POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica
(Computer Engineering)

Politecnico
di Torino

Tesi di Laurea Magistrale

Implementazione di un'architettura a microservizi per un algoritmo di calcolo scientifico

Relatore	Candidato
Prof. Giovanni MALNATI	Federico BELTRAME

Luglio 2024

Abstract

Lo sviluppo di software, soprattutto in ambito distribuito, ha evidenziato alcune lacune tipiche delle architetture tradizionali. Una delle soluzioni più diffuse che rientrano in questa categoria prende il nome di monolitica, che ha come caratteristica principale quella di raggruppare in un singolo blocco tutti i servizi che il sistema offre. Questo risulta essere un problema considerevole, principalmente nella correzione di eventuali errori e nell'implementazione di nuove funzionalità nel codice. Questo lavoro di tesi ha come studio principale quello di trovare delle soluzioni alternative a quelle più tradizionali, e tra le varie ipotesi, si è scelto di sviluppare un'architettura a microservizi. Questa soluzione è molto utilizzata nell'ambiente cloud, soprattutto per i svariati vantaggi che essa offre rispetto ad altre soluzioni, come quella monolitica indicata in precedenza. Difatti, si può notare che un'architettura a microservizi permette di definire dei servizi di piccole dimensioni, ognuno con specifiche responsabilità e indipendenti gli uni dagli altri, ma in grado di comunicare tra loro in modo efficiente. Il fatto che l'intero sistema non sia relegato ad un singolo blocco, porta a favorire aggiornamenti e correzioni di eventuali errori senza interrompere l'intera esecuzione del programma. Tuttavia, bisogna prestare attenzione ad altre sfide, come ad esempio lo studio sui collegamenti tra i vari microservizi, in modo tale che essi possano comunicare. Tutto ciò risulta essere un problema accettabile, in quanto rimane un metodo efficiente per la creazione di software. Questo approccio, porta ad un cambiamento nella visione totale del sistema, favorendo uno sviluppo orizzontale aggiungendo repliche dei microservizi, a discapito di uno sviluppo più verticale basato sull'aumento della potenza di calcolo della singola macchina dove il servizio viene eseguito. Nella tesi viene presentata un'implementazione di un'architettura a microservizi, che sarà integrata con un programma chiamato EC700, software proprietario dell'azienda Edilclima. L'obiettivo è quello di creare un'architettura in grado di ospitare un programma che effettua calcoli scientifici, anche di complessità elevata in ambito computazionale, e che possa essere eseguito in ambiente cloud, in modo sicuro e ottimale. I risultati acquisiti testando l'intera soluzione sono incoraggianti, in quanto le prestazioni ottenute sono risultate positive.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Prima di analizzare nello specifico di cosa si vuole discutere in questa tesi, è buona prassi fare una breve introduzione sul programma principale coinvolto nello studio e di tutto ciò che ne consegue.

Il programma prende il nome di EC700, software proprietario e sviluppato dall'azienda Edilclima S.R.L., il quale presenta come obiettivo quello di calcolare le prestazioni energetiche degli edifici. Esso si basa su un calcolo scientifico, dove vengono mandati in input dei valori dai quali il motore ne calcola e restituisce dei risultati.

Esistono due tipologie di calcolo:

- stagionale: calcola su base mensile il fabbisogno energetico di un edificio o locale specificato, seguendo le normative tecniche UNI/TS 11300;
- orario: effettua il calcolo per ogni ora, con conseguente aumento nella precisione del calcolo e nelle stime del fabbisogno energetico di ogni edificio, seguendo la normativa UNI EN ISO 52016-1.

Attualmente quest'ultima tipologia di calcolo non è ancora obbligatoria a livello legislativo italiano o europeo, ma si ipotizza lo possa diventare nel breve periodo in modo da uniformare tutti gli Stati.

Tuttavia, un aumento nella precisione porta con sé, anche, alcuni aspetti problematici da tenere in considerazione. I dati di input risultano essere di dimensioni importanti in alcuni casi. Il calcolo dinamico orario risulta essere più complicato di quello stagionale, e necessita di un maggior tempo di risoluzione per ottenere i risultati. Risulta di primaria importanza che il cliente possa utilizzare il programma effettuando le operazioni in tempi ragionevoli, anche per svolgere progetti complessi.

Per risolvere il punto precedente, bisogna fare uso di tecnologie con risorse di elaborazione superiori rispetto a prima, che permettano di ridurre i tempi di esecuzione. Sicuramente, questa soluzione può migliorare il tempo di calcolo del

motore, ma porta con sé ulteriori problemi. Ovviamente, una dipendenza a livello di componenti tecnologici che deve disporre il computer dell'utente, poiché con risorse più limitate risulta essere infattibile il calcolo (scalabilità verticale). Inoltre, questo porta ad un aumento dei costi dovuto al miglioramento delle risorse computazionali per effettuare il calcolo.

Tutta questa catena di problematiche evidenzia punti chiave da tenere in considerazione, se si vuole fare in modo che il programma possa essere utilizzato in un modo ragionevole dal cliente, ed è su questi aspetti che si vuole far ruotare la tesi descritta di seguito nei prossimi capitoli.

Capitolo 2

Background e stato dell'arte

2.1 Motivazione della soluzione proposta

Facendo un breve elenco sulle problematiche evidenziate nell'introduzione, si può dire che:

- Il tempo dovuto al calcolo dinamico di tipo orario di un progetto risulta essere troppo elevato per essere considerato accettabile;
- Per ridurre il tempo, bisogna fare uso di risorse di elaborazione migliori;
- Si presenta una dipendenza computazionale del computer del cliente;
- Conseguente aumento dei costi da parte di quest'ultimo per ottenere le tecnologie migliori.

Tutti questi aspetti hanno portato allo studio di una soluzione migliore e che possa mitigare questi punti appena descritti.

Una delle soluzioni più vantaggiose, che corrisponde anche a quella pensata effettivamente, è il passaggio al Cloud Computing. Con questa scelta è possibile spostare nel cloud, ovvero su Internet, tutta la parte del motore di calcolo. Così facendo si possono sfruttare tutte le potenzialità che questo tipo di tecnologia offre.

Tra i vantaggi forniti, si risolvono i problemi principali spiegati all'inizio di questo paragrafo. Come si può osservare sul sito di Microsoft [1], nella sezione dedicata al Cloud Computing, tra i principali troviamo:

- Prestazioni: i servizi vengono eseguiti su data center sicuri, sparsi per il mondo, e aggiornati con hardware di ultima generazione. In questo modo si garantisce un servizio veloce ed efficiente;

- Velocità: i servizi vengono forniti nella maggior parte dei casi su richiesta, permettendo di avere maggiore flessibilità e minore attenzione alla pianificazione sull'aspetto della capacità;

- Costi: è possibile risparmiare sui costi per l'utilizzo di un servizio. In pratica, si paga solo per il tempo effettivo che si utilizza tale servizio, quindi in modo variabile, eliminando spese fisse.

Oltre ai tre precedenti che permettono di risolvere già i problemi enunciati in precedenza, il Cloud Computing offre altri vantaggi che orientano, sempre di più, la decisione verso questa scelta. Tra questi ci sono una maggiore affidabilità, grazie a vari backup sui dati che possono essere utilizzati in casi di emergenza; una maggiore sicurezza, dovuta al fatto che viene eseguita una protezione corposa e sempre aggiornata contro vari tipi di minacce; la possibilità di sviluppare una scalabilità di tipo orizzontale, puntando maggiormente su più istanze del medesimo servizio, rispetto a potenziare una macchina singola dove esso è in esecuzione.

Una volta appurato che la soluzione basata su Cloud Computing risulta, nel complesso, essere la più efficiente, bisogna creare un'architettura che possa "sposarsi" bene con questa scelta. L'architettura migliore per sfruttare le capacità appena descritte corrisponde all'architettura a microservizi, di cui si parlerà più approfonditamente, nel prossimo paragrafo. Essa, inoltre, coincide anche con l'oggetto di studio di questa tesi e la sua effettiva implementazione verrà spiegata più nel dettaglio nel prossimo capitolo (3).

2.2 Microservizi

2.2.1 Architettura a microservizi VS architettura monolitica

L'architettura a microservizi corrisponde ad un tipo di pattern molto utilizzato per lo sviluppo di applicazioni software [2]. I microservizi hanno preso sempre più piede nell'ambito del Cloud Computing e in altre tipologie di soluzioni. Le applicazioni sono strutturate come un insieme di componenti o servizi indipendenti, di piccole dimensioni e debolmente accoppiati, ognuno dei quali avente un ruolo ben specifico. Ogni microservizio, tendenzialmente, è collegato ad un singolo database e gestisce i dati che transitano su di esso. Quindi, è come se ogni servizio nascondesse i dettagli specifici di ciò che può svolgere, in modo tale che solo lui sappia come eseguire una determinata responsabilità, che ricadrà sull'intera applicazione.

Per effettuare comunicazioni tra loro vengono utilizzate API di tipo REST o altri tipi di servizi, tra i quali si ricordano i message broker che verranno impiegati anche nel progetto descritto in questo documento.

Questo tipo di architettura è in contrapposizione con un'altra soluzione antecedente chiamata monolitica. La principale differenza è dovuta al fatto che tutti i processi presenti al suo interno sono strettamente collegati e vengono eseguiti come un singolo servizio [3]. Favorisce lo sviluppo verticale, ovvero il potenziamento computazionale di una singola macchina.

Questo tipo di approccio di costruzione del codice è caratterizzato da vari svantaggi. Effettuare migliorie e aggiornamenti risulta più complicato, poiché si aumenta la mole di codice presente. Tutto ciò disincentiva la sperimentazione di nuove idee, poiché significherebbe andare ad aumentare il numero di righe di software, portando anche una sempre maggiore disorganizzazione del lavoro.

Un altro svantaggio si verifica con la presenza di un errore. Essendo tutti i processi strettamente collegati, rappresenta un fattore di rischio per la corretta esecuzione dell'applicazione. Quindi, un problema in una parte di codice può bloccare l'intera esecuzione del programma.

L'architettura a microservizi è stata pensata per risolvere le problematiche riscontrate in quella monolitica. Infatti, tra le caratteristiche principali si elencano le seguenti:

- Specializzazione: ogni componente svolge una funzione ben specifica, in modo tale che eventuali migliorie e/o aggiustamenti a linee di codice siano attuabili più semplicemente;
- Autonomia: tutti i componenti sono strettamente indipendenti, quindi in caso di errore o vulnerabilità si andrà a modificare solo la parte interessata, mantenendo attive le parti restanti dell'applicazione.

Una differenza sostanziale rispetto alla soluzione monolitica si riscontra nella visione dell'insieme del sistema, in quanto viene favorito uno sviluppo più orizzontale, rispetto a quello verticale definito in precedenza. Con esso si punta a creare diverse repliche del medesimo servizio, in modo da bilanciare maggiormente le risorse disponibili.

2.2.2 Vantaggi di un'architettura a microservizi

Spiegata la differenza tra architettura monolitica e architettura basata su microservizi, è tempo di scrivere quali sono i motivi per scegliere la seconda opzione rispetto alla prima.

I microservizi presentano vantaggi molto interessanti nello sviluppo delle applicazioni. Qui di seguito un piccolo elenco dei principali:

- Agilità: ogni microservizio può essere sviluppato da un ristretto gruppo di persone, responsabile del suo funzionamento, facilitando l'organizzazione

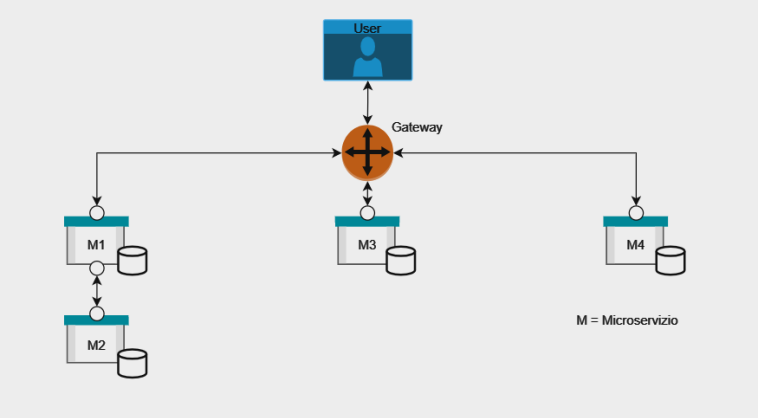

Figura 2.1: Esempio di architettura a microservizi

generale dell'applicazione. Inoltre, è possibile utilizzare diverse tecnologie, in base alla convenienza e alle competenze che possiedono le persone presenti nel team di sviluppo;

- Scalabilità: è possibile creare diverse istanze del medesimo servizio, in modo tale da soddisfare le richieste in entrata dell'applicazione;

- Riutilizzabilità: è possibile riutilizzare i servizi in varie parti del codice, essendo tutti indipendenti tra loro;

- Flessibilità: ogni microservizio è capace di adattarsi alle mutazioni delle condizioni provenienti dall'esterno o da altri microservizi;

- Gestione errori: è possibile correggere in modo isolato e organizzato un singolo microservizio, senza andare ad influire sull'intera applicazione e al normale funzionamento di essa.

È da tenere in considerazione, comunque, che tutti questi vantaggi sono accompagnati da ulteriori problematiche da non sottovalutare. Questa architettura, rispetto a quella monolitica, ha come principale ostacolo il fatto di dover studiare e, successivamente, implementare i vari collegamenti tra i microservizi che compongono l'applicazione, prestando particolare attenzione alla loro corretta realizzazione. Nonostante ciò, rimane una soluzione molto valida nello sviluppo di un'applicazione software, soprattutto in ambiente cloud based.

2.3 Apache Kafka

2.3.1 Cosa sono i message broker?

Nel paragrafo 2.2, si fa riferimento ai message broker come una delle modalità che è possibile utilizzare per far comunicare diversi microservizi tra loro. Come riportato sul sito di IBM [4], un message broker è un software che consente a varie applicazioni o servizi di comunicare tra loro scambiandosi informazioni. Esso gestisce anche la parte di trasformazione delle informazioni, in modo tale che diversi sistemi che utilizzano linguaggi differenti possano relazionarsi in modo corretto.

I message broker permettono di memorizzare, instradare e consegnare delle informazioni al destinatario corretto, senza che ci sia una correlazione diretta con il mittente. I messaggi vengono memorizzati in alcune code, pronti per essere letti da chi ne richiede l'utilizzo.

Uno dei vantaggi principali dei message broker riguarda la protezione da eventuali errori. Se per caso ci fossero problemi di connessione in una delle due parti coinvolte nella comunicazione, esso garantisce che i messaggi non vengano persi, non arrivino in ordine sparso e vengano inviati esattamente una volta. Questo tipo di comunicazione prende il nome di messaggistica asincrona.

Per capire maggiormente come funziona un broker, bisogna, necessariamente, fare riferimento al pattern Consumer-Producer, da cui prende spunto. Questo tipo di comunicazione è molto frequente in svariate applicazioni e architetture distribuite. Esso si basa su due componenti principali:

- Producer: colui che genera un'informazione o un dato;
- Consumer: colui che processa un'informazione o un dato.

Il vantaggio principale è il disaccoppiamento dei due componenti, fattore rilevante come detto in precedenza anche per i message broker.

Tra i message broker più diffusi si fa riferimento, sicuramente, ad Apache Kafka, di cui si discuterà qui di seguito.

2.3.2 Apache Kafka nel dettaglio

Apache Kafka è una piattaforma open source basata su eventi, utilizzata per sviluppare applicazioni in tempo reale. Permette di catturare dati da varie sorgenti, come ad esempio dai database, e di raccoglierne delle informazioni specificando uno schema, e viceversa. Inoltre, tiene traccia della data e dell'ora di memorizzazione.

Come ampiamente descritto sul sito ufficiale di Apache Kafka [5], questa piattaforma può essere utilizzata in vari contesti produttivi, combinando tre funzioni chiave:

- Implementa il pattern publish/subscribe;
- Memorizza i messaggi in modo duraturo e affidabile;
- Processa i messaggi quando richiesti.

Il pattern publish/subscribe corrisponde ad una variante del pattern producer/consumer descritto in precedenza. Questi due "attori", che corrispondono al mittente (publisher) e al destinatario (subscriber), comunicano attraverso un broker, che funge da intermediario. Il publisher crea delle informazioni e le invia al broker che può memorizzarle in alcune strutture, come ad esempio delle code. Il subscriber, invece, si registra ad una struttura presente sul broker, in modo tale che possa leggere i dati una volta disponibili. Questo tipo di comunicazione asincrona permette che i due componenti principali non si conoscano, favorendone il loro disaccoppiamento. Questo concetto ha il vantaggio di garantire un'alta scalabilità.

Tornando ad Apache Kafka, tutti i record vengono memorizzati e organizzati all'interno di topic. Essi sono simili a delle vere e proprie code, che possono essere manipolati da vari publisher e letti da vari subscriber. I dati rimangono memorizzati all'interno del topic anche dopo essere stati consumati per un tempo variabile, che può essere specificato nelle impostazioni di configurazione.

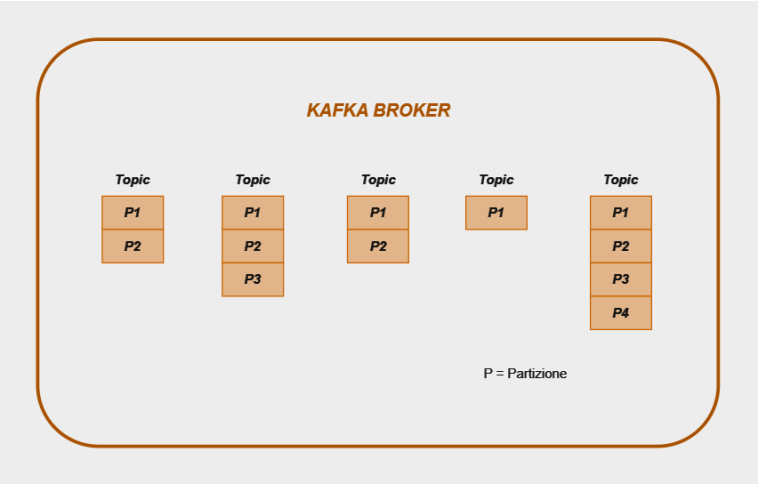

Figura 2.2: Esempio di topic e partizioni

Inoltre, è possibile dividere un topic in diverse partizioni, le quali possono essere salvate in broker diversi, aumentando la scalabilità delle operazioni. In questo modo si possono creare delle situazioni specifiche in base alle necessità. Ad esempio, è possibile che un producer scriva solo in determinate partizioni in base ad un valore key specifico.

Stesso discorso vale anche per i client che richiedono i dati. Esiste il concetto di Group-Id, che permette di dividere i subscriber in diversi gruppi. Si prenda come esempio un topic diviso in varie partizioni e diversi client iscritti allo stesso topic. Se i client possiedono lo stesso Group Id, le partizioni verranno spartite tra i vari client, mentre se possiedono Group Id diversi, tutti possono accedere a tutte le partizioni. Ovviamente, vale sempre la regola che un messaggio può essere letto una e una sola volta.

Kafka, inoltre, garantisce che i dati appartenenti alle varie partizioni vengano letti nell'esatto ordine di arrivo, seguendo una politica di tipo FIFO (First-In-First-Out).

Infine, per avere una protezione da eventuali guasti, è possibile creare delle repliche dei topic o delle singole partizioni, in diversi broker. In questo modo è sempre possibile avere una copia delle informazioni anche in caso di malfunzionamenti improvvisi. Questo processo permette di aumentare la sicurezza e l'affidabilità nell'utilizzo di Apache Kafka.

2.4 Keycloak

2.4.1 Autenticazione e autorizzazione

Un altro aspetto molto importante nello sviluppo di un'applicazione di questo tipo è, sicuramente, la gestione degli account dei clienti. Oltre al normale funzionamento del programma, risulta importante che un utente possa accedere ai propri servizi e che abbia i privilegi giusti per accedervi. Questi due aspetti appena citati corrispondono al significato di autenticazione e autorizzazione.

Andando un po' più nello specifico, con autenticazione si intende la capacità di un utente di confermare la propria identità all'interno di un servizio. Per fare ciò, nella maggior parte dei casi, può essere necessario indicare alcuni parametri relativi all'utente, come ad esempio username e password.

Invece, con autorizzazione si intende la capacità di un utente di poter accedere a determinati servizi o risorse. Per verificarlo si può, ad esempio, far riferimento ai ruoli definiti nell'identità di un utente. In questo modo, se un determinato cliente ha come ruolo "customer" non potrà accedere ai servizi specifici degli utenti "admin".

Quest'ultimo esempio, fa capire come sia importante gestire in modo ottimale gli account dei vari clienti. Uno strumento dedito a questo compito è chiamato Identity and Access Management, di cui si discuterà più nello specifico qui di seguito.

2.4.2 Identity and Access Management

Un Identity and Access Management, più comunemente chiamato solo IAM, è uno strumento che permette di gestire in modo ottimale i profili degli utenti collegati ad un'applicazione o servizio e tutte le funzionalità correlate ad essi. Tra le feature più importanti che uno IAM offre, oltre ad autenticazione e autorizzazione già citate in precedenza, troviamo:

- Gestione degli utenti: permette di creare, cancellare e raccogliere tutti gli account con i valori associati ad essi. Inoltre, permette di abilitare o disabilitare un utente, forzare il cambio di una password e molto altro;
- Single-Sign On: una volta che un utente ha effettuato il login con determinate credenziali, esse possono essere riutilizzate in diversi servizi. In questo modo non è necessario salvare una password diversa per ognuno di essi;
- Autenticazione a più fattori: oltre alle normali credenziali, viene richiesta una seconda opzione per autenticare in modo corretto un utente. Tra queste ricordiamo, ad esempio, un codice inviato tramite SMS o e-mail, oppure un'applicazione che svolge il ruolo di autenticatore, come può essere Microsoft Authenticator[1].

