Micro e Nano servizi

Una discussione recente introduce una distinzione tra microservizi e nanoservizi, che nasce dall'evoluzione delle architetture distribuite e dei paradigmi di sviluppo software. Mentre i microservizi sono ormai una pratica consolidata, i nanoservizi rappresentano un concetto più estremo e controverso.

In ogni caso, sia i microservizi che i nanoservizi implicano forme di interazione a scambio di messaggi, che rientrano nel tema *ComputareComunicare*.

Microservizio

Un microservizio è un'unità funzionale indipendente che ha il compito di gestire una specifica funzionalità di business. I microservizi sono progettati per essere altamente modulari e autonomi, con l'obiettivo di sviluppare, distribuire e scalare i componenti dell'applicazione in modo indipendente.

Nel nostro contesto, possiamo pensare ad un attore qak *QakActors24* come una forma di *Nanoservizio*, che può essere usata al posto o in aggiunta ai *POJO* (**Plain Old Java Object**) per implementare un'architettura a microservizi.

Caratteristiche chiave dei microservizi:

- *Autonomia*: Ogni microservizio può essere sviluppato, testato, distribuito e scalato indipendentemente dagli altri.
- *Funzionalità ben definite*: Ogni microservizio si occupa di una funzione di business chiara, come la gestione degli utenti, dei pagamenti, delle notifiche, ecc.
- *Comunicazione via rete*: I microservizi comunicano tra di loro generalmente tramite API (REST, gRPC) o messaggistica asincrona.
- *Gestione di stati*: Molti microservizi mantengono uno stato interno o accedono a un database proprio.
- *Manutenzione e aggiornamento*: I microservizi possono essere aggiornati o modificati senza impattare l'intero sistema.

I Microservizi:

- Sono adatti per applicazioni di medie o grandi dimensioni che richiedono una chiara separazione delle responsabilità.

- Ottimi quando si ha bisogno di scalabilità indipendente delle funzionalità.
- Funzionano bene quando il team di sviluppo è suddiviso in piccole squadre, ognuna responsabile di un servizio specifico.

Vantaggi dei microservizi

Alcuni vantaggi dei microservizi includono:

- *Scalabilità*: I microservizi sono progettati per essere scalabili in modo indipendente. Ciò significa che è possibile scalare solo i servizi che richiedono più risorse senza dover scalare l'intero sistema. Questo può migliorare le prestazioni complessive e ridurre i costi.
- *Manutenibilità*: Scomporre un'applicazione in microservizi rende più facile gestire e mantenere il codice. Ogni servizio può essere sviluppato, testato e distribuito indipendentemente dagli altri, semplificando il processo di sviluppo e riducendo la complessità del codice.
- *Flessibilità tecnologica*: Utilizzando microservizi, è possibile utilizzare tecnologie diverse per ogni servizio, in base alle esigenze specifiche del servizio. Questo consente di adottare nuove tecnologie più facilmente e di sfruttare al meglio gli strumenti più adatti per ogni compito.
- *Isolamento dei problemi*: In un'architettura a microservizi, un problema in un servizio non influisce sugli altri servizi. Questo isolamento può facilitare il debugging e la risoluzione dei problemi, limitando l'impatto delle modifiche o dei guasti.
- *Distribuzione e scalabilità orizzontale*: I microservizi possono essere distribuiti su più server o contenitori, consentendo una scalabilità orizzontale più efficace. Ciò significa che è possibile gestire carichi di lavoro più elevati aggiungendo semplicemente nuove istanze di un determinato servizio.
- *Sviluppo distribuito*: L'architettura a microservizi è ben adattata allo sviluppo distribuito. Diversi team possono lavorare su diversi servizi contemporaneamente, riducendo il time-to-market e consentendo una maggiore flessibilità nell'organizzazione del lavoro.
- *Riuso del codice*: Suddividere un'applicazione in microservizi può favorire il riuso del codice. I servizi possono essere progettati in modo da essere riutilizzabili in più contesti, migliorando l'efficienza dello sviluppo e riducendo la duplicazione del codice.
- *Resilienza*: Un'architettura a microservizi ben progettata può essere più resiliente rispetto a un'architettura monolitica. I servizi possono essere proget-

tati per gestire i fallimenti in modo isolato e per riprendersi autonomamente dai guasti, migliorando la disponibilità complessiva del sistema.

