ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Lezione 6 – Architetture Moderne

Davide Luppoli - davide.luppoli2@unibo.it

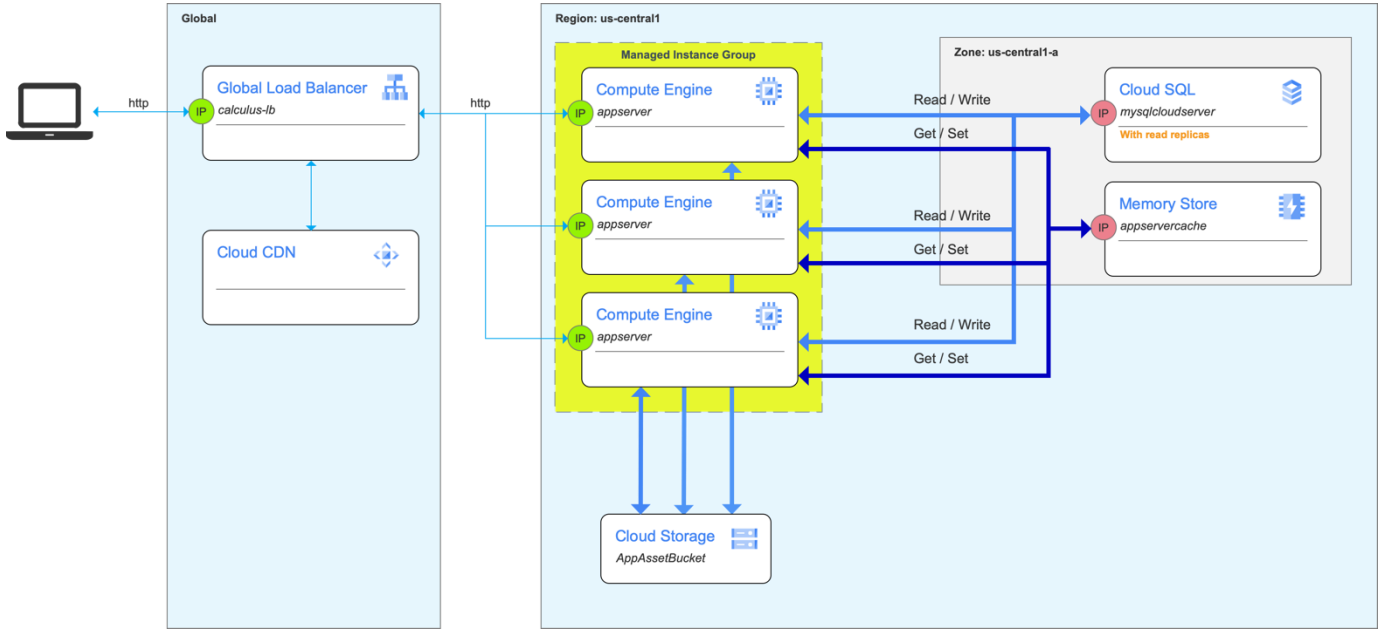
Applicazioni Monolitiche

Con il termine di **applicazione monolitica** si intende una applicazione software nella quale tutte le funzionalità sono inserite all'interno della stessa codebase.

- Tutte le nuove funzionalità sono inserite all'interno di una codebase
- Tutti gli sviluppatori lavorano, collaborando, sulla stessa codebase

Applicazioni Monolitiche

CalculusMasterV2, sebbene utilizzi componenti esterni (LB, DB, Cache), è una applicazione monolitica in quanto tutto il codice è inserito in un unico progetto e le funzionalità non sono separate.



Applicazioni Monolitiche - Vantaggi

L'approccio monolitico, adottato già da molto tempo, presenta alcuni indiscussi vantaggi:

- **Semplicità:** L'applicazione contiene tutto il necessario per funzionare, mantenendo semplice l'architettura del sistema e ottenendo vantaggi in fase di:
  - Sviluppo
  - Deploy
  - Test e manutenzione

Applicazioni Monolitiche - Vantaggi

- **Comunicazione interna:** I diversi moduli comunicano tra loro attraverso chiamate di funzione o librerie condivise all'interno del processo dell'applicazione
- **Scalabilità verticale:** Scalando verticalmente il server, l'intera applicazione beneficia di un aumento delle performance

Applicazioni Monolitiche - Svantaggi

L'approccio monolitico porta con se anche alcuni svantaggi, che rendono necessaria la ricerca di approcci alternativi/complementari:

- **Scalabilità:** E' necessario scalare l'intero sistema anche se è un solo componente ad aver bisogno di maggior potenza
- **Aggiornamenti:** Il forte accoppiamento tra i componenti può rendere più complicato l'aggiornamento
- **Single Deployment Unit:** A fronte di ogni modifica, anche se piccola, è necessario aggiornare l'intero sistema

Applicazioni Monolitiche - Svantaggi

- **Complessità di gestione:** al crescere della dimensione dell' app può diventare complicato sistemare i bug, così come fare test isolati
- **Tecnologia singola**: le varie componenti sono vincolate all'utilizzo della stessa tecnologia, non potendo quindi trarre vantaggio dall'utilizzo combinato di più tecnologie e linguaggi di programmazione

Applicazioni Monolitiche – Use Cases

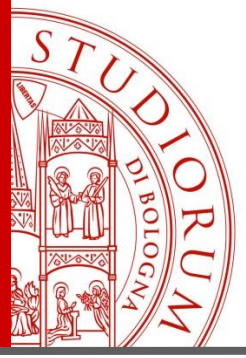
Considerando i pro e i contro una applicazione monolitica può essere preferita nei seguenti casi:

- **Progetti medio-piccoli:** poche funzionalità e team di sviluppo piccoli limitano le complessità di gestione
- **Time to market:** La semplicità di sviluppo diminuisce i tempi di realizzazione e di deploy
- **Sistemi legacy:** Una applicazione monolitica potrebbe essere più facilmente integrabile con sistemi legacy

Approcci alternativi (moderni)

Il superamento dei limiti delle applicazioni monolitiche passa dall'adozioni di alcuni approcci alternativi, alcuni dei quali giù affrontati in precedenza:

- Disaccoppiamento dell'applicazione dal suo stato applicativo
  - Scalabilità orizzontale, proxy e load balancing, della componente stateless
- Utilizzo di sistemi di cache
- Divisione dell'applicazione in "servizi" debolmente accoppiati
  - Mantenendoli semplici (approccio KISS)
  - Con possibilità di comunicazione asincrona



SOA – Service Oriented Architecture

SOA – Service Oriented Architecture

- E' un pattern che emerge ad inizio anni 2000
- Struttura l'applicazione dividendola in **servizi**
- Il software è composto da insieme servizi di piccole dimensioni e autonomi da tutti gli altri.
- Ogni servizio si occupa di una funzione specifica e comunica con gli altri sia per scambiarsi informazioni che per coordinarsi (Single Responsability & Separation of Concerns)

SOA – Caratteristiche

- **Basato sui Servizi:** il software è scritto come piccole unità autonome che svolgono una specifica funzione di business e sono accessibili tramite una loro interfaccia (web services)
- **Basso accoppiamento:** i servizi non hanno forti dipendenze tra di loro. Possono quindi essere sviluppati e aggiornati indipendentemente l'uno dall'altro, rendendo l’architettura complessiva più flessibile