Uno degli IAM più conosciuti e all'avanguardia è Keycloak, che corrisponde anche a quello scelto per essere utilizzato in questo progetto.

2.4.3 Come funziona Keycloak?

Keycloak è uno IAM di tipo open source, tra i più interessanti nel panorama mondiale. Esso offre tutte le funzionalità che un normale IAM offre sulla gestione degli account degli utenti. Inoltre, dispone di un'interfaccia grafica facile ed intuitiva da utilizzare per gli amministratori degli account. Tra i vantaggi nell'utilizzo di Keycloak ci sono affidabilità e protezione della privacy dell'utente, soprattutto per il fatto che neanche un admin ha accesso alla password di un account. Keycloak, difatti, implementa i concetti di OAuth 2.0, di cui si andrà a parlare maggiormente nel sottoparagrafo successivo.

Come si può leggere dalla documentazione [6], Keycloak permette di gestire diversi realms. Esso permette di organizzare in modo separato e isolato un insieme di utenti, clienti, gruppi e ruoli. Risulta doveroso fare una precisazione sulla differenza tra un client e un user perché potrebbe essere forviante all'interno dell'ambiente di

[1]Maggiori dettagli sull'applicazione: https://support.microsoft.com/it-it/account-billing/informazioni-su-microsoft-authenticator-9783c865-0308-42fb-a519-8cf666fe0acc

Keycloak. Un user corrisponde alla persona/entità che richiede di effettuare un login, mentre il client è l'entità che richiede a Keycloak di autenticare un utente. Infine, per gruppi si intende un insieme di utenti che condividono gli stessi livelli di permessi, rappresentati dai ruoli che possiedono.

2.4.4 OAuth2.0 e Authorization Code Flow

OAuth 2.0 è il protocollo standard attuale per quanto riguarda l'autorizzazione. Viene utilizzato da Keycloak e permette ad un'applicazione (client) di accedere alle risorse specifiche di un utente, senza che quest'ultimo riveli le proprie informazioni segrete, tra le quali la propria password.

Ad esempio, un utente può effettuare il login con Keycloak per accedere a delle risorse private presenti su un sito, il quale non riceverà mai le credenziali, ma verrà avvisato dallo stesso IAM che l'utente ha i privilegi, e quindi l'autorizzazione, per accedervi.

Questo meccanismo favorisce un incremento notevole sotto l'aspetto della sicurezza e della protezione delle informazioni personali di un utente. Per quanto riguarda l'autenticazione, Keycloak implementa il concetto di OpenIDConnect, che coincide con un livello del protocollo OAuth 2.0. Esso permette di autenticare un utente o capire se è già autenticato e recuperare le informazioni di un utente dallo IAM, sotto forma di JWT token [7]. Prima di proseguire con il concetto di OAuth 2.0, è bene fare alcune precisazioni sugli "attori" che prendono parte a questo meccanismo di autenticazione e autorizzazione. Tra questi si descrivono i concetti di:

- Access Token: token di breve durata che contiene informazioni sull'utente, utilizzato per l'autorizzazione;
- ID Token: token che contiene informazioni identificative di un utente, utilizzato per l'autenticazione;
- Refresh Token: token di lunga durata, utilizzato per ottenere un nuovo access token una volta scaduto;
- Authorization Code: codice utilizzato per ottenere i vari token.

I token appena descritti sono anche chiamati JWT (JSON Web Token) e vengono utilizzati in base alla necessità.

Ritornando al discorso inerente all'autorizzazione, Keycloak implementa diversi tipi di Access Grant per recuperare i token di accesso, anche se quello standard prende il nome di Authorization Code Grant, il quale ha sostituito il precedente Implicit Grant ritenuto meno sicuro.

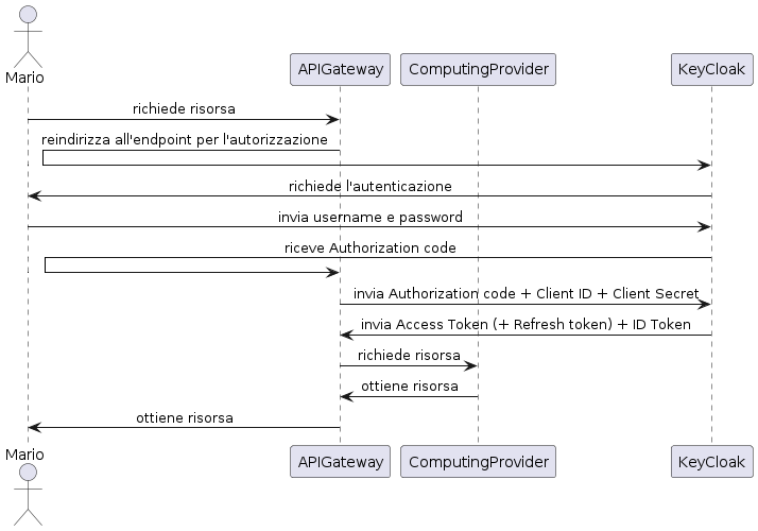

Figura 2.3: Funzionamento Authorization Code Flow

Prendendo spunto direttamente dalla documentazione di Keycloak [8], possiamo descrivere il funzionamento dell'Authentication Code Grant in questo modo:

1. Un utente vuole accedere ad una risorsa presente su un'applicazione, ma senza essersi autenticato;

2. L'utente, allora, viene reindirizzato alla pagina di login dove inserisce le proprie credenziali;

3. Una volta che esse saranno state verificate, Keycloak genererà un nuovo codice chiamato Authorization Code, il quale verrà inviato al Client;

4. Il Client, a sua volta, invierà Client ID (con il Secret nel caso di Client protetto) insieme all'Authorization Code generato in precedenza;

5. Una volta confermati anche questi ultimi, Keycloak restituirà al Client l'ID token e l'Access token (insieme al Refresh token se specificato) relativi all'utente che ha richiesto l'autenticazione;

6. In questo modo l'utente terminerà la fase di login e otterrà le risorse richieste.

L'Implicit Grant è stato deprecato perché, come spiegato nelle Best Practice di OAuth 2.0 [9], l'Access token veniva passato direttamente insieme all'Authorization Code, andando di fatto a saltare un passaggio rispetto all'Authorization Code Flow. In questo modo era possibile la perdita oppure il possesso e l'utilizzo dell'Access token da parte di malintenzionati. Con questo confronto, termina il paragrafo

relativo all'autenticazione e autorizzazione mediante Keycloak per passare alla trattazione di Docker nel prossimo.

2.5 Microservizi e container

2.5.1 Introduzione

L'avvento dell'architettura a microservizi ha fatto in modo che si sviluppassero a pari passo le piattaforme che gestiscono i container. Il software di punta in questo contesto prende il nome di Docker e corrisponde al punto focale della discussione presente nel paragrafo 2.5.

2.5.2 Docker

Docker è una piattaforma per lo sviluppo, la distribuzione e l'esecuzione di applicazioni [10]. Permette di separare le applicazioni dall'infrastruttura, in modo tale da facilitare e velocizzare la fase di sviluppo e di testing del codice. Esso verrà eseguito all'interno dei container. Il suo utilizzo si basa su alcuni concetti chiave, tra i quali:

- DockerFile: corrisponde ad un file dove vengono definite le istruzioni per creare un'immagine di Docker, con l'aggiunta di alcuni dati di configurazione come ad esempio le variabili d'ambiente.

- Docker Compose: permette di creare gruppi di container, che prendono il nome di multi-container. Viene creato un file con estensione ".yml", dove vengono definiti tutti i container con le relative immagini, che andranno a definire il gruppo. È possibile definire anche in questo caso alcune impostazioni di configurazione, quali variabili d'ambiente, volumi, dipendenze tra container, network, ecc...

Essendo i microservizi, generalmente, leggeri e di dimensioni ridotte, si sposano perfettamente con il concetto di container. Come ampiamente spiegato sul sito di IBM [11], i container corrispondono ad unità eseguibili di software, in cui viene impacchettato il codice applicativo con l'aggiunta di librerie e dipendenze. Questo processo prende il nome di containerizzazione. I container implementano una sorta di virtualizzazione, infatti vengono spesso paragonati con le Virtual Machine. Tra i vantaggi dei primi, si vuole indicare il fatto che sono piccoli, veloci e, soprattutto, portabili e indipendenti dalla piattaforma, in quanto non necessitano di definire un sistema operativo guest, in favore delle risorse rese disponibili dal sistema operativo dove sono definiti, comportamento tipico delle macchine virtuali.

Qui termina il capitolo 2, che ha permesso di capire meglio tutte le tecnologie utilizzate nel progetto trattato in questa tesi. Nel prossimo capitolo si andrà a descrivere dettagliatamente il sistema nella sua interezza, descrivendone architettura, comportamenti, componenti e molto altro.

Capitolo 3

Materiali e metodi

3.1 Architettura generale del progetto

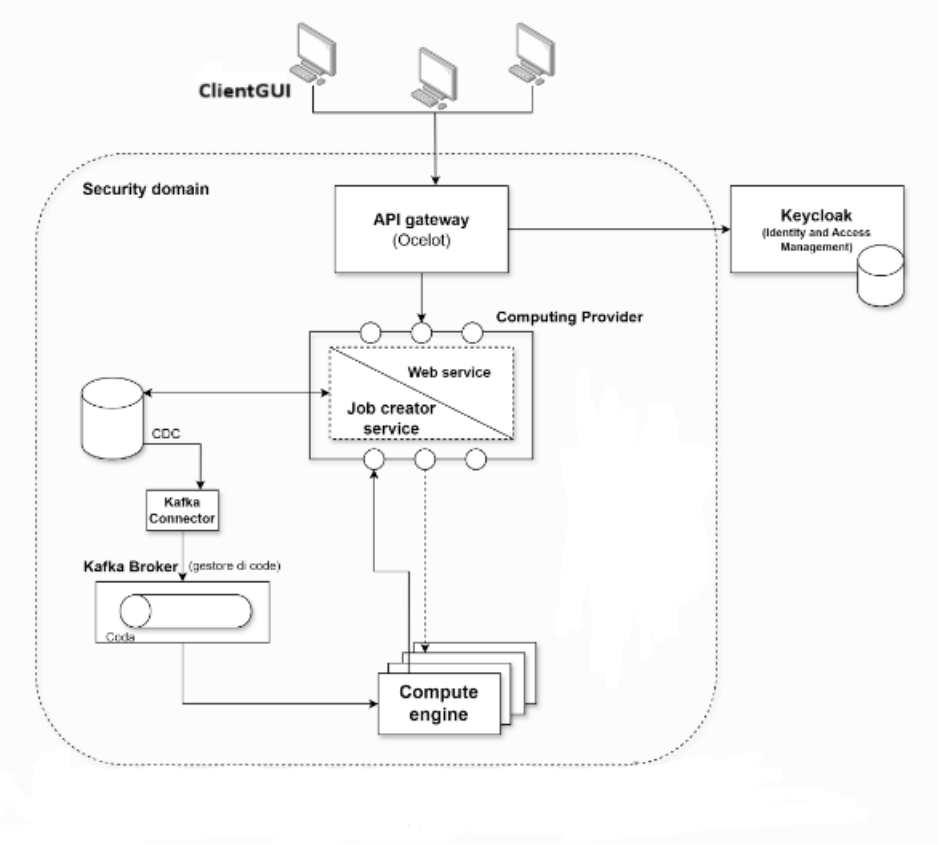

Figura 3.1: Architettura del progetto

3.1.1 Funzionamento generale

Nella figura 3.1 mostrata nella pagina precedente, si può osservare l'architettura generale del sistema trattato in questo progetto di tesi. È stato utilizzato .NET per sviluppare l'intero sistema, framework molto utilizzato nella creazione di applicazioni software. Questo, inoltre, è stato il primo ostacolo nella realizzazione del progetto e oggetto di studio preliminare, insieme alla creazione di soluzioni di prova precedenti a questo sistema.

Qui di seguito si vuole descrivere, senza scendere nello specifico, il funzionamento generale con l'obiettivo di sottomettere un task. Il funzionamento è il seguente:

1. Un cliente decide di sottomettere un nuovo task da mandare in esecuzione al motore per il calcolo dinamico orario.
2. La richiesta passa per l'API-gateway, il quale richiede che l'utente sia autenticato per poter continuare.
3. Una volta effettuato il login, la richiesta passa per il Computing Provider, il quale elabora la richiesta e la inserisce all'interno del database.
4. Kafka rileva che è stato sottomesso un nuovo task, il quale lo inserisce all'interno di un topic.
5. Il Compute Engine, che nel frattempo si è iscritto al topic, preleva il messaggio da Kafka e lo manda in esecuzione al motore di calcolo.
6. Compute Engine e Computing Provider si scambiano i risultati parziali del calcolo, i quali vengono salvati all'interno del database.
7. Alla fine del calcolo viene aggiornato lo status del task, il risultato completo risulta disponibile all'interno del database e pronto per essere visualizzato dal cliente.

Questi step vogliono rappresentare un punto di partenza rispetto a ciò che verrà raccontato nei prossimi paragrafi, dove si andrà più nello specifico nel descrivere ogni "attore" presente nell'architettura.

3.2 Computing Provider

3.2.1 Panoramica generale

Il Computing Provider è un'applicazione che fa parte del progetto per il calcolo orario delle prestazioni energetiche degli edifici. Essa si pone come intermediario tra il cliente e il Compute Engine (approfondito nel paragrafo 3.4).

Svolge il lavoro sia di Web Service che di Job Creator Service. Un Web Service corrisponde ad un'interfaccia software che offre una serie di funzionalità ad un client [12]. Per Job Creator Service, invece, si vuole intendere, di fatto, che offre il servizio di creazione di una nuova computazione (Job) quando il client effettua una richiesta di calcolo.

Il client invierà la richiesta per il calcolo delle prestazioni energetiche di un edificio al Computing Provider, il quale, prima di sottometterlo, effettuerà i vari controlli preliminari. Tra questi, ad esempio, ci sarà il controllo sull'autenticazione e sull'autorizzazione di un determinato utente (si andrà maggiormente nel dettaglio nel paragrafo 3.5 dedicato all'API-gateway), per verificare se egli ha i permessi per effettuare una richiesta.

Una volta passati tutti i controlli, verrà inserita la richiesta di calcolo dell'utente all'interno del database. Il database scelto per questo lavoro è PostgreSQL, un database di tipo relazionale avente al suo interno un insieme di tabelle, dove verranno salvati tutti i dati di input e i risultati (per ulteriori dettagli si rimanda al paragrafo 3.3 relativo al Database).

Quando il Compute Engine avrà effettuato le varie operazioni sui dati del progetto, invierà i risultati direttamente al Computing Provider, per poi restituirli al client che ne ha fatto richiesta.

Il Computing Provider mette a disposizione del client ulteriori servizi, oltre alla già citata richiesta di calcolo. Tra le altre funzionalità, è possibile salvare nel database i dati parziali di una computazione, come quello inerente alla singola ora, in modo tale che il client possa avere un'idea preliminare sul risultato completo del progetto.

Un altro servizio offerto, direttamente collegato al precedente, è la possibilità di interrompere anticipatamente il calcolo del progetto, dovuto al fatto che l'utente è già soddisfatto del risultato ottenuto oppure si è accorto di volere cambiare alcuni parametri relativi al lavoro inserito in precedenza.

Un'altra funzione correlata è la possibilità di richiedere alcune statistiche o effettuare calcoli preliminari sui calcoli già ottenuti, come ad esempio la percentuale di completamento delle operazioni.

Il Computing Provider permette di ottenere varie tipologie di informazioni e non solo inerenti ai task, ma anche agli utenti. L'applicazione offre altri tipi di servizi di cui, però, si farà riferimento in seguito.

Nei prossimi sottoparagrafi si andranno a descrivere più nel dettaglio la struttura del Computing Provider, partendo dai Model e i vari Data Transfer Object (DTO) implementati nel codice.

3.2.2 Model e DTO

Prima di entrare nel merito dei vari model presenti in questa applicazione, alcuni cenni su cosa sono effettivamente e su cosa rappresenta il pattern Model-View-Controller, anche noto come MVC, su cui si basano sia il Computing Provider che il Compute Engine.

Il pattern MVC permette di definire delle aree separate all'interno del codice di un programma, ognuna delle quali ha delle responsabilità ben precise e distinte [13]. Esso è composto da tre "attori" principali:

- Model: definisce i metodi che saranno utilizzati per accedere ai dati;
- View: permette di realizzare una rappresentazione visiva dei dati all'utente e regola l'interazione tra quest'ultimo e il sistema generale;
- Controller: accetta le richieste di un utente che necessita di una determinata risorsa, portando dei cambiamenti alla View, grazie all'interazione con i Model.

Tornando al microservizio analizzato, la View non è stata definita al livello del Computing Provider, ma è stata sostituita da un'altra applicazione con il ruolo di GUI, di cui si parlerà in seguito nel paragrafo 3.6.

Inoltre, sono state definite diverse classi che svolgono il ruolo di Model, le quali verranno rappresentate subito più avanti. Ogni attributo delle classi corrisponde ad un campo sulla tabella presente sul database. L'annotazione [Key] presente sopra un determinato attributo permette di definire che quel campo coincide con la chiave primaria della tabella.

La prima classe che si vuole rappresentare prende il nome di ComputingTask e corrisponde all'entità che mappa la tabella omonima sul database. Essa permette di raccogliere le informazioni inerenti ai dati di input di un Job.

```
1 public class ComputingTask
2 {
3     [Key]
4     public int Id { get; set; }
5     public required byte[] FileContent { get; set; }
6     public required DateTime TsCreation { get; set; }
7     public required string EmailCustomer { get; set; }
8 }
```

La spiegazione di ogni attributo è rinviata al capitolo inerente al database e alle sue tabelle.

La prossima classe che si vuole introdurre è chiamata ComputingTasksMetadata. Come da nome, essa permette di rappresentare i metadati, ovvero tutte le informazioni non pertinenti direttamente con il calcolo, come il nome del file, il tempo di Upload, ecc...

```
public class ComputingTasksMetadata
{
    [Key]
    public int Id { get; set; }
    public string FileName { get; set; }
    public string Status { get; set; }
    public DateTime TsUploaded { get; set; }
    public DateTime TsStarted { get; set; }
    public DateTime TsCompleted { get; set; }
}
```

Il terzo Model descritto prende il nome di Customer e rappresenta i clienti che vogliono sottomettere un lavoro.

```
public class Customer
{
    [Key]
    public string Email { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
    public int NumTasks { get; set; }
}
```

Infine, le informazioni dei risultati del calcolo vengono gestite dall'ultima classe rimanente chiamata RisultatiOra.

```
[PrimaryKey(nameof(Id_File), nameof(YY), nameof(MM), nameof(GG),
    nameof(HH))]
public class RisultatiOra
{
    public int Id_File { get; set; }
    public Int32 YY { get; set; }
    public Int32 MM { get; set; }
    public Int32 GG { get; set; }
    public Int32 HH { get; set; }
    public Int32 Res1 { get; set; }
    public Int32 Res2 { get; set; }
    public Int32 Res3 { get; set; }
    public int Time_Execution { get; set; }
    public int Step { get; set; }
}
```

Come si può vedere dal codice precedente sono presenti i valori Res1, Res2 e Res3. Essi rappresentano dei valori fittizi relativi ai risultati, scritti in questo modo per evitare di scrivere informazioni private. Da questo momento in poi i risultati verranno sempre caratterizzati con questa nomenclatura.

Di solito, per comunicare con il livello di servizio non viene utilizzato direttamente il Model, bensì si fa uso dei cosiddetti Data-Transfer-Object, più comunemente chiamati DTO. Prima di descrivere i DTO presenti, è doveroso fare alcuni accenni sul loro scopo.

I DTO sono oggetti che definiscono come i dati verranno trasferiti nella rete [14]. Questo pattern presenta vari vantaggi per i quali risulta comodo il suo utilizzo, tra i quali:

- Nascondere alcune informazioni che gli utenti non devono poter visualizzare;
- Omettere alcune proprietà per ridurre le informazioni da inviare, abbassando le dimensioni;
- Delineare una separazione tra il livello di servizio e quello di Data Access.

In questo caso specifico, sono stati definiti diversi DTO, utilizzati in diverse parti del codice in base alle funzionalità richieste.