Sostituire un POJO con un servizio o un microservizio può portare a una maggiore flessibilità, scalabilità, manutenibilità e resilienza del sistema, consentendo di gestire meglio i requisiti in continua evoluzione e di adattarsi più facilmente alle esigenze del business.

Rispetto ai POJO in un'architettura monolitica, i microservizi richiedono un cambiamento radicale nella progettazione, implementazione e gestione dell'infrastruttura.

Motivazioni all'uso dei microservizi

Le motivazioni che spingono le aziende industriali verso i microservizi sono molteplici e spesso legate alla necessità di gestire sistemi complessi e scalabili, migliorare la resilienza, accelerare l'innovazione, e supportare la trasformazione digitale. L'architettura a microservizi offre la flessibilità necessaria per rispondere a queste sfide, permettendo alle aziende di adattarsi più facilmente alle evoluzioni del mercato e alle esigenze operative.

1. Scalabilità e Prestazioni
   - *Motivazione*: Le aziende industriali spesso gestiscono sistemi di produzione e automazione che richiedono una scalabilità elevata per supportare un numero crescente di sensori, macchine, impianti o clienti. L'architettura monolitica può diventare inefficiente quando i sistemi crescono in termini di carico e complessità.
   - *Microservizi*: Permettono di scalare singoli componenti (servizi) indipendentemente, senza dover scalare l'intera applicazione. Ciò è utile quando alcuni servizi, come l'elaborazione dei dati dei sensori o l'analisi in tempo reale, richiedono più risorse rispetto ad altri. Questo riduce i costi e ottimizza l'uso delle risorse.
2. Manutenibilità e Aggiornamento
   - *Motivazione*: I sistemi industriali tendono a diventare molto complessi e possono richiedere aggiornamenti continui per adattarsi a nuove tecnologie o standard di settore. La manutenzione di una grande applicazione monolitica può essere difficile e costosa, con rischi di downtime elevati.

- *Microservizi*: Consentono di aggiornare e mantenere singoli componenti senza interrompere l'intero sistema. Le aziende possono effettuare modifiche e aggiornamenti a singoli servizi in maniera più rapida e con minori rischi, migliorando la produttività e riducendo il rischio di guasti durante gli aggiornamenti.

3. Sviluppo autonomo e Time-to-Market
   - *Motivazione*: In ambito industriale, le aziende spesso necessitano di sviluppare nuove funzionalità in tempi rapidi per rispondere a nuove richieste del mercato, integrare nuovi macchinari o tecnologie, o offrire nuove soluzioni ai clienti. Gli approcci monolitici rallentano lo sviluppo perché richiedono la coordinazione tra più team su un'unica base di codice.
   - *Microservizi*: Permettono ai team di sviluppo di lavorare in parallelo su servizi separati, con cicli di sviluppo indipendenti. Questo riduce i tempi di rilascio delle nuove funzionalità, migliorando il time-to-market delle innovazioni, e consente alle aziende di rispondere più rapidamente alle esigenze del settore.
4. Flessibilità Tecnologica
   - *Motivazione*: Le aziende industriali spesso utilizzano una vasta gamma di tecnologie e strumenti, che vanno dai sistemi legacy agli impianti moderni basati su IoT (Internet of Things) o AI. Una singola tecnologia o piattaforma potrebbe non essere adatta per tutti i casi d'uso.
   - *Microservizi*: Consentono di utilizzare diversi stack tecnologici per servizi diversi, permettendo alle aziende di scegliere la tecnologia più adatta per ogni componente del sistema. Questo offre una grande flessibilità e consente di integrare più facilmente nuovi strumenti o tecnologie all'interno dell'architettura aziendale.
5. Resilienza e Tolleranza ai Guasti
   - *Motivazione*: Le aziende industriali richiedono un'alta affidabilità nei loro sistemi, poiché anche brevi interruzioni nei processi produttivi possono causare perdite significative. In un'applicazione monolitica, un errore in un singolo componente può bloccare l'intero sistema.
   - *Microservizi*: Migliorano la resilienza poiché i servizi sono isolati e l'errore di un singolo servizio non comporta necessariamente il fallimento dell'intero sistema. Questo approccio permette di costruire architetture più robuste e con capacità di recupero automatico (self-healing), minimizzando l'impatto di eventuali guasti.
6. Supporto per la Digital Transformation

