizione

I **Design Patterns** rappresentano soluzioni di progettazione generiche applicabili ad alcuni problemi ricorrenti, all'interno di contesti eterogenei.

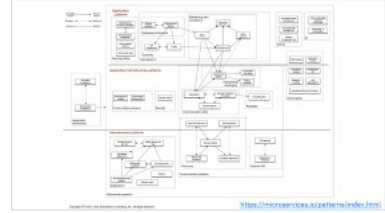
A fronte di specifiche esigenze di sviluppo, nella creazione dei microservizi si potranno applicare diverse famiglie di pattern.

L'immagine a lato raccoglie i principali pattern applicabili ad architetture a microservizi.

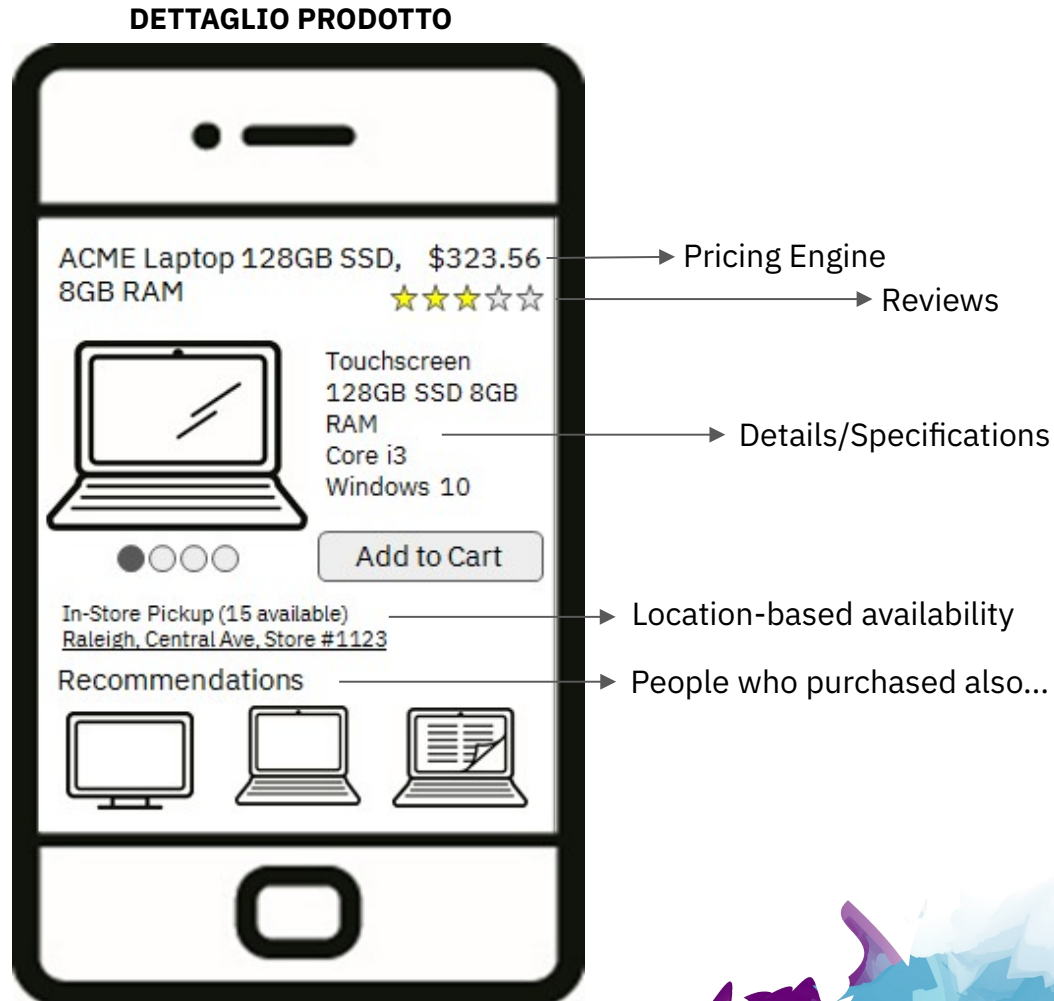




# Applicazione di riferimento

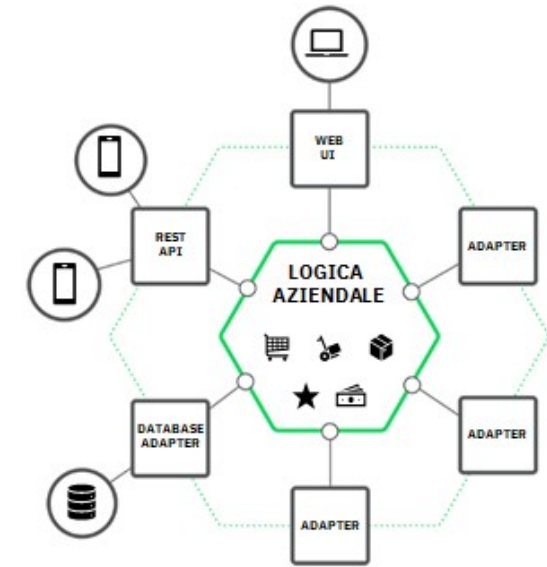
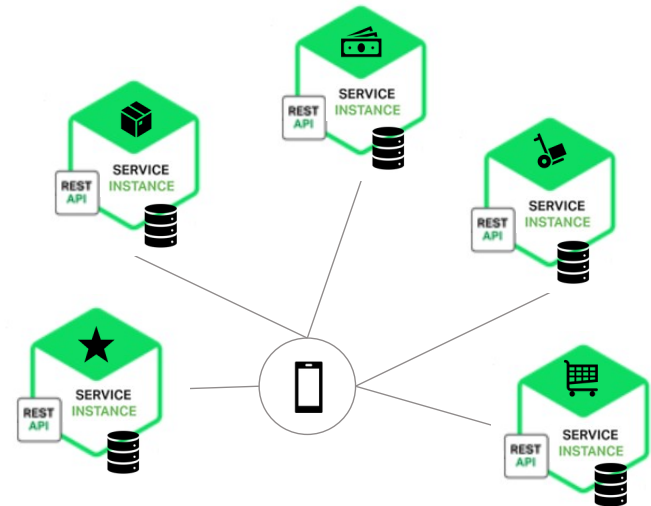


Immaginiamo di voler implementare un nuovo **client mobile** per un sistema di shopping online.



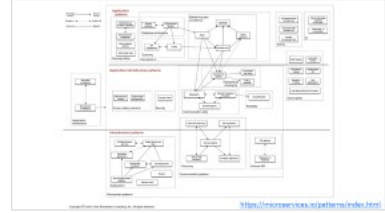
# Client-to-Backend Communication

Quando si utilizza **un'architettura monolitica**, un client recupera le informazioni effettuando una singola chiamata REST all'applicazione. Un bilanciamento del carico indirizza la richiesta a una delle N istanze identiche dell'applicazione. L'applicazione effettua delle elaborazioni interne e restituisce una risposta al client.



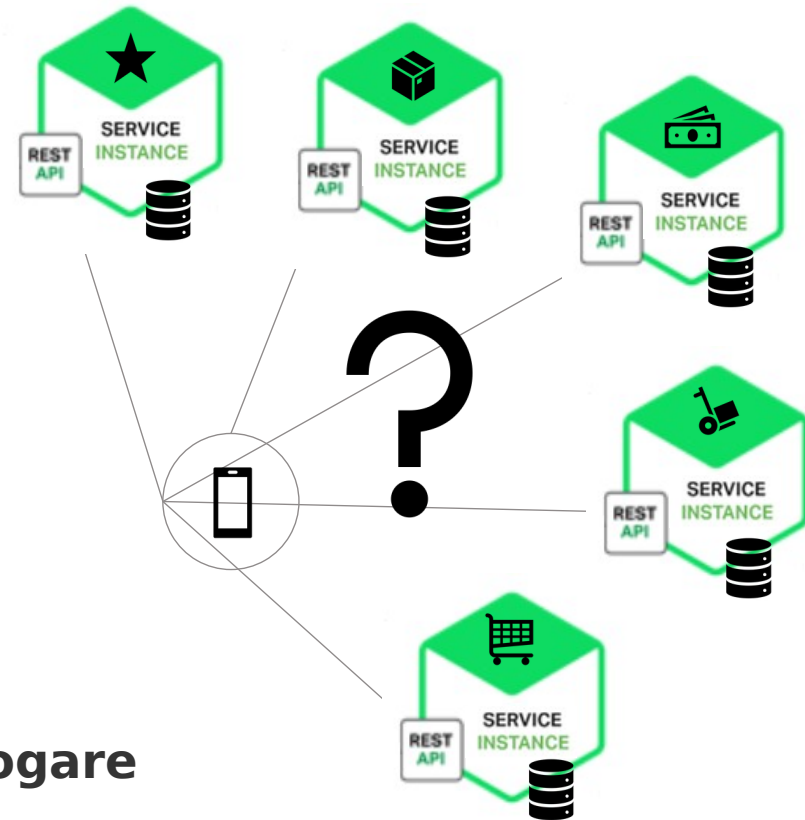
Al contrario, quando si utilizza **l'architettura dei microservizi** i dati sono di proprietà di più servizi. In teoria, un client potrebbe fare richieste direttamente a ciascuno dei microservizi ed effettuare un mashup dei dati. Ogni microservizio avrebbe un endpoint pubblico.