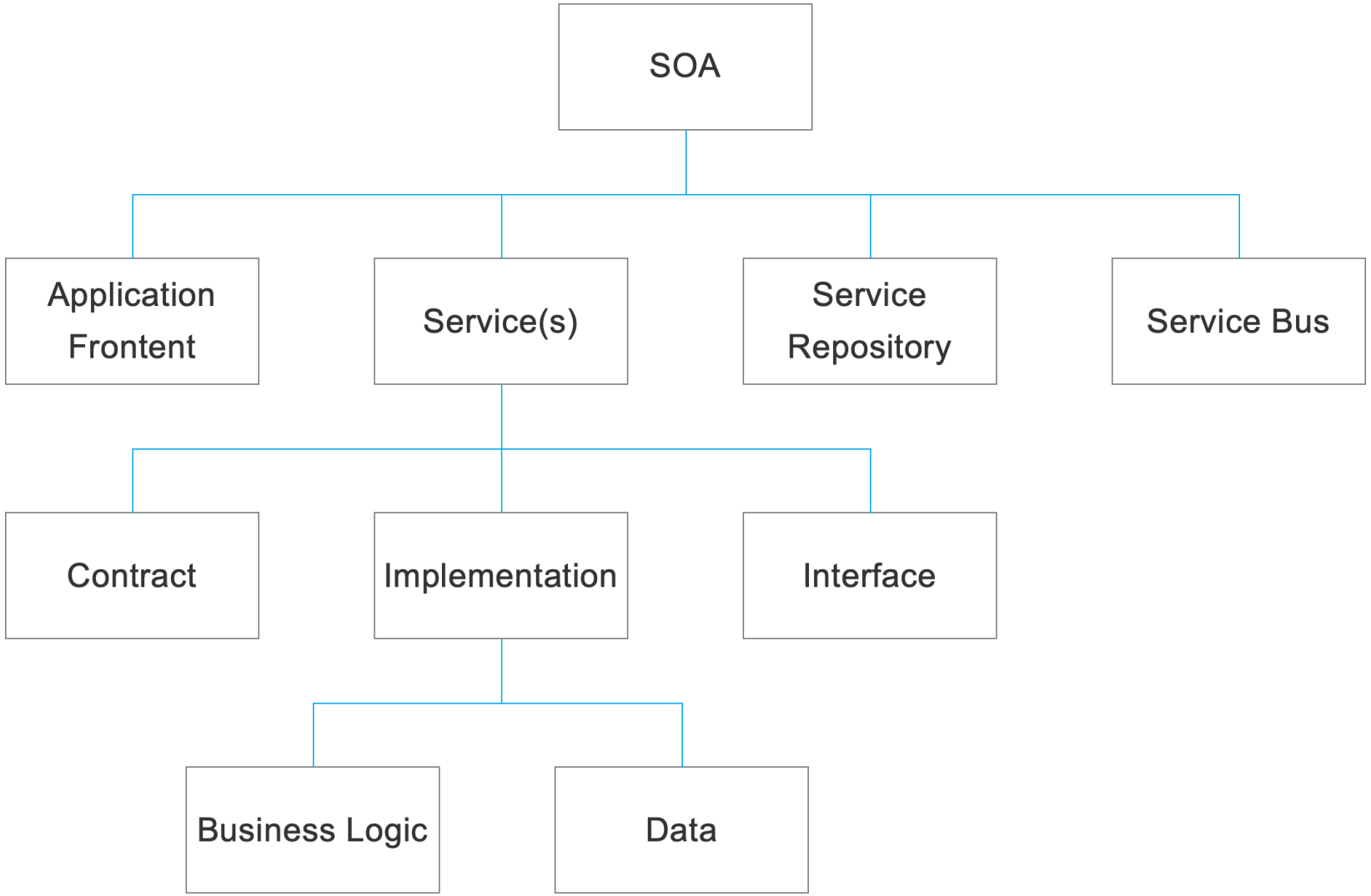
SOA – Caratteristiche

- **Riusabilità:** l'implementazione di una singola funzionalità e l'interfaccia di comunicazione, rende i servizi riutilizzabili in altri progetti, aumentando il riuso del codice e limitando i tempi di sviluppo
- **Modularità:** i servizi si possono combinare tra loro per ottenere funzionalità più complesse
- **Astrazione**: Il servizio diventa un layer di astrazione del software, dove il lavoro si concentra sullo sviluppo della funzionalità

SOA – Caratteristiche

- **Manutenzione:** separando gli ambiti, diventa più semplice gestire ed organizzare il deployment e gli aggiornamenti
- **Scalabilità:** ogni servizio può scalare indipendentemente da tutti gli altri, sia in senso verticale che orizzontale

SOA – Architettura



SOA – Architettura

L'architettura SOA prevede la presenza di alcune componenti:

- **Services**: che implementano le funzionalità. Sono composti da:
  - **Contract:** Definisce i dettagli tecnici e funzionali del servizio, compresi i parametri di input, di output e le operazioni supportate. Definiscono un "accordo" formale tra servizio e suo utilizzatore (consumer)
  - **Implementation:** E' il codice che implementa il contract, sia in termini di funzionalità che di gestione dei dati
  - **Interface:** E' il punto di accesso al servizio, quello che espone le funzionalità verso il consumer:

SOA – Architettura

- **Service Repository:** E' un archivio centralizzato che contiene le informazioni di tutti i servizi disponibili nell'architettura
- **Service Bus:** Canale di comunicazione condiviso tra tutti i servizi
- **Application Frontend:** Applicazione finale che utilizza i servizi (webapp, mobile app…)

Service Bus

E' l'intermediario (middleware layer) che facilita la comunicazione e lo scambio di dati tra i diversi servizi all'interno dell'architettura SOA. Il Service Bus agisce come un canale centralizzato attraverso il quale i servizi possono inviare e ricevere messaggi in modo sicuro ed efficiente.

Svolge un ruolo fondamentale nell'orchestrare e facilitare la comunicazione tra i servizi, migliorando l'interoperabilità, la flessibilità e l'affidabilità dell'intero sistema.

Service Bus

Le funzioni principali di un Service Bus sono:

- **Mediazione:** Consente ai servizi di comunicare in modo disaccoppiato, ovvero senza essere pienamente consapevoli degli altri servizi presenti nel sistema.
- **Routing dei messaggi**: Instrada i messaggi dai servizi mittenti ai servizi destinatari in base alle regole di routing configurate. Si possono implementare logiche di instradamento complesse ed è possibile supportare il pattern di comunicazione come il pub-sub.

Service Bus

- **Transformazione dei messaggi:** Esegue, se necessario, la trasformazione dei messaggi da un formato a un altro, ad esempio da XML a JSON o viceversa. Aumentano quindi le possibilità di interoperabilità tra servizi differenti.
- **Gestione delle transazioni:** Coordina le operazioni di più servizi all'interno di una transazione complessiva.

Service Bus

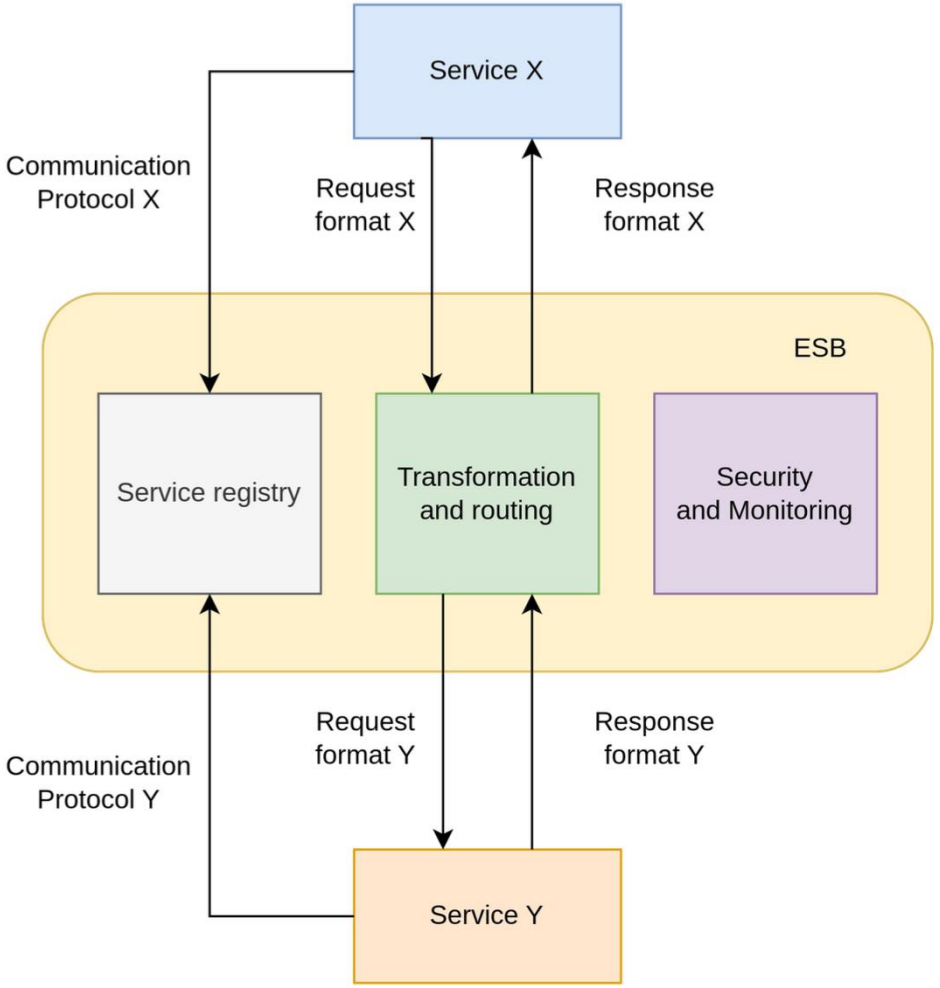
- **Gestione degli errori e del monitoraggio:** Fornisce funzionalità per il monitoraggio dei servizi e per la gestione degli errori. Può includere meccanismi per la gestione delle code, il ripristino automatico delle operazioni fallite e la registrazione degli eventi per scopi di monitoraggio e audit.
- **Sicurezza:** fornisce funzionalità di sicurezza per proteggere le comunicazioni tra i servizi, ad esempio mediante l'uso di autenticazione, autorizzazione, crittografia e firme digitali.

Service Bus - Funzionamento

La comunicazione tra due servizi tramite Service Bus avviene in più fasi:

1. Il mittente (Service Consumer) contatta il Service Bus utilizzando il proprio formato di richiesta
2. Il Service Bus controlla il Service Registry per determinare il formato di richiesta accettato dal destinatario. Se necessario effettua la traduzione
3. Il Service Bus contatta il destinatario (Service Producer) usando il formato da esso richiesto
4. Il Service producer risponde utilizzando il proprio formato, che sarà tradotto dal service bus prima di inviarlo al service consumer

Service Bus - Funzionamento



SOAP - Simple Object Access Protocol

- SOAP è un protocollo di comunicazione che definisce un formato standard per la struttura dei messaggi scambiati tra i servizi web su una rete.
- Evita che il Service Bus debba effettuare traduzioni
- Utilizza XML per la codifica dei messaggi e può operare su diversi protocolli di trasporto, come HTTP, SMTP, FTP, ecc.
- Definisce anche un framework per la definizione di metodi, parametri e tipi di dati utilizzati nei messaggi

SOAP - Simple Object Access Protocol

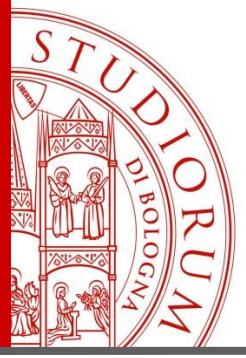
- SOAP è stato presentato al W3C nel 1998 e pubblicato nel 2000
- E' indipendente da qualunque linguaggio di programmazione o piattaforma, il che consente a qualsiasi web service di poterlo adottare senza stravolgere la sua implementazione
- Utilizza un modello richiesta/risposta (tipo HTTP), dove un'applicazione (sender) manda un messaggio SOAP ad un'altra applicazione (receiver) che lo elabora e fornisce una risposta al sender

SOAP – Struttura dei messaggio

- Un messaggio SOAP è formato da una “envelope” che contiene l’header e il body della richiesta
- L’header contiene le informazioni del messaggio, quali mittente, destinatario, tipo di autenticazione e altri attributi
- Il body ha il contenuto effettivo del messaggio con i dati scambiati utilizzando i formati specifici in base al tipo di dato

WSDL – Web Services Definition Language

- WSDL è un linguaggio di descrizione di servizi utilizzato per definire l'interfaccia in modo standardizzato e indipendente dal linguaggio di programmazione.