- *Motivazione*: La trasformazione digitale è una priorità per molte aziende industriali, che puntano a modernizzare i loro impianti e processi tramite automazione, IoT, AI e analisi avanzata dei dati. Un'architettura monolitica può essere difficile da adattare a queste esigenze, limitando l'integrazione di nuove tecnologie.
- *Microservizi*: Facilitano l'integrazione di nuovi paradigmi tecnologici come l'IoT, l'Industria 4.0 e la manutenzione predittiva. Offrono un'architettura flessibile per gestire grandi quantità di dati in tempo reale e permettono alle aziende di implementare strategie di trasformazione digitale in modo più efficace.

7. Facilità di Deployment e Automazione
   - *Motivazione*: L'automazione e il deployment continuo sono fondamentali per aziende che gestiscono complessi sistemi produttivi distribuiti in diverse località. La configurazione e il deployment di una grande applicazione monolitica possono essere lenti e complessi.
   - *Microservizi*: Si adattano bene agli approcci DevOps e CI/CD (Continuous Integration/Continuous Deployment). Permettono di automatizzare il deployment di singoli servizi e di gestire in modo efficiente l'infrastruttura distribuita tramite container e orchestratori come Docker e Kubernetes. Ciò accelera il rilascio di nuove versioni e migliora l'efficienza operativa.
8. Conformità e Regolamentazioni
   - *Motivazione*: Le industrie devono spesso rispettare normative complesse (es. norme di sicurezza o ambientali) che possono variare a seconda della regione o del mercato. L'aggiornamento di un'applicazione monolitica per garantire la conformità in più giurisdizioni può essere difficile e rischioso.
   - *Microservizi*: Consentono alle aziende di implementare logiche di conformità e requisiti regolatori specifici solo nei servizi necessari, senza dover modificare l'intero sistema. Questo facilita l'adattamento delle applicazioni alle normative locali e alle regole settoriali.
9. Monitoraggio e Analisi in Tempo Reale
   - *Motivazione*: Il monitoraggio in tempo reale è cruciale per le aziende industriali che gestiscono processi produttivi, dove il rilevamento immediato di guasti o inefficienze può prevenire danni o interruzioni.
   - *Microservizi*: Facilitano il monitoraggio dettagliato di ogni servizio e componente, offrendo una maggiore visibilità sui processi. I microservizi pos-

sono essere monitorati in modo indipendente, permettendo di analizzare i problemi in tempo reale e migliorare le prestazioni del sistema globale.

10. Gestione delle Dipendenze e Interoperabilità
    - *Motivazione*: In ambito industriale, le applicazioni devono spesso integrarsi con una varietà di sistemi legacy, impianti di produzione, robotica, sensori e strumenti di terze parti. Integrare queste dipendenze in un'applicazione monolitica può essere complesso e richiede un approccio monolitico pesante.
    - *Microservizi*: Ogni servizio può gestire la propria integrazione con strumenti e tecnologie specifiche, riducendo la complessità e migliorando l'interoperabilità tra sistemi diversi. L'uso di API ben definite facilita l'integrazione e la comunicazione tra i vari componenti.