# Client-to-Microservice Communication



Si possono elencare numerosi **svantaggi** nell'adozione di tale approccio:

- Mancanza di corrispondenza tra le esigenze del client e le API a grana fine esposte da ciascuno dei microservizi.
- Possibile utilizzo di protocolli non compatibili con il web (AMQP, protocolli binari, ..)
- Difficile il refactoring dei microservizi (modifica del partizionamento del sistema in servizi, ...)

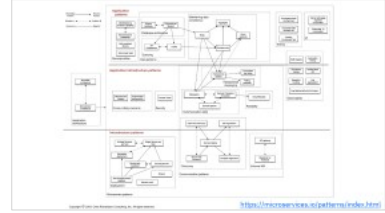
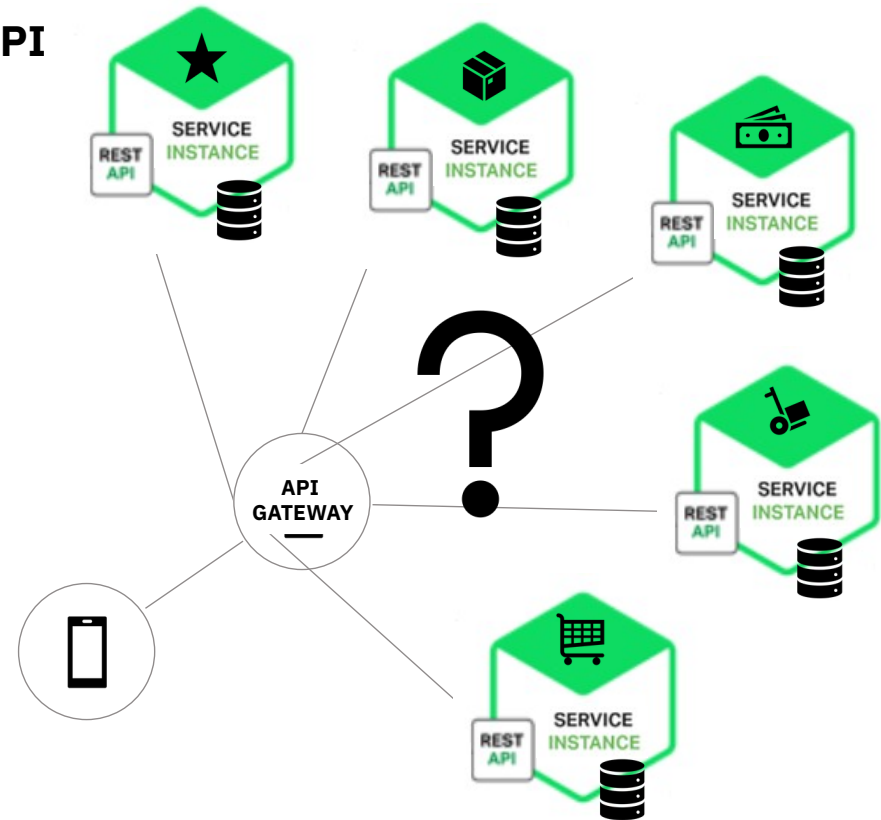


**Per questi problemi, non ha senso per i client interrogare direttamente i microservizi.**

# API Gateway

Un approccio diverso è che un client faccia una singola richiesta a un **API gateway**, un servizio che funge da punto di accesso singolo per le richieste in un'applicazione.

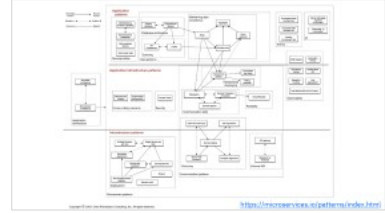
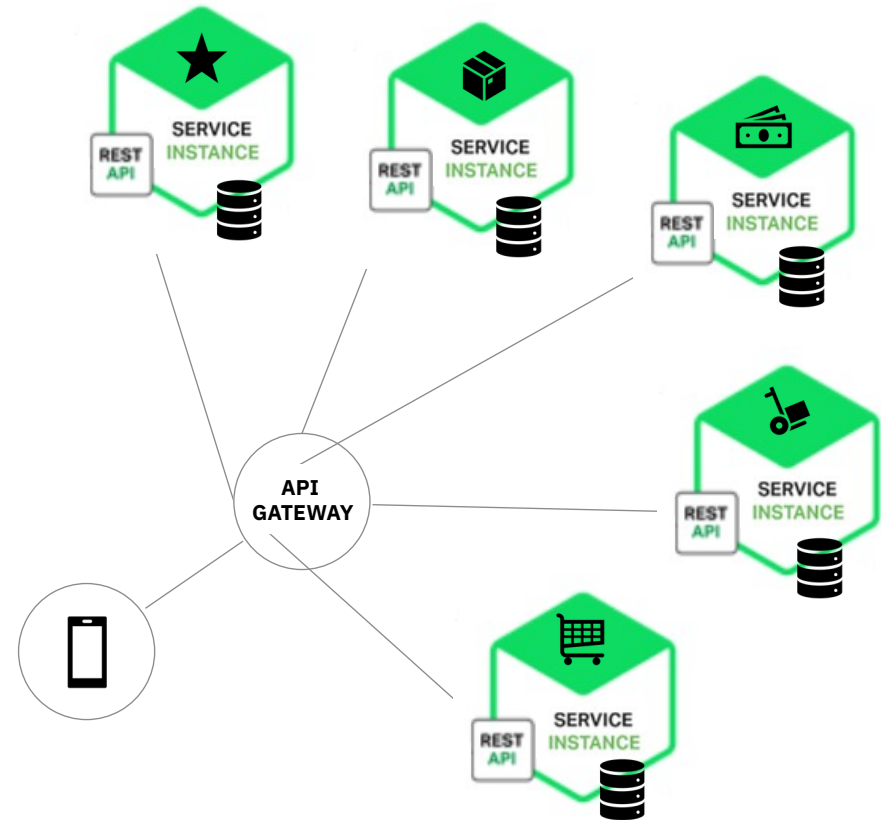
- Incapsula l'architettura interna dell'applicazione e fornisce delle API ai suoi client.
- È responsabile del routing delle richieste: tutte le richieste dei client passano prima attraverso questo componente che le indirizza verso il servizio appropriato.
- Gestisce spesso una richiesta richiamando più microservizi e aggregando i risultati.



# API Gateway

Un grande **vantaggio** dell'utilizzo di un gateway API è che incapsula la struttura interna dell'applicazione. Invece di dover invocare X servizi specifici, i client interrogano semplicemente il gateway.

- Ciò riduce il numero di richieste tra il client e l'applicazione.
- Semplifica il codice lato client spostando la logica per chiamare più servizi dal client al gateway.
- Funzionalità edge: è possibile effettuare una serie di verifiche (autenticazione delle richieste, autorizzazione, analisi, ...) che l'invocazione Client-to-Microservice non permette in modalità agile.



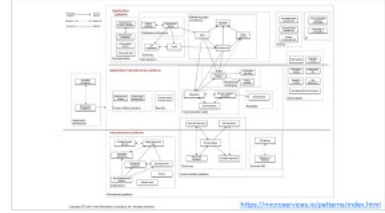
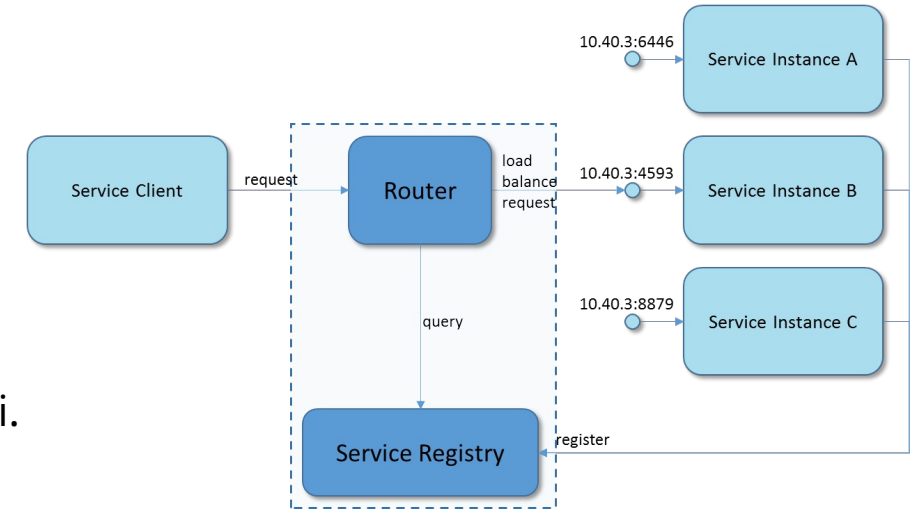
# API Gateway

## PUNTO DI ATTENZIONE: INDIVIDUAZIONE DEI SERVIZI

Il gateway API deve conoscere la posizione (indirizzo IP e porta) di ciascun servizio con cui comunica.

