PROGRAMMAZIONE ORIENTATA AI MICROSERVIZI

Architettura dei microservizi

Tommaso Nanu
tommaso.nanu@alad.cloud

UNIVERSITÀ DI PARMA
DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE
Corso di Laurea in Informatica

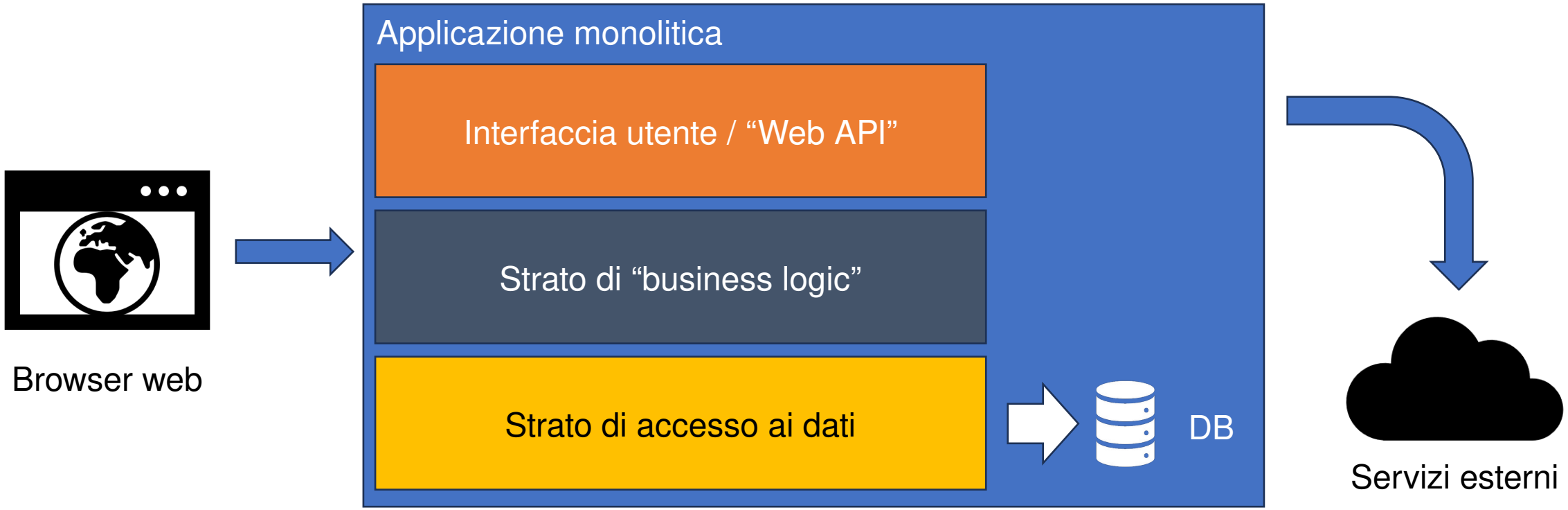
Architettura monolitica

"C’era una volta in un server lontano lontano"

Architettura monolitica

Sviluppo di applicativi software in cui una singola applicazione viene costruita come un’unica unità

Architettura monolitica



Caratteristiche

- Tutte le sue funzioni all'interno di un unico blocco
- Logicamente autonomo
- Componenti interni difficilmente separabili
  - Una modifica comporta la ricompilazione dell'intero codice
  - Non è possibile fare il deploy di una singola funzionalità modificata

Architettura monolitica

Vantaggi

- Debug più agevole
- Design più semplice
  - Progettazione da zero più semplice
  - Nessun componente disaccoppiato
- Meno overhead
- Atomicità nativa garantita dal DB

Svantaggi

- No scalabilità dei singoli componenti
- Complessità maggiore dell'intera applicazione
- Avanzamento tecnologico lento
- No continuous delivery
- Intersezione delle competenze (manutenibilità dispendiosa)
  - Poca flessibilità

