SVILUPPO DI UN BOT DISCORD PER COLLEZIONARE DATI DAI SITI DI SCOMMESSE

di Claudio Sicari e Alessandro Bellia

Distributed System and Big Data course final essay

A.A. 2021-2022

INTRODUZIONE

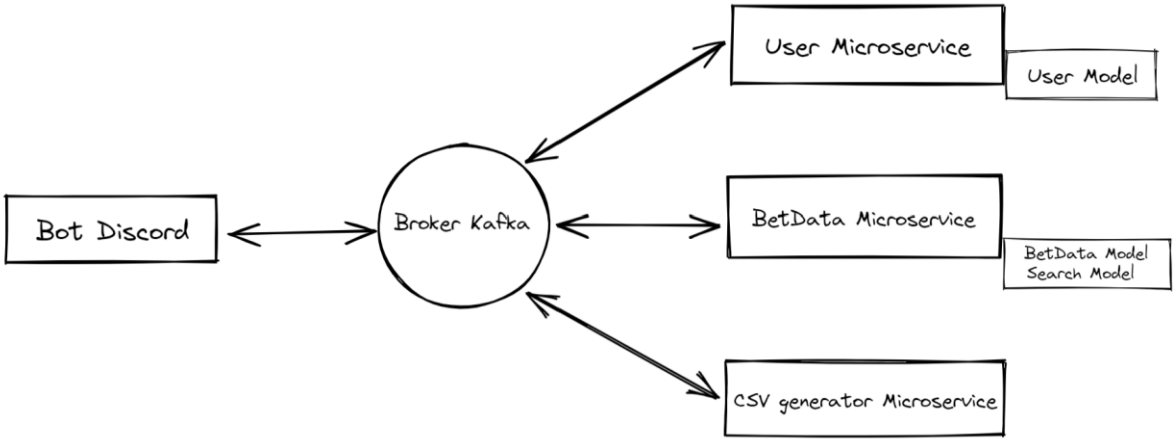
Discord è una piattaforma di messagistica istantanea e VoIP in cui gli utenti possono avviare videochiamate o singolarmente (tra loro) o all'interno di un canale, o server. Un server Discord divide i propri canali tra quelli vocali e quelli testuali, ovvero di sola messagistica, e non è possibile trovarvi all'interno solo utenti, ma anche uno o più bot. È possibile creare un bot attraverso il *Discord Developer Portal* (https://discord.com/developers), in cui è possibile registrare i propri bot e ottenere il token univoco da inserire nel proprio progetto per espanderne e definirne lo scopo e l'utilizzo.

L'elaborato di fine corso prevede lo sviluppo proprio di un bot che ha il compito di collegarsi ad un sito web di scommesse e collezionare i dati e le quote richieste, metterle dentro un file CSV e ritornarlo all'utente. Quest'ultimo, digitando un comando in chat, decide quale azione deve intraprendere il bot all'interno del suo raggio d'azione. Infatti, al momento della creazione di un bot, vanno specificati i permessi di cui esso gode: banalmente può sia fungere da amministratore del server (con politiche di creazione/rimozione utenti, contenuti, ecc.) sia da "solo lettore".

I comandi del Bot devono sempre iniziare con un carattere speciale (nel nostro caso "!") seguito dal tipo di comando e da ulteriori parametri, opzionali o meno. Ecco una lista dei comandi possibili per il bot in questione:

- *!help* : comando di default del bot, mostra la lista dei comandi eseguibili da quell'utente
- *!goldbet <nome_categoria>* : i dati vengono presi dalla categoria richiesta (serie A, serie B, Champions League, ecc.) nel sito Goldbet
- *!bwin <nome_categoria>* : analogo al precedente, tranne che per il sito che è appunto Bwin
- !*settings <comando> <utente> <parametro>* : sospensione dell'utente per un determinato periodo per l'utilizzo del bot o limitazione sul numero di richieste che può fare
- !*stat <numero>* : statistiche relative all'utilizzo del bot, i numeri vanno da 1 a 4 in base a ciò che si desidera verificare

ARCHITETTURA E OBIETTIVO



Sopra è mostrato lo schema che riassume l'architettura elaborata per lo sviluppo dell'applicazione. Il bot comunica con un broker, ovvero un intermediario nella comunicazione tra più entità, che a sua volta inoltra le varie richieste tra i microservizi. Questi ultimi comunicano tra di loro sempre e solo mediante questo broker e si coordinano per portare a termine quella che viene definita una transazione distribuita mediante pattern Saga. Procediamo per ordine: l'obiettivo di questa applicazione è mettere a disposizione di un utente delle quote appartenenti ad una determinata categoria del mondo calcistico. L'utente, specificando la categoria (Serie A, Premier League, Bundesliga, ecc.), ottiene delle quote calcistiche: quest'ultime vengono prese da due siti web, quali Goldbet e Bwin. Vengono proposti due siti web per dare una maggiore libertà al fruitore del bot, in quanto potrebbe voler fare un paragone e vedere dove conviene puntare. Per chi utilizza assiduamente Discord, l'utilizzo di questo bot può evitare al fruitore di dover cercare in qualunque sito web di scommesse quale sia quello più conveniente: il bot, in maniera diretta, offre all'utente questa possibilità.

