



Microservizi

Introduzione ai concetti principali

Presented by

Giorgio Dramis

IBM Client Innovation Center

Dicembre 2023

IBM Client Innovation Center
Italy

Agenda



Introduzione ai microservizi



Caratteristiche



Benefici introdotti



Punti di attenzione



Demo time ed informazioni utili



Esercizi, domande e risposte

Agenda



Introduzione ai microservizi



Caratteristiche



Benefici introdotti



Punti di attenzione

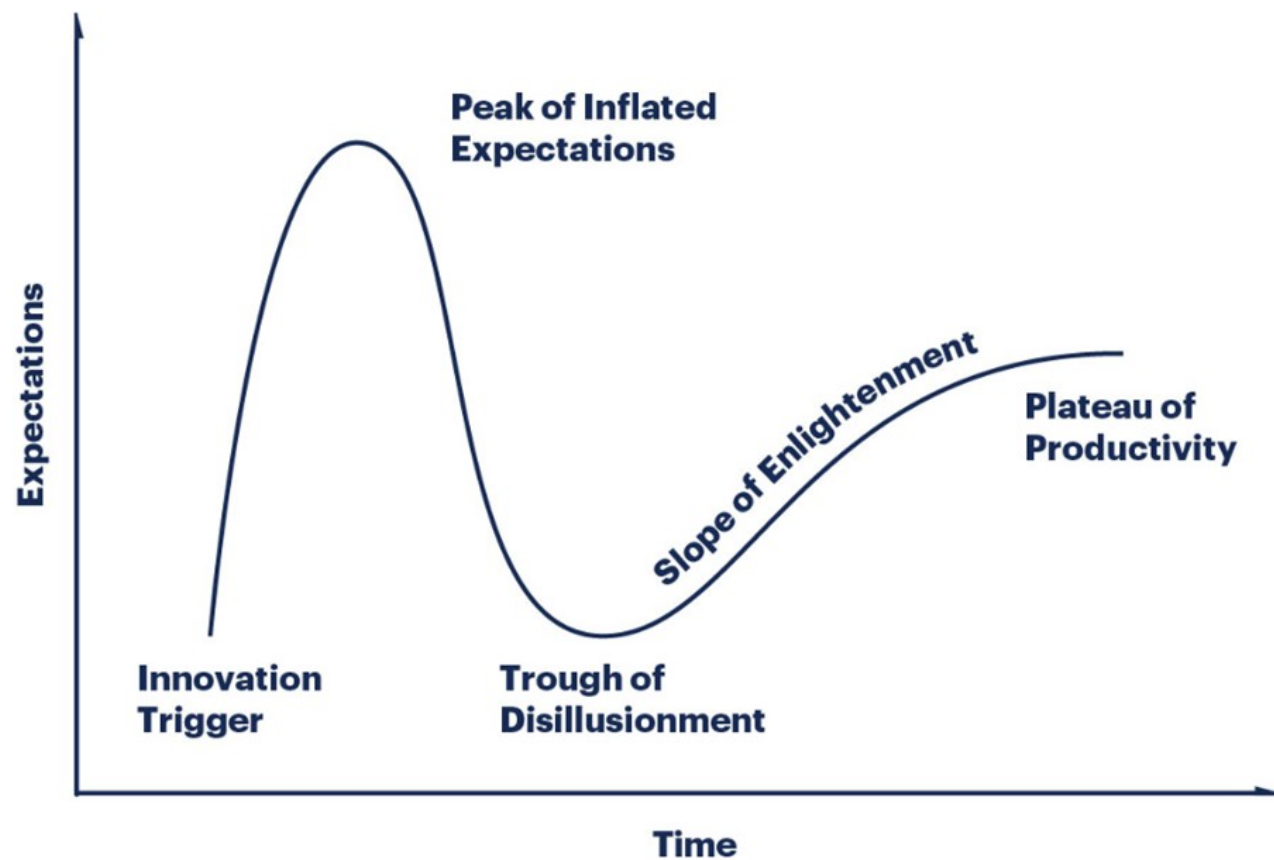


Demo time ed informazioni utili



Esercizi, domande e risposte

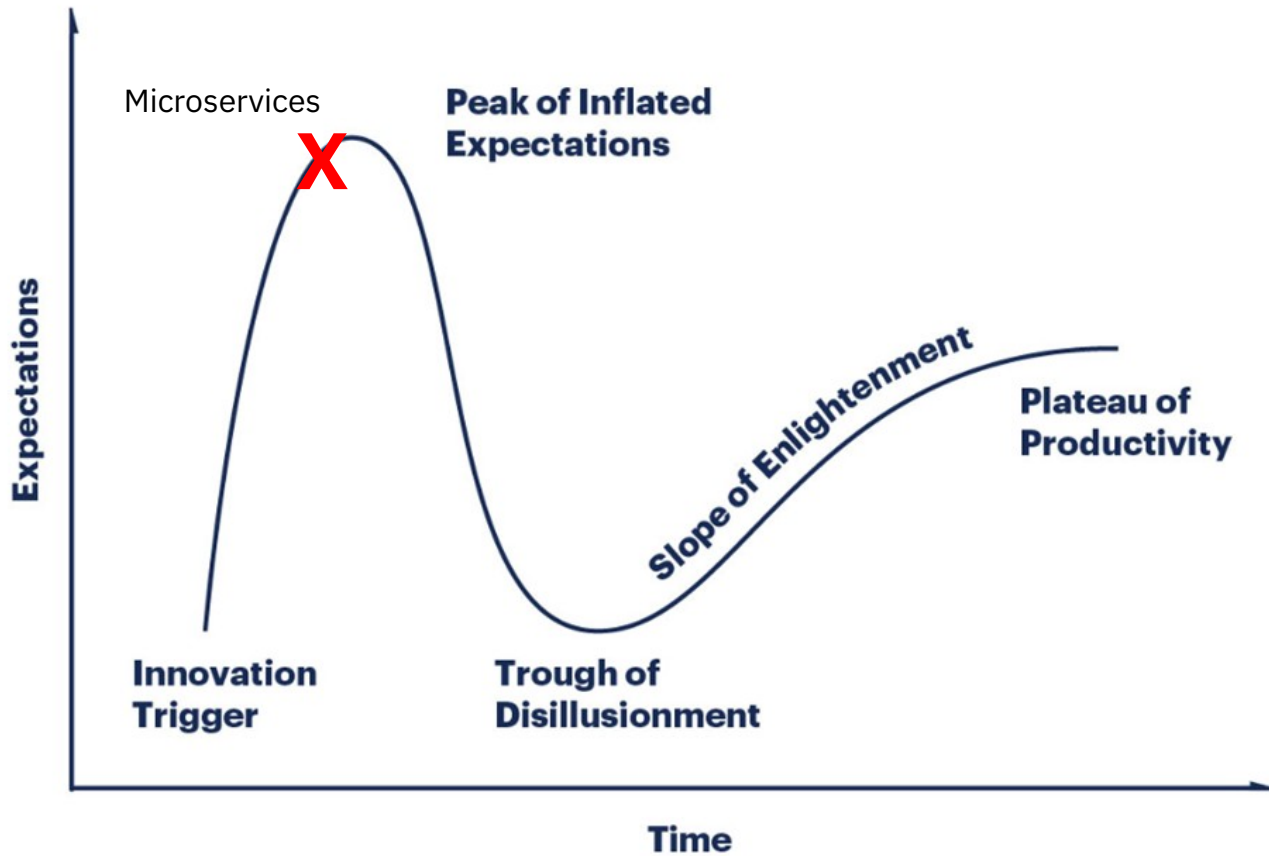
Contesto di riferimento



I microservizi stanno attualmente ottenendo molta attenzione: articoli, blog, discussioni sui social media e presentazioni nelle conferenze.

<https://www.gartner.com/en/research/methodologies/gartner-hype-cycle>

Contesto di riferimento



I microservizi stanno attualmente ottenendo molta attenzione: articoli, blog, discussioni sui social media e presentazioni nelle conferenze. Si stanno rapidamente dirigendo verso il picco delle aspettative esagerate del modello Hype Cycle di Gartner.

<https://www.gartner.com/en/research/methodologies/gartner-hype-cycle>

Introduzione ai microservizi

Applicazioni monolitiche

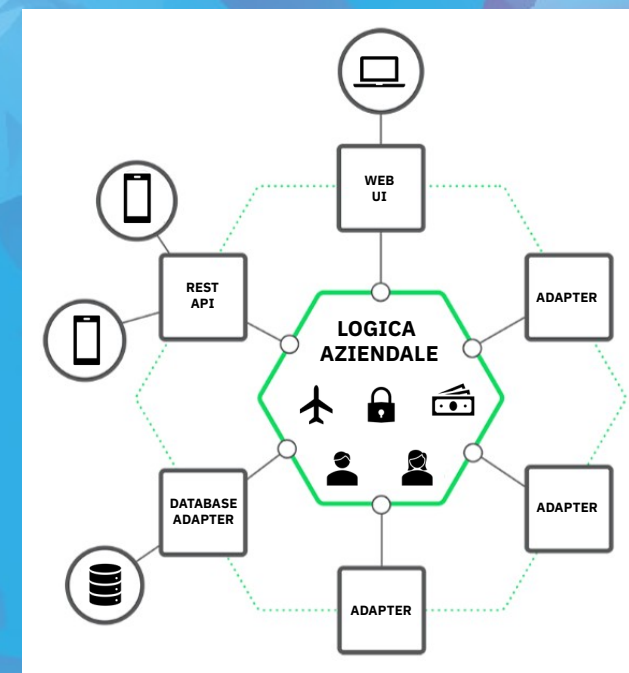


IBM Client Innovation Center
Italy

Applicazioni monolitiche

Immaginiamo di voler implementare un nuovo sistema di **gestione del personale**.

Il modello architetturale scelto nella progettazione è quello **esagonale** modulare.



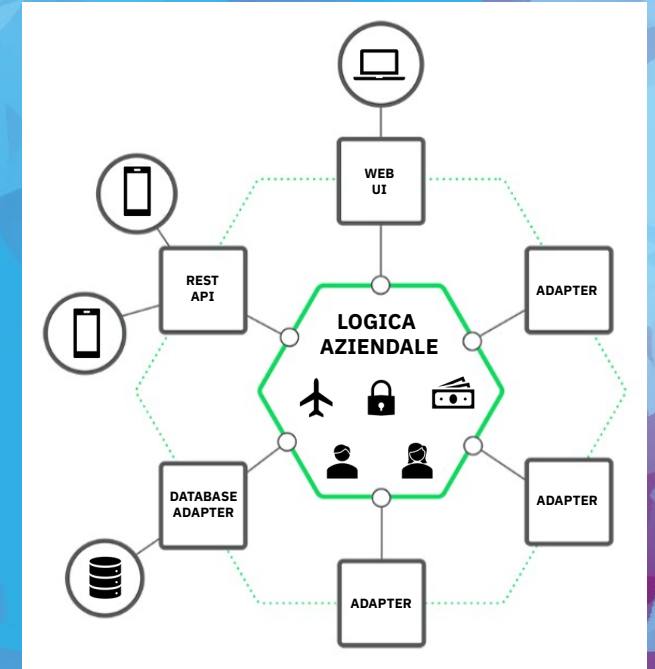
<https://www.infoq.com/news/2014/10/exploring-hexagonal-architecture>

Applicazioni monolitiche

Immaginiamo di voler implementare un nuovo sistema di **gestione del personale**.