Microservizi e IOT

I microservzi nascono storicamente nel cloud, ma l'architettura a microservizi è vantaggiosa per applicazioni IoT per diverse ragioni:

- *Scalabilità*: le applicazioni IoT devono spesso gestire un numero elevato di dispositivi e dati. I microservizi consentono di scalare i componenti in modo indipendente, garantendo una gestione efficiente del carico di lavoro.
- *Manutenibilità e aggiornamenti indipendenti*: i microservizi permettono agli sviluppatori di aggiornare, sostituire o aggiungere nuove funzionalità senza compromettere l'intera applicazione. Ciò è particolarmente utile nelle reti IoT in continua evoluzione.
- *Resilienza*: l'architettura a microservizi riduce l'impatto di guasti isolando i problemi a un singolo servizio, evitando che un errore comprometta l'intero sistema.
- *Flessibilità tecnologica*: nei progetti IoT, diverse parti del sistema possono richiedere tecnologie specifiche (ad esempio, un protocollo di comunicazione o un database specifico). I microservizi consentono di scegliere la tecnologia migliore per ciascun componente.
- *Deployment distribuito*: nei sistemi IoT, ci può essere un'infrastruttura distribuita (con dispositivi geograficamente distanti). I microservizi facilitano il deployment distribuito e la gestione del traffico di rete.

Problematiche dei microservizi

I microservizi offrono molti vantaggi in termini di modularità, scalabilità e indipendenza nello sviluppo, ma introducono anche diverse problematiche e sfide rispetto all'approccio monolitico tradizionale basato su *POJO*. Ecco alcune di queste problematiche:

1. *Gestione della complessità*
   - Con i POJO (monolitico): In un'architettura monolitica, tutti i componenti risiedono nello stesso processo, il che semplifica la gestione della complessità interna. Le interazioni tra oggetti e componenti avvengono attraverso semplici chiamate di metodo e la gestione degli errori è più lineare.
   - Con i microservizi: La complessità aumenta perché ogni microservizio è un'entità separata, e la comunicazione tra loro deve avvenire tramite chiamate di rete (HTTP, gRPC, messaggistica, ecc.). Questo introduce problemi di latenza, gestione degli errori distribuiti e coordinamento delle dipendenze tra i servizi.
2. *Comunicazione distribuita*
   - Con i POJO: Le interazioni tra i componenti avvengono tramite invocazioni di metodi locali, che sono rapide e sicure, senza il rischio di fallimenti di rete.
   - Con i microservizi: La comunicazione tra i microservizi avviene tramite la rete, il che comporta latenza, errori di rete, timeout, e problematiche legate alla serializzazione/deserializzazione dei dati. Questo richiede anche la gestione delle API e protocolli di comunicazione, introducendo ulteriori livelli di complessità. Inoltre, è necessario implementare meccanismi di retry, circuit breaker, e gestione delle fallimenti parziali.
3. *Consistenza dei dati*
   - Con i POJO: I dati sono gestiti in un database comune all'interno di un'applicazione monolitica. È possibile garantire consistenza transazionale in modo relativamente semplice utilizzando **transazioni ACID**.
   - Con i microservizi: Ogni microservizio potrebbe avere il proprio database, il che rende difficile garantire la consistenza transazionale globale (es. le transazioni distribuite). I microservizi spesso adottano modelli di consistenza eventuale attraverso la propagazione asincrona degli eventi o l'uso di saghe. Questo introduce complessità nella gestione dei dati e nella coerenza dei sistemi.
4. *Gestione delle dipendenze e orchestrazione*

- Con i POJO: Tutte le classi e i componenti risiedono nello stesso ambiente di runtime, e le dipendenze tra di loro possono essere gestite attraverso un gestore delle dipendenze (come Maven o Gradle).
- Con i microservizi: Ogni microservizio è un'entità separata con le sue dipendenze. Questo può portare a problemi di compatibilità delle versioni tra servizi diversi. Inoltre, è necessario un sistema di orchestrazione per coordinare l'avvio, lo scaling, la gestione e l'aggiornamento di tutti i servizi, spesso con l'aiuto di strumenti come Kubernetes o Docker.