Microservizi

Limiti di SOA

- SOA ha portato un buon grado di separazione rispetto all'approccio monolitico, mantenendo anche l’integrazione con i sistemi legacy
- Il modello è però ancora troppo centralizzato ed articolato. I servizi fanno riferimento ad un bus centralizzato per la comunicazione e a un registry centralizzato.
- Spesso i servizi tendono ad essere unità di grandi dimensioni con, funzionalità non completamente indipendenti (si parla anche di servizi monolitici)

Limiti di SOA

- La necessità di componenti centralizzati e di una architettura di supporto porta SOA ad essere adatto per aziende grandi e strutturate, ma non per aziende piccole che devono muoversi in maniera rapida per adattarsi in fretta ai cambiamenti del mercato
- Serve maggiore elasticità ed agilità

Microservizi

- **Pattern architetturale** che struttura un'applicazione come un insieme di servizi piccoli ed indipendenti. Ognuno è responsabile per una specifica funzionalità ed espone una sua interfaccia per poter comunicare **direttamente** con gli altri mediante protocolli leggeri e standard (HTTPs principalmente).
- **Approccio completamente decentralizzato:** ogni servizio è indipendente dagli altri, sia nello sviluppo che nella comunicazione che nella gestione.

Microservizi

La nascita del pattern a microservizi è stato agevolato anche da altri fattori:

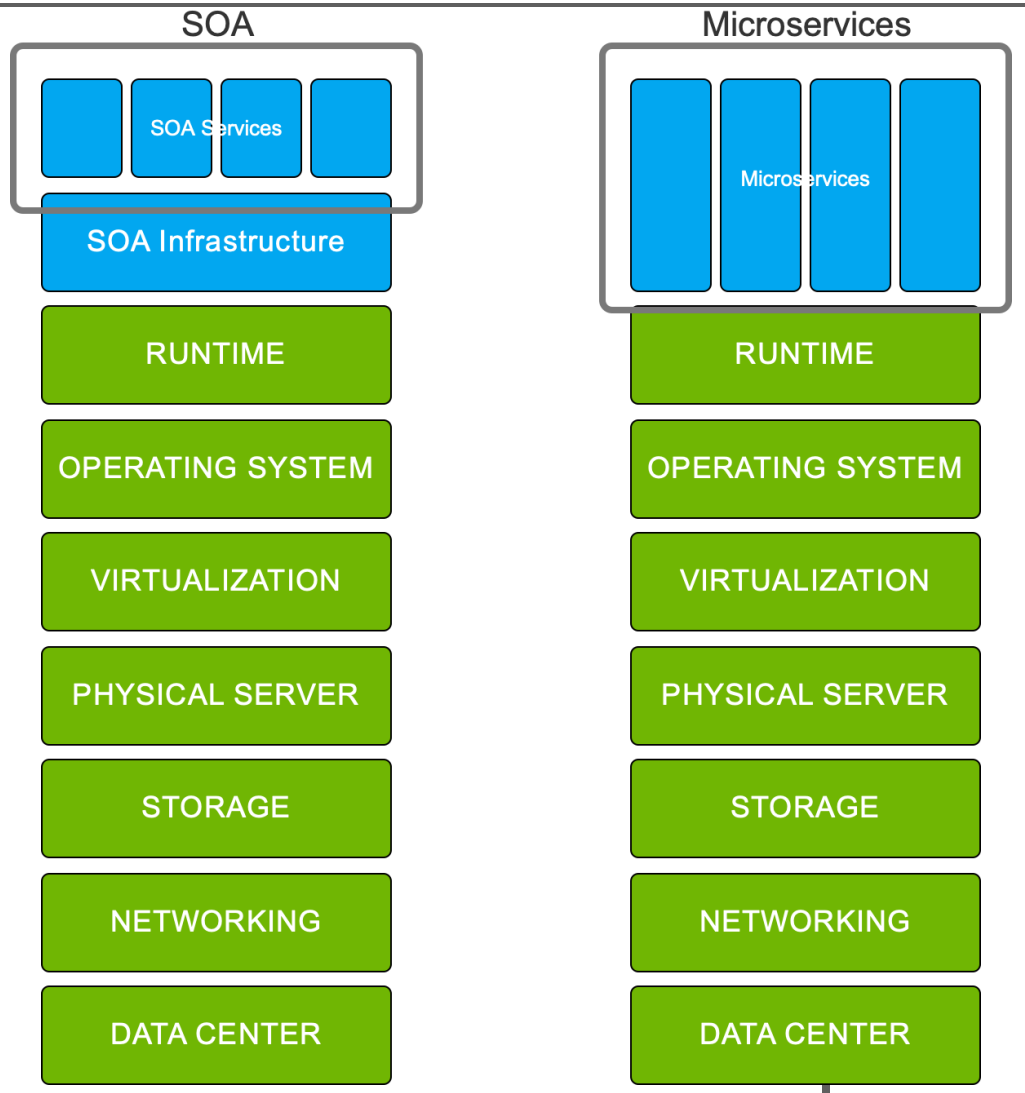
- **Virtualizzazione, cloud, container, IaC:** tecnologie che rendono molto più semplice e veloce lo sviluppo e il deploy di piccoli servizi indipendenti
- **Agile Movement:** approccio metodologico allo sviluppo software che consente di sviluppare più rapidamente un software e le sue funzionalità

Microservizi - Svantaggi

- **Cambiamento significativo** nel modo in cui si progetta, sviluppa, testa e distribuisce un'applicazione (possibile aumento dei tempi)
- **Complessità:** l'architettura sottostante diventa più complessa. Ogni microservizio è una istanza applicativa a cui deve essere garantita la possibilità di comunicare con le altre.
- **Operations:** I temi di load balancing, sicurezza, monitoraggio devono tenere conto di molteplici elementi e diventa più complicato mantenere tutto sotto controllo

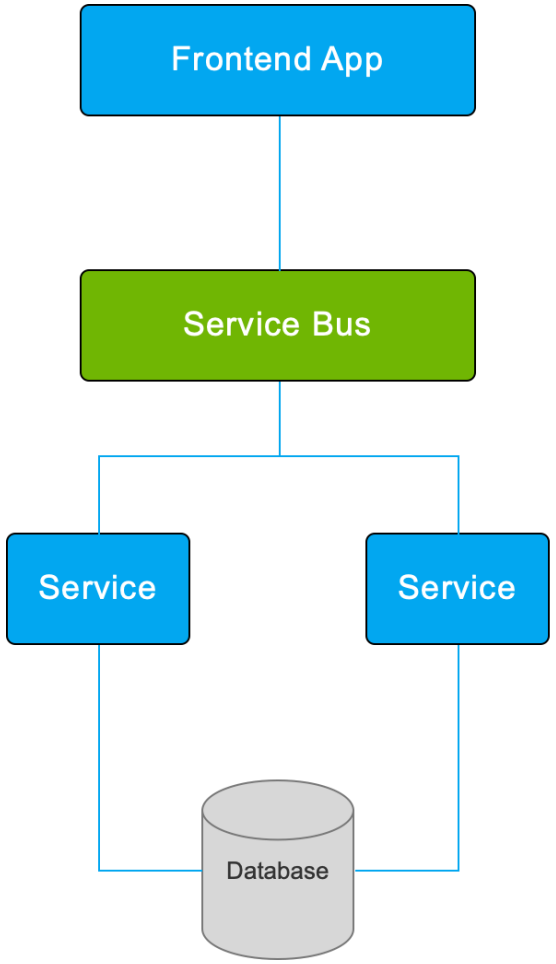
SOA vs Microservizi

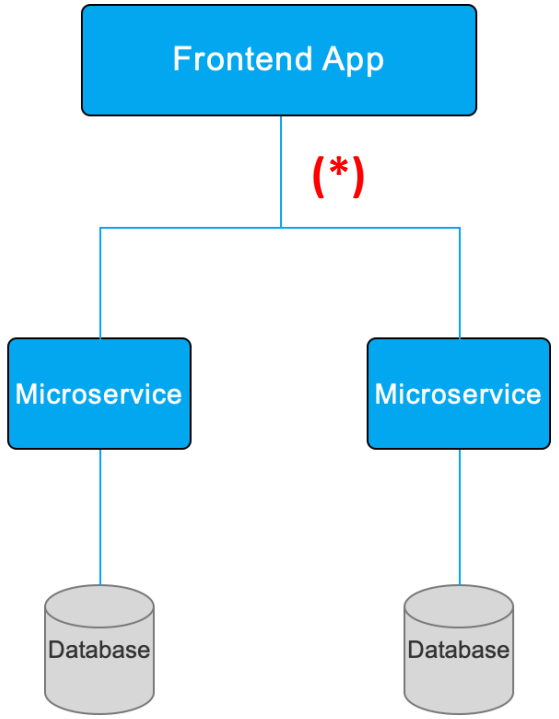
- SOA può essere visto come un livello applicativo unico (data la necessità dei componenti centralizzati) su cui costruire i servizi
- Nell'approccio a microservizi invece ogni servizio è una applicazione a sé stante



SOA vs Microservizi

- Nell'approccio SOA il database tende ad essere condiviso tra tutti i servizi
- Nell'approccio a microservizi invece ogni servizio ha il proprio database





(*) Analizzeremo meglio questa area in seguito

Da SOAP a REST

- E' necessario trovare una semplificazione anche a SOAP (così come i microservizi semplificano SOA)
- Il formato XML utilizzato da SOAP è infatti di difficile lettura e la sua elaborazione è computazionalmente onerosa.
- Il protocollo HTTP(s), usato anche da SOAP, è una buona scelta ma è necessario eliminare le sovrastrutture imposte dal formato XML
- Sfrutta questo principio il **paradigma REST**, introdotto nel 2000 da Roy Fielding nella sua tesi di dottorato.

