Martin Fowler - Microservices

Source: https://martinfowler.com/articles/microservices.html

Introduzione

L'espressione "*Architettura a microservizi*" è sempre più comune tra gli sviluppatori di applicazioni enterprise per descrivere un metodo di progettazione delle applicazioni come **insiemi di servizi eseguibili indipendentemente** (ogni servizio esegue su un processo indipendente), **che comunicano tra loro grazie a meccanismi di comunicazione leggeri** (solitamente attraverso API HTTP).

Architetture a confronto: Microservizi vs Monolitiche

Un'applicazione monolitica è **progettata e costruita per essere una singola unità in esecuzione**. Solitamente un'applicazione web monolitica è divisa in 3 parti:

1. interfaccia utente (pagine web);
2. database;
3. applicazione server.

A sua volta, l'applicazione server:

- gestisce le richieste HTTP;
- esegue la business logic dell'applicazione;
- carica e aggiorna dati dal/nel database;
- seleziona e popola le pagine web inviate al browser dell'utente.

Nelle applicazioni monolitiche si cerca di organizzare le **componenti** del sistema sfruttando i costrutti fondamentali dei linguaggi di programmazione:

- funzioni;
- classi;
- namespace o package.

Per aumentare la **disponibilità** delle applicazioni monolitiche si usa **replicare istanze**

dell'applicazione in molteplici server, avendo un server di load balancing che bilancia il traffico nel modo più appropriato.

Tra i difetti delle applicazioni monolitiche si possono evidenziare:

- modifiche a una piccola parte all'applicazione richiedono la **ricompilazione** e la **ridistribuzione** dell'applicazione;
- all'accrescere della complessità dell'applicazione aumenta anche la **difficoltà nel mantenere le modifiche isolate ai moduli di competenza**;
- scalare l'applicazione richiede l'esecuzione di istanze multiple della stessa applicazione, ignorando di fatto eventuali requisiti di efficienza (solitamente alcune componenti del sistema non richiedono un aumento di throughput[1]).

Per lo stile architetturale a microservizi non esistono definizioni formali, tuttavia è possibile ricavare delle caratteristiche comuni nei progetti che sono diventati nel tempo esempi di *best-practice*. Non tutte le architetture a microservizi hanno tutte le caratteristiche elencate in seguito, ma ci si aspetta che la maggior parte delle architetture esibisca quante più caratteristiche possibili.

L'aspetto cruciale delle architetture a microservizi verte sulla definizione di **componente**: la definizione comunemente accettata è quella di "*unità di software che è indipendentemente aggiornabile e sostituibile in un sistema*". Le architetture a microservizi usano i **servizi** per realizzare tale definizione di componente. A titolo di confronto con gli approcci di sviluppo tradizionali è possibile introdurre la nozione di **libreria**. Le librerie sono componenti insiti in un'applicazione tanto da risiedere nello stesso spazio di memoria dell'applicazione e che per essere invocate richiedono una chiamata di funzione in memoria. I servizi sono componenti che vivono nel sistema come processi separati, sfruttando vari tipi di comunicazione interprocesso: richieste web, chiamate di funzione remote (RPC[2]).

Il vantaggio principale dei servizi rispetto alle librerie consiste nel fatto che i **servizi sono rilasciabili indipendentemente dal sistema**. Data la natura dell'architettura a microservizi, **modifiche a un singolo servizio comportano il rilascio di una nuova versione solamente per quel servizio e non dell'intera applicazione**. Una buona architettura a microservizi quindi mira a progettare e implementare servizi che circoscrivano chiaramente il loro scopo.

L'uso di servizi come componenti consente inoltre di rendere esplicita l'interfaccia dei componenti[3]. Spesso solamente la documentazione e la disciplina prevengono usi impropri di una componente da parte di uno sviluppatore esterno, rischiando di causare un alto accoppiamento[4] tra componenti. **I servizi facilitano il rispetto delle interfacce pubblicate attraverso l'uso di meccanismi di chiamate remote esplicite**.

Il difetto che si annovera all'uso di servizi come componenti risiede nell'utilizzo di **chiamate remote** per la comunicazione tra servizi: esse **richiedono più risorse rispetto alle chiamate di funzione intraprocesso** e quindi è necessario progettare le API di ciascun servizio rivolgendo maggiore attenzione all'aspetto prestazionale delle stesse.

Note

1: Throughput indica la capacità di un canale di comunicazione di processare o trasmettere dati in uno specifico periodo di tempo. É una misura di produttività.

2: RPC (Remote Procedure Call) si riferisce all'invocazione di una funzionalità pubblicata da un altro elaboratore rispetto a quello in cui l'applicazione è in esecuzione.

3: Con interfaccia Fowler intende l'insieme di funzionalità offerte da una componente disponibile per l'utilizzo in altri software.

4: L'accoppiamento indica il grado di dipendenza che ciascuna componente del sistema ha con le altre componenti. Un basso grado di accoppiamento è indice di un sistema ben strutturato, in cui le componenti svolgono le rispettive funzionalità indipendentemente dalle altre.