Il modello architetturale scelto nella progettazione è quello **esagonale** modulare.

Nonostante abbia un'architettura logicamente modulare, l'applicazione è impacchettata e distribuita come un **monolite**: tutte le sue funzionalità sono strettamente collegate tra loro e vengono eseguite come un singolo processo.



<https://microservices.io/patterns/monolithic.html>

Applicazioni monolitiche

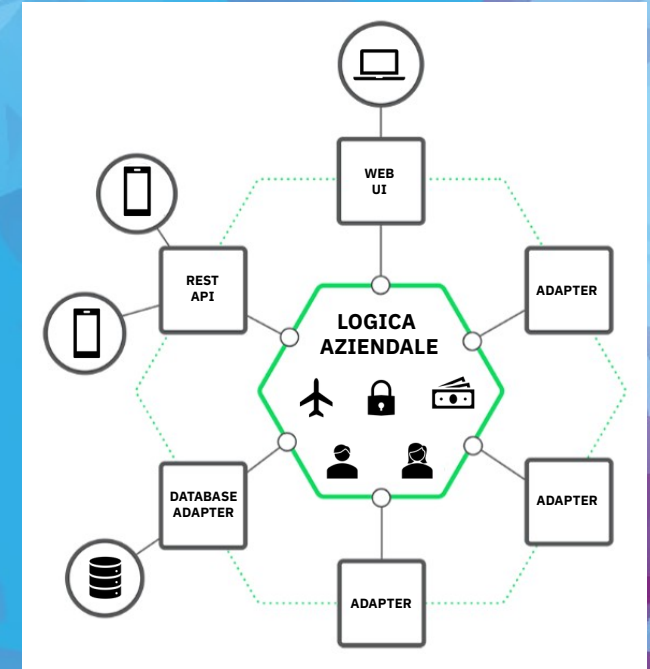
Immaginiamo di voler implementare un nuovo sistema di **gestione del personale**.

Il modello architetturale scelto nella progettazione è quello **esagonale** modulare.

Nonostante abbia un'architettura logicamente modulare, l'applicazione è impacchettata e distribuita come un **monolite**: tutte le sue funzionalità sono strettamente collegate tra loro e vengono eseguite come un singolo processo.

Nelle prime fasi di un progetto, queste applicazioni sono:

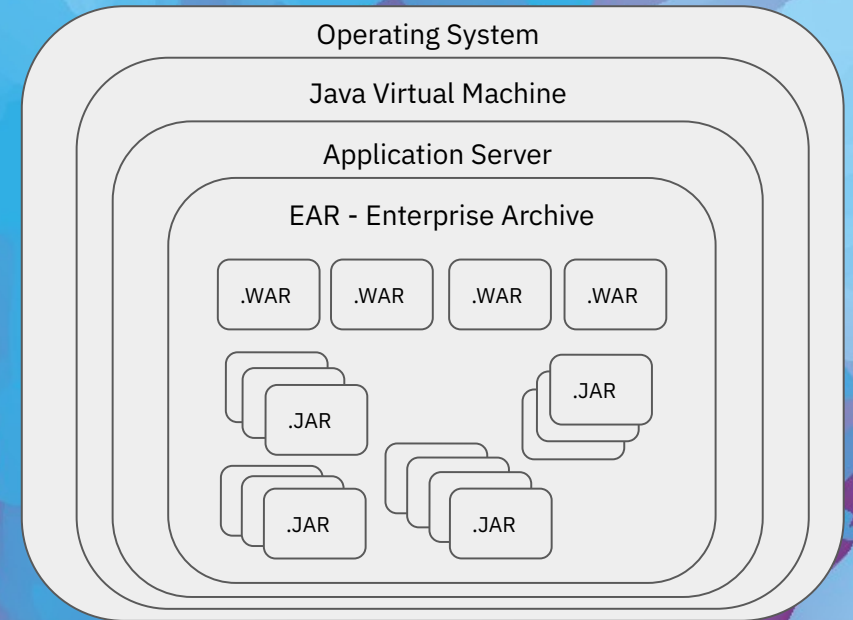
- **semplici da sviluppare**: i nostri ambienti di sviluppo sono focalizzati sulla creazione di una singola applicazione.
- **semplici da testare**: è possibile implementare test end-to-end semplicemente avviando l'applicazione e testando l'interfaccia utente.
- **semplici da deployare**: è necessario solo copiare l'applicazione pacchettizzata su un server e ridimensionarla, eseguendo più copie dietro un load balancer.



<https://microservices.io/patterns/monolithic.html>

Applicazioni monolitiche

Sfortunatamente le applicazioni hanno l'abitudine di crescere nel tempo. Dopo alcuni anni, la piccola e semplice applicazione diventerà una grande, complessa e incomprensibile **palla di fango**.



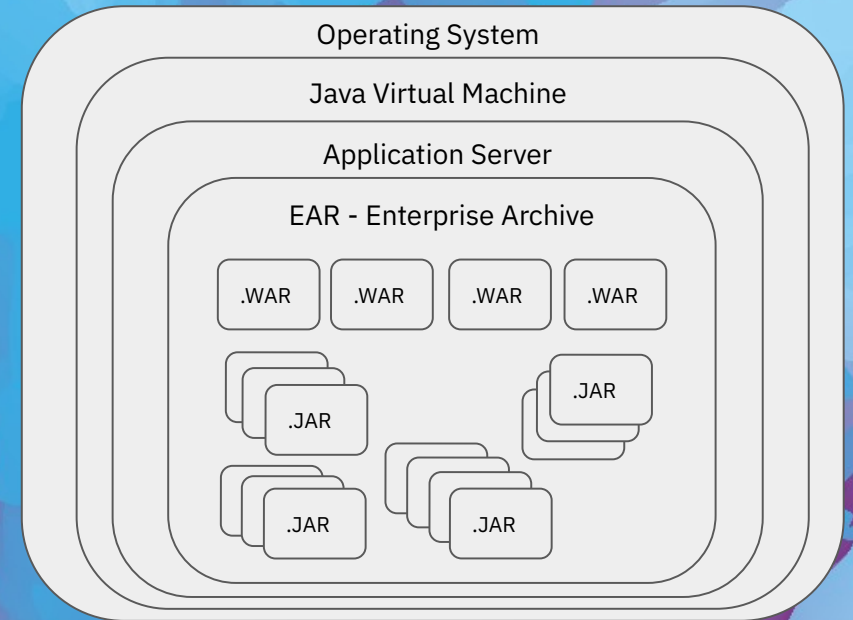
<https://microservices.io/patterns/monolithic.html>

Applicazioni monolitiche

Sfortunatamente le applicazioni hanno l'abitudine di crescere nel tempo. Dopo alcuni anni, la piccola e semplice applicazione diventerà una grande, complessa e incomprensibile **palla di fango**.

Un approccio monolitico porta numerosi svantaggi:

- L'enorme mole dell'applicazione **rallenta lo sviluppo**.
- Un'applicazione monolitica complessa rappresenta un **ostacolo al continuous deployment**. Oggi, la tendenza è quella di effettuare deploy in produzione più volte al giorno. Questo è estremamente difficile da fare con un monolite complesso poiché è necessario ridistribuire l'intera applicazione per aggiornarne solo una parte.



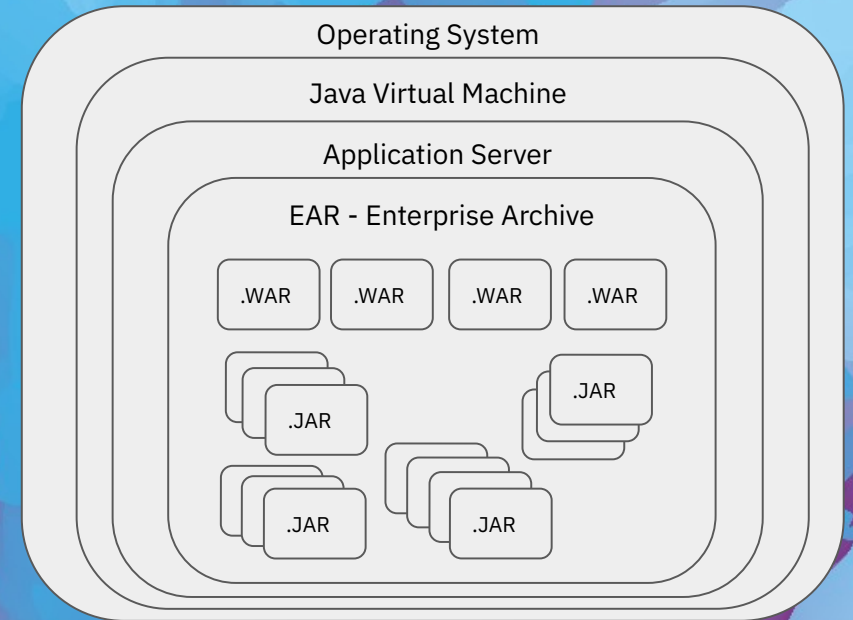
<https://microservices.io/patterns/monolithic.html>

Applicazioni monolitiche

Sfortunatamente le applicazioni hanno l'abitudine di crescere nel tempo. Dopo alcuni anni, la piccola e semplice applicazione diventerà una grande, complessa e incomprensibile **palla di fango**.

Un approccio monolitico porta numerosi svantaggi:

- L'enorme mole dell'applicazione **rallenta lo sviluppo**.
- Un'applicazione monolitica complessa rappresenta un **ostacolo al continuous deployment**. Oggi, la tendenza è quella di effettuare deploy in produzione più volte al giorno. Questo è estremamente difficile da fare con un monolite complesso poiché è necessario ridistribuire l'intera applicazione per aggiornarne solo una parte.
- Le applicazioni monolitiche possono anche essere **difficili da scalare quando moduli diversi hanno requisiti di risorse contrastanti**. È necessario scendere a compromessi sulla scelta dell'hardware.



