TURNING MONOLITHS INTO MICROSERVICES: TIPS AND TRICKS

Microservices | Software as a Service (SaaS)

beSharp | 7 Giugno 2019

Nel primo articolo del nostro viaggio in tre parti abbiamo iniziato parlando dei vantaggi di un'applicazione distribuita composta da microservizi rispetto ad una completamente monolitica. In questa seconda parte inizieremo ad avvicinarci a tecniche e suggerimenti per aiutarvi a superare il processo di migrazione in modo più sicuro e consapevole.

DEFINIRE CHIARAMENTE IL DOMINIO DELLA VOSTRA LOGICA DI BUSINESS

Questo è di gran lunga il passo più importante, in quanto indica chiaramente se l**a conoscenza** della logica del proprio dominio è chiara e completa. Infatti, un modo possibile per individuare quali parti della vostra applicazione possono essere convertite in microservizi, è proprio quello di verificare se avete una comprensione completa del dominio di tali servizi, il che significa che sarà più facile per il vostro team di sviluppo **isolarne e migrarne** la logica.

Comprendere il dominio di una feature significa anche che, probabilmente, il suo accoppiamento con il resto dell'applicazione è molto basso, idealmente nullo, facilitando così la sua separazione dall'applicazione principale.

Conoscere il proprio dominio significa definire con chiarezza quali servizi sono **verticali all'applicazione**, o in altre parole, quali servizi sono più importanti e mirano alle esigenze specifiche di una determinata base utenti (definendo così l'obiettivo della propria applicazione) o sono particolarmente strategici per la propria azienda.

I servizi cosiddetti verticali sono spesso costituiti da codice complesso e possono essere molto grandi, ma, come detto in precedenza, non definiamo i microservizi in base alla quantità di codice contenuto, ma mediante la **definizione degli scopi** e **dei confini di un particolare contesto logico**.

DEFINIRE IL VOSTRO OBIETTIVO TECNOLOGICO

Arrivati a questo punto è importante definire chiaramente il nostro **target tecnologico**, nel senso che vogliamo avere ben chiaro che tipo di **piattaforma** o **framework** vogliamo utilizzare per lo sviluppo e per le procedure di CD/CI (es. Serverless Framework e AWS Lambda come ambiente di sviluppo) e che tipo di linguaggio di programmazione è più adatto per migrare quella precisa parte della vostra logica di business.

Avere chiaro il target è cruciale in quanto aiuta a pensare alla **fattibilità di una particolare migrazione**. Dobbiamo anche capire che, anche se la migrazione verso i microservizi è, in generale, una pratica consigliata, non significa che per la vostra particolare applicazione, o parte di essa, sia una buona scelta, ma analizzeremo nel dettaglio questo aspetto più avanti.

In generale durante il brainstorming coinvolto in questa fase è necessario verificare i costi in termini di **migrazione del codice** e delle **infrastrutture** e il **valore strategico** rispetto allo **sviluppo di nuove funzionalità**.

DA DOVE COMINCIARE A SCOMPORRE UN MONOLITE: STRANGLER PATTERN

Finora abbiamo parlato di preparativi, iniziamo ora a spiegare come scomporre efficacemente la nostra infrastruttura monolitica e quali caratteristiche deve avere un microservizio per essere considerato tale una volta estratto con successo.

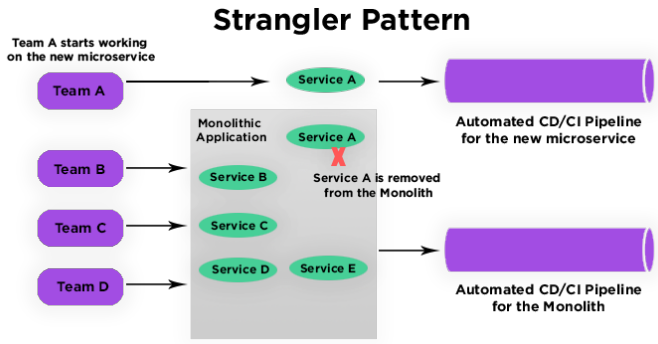
Quando pensiamo di passare da un approccio monolitico ad un ecosistema di microservizi possiamo intraprendere due possibili strade: a) riscrivere il codice da zero utilizzando il nuovo paradigma o b) migrare da quello vecchio.