REST – Representational State Transfer

- REST è un paradigma di accesso a microservizi che si basa sui principi base del protocollo HTTP:
  - **E' orientato alle risorse:** I microservizi pertanto "espongono" risorse a cui si accede tramite REST
  - **E' client-server e request-response:** Il server si mette in attesa di ricevere richieste da parte dei client. Ad ogni richiesta segue una risposta. Il server non esegue azioni se non a seguito di una richiesta.
  - **E' stateless:** Il server non mantiene stati applicativi. Ogni richiesta viene quindi trattata in modo indipendente dalle richieste precedenti

REST – Representational State Transfer

- Le risorse sono identificate da URI
- Le azioni da eseguire sulle risorse sono specificate utilizzando i verbi HTTP (GET, POST, PUT, DELETE…)
- Lo scambio dati con le risorse può avvenire in vari formati standard, tra cui:
  - Testo
  - XML
  - Json

RESTful API

- I microservizi espongono la propria interfaccia (API – Application Programming Interface) in termini di risorse, specificandone:
  - URI identificativo
  - Verbi HTTP ammessi
  - Formato dello scambio dei dati e dei parametri
- Un servizio che espone le proprie risorse in modo conforme con il paradigma REST è detto Restful API

RESTful API

Le API RESTful devono rispettare una serie di requisiti:

- Le risorse sono identificate da nomi e non da verbi:
  - `/articoli/`
  - `/articoli/{id}`
- Le risorse possono essere innestate una dentro l'altra
  - `/articoli/{id}/commenti`
  - `/articoli/{id}/commenti/{id}`
  - `/articoli/{id}/commenti/{id}/autore`

RESTful API

- Le risorse possono essere raggiungibili in modi differenti
  - `/articoli/{id}/commenti/{id}/autore`
  - `/autori/{id}`
- Le azioni CRUD da eseguirsi sulle risorse sono specificate associando un significato preciso ai verbi HTTP: GET (lettura), POST (creazione), PUT (aggiornamento), DELETE (cancellazione)
  - GET /articoli/{id}
  - PUT /articoli/{id}

RESTful API

- Le azioni NON CRUD possono essere specificate inserendo un verbo alla fine dell'url. Nell'url non devono mai apparire metodi crud.
  - `/autori/{id}/sospendi-account`
  - `/playlists/{id}/play`
- Le azioni restituiscono codici standard: 2xx, 4xx, 5xx…
- I nomi di risorse e di azioni sono significativi ed utilizzano – per separare le parole e migliorare la leggibilità
- Tutti gli URL hanno una struttura uniforme e consistente

RESTful API - Vantaggi

- **Indipendenza:** in termini di piattaforma (bastano client e server che supportino HTTP) e di linguaggio (i formati XML/JSON sono facilmente interpretabili da qualsiasi linguaggio di programmazione)
- **Semplicità e flessibilità**: Agevola e semplifica la definizione delle interfacce dei sercizi
- **Scalabilità e performance:** il paradigma stateless consente ai microservizi di scalare orizzontalmente. Il formato richiesta/risposta è inoltre molto efficiente anche dove la larghezza di banda è limitata

RESTful API - Vantaggi

- **Caching:** le risposte fornite possono essere archiviate in qualsiasi livello di cache (browser, CDN o cache server interni) per migliorare ulteriormente le performance

RESTful API - Esempio

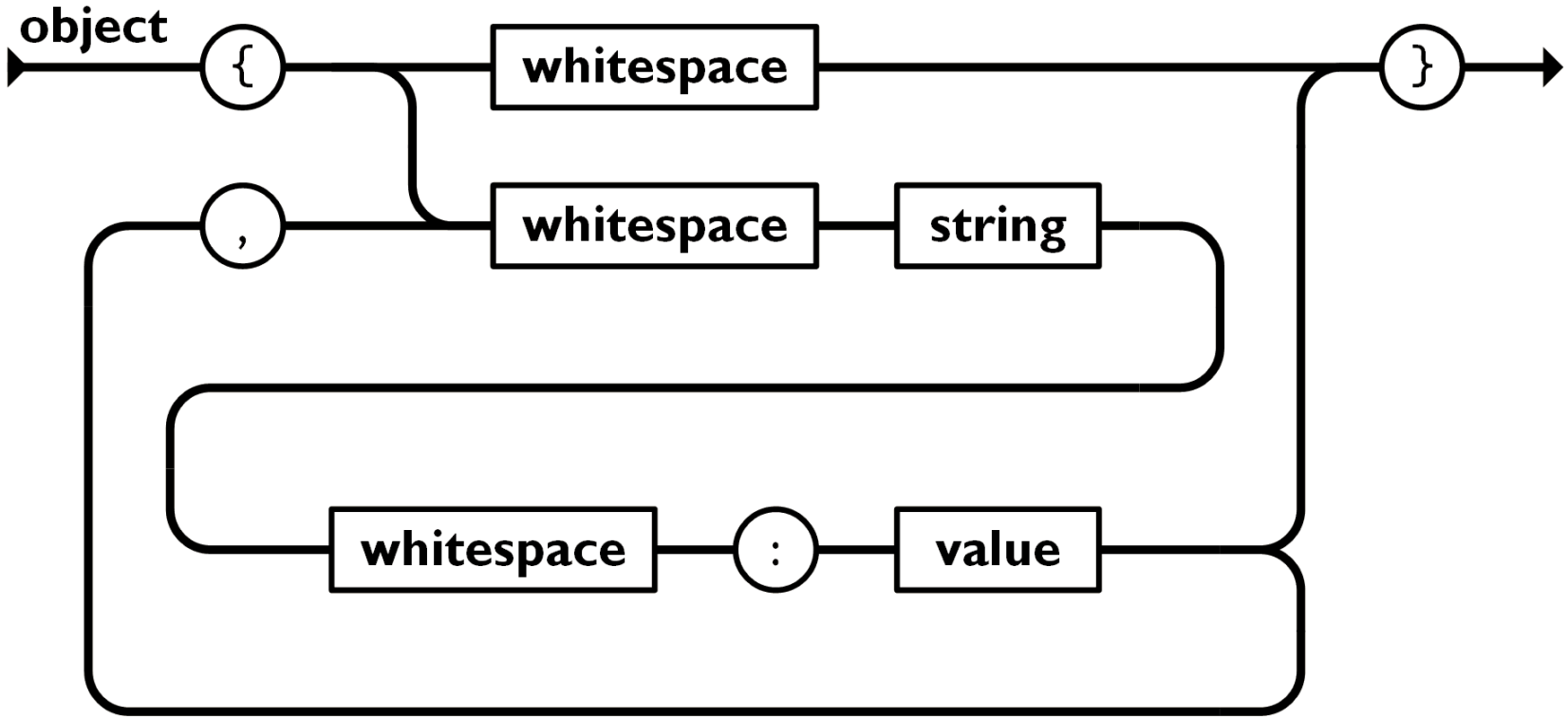
- `GET /students` → Recupera tutti gli studenti
- `POST /students` → Aggiunge un nuovo studente
- `GET /students/{id}` → Recupera uno studente dato il suo id
- `PUT /students/{id}` → Modifica uno studente identificato dal suo id
- `DELETE /students/{id}` → Cancella uno studente identificato dal suo id
- `GET /students/{id}/esami` → Recupera i voti di uno studente
- `GET /students/{id}/media-voti` → Recupera la media vori di uno studente

Json

- JSON (JavaScript Object Notation) è un formato di rappresentazione dei dati nato al fine di agevolarne lo scambio. E' un sottoinsieme del linguaggio di programmazione Javascript ed è fortemente utilizzato nelle API Restful
- Json è un formato molto semplice da leggere e da scrivere. I dati sono rappresentati come testo e scambiati come tali.
- Json si basa su due strutture dati principali:
  - Gli oggetti e Le collezioni (array)