- In **un'applicazione monolitica**, è possibile cablare con certezza le posizioni.
- In una **applicazione a microservizi** si tratta di un problema non banale.  
I servizi applicativi hanno posizioni assegnate dinamicamente e l'insieme della istanze di uno specifico servizio cambiano in modo dinamico a causa del ridimensionamento automatico.

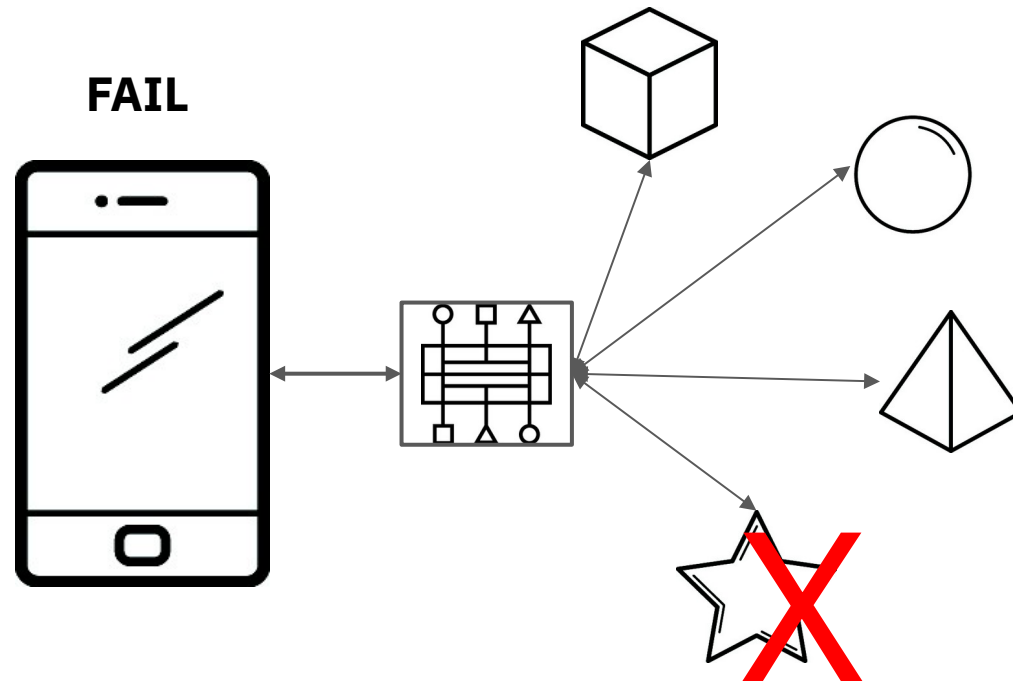
Di conseguenza, l'API gateway, deve utilizzare un **meccanismo di individuazione del servizio**.



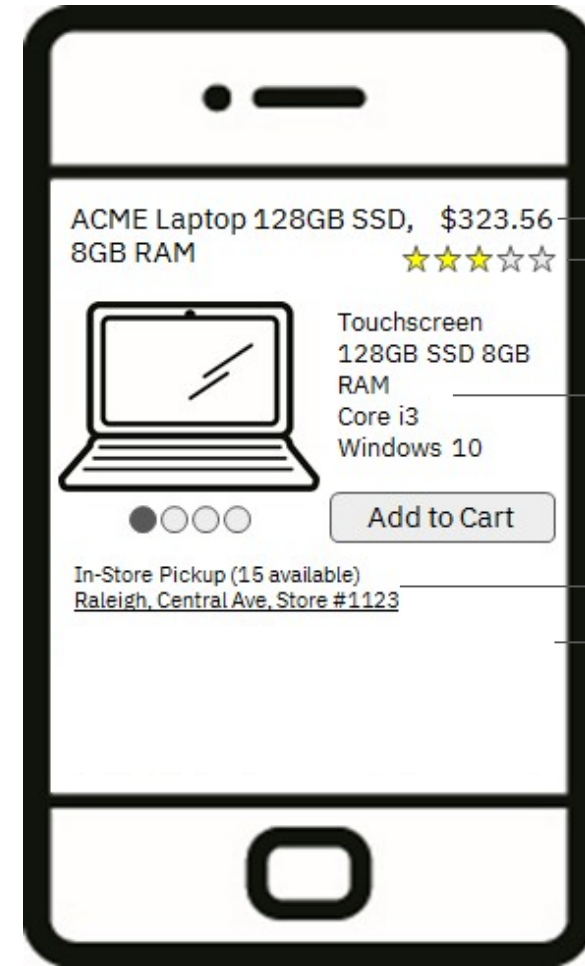


# API Gateway

## PUNTO DI ATTENZIONE: GESTIONE DEGLI ERRORI PARZIALI



Il servizio potrebbe essere inattivo a causa di un errore



Pricing Engine

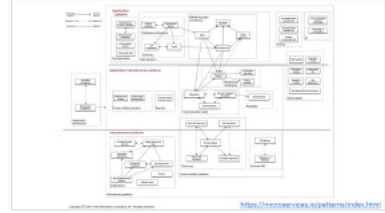
Reviews

Details/Specifications

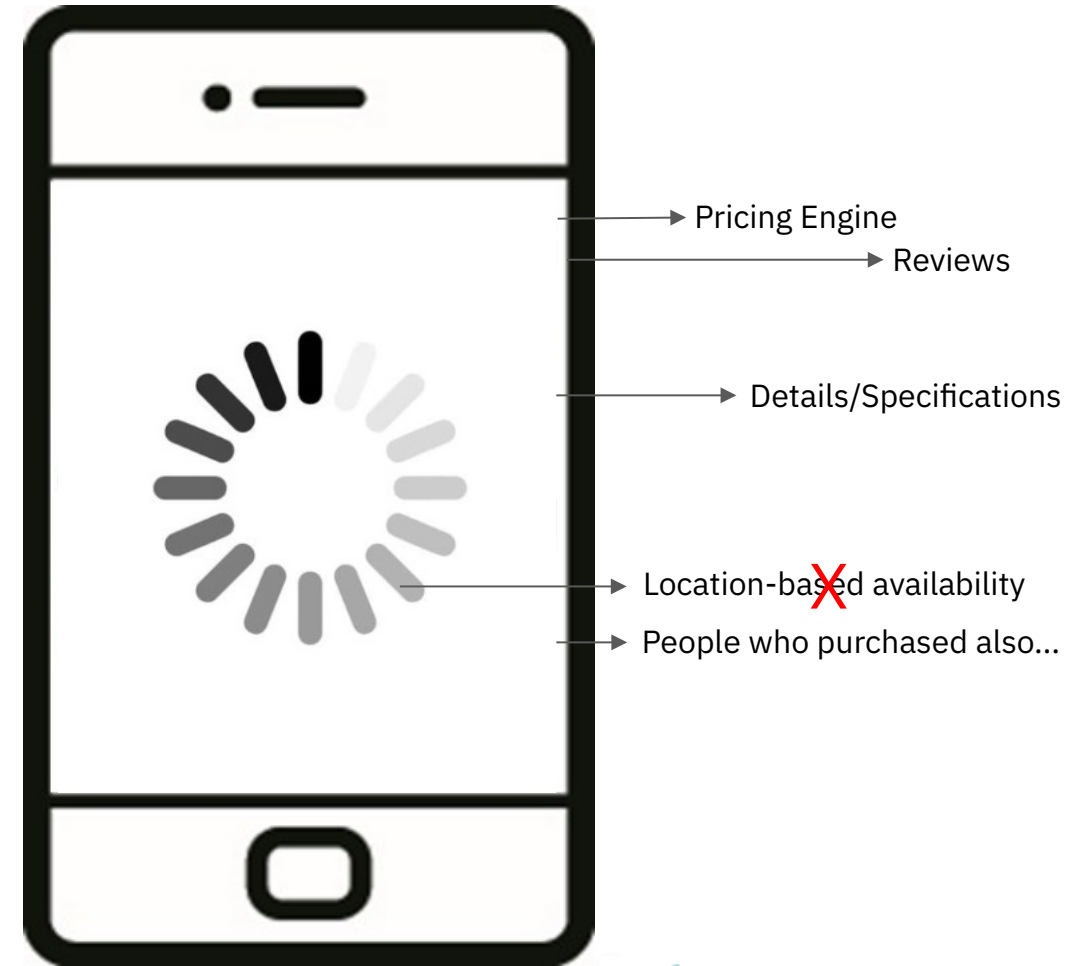
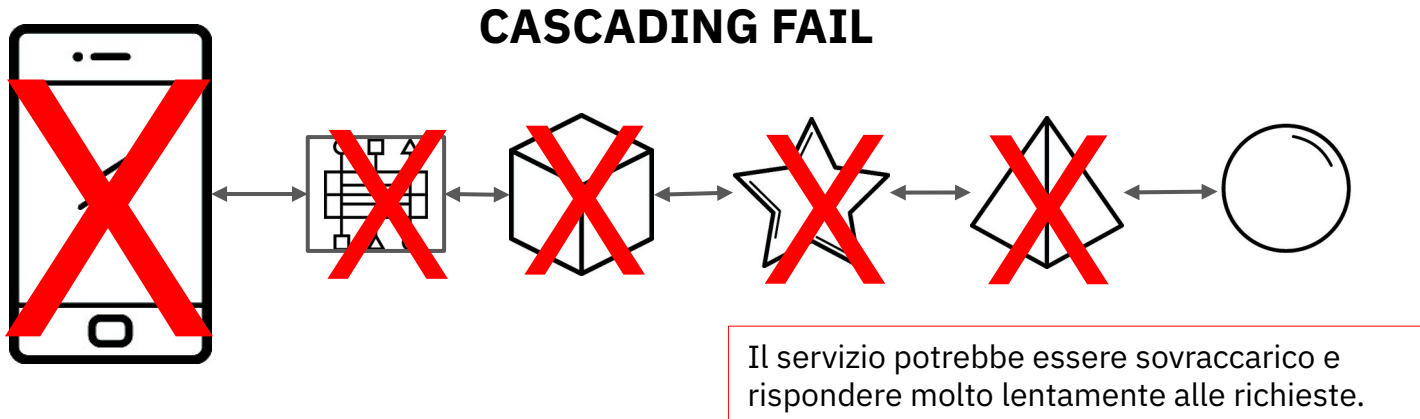
Location-based availability