Partendo dal model ComputingTask, sono stati definiti tre diversi DTO: ComputingTaskInputDatatDto, utilizzato quando vengono richiesti i dati di input di un progetto; ComputingTaskNewDto, utilizzato all'atto della sottomissione di un nuovo lavoro; ComputingTaskSubmissionDto, utilizzato quando si vogliono recuperare tutti i lavori sottomessi in precedenza.

```
1  public class ComputingTaskInputDataDto
2  {
3
4      public ComputingTaskInputDataDto(int Id, string FileContent,
       DateTime TsCreation, string EmailCustomer)
5      {
6          this.Id = Id;
7          this.FileContent = FileContent;
8          this.TsCreation = TsCreation;
9          this.EmailCustomer = EmailCustomer;
10     }
11
12     [Key]
13     public int Id { get; set; }
14     public string FileContent { get; set; }
15     public DateTime TsCreation { get; set; }
16     public string EmailCustomer { get; set; }
17 }
18
19 public class ComputingTaskNewDto
20 {
21     public byte[] FileContent { get; set; }
```

```
22      public string EmailCustomer {  get; set; }
23  }
24
25  public class ComputingTaskSubmissionDto
26  {
27      [Key]
28      public int Id { get; set; }
29      public required DateTime TsCreation { get; set; }
30      public required string EmailCustomer { get; set; }
31  }
```

Il DTO seguente prende il nome di RisultatiOraStatsDto e deriva dalla classe RisultatiOra, dove vengono rappresentati solo tre campi rispetto all'intero model.

```
1  public class RisultatiOraStatsDto
2  {
3      [Key]
4      public int Id_File { get; set; }
5      public int Time_Execution { get; set; }
6      public int Step { get; set; }
7  }
```

SubmissionAndMetadataDto è un DTO diverso, rispetto agli altri appena rappresentati. Esso, infatti, rappresenta l'unione di due model, dove alla classe ComputingTaskMetadata sono stati aggiunti i campi TsCreated e EmailCustomer presenti nella tabella ComputingTask.

```
1  public class SubmissionAndMetadataDto
2  {
3      public SubmissionAndMetadataDto(int id, string fileName, string
       status, DateTime tsUploaded, DateTime tsCreation, DateTime
       tsStarted, DateTime tsCompleted, string emailCustomer)
4      {
5          Id = id;
6          FileName = fileName;
7          Status = status;
8          TsUploaded = tsUploaded;
9          TsCreation = tsCreation;
10         TsStarted = tsStarted;
11         TsCompleted = tsCompleted;
12         EmailCustomer = emailCustomer;
13     }
14
15     [Key]
16     public int Id { get; set; }
17     public string FileName { get; set; }
18     public string Status { get; set; }
```

```
19      public DateTime TsUploaded { get; set; }
20      public DateTime TsCreation { get; set; }
21      public DateTime TsStarted { get; set; }
22      public DateTime TsCompleted { get; set; }
23      public string EmailCustomer { get; set; }
24  }
```

Come si può ben vedere, tutti i DTO presentano una parte degli attributi delle classi definite poc'anzi, con l'esclusione degli altri che vengono nascosti al client poiché non necessari allo scopo in questione. Quando il client effettua una nuova richiesta, i campi non presenti, se necessari allo scopo, vengono popolati in automatico da valori inseriti direttamente nel codice. Un esempio è il campo TsCreated quando si vuole sottomettere un nuovo task. Esso non viene popolato dal cliente, ma appena prima che il nuovo task venga salvato nel database.

Per mappare i DTO con i Model è stata definita la classe MappingProfiles, che deriva dalla classe di base Profile del pacchetto AutoMapper. Con essa è possibile effettuare il mapping tra due oggetti utilizzando il metodo CreateMap, con il quale si associa l'oggetto sorgente con quello di destinazione, rispettivamente il primo e il secondo argomento del metodo. Quindi, per avere una correlazione bilaterale, è necessario dichiarare due volte il medesimo metodo con gli argomenti invertiti.

```
1  public class MappingProfiles : Profile
2  {
3      public MappingProfiles()
4      {
5          CreateMap<ComputingTask, ComputingTaskNewDto>();
6          CreateMap<ComputingTaskNewDto, ComputingTask>();
7          CreateMap<ComputingTask, ComputingTaskSubmissionDto>();
8          CreateMap<ComputingTaskSubmissionDto, ComputingTask>();
9          CreateMap<RisultatiOra, RisultatiOraStatsDto>();
10         CreateMap<RisultatiOraStatsDto, RisultatiOra>();
11     }
12 }
```

Terminata la sezione dedicata ai Model e ai DTO, nel prossimo sottoparagrafo si entrerà nel dettaglio del Controller, secondo ruolo facente parte del pattern MVC descritto in precedenza.

3.2.3 Livello di presentazione

Il controller, generalmente, riceve delle richieste da un client e permette di generare dei cambiamenti sui dati. Esso offre delle API che permettono di chiamare dall'esterno dell'applicazione delle azioni contattando l'URL correlato [15]. Con azioni si intendono dei metodi pubblici definiti all'interno della classe del controller.

Ogni metodo possiede uno specifico HTTP verbs, il quale serve a descrivere che tipo di azione effettuerà sulle risorse [16]. Questi metodi HTTP permettono di svolgere delle operazioni di tipo CRUD (Create, Read, Update e Delete). Esistono 5 tipi principali:

- GET: permette di ottenere dei dati;
- POST: permette di creare una nuova risorsa;
- PUT: permette di sostituire una risorsa esistente con un'altra;
- PATCH: permette di aggiornare una risorsa esistente senza sostituirla;
- DELETE: permette di eliminare una risorsa.

Nel contesto dell'applicazione Computing Provider, la classe del controller prende il nome di ComputingProviderController e definisce tutte le azioni che possono essere chiamate dall'esterno. Questa classe deriva dalla classe di base Controller.

L'analisi dei servizi chiamati all'interno dei metodi riportati di seguito è rimandata al sottoparagrafo successivo.

Ogni metodo è preceduto dall'annotazione [Authorize(Policy="...")], il quale permette di definire il livello di privilegio che un utente deve possedere per accedere ad una determinata risorsa. Attualmente, sono definite due tipologie di policy: Administration, dove solo gli admin sono inclusi, e Customer, dove sono ammessi i precedenti e i clienti normali. Per una spiegazione più approfondita sull'autenticazione e sull'autorizzazione si rimanda al paragrafo 3.5.

Qui di seguito verranno elencati i metodi presenti nella classe del controller.

GetCustomers()

```
[Authorize(Policy = "Administration")]
[HttpGet("/customers")]
[ProducesResponseType(200, Type = typeof(IEnumerable<Customer>))]
public IActionResult GetCustomers()
{
    var customers = _customerService.GetCustomers();

    return Ok(JsonConvert.SerializeObject(customers));
}
```

Permette di restituire la lista di tutti i clienti che hanno sottomesso almeno un lavoro in formato JSON. Come si può vedere dal codice, corrisponde ad un metodo richiamabile solo da un utente admin. In caso di esito negativo, può restituire anche uno status code di errore (400, Bad Request).

GetComputingTaskAdminSubmission()

```
[Authorize(Policy = "Administration")]
[HttpGet("/tasks/administration/submission")]
[ProducesResponseType(200, Type = typeof(List<
    SubmissionAndMetadataDto>))]
public IActionResult GetComputingTaskAdminSubmission()
{

    var computingTaskSubmission = _computingProviderService.
    GetComputingTaskAdminSubmission();
    var allMetadata = _computingTasksMetadataService.GetAllMetadata()
    ;
    var submissionAndMetadata = _computingProviderService.
    GetSubmissionAndMetadata(computingTaskSubmission, allMetadata);

    return Ok(JsonConvert.SerializeObject(submissionAndMetadata));
}
```

Permette di restituire la lista di task sottomessi da tutti i clienti. Per soddisfare la richiesta, risulta necessario combinare i dati presenti in due tabelle distinte utilizzando SubmissionAndMetadataDto, descritto nella sezione relativa ai model e ai DTO. Come nel caso precedente, si tratta di un endpoint riferito agli utenti admin e in caso di esito negativo può restituire il codice 400.

GetComputingTaskResult()

```
[Authorize(Policy = "Customer")]
[HttpGet("/tasks/{id}/result")]
[ProducesResponseType(200, Type = typeof(IEnumerable<RisultatiOra>))]
public IActionResult GetComputingTaskResult([FromRoute] int id)
{

    if (!_resultsService.ComputingTaskResultsExists(id))
        return NotFound();

    var computingTaskResults = _resultsService.
    GetComputingTaskResultsById(id);

    var emailCustomerTask = _computingProviderService.
    GetEmailCustomerById(id);
    var emailUser = User.Claims.Where(c => c.Type == "http://schemas.
    xmlsoap.org/ws/2005/05/identity/claims/emailaddress").
    FirstOrDefault().Value;
    if (User.Claims.Where(c => c.Type == "realm_roles").
    FirstOrDefault(r => r.Value == "customer") != null
```

```
15          && emailCustomerTask != emailUser)
16          return Unauthorized();
17
18      return Ok(JsonConvert.SerializeObject(computingTaskResults));
19  }
```

Permette di restituire i risultati di un determinato task, specificandone l'id. Vengono effettuati vari controlli prima di restituire i dati richiesti, ognuno dei quali, se non passato, genera un codice di errore diverso. Andando più nello specifico, se il task non esiste restituisce status code 404 (Not Found), se un utente tenta di accedere al task di un altro cliente restituisce 401 (Not Authorized), mentre nel caso di errore nella definizione dei parametri restituisce 400 (Bad Request).

GetComputingTaskResultCSV()

```
1  [Authorize(Policy = "Customer")]
2  [HttpGet("/tasks/{id}/resultCSV")]
3  [ProducesResponseType(200)]
4  public IActionResult GetComputingTaskResultCSV([FromRoute] int id)
5  {
6      if (!_resultsService.ComputingTaskResultsExists(id))
7          return NotFound();
8
9      var computingTaskResults = _resultsService.
       GetComputingTaskResultsById(id);
10
11     var emailCustomerTask = _computingProviderService.
       GetEmailCustomerById(id);
12     var emailUser = User.Claims.Where(c => c.Type == "http://schemas.
       xmlsoap.org/ws/2005/05/identity/claims/emailaddress").
       FirstOrDefault().Value;
13     if (User.Claims.Where(c => c.Type == "realm_roles").
       FirstOrDefault(r => r.Value == "customer") != null
14         && emailCustomerTask != emailUser)
15         return Unauthorized();
16
17     var content = _resultsService.CreateCSVFile(computingTaskResults)
       ;
18
19     var contentType = "APPLICATION/octet-stream";
20     var fileName = "Results.csv";
21     return File(content, contentType, fileName);
22 }
```

Corrisponde alla stessa richiesta descritta nel metodo precedente, con la differenza che questo permette di effettuare il download di un file di tipo CSV,

contenente i risultati di un task con l'id specificato. Anche i controlli sono i medesimi dell'endpoint precedente.

GetComputingTaskInputData()

```
[Authorize(Policy = "Customer")]
[HttpGet("/tasks/{id}/input_data")]
[ProducesResponseType(200, Type = typeof(ComputingTaskInputDataDto))]
public IActionResult GetComputingTaskInputData([FromRoute] int id)
{
    if (!_computingProviderService.ComputingTaskExists(id))
        return NotFound();

    var computingTaskInputData = _computingProviderService.
    GetComputingTaskInputData(id);

    var emailUser = User.Claims.Where(c => c.Type == "http://schemas.
    xmlsoap.org/ws/2005/05/identity/claims/emailaddress").
    FirstOrDefault().Value;
    if (User.Claims.Where(c => c.Type == "realm_roles").
    FirstOrDefault(r => r.Value == "customer") != null
        && computingTaskInputData.EmailCustomer != emailUser)
            return Unauthorized();

    return Ok(JsonConvert.SerializeObject(computingTaskInputData));
}
```

Permette di restituire un JSON contenente i dati di input di un lavoro sottomesso da un utente, passando come parametro l'id.

GetComputingTaskCustomerSubmission()

```
[Authorize(Policy = "Customer")]
[HttpGet("/tasks/customer/submission")]
[ProducesResponseType(200, Type = typeof(List<
    SubmissionAndMetadataDto>))]
public IActionResult GetComputingTaskCustomerSubmission()
{
    var userEmail = User.Claims.Where(c => c.Type == "http://schemas.
    xmlsoap.org/ws/2005/05/identity/claims/emailaddress").
    FirstOrDefault().Value;
    Customer customer = _customerService.CustomerExist(userEmail);
    if (customer == null)
        return NotFound();
    var computingTaskSubmission = _computingProviderService.
    GetComputingTaskCustomerSubmission(userEmail);
```

```
11      var allMetadata = _computingTasksMetadataService.GetAllMetadata()
        ;
12      var submissionAndMetadata = _computingProviderService.
        GetSubmissionAndMetadata(computingTaskSubmission, allMetadata);
13
14      return Ok(JsonConvert.SerializeObject(submissionAndMetadata));
15  }
```

Permette di restituire la lista di tutti i task sottomessi da un determinato utente. Corrisponde allo stesso endpoint visto per gli admin, con la differenza che non vengono visualizzati i lavori di tutti i clienti.

GetComputingTaskStats()

```
1   [Authorize(Policy = "Customer")]
2   [HttpGet("/tasks/{id}/stats")]
3   [ProducesResponseType(200, Type = typeof(string))]
4   public IActionResult GetComputingTaskStats([FromRoute] int id)
5   {
6       if (!_computingProviderService.ComputingTaskExists(id))
7           return NotFound();
8
9       if (!_resultsService.ComputingTaskResultsExists(id))
10      {
11          return Ok(JsonConvert.SerializeObject(new
12          {
13              percentage = 0.0,
14              eta = 0,
15          }));
16      }
17      var lastResult = _resultsService.GetLastResultById(id);
18
19      var computingTaskStats = _resultsService.GetComputingTaskStats(
        lastResult);
20
21      return Ok(computingTaskStats);
22  }
```

Questo endpoint permette di restituire alcune statistiche relative al task con l'id specificato, ovvero la percentuale di completamento e il tempo stimato di completamento. Restituisce il codice di errore Not Found se il task con l'id inserito nell'URL non esiste. Se il task esiste, ma non è ancora stato mandato in esecuzione o ci si trova nella fase di precalcolo del motore, restituisce i valori delle statistiche azzerati.

AddComputingTask()

```
[Authorize(Policy = "Customer")]
[HttpPost("/task")]
[ProducesResponseType(200, Type = typeof(int))]
[RequestSizeLimit(500_000_000)]
public IActionResult AddComputingTask([FromForm] IFormFile file, [FromForm] long TsUploaded, [FromForm] string fileName)
{
    if (file == null)
        return BadRequest();

    try
    {
        if (file.Length < 0)
            return BadRequest();
        var TsUploadedDT = DateTimeOffset.FromUnixTimeMilliseconds(TsUploaded).DateTime.ToUniversalTime();
        var fileStreamHash = file.OpenReadStream();
        using (var md5 = MD5.Create())
        {
            var hash = md5.ComputeHash(fileStreamHash);
            Console.WriteLine(BitConverter.ToString(hash).Replace("-", "").ToLowerInvariant());
        }
        fileStreamHash.Close();

        using var fileStream = file.OpenReadStream();
        byte[] bytes = new byte[file.Length];
        fileStream.Read(bytes, 0, (int)file.Length);
        byte[] r1 = Utilities.ZipService.Zip(bytes);
        var userEmail = User.Claims.Where(c => c.Type == "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress").FirstOrDefault().Value;
        Customer customer = _customerService.CustomerExist(userEmail);
        if (customer == null)
        {
            Customer newCustomer = new Customer();
            newCustomer.Email = userEmail;
            newCustomer.Name = User.Claims.Where(c => c.Type == "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname").FirstOrDefault().Value;
            newCustomer.Surname = User.Claims.Where(c => c.Type == "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname").FirstOrDefault().Value;
            newCustomer.NumTasks = 1;
            if (!_customerService.AddCustomer(newCustomer))
            {
```

```
38                  ModelState.AddModelError("", "Something went wrong
    while adding a new customer");
39                  return StatusCode(500, ModelState);
40              }
41          }
42          else
43          {
44              customer.NumTasks++;
45              if (!_customerService.UpdateCustomer(customer))
46              {
47                  ModelState.AddModelError("", "Something went wrong
    while updating a customer");
48                  return StatusCode(500, ModelState);
49              }
50          }
51          ComputingTaskNewDto computingTask = new ComputingTaskNewDto()
    ;
52          computingTask.FileContent = r1;
53          computingTask.EmailCustomer = userEmail;
54          var taskId = _computingProviderService.AddComputingTask(
    computingTask);
55          if (!_computingTasksMetadataService.AddFileMetadata(taskId,
    fileName, TsUploadedDT))
56          {
57              ModelState.AddModelError("", "Something went wrong while
    adding a new customer");
58              return StatusCode(500, ModelState);
59          }
60          return Ok(taskId);
61      }
62      catch (InvalidAddTaskException ex)
63      {
64          switch (ex.StatusCode)
65          {
66              case 500:
67                  ModelState.AddModelError("", "Something went wrong
    while saving the computing task");
68                  return StatusCode(500, ModelState);
69              default:
70                  return BadRequest(ex);
71          }
72      }
73  }
```

Questo metodo è il primo e unico di tipo POST dell'intera applicazione e permette di sottomettere un nuovo task. Esso riceve in ingresso il file contenente gli input del calcolo, il quale viene compresso per ridurne la dimensione.

Successivamente, si effettua il controllo sul cliente che ha richiesto la sottomissione. Se si tratta del primo task sottomesso, egli viene salvato nella tabella

dedicata agli utenti, altrimenti viene semplicemente aggiornato il contatore che tiene traccia del numero totale di lavori sottomessi.

Prima di aggiungere il nuovo task nel database, viene creata anche una nuova riga in un'altra tabella contenente i metadati. Un maggiore approfondimento sul contenuto di tutte le tabelle verrà presentato nel prossimo paragrafo, dedicato al database.

In caso di esito negativo in qualche operazione relativa al database viene restituito il codice di errore 500. Mentre, viene generato l'errore BadRequest se non è presente il file tra i valori in ingresso o se non presenta contenuto. In caso di esito positivo viene restituito l'Id del task sottomesso collegato alla tupla presente nella tabella.

Gli endpoint descritti fino a questo momento gestivano comunicazioni esclusivamente tra l'API-gateway e il Computing Provider. Gli ultimi metodi descritti, invece, hanno il compito principale di comunicare o ricevere informazioni da e verso il Compute Engine.

UpdateFileMetadata()

```
1  [HttpPut("/task/fileMetadata")]
2  public IActionResult UpdateFileMetadata([FromBody]
       ComputingTasksMetadata newMetadata)
3  {
4      if (newMetadata == null || !ModelState.IsValid)
5          return BadRequest(ModelState);
6
7      ComputingTasksMetadata oldMetadata =
       _computingTasksMetadataService.GetMetadata(newMetadata.Id);
8      if (oldMetadata == null)
9      {
10         return NotFound();
11     }
12
13     if (!_computingTasksMetadataService.UpdateMetadata(newMetadata,
       oldMetadata))
14     {
15         ModelState.AddModelError("", "Something went wrong updating
       metadata");
16         return StatusCode(500, ModelState);
17     }
18
19     return NoContent();
20 }
```

Unico metodo di tipo PUT, permette di aggiornare la tabella contenente i metadati di un Job. È l'unica API chiamata dal Compute Engine. Restituisce NotFound nel caso non fossero stati trovati metadati con l'id specificato, BadRequest

alla presenza di errori dovuti alla validazione dei parametri o codice di errore 500 in caso di mancato aggiornamento della tabella nel database.

StopPartialTask()

```
[Authorize(Policy = "Customer")]
[HttpDelete("/tasks/{id}")]
public IActionResult StopPartialTask([FromRoute] int id)
{
    if (!_computingProviderService.ComputingTaskExists(id))
        return NotFound();

    if (!_computingProviderService.StopPartialTask(id).Result)
        return BadRequest();
    return NoContent();
}
```

Viene chiamato da un utente che ha intenzione di interrompere, anticipatamente, il calcolo di un task in esecuzione. Questo metodo permette di avvisare il Compute Engine di tale richiesta. Se il task specificato dall'id non esiste restituirà NotFound, mentre in caso di errore dovuto alla comunicazione con il Compute Engine restituirà BadRequest.

Arrivati a questo punto, è terminata la parte relativa al livello di presentazione del Computing Provider. Come già affermato in precedenza, i metodi chiamati all'interno delle varie funzioni verranno gestiti a livello di servizio e spiegati nel sottoparagrafo che segue.

3.2.4 Livello di servizio

Seguendo le best practice nello sviluppo di un'applicazione web, sono stati definiti diversi livelli. Ognuno di essi possiede un compito ben specifico, in modo tale da avere una gestione ordinata del codice. Ogni livello avrà la possibilità di comunicare con il livello subito adiacente tramite le interfacce.

Qui di seguito sono descritti i vari livelli e quale funzione principale ricoprono.

Il Presentation layer si occupa di gestire tutte le richieste provenienti dall'esterno dell'applicazione e di offrire le relative risposte. In questo caso, tale livello è fornito dal Controller di cui si è già parlato in precedenza.

Il Service layer, di cui si discuterà qui di seguito, contiene tutta la business logic dell'applicazione. Preferibilmente si utilizzano i DTO per manipolare i dati, invece di usare direttamente i Model.

Il Data Access layer comunica direttamente con le entità presenti sul database, implementando le operazioni CRUD. Questo livello sarà oggetto di studio del prossimo sottoparagrafo.

Figura 3.2: Esempio di interfaccia presente nel codice

Questa suddivisione permette di modificare il codice solo in un determinato blocco senza intaccare i restanti, riducendo tempi e facilitandone la manutenzione.

Terminata questa premessa sull'architettura dell'applicazione, si può entrare nel merito delle informazioni gestite nel livello di servizio.

I vari metodi utilizzati in questa applicazione sono stati divisi in diversi file, in base a quale tabella o tipo di informazione gestiscono.