Iniziare a riscrivere l'intera applicazione da zero in un'unica passata non è generalmente una buona scelta perché:

- Può portare via molto tempo di sviluppo.
- Può portare con sé **debito tecnico** durante il processo di riscrittura.
- L'applicazione non può essere utilizzata fino a quando la riscrittura non è completa.
- Non possiamo davvero concentrarci sulle nuove funzionalità, in quanto è necessario che la logica dell'applicazione sia completamente trasferita, causando potenziali ritardi per il team di business.

Quindi l'opzione migliore è, come abbiamo discusso fino a questo punto, estrarre e convertire l'applicazione **passo dopo passo**.

Questo approccio si chiama **Strangler Pattern**, un modo per **trasformare in modo incrementale la vostra applicazione monolitica in microservizi sostituendo le funzionalità una alla volta**. Una volta codificata la nuova funzionalità, il vecchio componente viene "**strangolato**", sostituito e infine **disattivato**.

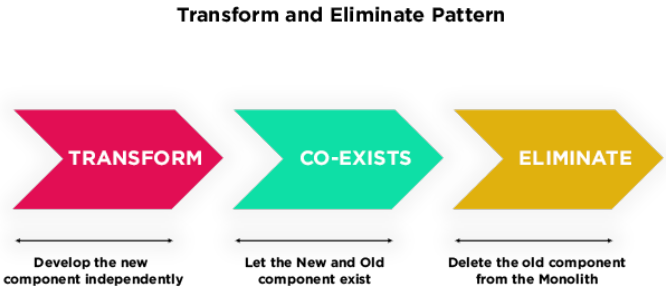


Questo approccio è molto efficace per una serie di ragioni:

- Consente uno **sforzo flessibile in termini di tempo**, che può essere adattato all'attuale situazione di sviluppo dei vostri team.
- L'applicazione può ancora essere utilizzata perché si sta migrando solo una piccola parte di essa alla volta.
- È possibile continuare a sviluppare nuove funzionalità creandole direttamente come nuovi microservizi.

Per implementare lo Strangler Pattern, è possibile seguire tre passi: **Trasformare**, **Coesistere** (la coesistenza è necessaria per testare la nuova funzionalità con parte della base utenti), ed **Eliminare**.

Vale la pena spendere due parole in più sulla fase di Coesistenza in quanto richiede uno sforzo ed una analisi accurata da parte dei vostri team di sviluppo, necessari a mantenere entrambe le code base e i servizi di supporto all'utente fino a quando il componente facente parte del monolite non viene decommissionato. In questa fase è bene pianificare le operazioni in anticipo per evitare brutte sorprese.



Per iniziare ad estrarre gli elementi fondamentali della vostra logica di business, soprattutto se siete nuovi a questo modello di sviluppo, scegliete quelli che hanno:

- Un dominio chiaro, un ambito preciso.
- Nessun tipo di base dati o una base dati che può essere facilmente isolata ed essere resa indipendente.
- Nessun accoppiamento forte con il resto del vostro codice applicativo.

Inoltre:

- Un componente che ha una buona copertura lato testing e poco o nessun debito tecnico ad esso associato, è un buon candidato (se la logica del dominio è sufficientemente chiara).

- Se un componente ha requisiti di scalabilità, sceglietelo.
- Se c'è un componente che ha forti esigenze di business e deve essere distribuito molto più regolarmente, si può iniziare con quel componente.