5. *Gestione dello stato*
   - Con i POJO: Lo stato viene gestito in memoria o attraverso il database centralizzato, ed è facile mantenere lo stato condiviso tra diversi componenti dell'applicazione.
   - Con i microservizi: I microservizi dovrebbero essere stateless o mantenere uno stato isolato, poiché la gestione dello stato distribuito è complessa. Se un microservizio ha bisogno di gestire lo stato, deve fare affidamento su sistemi di storage distribuito o cache distribuita, il che introduce ulteriori complessità e sfide nella coerenza dello stato.
6. *Deployment e monitoraggio*
   - Con i POJO: In un'applicazione monolitica, il deployment è semplice: si distribuisce un singolo artefatto (es. un file .war o .jar) su un server. Il monitoraggio è anch'esso più semplice perché tutte le informazioni sullo stato del sistema sono in un unico posto.
   - Con i microservizi: Ogni microservizio ha il proprio ciclo di vita di deployment e potrebbe essere distribuito indipendentemente. Questo richiede orchestrazione automatizzata, gestione delle versioni e coordinamento per evitare downtime. Inoltre, il monitoraggio diventa più complesso: è necessario tracciare il comportamento di ciascun microservizio, le loro dipendenze e le interazioni tra loro, spesso utilizzando strumenti come Prometheus, ELK stack o Jaeger per il tracing distribuito.
7. *Gestione degli errori e tolleranza ai guasti*
   - Con i POJO: Gli errori e le eccezioni possono essere gestiti in modo centralizzato, e in un ambiente monolitico è più facile fare rollback completo o ripristinare uno stato coerente in caso di guasti.
   - Con i microservizi: Gli errori possono verificarsi in punti diversi del sistema, e devono essere gestiti a livello di ogni microservizio. Il fallimento di un servizio può non impattare l'intero sistema, ma i fallimenti parziali (ad es. un servizio chiave che non risponde) devono essere gestiti atten-

tamente per evitare che il sistema vada in crash o degradi le performance. Tecniche come il circuit breaker, bulkhead pattern e retry sono essenziali per garantire la resilienza dell'architettura.

8. *Test e debugging*
   - Con i POJO: Il testing di un'applicazione monolitica è relativamente semplice. Si possono eseguire test unitari e di integrazione sull'intero sistema in un unico ambiente. Il debugging è diretto, poiché tutte le classi e i componenti sono parte dello stesso processo.
   - Con i microservizi: Il testing diventa più complicato, poiché è necessario testare non solo i singoli servizi (test unitari) ma anche le interazioni tra di essi (test di integrazione e contratti). Il debugging di un sistema distribuito è molto più difficile a causa della natura asincrona e della comunicazione tra processi separati. Tecniche come il contract testing e l'uso di strumenti di tracing distribuito diventano fondamentali.
9. *Sicurezza*
   - Con i POJO: La sicurezza può essere gestita a livello di applicazione centralizzata, utilizzando autenticazione e autorizzazione in un unico punto.
   - Con i microservizi: Ogni microservizio potrebbe richiedere un proprio sistema di sicurezza, o deve integrarsi in un framework di sicurezza distribuito (ad es. OAuth, JWT). Questo introduce complessità nella gestione delle credenziali, autenticazione tra microservizi, e nella protezione dei dati durante le comunicazioni. La gestione della sicurezza deve essere implementata in maniera consistente su tutti i servizi.
10. *Sovraccarico infrastrutturale*
    - Con i POJO: Le risorse di calcolo, memoria e archiviazione sono generalmente centralizzate in un'unica applicazione. La gestione delle risorse è relativamente semplice, poiché tutte le componenti condividono lo stesso ambiente.
    - Con i microservizi: Ogni microservizio richiede un proprio ambiente di esecuzione, che può comportare un aumento significativo del carico infrastrutturale. Ad esempio, potrebbero essere necessari più container, database separati, logging distribuito e capacità di gestione delle risorse. Questo può aumentare i costi operativi e richiede un'architettura cloud ben progettata per scalare in modo efficiente.