Partendo dal file ComputingProviderService, esso offre i seguenti metodi:

AddComputingTask()

```
1  public int AddComputingTask(ComputingTaskNewDto computingTask)
2  {
3      var computingTaskMap = _mapper.Map<ComputingTask>(computingTask);
4      computingTaskMap.TsCreation = DateTime.Now.ToUniversalTime();
5      var newTaskId = _computingProviderRepository.AddComputingTask(
       computingTaskMap);
6      if (newTaskId == 0)
7          throw new InvalidAddTaskException(500, "Something went wrong
       while saving the computing task");
8      Task.Run(() => RetrievePartialData(newTaskId));
9      return newTaskId;
10 }
```

AddComputingTask riceve in ingresso un oggetto di tipo ComputingTaskNewDto e permette di aggiungere un nuovo task. Successivamente, se l'inserimento è andato a buon fine, chiama il metodo RetrievePartialData assegnandolo ad un altro thread tramite l'istruzione Task.Run() e ritorna l'identificativo del task appena creato.

RetrievePartialData()

```
public async Task RetrievePartialData(int id)
{
    try
    {
        using var scope = _serviceScopeFactory.CreateScope();
        var repository = scope.ServiceProvider.GetRequiredService<
    IResultsRepository>();
        using var streamReader = new StreamReader(await Client.
    GetStreamAsync("subscriptions/" + Guid.NewGuid().ToString() + "/
    tasks/" + id.ToString() + "/partialResult"));
        Console.WriteLine($"START TRASMISSION OF PARTIAL DATA");
        List<RisultatiOra> results = [];
        var time = DateTime.Now;
        while (!streamReader.EndOfStream)
        {
            var message = streamReader.ReadLine();
            var result = JsonConvert.DeserializeObject<RisultatiOra>(
    message);
            results.Add(result);
            if ((DateTime.Now - time).TotalSeconds > 5 & results.
    Count != 0)
            {
                time = DateTime.Now;
                if (!repository.AddPartialResult(results))
                    Console.WriteLine($"ERROR DURING THE UPDATE OF
    THE DB");
                else
                    results.Clear();
            }
        }
        streamReader.Close();
        if (results.Count != 0)
        {
            if (!repository.AddPartialResult(results))
                Console.WriteLine($"ERROR DURING THE UPDATE OF THE DB
    ");
            else
                results.Clear();
        }
        Console.WriteLine($"END TRASMISSION OF PARTIAL DATA");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error: {ex.Message}");
    }
}
```

RetrievePartialData permette di effettuare una richiesta HTTP all' endpoint di tipo Server Sent Event presente sul Compute Engine (di cui se ne parlerà nel paragrafo

3.4). Come risposta otterrà i dati parziali del calcolo di un task, che verranno salvati prima in una lista temporanea e, successivamente, all'interno del database.

È stato necessario utilizzare un oggetto di tipo IServiceScopeFactory, che permette di definire l'interfaccia del repository dei risultati (di cui si parlerà nel sottoparagrafo successivo) creando un nuovo scope. Si effettua questo passaggio perché la funzione è stata chiamata nello stesso metodo, presente nel controller, che permette l'aggiunta di un nuovo task. Il problema risulta essere che una volta terminato l'inserimento, con conseguente terminazione anche del metodo e della richiesta ad esso correlata, la funzione RetrievePartialData doveva continuare la sua esecuzione, ma non era più in grado di accedere ai repository, in quanto lo scope assegnato in precedenza risultava chiuso.

Per chiamare il SSE è stato definito Client, che corrisponde ad un oggetto di tipo HttpClient che viene generato con il seguente codice:

```
private static readonly HttpClient Client = new(new
    SocketsHttpHandler
{
    PooledConnectionLifetime = TimeSpan.FromSeconds(10)
})
{
    BaseAddress = new Uri("http://compute-engine:8080/"),
    Timeout = Timeout.InfiniteTimeSpan
};
```

Come si può vedere dal codice è possibile definire un pool di connessioni, in modo tale da poterle chiudere dopo un periodo specificato di inutilizzo (in questo esempio, ogni dieci secondi). In questo modo si può ridurre la saturazione di connessioni aperte e riutilizzare quelle già presenti nel pool [17]. Verrà fatto uso di questo oggetto anche per altri metodi.

ComputingTaskExists()

```
public bool ComputingTaskExists(int id)
{
    return _computingProviderRepository.ComputingTaskExists(id);
}
```

ComputingTaskExists verifica l'esistenza di un task, dato il suo identificativo.

GetEmailCustomerById()

```
public string GetEmailCustomerById(int id)
```

```
{
    return _computingProviderRepository.GetEmailCustomerById(id);
}
```

Permette di restituire l'e-mail del cliente che ha sottomesso il task con l'id specificato.

GetComputingTaskInputData()

```
public ComputingTaskInputDataDto GetComputingTaskInputData(int id)
{
    var computingTask = _computingProviderRepository.
    GetComputingTaskInputData(id);
    string jsonFile = Utilities.ZipService.Unzip(computingTask.
    FileContent);
    ComputingTaskInputDataDto computingTaskInputData = new
    ComputingTaskInputDataDto(computingTask.Id, jsonFile,
    computingTask.TsCreation, computingTask.EmailCustomer);
    return computingTaskInputData;
}
```

GetComputingTaskinputData permette di restituire i valori di input di un determinato task, mappandoli in un oggetto di tipo ComputingTaskInputDataDto. Permette di restituire, anche, il contenuto del file di input "unzippato".

GetComputingTaskCustomerSubmission() e GetComputingTaskAdminSubmission()

```
public List<ComputingTaskSubmissionDto>
    GetComputingTaskCustomerSubmission(string userEmail)
{
    var computingTasksSubmissions = _computingProviderRepository.
    GetComputingTaskCustomerSubmission(userEmail);
    var computingTasksSubmissionsMap = new List<
    ComputingTaskSubmissionDto>();
    foreach(var cts in computingTasksSubmissions)
    {
        computingTasksSubmissionsMap.Add(_mapper.Map<
    ComputingTaskSubmissionDto>(cts));
    }
    return computingTasksSubmissionsMap;
}

public List<ComputingTaskSubmissionDto>
    GetComputingTaskAdminSubmission()
{
```

```
14      var computingTasksSubmissions = _computingProviderRepository.
    GetComputingTaskAdminSubmission();
15      var computingTasksSubmissionsMap = new List<
    ComputingTaskSubmissionDto>();
16      foreach (var cts in computingTasksSubmissions)
17      {
18          computingTasksSubmissionsMap.Add(_mapper.Map<
    ComputingTaskSubmissionDto>(cts));
19      }
20      return computingTasksSubmissionsMap;
21  }
```

Questi due metodi permettono di restituire una lista con i task sottomessi dagli utenti, utilizzando la classe ComputingTaskSubmissionDto. La differenza tra i due è che il primo permette di ottenere solo i task specifici dell'utente che ne ha fatto richiesta.

GetSubmissionAndMetadata()

```
1   public List<SubmissionAndMetadataDto> GetSubmissionAndMetadata(List<
    ComputingTaskSubmissionDto> submissions, List<
    ComputingTasksMetadata> metadata)
2   {
3       List<SubmissionAndMetadataDto> submissionAndMetadata = new();
4       foreach (var cts in submissions)
5       {
6           foreach (var m in metadata)
7           {
8               if (m.Id == cts.Id)
9               {
10                  SubmissionAndMetadataDto result = new
    SubmissionAndMetadataDto(cts.Id, m.FileName, m.Status, m.
    TsUploaded, cts.TsCreation, m.TsStarted, m.TsCompleted, cts.
    EmailCustomer);
11                  submissionAndMetadata.Add(result);
12                  break;
13              }
14          }
15      }
16      return submissionAndMetadata;
17  }
```

GetSubmissionAndMetadata permette di combinare i dati inerenti alle sottomissioni di un task e i metadati, restituendo una lista di oggetti di tipo SubmissionAndMetadataDto.

StopPartialTask()

```
public async Task<bool> StopPartialTask(int id)
{
    var httpResponseMessage = await Client.DeleteAsync("/tasks/" + id
    .ToString());
    var response = httpResponseMessage.StatusCode;
    if (response == System.Net.HttpStatusCode.OK)
    {
        Console.WriteLine($"STOP PROCESSING THE TASK");
        return true;
    }
    Console.WriteLine($"ERROR DURING THE STOP OF THE PARTIAL
    COMPUTING TASK");
    return false;
}
```

StopPartialTask effettua una richiesta HTTP ad una API presente sul Compute Engine, avvisandolo che si vuole interrompere l'esecuzione di un calcolo. Se è andato a buon fine restituisce true, altrimenti false.

Ora verranno descritti i metodi presenti nel file ComputingTaskMetadataService:

AddFileMetadata()

```
public bool AddFileMetadata(int taskId, string file_name, DateTime
    TsUploadedDT)
{
    ComputingTasksMetadata metadata = new()
    {
        Id = taskId,
        FileName = file_name,
        Status = "NOT_ASSIGNED",
        TsUploaded = TsUploadedDT,
        TsStarted = default,
        TsCompleted = default
    };
    return _computingTasksMetadataRepository.AddFileMetadata(metadata
    );
}
```

AddFileMetadata crea un oggetto nuovo di tipo ComputingTaskMetadata aggiungendo il taskId, il nome del file e il tempo di upload, inseriti come parametri in input. Ritorna true se è andato a buon fine, l'inserimento nel database.

GetAllMetadata(), GetMetadata(), MetadataExists()

```
public List<ComputingTasksMetadata> GetAllMetadata()
{
    return _computingTasksMetadataRepository.GetAllMetadata();
}

public ComputingTasksMetadata GetMetadata(int id)
{
    return _computingTasksMetadataRepository.GetMetadata(id);
}

public bool MetadataExists(int id)
{
    return _computingTasksMetadataRepository.MetadataExists(id);
}
```

Questi tre metodi permettono, rispettivamente, di recuperare tutti i metadati di tutti i task, i metadati di un task specifico e verificare se i metadati correlati ad un task sono già presenti nel database.

UpdateMetadata()

```
public bool UpdateMetadata(ComputingTasksMetadata newMetadata,
    ComputingTasksMetadata oldMetadata)
{
    try
    {
        newMetadata.FileName = oldMetadata.FileName;
        newMetadata.TsUploaded = oldMetadata.TsUploaded;
        newMetadata.TsStarted = newMetadata.Status == "STARTED" ?
    newMetadata.TsStarted : oldMetadata.TsStarted;
        var success = _computingTasksMetadataRepository.
    UpdateMetadata(newMetadata);
        return success.Result > 0 ? true : false;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        return false;
    }
}
```

Dati in ingresso i metadati già presenti nel database e quelli aggiornati, vengono inseriti alcuni campi dei primi nei secondi e, successivamente, inviati nuovamente al db per l'aggiornamento.

I prossimi metodi descritti riguardano i risultati del calcolo e sono inseriti all'interno del file ResultsService:

ComputingTaskResultsExists(), GetComputingTaskResultsById()

```
public bool ComputingTaskResultsExists(int id)
{
    return _resultsRepository.ComputingTaskResultsExists(id);
}

public IEnumerable<RisultatiOra> GetComputingTaskResultsById(int id)
{
    var computingTaskResults = _resultsRepository.
    GetComputingTaskResultsById(id);
    return computingTaskResults;
}
```

I primi due metodi descritti di questo file permettono, rispettivamente, di valutare se sono presenti dei risultati per un determinato task e di recuperarli.

CreateCSVFile()

```
public MemoryStream CreateCSVFile(IEnumerable<RisultatiOra> results)
{
    var csvPath = Path.Combine(Environment.CurrentDirectory, $"
    Results" + DateTime.UtcNow.Ticks.ToString() +".csv");
    using (var streamWriter = new StreamWriter(csvPath))
    {
        using (var csvWriter = new CsvWriter(streamWriter,
    CultureInfo.InvariantCulture))
        {
            csvWriter.Context.RegisterClassMap<RisultatiOraClassMap
    >();
            csvWriter.WriteRecords(results);
        }
    }
    byte[] fileBytes = System.IO.File.ReadAllBytes(csvPath);
    File.Delete(csvPath);
    var content = new MemoryStream(fileBytes);
    return content;
}
```

Dati in ingresso una collezione di risultati, permette di creare il contenuto del file con estensione .csv, che sarà scaricato dall'utente.

GetComputingTaskStats()

```
public string GetComputingTaskStats(RisultatiOraStatsDto results)
{
    var percentage = Utilities.Utilities.Percentage(results.Step, "
    Step_totali");
    var eta = Utilities.Utilities.Time_To_Completion(results.
    Time_Execution, results.Step, "Step_totali");
    var stats = JsonConvert.SerializeObject(new
    {
        percentage = percentage * 100,
        eta = (int)eta
    });
    return stats;
}
```

Ricevuti in ingresso i dati Step e TimeExecution corrispondenti all'ultimo risultato disponibile di un calcolo, permette di calcolare la percentuale di completamento e il tempo stimato per terminare il calcolo. Nel primo caso il calcolo corrisponde, banalmente, ad una divisione tra gli step già calcolati e il numero di step totali. La seconda statistica si basa sulla formula per calcolare il time to completion (ETA), ovvero: $ETA = \frac{TempoMedioDiIterazione}{NumeroDiIterazioniRimanenti}$

GetLastResultById()

```
public RisultatiOraStatsDto GetLastResultById(int id)
{
    var computingTaskResults = _resultsRepository.GetLastResultById(
    id);
    return _mapper.Map<RisultatiOraStatsDto>(computingTaskResults);
}
```

GetLastResultById è strettamente correlato al metodo precedente e, come si può intuire dal titolo, permette di recuperare gli ultimi dati, più nello specifico Step e TimeExecution, relativi all'ultimo risultato.

Terminati anche i metodi relativi ai risultati, gli ultimi da descrivere rimangono quelli relativi agli utenti, contenuti nel file CustomerService:

AddCustomer(), CustomerExist(), GetCustomers(), UpdateCustomer()

```
public bool AddCustomer(Customer customer)
{
    var success = _customerRepository.AddCustomer(customer);
    return success;
}

```

```
7  public Customer CustomerExist(string email)
8  {
9      var customer = _customerRepository.CustomerExist(email);
10     return customer;
11 }
12
13 public ICollection<Customer> GetCustomers()
14 {
15     var customers = _customerRepository.GetCustomers();
16     return customers;
17 }
18
19 public bool UpdateCustomer(Customer customer)
20 {
21     var success = _customerRepository.UpdateCustomer(customer);
22     return success;
23 }
```

Tutti e quattro i metodi chiamano direttamente la funzione presente nella sezione relativa al Data Access. Andando maggiormente nello specifico, permettono di aggiungere un nuovo utente, verificare se un utente esiste data la mail, ottenere tutti i clienti che hanno sottomesso almeno un task e aggiornare i dati di un utente specificato.

Tutti i metodi appena rappresentati non comunicano direttamente con il database, ma si interfacciano con un altro livello presente nell'applicazione, chiamato Data Access layer. Terminata la trattazione del livello di servizio, nel sottoparagrafo successivo si farà riferimento proprio alla comunicazione con la base di dati.

3.2.5 Livello di Data Access

Il Data Access layer comunica da una parte con il livello di servizio esposto in precedenza e dall'altra direttamente con le tabelle presenti nel database.

Tale processo è reso possibile grazie alle classi che derivano dalla classe DbContext. Essa permette di creare una sessione con il database, abilitando l'esecuzione di query e la manipolazione dei dati presenti nelle tabelle [18]. Nei metodi seguenti il campo _context rappresenta un'istanza delle varie classi appena definite.

Come nel livello di servizio sono state definite diverse classi per implementare il Data Access layer, in base a con quale tabella presente all'interno del database comunicano. Esse verranno definite più avanti con il termine repository. La descrizione di ogni singola tabella è rimandata al paragrafo successivo, mentre in questo momento verranno solo nominate.

Il primo di cui si vuole discutere prende il nome di ComputingProviderRepository e utilizza il seguente Context:

```
public class ComputingProviderContext : DbContext
{
    public ComputingProviderContext(DbContextOptions<
    ComputingProviderContext> options) : base(options)
    {
    }

    public DbSet<ComputingTask> ComputingTasks { get; set; }
}
```

Esso comunica con la tabella ComputingTasks e i metodi che implementa sono:

AddComputingTask()

```
public int AddComputingTask(ComputingTask computingTask)
{
    _context.Add(computingTask);
    var saved = _context.SaveChanges();
    if (saved > 0)
        return computingTask.Id;
    return 0;
}
```

AddComputingTask permette di aggiungere un nuovo task nel database e, in caso di esito positivo, ne ritorna l'id.

ComputingTaskExists()

```
public bool ComputingTaskExists(int id)
{
    return _context.ComputingTasks.Any(c => c.Id == id);
}
```

ComputingTaskExists controlla se esiste almeno un task con l'id specificato ritornando true in caso di esito positivo.

GetComputingTaskAdminSubmission(), GetComputingTaskCustomerSubmission()

```
public IEnumerable<ComputingTask> GetComputingTaskAdminSubmission()
{
    return _context.ComputingTasks.ToList();
}

```

```
6 public IEnumerable<ComputingTask> GetComputingTaskCustomerSubmission(
      string userEmail)
7 {
8     return _context.ComputingTasks.Where(c => c.EmailCustomer ==
      userEmail).ToList();
9 }
```

Prelevano dalla tabella la lista di tutti i task sottomessi da tutti i clienti o da un singolo cliente, con la mail specificata come parametro in ingresso.

GetComputingTaskInputData()

```
1 public ComputingTask GetComputingTaskInputData(int id)
2 {
3     return _context.ComputingTasks.Where(c => c.Id == id).
      FirstOrDefault();
4 }
```

ComputingTaskInputData permette di recuperare i valori di input di un task dato il suo id.

GetEmailCustomerById()

```
1 public string GetEmailCustomerById(int id)
2 {
3     return _context.ComputingTasks.Where(c => c.Id == id).
      FirstOrDefault().EmailCustomer;
4 }
```

GetEmailCustomerById permette di recuperare, dato l'id del task, la mail del cliente che l'ha sottomesso.

Il secondo repository comunica con la tabella ComputingTasksMetadata e utilizza il seguente Context:

```
1 public class ComputingTasksMetadataContext : DbContext
2 {
3     public ComputingTasksMetadataContext(DbContextOptions<
      ComputingTasksMetadataContext> options) : base(options)
4     {
5     }
6
7     public DbSet<ComputingTasksMetadata> ComputingTasksMetadata { get
      ; set; }
8 }
```

I metodi usati sono:

AddFileMetadata(), MetadataExists(), GetMetadata(), GetAllMetadata()

```
public bool AddFileMetadata(ComputingTasksMetadata metadata)
{
    _context.Add(metadata);
    var saved = _context.SaveChanges();
    return saved > 0 ? true : false;
}

public bool MetadataExists(int id)
{
    return _context.ComputingTasksMetadata.Any(c => c.Id == id);
}

public ComputingTasksMetadata GetMetadata(int id)
{
    return _context.ComputingTasksMetadata.Where(c => c.Id == id).
    AsNoTracking().FirstOrDefault();
}

public List<ComputingTasksMetadata> GetAllMetadata()
{
    return _context.ComputingTasksMetadata.ToList();
}
```

Questi primi quattro metodi sono simili ad altri presenti nel repository precedente, con la differenza che comunicano con una tabella diversa. Essi permettono, nell'ordine sopra indicato, di salvare una nuova tupla di metadati, controllare se ne esiste una in particolare e ottenere un insieme specifico di metadati, specificando l'id del task, o ottenere tutti quelli presenti.

UpdateMetadata()

```
public async Task<int> UpdateMetadata(ComputingTasksMetadata metadata
    )
{
    _context.Update(metadata);
    return await _context.SaveChangesAsync();
}
```

UpdateMetadata permette di aggiornare i valori di un insieme di metadati già presenti sul db con quelli inviati come argomento in ingresso.

Terminata la descrizione sui metadati, si passa a quella del repository relativo ai risultati, con il seguente Context:

```
public class ResultsContext : DbContext
{
    public ResultsContext(DbContextOptions<ResultsContext> options) :
    base(options)
    {
    }

    public DbSet<RisultatiOra> ComputingTasksResults { get; set; }
}
```

Esso prende il nome di ResultsRepository e implementa i seguenti metodi:

AddPartialResult()

```
public bool AddPartialResult(List<RisultatiOra> results)
{
    foreach (var r in results.ToList())
    {
        _context.Add(r);
    }
    var saved = _context.SaveChanges();
    return saved > 0 ? true : false;
}
```

Il metodo AddPartialResult permette di salvare una lista di risultati parziali, calcolati fino al momento della sua chiamata, all'interno del db. Restituisce true se il salvataggio di tutta la lista è andato a buon fine.

ComputingTaskResultsExists(), GetComputingTaskResultsById()

```
public bool ComputingTaskResultsExists(int id)
{
    return _context.ComputingTasksResults.Any(c => c.Id_File == id);
}

public IEnumerable<RisultatiOra> GetComputingTaskResultsById(int id)
{
    return _context.ComputingTasksResults.Where(c => c.Id_File == id)
    ;
}
```

Dato un task specifico, permettono di verificare l'esistenza dei risultati e, nel secondo metodo, di recuperarli.

GetLastResultById()

```
public RisultatiOra GetLastResultById(int id)
{
    return _context.ComputingTasksResults.Where(c => c.Id_File == id)
    .OrderBy(c => c.Step).Last();
}
```

L'ultimo metodo di questo repository è GetLastResultById, che permette di recuperare solo l'ultima riga della tabella dei risultati. Esso è utilizzato per scopi statistici come già descritto nel sottoparagrafo precedente.