Infine, verificare se esistono parti di codice che possono essere facilmente riutilizzabili in altri progetti, quindi per loro stessa natura **atomiche** e **riutilizzabili:** ottimi candidati per la promozione a microservizi**.**

Ci sono strumenti che possono aiutare nella processo di suddivisione, come quelli per la **Social code analysis** che arricchisce la nostra comprensione della qualità del codice sovrapponendo il comportamento di uno sviluppatore con l'**analisi strutturale del codice**.

Utilizzano informazioni derivanti dai **Sistemi di Controllo di Versione**. Uno di questi è **CodeScene.**

COSA RENDE UN MICROSERVICE TALE?

Per capire se si sta creando un vero e proprio microservizio bisogna verificare se esso aderisce a specifiche **proprietà di isolamento** sia a **livello infrastrutturale** che **logico**.

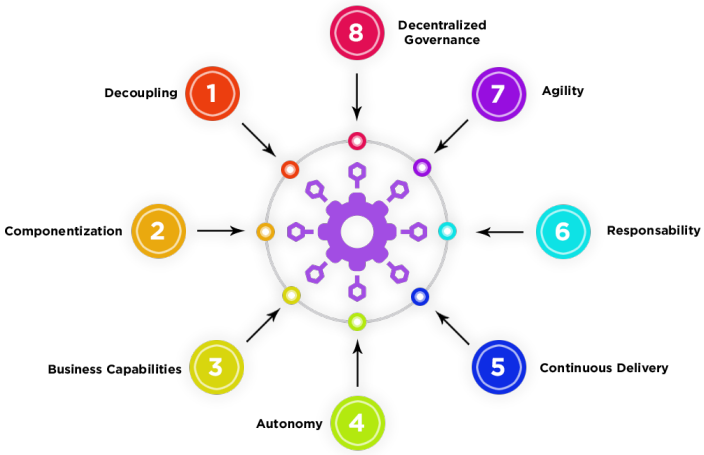
Un microservizio deve essere **indipendente** e **atomico** (può esistere e fornire un servizio in autonomia, possibilmente senza sapere nulla dell'intera applicazione) a livello di **sviluppo**, **test**, **distribuzione**, **monitoraggio**, **debug** e **recovery**. Questo assicura che possiamo svilupparlo, gestirlo e distribuirlo in modo sicuro **senza interrompere** il **ciclo di vita** dell'applicazione principale.

Un microservizio deve essere indipendente anche a **livello di API**, **Business Logic** e **Base Dati**, per garantire l'**isolamento del codice** (l'accesso alle funzionalità del servizio avviene solo tramite API).

Ciò significa che è buona norma anche preparare e definire un database indipendente per ogni microservizio invece di un'unica base dati condivisa.

Anche se questa operazione comporta la necessità di definire dei **descrittori specifici**, necessari per rendere i diversi microservizi parlanti tra loro in termini di dati, si evita un possibile **single point of failure derivante** dall'avere un unico database (un malfunzionamento di quest'ultimo può compromettere l'intero set di microservizi invece di uno solo).

Come regola d'oro inoltre, verificare sempre che **nessun microservizio dipenda dal monolite**.



In questa seconda parte del nostro viaggio in tre parti su come scomporre un'applicazione monolitica, abbiamo iniziato ad analizzare in profondità alcune tecniche e idee che aiutano il team di sviluppo e di business a decidere quando e come sia possibile dividere il codice legacy in microservizi atomici e indipendenti, come si possa affrontare il ciclo di vita della migrazione e in generale quali proprietà deve avere un microservizio per essere considerato tale.

Nel prossimo articolo concluderemo il nostro viaggio, descrivendo come iniziare a codificare efficacemente i vostri microservizi, come affrontare codice complesso ad alto valore intellettuale e anche quando passare ad un ecosistema a microservizi possa non essere una buona scelta.

Rimanete con noi!

<< Leggi la prima parte | Leggi la terza parte >>

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it