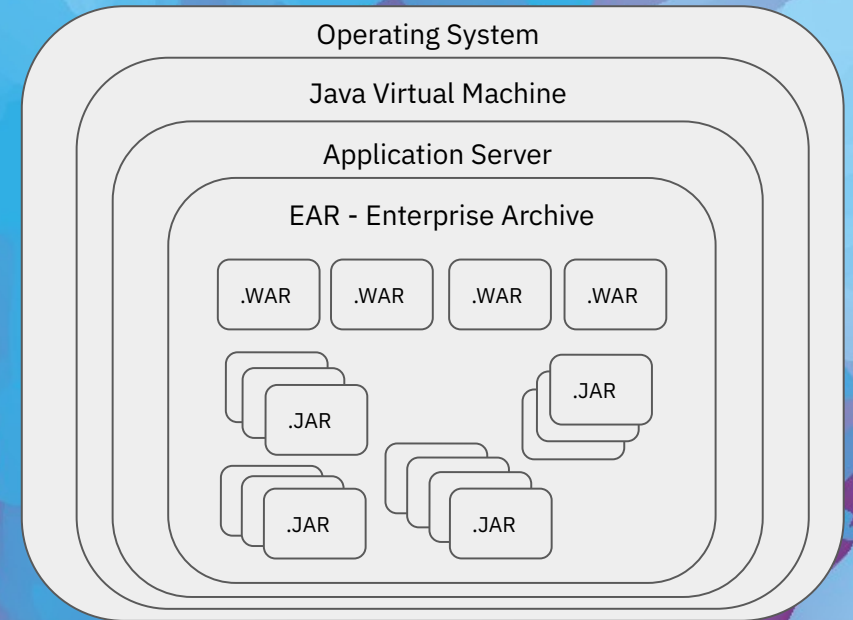
<https://microservices.io/patterns/monolithic.html>

Applicazioni monolitiche

Sfortunatamente le applicazioni hanno l'abitudine di crescere nel tempo. Dopo alcuni anni, la piccola e semplice applicazione diventerà una grande, complessa e incomprensibile **palla di fango**.

Un approccio monolitico porta numerosi svantaggi:

- Un altro problema con le applicazioni monolitiche è **l'affidabilità**. Poiché tutti i moduli sono in esecuzione all'interno dello stesso processo, un bug in qualsiasi modulo, es. memory leak, può potenzialmente arrestare l'intero processo.



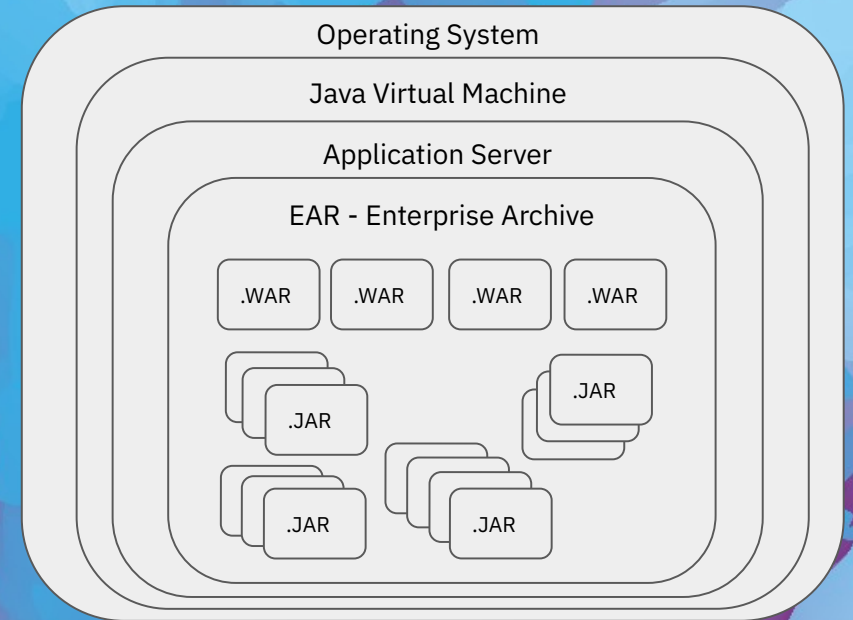
<https://microservices.io/patterns/monolithic.html>

Applicazioni monolitiche

Sfortunatamente le applicazioni hanno l'abitudine di crescere nel tempo. Dopo alcuni anni, la piccola e semplice applicazione diventerà una grande, complessa e incomprensibile **palla di fango**.

Un approccio monolitico porta numerosi svantaggi:

- Un altro problema con le applicazioni monolitiche è **l'affidabilità**. Poiché tutti i moduli sono in esecuzione all'interno dello stesso processo, un bug in qualsiasi modulo, es. memory leak, può potenzialmente arrestare l'intero processo.
- Infine, le applicazioni monolitiche rendono estremamente **difficile l'adozione di nuovi framework e linguaggi di programmazione**. Si è bloccati con qualsiasi scelta tecnologica si è fatto all'inizio del progetto.



<https://microservices.io/patterns/monolithic.html>

Spaghetti Architecture



SPAGHETTI-ORIENTED
ARCHITECTURE

La *Spaghetti Architecture*, è l'**anti-pattern** per le applicazioni monolitiche, in cui la struttura e le relazioni tra i moduli non sono così esplicite, la coesione strutturale e l'incapsulamento non esistono o sono minimi, le dipendenze non seguono regole ed è molto difficile apportare modifiche ed effettuare attività di refactoring.

Introduzione ai microservizi

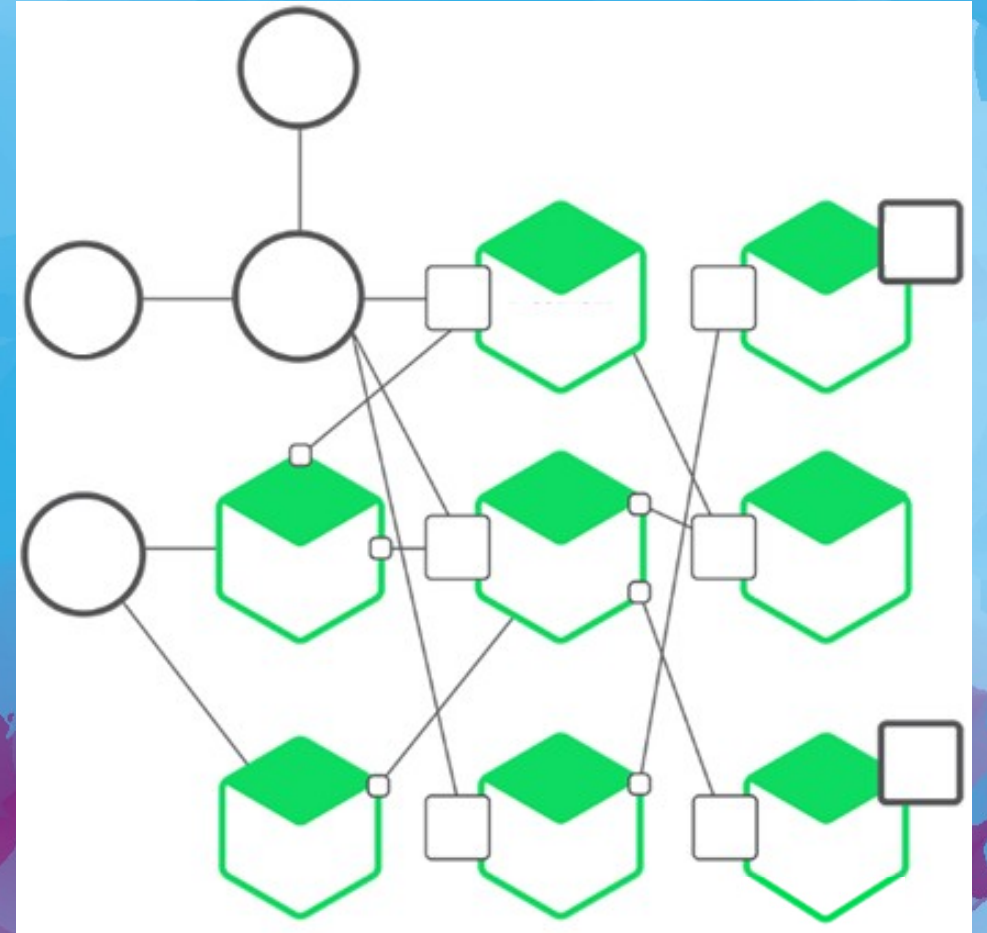
Affrontare la complessità



IBM Client Innovation Center
Italy

Affrontare la complessità

Molte organizzazioni hanno risolto questo problema adottando quella che ora è nota come **architettura a microservizi**. Invece di creare una singola applicazione monolitica, l'idea è di suddividere l'applicazione in un insieme di servizi più piccoli e interconnessi.

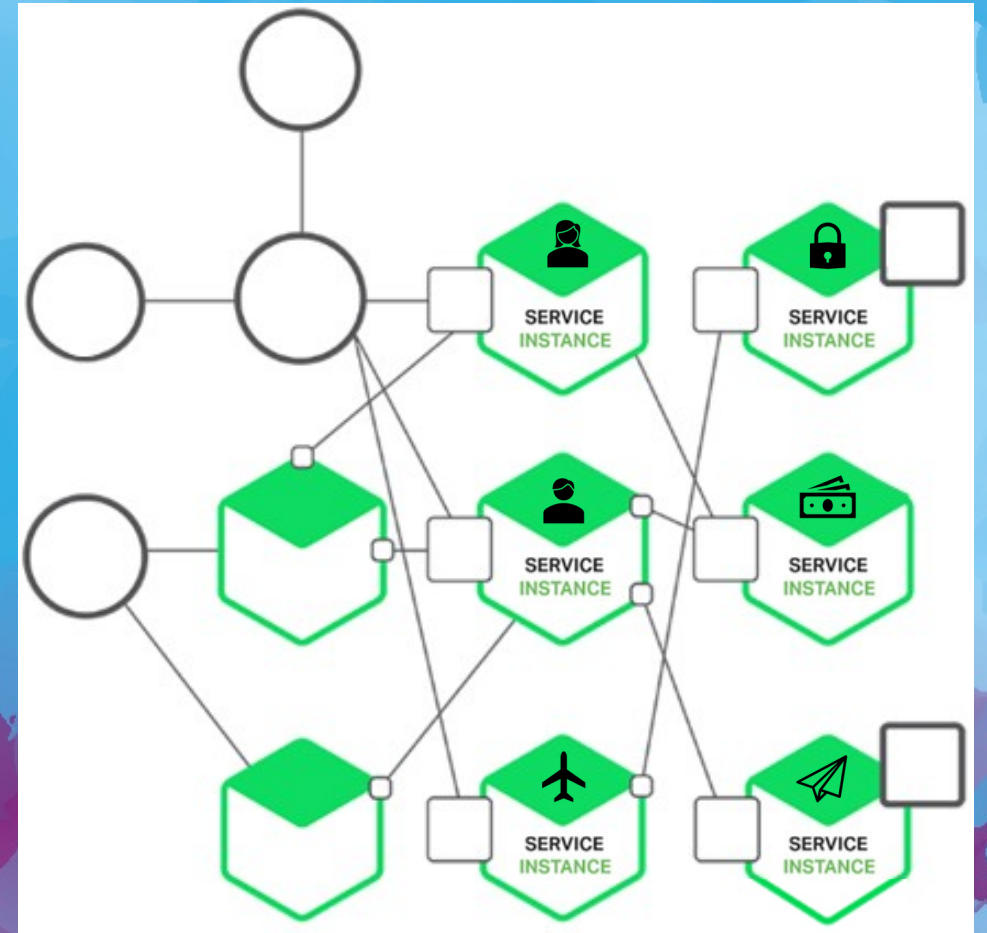


<https://microservices.io/patterns/microservices.html>

Affrontare la complessità

Molte organizzazioni hanno risolto questo problema adottando quella che ora è nota come **architettura a microservizi**. Invece di creare una singola applicazione monolitica, l'idea è di suddividere l'applicazione in un insieme di servizi più piccoli e interconnessi.