L'ultimo repository rimasto da mostrare è quello inerente agli utenti, il quale utilizza il seguente Context:

```
public class CustomerContext : DbContext
{
    public CustomerContext(DbContextOptions<CustomerContext> options)
    : base(options)
    {
    }

    public DbSet<Customer> Customers { get; set; }
}
```

Per terminare questo argomento, vengono mostrati anche i suoi metodi:

AddCustomer(), CustomerExist(), GetCustomers(), UpdateCustomer()

```
public bool AddCustomer(Customer customer)
{
    _context.Add(customer);
    var saved = _context.SaveChanges();
    return saved > 0 ? true : false;
}

public Customer CustomerExist(string email)
{
    return _context.Customers.Where(c => c.Email == email).
    FirstOrDefault();
}

public ICollection<Customer> GetCustomers()
{
    return _context.Customers.ToList();
}
```

```
17
18 public bool UpdateCustomer(Customer customer)
19 {
20     _context.Update(customer);
21     var saved = _context.SaveChanges();
22     return saved > 0 ? true : false;
23 }
```

Questi quattro metodi permettono di salvare un nuovo cliente, controllare se esiste l'utente specificato da una mail in ingresso, ottenere tutti gli utenti che hanno sottomesso un task e aggiornare i valori già presenti con le nuove informazioni inviate come argomento.

Con la fine della descrizione del livello di Data Access, termina anche il paragrafo relativo al Computing Provider. In quello successivo si andrà maggiormente nello specifico nella sezione dedicata alla persistenza dei dati, ovvero il database.

3.3 Database

3.3.1 PostgreSQL

Terminato il discorso riguardante il Computing Provider e prima di descrivere la prossima applicazione che compone il sistema, la domanda sorge spontanea. Ma dove verranno salvate tutte le informazioni riguardanti i task e gli utenti? Ovviamente la risposta a tale quesito corrisponde al database, dove ogni tipologia di dato verrà diviso in diverse sezioni. Per questo progetto si è deciso di utilizzare PostgreSQL, ovvero un database di tipo relazione molto utilizzato e supportato a livello mondiale. Come possiamo osservare in modo più approfondito sul sito di IBM [19], PostgreSQL risulta essere famoso per la sua affidabilità, flessibilità e sul fatto che supporti non solo dati di tipo relazionale, come da sua tipologia, ma anche quelli di tipo non relazionale rendendolo uno dei database più interessanti nel panorama generale. Esso, inoltre, offre prestazioni molto elevate presentando sempre nuove ottimizzazioni e implementa, tra le tante, il supporto geospaziale, aumentandone la scalabilità. Un'altra feature di rilevata importanza consiste nel fatto che le operazioni di scrittura e lettura possono essere svolte in parallelo, senza necessariamente che ci sia una dipendenza temporale tra loro. La sicurezza che esse vengano svolte in modo appropriato la si ottiene grazie all'utilizzo del Multiversion Concurrency Control, che permette di ottenere un comportamento di tipo transazionale nelle scritture nella base di dati. Ovviamente, tra i motivi per cui PostgreSQL dovrebbe essere la base di dati utilizzata per un sistema c'è il fatto che sia open source, sia supportato per vari tipi di linguaggi di programmazione e che sia uno dei database più longevi, tutte garanzie di affidabilità e supporto continuo. Per tornare sul progetto trattato in questa tesi, verranno descritte nel

prossimo sottoparagrafo le tabelle definite per soddisfare le varie richieste.

3.3.2 Tabelle progetto

Una volta chiarite le motivazioni sulla scelta di PostgreSQL come database per salvare i dati, qui di seguito verranno descritte le tabelle e i campi che le compongono.

Customers

```
CREATE TABLE "Customers" (
    "Email" TEXT NOT NULL PRIMARY KEY,
    "Name" TEXT NOT NULL,
    "Surname" TEXT NOT NULL,
    "NumTasks" INT NOT NULL
);
```

Si parte con la descrizione della tabella dedicata al salvataggio delle informazioni principali riguardanti i clienti. Nello specifico si vanno a salvare l'indirizzo e-mail, che corrisponde anche alla chiave primaria, il nome e il cognome. L'ultimo campo "NumTasks" indica il numero di lavori sottomessi da quello specifico utente.

ComputingTasks

```
CREATE TABLE "ComputingTasks" (
    "Id" INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    "FileContent" BYTEA NOT NULL,
    "TsCreation" TIMESTAMP WITH TIME ZONE NOT NULL,
    "EmailCustomer" TEXT NOT NULL,
    FOREIGN KEY ("EmailCustomer") REFERENCES "Customers"("Email")
);
```

Questa tabella salva il progetto che l'utente ha sottomesso da mandare in input al motore di calcolo. Il campo "Id" corrisponde ad un valore univoco per ogni progetto e verrà ripreso anche nelle future tabelle. "FileContent" contiene un blob di byte corrispondente al file di cui l'utente ha fatto l'upload e contenente i dati di input. Esso in precedenza viene compresso in modo tale da ridurre lo spazio occupato. "EmailCustomer" corrisponde all'email presente nella tabella "Customers" descritta precedentemente, in modo tale che ogni progetto sia collegato ad un utente specifico. Infine, "TsCreation" indica la data e ora di creazione di un task nuovo. Questo campo è stato separato rispetto agli altri valori utilizzati di tipo timestamp per motivi di comodità, in quanto questa tabella sarà quella che Kafka userà per prelevare i valori di input per il motore di calcolo. "TsCreation" corrisponde proprio al valore su cui si fa il controllo.

ComputingTasksMetadata

```
CREATE TABLE "ComputingTasksMetadata" (
    "Id" INT PRIMARY KEY REFERENCES "ComputingTasks" ("Id"),
    "FileName" TEXT,
    "Status" TEXT NOT NULL,
    "TsUploaded" TIMESTAMP WITH TIME ZONE,
    "TsStarted" TIMESTAMP WITH TIME ZONE,
    "TsCompleted" TIMESTAMP WITH TIME ZONE
);
```

È stato deciso di separare i valori effettivi da mandare in input al motore, da quelli che possono essere chiamati metadati dei valori stessi, che non serviranno a Kafka o per il calcolo effettivo. Infatti, in questa tabella, oltre al valore "Id" che, come si può notare nel codice soprastante, corrisponde al valore "Id" definito nella tabella "ComputingTasks", si possono trovare varie caratteristiche sempre conformi ai valori di input. Nello specifico troviamo "FileName", ovvero il nome del file di cui si è fatto l'upload contenente gli input di progetto. "Status" indica a che punto si trova il calcolo e può avere quattro valori:

- "NOT ASSIGNED": il file è stato caricato, ma ancora non assegnato a nessuna istanza del motore;
- "STARTED": il calcolo su quel determinato task è stato letto e preso in esecuzione dal Compute Engine;
- "INTERRUPTED". il calcolo è stato interrotto dall'utente per qualsiasi motivo;
- "COMPLETED": il calcolo è stato completato al 100%.

Infine, ci sono i restanti tre valori riguardanti le tempistiche:

- "TsUploaded": indica quando è stato eseguito l'upload del file contenente i valori di input;
- "TsStarted": indica quando è partito il calcolo sui dati indicati;
- "TsCompleted": indica quando il calcolo è stato completato.

ComputingTasksResults

```
CREATE TABLE "ComputingTasksResults" (
    "Id_File" INT REFERENCES "ComputingTasks" ("Id"),
    "YY" INT NOT NULL,
```

```
    "MM" INT NOT NULL,
    "GG" INT NOT NULL,
    "HH" INT NOT NULL,
    "Res1" INT NOT NULL,
    "Res2" INT NOT NULL,
    "Res3" INT NOT NULL,
    "Time_Execution" INT NOT NULL,
    "Step" INT NOT NULL,
    PRIMARY KEY ("Id_File","YY","MM","GG","HH")
);
```

Questa è l'ultima tabella presente all'interno del database e permette di salvare i risultati di un task inviato dall'utente. Ogni riga corrisponde al risultato di un'ora di calcolo, infatti la chiave primaria di questa tabella è di tipo composto ed è formata da "Id_File", che indica sempre l'id dei valori di input, anno ("YY"), mese ("MM"), giorno ("GG"), ora ("HH"). Come si può notare oltre ai valori effettivi dei risultati, sono presenti due campi utili per il calcolo di statistiche e stime. Essi sono "Step" e "Time_Execution", il quale indica il tempo in secondi dall'inizio dell'esecuzione per completare il determinato step al quale si riferisce.

3.3.3 Relazioni tra tabelle

Come si può notare dalle righe di codice e dalle spiegazioni precedenti esistono delle relazioni tra le varie tabelle. Quindi, risulta fondamentale capire come queste entità siano collegate tra loro. Le relazioni individuate sono le seguenti:

- Tra "Customers" e "ComputingTasks" è presente una relazione di tipo 1 a n (molti)
- Tra "ComputingTasks" e "ComputingTasksResults" è presente una relazione di tipo 1 a n (molti)
- Tra "ComputingTasks" e "ComputingTasksMetadata" è presente una relazione di tipo 1 a 1. In questo caso specifico è stato necessario individuare una relazione di tipo Padre-Figlio tra le due tabelle, definita da "ComputingTasks" e "ComputingTasksMetadata" rispettivamente.

Terminato il breve, ma doveroso, discorso sulla memorizzazione dei dati, nel prossimo paragrafo si andrà a descrivere dettagliatamente la seconda applicazione di questo sistema, il Compute Engine.

3.4 Compute Engine

3.4.1 Panoramica generale

Il Compute Engine è la seconda applicazione di cui si vuole discutere in questo progetto di tesi. Essa è direttamente collegata con il Computing Provider e comunica con la piattaforma Kafka per prelevare le informazioni di input dei task.

Questa applicazione corrisponde al "cuore" di questo sistema poiché contiene il motore di calcolo e i vari blocchi di codice che aiutano a raggiungere lo scopo finale del progetto.

Il lavoro del Compute Engine inizia mettendosi in ascolto di una nuova computazione di calcolo richiesta da un utente. Più nello specifico, si iscrive ad un topic presente su Kafka, in modo tale da consumarne il messaggio una volta disponibile.

Una volta ottenuto, viene gestito per essere compatibile con il motore di calcolo e inviato in ingresso a quest'ultimo. Una volta terminato uno step, i risultati ottenuti verranno inviati al Computing Provider, per essere resi disponibili all'utente. Ogni step equivale ad un'ora di calcolo.

Il Compute Engine può interrompere anticipatamente i calcoli sul progetto, sempre su richiesta dell'utente, per varie motivazioni come una insoddisfazione dei risultati parziali già ottenuti o di una prima impressione positiva su di essi.

Nei prossimi sottoparagrafi si andranno a specificare più internamente tutti i vari blocchi che compongono il Compute Engine, partendo dai Model utilizzati e la definizione delle classi globali.

3.4.2 Model e classi globali

Per gestire le varie operazioni sui dati, il Compute Engine utilizza tre Model già utilizzati nel Computing Provider. Elencandoli brevemente, essi risultano essere: ComputingTask, ComputingTasksMetadata e RisultatiOra. Per avere una maggiore descrizione e spiegazione su di essi si rimanda al paragrafo relativo al Computing Provider (3.2).

È presente anche una nuova classe che permette di definire un Model, esclusiva del Compute Engine. Essa prende il nome di Notification e presenta i seguenti campi:

```
1 public class Notification
2 {
3     public string client_Id { get; set; }
4     public int task_Id { get; set; }
5     public string Message { get; set; }
6     public CancellationTokenSource TokenSource { get; set; }
7 }
```

Essa viene utilizzata per scambiare i risultati parziali di un calcolo con il Computing Provider. Come si può vedere contiene diversi campi:

- Client_Id: indica un id univoco del client che ha fatto richiesta del task;
- Task_Id: corrisponde all'id univoco del task di cui si sta effettuando il calcolo;
- Message: contiene le informazioni da passare al Computing Provider;
- TokenSource: token utilizzato per capire se è stata effettuata una richiesta di cancellazione per un progetto.

Questa classe risulta essere di fondamentale importanza per scambiare informazioni con il Computing Provider, come si vedrà anche in seguito.

Un'ulteriore aggiunta corrisponde alla definizione di una classe di tipo statico che prende il nome di Globals:

```
1 public static class Globals
2 {
3     public static ConcurrentDictionary<int, Channel<string>>
    concurrentDictionary = new ConcurrentDictionary<int, Channel<
    string>>();
4     public static ConcurrentDictionary<int, Notification>
    notifications = new ConcurrentDictionary<int, Notification>();
5 }
```

Essa permette di definire due dizionari concorrenti. Entrambi usano come chiave l'id del task, mentre differiscono con il dato salvato come valore. Il primo definisce un oggetto di tipo Channel che permette di raggruppare tutti i risultati parziali ottenuti e di recuperarli successivamente, seguendo una logica di tipo FIFO. Il secondo permette di salvare un oggetto di tipo Notification, definito dalla classe descritta in precedenza.

Terminata la breve sezione sui Model implementati, nel prossimo sottoparagrafo si andrà a discutere la parte relativa al Controller.

3.4.3 Controller-Endpoint

Le API offerte dal Compute Engine vengono usate per aprire delle comunicazioni dirette con il Computing Provider. La classe del controller prende il nome di ComputeEngineController e definisce i seguenti metodi:

Subscribe()

```
[HttpGet("/subscriptions/{client_id}/tasks/{task_id}/partialResult")]
public async Task<IActionResult> Subscribe([FromRoute] string
    client_id, [FromRoute] int task_id)
{
    var response = Response;
    response.Headers.Add("Cache-Control", "no-cache");
    response.Headers.Add("Connection", "keep-alive");
    response.Headers.Add("Content-Type", "text/event-stream");
    try
    {
        await _partialDataService.getPartialData(response, client_id,
    task_id);
    }
    catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
    finally
    {
        _messageQueue.Deregister(task_id);
    }
    return new EmptyResult();
}
```

Il campo client_id identifica in modo univoco il cliente che ha chiamato questa API, mentre task_id corrisponde all'identificativo del task. Questo endpoint presenta una particolarità perché è di tipo Server Sent Event (SSE). SSE è un protocollo utilizzato per inviare informazioni sotto forma di Stream [20]. Permette di aprire una connessione sempre aperta di tipo unidirezionale tra un server e un client. Questa connessione verrà terminata quando saranno terminati i dati da inviare. Il vantaggio principale consiste nello scambiare informazioni appena esse risultano essere disponibili, con un aumento nelle performance, in quanto non si deve aprire una connessione nuova per ogni dato generato.

Nell'header della risposta vengono definiti vari parametri:

- Cache-Control: no-cache. Indica che è possibile salvare la risposta in memoria cache, ma deve essere sempre validata prima di ogni riutilizzo dal server originale [21];
- Connection: keep-alive. Serve ad indicare che la connessione rimarrà sempre aperta;
- Content-Type: text/event-stream. Indica che il contenuto deve essere scambiato in formato Stream.

All'interno del blocco try è presente il metodo getPartialData che effettua tutta la business logic dell'endpoint SSE, mentre nel blocco finally è presente il metodo Deregister che permette di eseguire le parti di terminazione, con successiva chiusura della connessione.

CancelToken()

```
[HttpDelete("/tasks/{task_id}")]
public IActionResult CancelToken([FromRoute] int task_id)
{

    if(_partialDataService.CancelToken(task_id))
        return Ok();
    return BadRequest();
}
```

Metodo che permette di gestire la cancellazione di un calcolo, dato l'id del task in ingresso. In caso di esito positivo, restituisce il codice di stato 200, altrimenti BadRequest (400).

Le variabili _partialDataService e _messageQueue corrispondono alle interfacce che permettono di chiamare i servizi definiti rispettivamente nelle classi PartialDataService e CustomeMessageQueue, che si andranno a descrivere nel prossimo sottoparagrafo.

3.4.4 Livello di servizio

In questa sezione si descrivono i due file contenenti i metodi che implementano la business logic dell'applicazione, con eccezione del file ComputeEngineService di cui si andrà a discutere nel prossimo sottoparagrafo.

Si può iniziare parlando del primo dei due file chiamato CustomMessageQueue, che presenta le seguenti funzioni:

Register()

```
public void Register(int task_id)
{
    bool success = Globals.concurrentDictionary.TryAdd(task_id,
    Channel.CreateUnbounded<string>());

    if (!success)
    {
        throw new ArgumentException($"The task Id {task_id} is
    already registered");
```

```
8       }
9   }
```

Esso viene chiamato per definire un nuovo Channel all'interno del dizionario chiamato concurrentDictionary. Restituisce un errore nel caso sia già presente un canale con l'id del task specificato.

Deregister()

```
1   public void Deregister(int task_id)
2   {
3       Globals.concurrentDictionary.TryRemove(task_id, out _);
4   }
```

Questo metodo svolge il compito inverso rispetto al precedente, infatti, permette di rimuovere l'oggetto all'interno del dizionario, avente l'id del task specificato come chiave.

EnqueueAsync()

```
1   public async Task EnqueueAsync (Notification notification)
2   {
3       bool success = Globals.concurrentDictionary.TryGetValue(
        notification.task_Id, out Channel<string> channel);
4
5       if (!success)
6       {
7           throw new ArgumentException($"Error encountered when adding a
        new message to the queue.");
8       }
9       else
10      {
11          await channel.Writer.WriteAsync(notification.Message);
12      }
13  }
```

Permette di ottenere il canale definito dall'id di un task e, in caso positivo, di scriverci sopra un messaggio definito all'interno di un oggetto di tipo Notification.

DequeueAsync()

```
1   public IAsyncEnumerable<string> DequeueAsync(int task_id)
2   {
```

```
3       bool success = Globals.concurrentDictionary.TryGetValue(task_id,
        out Channel<string> channel);
4
5       if (success)
6       {
7           return channel.Reader.ReadAllAsync();
8       }
9       else
10      {
11          throw new ArgumentException($"The task Id {task_id} isn't
        registered");
12      }
13  }
```

Permette di ottenere il canale, con chiave uguale all'id del task specificato come parametro e di prelevare tutti i messaggi che sono presenti all'interno dell'oggetto Channel.

Terminata la discussione sul file CustomMessageQueue, si può passare al secondo insieme di metodi definiti in questa sezione. Essi sono definiti all'interno del file PartialDataService:

GetPartialData()

```
1  public async Task getPartialData(HttpResponse response, string
       client_id, int task_id)
2  {
3       _messageQueue.Register(task_id);
4       StreamWriter streamWriter = new StreamWriter(response.Body);
5       var count = 0;
6       await foreach (var message in _messageQueue.DequeueAsync(task_id)
       )
7       {
8           count++;
9           await streamWriter.WriteLineAsync($"{message}");
10          await streamWriter.FlushAsync();
11      }
12 }
```

Questo metodo permette di registrare un nuovo Channel, dato l'id di un task passato come parametro. Successivamente indica che i risultati parziali verranno inviati nel Body della richiesta del SSE, utilizzando uno StreamWriter. Esso verrà popolato andando a prelevare le informazioni direttamente dal Channel definito all'interno del dizionario.

CancelToken()

```
public bool CancelToken(int task_id)
{
    var success = Globals.notifications.TryGetValue(task_id, out
    Notification oldNotification);
    if (success)
    {
        Notification newNotification = oldNotification;
        newNotification.TokenSource.Cancel();
        var updated = Globals.notifications.TryUpdate(task_id,
    newNotification, oldNotification);
        if (updated)
            return true;
    }
    Console.WriteLine($"Error encountered during the update of the
    interrupted flag.");
    return false;
}
```

CancelToken() viene chiamato quando è stata richiesta una cancellazione del calcolo in corso. Per svolgere tale richiesta, si va a settare il token di un oggetto di tipo Notification situato all'interno del dizionario, indicando che è pronto per essere cancellato. In caso di esito positivo, restituisce true.

Terminata la discussione sui metodi che compongono parte della business logic, si può passare alla discussione dell'ultimo file rimasto e sul perché esso corrisponde a quello principale dell'applicazione.

3.4.5 Background service

Il servizio che si vuole descrivere in questa sezione corrisponde a quello che permette di chiamare il calcolo scientifico. Per questo motivo risulta essere di rilevata importanza nel complesso dell'applicazione.

La classe che lo descrive deriva da BackgroundService, che implementa l'interfaccia IHostedService. La classe derivata permette di definire un servizio di lunga durata che verrà eseguito in background dal sistema [22]. Questo servizio ha la particolarità che, a differenza di quelli descritti in precedenza, esegue la logica del Compute Engine non correlata ad una richiesta proveniente dal Controller.

Il servizio inizia contattando il message broker Kafka e iscrivendosi al topic relativo alle informazioni di input.

```
using (_consumer)
{
    _consumer.Subscribe(_topicName);

```

```
5       while (!stoppingToken.IsCancellationRequested)
6       {
7           try
8           {
9               var result = _consumer.Consume(stoppingToken);
10              var data = JsonConvert.DeserializeObject<ComputingTask>(
    result.Message.Value);
11              string input = ComputeEngine.Utils.ZipService.Unzip(data.
    FileContent);
12              idFile = data.Id;
13              [...]
14              var httpClient = new HttpClient()
15              {
16                  BaseAddress = new Uri("http://computingprovider:80/")
    ,
17              };
18              ComputingTasksMetadata metadata = new()
19              {
20                  Id = idFile,
21                  FileName = "",
22                  Status = "STARTED",
23                  TsUploaded = default(DateTime),
24                  TsStarted = DateTime.Now.ToUniversalTime(),
25                  TsCompleted = default(DateTime)
26              };
27              var metadataJson = new StringContent(
28                  JsonConvert.SerializeObject(metadata),
29                  Encoding.UTF8,
30                  Application.Json);
31              var httpResponseMessage = await httpClient.PutAsync("task
    /fileMetadata", metadataJson);
32              var response = httpResponseMessage.StatusCode;
33              Console.WriteLine(response);
34
35              await Entry_Point(input);
36
37          }
38          catch (Exception ex)
39          {
40              Console.WriteLine($"Failed to consume events on topic '{
    _topicName}': {ex.Message}");
41              Thread.Sleep(10000);
42          }
43      }
44
45      _consumer.Close();
46  }
```

Come si può vedere, il servizio tenta di consumare un messaggio proveniente

da Kakfa e, in caso di esito positivo, decomprime le informazioni e avvisa il Computing Provider che la richiesta è stata presa in carico, aggiornando lo status della computazione.