Progettazione di sistemi a microservizi

- Richardson: *https://microservices.io/index.html*
- Richardson Understanding Microservices: *https://microservices.io/microservices/2020/02/04/jfokus-geometry-of-microservices.html*
- Richardson Microservice architecture patern languages: *https://microservices.io/patterns/index.html*
- Richardson Decomposizione per sottodominio: *https://microservices.io/patterns/decomposition/decompose-by-subdomain.html* :
- Richardson Microservice Architecture: *https://microservices.io/patterns/microservices.html* :
- Api Gateway, Service Discovery, Circuit Breaker, Event Sourcing, CQRS, Saga, etc.
- Il concetto di *aggregator*. Un possibile riferimento *https://medium.com/nerd-for-tech/design-patterns-for-microservices-aggregator-pattern-99c122ac6b73*

1. API Gateway
2. Service catalogs and orhestrators
3. Microservices aggregators
4. Microservices patterns (MicroservicePatternLanguage.pdf Richardson)
5. Circuit Breaker
6. Service Discovery
7. Load Balancing
8. Bulkhead Pattern
9. Retry
10. Saga Pattern
11. Event Sourcing
12. CQRS
13. Distributed Tracing
14. Monitoring and Logging
15. Security
16. Deployment Automation
17. Testing Strategies
18. Continuous Integration/Continuous Deployment (CI/CD)
19. DevOps Culture

Nanoservizio

Un nanoservizio rappresenta un'evoluzione più radicale del concetto di microservizio. La differenza principale sta nella granularità. Un nanoservizio è ancora più piccolo e svolge un compito molto più specifico di un microservizio. Spesso si occupa di un'unica operazione o di una singola responsabilità molto limitata.

Caratteristiche chiave dei nanoservizi:

- *Responsabilità limitatissima*: Un nanoservizio è progettato per svolgere un singolo compito atomico, come la convalida di un campo di input, la gestione di una richiesta HTTP o l'invio di una singola notifica.
- *Dimensioni ridotte*: Il codice e la logica di un nanoservizio sono minimi, riducendo la complessità e rendendoli facilmente manutenibili, ma solo in contesti molto specifici.
- *Estrema modularità*: I nanoservizi portano all'estremo la filosofia della modularità e della separazione delle responsabilità.
- *Sovraccarico di comunicazione*: Poiché i nanoservizi sono estremamente piccoli, richiedono molta più comunicazione tra loro rispetto ai microservizi, il che può aumentare il traffico di rete e la latenza.
- *Manutenzione e distribuzione complicata*: Sebbene siano facili da manutenere, la distribuzione e l'orchestrazione di una vasta quantità di nanoservizi può diventare molto complessa.

I Nanoservizi:

- Potrebbero essere usati in contesti in cui si vogliono implementare operazioni estremamente semplici e modulari, ad esempio in un'architettura serverless.
- Si adattano bene quando si ha bisogno di risposte rapide a eventi specifici, come la convalida di campi o la gestione di eventi atomici.
- La loro complessità di gestione (soprattutto in fase di orchestrazione e comunicazione) li rende meno adatti a progetti di grandi dimensioni o in team più tradizionali.
- possono essere utili per i dispositivi *IoT edge*, ovvero dispositivi con capacità limitate o che devono operare in ambienti con poche risorse (ad esempio, dispositivi a bassa potenza, sensori remoti, ecc.). minimizzando il carico di elaborazione, permettendo operazioni più mirate ed efficienti.

Dal macro al micro

Nel nostro contesto, possiamo pensare ad un attore qak come una forma di Nanoservizio, che può essere usata al posto o in aggiunta ai *POJO* (**Plain Old**

Java Object) per avverci di un'architettura a microservizi in un contesto locale.

Il contesto locale elimina molte problematiche dei microservizi, come la comunicazione via rete, il service discovery, etc,

Un esempio è il progetto **cargoservice** in cui il servizio è 'avvilupato' in un attore qak e non in un *RestController* SpringBoot.