- Ogni servizio è una mini-applicazione con la propria architettura esagonale che implementa un insieme di caratteristiche o funzionalità distinte.

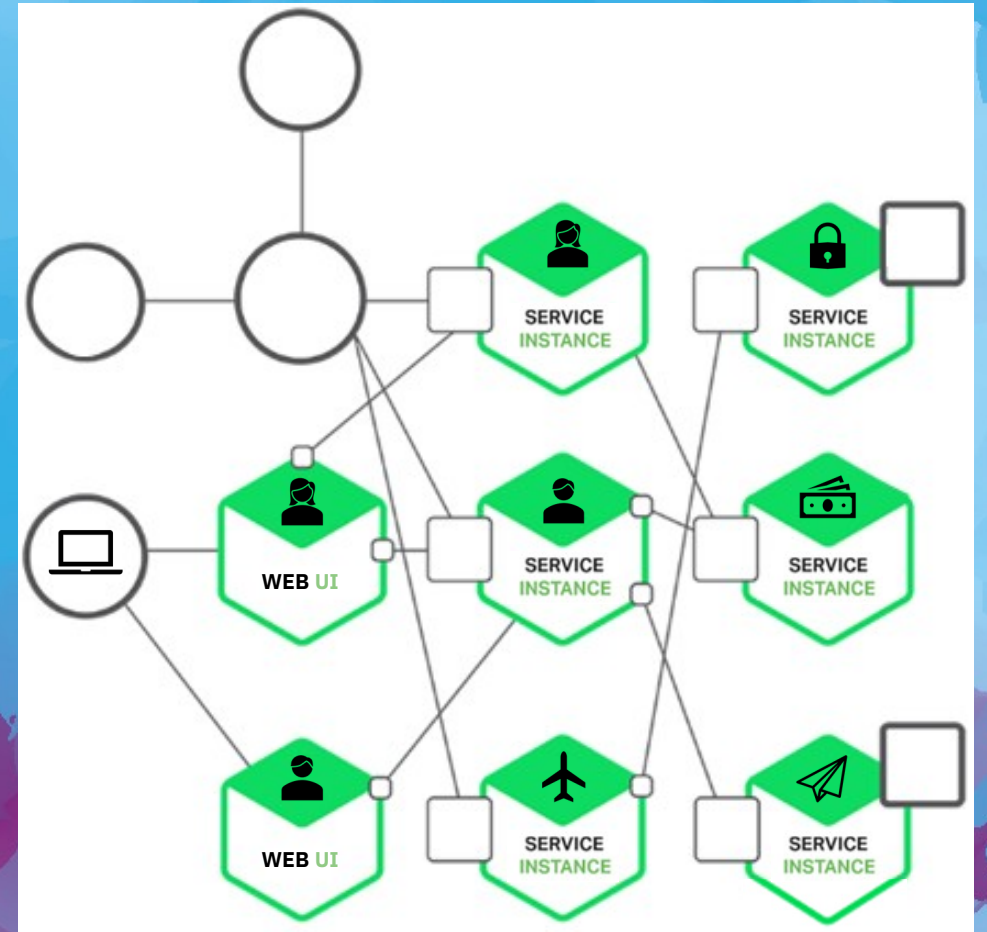


<https://microservices.io/patterns/microservices.html>

Affrontare la complessità

Molte organizzazioni hanno risolto questo problema adottando quella che ora è nota come **architettura a microservizi**. Invece di creare una singola applicazione monolitica, l'idea è di suddividere l'applicazione in un insieme di servizi più piccoli e interconnessi.

- Ogni servizio è una mini-applicazione con la propria architettura esagonale che implementa un insieme di caratteristiche o funzionalità distinte.
- L'applicazione web è suddivisa in una serie di applicazioni web più semplici. Questo rende più facile implementare esperienze distinte per utenti specifici, dispositivi o casi d'uso specializzati.



<https://microservices.io/patterns/microservices.html>

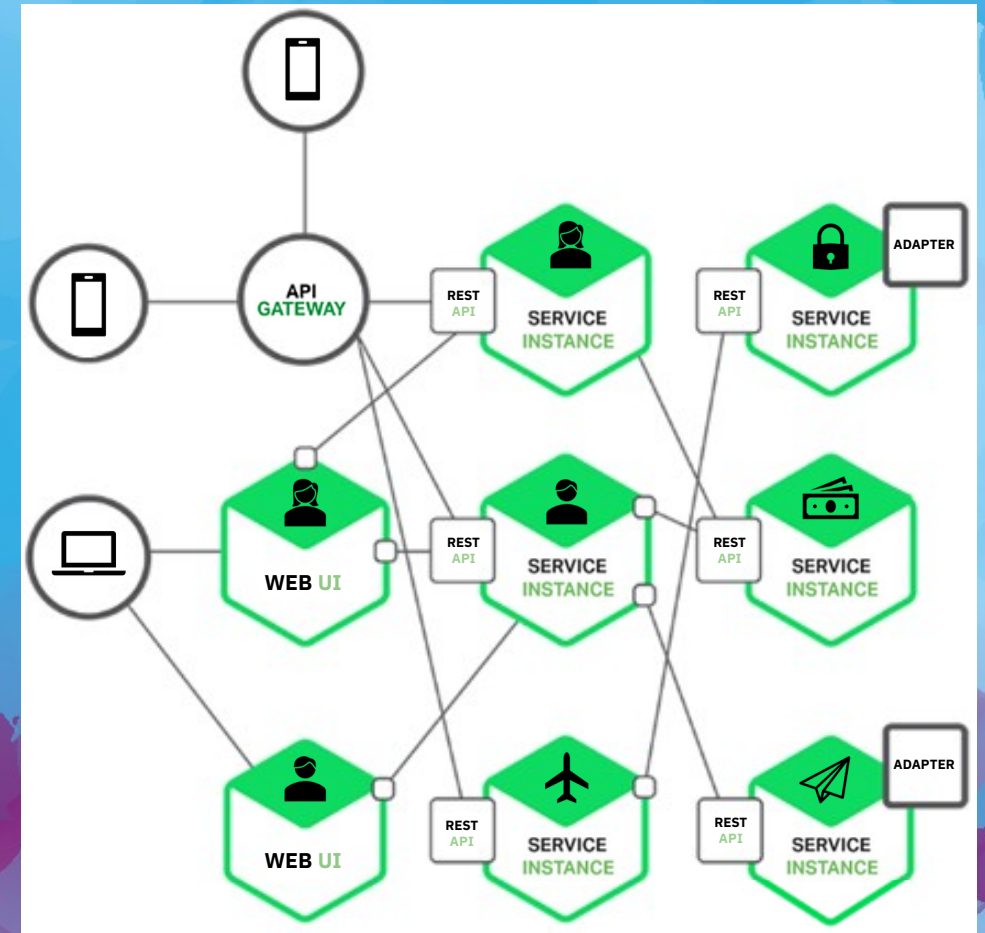
- Ogni servizio è una mini-applicazione con la propria architettura esagonale che implementa un insieme di caratteristiche o funzionalità distinte.
- L'applicazione web è suddivisa in una serie di applicazioni web più semplici. Questo rende più facile implementare esperienze distinte per utenti specifici, dispositivi o casi d'uso specializzati.
- Ogni servizio espone un'API REST e la maggior parte dei servizi utilizza API fornite da altri servizi. I servizi potrebbero anche usare una comunicazione asincrona basata su messaggi.



Affrontare la complessità

Molte organizzazioni hanno risolto questo problema adottando quella che ora è nota come **architettura a microservizi**. Invece di creare una singola applicazione monolitica, l'idea è di suddividere l'applicazione in un insieme di servizi più piccoli e interconnessi.

- In questo caso specifico, solo le app mobile non hanno accesso diretto ai servizi. La comunicazione è mediata da un intermediario noto come API gateway.

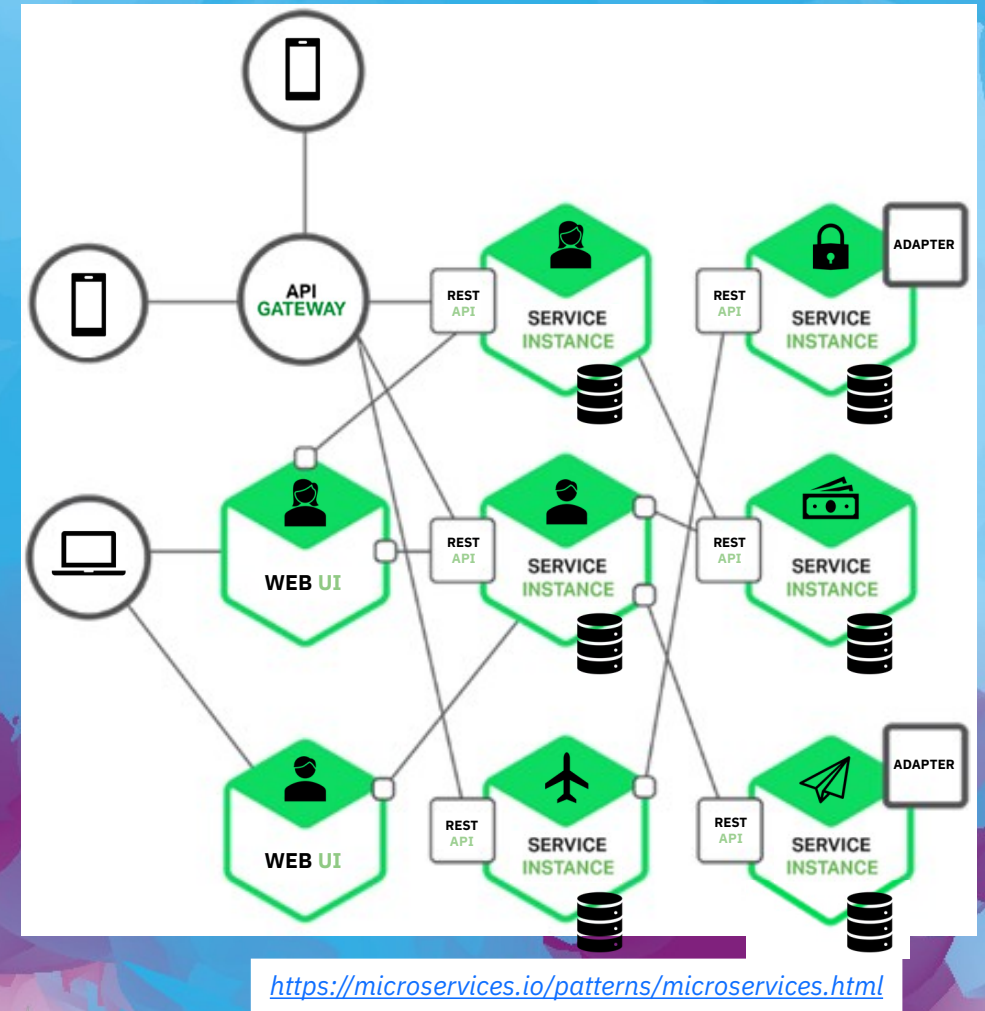


<https://microservices.io/patterns/microservices.html>

Affrontare la complessità

Molte organizzazioni hanno risolto questo problema adottando quella che ora è nota come **architettura a microservizi**. Invece di creare una singola applicazione monolitica, l'idea è di suddividere l'applicazione in un insieme di servizi più piccoli e interconnessi.