Una volta terminati questi passaggi preliminari, il motore di calcolo inizia il suo lavoro. Ad ogni step completato viene aggiunto il risultato parziale all'interno del Channel che raggruppa tutti i risultati parziali da inviare al Computing Provider.

```
var partialResult = JsonConvert.SerializeObject(item);
notification.Message = partialResult;
if (Globals.concurrentDictionary.TryGetValue(idFile, out _))
    _messageQueue.EnqueueAsync(notification);
```

Inoltre, all'inizio di ogni calcolo orario, viene effettuato il controllo sul CancellationToken definito in precedenza, che indica se è stata richiesta una terminazione anticipata del task.

```
if (Globals.notifications.TryGetValue(idFile, out Notification n))
{
    if (n.TokenSource.Token.IsCancellationRequested)
    {
        [...]
        Console.WriteLine("CALCULATION CANCELLED");
    }
}
```

Una volta terminati il calcolo e l'invio di tutti i risultati, viene chiamata l'ultima funzione descritta in questo servizio.

```
public async Task SendMetadata()
{
    try
    {
        var httpClient = new HttpClient()
        {
            BaseAddress = new Uri("http://computingprovider:80/"),
        };
        ComputingTasksMetadata metadata = new()
        {
            Id = idFile,
            FileName = "",
            Status = step < Max_Step ? "INTERRUPTED" : "COMPLETED",
            TsCompleted = DateTime.Now.ToUniversalTime(),
            TsUploaded = default(DateTime),
            TsStarted = default(DateTime),
```

```
17          };
18          var metadataJson = new StringContent(
19              JsonConvert.SerializeObject(metadata),
20              Encoding.UTF8,
21              Application.Json);
22          var httpResponseMessage = await httpClient.PutAsync("task/
    fileMetadata", metadataJson);
23          var response = httpResponseMessage.StatusCode;
24          Console.WriteLine(response);
25      }
26      catch (Exception ex)
27      {
28          Console.WriteLine($"Failed to send data: {ex.Message}");
29          Thread.Sleep(3000);
30      }
31  }
```

Essa permette di aggiornare nuovamente i metadati relativi al task completato e di contattare il Computing Provider.

Idealmente, sfruttando la definizione di scalabilità orizzontale, si possono creare diverse repliche dell'applicazione Compute Engine. In questo modo risulta possibile gestire il traffico delle varie richieste di calcolo e le risorse disponibili, estendendo il discorso in un ambiente Cloud-based. Con questa considerazione finale termina il capitolo dedicato al Compute Engine, passando alla trattazione dell'API-Gateway, il quale ha un ruolo fondamentale nell'architettura generale del sistema.

3.5 API-Gateway

3.5.1 Cos'è un API-Gateway?

Nell'architettura del sistema, tra il Computing Provider e l'interfaccia utente è presente un altro componente che permette a loro di comunicare, senza che essi siano collegati direttamente. Questo componente prende il nome di API-gateway e questo paragrafo è completamente dedicato ad esso, specialmente a come funziona e ai motivi che stanno dietro la sua implementazione.

Lo sviluppo di un API-gateway permette di ottenere numerosi vantaggi, soprattutto in un'architettura a microservizi come in questo caso. Ad esempio, nel caso si implementasse un nuovo microservizio risulta semplice collegarlo direttamente al gateway per includerlo all'interno del sistema generale, rispetto a modificare gli altri servizi già definiti. In questo modo i microservizi possono continuare ad essere considerati come delle entità indipendenti, con le quali si può comunicare attraverso delle API.

Sotto questo punto di vista l'API-gateway svolge il ruolo di proxy inverso, in modo tale che possa essere raggiunto da chiunque provenga dall'esterno e sia esso

a capire dove reindirizzare le richieste internamente.

Un altro aspetto importante che contraddistingue l'API-gateway è il fatto che agisca da intermediario tra la rete esterna, dove è presente il client, e la rete interna dove si trova il sistema. Questo permette di aumentare, significativamente, la sicurezza generale. Inoltre, al suo interno si possono sviluppare delle logiche di autenticazione e autorizzazione, come si vedrà tra poco, per limitare gli accessi solo a determinati tipi di utenti.

Su un servizio possono essere effettuate diverse richieste che porterebbero ad un sovraccarico [23]. Per risolvere questo problema, un API-gateway è in grado di gestire lo smistamento delle risorse, tenendo conto della capacità generale, svolgendo la funzione detta Load Balancing.

Ritornando a questo sistema, è stato scelto di utilizzare Ocelot come API-gateway per gestire le varie funzionalità appena descritte.

3.5.2 Ocelot: funzionamento generale

Ocelot è l'API-gateway scelto per questo progetto. I motivi della scelta sono riconducibili soprattutto al fatto che è specifico per i microservizi realizzati con .NET core, lo stesso framework utilizzato per scrivere il codice di questo progetto.

Ocelot, inoltre, è di tipo open-source e risulta essere leggero, veloce e semplice da implementare poiché si basa sulla scrittura di un semplice file JSON [24]. Fatto questo breve preambolo, da qui in avanti verrà discussa l'implementazione vera e propria dell'API-gateway.

Partendo dal file "Program.cs" sono state definite le seguenti righe di codice:

```
builder.Configuration.AddJsonFile("ocelot.json", optional: false,
    reloadOnChange: true);

builder.Services
    .AddOcelot(builder.Configuration)
    .AddDelegatingHandler<HeaderDelegatingHandler>();

await app.UseOcelot(configuration);
```

Esse permettono di indicare il nome del file dove verrà definito il JSON contenente tutte le configurazioni degli endpoint.

Il file JSON è composto da un array di coppie di "routes" che corrispondono ad un "mapping" tra gli endpoint visti dall'esterno e le relative API del servizio interno. Un esempio di quello che è appena stato detto è il seguente:

```
{
```

```
  "DownstreamPathTemplate": "/customers",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "computingprovider",
      "Port": "80"
    }
  ],
  "UpstreamPathTemplate": "/customers",
  "UpstreamHttpMethod": [ "GET" ],
  "AuthenticationOptions": {
    "AuthenticationProviderKey": "OpenIdConnect",
    "AllowedScopes": []
  },
  "RouteClaimsRequirement": {
    "realm_roles": "administration"
  },
  "DelegatingHandlers": [
    "HeaderDelegatingHandler"
  ]
}
```

Questo è come appare una correlazione tra API vista dall'interno e dall'esterno. Tralasciando un attimo i parametri "AuthenticationOptions", "RouteClaimsRequirement" e "DelegatingHandlers" di cui si discuterà in seguito, in queste righe di codice possiamo notare alcuni campi che definiscono il mapping:

- "DownstreamPathTemplate": indica la sintassi dell'API all'interno del microservizio;
- "DownstreamScheme": indica lo schema utilizzato per l'API in questione;
- "DownstreamHostAndPorts": permette di specificare l'hostname e la porta relativa al microservizio in cui è definita l'API;
- "UpstreamPathTemplate": indica la sintassi da utilizzare per chiamare quella determinata risorsa dall'esterno;
- "UpstreamPathMethod": indica il tipo di azione della risorsa in questione e può possedere i valori dei vari metodi http (GET, POST, PUT, DELETE, ...).

Un'altra sezione molto importante presente all'interno del file è quella relativa al BaseUrl.

```
"GlobalConfiguration": {
  "BaseUrl": "http://localhost:5193"
}
```

Quindi, nel modo appena descritto, è possibile definire in modo semplice e veloce un API-gateway per la gestione delle richieste. Come già annunciato in precedenza, con Ocelot è possibile gestire anche la parte riguardante l'autenticazione e l'autorizzazione, per bloccare o consentire gli accessi solo agli utenti che hanno il diritto di ottenere una determinata risorsa.

3.5.3 Ocelot: Autenticazione e autorizzazione

All'interno dell'API-gateway è stata implementata la parte di autenticazione e una prima parte di autorizzazione per un utente che vuole accedere ad una risorsa. Per descrivere più nel dettaglio il metodo utilizzato, bisogna sempre partire dal file "Program.cs":

```
builder.Services.AddAuthentication(options =>
{
    //Sets cookie authentication scheme
    options.DefaultAuthenticateScheme = CookieAuthenticationDefaults.
    AuthenticationScheme;
    options.DefaultSignInScheme = CookieAuthenticationDefaults.
    AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.
    AuthenticationScheme;
})
.AddCookie(cookie =>
{
    cookie.Cookie.Name = "keycloak.cookie";
    cookie.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;
    cookie.SlidingExpiration = true;
})
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, options
    => //OpenIdConnectDefaults.AuthenticationScheme => scheme used
    inside ocelot.json for "AuthenticationProviderKey"
{
    options.SignInScheme = CookieAuthenticationDefaults.
    AuthenticationScheme;
    options.Authority = "http://keycloak:8080/realms/Customers";
    options.ClientId = "***";
    options.ClientSecret = "***";
    options.RequireHttpsMetadata = false;
    options.GetClaimsFromUserInfoEndpoint = true;
```

```
22      options.Scope.Add("openid");
23      options.Scope.Add("profile");
24      options.Scope.Add("email");
25      options.SaveTokens = true;
26      options.ResponseType = OpenIdConnectResponseType.Code;
27      options.NonceCookie.SameSite = SameSiteMode.None;
28      options.CorrelationCookie.SameSite = SameSiteMode.None;
29  });
30
31  app.UseAuthentication();
32  app.UseAuthorization();
```

Tutte queste righe di codice permettono di inizializzare la parte di autenticazione e autorizzazione. Andando maggiormente nello specifico, si può osservare che, come IAM, è stato appunto utilizzato Keycloak e lo si può vedere controllando l'opzione relativa all'Authority. Essa corrisponde all'indirizzo dove è definito il client utilizzato. Per quanto riguarda quest'ultimo bisogna indicare tra le opzioni il ClientId e, in questo caso poiché è di tipo protetto, il ClientSecret. Vengono, inoltre, specificate altre opzioni come gli scope che permettono di indicare cosa sarà presente all'interno del token. Il collegamento tra Keycloak e l'API-gateway è stato un ulteriore ostacolo durante l'implementazione del servizio, soprattutto per la parte relativa ai ruoli di cui si parlerà tra poco.

All'interno dell'API-gateway è stato definito anche un controller, che espone degli endpoint chiamabili dall'esterno, tutti inerenti all'autenticazione.

LoginOptions() e getUserInfo()

```
1   [HttpGet("/login-options")]
2   public async Task<IActionResult> LoginOptions()
3   {
4       if (HttpContext.User.Identity.IsAuthenticated) //UTENTE LOGGATO
5           return Ok(JsonConvert.SerializeObject("Sei già loggato"));
6       else
7       {
8           Dictionary<string, string> urls = new Dictionary<string,
        string>
9           {
10              { "keycloak", "http://localhost:5193/oauth2/login" }
11          };
12          return Ok(urls);
13      }
14
15
16  }
17
```

```
18 [HttpGet("/me")]
19 public IActionResult getUserInfo()
20 {
21     if (HttpContext.User.Identity.IsAuthenticated)
22     {
23         var roles = User.Claims.Where(c => c.Type == "realm_roles").
       ToList();
24         var role = "";
25         foreach (var r in roles)
26         {
27             if (r.Value == "customer" || r.Value == "administration")
28             {
29                 role = r.Value;
30                 break;
31             }
32         }
33         var response = new
34         {
35             islogged = true,
36             name = User.Claims.Where(c => c.Type == "name").
       FirstOrDefault().Value,
37             email = User.Claims.Where(c => c.Type == "http://schemas.
       xmlsoap.org/ws/2005/05/identity/claims/emailaddress").
       FirstOrDefault().Value,
38             role
39         };
40         return Ok(response);
41     } else
42     {
43         var response = new{
44             islogged= false
45         };
46         return Ok(response);
47     }
48 }
```

I primi due endpoint di cui si vuole discutere sono i due rappresentati qui sopra. Essi sono correlati tra loro perché, entrambi, vengono chiamati immediatamente dal client attraverso la propria GUI. “LoginOptions()” permette di restituire, in caso di utente non loggato, un dizionario contenente le coppie chiave-valore “(IAM, URL per effettuare il login)”. Il metodo “getUserInfo()” permette di restituire, nel caso di utente loggato, informazioni che caratterizzano la persona loggata.

Login()

```
1 [Authorize]
2 [HttpGet("/oauth2/login")]
```

```
public async Task Login()
{
    var authResult = await HttpContext.AuthenticateAsync(
    OpenIdConnectDefaults.AuthenticationScheme);
    if (authResult?.Succeeded != true)
    {
        // ERRORE
        Console.WriteLine("ERRORE");
        HttpContext.Response.Redirect("http://localhost:5192/Home");
    }

    var accessToken = authResult.Properties.GetTokenValue("
    access_token");
    var refreshToken = authResult.Properties.GetTokenValue("
    refresh_token");
    var ID_token = authResult.Properties.GetTokenValue("id_token");

    // Salvo i token nella sessione
    HttpContext.Session.SetString("access_token", accessToken);
    HttpContext.Session.SetString("refresh_token", refreshToken);
    HttpContext.Session.SetString("ID_token", ID_token);

    HttpContext.Response.Redirect("http://localhost:5192/Home");
}
```

Il metodo “Login()” viene chiamato quando un utente vuole autenticarsi.

L’annotazione “[Authorize]” permette di controllare se l’utente è già autenticato oppure no. In caso di esito negativo, viene reindirizzato alla pagina di login dello IAM per effettuare l’autenticazione. In questo caso, si viene reindirizzati alla pagina di login di Keycloak. Una volta completata l’autenticazione vengono salvati in una sessione i token dell’utente in questione.

Logout()

```
[HttpPost("/oauth2/logout")]
public void Logout()
{
    var ID_token = HttpContext.Session.GetString("ID_token");
    CookieOptions cookieOptions = new CookieOptions();
    cookieOptions.Expires = DateTime.Now.AddDays(-1);
    HttpContext.Response.Cookies.Append("keycloak.cookie", "",
    cookieOptions);

    HttpContext.Response.Redirect("http://keycloak:8080/realms/
    Customers/protocol/openid-connect/logout?id_token_hint=" +
    ID_token + "&post_logout_redirect_uri=http://localhost:5192/Home")
    ;
```

```
}
```

Il seguente metodo, invece, permette di effettuare il logout, facendo attenzione di eliminare la sessione in modo corretto ambo le parti, ovvero sia lato client, sia lato Keycloak. Se, non si presta particolare attenzione a ciò, può sembrare che il logout sia andato a buon fine, ma in realtà ricaricando la pagina si risulta ancora loggati.

Con questo ultimo metodo termina la descrizione degli endpoint presenti nel controller.

Arrivati a questo punto, ritornando al file JSON dove sono definite le configurazioni, si può parlare dei parametri saltati, ovvero "AuthenticationOptions", "RouteClaimsRequirement" e "DelegatingHandlers".

```
"AuthenticationOptions": {
  "AuthenticationProviderKey": "OpenIdConnect",
  "AllowedScopes": []
},
"RouteClaimsRequirement": {
  "realm_roles": "administration"
},
"DelegatingHandlers": [
  "HeaderDelegatingHandler"
]
```

Partendo dal primo, con il parametro "AuthenticationOptions" si vuole indicare che per accedere ad una determinata risorsa è necessario essere autenticati. Al suo interno troviamo il campo "AuthenticationProviderKey", che indica lo schema utilizzato per l'autenticazione e corrisponde al primo valore inserito come parametro nel metodo "AddOpenIdConnect", all'interno di "Program.cs". Il secondo campo, chiamato "AllowedScopes", permette, inizialmente, a tutti gli utenti di accedere alla risorsa. Nonostante ad una prima impressione risulta essere un errore, questo è corretto perché il controllo sull'autorizzazione viene svolto con i successivi due campi.

Nel campo "RouteClaimsRequirement", vengono specificati i permessi che un utente deve avere per poter accedere al servizio. Se si vuole specificare più ruoli è necessario separarli tutti con una virgola (Esempio: "administration, customer"). Per soddisfare tale richiesta, si va a controllare il campo "realm_roles" presente all'interno dell'access token, il quale corrisponde ad un vettore composto dall'insieme di ruoli che un utente possiede.

L'implementazione di quello che si è appena descritto corrisponde alla scrittura di una parte di configurazione di Ocelot, sempre presente sul file "Program.cs".

```
        "realm_roles": [
          "default-roles-customers",
          "offline_access",
          "uma_authorization",
          "customer"
        ],
        "email_verified": false,
        "name": "Mario Rossi",
        "preferred_username": "mario@rossi.com",
        "given_name": "Mario",
        "family_name": "Rossi",
        "email": "mario@rossi.com"
      }
```


Figura 3.3: Porzione di token

```
await app.UseOcelot(configuration);

var configuration = new OcelotPipelineConfiguration
{
    AuthorizationMiddleware = async (ctx, next) =>
    {
        if (Authorize(ctx))
        {
            await next.Invoke();
        }
        else
        {

            ctx.Items.SetError(new UnauthorizedError("ERROR FORBIDDEN
    "));
            return;
        }

    }
};
```

Nella riga di codice dove c'è scritto "app.UseOcelot()" viene inserito come unico parametro la configurazione appena rappresentata. Essa corrisponde ad un middleware custom che richiama una funzione chiamata "Authorize(ctx)".

```
bool Authorize(HttpContext ctx)
{
```

```
3       DownstreamRoute route = (DownstreamRoute)ctx.Items["
    DownstreamRoute"];
4       string key = route.AuthenticationOptions.
    AuthenticationProviderKey;
5
6       if (key == null || key == "") return true;
7       if (route.RouteClaimsRequirement.Count == 0) return true;
8       else
9       {
10          //flag for authorization
11          bool auth = false;
12          var count = 0;
13          //where are stored the claims of the jwt token
14          var claims = ctx.User.Claims.Where(c => c.Type == "
    realm_roles");
15
16          //where are stored the required claims for the route
17          Dictionary<string, string> required = route.
    RouteClaimsRequirement;
18          foreach (KeyValuePair<string, string> claim in required)
19          {
20              var claimRequired = claim.Value.Split(",");
21              foreach (var cr in claimRequired)
22              {
23                  Console.WriteLine(cr);
24                  foreach (var cl in claims)
25                  {
26                      if (cl.Value == cr.Trim())
27                      {
28                          count++;
29                      }
30                  }
31
32                  if (count > 0)
33                  {
34                      auth = true;
35                      break;
36                  }
37              }
38          }
39          return auth;
40      }
41  }
```

Questa funzione, semplicemente, preleva dai Claims tutti i ruoli che un utente possiede all'interno del proprio access token e li confronta con quelli inseriti all'interno della configurazione di Ocelot nel file "ocelot.json". Ritorna "true" in caso di esito positivo, altrimenti "false". In questo modo è possibile filtrare gli utenti in base ai ruoli che possiedono.

Finita la spiegazione riguardante il campo "RouteClaimsRequirement", l'ultimo rimasto risulta essere "HeaderDelegatingHandler". Nonostante l'autorizzazione di un utente viene verificata nel punto precedente, si vuole implementare una politica di Zero-Trust per questo sistema, ovvero bisogna effettuare in diversi punti i controlli riguardanti la sicurezza, come se ognuno di essi fosse l'unico meccanismo di controllo. Ed è secondo questo principio che subentra l'ultimo campo rimanente, dove viene semplicemente scritto il nome della seguente classe:

```
public class HeaderDelegatingHandler : DelegatingHandler
{
    private readonly IHttpContextAccessor _contextAccessor;

    public HeaderDelegatingHandler(IHttpContextAccessor
    contextAccessor)
    {
        _contextAccessor = contextAccessor;
    }

    protected override async Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request, CancellationToken cancellationToken)
    {

        var securityToken = _contextAccessor.HttpContext.Session.
    GetString("access_token");

        if (!string.IsNullOrWhiteSpace(securityToken))
        {
            request.Headers.Authorization = new
    AuthenticationHeaderValue("Bearer", securityToken);

        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

Essa permette di recuperare l'access token dell'utente, in modo tale da essere inserito all'interno degli Headers, nella sezione Authorization. Esso verrà inviato insieme alla richiesta della risorsa al servizio interno, in modo tale da effettuare la verifica anche in un punto più interno del sistema. Il token verrà poi utilizzato per creare le policies, situate all'interno del Computing Provider.