People who ~~p~~urchased also...

# API Gateway

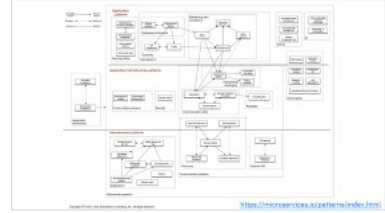


## PUNTO DI ATTENZIONE: GESTIONE DEGLI ERRORI PARZIALI





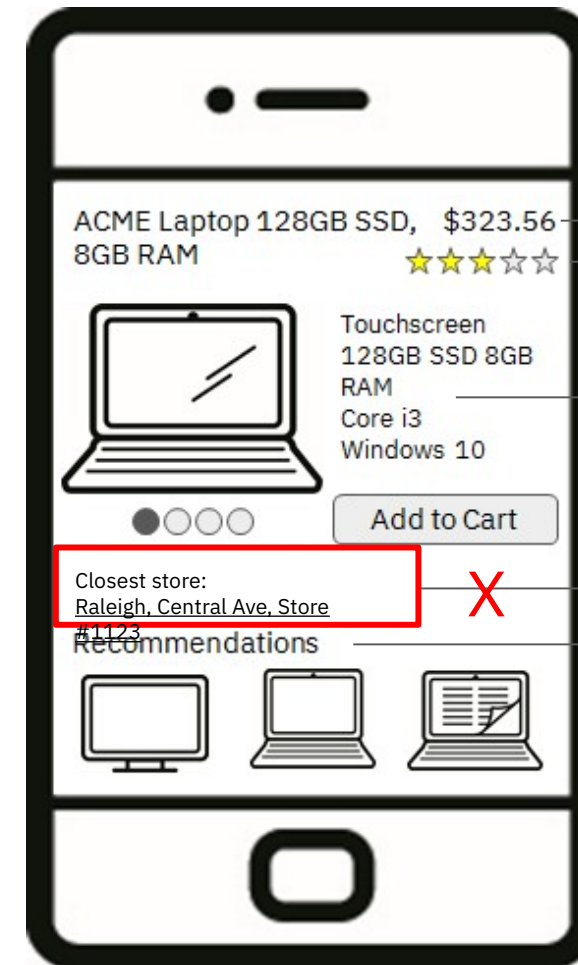
# API Gateway



## PUNTO DI ATTENZIONE: GESTIONE DEGLI ERRORI PARZIALI

L'API gateway non dovrebbe mai bloccarsi indefinitamente in attesa di un servizio, garantendo che eventuali guasti non influiscano sull'esperienza dell'utente. Le modalità con cui si gestisce la problematica dipendono dallo scenario specifico e dal servizio che non funziona:

- si potrebbe non restituire una determinata informazione o restituire dei dati statici cablati (prodotti consigliati)
- si potrebbero restituire dati memorizzati in cache (prezzi dei prodotti)
- si potrebbe restituire un errore (informazioni sul prodotto)



## FALLBACK

→ Pricing Engine

→ Reviews

→ Details/Specifications

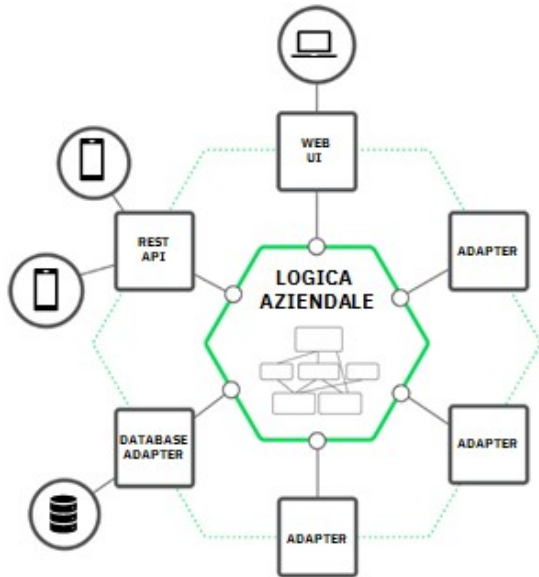
→ Location-based availability

→ People who purchased also...

# API Gateway

## PUNTO DI ATTENZIONE: INTEROPERABILITÀ

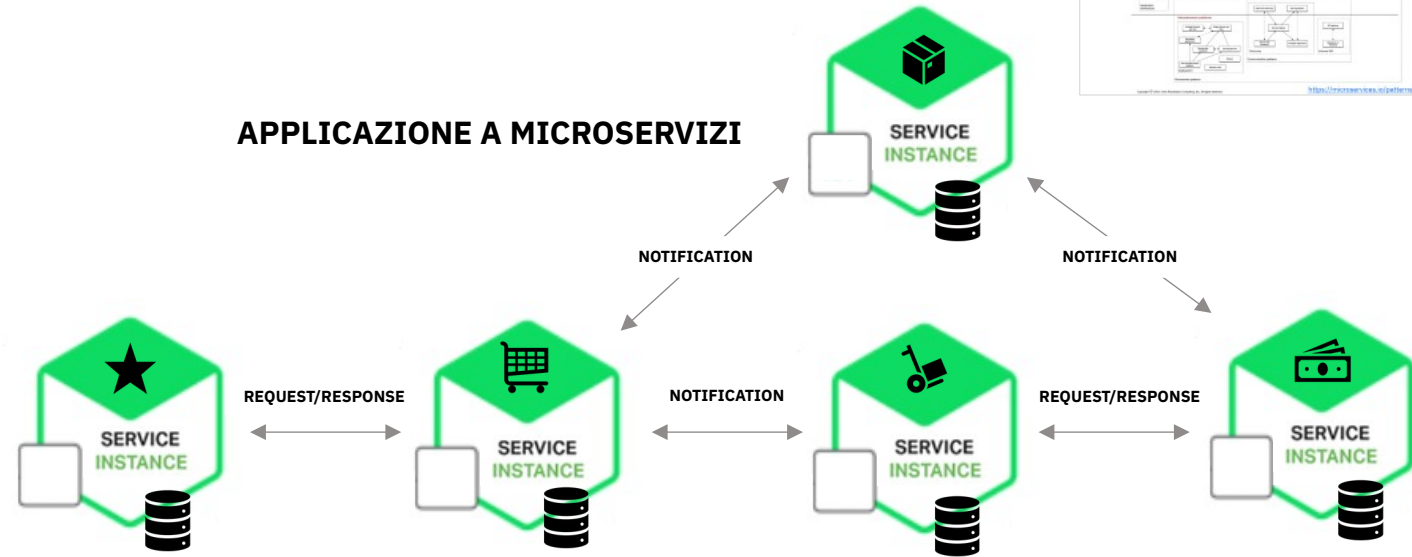
I componenti si invocano reciprocamente tramite chiamate a metodi o funzioni (language-level).