- In questo caso specifico, solo le app mobile non hanno accesso diretto ai servizi. La comunicazione è mediata da un intermediario noto come API gateway.
- Il modello dell'architettura dei microservizi ha un impatto significativo sulla relazione tra l'applicazione e il database. Anziché condividere un singolo database con altri servizi, ogni servizio ha il proprio database.
- In fase di esecuzione, ogni istanza è spesso una VM o un container.



Agenda



Introduzione ai microservizi



Caratteristiche



Benefici introdotti



Punti di attenzione



Demo time ed informazioni utili



Esercizi, domande e risposte

Diamo una definizione...

<https://martinfowler.com/articles/microservices.html>

*“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**, often an HTTP resource API. 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**.”*

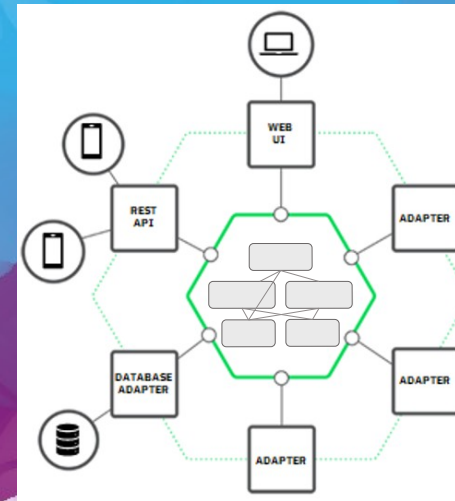
James Lewis and Martin Fowler

Caratteristiche

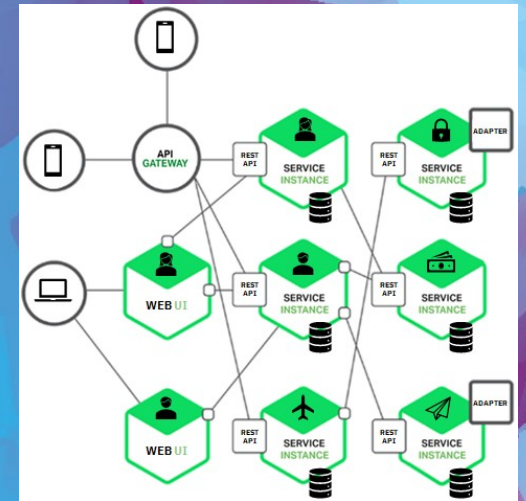
Componentizzazione in servizi e non più in librerie

Componentizzazione in servizi e non più in librerie

Le architetture a microservizi usano le librerie, ma il loro modo principale di strutturare il codice è quello di **suddividersi in servizi**. Uno dei motivi principali per l'utilizzo dei servizi come componenti è che sono **distribuibili in modo indipendente** ed **autonomi**.



Architettura monolitica

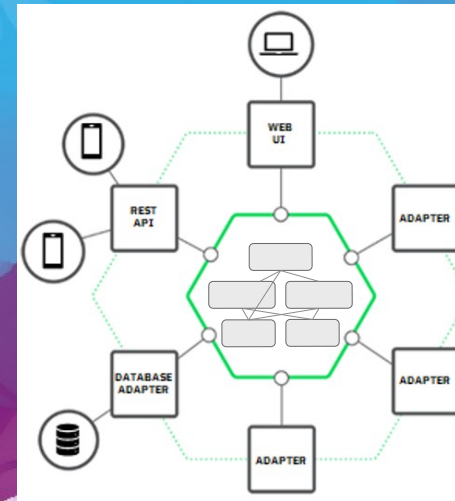


Architettura a microservizi

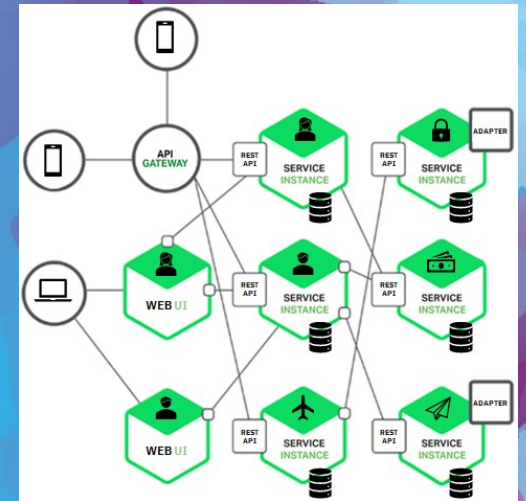
Componentizzazione in servizi e non più in librerie

Le architetture a microservizi usano le librerie, ma il loro modo principale di strutturare il codice è quello di **suddividersi in servizi**. Uno dei motivi principali per l'utilizzo dei servizi come componenti è che sono **distribuibili in modo indipendente** ed **autonomi**.

- Se si dispone di un'applicazione monolitica che consiste in più librerie in un singolo processo, **una modifica** a qualsiasi singolo componente comporta la **ridistribuzione dell'intera applicazione**. Ma se tale applicazione viene scomposta in più servizi, è possibile aspettarsi che **modifiche al singolo servizio** richiedano **solo la ridistribuzione del servizio stesso**. Questo garantisce **indipendenza nelle fasi di sviluppo, building e distribuzione del software**.



Architettura monolitica



Architettura a microservizi

Componentizzazione in servizi e non più in librerie

Le architetture a microservizi usano le librerie, ma il loro modo principale di strutturare il codice è quello di **suddividersi in servizi**. Uno dei motivi principali per l'utilizzo dei servizi come componenti è che sono **distribuibili in modo indipendente** ed **autonomi**.

- Sono, inoltre, ideali per essere **eseguiti in singoli container**, implementando logica stateless, in modo da massimizzare la scalabilità orizzontale, ed esponendo **interfacce language-agnostic** con l'utilizzo di formati leggeri come **JSON**.