```
builder.Services
        .AddAuthentication(auth =>
        {
```

```
4               auth.DefaultAuthenticateScheme = JwtBearerDefaults.
    AuthenticationScheme;
5               auth.DefaultChallengeScheme = JwtBearerDefaults.
    AuthenticationScheme;
6           }).AddJwtBearer(options =>
7           {
8               options.RequireHttpsMetadata = false;
9               options.Authority = "http://keycloak:8080/realms/
    Customers";
10              options.Audience = "***";
11              options.IncludeErrorDetails = true;
12              options.SaveToken = true;
13
14              options.TokenValidationParameters = new()
15              {
16                  ValidateIssuer = true,
17                  ValidateAudience = true,
18                  ValidateLifetime = true,
19                  ValidateIssuerSigningKey = false,
20                  ValidIssuer = "http://keycloak:8080/realms/Customers"
    ,
21                  NameClaimType = System.Security.Claims.ClaimTypes.
    Name
22              };
23          });
24
25  builder.Services.AddAuthorization(options =>
26  {
27      options.AddPolicy("Customer", policy =>
28              policy.RequireClaim("realm_roles", ["customer", "
    administration"]));
29      options.AddPolicy("Administration", policy =>
30              policy.RequireClaim("realm_roles", "administration"));
31  });
```

Come si può notare la prima parte è simile a quella presente nel file Program.cs all'interno dell'API-gateway, con la differenza che viene utilizzato il metodo AddJwtBearer(), invece di AddOpenIdConnect(). Esso permette di prelevare il token definito all'interno dell'header della richiesta HTTP presente nella sezione Authorization e di utilizzarlo, tra le varie opzioni, per la definizione delle policies.

Qui termina la parte relativa all'API-gateway, passando alla trattazione dell'ultima applicazione presente nel sistema.

3.6 Client-GUI

3.6.1 Panoramica

L'ultima applicazione di cui si vuole discutere, facente sempre parte dell'architettura del progetto, prende il nome di Client-GUI. Essa consiste in una simulazione di un'interfaccia grafica, capace di comunicare con il resto del sistema. Si utilizza il termine simulazione, nonostante sia un'applicazione di tipo front-end a tutti gli effetti, perché il suo reale scopo è quello di testare che tutti servizi soddisfino le richieste in modo corretto.

La Client-GUI è di tipo Single Page Application ed è stata implementata utilizzando il framework React, insieme ai linguaggi di programmazione HTML, CSS e Javascript nativo. È possibile spostarsi da una pagina ad un'altra grazie alla definizione di un Browser Router, facente parte della libreria "react-router-dom". Esso al suo interno sarà popolato da diverse Route, che corrisponderanno a diverse pagine.

3.6.2 Funzionamento generale

In questo sottoparagrafo si andrà a descrivere il funzionamento dell'applicazione, attraverso varie immagini in modo da simulare una demo.

L'esecuzione parte con la schermata iniziale dove si chiede all'utente di effettuare il login. Immediatamente, una volta caricata la pagina verranno chiamate le API "/me" e "/login-options" definite all'interno dell'API-gateway, di cui si è già discusso nel paragrafo precedente. Queste due chiamate permettono, rispettivamente, di ottenere informazioni sull'utente nel caso fosse già loggato e di ottenere un dizionario contenente le informazioni per effettuare il login, come l'URL specifico per un determinato IAM. In questo caso, è stato scelto Keylcoak. Nella figura 3.4 è

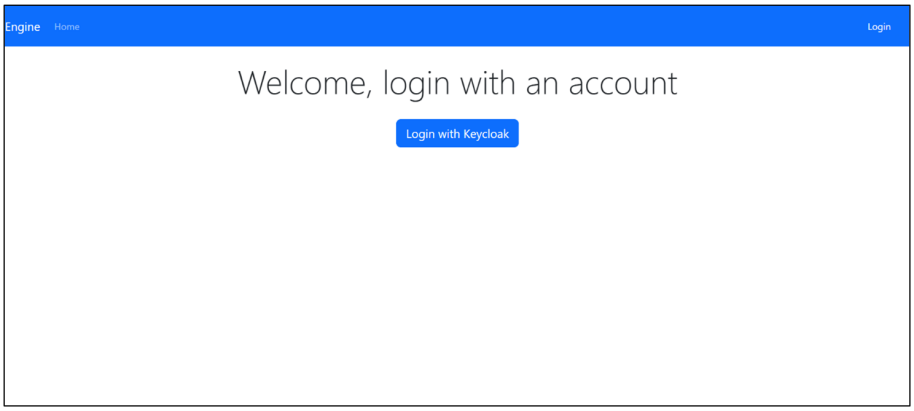

Figura 3.4: Pagina con route "/Home"

rappresentata la pagina iniziale che permette di cliccare nel tasto centrale o in quello in alto a destra presente sulla navbar per effettuare l'autenticazione. Con tale azione, si effettua la chiamata all'endpoint "/login" presente sull'API-gateway, il quale nel caso di utente non autenticato, reindirizza il cliente alla pagina di login di Keycloak.

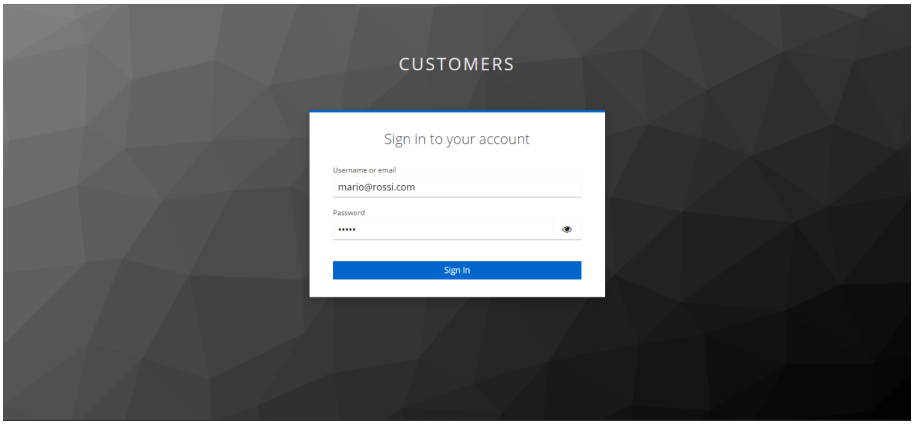

Figura 3.5: Pagina di login di Keycloak

Nella figura 3.5 si può osservare un cliente fittizio (Mario Rossi) che si logga nella pagina di Keycloak designata per effettuare il login.

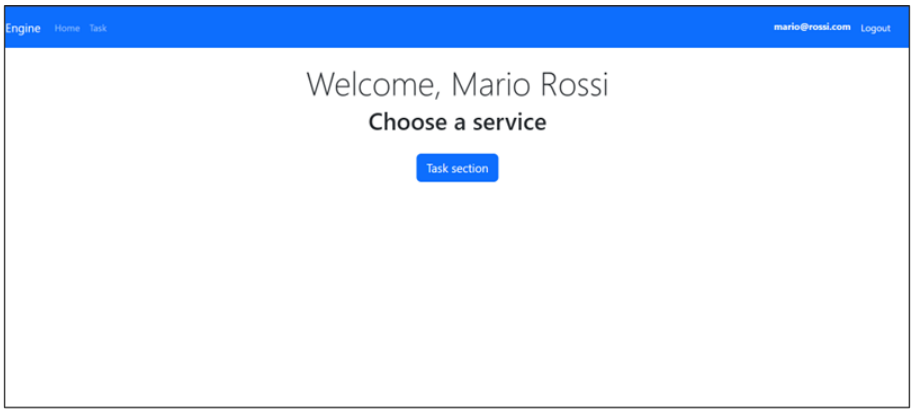

Figura 3.6: Pagina con route "/Home" con utente loggato

Una volta completata l'operazione si ritorna nella schermata precedente come mostrato in figura 3.6. È stato utilizzato il verbo "ritorna" perché essa corrisponde sempre alla pagina raggiungibile con il percorso "/Home", con la differenza che adesso si possono vedere alcune informazioni dell'utente che ha effettuato il login, quali il nome, il cognome e la mail in alto a destra. Inoltre, risulta accessibile la sezione per provare i vari servizi che il sistema offre e il tasto per effettuare il logout

sulla barra di navigazione. Nel caso in cui l'utente fosse già stato autenticato in precedenza, si sarebbe ritrovato direttamente in questa schermata saltando le due precedenti.

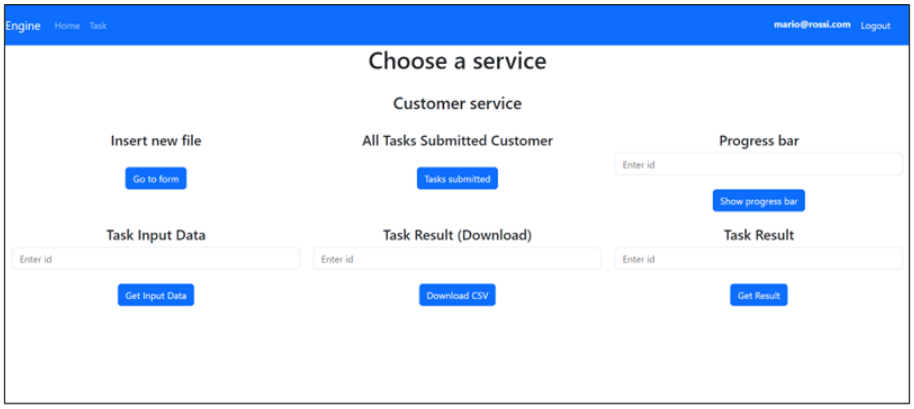

Figura 3.7: Pagina con route "/Task"

Cliccando sul pulsante "Task section" risulta possibile provare i vari servizi (Figura 3.7). Essendo, in questo esempio, Mario Rossi un utente di tipo "customer" risulta possibile accedere solo ai servizi corrispondenti al suo livello di privilegio. Più avanti si andrà a mostrare anche la schermata visualizzata da un admin, per avere un confronto visivo tra i due ruoli. Cliccando sul primo servizio, l'utente può sottomettere un nuovo lavoro.

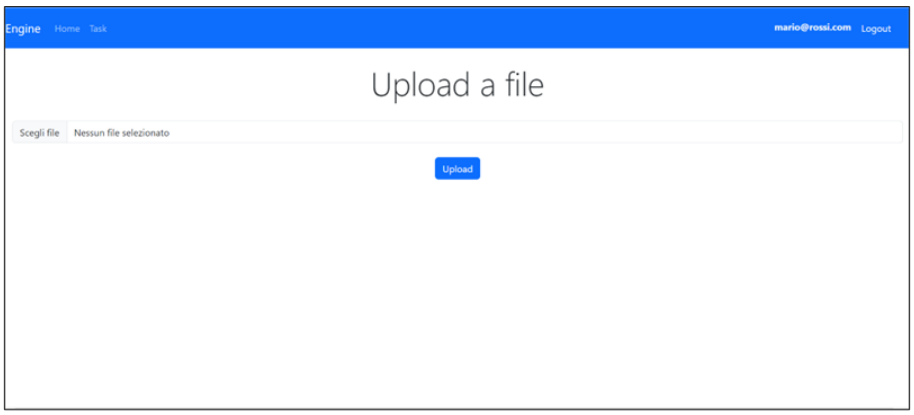

Figura 3.8: Pagina con route "/FileUpload"

Egli raggiungerà la pagina che permette di effettuare l'upload di un file di tipo JSON, contenente i valori di input da mandare in esecuzione sul motore (Figura 3.8).

Una volta cliccato sul pulsante sottostante, si attende che il task venga preso

in carico dal Compute Engine. A questo punto ci sono due possibilità, o si attende fino a visualizzare la progress bar (Figura 3.9) che indica la percentuale di completamento del calcolo, oppure cliccando sulla scritta “Task” presente sulla navbar è possibile tornare alla visualizzazione di tutti i servizi.

Nonostante si scelga la seconda opzione, risulta sempre possibile visualizzare la barra di completamento in vari modi, tra questi cliccando sul terzo servizio, specificando l’id del task.

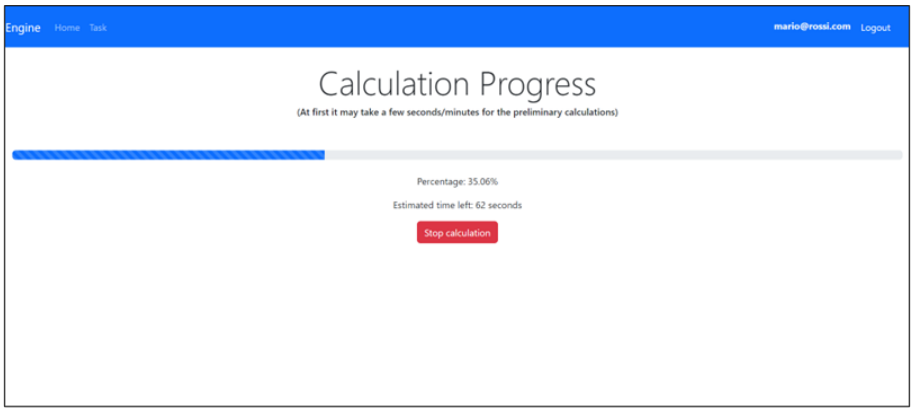

Figura 3.9: Pagina con route “/ProgressBar"

La pagina "Calculation Progress" permette di visualizzare una barra in continuo aggiornamento, con annessi percentuale di caricamento e tempo stimato per il completamento. Nel caso si volesse terminare in anticipo il calcolo, si può cliccare sul pulsante “Stop calculation” che permette di settare lo status del lavoro a “INTERRUPTED” e reindirizzare l’utente alla schermata con tutti i servizi.

Se l’utente volesse scoprire tutti i task sottomessi, dovrebbe cliccare sul bottone sotto la scritta “All Tasks Submitted Customer”. Una volta cliccato, comparirà una lista con tutti i lavori sottomessi dall’utente che ne ha fatto richiesta (Figura 3.10). Inoltre, per i task in corso, risulta possibile accedere alla barra di completamento relativa al lavoro presente sulla medesima riga.

Un altro servizio disponibile all’utente è la possibilità di ottenere i valori di input di un determinato task, specificandone l’Id, cliccando sul bottone relativo a “Task Input Data”, come mostrato in Figura 3.11. In questo modo risulta possibile visualizzare il contenuto della tabella “ComputingTasks” presente nel database. Il campo FileContent va a simulare il rispettivo campo nella tabella riportata in precedenza perché, nonostante venga restituito alla chiamata dell’API nell’oggetto di tipo JSON, esso corrisponde al contenuto del file di input, quindi non visualizzabile per motivi di spazio.

Ritornando alla schermata della scelta dei servizi, gli ultimi due pulsanti svolgono la medesima funzione, ovvero la stampa dei risultati di un lavoro sottomesso in

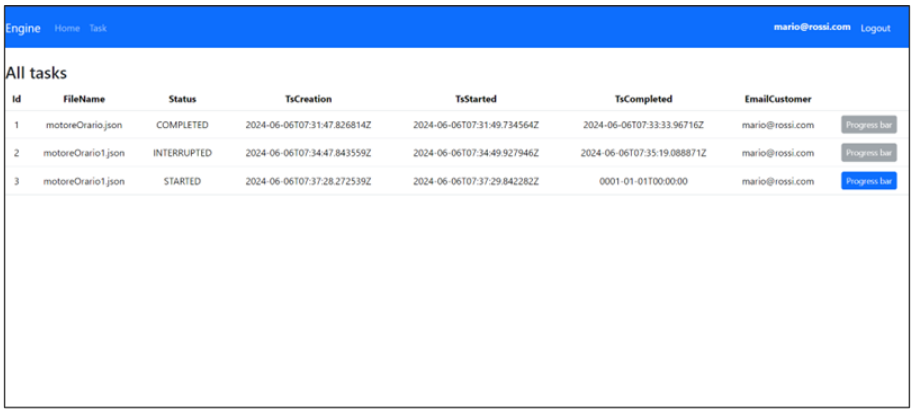

Figura 3.10: Pagina con route “/customers/all-task"

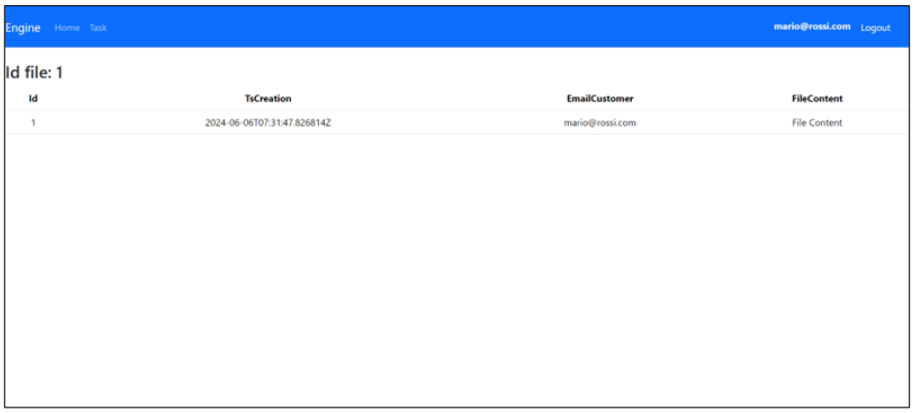

Figura 3.11: Pagina con route “/customers/task-input"

precedenza, sempre specificandone l’Id. Nel primo caso vengono inseriti all’interno di un file con estensione .CSV, mentre nel secondo caso vengono stampati a schermo (Figura 3.12).

Essi sono suddivisi in diverse pagine, dove vengono rappresentati dieci giorni di risultati per ognuna di esse. Inoltre, è possibile visualizzare maggiori dettagli sui risultati cliccando sul componente denominato "More details".

Quest’ultimo punto termina la parte relativa ad un utente di tipo "customer", spostando ora il discorso ad un utente di tipo "administration".

Come si può vedere dalla figura 3.13, se l’utente collegato ha il ruolo di "administration", vengono visualizzati due ulteriori servizi. Il primo permette di ottenere una lista di tutti i clienti che hanno sottomesso almeno un task all’interno di una pagina (Figura 3.14).

Mentre il secondo permette di visualizzare tutti i task sottomessi da tutti i clienti (Figura 3.15).

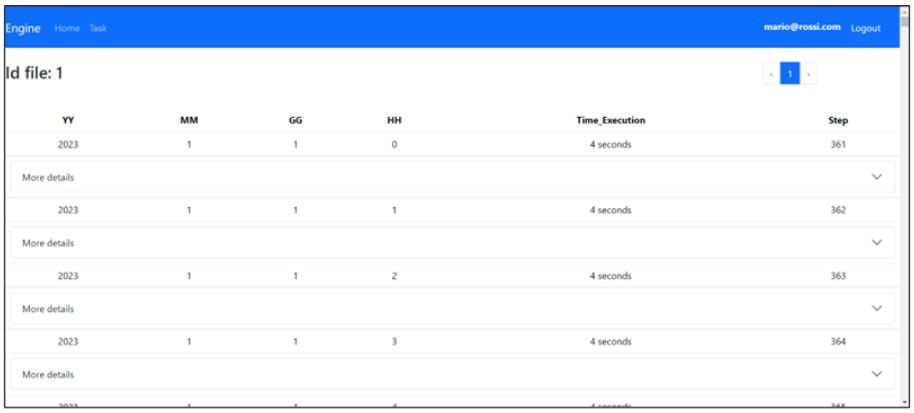

Figura 3.12: Pagina con route “/customers/task-result"

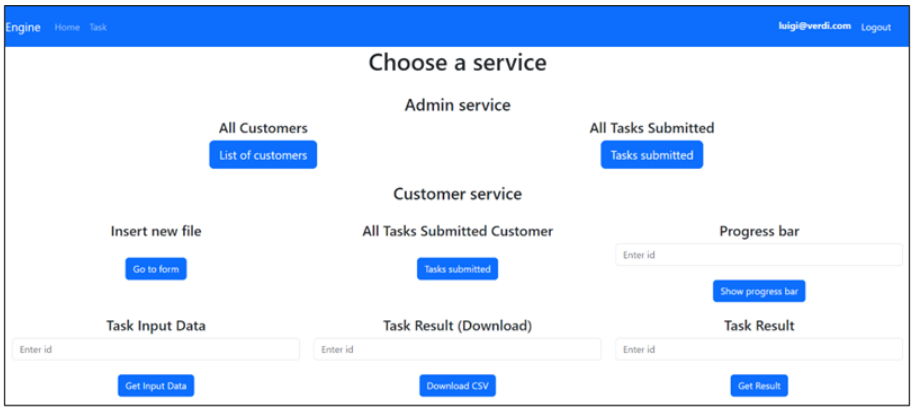

Figura 3.13: Pagina con route “/Task" versione utente "admin"

Con la visualizzazione dell’applicazione, secondo il punto di vista di un admin, termina il paragrafo relativo alla ClientGUI. Il prossimo verterà sull’implementazione dei vari container di Docker, utilizzati per far funzionare l’intero sistema.

3.7 Docker-Compose

3.7.1 Panoramica

Per eseguire tutte le applicazioni in modo semplice e veloce, viene utilizzata la piattaforma chiamata Docker. In questo paragrafo si vuole descrivere come è stato integrato in questo sistema. È stato definito un file chiamato "docker-compose.yml", che permette di definire tutti i container facente parte del progetto. Prima di descriverlo più nel dettaglio, risulta prima doveroso fare delle precisazioni sui file utilizzati per creare i singoli container, ovvero i vari Dockerfile. Essi, come si

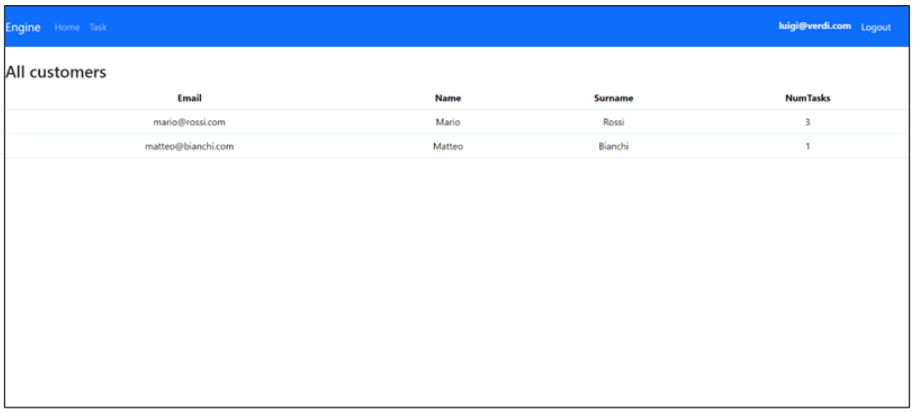

Figura 3.14: Pagina con route “/admin/lists-customers"

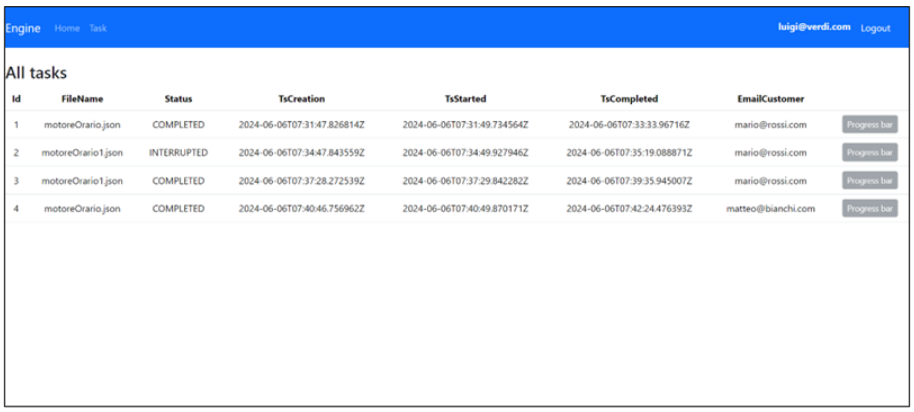

Figura 3.15: Pagina con route “/admin/all-tasks"

ricorda, permettono di definire un’immagine, che fungerà da base per i container. Nel sottoparagrafo successivo, si andrà a descrivere il file "docker-compose.yml", specificando gli elementi chiave che lo compongono.

3.7.2 Il file "docker-compose.yml"

Prima di iniziare a trattare più nel dettaglio il file si può notare che, in alcuni container, sono definite delle coppie di porte. Esse corrispondono, rispettivamente, alla porta vista dall’esterno del network e a quella utilizzata internamente per comunicare (Host_Port:Container_Port).

La descrizione parte con il container dedicato al Computing Provider.