Procedendo per ordine, ecco le specifiche implementate nel progetto:

- API Gateway
- Saga pattern
- Utilizzo di REST API
- Strategia di deployment mediante Docker
- Gestione cluster mediante Kubernetes
- Utilizzo di broker Kafka
- White-box monitoring mediante Prometheus

RUOLO DEI MICROSERVIZI

I microservizi sono quattro, come dalla figura precedente.

Il primo microservizio spiegato è lo *User Microservice*. Quando un utente effettua una ricerca il primo microservizio a venire interpellato è proprio questo: esso crea l'utente se non v'è traccia di esso all'interno del database, altrimenti verifica se sono poste limitazioni su di esso. In particolare, viene verificato se l'utente non è stato bannato o se non vi è un limite sul numero di ricerche che esso può effettuare. Viene creata una entry, nella tabella User, contenente il nickname e l'id dell'utente e, di default, il periodo di ban (posto a null) e il numero massimo di ricerche effettuabili (inizialmente -1). L'amministratore del bot può decidere se sospendere un utente per un periodo di tempo o può settare il numero di ricerche massime che esso può fare. Questo per evitare che utenti indesiderati usino il bot o ne facciano un uso in proprio, impedendone il corretto funzionamento. Le richieste http effettuate dal microservizio riguardano due statistiche: il numero di utenti che ha utilizzato il bot (numero e nickname delle persone) e il numero di ricerche effettuate da ogni singolo utente.

Il secondo microservizio che viene analizzato è il *BetData Microservice*. Il suo compito è quello di registrare ogni ricerca effettuata dagli utenti e i dati relativi ad essa. Cioè, si occupa di creare una entry, nella tabella relativa alle ricerche, contenente l'id della ricerca, il sito web in cui è stata svolta, l'id dell'utente che l'ha richiesta e l'url del file csv (campo inizialmente vuoto). Una volta aggiunta questa entry, ne viene aggiunta un'altra nella tabella relativa ai BetData che contiene i dati della ricerca associati all'id della ricerca stessa. I dati estrapolati per ogni partita sono: il giorno in cui questa avviene, le squadre avversarie, il moltiplicatore in caso di vittoria, pareggi o sconfitta per la squadra di casa e la quantità di gol previsti in generale nel match (quindi over 2,5 significa almeno 3 gol). Questo è ciò che fa il microservizio. In aggiunta, come REST API implementate si hanno due statistiche: il numero di ricerche effettuate dagli utenti e il numero di ricerche per sito web.

Il terzo microservizio che verrà spiegato è il *CSV generator Microservice*. Questo microservizio utilizza i dati collezionati dal bot per generare un file CSV che viene passato all'utente. Discord poi genera un url che conserva nella CDN (Content Delivery Network), che viene poi preso ed inviato al BetData Microservice che aggiorna il corrispondente campo nella tabella relativa alla Search. Viene, cioè, generato il file CSV, posto sulla chat di Discord, da essa viene preso l'url associato al file e passato al BetData Microservice, che lo pone nel correlato campo della tabella Search relativo alla ricerca. Nessuna REST API è stata implementata per questo microservizio.

Il quarto e ultimo microservizio è il *Bot Discord*. Esso si occupa di ricevere le richieste da parte di Discord e, in base alla tipologia di esse, svolge un determinato compito. La granularità del microservizio è più ampia rispetto ai precedenti. Cioè, il compito svolto dal bot è sempre uno: gestire i comandi che riceve dall'utente. Tuttavia, i comandi sono molteplici e ciò significa che il bot si ritrova a dover svolgere più casi d'uso: il principale è la collezione di dati dai siti web mediante i comandi !goldbet o !bwin, il secondo è la sospensione di un utente mediante il comando !ban e l'ultimo caso d'uso è il calcolo delle statistiche mediante il comando !stat. Ripetendo, lo spettro d'azione si limita a gestire i comandi, che sono per l'appunto molteplici.