APPLICAZIONE MONOLITICA



## APPLICAZIONE A MICROSERVIZI

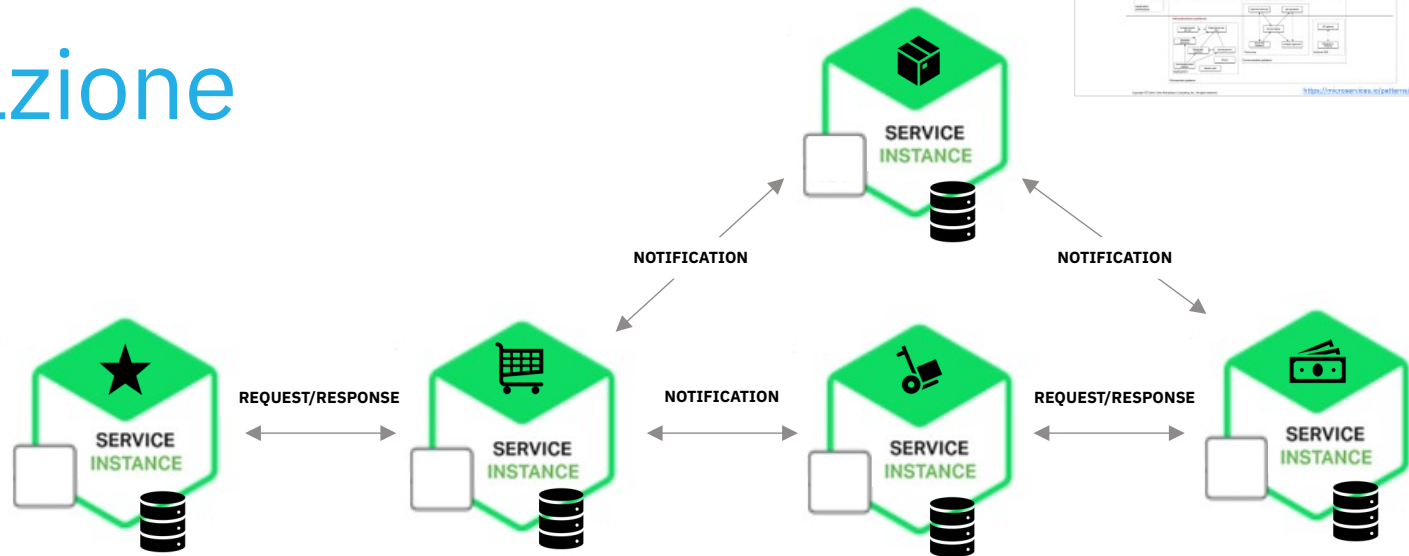


Un'applicazione a microservizi è un sistema distribuito in esecuzione su più macchine. Ogni servizio è in genere un processo separato. Di conseguenza, i servizi possono interagire utilizzando diversi **meccanismi di comunicazione** tra processi, che l'API Gateway dovrà supportare.

# Meccanismi di comunicazione

Possono essere classificate in due diverse dimensioni:

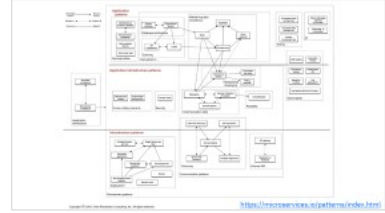
- **One-to-One:** ogni richiesta di un client viene elaborata esattamente da un'istanza di servizio.
  - **One-to-Many:** ogni richiesta di un client viene elaborata da più istanze di servizio.
- 
- **Synchronous:** il client si aspetta una risposta tempestiva dal servizio e potrebbe persino bloccarsi durante l'attesa.
  - **Asynchronous:** il client non si blocca durante l'attesa di una risposta e la risposta, se presente, non viene necessariamente inviata immediatamente.



	One-to-One	One-to-Many
Synchronous	Request/response	
Asynchronous	Notification	Publish/subscribe

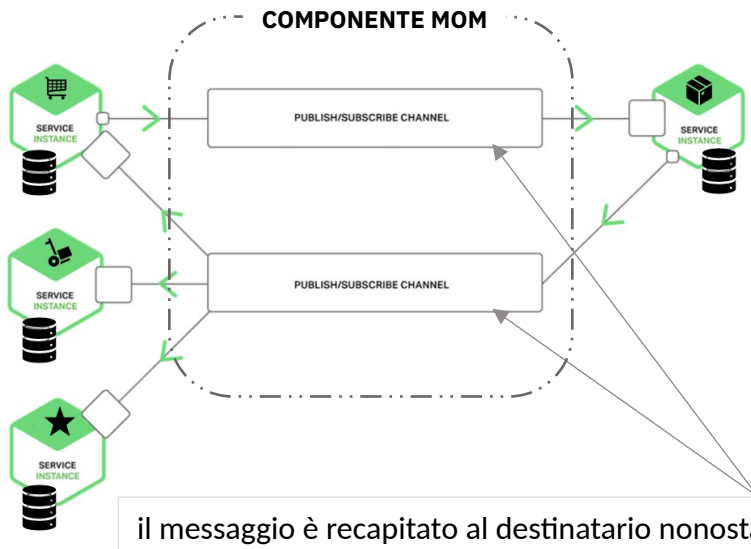
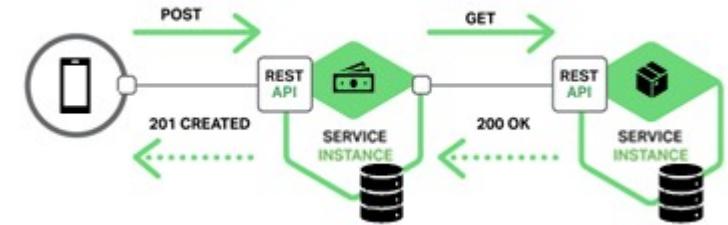
Ogni servizio in genere utilizza una combinazione di queste modalità

# Meccanismi di comunicazione



## Synchronous, Request/Response

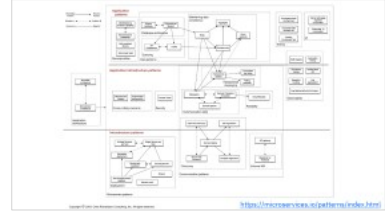
Un client invia una richiesta a un servizio. Il servizio elabora la richiesta e invia una risposta. Il client che effettua la richiesta si blocca durante l'attesa di una risposta. Si presume, quindi, che la risposta arrivi in modo tempestivo.



**Asynchronous, Message-Based Communication**

I servizi comunicano scambiandosi messaggi. Un client invia una richiesta a un servizio inviandogli un messaggio. Se si prevede che il servizio risponda, lo fa inviando un messaggio separato al client. Poiché la comunicazione è asincrona, il client si non blocca in attesa di una risposta.

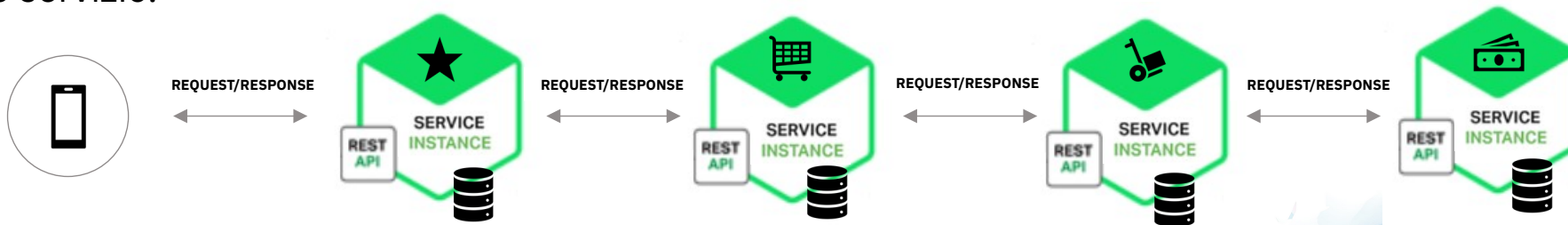
# Gestione degli errori parziali



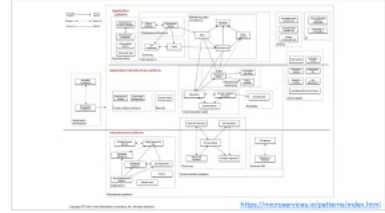
Ogni volta che un servizio invia una richiesta sincrona a un altro servizio, esiste un rischio sempre presente di guasto parziale. Un servizio, per esempio, potrebbe non essere in grado di rispondere in modo tempestivo alla richiesta di un client. Poiché il client è bloccato in attesa di una risposta, il pericolo è che l'errore possa arrivare a cascata su altri client e causare un'interruzione.