```
1 computingprovider:
```

```
    container_name: computingprovider
    image: computingprovider:latest
    restart: unless-stopped
    ports:
      - '3001:80'
      - '3002:443'
    environment:
      - DefaultConnection=${DEFAULT_CONNECTION}
      - Authority=${KEYCLOAK_AUTHORITY}
      - Audience=${KEYCLOAK_AUDIENCE}
```

Gli sono state assegnate due coppie di porte, in base al tipo di protocollo utilizzato per la trasmissione delle informazioni. La coppia 3001:80 corrisponde al protocollo http, mentre 3002:443 per il protocollo https.

Per quanto riguarda l'ambiente di Kafka sono stati definiti diversi container. Nel seguente elenco sono descritti in base al nome:

- kafka: utilizza l'immagine di Kafka sviluppata da Bitnami[1] (bitnami/kafka) e permette di definire il broker;
- kafka-init: utilizza la medesima immagine del container precedente. Corrisponde ad uno dei due container di configurazione presenti nel file e permette di creare il topic che conterrà i vari messaggi;
- kafka-connector: utilizza un'immagine custom descritta subito dopo. Corrisponde al Source Connector di Kafka, il quale ha il compito di prelevare i dati dal db e inserirli nel topic;
- kafka-connector-init: secondo container di configurazione, che permette di creare il connector, specificando le varie proprietà.

Il container kafka-init utilizza il seguente file di script per creare il topic:

```
#/bin/bash

/opt/bitnami/kafka/bin/kafka-topics.sh --bootstrap-server kafka:29092
    --list
echo -e 'Creating kafka topics'

/opt/bitnami/kafka/bin/kafka-topics.sh --bootstrap-server kafka:29092
    --create --if-not-exists --topic $TO_ELABORATE_TOPIC_NAME --
    partitions 7
```

[1] https://bitnami.com/

```
echo "Topic $TO_ELABORATE_TOPIC_NAME was create"
```

Kafka-connector come immagine utilizza quella di partenza di Confluent[2] (confluentinc/cp-kafka-connect), al quale si aggiungono due plugin che permettono la comunicazione con il database. Inoltre, viene svolto un controllo tramite file di script per valutare se il container si trova nello stato di Healthy.

```
#!/bin/sh

echo "Waiting for Kafka Connect to start listening on kafka-connect"
if [ $(curl -s -o /dev/null -w %{http_code} http://kafka-connector
    :8083/connectors) -ne 000 ] ; then
  echo -e $(date) " Kafka Connect listener HTTP state: " $(curl -s -o
    /dev/null -w %{http_code} http://kafka-connector:8083/connectors)
    " "
  echo " Kafka Connect is ready "
  exit 0
fi
  echo -e $(date) " Kafka Connect listener HTTP state: " $(curl -s -o
    /dev/null -w %{http_code} http://kafka-connector:8083/connectors)
    " (waiting for 200)"
  exit 1
```

Infine, kafka-connector-init esegue un file di configurazione per inizializzare il source connector utilizzato in questo sistema:

```
#!/bin/sh

echo -e "\n--\n+> Creating Kafka Connect source"

# Source connector init
response=404
until [ $response != 404 ]
do
    response=$( curl -s -o /dev/null -w "%{http_code}" -X "POST" "
    http://kafka-connector:8083/connectors/" \
                -H "Content-Type: application/json" \
                -d '{
                    "name": "Source-connect",
                    "config": {
                        [...]
                    }
                }'
```

[2]https://www.confluent.io/

```
    )
    echo $response
    sleep 1
done
```

Al suo interno vengono definite alcune proprietà come il nome della tabella da cui prelevare i dati, la modalità, lo schema che deve avere il risultato e molto altro.

Un altro container definito è quello inerente al database (postgres) dove verranno salvate le informazioni. Per inizializzarlo viene utilizzato un file contenente tutte le configurazioni per creare le tabelle in linguaggio SQL.

```
postgres:
    image: postgres
    restart: always
    ports:
      - 5433:5432
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    volumes:
      - ./postgres-data-tables:/var/lib/postgresql/data
      - ./sql/create_tables.sql:/docker-entrypoint-
  initdb.d/create_tables.sql
```

Il prossimo servizio da analizzare è Keycloak, per il quale sono stati definiti due container:

- keycloak:

```
    keycloak:
    image: keycloak/keycloak:latest
    container_name: keycloak
    restart: unless-stopped
    ports:
      - "8080:8080"
    expose:
      - "8080"
    command: start-dev --import-realm
    volumes:
      - ./keycloak:/opt/keycloak/data/import
    environment:
```

```
13        KC_DB: postgres
14        KC_DB_USERNAME: ${POSTGRES_KEYCLOAK_USER}
15        KC_DB_PASSWORD: ${POSTGRES_KEYCLOAK_PASSWORD}
16        KC_DB_URL: ${POSTGRES_KEYCLOAK_URL}
17        KEYCLOAK_ADMIN: ${KEYCLOAK_ADMIN}
18        KEYCLOAK_ADMIN_PASSWORD: ${
      KEYCLOAK_ADMIN_PASSWORD}
19
```

Per importare le configurazioni di un realm, si può utilizzare il comando definito nella sezione corrispettiva, che va a leggere il file presente nella cartella "/opt/keycloak/data/import".

- keycloak-db: corrisponde ad un database postgreSQL, però esclusivo di Keycloak. Al suo interno verranno salvate tutte le informazioni inerenti agli utenti, i client, ecc...

È stato creato un container per redis, che permette di definire una memoria cache distribuita.

Infine, gli ultimi container definiscono le tre applicazioni rimanenti del sistema: Compute Engine, API-gateway e Client-GUI. In tutti e tre i casi sono stati definiti dei Dockerfile per creare le immagini. Partendo dall'immagine dotnet/sdk, le due istruzioni principali sono le seguenti:

```
1 RUN dotnet restore
2 RUN dotnet publish -c Release -o out
```

Esse permettono, rispettivamente, di ripristinare le dipendenze di un progetto, tra i quali i pacchetti NuGet[3], e di compilare il progetto con il tag Release.

3.7.3 Considerazioni finali sul file

Terminata la spiegazione sul file "docker-compose.yml", si possono fare alcune considerazioni finali su di esso prima di terminare il terzo capitolo. È possibile notare che in alcuni servizi, nella sezione environment, sono presenti delle variabili assegnate con dei valori scritti con la sintassi ${...}. Esse indicano dei valori che sono definiti in un altro file nascosto che prende il nome di .env. Al suo interno è possibile racchiudere tutte le variabili che non si vogliono dichiarare direttamente nel file di docker compose.

[3]Permettono di definire le librerie. Più dettagli sulla documentazione di Microsoft: https://learn.microsoft.com/it-it/nuget/what-is-nuget

I parametri presenti nella sezione delle variabili d'ambiente sono utilizzati all'interno dei servizi in cui sono definite. Vengono definite a questo livello, in modo tale che eventuali modifiche possano essere effettuate senza andare a cambiare linee di codice internamente nel servizio. Questa operazione prende il nome di externalized configuration.

Con questo ultimo concetto termina il paragrafo relativo a Docker e, più in generale, il capitolo 3 relativo all'implementazione del progetto. Nel prossimo capitolo verranno esposti i risultati ottenuti, gli sviluppi futuri e le conclusioni della tesi.

Capitolo 4

Risultati e Conclusioni

4.1 Risultati

Terminato il capitolo dedicato all'implementazione del sistema, si possono stilare i risultati finali ottenuti.

Effettuando dei test sulla soluzione proposta, gli esiti raccolti sono stati incoraggianti. L'architettura sviluppata è in grado di ospitare un programma per il calcolo scientifico e di eseguirlo in maniera corretta e ottimizzata. Nonostante tutte le prove siano state realizzate su un singolo PC, si ricorda che l'architettura a microservizi risulta essere una soluzione molto valida soprattutto in un ambiente distribuito o nel contesto del Cloud Computing.

Nel grafico rappresentato in figura 4.1 sono stati analizzati tre file che contengono informazioni di input diverse, in modo da valutare le differenze nelle tempistiche e nelle performance generali. Il confronto è tra la soluzione in locale del programma su ambiente Windows e la soluzione containerizzata, basata su WSL. I file descritti sono definiti con nomi fittizi, ovvero File1, File2 e File3, rispettivamente di dimensione 1.809KB, 1.966KB e 1.250KB.

Come si può osservare le tempistiche per il completamento di un calcolo scientifico sono molto simili. Tuttavia, il vantaggio principale della versione descritta in questo documento, è quello di permettere lo sviluppo orizzontale della soluzione, cioè con la possibilità di eseguire diverse repliche dello stesso programma.

Per testare la presenza di diverse repliche del Compute Engine, sono stati definiti tre container con la medesima immagine e sono stati inseriti tre file di input in contemporanea. Il comportamento risultante è stato soddisfacente in quanto, grazie alle configurazioni inserite nel message broker Kafka, è stato possibile dividere il traffico in entrata del motore. Sfruttando sempre la medesima tecnologia, risulta possibile definire diverse code (topic), ognuna delle quali aventi diverse partizioni, in modo tale da creare, ad esempio, delle priorità diverse, in base al peso di una

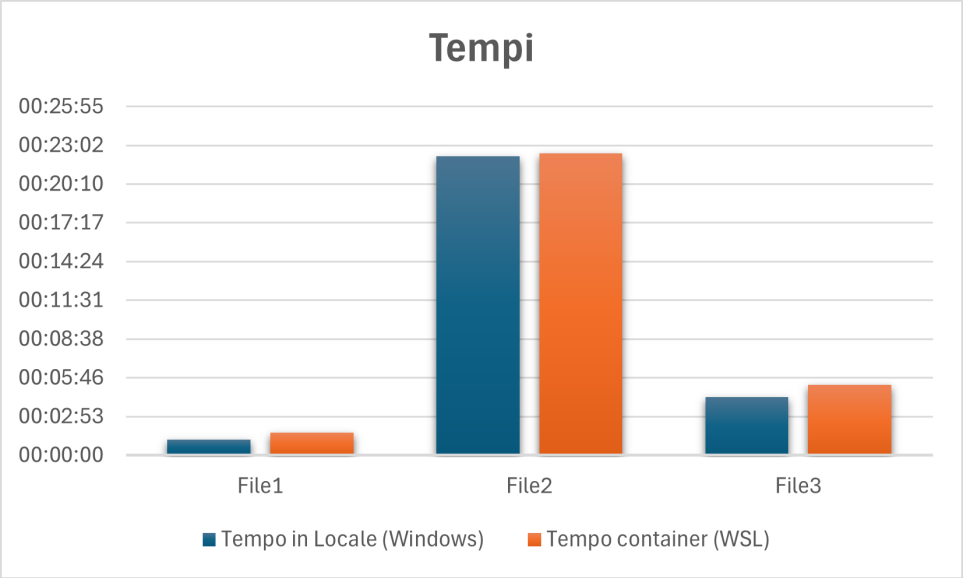

Figura 4.1: Istogramma tempistiche calcolo

computazione, al livello di account di un cliente e così via.

La presenza di diverse repliche permette di non rendere inoperabile il sistema in caso di errori, in quanto il traffico sarebbe ridistribuito con i servizi attivi. Inoltre, essendo il sistema definito all'interno di un'architettura a microservizi, risulta più semplice effettuare modifiche e miglioramenti per ottimizzare ulteriormente i tempi.

4.2 Progetti futuri

L'architettura a microservizi implementata e discussa in questa tesi è stata pensata per risolvere alcuni problemi, relativi alle tempistiche troppo elevate in caso di progetti molto complessi. Di conseguenza, si presentava l'esigenza di utilizzare le risorse di elaborazione migliori sul mercato. Da questi presupposti si può pensare a quali potrebbero essere i progetti futuri relativi a questo sistema.

Uno su tutti, è l'utilizzo di questa soluzione in un contesto di Cloud Computing, sfruttando tutti i vantaggi che essa offre, come l'utilizzo di risorse tecnologiche sempre all'avanguardia. Bisognerebbe svolgere ulteriori test, in modo da valutarne la corretta esecuzione e l'efficacia della soluzione proposta.

Un altro miglioramento possibile è un incremento della sicurezza e affidabilità generale. Ad esempio, si può valutare l'inserimento di diverse repliche per le code di Kafka, andando a proteggere eventuali errori improvvisi.

Un'altra ottimizzazione potrebbe essere l'introduzione di un orchestratore di container, come Kubernetes. Con esso è possibile definire diverse istanze (nodi) del

medesimo container in modo facile e veloce. Inoltre, risulta possibile effettuare un lavoro di Load Balancing, andando a bilanciare i lavori in entrata per ogni singolo servizio.

Infine, per sfruttare completamente il concetto di architettura a microservizi, si potrebbero aggiungere ulteriori servizi a quelli già esistenti. Un esempio potrebbe essere uno dedito alla raccolta di statistiche sui calcoli degli utenti oppure uno che svolga la funzione di datastore.

Bibliografia e Sitografia

[1] Microsoft. *Cos'è il cloud computing?* URL: https://azure.microsoft.com/it-it/resources/cloud-computing-dictionary/what-is-cloud-computing (visited on 2024) (cit. on p. 3).

[2] IBM. *Cosa sono i microservizi?* URL: https://www.ibm.com/it-it/topics/microservices (visited on 2024) (cit. on p. 4).

[3] Amazon. *Cosa sono i microservizi?* URL: https://aws.amazon.com/it/microservices/ (visited on 2024) (cit. on p. 5).

[4] IBM. *Cosa sono i broker di messaggi?* URL: https://www.ibm.com/it-it/topics/message-brokers (visited on 2024) (cit. on p. 7).

[5] Kafka. *Introduction.* URL: https://kafka.apache.org/intro (visited on 2024) (cit. on p. 7).

[6] Keycloak. *Server Administration Guide.* URL: https://www.keycloak.org/docs/latest/server_admin/ (visited on 2024) (cit. on p. 10).

[7] CyberSecurity360. *OpenID Connect: cos'è, a cosa serve e perché è importante.* URL: https://www.cybersecurity360.it/soluzioni-aziendali/openid-connect-cose-a-cosa-serve-e-perche-e-importante/ (visited on 2024) (cit. on p. 11).

[8] Keycloak. *Securing Applications and Services Guide.* URL: https://www.keycloak.org/docs/latest/securing_apps/ (visited on 2024) (cit. on p. 12).

[9] T. Lodderstedt, SPRIND, J. Bradley, Yubico, A. Labunets, Independent Researcher, and Authlete D. Fett. *OAuth 2.0 Security Best Current Practice.* URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-29#name-implicit-grant (visited on 2024) (cit. on p. 12).

[10] Docker Inc. *Docker overview.* URL: https://docs.docker.com/get-started/overview/ (visited on 2024) (cit. on p. 13).

[11] IBM. *Cosa sono i contenitori?* URL: https://www.ibm.com/it-it/topics/containers (visited on 2024) (cit. on p. 13).

[12] Giulio Turetta. *Cosa sono i Web service*. URL: https://www.html.it/pag/16448/cosa-sono-i-web-service/ (visited on 2024) (cit. on p. 17).

[13] Francesco Camarlinghi. *Il pattern MVC*. URL: https://www.html.it/pag/18299/il-pattern-mvc/ (visited on 2024) (cit. on p. 18).

[14] Microsoft. *Creare oggetti di trasferimento dati (DTO)*. URL: https://learn.microsoft.com/it-it/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5 (visited on 2024) (cit. on p. 20).

[15] Microsoft. *Panoramica del controller ASP.NET MVC*. URL: https://learn.microsoft.com/it-it/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs (visited on 2024) (cit. on p. 22).

[16] The Postman Team. *What are HTTP methods?* URL: https://blog.postman.com/what-are-http-methods/ (visited on 2024) (cit. on p. 23).

[17] My tech ramblings. *Back to .NET basics: How to properly use HttpClient*. URL: https://www.mytechramblings.com/posts/dotnet-httpclient-basic-usage-scenarios/ (visited on 2024) (cit. on p. 34).

[18] Microsoft. *DbContext Classe*. URL: https://learn.microsoft.com/it-it/dotnet/api/microsoft.entityframeworkcore.dbcontext?view=efcore-8.0 (visited on 2024) (cit. on p. 41).

[19] IBM. *Che cos'è PostgreSQL?* URL: https://www.ibm.com/it-it/topics/postgresql (visited on 2024) (cit. on p. 47).

[20] Tomasz Pęczek. *Server-Sent Events (SSE) support for ASP.NET Core*. URL: https://www.tpeczek.com/2017/02/server-sent-events-sse-support-for.html (visited on 2024) (cit. on p. 53).

[21] Mozilla Developer Network Web Docs. *Cache-Control*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control (visited on 2024) (cit. on p. 53).

[22] Microsoft. *Background tasks with hosted services in ASP.NET Core*. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-8.0&tabs=visual-studio (visited on 2024) (cit. on p. 57).

[23] Red Hat. *Cos'è e a cosa serve un gateway API?* URL: https://www.redhat.com/it/topics/api/what-does-an-api-gateway-do (visited on 2024) (cit. on p. 61).

[24] Dijin Augustine. *Ocelot API Gateway*. URL: https://medium.com/@dijin123/ocelot-api-gateway-31cc1430b530 (visited on 2024) (cit. on p. 61).

Ringraziamenti

Voglio terminare questo progetto di tesi con un breve ringraziamento a tutte le persone che mi hanno aiutato a raggiungere questo obiettivo. Prima di tutto volevo ringraziare il mio relatore, il Professore Giovanni Malnati, per avermi permesso di svolgere questa tesi che ha portato ad aumentare la mia passione e il mio bagaglio culturale sulla materia non sempre con facilità, ma con dedizione ed impegno. Un grazie anche al Dottorando Gabriele Scaffidi Militone, che è sempre stato disponibile ad aiutarmi in casi di difficoltà portandomi in primis a ragionare sempre sui motivi di quello che stavo facendo. Un grazie a tutte le persone che ho incontrato in ufficio a Torino e dell'azienda, nonostante non sia andato molte volte in presenza, ma che comunque mi hanno accolto e fatto sentire parte del gruppo. Grazie a tutti i professori che ho avuto in questi anni di università che mi hanno permesso di imparare delle nozioni importanti e che mi serviranno nella vita. Grazie a tutti i miei compagni di corso con cui ho stretto nuove amicizie, condiviso questo percorso e con cui ho imparato a lavorare in team. Grazie ai miei due coinquilini con cui ho condiviso le giornate e le serate durante le varie settimane e con cui sono riuscito a staccare la spina dallo studio. Un immenso e sentito grazie, soprattutto, ai miei genitori, che mi hanno permesso di fare questa esperienza formativa e che mi hanno supportato ad andare avanti, anche nei momenti di maggiore difficoltà. Un grazie sentito a tutti i miei amici di una vita, con cui sono riuscito a svagarmi e che mi hanno ascoltato quando ne avevo bisogno. Un grazie generale a tutti quelli che mi hanno accompagnato in questo mio percorso, è anche grazie a tutti voi se sono riuscito ad arrivare al suo completamento, attendendo le sfide che mi riserverà il futuro.