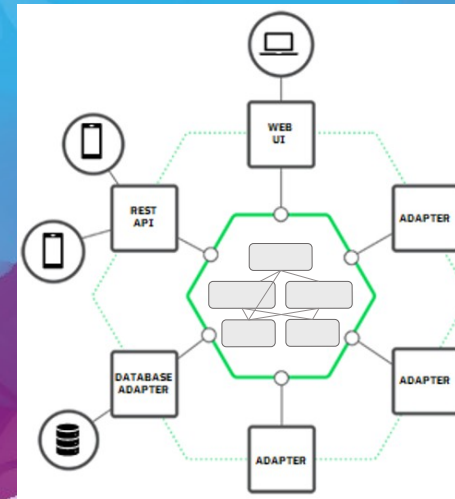
cri-o



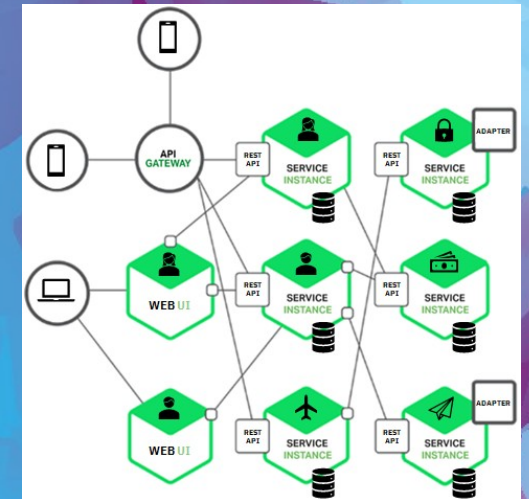
podman



LXD



Architettura monolitica



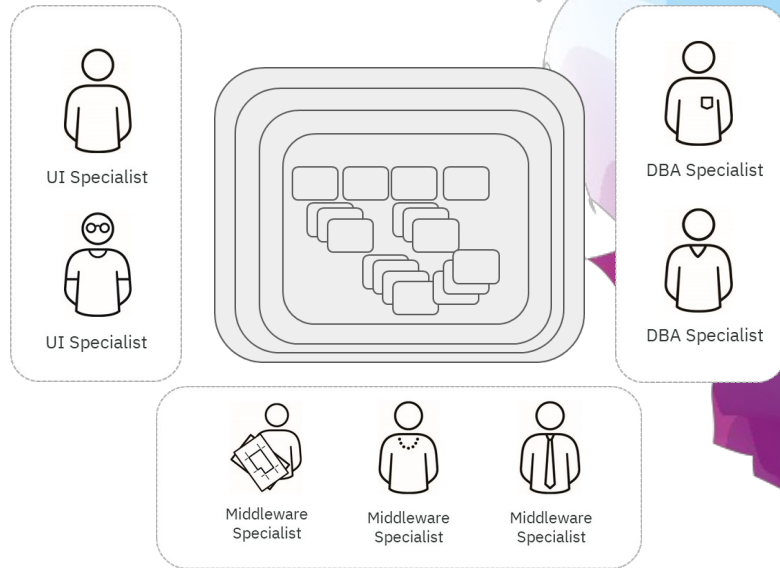
Architettura a microservizi

Caratteristiche

Organizzazione attorno alle capacità aziendali

Caratteristiche

Organizzazione attorno alle capacità aziendali

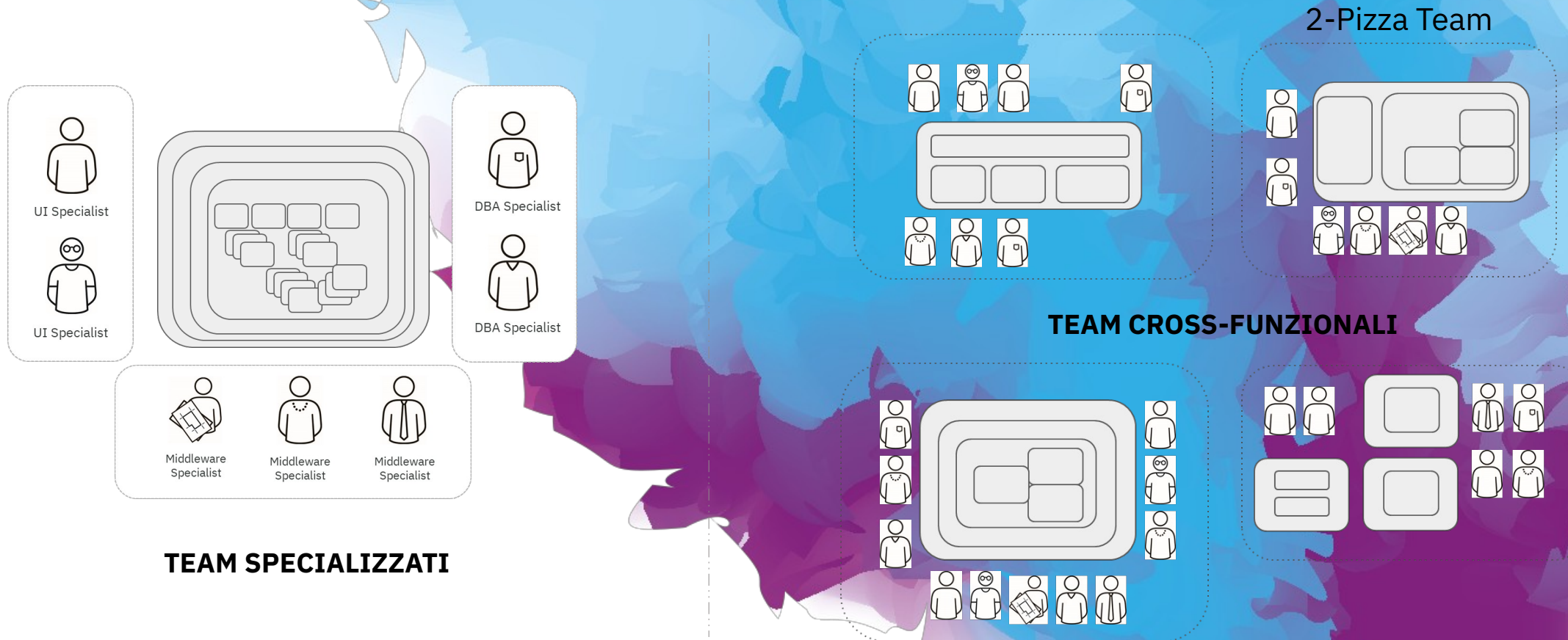


TEAM SPECIALIZZATI

IBM Client Innovation Center
Italy

Caratteristiche

Organizzazione attorno alle capacità aziendali



Caratteristiche

Responsabilità definita da confini ben definiti



IBM Client Innovation Center
Italy

Responsabilità definita da confini ben definiti

Ogni servizio dovrà implementare una singola funzionalità business e dovrà contenere tutto quello che è necessario a funzionare correttamente. Un modo utile di pensare a questo è la nozione data dal domain-driven design: i **bounded context**.

Potremmo definirli come degli ambiti di business indipendenti gli uni dagli altri ed ognuno con il proprio **ubiquitous language**, ossia un linguaggio univoco, comune e rigoroso tra tutti gli attori che partecipano ad un specifico contesto. Ogni bounded context deve essere una sorta di **black box** che comunica con altri context.



https://en.wikipedia.org/wiki/Domain-driven_design

Responsabilità definita da confini ben definiti

Ogni servizio dovrà implementare una singola funzionalità business e dovrà contenere tutto quello che è necessario a funzionare correttamente. Un modo utile di pensare a questo è la nozione data dal domain-driven design: i **bounded context**.

Potremmo definirli come degli ambiti di business indipendenti gli uni dagli altri ed ognuno con il proprio **ubiquitous language**, ossia un linguaggio univoco, comune e rigoroso tra tutti gli attori che partecipano ad un specifico contesto. Ogni bounded context deve essere una sorta di **black box** che comunica con altri context.

In definitiva, si divide un dominio complesso in più contesti limitati e decentralizzati, e si mappano le relazioni tra di loro. Ogni microservizio, quindi, dovrà essere **specializzato**, ossia progettato per una specifica funzionalità o sulla risoluzione di un problema specifico. Se, nel tempo, si inserisce del codice aggiuntivo a un servizio rendendolo più complesso, il servizio dovrà essere scomposto in servizi più piccoli.



https://en.wikipedia.org/wiki/Domain-driven_design

Caratteristiche

Governance dei contenuti decentralizzata




IBM Client Innovation Center
Italy

Governance dei contenuti decentralizzata

Oltre a decentralizzare i modelli concettuali, i microservizi decentralizzano anche l'archiviazione dei contenuti. Mentre le applicazioni monolitiche preferiscono un unico database logico per i dati persistenti, i microservizi preferiscono consentire a ciascun servizio di gestire il proprio database, sia istanze diverse della stessa tecnologia di database, sia sistemi di database completamente diversi, un approccio chiamato **polyglot persistence**.

 CouchDB

 mongoDB

ORACLE
DATABASE

 MySQL


PostgreSQL

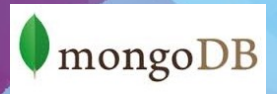

Couchbase

Governance dei contenuti decentralizzata

Oltre a decentralizzare i modelli concettuali, i microservizi decentralizzano anche l'archiviazione dei contenuti. Mentre le applicazioni monolitiche preferiscono un unico database logico per i dati persistenti, i microservizi preferiscono consentire a ciascun servizio di gestire il proprio database, sia istanze diverse della stessa tecnologia di database, sia sistemi di database completamente diversi, un approccio chiamato **polyglot persistence**.

La decentralizzazione della responsabilità dei dati tra i diversi microservizi ha implicazioni sulla **gestione degli aggiornamenti sui quei dati**.

L'approccio comune alla gestione degli aggiornamenti è stato quello di utilizzare le **transazioni** per garantire **coerenza** in stile ACID durante l'aggiornamento di più risorse. Questo approccio viene spesso utilizzato all'interno dei monoliti. **L'uso di transazioni** aiuta la coerenza dei dati, ma **impone un significativo accoppiamento**, il che è problematico in contesti a microservizi.



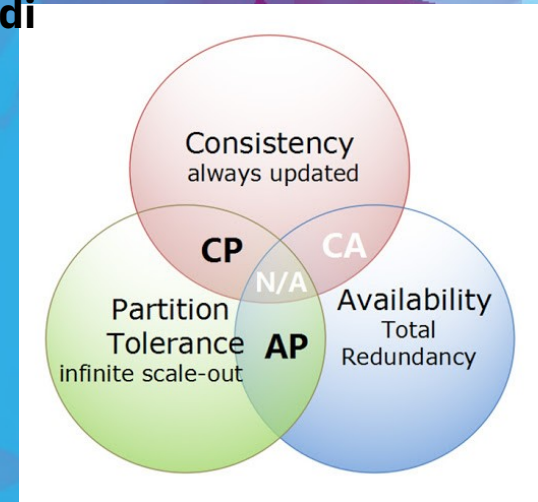
Governance dei contenuti decentralizzata

Le transazioni distribuite (su protocollo 2PC) sono notoriamente difficili da implementare e, tuttavia, non sono un'opzione praticabile con molte tecnologie moderne. La maggior parte dei database NoSQL non supportano 2PC, per esempio.

Governance dei contenuti decentralizzata

Le transazioni distribuite (su protocollo 2PC) sono notoriamente difficili da implementare e, tuttavia, non sono un'opzione praticabile con molte tecnologie moderne. La maggior parte dei database NoSQL non supportano 2PC, per esempio.

Un suggerimento arriva dal **teorema CAP** che chiede di scegliere tra **totale consistenza** in stile ACID e **continua disponibilità**. In un sistema distribuito è molto probabile avere **problemi di rete** e, quindi, la tolleranza alle partizioni è fondamentale e non può essere esclusa.

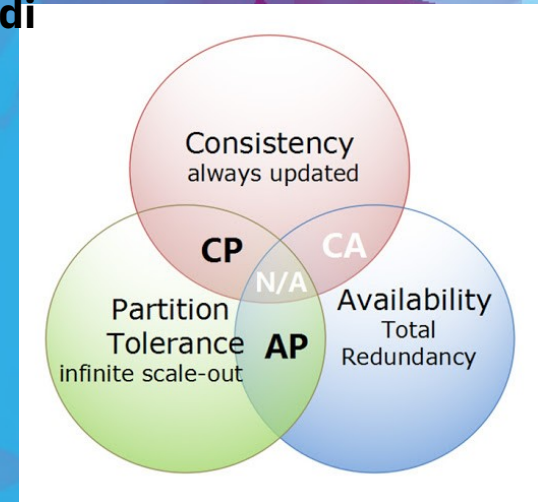


Governance dei contenuti decentralizzata

Le transazioni distribuite (su protocollo 2PC) sono notoriamente difficili da implementare e, tuttavia, non sono un'opzione praticabile con molte tecnologie moderne. La maggior parte dei database NoSQL non supportano 2PC, per esempio.

Un suggerimento arriva dal **teorema CAP** che chiede di scegliere tra **totale consistenza** in stile ACID e **continua disponibilità**. In un sistema distribuito è molto probabile avere **problemi di rete** e, quindi, la tolleranza alle partizioni è fondamentale e non può essere esclusa.

- Le architetture monolitiche, scelgono la **consistenza** rispetto alla disponibilità. L'approccio alla gestione degli aggiornamenti è quello di utilizzare le **transazioni**.



Governance dei contenuti decentralizzata

Le transazioni distribuite (su protocollo 2PC) sono notoriamente difficili da implementare e, tuttavia, non sono un'opzione praticabile con molte tecnologie moderne. La maggior parte dei database NoSQL non supportano 2PC, per esempio.

Un suggerimento arriva dal **teorema CAP** che chiede di scegliere tra **totale consistenza** in stile ACID e **continua disponibilità**. In un sistema distribuito è molto probabile avere **problemi di rete** e, quindi, la tolleranza alle partizioni è fondamentale e non può essere esclusa.

- Le architetture monolitiche, scelgono la **consistenza** rispetto alla disponibilità. L'approccio alla gestione degli aggiornamenti è quello di utilizzare le **transazioni**.
- Mentre le architetture a microservizi, scelgono la **disponibilità** piuttosto che la consistenza, enfatizzando il concetto di **coordinamento**, con la risoluzione di problematiche legate alla coerenza dei dati mediante operazioni di **compensazione**.



<https://microservices.io/patterns/data/saga.html>

Agenda



Introduzione ai microservizi



Caratteristiche



Benefici introdotti



Punti di attenzione



Demo time ed informazioni utili



Esercizi, domande e risposte

Benefici introdotti

Decompose complexity



IBM Client Innovation Center
Italy

Decompose complexity

1

Si affronta il problema della complessità: si decompone una applicazione monolitica in un insieme di servizi. Mentre la quantità totale di funzionalità è rimasta invariata, l'applicazione è stata suddivisa in blocchi o servizi più gestibili e snelli. Di conseguenza, sono molto più veloci da sviluppare e molto più facili da comprendere e mantenere.