Si rende necessario, quindi, utilizzare una combinazione dei seguenti meccanismi:

1. **Limitazione del numero di richieste in sospeso da un cliente a un servizio:** imporre un limite superiore al numero di richieste in sospeso che un cliente può effettuare a un determinato servizio.
2. **Time-out di rete:** non bloccare mai a tempo indeterminato e utilizzare sempre i time-out durante l'attesa di una risposta.
3. **Circuit Breaker**



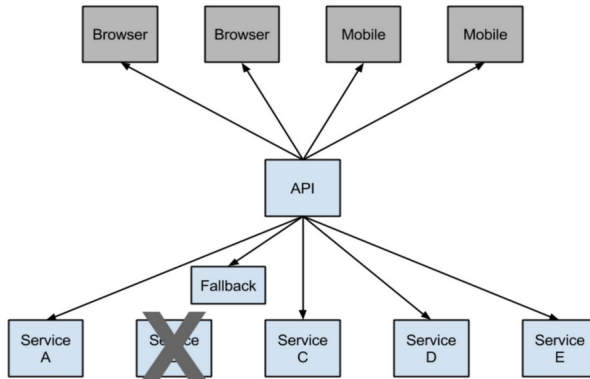
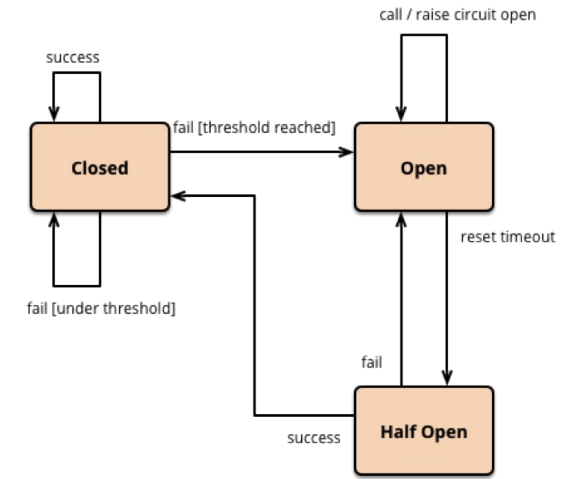
# Circuit Breaker



Si tiene traccia del numero di richieste riuscite e non riuscite e, se il tasso di errore supera una certa soglia, ulteriori tentativi falliscano immediatamente per un tempo concordato (time-out).

Allo scadere del time-out, si consentirà il passaggio di un numero limitato di richieste di test:

- Se tali richieste avranno esito positivo, il servizio riprenderà a funzionare normalmente.
- Altrimenti, se si verificherà nuovamente un errore, il periodo di time-out ricomincerà.

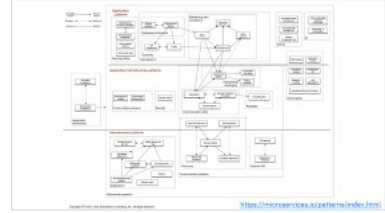


È inoltre necessario decidere come si dovrebbero recuperare informazioni da un servizio remoto che non risponde. Esistono due possibili opzioni:

- Un'opzione è che un servizio restituisca semplicemente un errore al suo client.
- In altri scenari, può essere utile restituire un valore di fallback, ad esempio un valore predefinito o una risposta memorizzata nella cache.

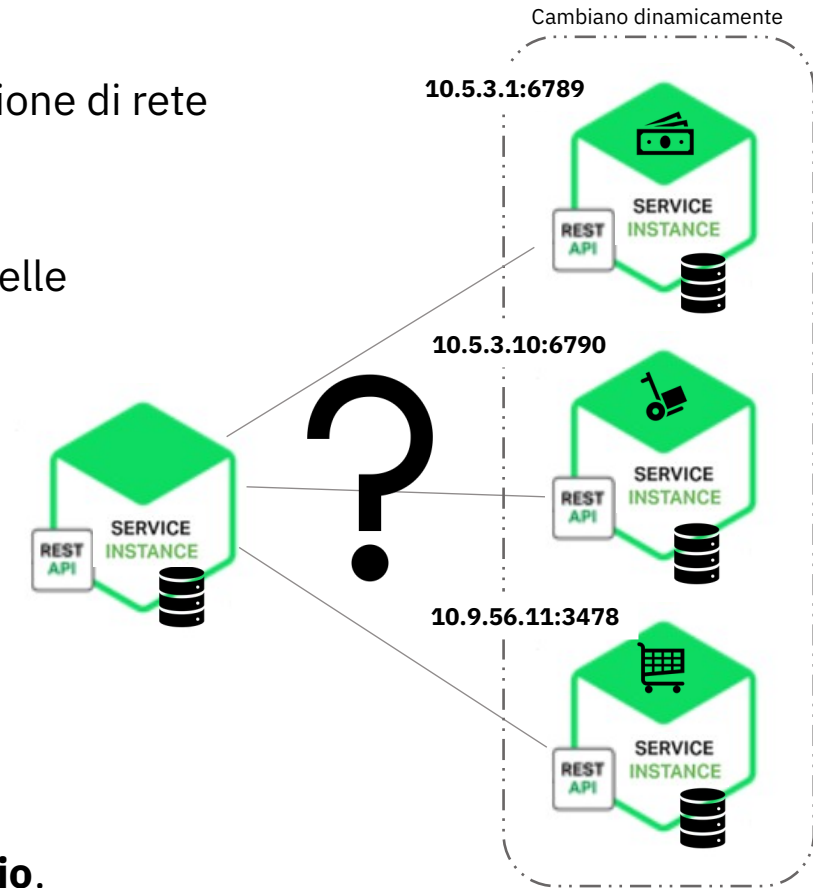


# Service Discovery



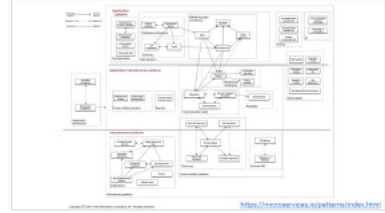
Per effettuare una richiesta ad un determinato servizi, il client deve conoscere la posizione di rete (indirizzo IP e porta) di un'istanza.

- In un'applicazione **monolitica** in esecuzione su hardware fisico, i percorsi di rete delle istanze di servizio sono relativamente statici.
- In una moderna applicazione a **microservizi**, tuttavia, questo è un problema molto più difficile da risolvere. Le istanze di servizio hanno posizioni di rete assegnate in modo dinamico. Inoltre, l'insieme di istanze di servizio cambia in modo dinamico a causa del ridimensionamento automatico, degli errori e degli aggiornamenti.



Di conseguenza, il client deve utilizzare un **meccanismo di individuazione del servizio**.

# Service Discovery

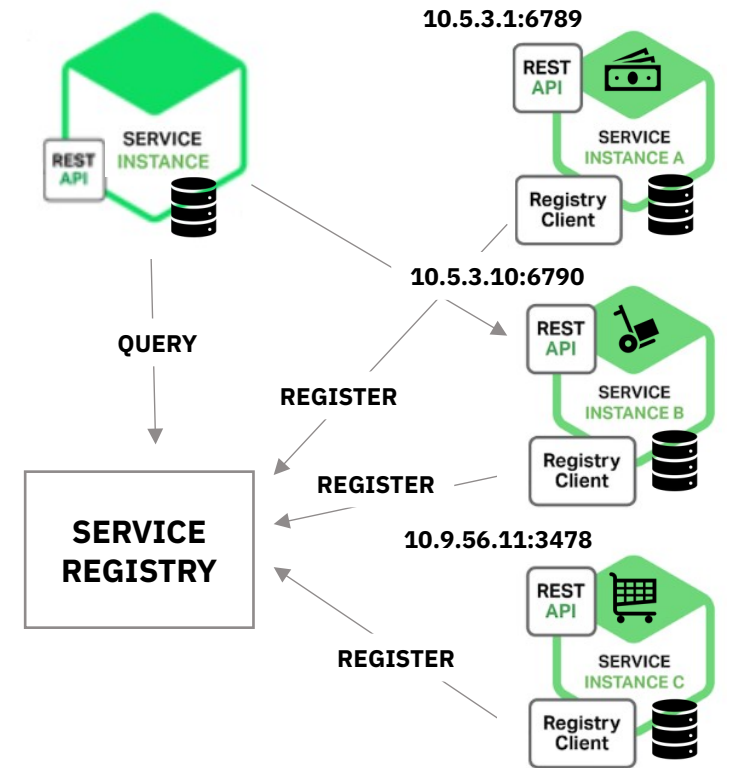


## CLIENT-SIDE DISCOVERY

Il client è responsabile della determinazione delle posizioni di rete delle istanze di servizio disponibili e del bilanciamento del carico.