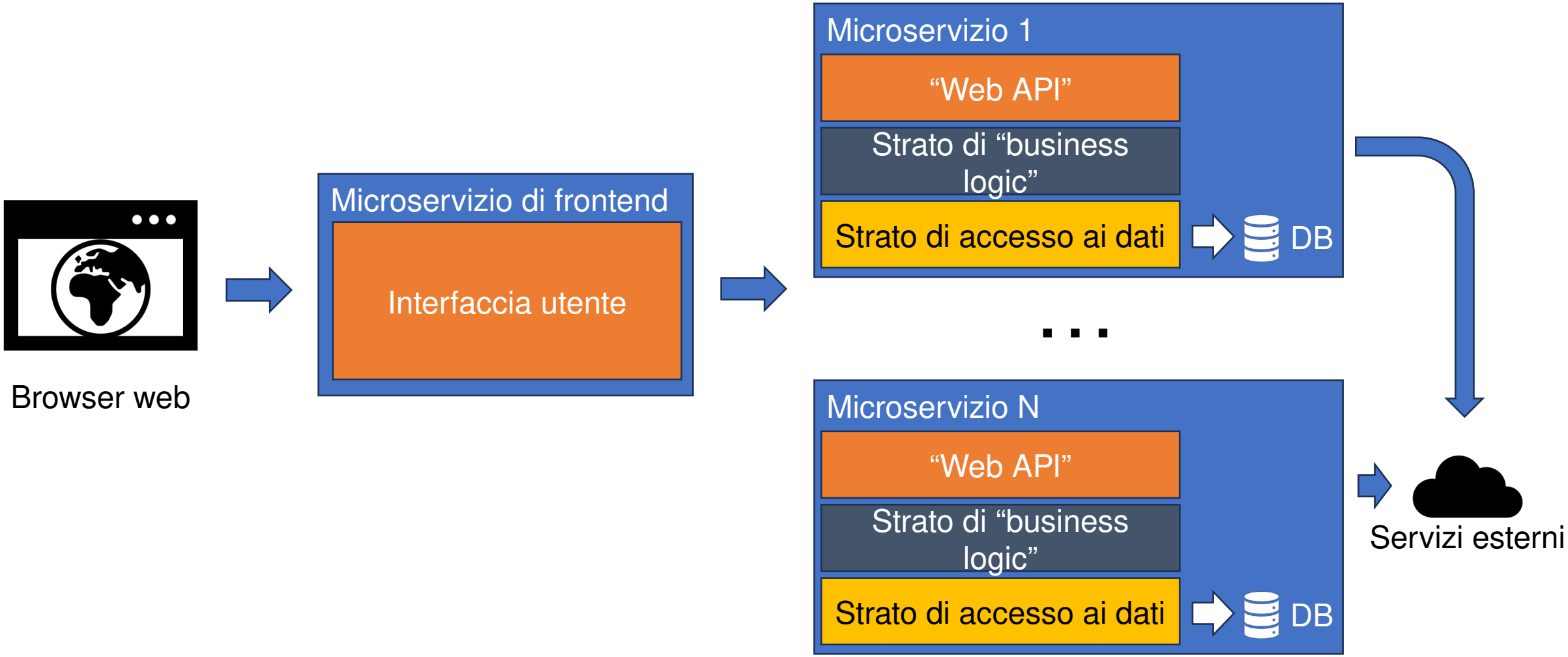
Architettura a microservizi

"E vissero tutti felici e contenti"

Architettura a microservizi

Approccio alla creazione di un’applicazione server come un set di piccoli servizi

Architettura a microservizi



Architettura dei microservizi

Caratteristiche dei microservizi

- Piena autonomia di sviluppo, distribuzione e scalabilità
- Distinzione netta delle funzionalità
- Comunicazione tra microservizi
- Proprietario del modello dei dati e della logica di dominio
- Più linguaggi di programmazione
- Diverse tecnologie di archiviazione dei dati

Vantaggi

- Scalabilità orizzontale: più istanze dello stesso microservizi
- Scalabilità verticale: potenziamento dell'hardware
- Chiara distinzione delle competenze
  - Flessibilità nell'aggiunta di nuove funzionalità
  - Libertà tecnologica
- Continuous delivery pienamente applicabile
- Incremento di Resilienza

Svantaggi

- Generalmente più complessità
- Debug complesso
- Sovraccarico organizzativo per coordinare gli aggiornamenti
- Atomicità tra operazioni che coinvolgono più microservizi non garantita

Architettura monolitica vs architettura a microservizi

Quale scegliere e quando?