Decompose complexity

1

Si affronta il problema della complessità: si decompone una applicazione monolitica in un insieme di servizi. Mentre la quantità totale di funzionalità è rimasta invariata, l'applicazione è stata suddivisa in blocchi o servizi più gestibili e snelli. Di conseguenza, sono molto più veloci da sviluppare e molto più facili da comprendere e mantenere.

2

Questa architettura consente a ciascun servizio di essere sviluppato in modo indipendente da un team focalizzato su quel servizio: gli sviluppatori sono liberi di scegliere qualsiasi tecnologia abbia senso, a condizione che il servizio onori le funzionalità descritte nelle API.

Il punto chiave è che non si è vincolati da decisioni passate

VERT.X



Benefici introdotti

Resilienza ai guasti e scalabilità



IBM Client Innovation Center
Italy

Resilienza ai guasti e scalabilità

3

Il modello di architettura di microservizi consente di ridimensionare ogni servizio in modo indipendente. In base all'utilizzo e alla richiesta, è possibile distribuire solo il numero di istanze di ciascun servizio che soddisfino tali richieste in termini di capacità e disponibilità. Inoltre, è possibile utilizzare l'hardware che soddisfa meglio i requisiti di risorse di un servizio.

Resilienza ai guasti e scalabilità

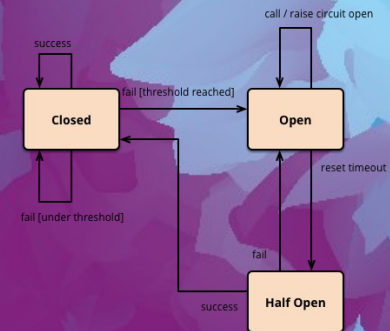
3

Il modello di architettura di microservizi consente di ridimensionare ogni servizio in modo indipendente. In base all'utilizzo e alla richiesta, è possibile distribuire solo il numero di istanze di ciascun servizio che soddisfino tali richieste in termini di capacità e disponibilità. Inoltre, è possibile utilizzare l'hardware che soddisfa meglio i requisiti di risorse di un servizio.

4

Resilienza: migliore gestione dei problemi: l'architettura a microservizi ha un migliore isolamento dei guasti. Un problema in un servizio influisce solo su quel servizio. Altri servizi continueranno a funzionare normalmente. In confronto, un componente che va in errore in un'architettura monolitica farà crollare l'intero sistema.

Suddividendo l'architettura in microservizi possiamo garantire che il **malfunzionamento di una singola componente non pregiudichi il funzionamento dell'intero sistema.**



Agenda



Introduzione ai microservizi



Caratteristiche



Benefici introdotti



Punti di attenzione



Demo time ed informazioni utili



Esercizi, domande e risposte

Ravioli Architecture

1

Un punto di attenzione è il nome stesso. Il termine *microservizio* pone un'enfasi eccessiva sulla dimensione del servizio. Sono preferibili piccoli servizi, ma è importante ricordare che sono un mezzo per raggiungere un fine e non l'obiettivo primario. L'obiettivo dei microservizi è decomporre in modo sufficiente l'applicazione al fine di facilitarne lo sviluppo e la distribuzione agile.

Ravioli Architecture è il nome comunemente usato per riferirsi all'anti-pattern relativo all'architettura a microservizi. Succede quando finiamo per creare un ecosistema di **micro**servizi in cui sono troppi, troppo piccoli e non sono in termini di dominio auto consistenti.



RAVIOLI-ORIENTED
ARCHITECTURE

Sistema distribuito e partizionato

2

Un altro grande punto d'attenzione dei microservizi è la **complessità** che deriva dal fatto che un sistema a microservizi è un **sistema distribuito**. E' necessario implementare un meccanismo di comunicazione tra servizi e gestire un eventuale errore, dovuto da richiesta verso un altro servizio lenta o non disponibile. È molto più complesso che in un'applicazione monolitica in cui i moduli si invocano l'un l'altro tramite chiamate in memoria.

Sistema distribuito e partizionato

2

Un altro grande punto d'attenzione dei microservizi è la **complessità** che deriva dal fatto che un sistema a microservizi è un **sistema distribuito**. E' necessario implementare un meccanismo di comunicazione tra servizi e gestire un eventuale errore, dovuto da richiesta verso un altro servizio lenta o non disponibile. È molto più complesso che in un'applicazione monolitica in cui i moduli si invocano l'un l'altro tramite chiamate in memoria.

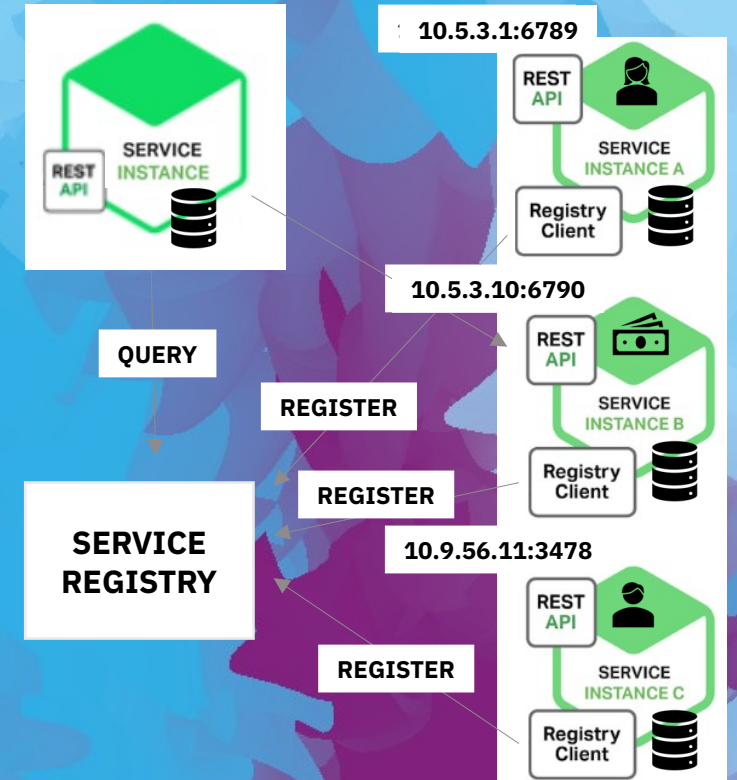
3

Un'altra sfida con i microservizi è **l'architettura del database partizionata**. Le transazioni che aggiornano più entità sono abbastanza comuni. Questi tipi di transazioni sono banali da implementare in un'applicazione monolitica perché incidono su un unico database. In un'applicazione basata su microservizi, tuttavia, è necessario aggiornare più database di proprietà di diversi servizi: concetto di **coordinamento** e **compensazione** visto precedentemente.

Discovery dei servizi

La distribuzione di un'applicazione basata su microservizi è anche molto complessa.

- Un'applicazione monolitica viene semplicemente distribuita su un insieme di server identici dietro un bilanciamento del carico tradizionale.
- Un'applicazione a microservizi è in genere costituita da un gran numero di servizi. Ogni servizio avrà più istanze di runtime. Sarà necessario implementare **un meccanismo di individuazione dei servizi** che consenta a un servizio di rilevare le posizioni (host e porte) di qualsiasi altro servizio con cui deve comunicare.



Agenda



Introduzione ai microservizi



Caratteristiche



Benefici introdotti



Punti di attenzione



Demo time ed informazioni utili



Esercizi, domande e risposte

Demo time

Java EE Runtime per Container



Spring Boot

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

<https://start.spring.io/>

VERT.X

<https://vertx.io/>

<https://start.vertx.io/>



QUARKUS

<https://quarkus.io/>

<https://code.quarkus.io/>

Spring Boot

Spring Boot è una soluzione "convention over configuration" per il framework Spring di Java, che è stato rilasciato nel 2012 e riduce la complessità di configurazione di nuovi progetti Spring. A questo scopo, Spring Boot definisce una configurazione di base che include le linee guida per l'uso del framework e tutte le librerie di terze parti rilevanti, rendendo quindi l'avvio di nuovi progetti il più semplice possibile. In questo modo, la creazione di applicazioni indipendenti e pronte per la produzione basate su Spring può essere notevolmente semplificata. Alcuni dei principali vantaggi:

- possibilità di incorporare direttamente applicazioni web server/container come Apache Tomcat o Jetty, per cui non è necessario l'uso di file WAR (Web Application Archive);
- configurazione Maven semplificata grazie ai POM "Starter" (Project Object Models);
- configurazione automatica di Spring, ove possibile;

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

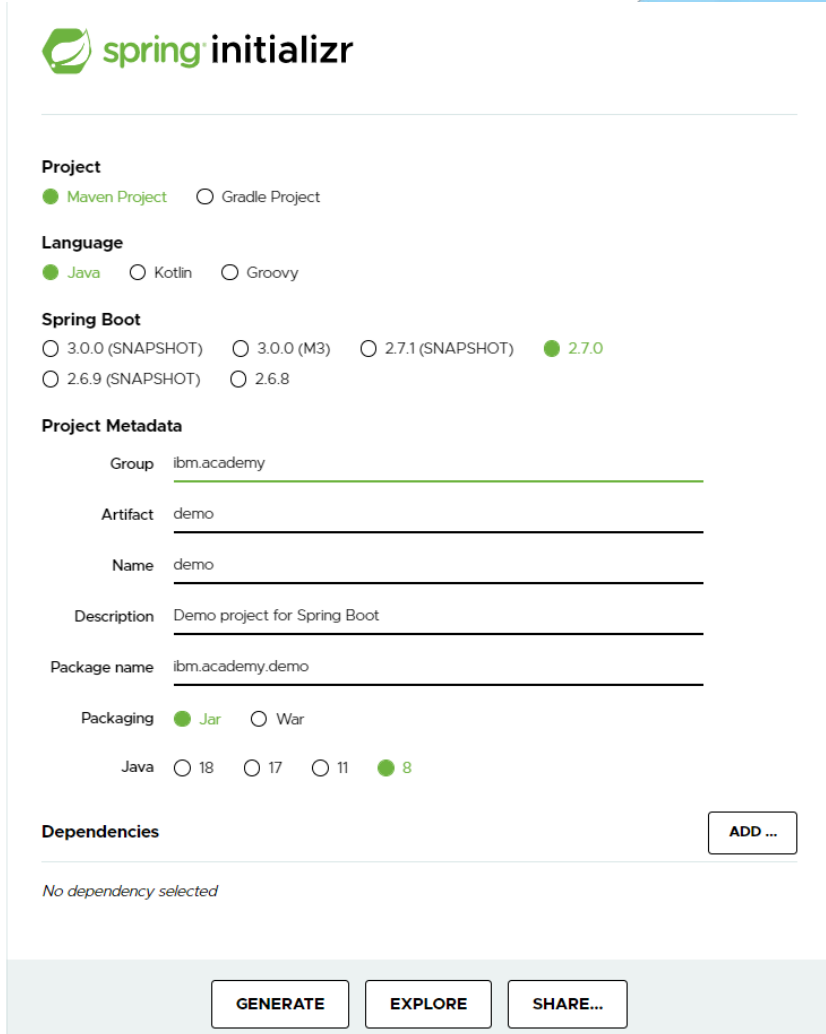
<https://start.spring.io/>



Spring Boot

Demo time

Start Spring



The Spring Initializr web app interface is shown. It features a form for configuring a new Spring Boot project. The form includes sections for Project, Language, Spring Boot, Project Metadata, and Dependencies. The Project section has radio buttons for Maven Project (selected) and Gradle Project. The Language section has radio buttons for Java (selected), Kotlin, and Groovy. The Spring Boot section has radio buttons for various versions, with 2.7.0 selected. The Project Metadata section includes fields for Group, Artifact, Name, Description, and Package name. The Dependencies section has an 'ADD ...' button and a message 'No dependency selected'. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'. A large black arrow points from the 'GENERATE' button to the project structure diagram on the right.