Problematiche usando attori

Ricordando che ogni qak elabora un messaggio per volta, si pongono i seguenti problemi:

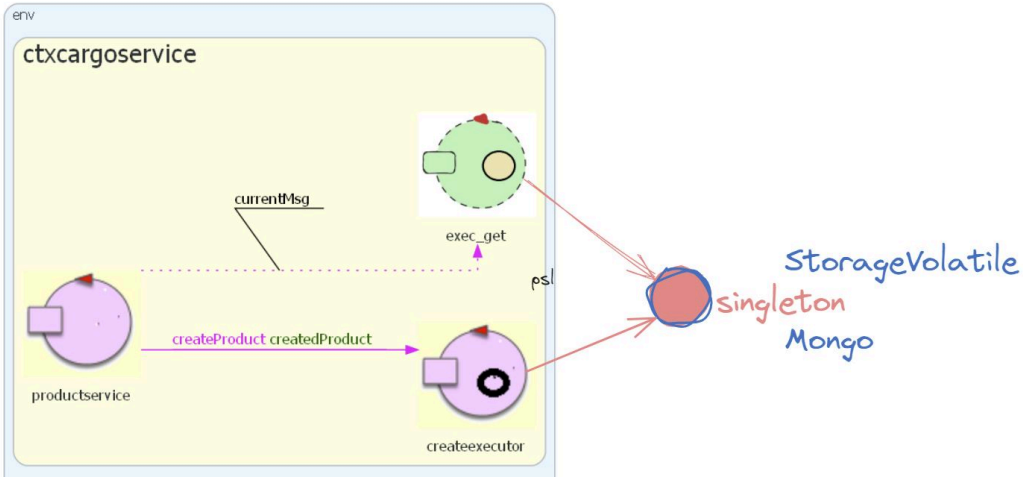
1. come realizzare una operatività asincorna senza ricorrere alla attivazione (low-levl) di nuovi Thread? Ad esempio, come permettere la esecuzione parallela di richieste *getProduct*?

   Risposta: usare il meccanicmo qak **delegateCurrentMsgTo** che crea un nuovo actor per gestire una richiesta **getProduct**.

2. l'operazione **createProduct** del POJO-core deve essere atomica, in modo da impedire la memorizzazione duplicata di uno stesso product. Come fare senza introdurre la specifica (low-level) synchronized sula metodo **createProduct** del POJO-core?

   Risposta: realizzare una esecuzione sequenziale dell'operazione usando il meccansimo qak **delegate createProduct to createexecutor**. In questo modo, ogni richiesta *createProduct* viene eseguita in modo sequenziale, senza bloccare la elabrazione delle **getProduct** da parte del servizio.

Questa impostazione è rappresentata dalla architettura che segue:



Si noti che ogni attore introduce una istanza locale del POJO-core **ProductServiceLogic**. Nasce quindi un nuovo problema:

1. come realizzare un acesso condiviso a uno stesso storage per gli attori **exec_get** e **createexecutor**?

   Risposta: ogni POJO-core deve fare riferimento a uno stesso storage. Nel caso lo storage sia un microservizo come Mongo, la condivions è assicurata. Nel caso di storgae realizzato sulla RAM in cui opera l'applicazione, occorre che ogni POJO-core faccia riferimento a un singleton.

StorageRAMActor

Ecco un motivo per cui anche la storage in RAM potrebbe essere deployed come un microservizio. Lo si potrebbe fare introducendo un attore **StorageRAMActor** che opera in un nuovo contesto qak.

La interazione con lo **StorageRAMActor** potrebbe avvenire con operazioni *publish-subscribe* su broker MQTT, per evitare l'accoppiamewnto stretto di visibilità tra i contesti e la relativa necessità di reciproca attesa all'attivazione.

Inoltre, un utilizatore dello **StorageRAMActor** sarebbe del tutto ignaro che questo microservizio è implementato con i qak, come potrebbe essere il caso di una modifica alla classe **AdapterStorageVolatile** o, meglio, della introduzione di una nuova classe **AdapterRemoteStorageVolatile**.