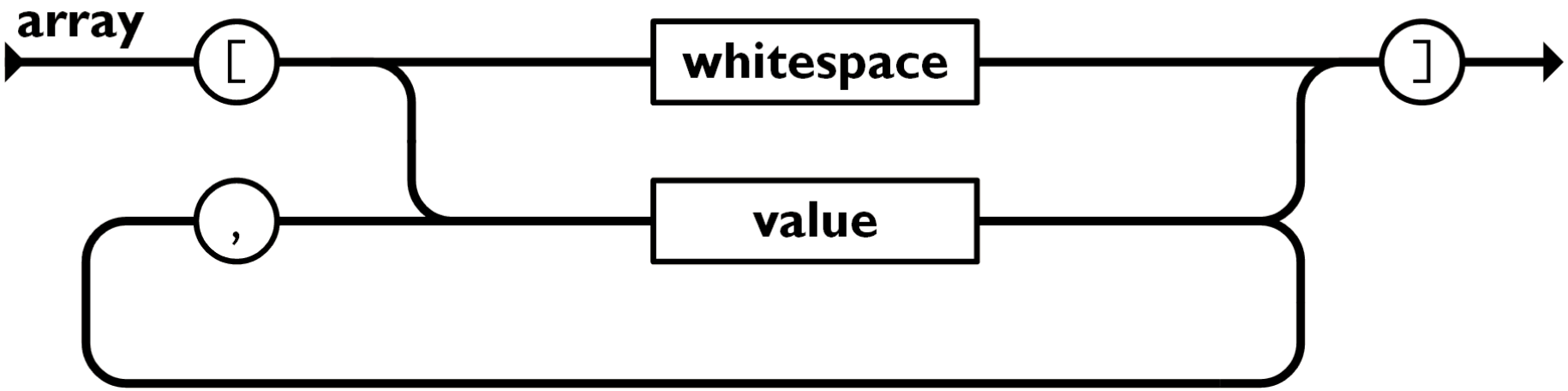
Json - Oggetti

Gli oggetti sono contenuti in una coppia di graffe e al loro interno contengono una serie di coppie chiave : valore separate da ;



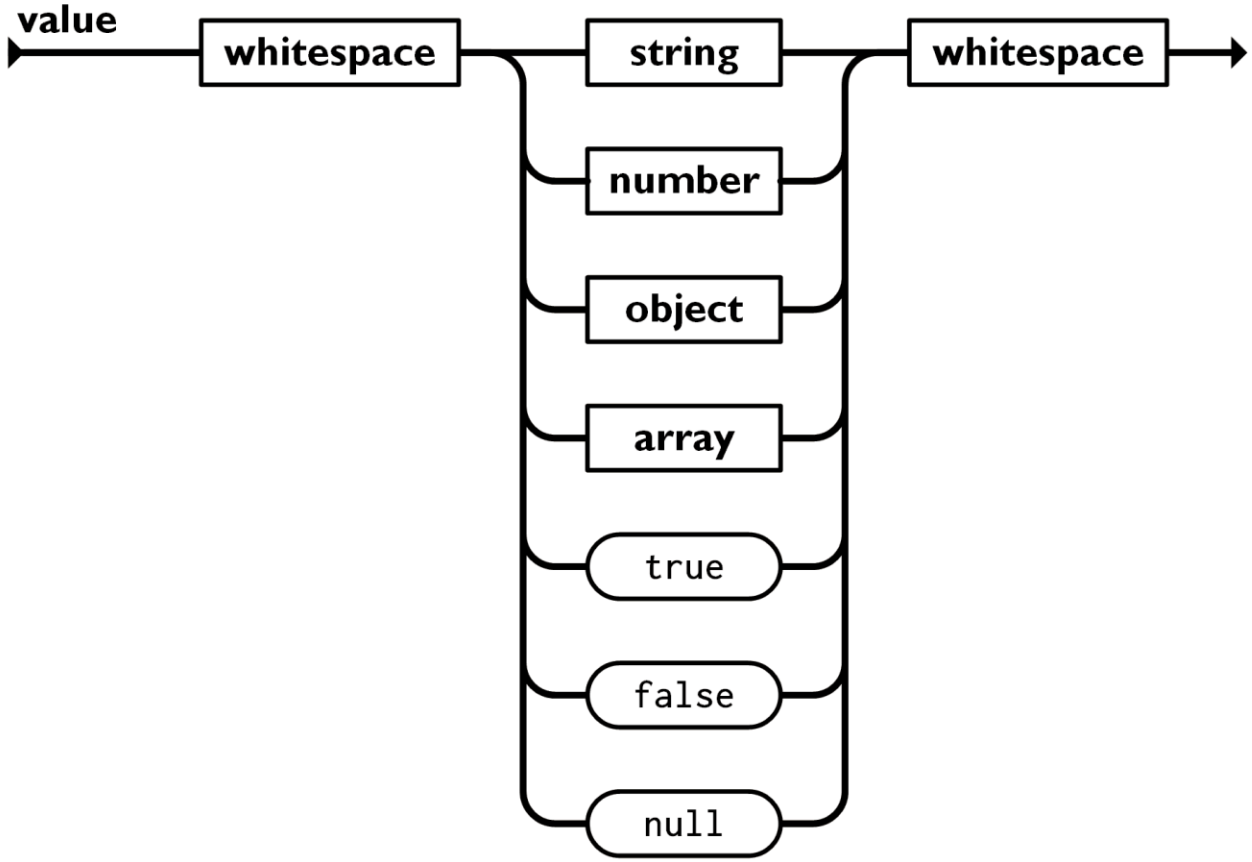
Json - Array

Gli array sono contenuti in una coppia di parentesi quadre e contengono una serie di valori separati da virgola