Project

☒ Maven Project ☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.0.0 (SNAPSHOT) ☐ 3.0.0 (M3) ☐ 2.7.1 (SNAPSHOT) ☒ 2.7.0 ☐ 2.6.9 (SNAPSHOT) ☐ 2.6.8

Project Metadata

Group

Artifact

Name

Description

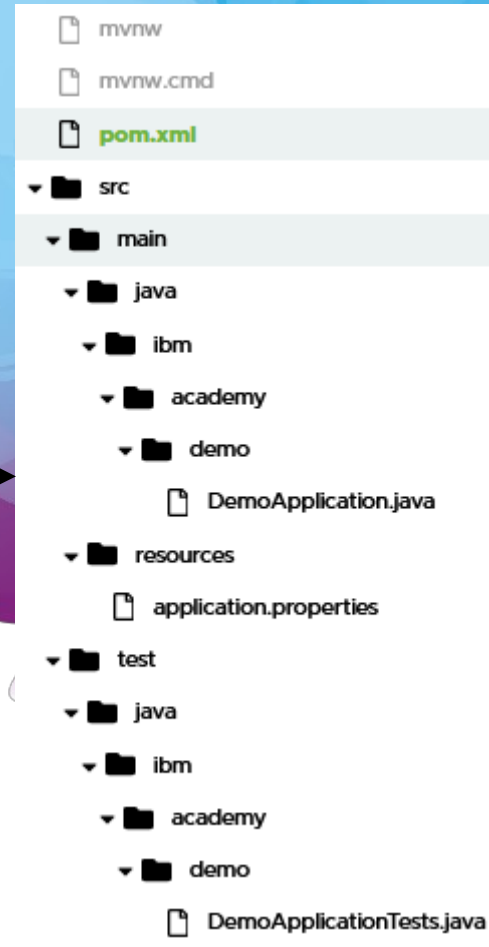
Package name

Packaging ☒ Jar ☐ War

Java ☐ 18 ☐ 17 ☐ 11 ☒ 8

Dependencies

No dependency selected



Tramite la web app (**Spring Initializr**) è possibile creare agilmente la struttura di un progetto Spring Boot.

Questo tool si occupa delle seguenti configurazioni:

- Build tool
- Versione di Spring Boot
- Dipendenze
- Linguaggio
- Metadata di progetto (nome, pacchettizzazione, descrizione...)

IBM Client Innovation Center
Italy

Demo time

@SpringBootApplication

Dove si usa:

- nella classe che contiene il metodo statico *main*.

Fornisce le seguenti funzionalità:

- Abilita la scansione dei componenti Spring e delle configuration class
- La classe annotata diventa essa stessa una configuration class
- Abilita l'autoconfigurazione, ovvero la configurazione automatica (per convenzione) di file JAR presenti nel classpath. (es. Tomcat)

```
@SpringBootApplication
public class BootDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(BootDemoApplication.class, args);
    }
}
```

Demo time

@RestController e @RequestMapping

Dove si usano:

- **@RestController** si applica alla classe che gestisce le richieste del client (abbreviazione di @Controller + @ResponseBody)
- **@RequestMapping** si applica ai metodi della classe annotata con @RestController.

```
@RestController
public class GreetingsController {
    final String hostname = System.getenv().getOrDefault("HOSTNAME", "unknown");

    @RequestMapping(value = "/hello", produces = "text/plain")
    public String sayHello() {
        return "Hello from Spring Boot! " + new java.util.Date() + " on " + hostname + "\n";
    }
}
```

http://localhost:8080/hello

Forniscono le seguenti funzionalità:

- Servono ad esporre le API REST dell'app che possono essere chiamati tramite richieste HTTP ad un determinato path e che rispondono tramite `HttpResponse` (del media-type indicato dall'attributo *produces*)
- Abilitano la classe ad essere autorilevata attraverso la scansione del classpath

Demo time

Random

Sviluppare un servizio di generazione di numeri casuali.



Suggerimento:

<https://docs.oracle.com/javase/8/docs/api/java/util/Random.html>

Demo time

RestTemplate

```
@RestController
public class PersonaController {

    RestTemplate restTemplate = new RestTemplate();

    @RequestMapping(value = "/persona", produces = "text/plain")
    public String persona() {

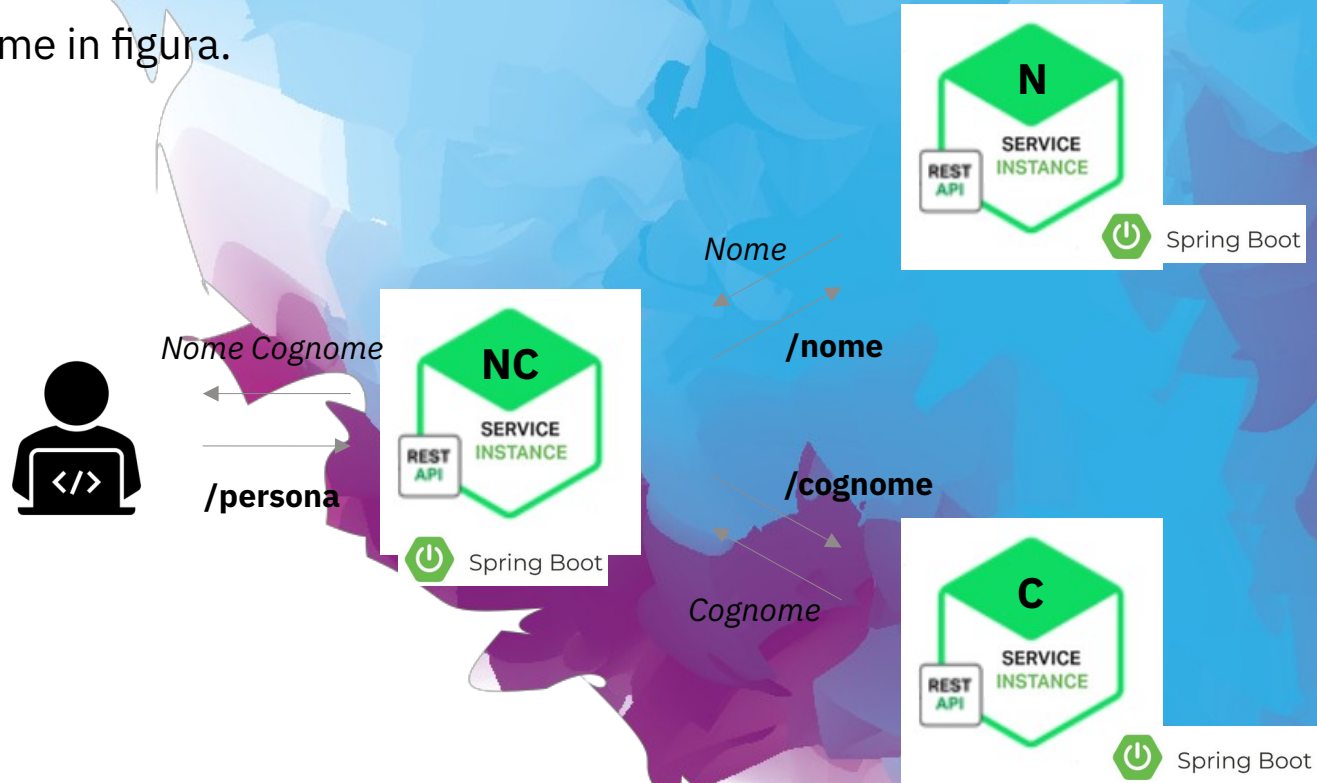
        String url = "http://nome:8080/nome";
        ResponseEntity<String> nomeRes = restTemplate.getForEntity(url, String.class);
    }
}
```

- È un client REST fornito da Spring che mette a disposizione dei metodi per “consumare” dati rest
- Viene pertanto utilizzato per effettuare chiamate verso altri servizi REST e convertire le risposte in oggetti Java che mappano esattamente il formato dei dati della risposta (in questo caso una stringa ritornata attraverso un’API che ritorna un media-type = text/plain)

Demo time

Nome Cognome

Sviluppare un set di servizi come in figura.



Informazioni Utili



Giorgio Dramis

Email: 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
- 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



Introduzione ai microservizi



Caratteristiche



Benefici introdotti



Punti di attenzione



Demo time ed informazioni utili

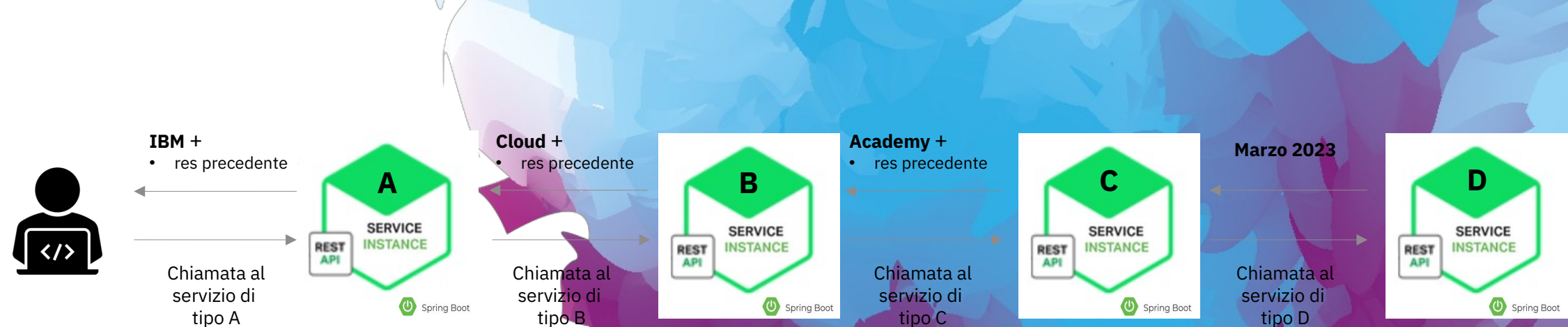


Esercizi, domande e risposte

Introduzione ai microservizi

Esercizio

Sviluppare un esempio di service chaining come mostrato in figura.





Grazie

Giorgio Dramis

IBM Client Innovation Center
Italy