PATTERN UTILIZZATI

Il primo pattern utilizzato è quello dell'*API Gateway*. Esso non viene direttamente implementato nel progetto per mano degli sviluppatori, ma viene indirettamente attribuito alla piattaforma di Discord. Il funzionamento è il seguente: l'utente immette una stringa nella chat e Discord, rendendosi conto che si tratta di un comando specifico per un bot, inoltra ad esso la richiesta. Quindi, qualsiasi comando che un utente inserisca viene riconosciuto da Discord come appartenente al campo operativo del bot e del sistema distribuito, per estensione. Quindi la piattaforma Discord, indirettamente, distribuisce le varie richieste dell'utente, nel nostro caso, al bot, che quindi funge da client dell'API Discord.

Il secondo pattern utilizzato è il pattern *Database Per Service*. In poche parole, in un sistema distribuito, per garantire la scalabilità, si evita di utilizzare un database centralizzato con cui ogni microservizio interagisce. Infatti, per fare in modo che ogni microservizio scali in maniera indipendente e ci sia un basso accoppiamento tra tutti i microservizi nel sistema, è preferibile che ogni servizio immagazzini i dati in un proprio ed unico database. Cioè, ogni microservizio opera su database indipendenti, la cui cooperazione è richiesta solo in caso di operazioni SQL-like, come le "join". Nel nostro progetto, ad esempio, il microservizio *User* ha un proprio DB relativo agli utenti in cui effettua operazioni di lettura e scrittura. Se il database fosse stato centrale, ogni volta questo microservizio doveva richiede l'accesso atomico al DB e procedere con l'aggiornamento dei dati. In termini di scalabilità ciò non poteva funzionare. Il microservizio *BetData* ha un proprio DB locale in cui sono presenti le Collection relative ai dati delle scommesse e alle ricerche effettuate dagli utenti. In sostanza, lo User Microservice lavora solo sulla Collection User e il BetData Microservice lavora solo sulle Collection relative alle ricerche e alle quote.

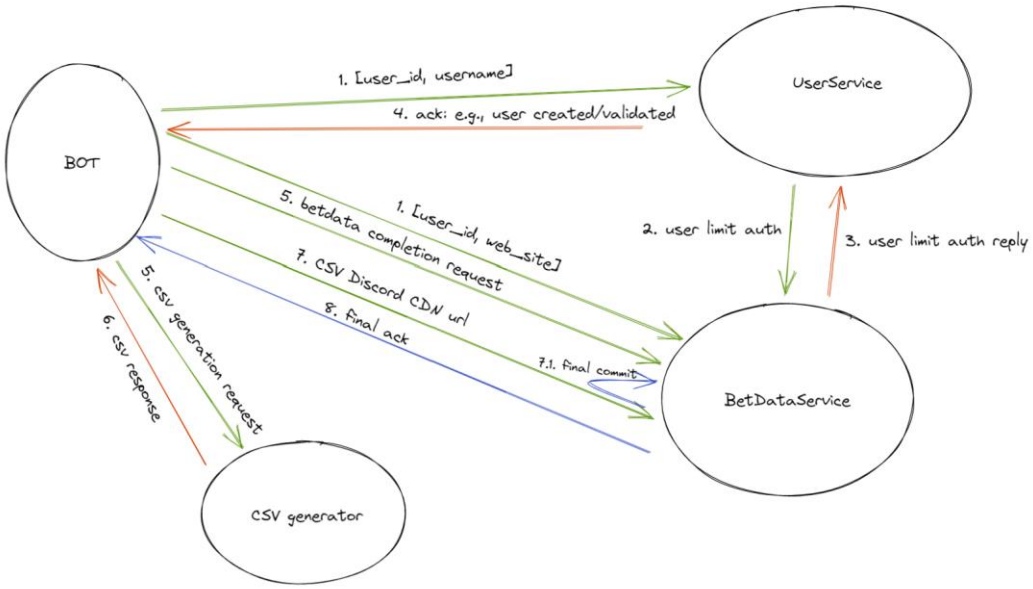
Il terzo pattern che viene analizzato è il pattern *Saga*, nello specifico nel progetto è stato implementato un ibrido: un'Orchestration in cui lo stato viene salvato localmente nei microservizi, come fosse una Choreography. Osservando la figura è più semplice comprendere:

- Il bot funge da Orchestrator, cioè una figura principale che se va in failure non permette lo svolgimento della transazione
- I vari microservizi registrano l'esito della propria transazione localmente, nel proprio DB

Questo significa che lo stato viene salvato nel DB locale del microservizio e qualora avvenga una rollback non sussistono problemi. È stata scelta questa soluzione per due motivi:

- Limiti del pattern db per service, in quanto ogni microservizio possiede un proprio db personale e non ve ne è uno centralizzato.