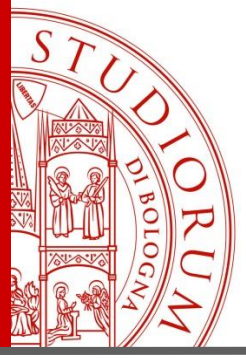
Json - Valori

I valori possono essere elementari (stringhe, numeri…) oppure array o oggetti



Json - Esempio


```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```



Event Driven Architecture

Event Driven Architecture (EDA)

- Pattern architetturale di comunicazione asincrona tra sistemi (p.e. servizi / microservizi).
- La comunicazione non si basa su richieste dirette e relative risposte bensì su eventi
- I sistemi che stanno comunicando non hanno un rapporto di client/server bensì di publisher/subscriber

Event Driven Architecture (EDA)

- Un sistema, detto **publisher**, pubblica un evento al verificarsi di determinate condizioni
- L'altro sistema, detto **subscriber**, si registra all'evento chiedendo di ricevere una notifica quando questo si verifica.
- E' inoltre necessario un ulteriore componente, detto Event Broker, che si occupa di gestire le pubblicazioni, le sottoscrizioni e le notifiche

Event Driven Architecture (EDA)

- L'Event-Driven Architecture è ampiamente utilizzata in una varietà di scenari, inclusi sistemi di messaggistica in tempo reale, applicazioni IoT, analisi dei dati in tempo reale…
- E' un fattore chiave per la scalabilità e l'indipendenza dei sistemi a microservizi

Event Driven Architecture (EDA) - Vantaggi

- **Indipendenza:** i servizi non hanno dipendenza nelle comunicazioni, quindi possono essere cambiati/rimpiazzati più facilmente e senza ripercussioni
- **Scalabilità:** le EDA sono altamente scalabili, in quanto le componenti si possono aggiungere orizzontalmente senza impatti sul sistema
- **Real-time:** grazie all'alto grado di parallelismo si possono gestire grandi volumi di messaggi e flussi dati, rendendo le EDA ideali per applicazioni IoT

Event Driven Architecture (EDA) – Casi d'uso

- Ideale per sistemi di messaggistica e notifica, dove le componenti devono comunicare tra loro in maniera asincrona
- Modello perfetto da adoperare nelle applicazioni cloud native grazie all’enorme elasticità con cui è possibile aggiungere/rimuovere elementi

Publisher / Subscriber model

- **Publisher:** le entità che producono e inviano il messaggio (evento) verso un message broker
- **Subscriber:** Entità che ricevono ed elaborano il messaggio da un message broker a cui si sono registrati
- **Message broker:** strato intermedio che riceve i messaggi dai publisher e li inoltra agli appropriati subscriber. Si compone di una serie di **topic**, ciascuno adibito a contenere messaggi di una determinata categoria

Publisher / Subscriber model

- I subscriber si **registrano** ad uno o più topic di un message broker, per riceverne i messaggi
- Il Publisher manda un **evento** ad un topic del message broker
- Il message broker riceve l'evento e lo archivia sul topic e da qui 2 modalità:
  - **Push:** il broker inoltra il messaggio ai subscriber registrati al topic
  - **Pull:** i subscriber verificano tramite polling se ci sono nuovi messaggi

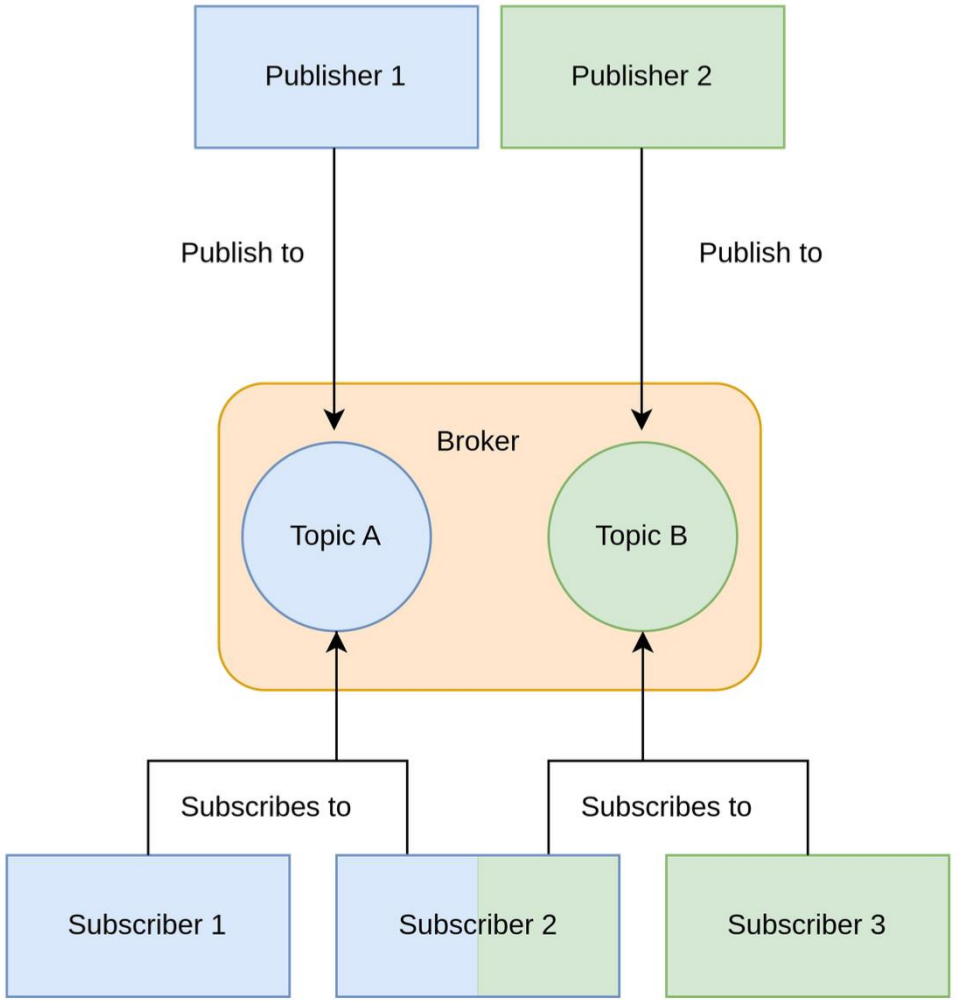
Publisher / Subscriber – Push vs Pull

- **Vantaggi push:** proattivo nel mandare i messaggi appena sono disponibili e più efficiente in quanto i subscriber non devono continuamente verificare la disponibilità di un nuovo messaggio
- **Svantaggi push:** se il publisher genera tanti messaggi per tanti subscriber il sistema di messaggistica può essere sovraccarico, oltre che i subscriber possono ricevere messaggi di cui non hanno bisogno, usando più risorse

Publisher / Subscriber – Push vs Pull

- **Vantaggi pull:** I subscriber controllano solo i nuovi messaggi quando necessario, riducendo i trasferimenti non necessari e rendendo il sistema di subscriber molto più scalabile dell'approccio push
- **Svantaggi pull:** i subscriber devono attivamente richiedere messaggi (polling) con impatto sulle performance e latenza per l'elaborazione dei messaggi

Publisher / Subscriber



Event Driven Architecture (EDA)

- Architetture EDA possono essere implementate utilizzando vari protocolli di coda di messaggi (messaging queues) tra cui MQTT, AMQP, SQS, XMPP.
- I principali strumenti EDA sono: Apache Kafka, Apace Pulsar, Rabbit MQ, Amazon EventBridge, Azure Event Grid, Google Cloud Pub/Sub

Riferimenti e approfondimenti

- SOA:
  - https://www.oreilly.com/library/view/service-oriented-architecture-analysis/9780133858709/
- Building Microservices:
  - https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/
- REST:
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- GCP PubSub:
  - https://cloud.google.com/pubsub/docs/publish-receive-messages-gcloud

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

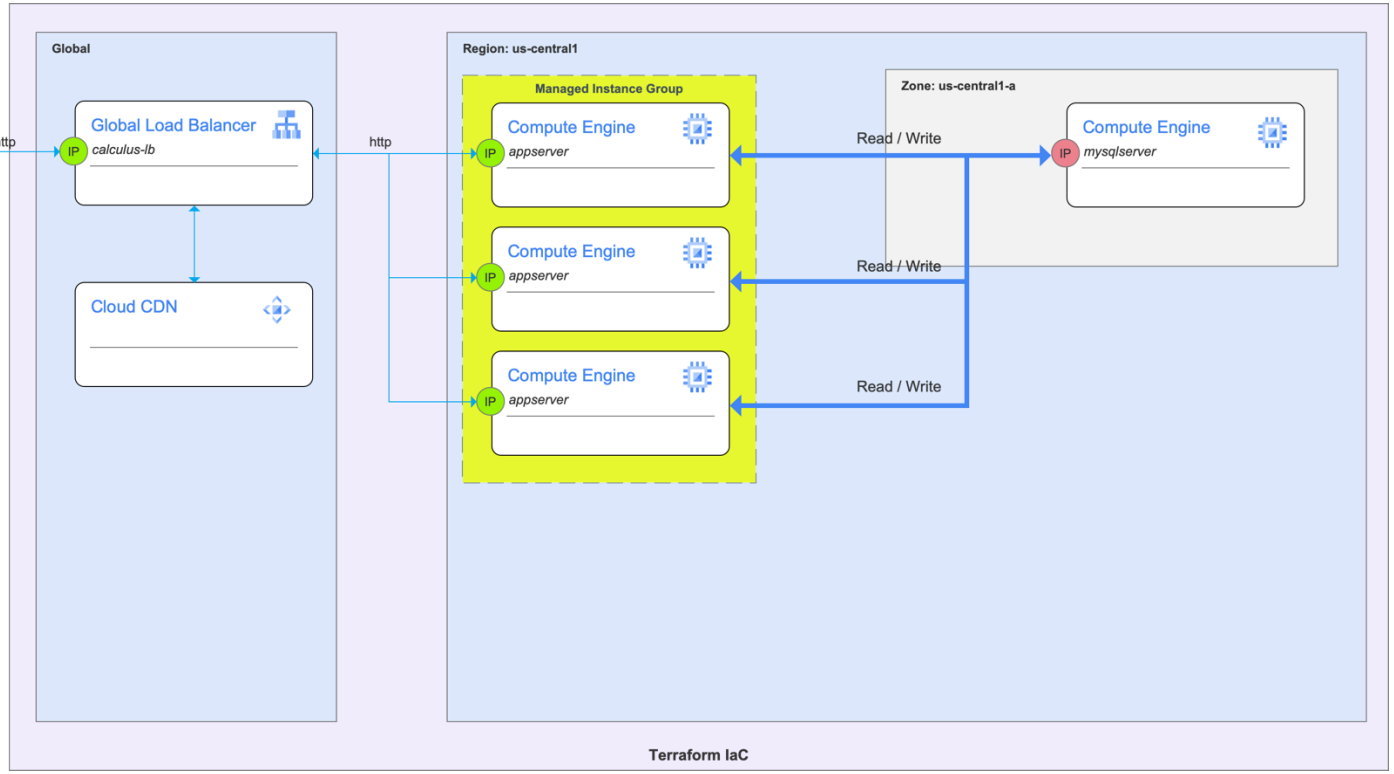
Lezione 6 – Architetture Moderne

HANDS ON

Davide Luppoli - davide.luppoli2@unibo.it

Situazione attuale (semplificata)

Usiamo lo scenario semplificato (quello richiesto nell'esercizio per casa) in quanto l'attenzione è sulla parte applicativa



Da monolitico a microservizi

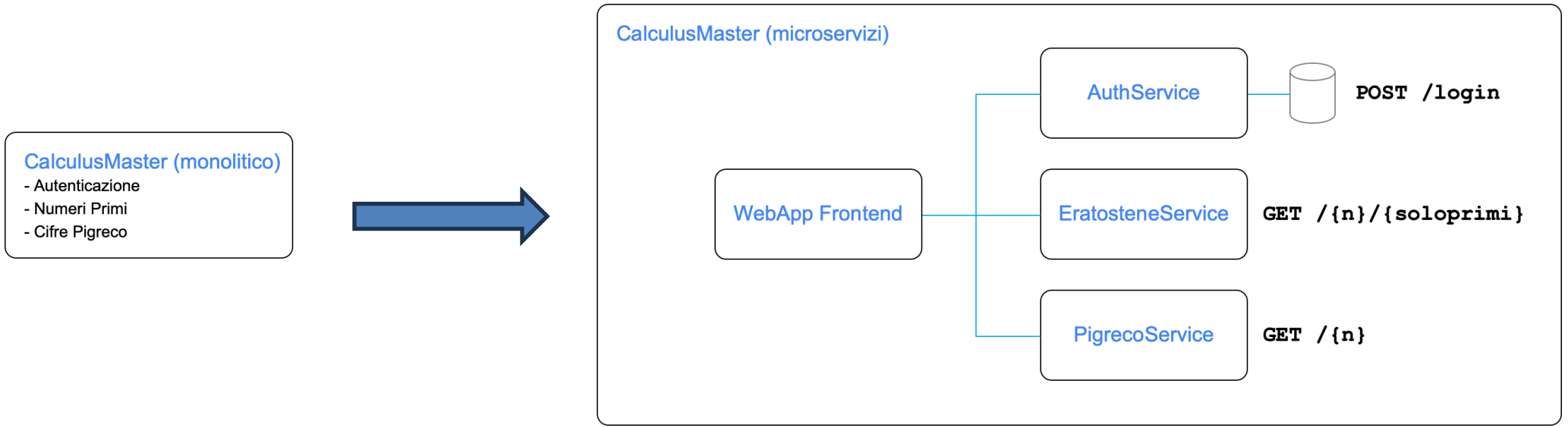
In Calculus Master si possono identificare 3 funzionalità:

- Autenticazione utenti
- Calcolo numeri primi
- Calcolo cifre π

Ogni funzionalità è indipendente dalle altre ed è quindi candidata a diventare un microservizio

Da monolitico a microservizi

- Ogni microservizio sarà realizzato come API Restful, indipendente dalle altre
- E' pertanto necessario aggiungere una interfaccia web di frontend



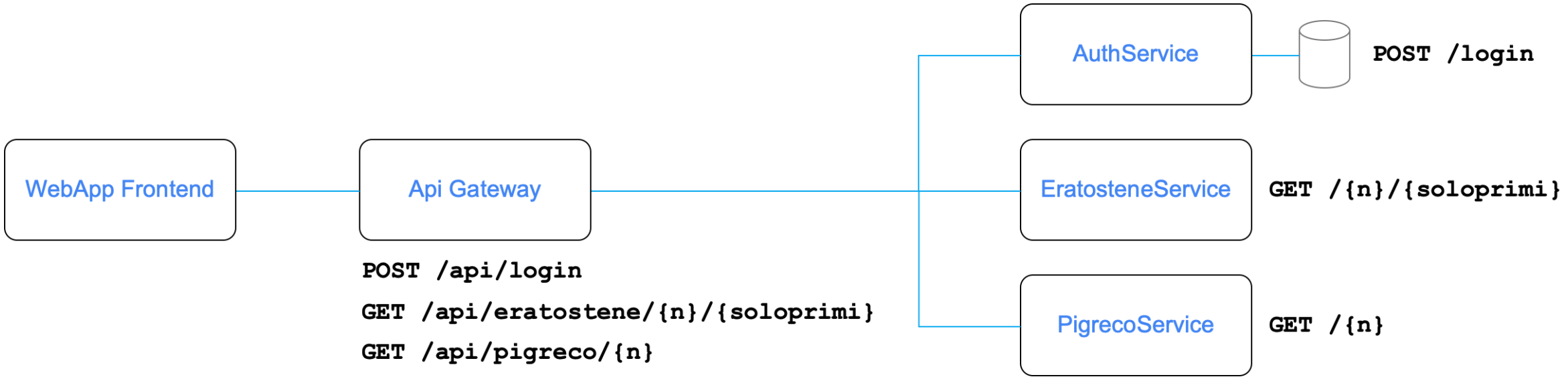
Da monolitico a microservizi

- Ogni microservizio, se necessario, utilizza il proprio database (nel nostro caso solo AuthService ne ha bisogno.
- Ogni microservizio può essere realizzato con tecnologie differenti, a patto che si rispetti l'interfaccia REST
  - AuthService, EratosteneService e PigrecoService continuano ad essere applicazioni node.js
  - Per il Frontend Web si è scelto di usare Angular (https://angular.io)

Ottimizzare l'uso dei microservizi

- L'architettura precedente porta l'applicazione web ad dover essere consapevole di tutti i microservizi esistenti, tenendo un riferimento ad ognuno di essi
- Per disaccoppiare ulteriormente è possibile inserire un api-gateway, ovvero un ulteriore microservizio che funge da proxy tra l'applicazione frontend e tutti i microservizi.
- L'applicazione frontend ha quindi un solo riferimento di backend

Ottimizzare l'uso dei microservizi



Autenticazione/Autorizzazione

- Per l'autenticazione e l'autorizzazione si sceglie di utilizzare un token jwt, emesso da AuthService
- Il token diventa però un fattore di accoppiamento tra i microservices, in quanto:
  - E' emesso da un servizio (AuthService)
  - E' utilizzato da altri servizi (EratosteneService e PigrecoService) per autorizzare le operazioni
- E' possibile procedere in più modi

Jwt Token 101

- I token JWT sono token che contengono informazioni strutturate come un oggetto json. (informazioni necessariamente non riservate).
- Sono firmati digitalmente dal server di autenticazione nel momento dell'emissione e pertanto non è possibile modificarli. Sono composti da tre parti:
  - Header: con informazioni relative agli algoritmi di firma
  - Payload: con le informazioni principali
  - Signature: con la firma dell'header e del payload

Jwt Token 101

- I token sono generati dal server in fase di autenticazione ed inviati al cliente.
- Il client invierà il token con ogni successiva richiesta, dando modo al server di verificare l'identità
- Il server che riceve il token verifica che non sia stato modificato (autenticità) e solo in seguito ne estrae le informazioni da usare per l'autorizzazione

Jwt Token 101

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.N3Hb-
h4CdvYDpm6iT-kQVAXt_q2vBnnZ-BDLfOPrd18
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secretpassword
) secret base64 encoded
```