1. Le istanze di servizio si registrano nel service registry, che è un database di istanze di servizio disponibili.
2. Il client effettua una richiesta al service registry e ricava tutte le istanze di servizio disponibili.
3. Il client utilizza un algoritmo di bilanciamento del carico per selezionare una delle istanze di servizio disponibili
4. Il client effettua una richiesta verso l'istanza selezionata

Si accoppia il client con il service registry. È necessario implementare la logica di individuazione del servizio lato client per ciascun linguaggio di programmazione e framework.



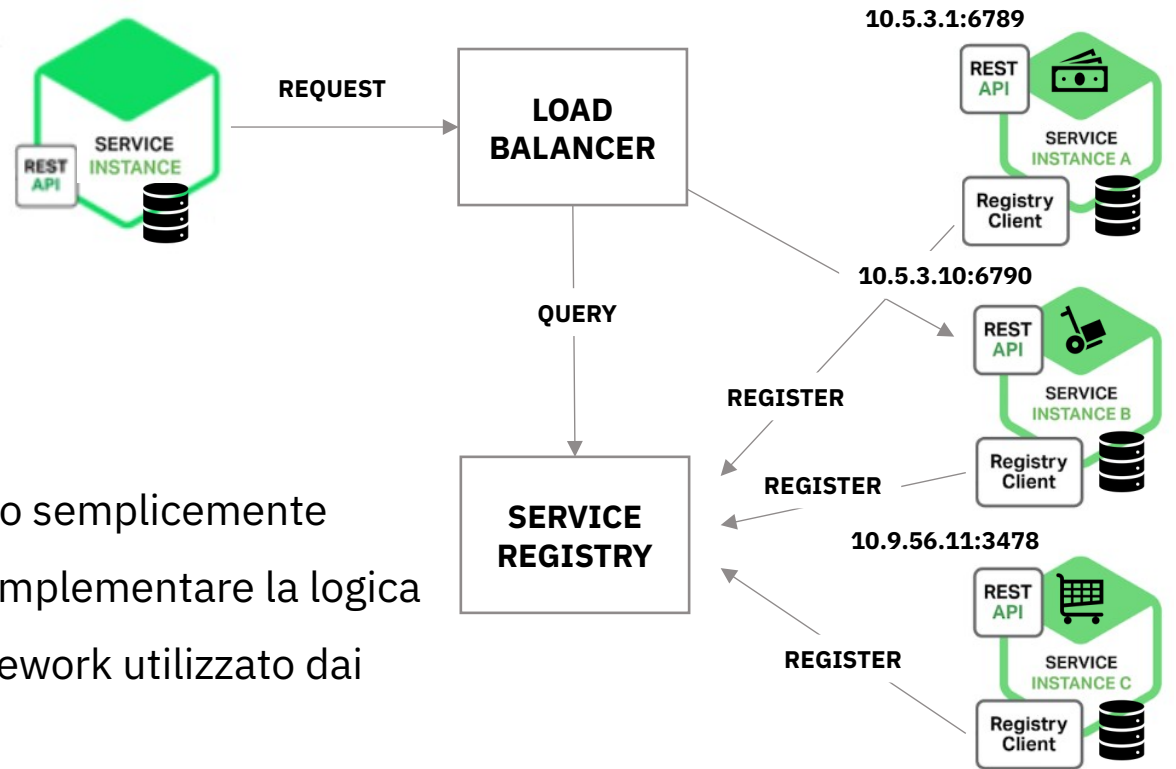


# Service Discovery

## SERVER-SIDE DISCOVERY

Il client effettua una richiesta a un servizio tramite un bilanciamento del carico. Il servizio di bilanciamento del carico interroga il service registry e indirizza ciascuna richiesta a un'istanza di servizio disponibile.

I dettagli del rilevamento vengono sottratti al client. I client fanno semplicemente richieste al bilanciamento del carico. Ciò elimina la necessità di implementare la logica di rilevamento per ciascun linguaggio di programmazione e framework utilizzato dai client che effettuano la richiesta.



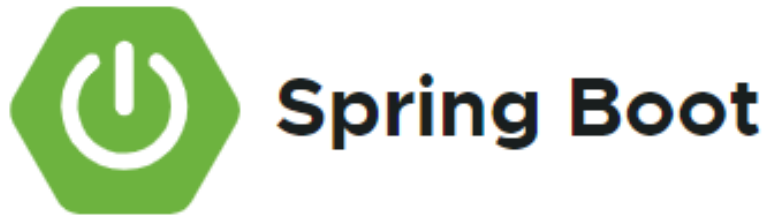
# Agenda

-  1 Riepilogo lezione precedente
-  2 Design Pattern – Parte 1
-  3 **Demo time ed informazioni utili**
-  4 Esercizi, domande e risposte

Demo time

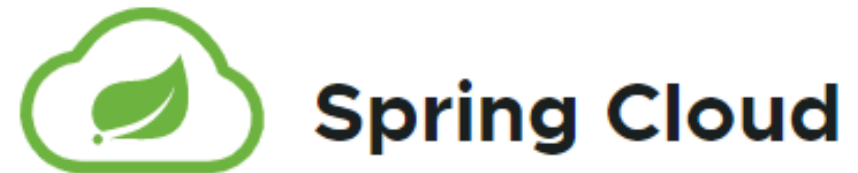
# Spring Cloud

Lo sviluppo dei singoli microservizi avverrà tramite l'utilizzo del framework **Spring Boot**, che permette di creare applicazioni stand-alone, occupandosi di tutta la parte di struttura e configurazione e focalizzando il lavoro dello sviluppatore nella creazione della logica di business. Inoltre, si avrà il supporto di **Spring Cloud**, suite di strumenti che aiutano lo sviluppatore a risolvere problematiche comuni nei sistemi distribuiti.



<https://spring.io/projects/spring-boot>

+



<https://spring.io/projects/spring-cloud>

Demo time

# Spring Cloud Netflix

Netflix Open Source Platform

**NETFLIX** | **OSS**

<https://github.com/netflix>

## Eureka

HOME    LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2016-08-19T07:30:13 +0200
Data center	default	Uptime	00:00
Lease expiration enabled	false	Renews threshold	1
Renews (last min)	0		

DS Replicas

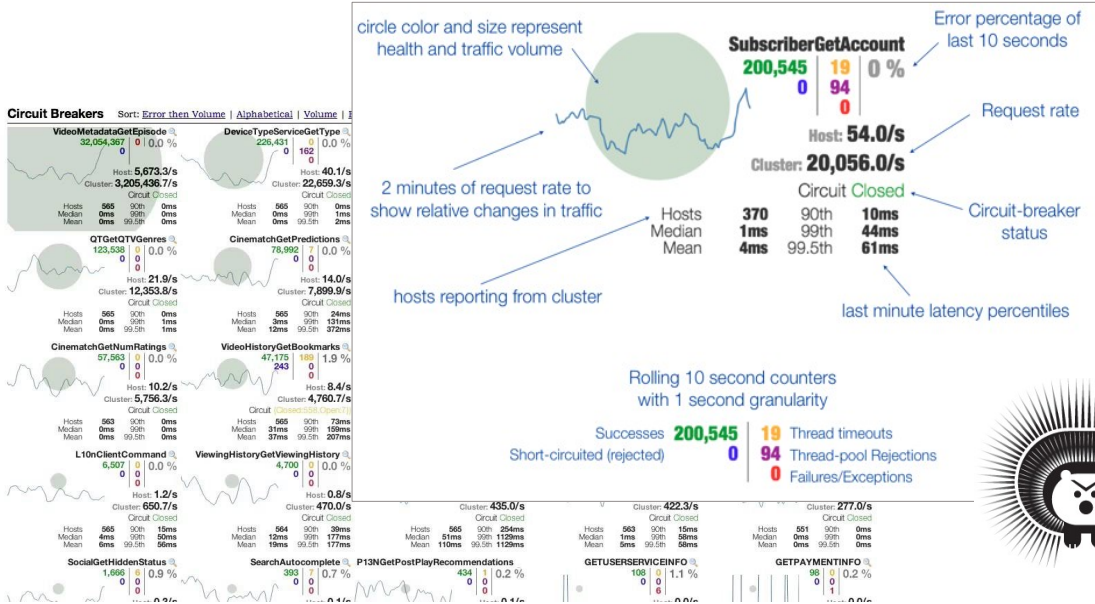
localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