- Crescita dell’organizzazione
- Aumento delle richieste di implementazione
- Alcune funzionalità del nostro applicativo necessitano scalabilità
- In generale difficile stabilire a priori

Migrazione applicazione monolitica in applicazione a microservizi

- Identificare la granularità di ogni servizio
- Necessario lavorare a stretto contatto con gli analisti funzionali per elencare la totalità delle funzionalità del nostro applicativo

Migrazione da microservizi a monolitico

- Prime video (https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90)
- Non tutto può e deve essere trasformato in microservizi
- David Heinemeier Hansson, creatore di Ruby on Rails: https://world.hey.com/dhh/how-to-recover-from-microservices-ce3803cc

Anatomia di un microservizio

Sovranità dei dati e indipendenza

- Ogni microservizio è proprietario dei suoi dati
- Importanti differenze con le applicazioni monolitiche:
  - Transazioni ACID non native su architetture a microservizi
  - Possiamo avere istanze DB differenti
- Stabilire la consistenza dei dati in un'architettura a microservizi senza un'unica transazione ACID

Dimensionamento

Oltre alla sovranità dei dati, la letteratura (e il nome) suggerisce che un microservizio debba avere dimensioni «contenute»

Questo aspetto è confermato dall'ambiente target dei microservizi (le architetture a container) caratterizzate da micro-kernel più adatti alla scalabilità orizzontale (replica delle istanze in esecuzione), piuttosto che a quella verticale (aumento delle risorse sulla singola istanza)

Tuttavia non esiste una regola precisa che indichi quanto debba/possa essere «grande» un microservizio

Dimensionamento

Quali metriche possiamo utilizzare, quindi, per dimensionare correttamente un microservizio?

- Numero di oggetti del modello dati del servizio
  - Un microservizio dovrebbe interagire/possedere con un numero «ragionevole» di entità relazionali
  - Un microservizio che interagisca con una grande quantità di oggetti sulla base dati potrebbe avere troppe responsabilità e, di conseguenza, essere troppo grande
  - Quanto è, quindi, «ragionevole» ? Anche in questo caso non esiste una regola precisa e il tutto va calato nel contesto funzionale. Tuttavia, si può prendere come parametro il rapporto *n oggetti del microservizio / n oggetti totali dell'applicazione*. Maggiore è questo rapporto, maggiore è la probabilità che il microservizio in questione abbia dimensioni eccessive e l'applicazione, nel suo complesso, non sia correttamente suddivisa in servizi

Dimensionamento

- Numero di API esposte
  - In molti casi, il numero di API esposte è proporzionale al numero di oggetti con cui il microservizio interagisca. In questo caso, valgono tutte le considerazioni fatte in precedenza (che sono la causa del problema)
  - Qualora non ci fosse questa dipendenza, anche in questo caso un numero eccessivo API, potrebbe suggerire una quantità eccessiva di responsabilità e una conseguente dimensione eccessiva
- Tempi di realizzazione
  - La stima dei tempi di realizzazione di un microservizio è un altro dei criteri utilizzabili per il suo dimensionamento
  - Un tempo di implementazione eccessivamente elevato potrebbe suggerire che il microservizio in questione abbia dimensioni eccessive
  - In letteratura è possibile trovare 15gg come tempo massimo di realizzazione di un microservizio. Nella pratica questo è un criterio ovviamente inapplicabile, tuttavia ci dà indicazione di come il tempo di realizzazione sia un criterio comunemente tenuto in considerazione per un corretto dimensionamento
  - **ATTENZIONE!** Il tempo di realizzazione potrebbe dipendere dalla complessità funzionale e algoritmica del microservizio, piuttosto che da un'eccessiva attribuzione di responsabilità. Pertanto è un criterio da valutare con molta attenzione, in relazione al contesto dell'applicazione

Dimensionamento - conclusioni

- Dimensionare correttamente un microservizio non è una «scienza esatta». E' importante approcciare il problema con la dovuta elasticità, tenendo presente il contesto applicativo e la complessità funzionale.
- Applicare i criteri descritti in modo «dogmatico», potrebbe generare una proliferazione di microservizi troppo piccoli, con un inevitabile numero eccessivo di interdipendenze e l'alta probabilità di necessità di transazioni distribuite (complesse e difficili da gestire).
- I criteri esposti vanno considerati con dei «buoni indicatori» e vanno valutati contestualmente, evitando di prenderne in considerazione solo uno a scapito degli altri.

Dimensionamento - conclusioni

- In contesti reali e complessi (applicazioni per sistemi informativi di grandi realtà), suddividere correttamente una problematica funzionale in microservizi e, di conseguenza, dimensionare correttamente gli stessi è un'attività complessa che non ha regole precise, ma solo indicatori
- In contesti reali, non è inusuale che questa attività avvenga per affinamenti successivi
- E' possibile, infatti, che una volta progettato l'intero sistema si decida di accorpare più microservizi in uno unico (p.e. per evitare inutili interdipendenze), o di separarne uno ritenuto troppo grande per rendere le rispettive funzionalità indipendenti

Dalla teoria alla pratica: i compromessi necessari in applicazioni reali

Una delle caratteristiche dei microservizi è quella di essere autonomi e indipendenti dal resto del sistema.