Jwt Token 101

- I token possono essere firmati utilizzando molteplici algoritmi, sia a chiave simmetrica che a chiave asimmetrica
- In caso di firma con algoritmi a chiave simmetrica (conosciuta solo al server di autenticazione) gli utilizzatori del token non saranno in grado né di modificarlo né di verificarne l'autenticità
- In caso di firma con algoritmi a chiave asimmetrica gli utilizzatori del token possono essere a conoscenza della chiave pubblica e pertanto possono verificare l'autenticità del token, senza però modificarlo

Autenticazione/Autorizzazione

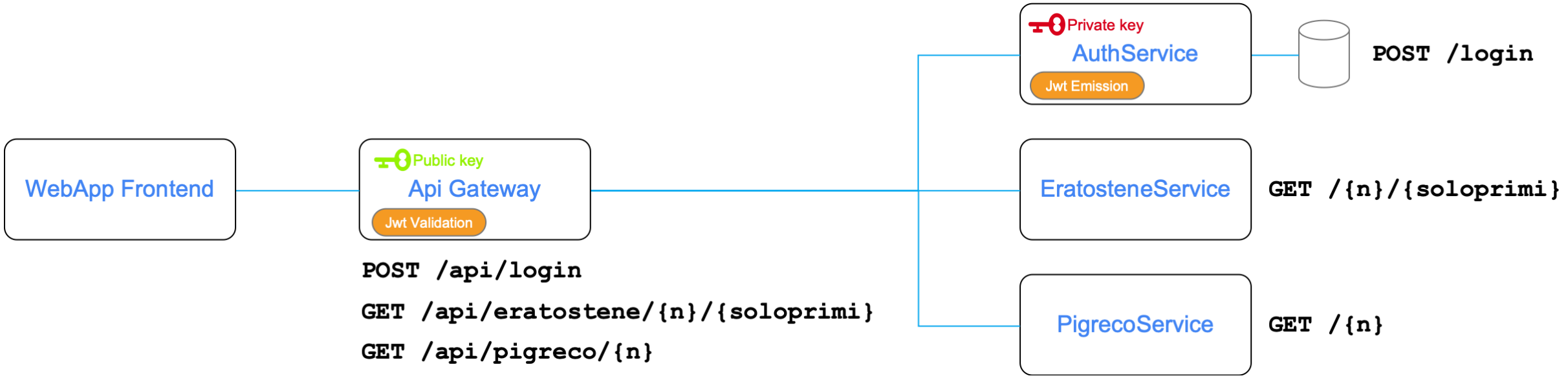
In una struttura a microservizi con API Gateway il controllo dell'autenticità del token può essere fatto in più modi, tra i principali:

1. L'api gateway inoltra il token al microservizio di destinazione il quale si deve occupare di validarlo prima di utilizzarlo
    1. Ogni microservizio deve quindi implementare la logica di verifica
    2. E' necessario che la firma del token avvenga con un algoritmo a chiave asimmetrica, condividendo la chiave pubblica tra tutti i microservizi