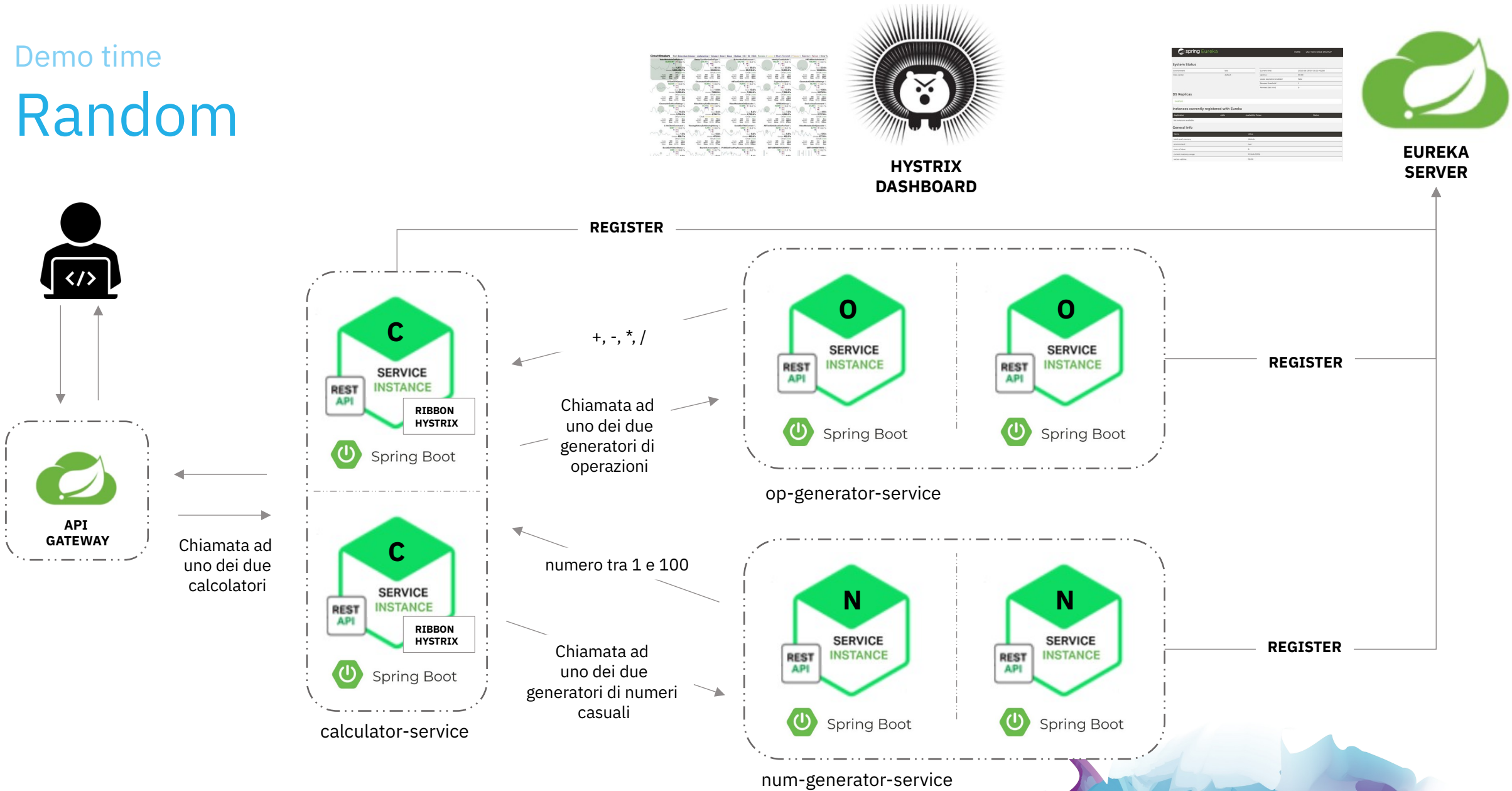
Name	Value
total-avail-memory	466mb
environment	test
num-of-cpus	8
current-memory-usage	153mb (32%)
server-upptime	00:00



Hystrix

Demo time

# Random



# Informazioni Utili



## Giorgio Dramis

Email: [Giorgio.Dramis-CIC-IT@ibm.com](mailto:Giorgio.Dramis-CIC-IT@ibm.com)

- Installazione Docker: <https://docs.docker.com/docker-for-windows/install/>
- Abilitazione Hyper-V: <https://docs.microsoft.com/it-it/virtualization/hyper-v-on-windows/quick-start/enable-hyper-v>
- Installazione Apache Maven: <https://maven.apache.org/install.html>
- Gartner Hype Cycle: <https://www.gartner.com/en/research/methodologies/gartner-hype-cycle>
- Architettura Esagonale: <https://www.infoq.com/news/2014/10/exploring-hexagonal-architecture>
- Architettura Monolitica: <https://microservices.io/patterns/monolithic.html>
- Architettura a Microservizi: <https://microservices.io/patterns/microservices.html>
- Scale Cube: <https://microservices.io/articles/scalecube.html>
- Architettura a Microservizi : <https://martinfowler.com/articles/microservices.html>
- Domain Driven Design: [https://en.wikipedia.org/wiki/Domain-driven\\_design](https://en.wikipedia.org/wiki/Domain-driven_design)
- Sam Newman 2015, Principles Of Microservices: <https://youtu.be/PFQnNFe27kU?t=1m50s>
- Transazioni distribuite: <https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/>
- Pattern SAGA: <https://microservices.io/patterns/data/saga.html>
- Cosa sono i microservizi? – <https://youtu.be/CdBtNQZH8a4>

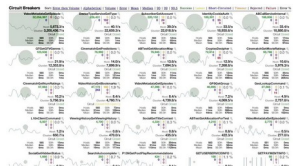
# Agenda

-  1 Riepilogo lezione precedente
-  2 Design Pattern – Parte 1
-  3 Demo time ed informazioni utili
-  4 **Esercizi, domande e risposte**

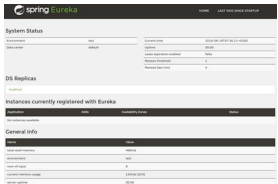


Design Pattern

# Esercizio

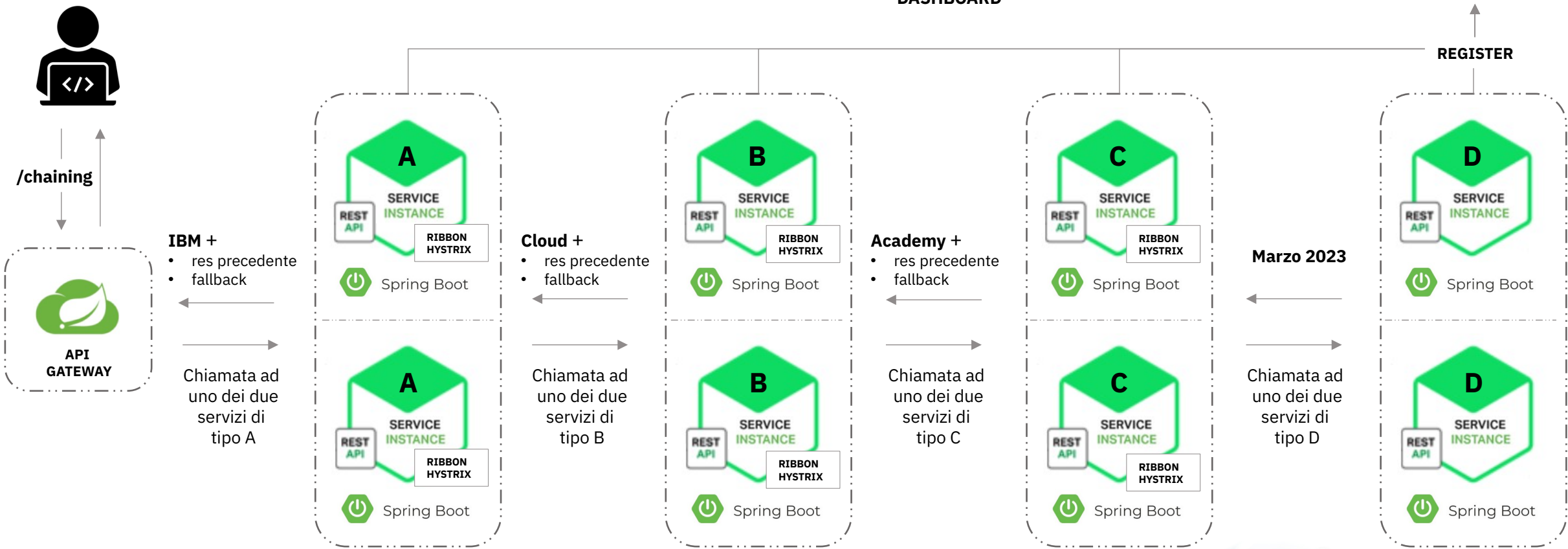


HYSTRIX  
DASHBOARD



EUREKA  
SERVER

REGISTER





# Questions & Answers





Experience.  
Create.  
Inspire.

# Grazie

Giorgio Dramis

IBM Client Innovation Center  
**Italy**

# Backup

Presented by

*Giorgio Dramis*

IBM Client Innovation Center - Bari

Roma, 18/12/2023



IBM Client Innovation Center  
**Italy**