Questo significa che il crash o lo spegnimento di uno dei microservizi che compongono l'applicazione non dovrebbe compromettere il funzionamento degli altri microservizi, ma mettere offline solo le funzionalità di responsabilità del servizio spento o andato in crash

Ma questo è sempre possibile nei casi reali?

Dalla teoria alla pratica: i compromessi necessari in applicazioni reali

Ma questo è sempre possibile nei casi reali?

Questo può avvenire solo in due casi:

- Tutti i microservizi non comunicano tra loro. Ossia sono completamente autonomi e posseggono tutte le informazioni utili per compiere il proprio compito.
- I microservizi comunicano tra loro, ma ciascuno è in qualche modo di continuare il proprio compito con i dati che abbia a disposizione (magari quelli riferiti all'ultima richiesta fatta al microservizio offline?)

Dalla teoria alla pratica: i compromessi necessari in applicazioni reali

La prima casistica non è realistica (o rarissima) in applicazioni complesse. Se tutti i servizi non avessero necessità di comunicare con nessuno, sarebbero silos sigillati come tante applicazioni monolitiche e non di microservizi

La seconda, prevederebbe che ogni microservizio avesse una copia locale del dato da richiedere al servizio offline, in modo da poter continuare il proprio compito indipendentemente da questi.

Questo secondo scenario è realistico e verrà approfondito nelle prossime lezioni. E' tuttavia irrealistico considerare questa tecnica valida per tutte le casistiche e tutte le applicazioni

Dalla teoria alla pratica: i compromessi necessari in applicazioni reali

Come osservato per il dimensionamento dei microservizi, anche in questo caso non esiste un metodo efficace per tutti gli scenari e tutte le casistiche. L'architettura corretta è quella che preveda di **minimizzare l'interdipendenza tra i vari servizi**. Dimensionandoli correttamente e separando le responsabilità in modo opportuno.

Dove non sarà possibile garantire questa indipendenza, approfondiremo pattern e tecniche per replicare i dati tramite comunicazione asincrona e per rendere sicura la comunicazione sincrona tra i vari end-point

Endpoint e Web API

«Tutte gli endpoint portano a Roma»

Endpoint

- Gli endpoint permettono di comunicare con un servizio web
- Ogni servizio web espone uno o più endpoint
- Esempio di endpoint: https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY

Web API

- Generalmente esposte su server HTTP(S)
- Pubbliche (internet) o private (internet o intranet)
- REST (REpresentational State Transfer)
  - Metodi HTTP: GET, POST, PUT, PATCH, DELETE, altri…
  - Formato: JSON, XML
  - Struttura libera a discrezione dell'implementazione
  - Integrato in ASP.NET Core senza bisogno di componenti aggiuntivi
- SOAP (Simple Object Access Protocol)
  - Formato: XML
  - Struttura formale definita da WSDL (Web Services Description Language)
  - Implementazione .NET: WCF (Windows Communication Foundation)
- GraphQL (Graph Query Language)
  - Contrariamente alla norma, in GraphQL è il client a decidere quali dati leggere
  - Formato richiesta: GraphQL + JSON, formato risposta: JSON
  - Struttura formale definita da SDL (Schema Definition Language)
  - Implementazioni .NET: GraphQL.net, HotChocolate

https://wikipedia.org/wiki/Web_API
https://graphql.org/learn/

API Gateway

È il controllore del traffico per le richieste ai microservizi.

API Gateway

- Mappatura di ciascun endpoint al microservizio specifico
  - Catalogo dei microservizi e delle varie API
- Versionamento delle API, con versioni diverse dei microservizi
  - https://example.com/api/v1/objects/list » microservizio versione 1
  - https://example.com/api/v2/objects/list » microservizio versione 2
- Deploy (rollout) di aggiornamenti senza downtime
  1. Si installa la nuova versione del microservizio
  2. Si aggiorna l'API gateway per puntare alla nuova versione
  3. Si disinstalla la vecchia versione del microservizio
- Discovery
- Load-balancing

Grazie a tutti

(inserire qui le references)