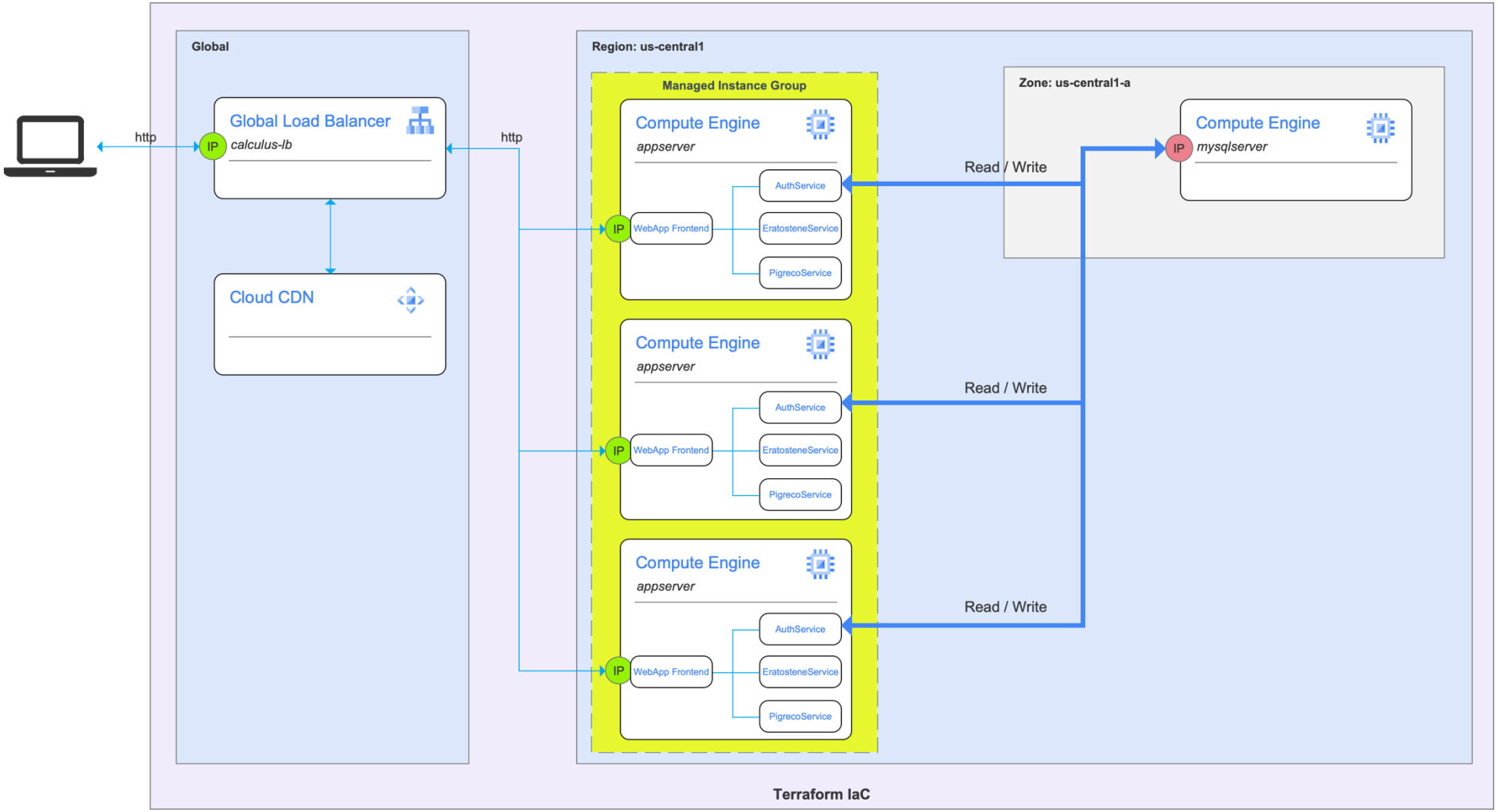
Autenticazione/Autorizzazione

2. L'api gateway valida il token e lo inoltra ai microservizi sono se è valido. I microservizi di destinazione lo possono usare senza ulteriori verifiche:
    1. I microservizi devono essere necessariamente privati e raggiungibili solo tramite gateway. Altrimenti non possono validare la correttezza del token
    2. Anche in questo caso è necessario che la firma del token avvenga con un algoritmo a chiave asimmetrica
    3. Il gateway può evitare di inoltrare il token ai microserviz se questi non ne hanno bisogno

Autenticazione/Autorizzazione



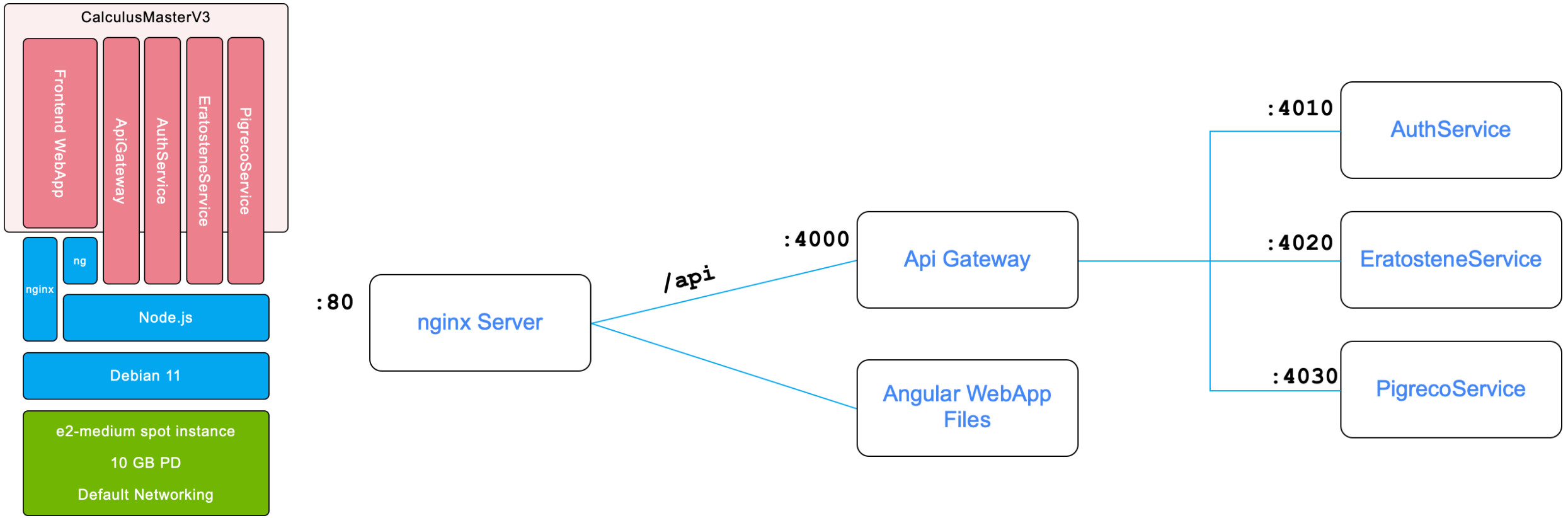
Deploy L - Microservizi



Deploy L - Microservizi

- Ogni istanza del MIG contiene tutti i microservizi (cambieremo approccio in futuro)
- Per poter eseguire l'applicazione frontend è necessaria l'installazione di angular e di un server web (nginx nel nostro caso)
- Per eseguire i microservizi è sufficiente la presenza di node.js
- I microservizi saranno in ascolto su porte bloccate dal firewall in modo da non essere raggiungibili direttamente
- Il server nginx sarà in ascolto sulla porta 80

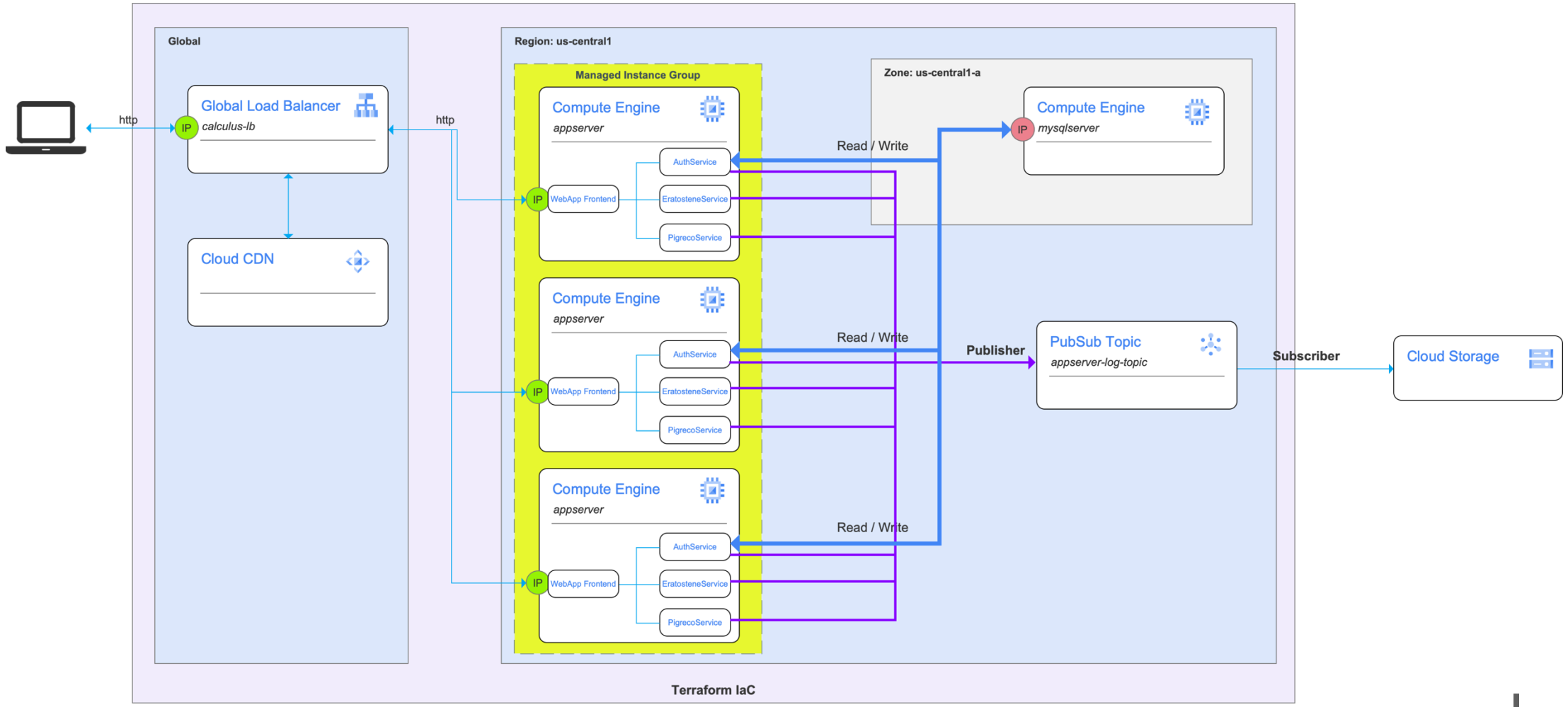
Deploy L - Microservizi



Deploy M – Logging con PubSub

- Supponiamo di voler aggiungere un sistema di log, per centralizzare tutti gli avvisi generati dai microservizi
- Il sistema di log può essere realizzato come ulteriore microservizio oppure in modo asincrono tramite Pub/Sub
- Realizziamo questa seconda opzione:
  - Ogni microservizio diventa un publisher
  - Il topic Pub/Sub salva una copia dei log anche su un bucket Cloud Storage (Cloud Storage diventa quindi un subscriber)

Deploy M – Logging con PubSub



Esercitazioni di laboratorio

- Extract, Analyze, and Translate Text from Images with the Cloud ML APIs
  - https://www.cloudskillsboost.google/focuses/1836?parent=catalog
- Pub/Sub: Qwik Start Console & Command Line
  - https://www.cloudskillsboost.google/focuses/3719?parent=catalog
  - https://www.cloudskillsboost.google/focuses/925?parent=catalog
  - https://cloud.google.com/pubsub/docs/publish-receive-messages-client-library