- Poiché la transazione globalmente ha 20 secondi di tempo per venire eseguita, se all'interno di questo lasso di tempo il bot crasha e viene riavviato la transazione continua da dove era stata interrotta, grazie al salvataggio locale dello stato da parte dei microservizi.



Da questo schema emerge l'ordine di esecuzione della transazione nella Saga:

- Il primo punto prevede l'invio asincrono dello user_id e dello username allo UserService affinché venga controllato che l'utente esista (viceversa viene creato) e che sia abilitato a poter procedere con la richiesta (no ban/limitazioni pendenti). Contestualmente, su un altro canale vengono inviati al BetData Service lo user_id e il web_site per inizializzare la entry da inserire nella collection Search.
- Il secondo e il terzo punto sono correlati al primo, in quanto viene verificato se l'utente può eseguire la ricerca e non ha, quindi, limitazioni o ban in corso. Per fare ciò è necessario contare le ricerche fatte dall'utente e questo dato prevede la comunicazione tra User e BetData Service.
- Se la validazione avviene con successo (quindi lo user è abilitato/stato creato) viene inviato un ack al bot.
- Al quinto punto i bet data vengono inviati al BetData Service (per la creazione dei corrispondenti campi per mano del microservizio nella Collection apposita) e al CSV generator microservice. Nell'ultimo caso si procede alla creazione del file csv.
- Al sesto punto il file CSV, restituito al bot, viene salvato nella CDN di Discord.
- Al settimo punto l'url salvato nella CDN di Discord viene inviato al BetData Service, che aggiorna la entry della Search rimasta parzialmente incompleta dal punto 1 (il campo "url_csv" è rimasto vuoto)
- Infine, se tutto ciò ha successo avviene la commit finale.

IMPIEGO DEL BROKER KAFKA

Apache Kafka è un broker utilizzato nei sistemi distribuiti per due motivi principali: rendere trasparente e asincrona la comunicazione tra più microservizi all'interno di un sistema distribuito. Un server funge da broker, mentre un client può essere sia produttore che consumatore. Infatti, nel progetto un microservizio può produrre dati che altri microservizi consumeranno e viceversa. Tali dati vengono appesi a dei topic sottoforma di record, o messaggi. I topic utilizzati nel progetto sono 10: uno per ogni richiesta/risposta all'interno della transazione.

Per ogni step della transazione vengono appesi messaggi dinamicamente per ogni topic. I topic possono essere frammentati in base al numero di repliche, grazie al load balancing. Siccome nel nostro progetto ci sono due repliche per microservizio, il numero di partizioni per topic è pari a due e questo garantisce concorrenza.

DOCKER E KUBERNETES

Per quanto riguarda l'implementazione di Docker, nel progetto sono stati creati quattro Dockerfile: uno per ogni microservizio. L'immagine di base per tutti i microservizi è unica ed *Python3.10.1-alpine*. Quindi è stata scelta la versione più recente di Python abbinata ad un'immagine piuttosto leggera (5 MB circa). Ovviamente una causa di ciò, ovvero di un'immagine light, è che tutte le dipendenze sono state installate manualmente in seguito. Inoltre, sono state settate delle variabili d'ambiente (sotto la dicitura ENV) che fanno riferimento una al package del microservizio (PYTHONPATH), mentre l'altra setta la modalità di scrittura dei log ad unbuffered (PYTHONUNBUFFERED) e questa seconda cosa viene fatta in quanto, se dovesse crashare l'applicazione, non si perderebbero eventuali messaggi. Successivamente vengono installate le dipendenze obbligatorie che si trovano in parte dentro il file *requirenments.txt* e altre sono globali (come quelle di Kafka). L'ultimo comando (CMD) serve per runnare l'application server in loop.

In totale il progetto consta di 12 container:

- 4 relativi ai microservizi implementati
- 2 relativi ai database presenti nei servizi
- 2 relativi alle interfacce grafiche dei database
- 1 per ZooKeeper
- 1 per lo Schema Registry, ovvero un microservizio consultato dai produttori e consumatori Kafka in fase di serializzazione o deserializzazione dei dati
- 1 per il broker Kafka
- 1 per il Control Center, cioè una dashboard che gestisce i prodotti Confluent (Kafka stesso)

Il file *docker-compose.yaml* permette di gestire e definire il cluster di container deployati. Vengono dichiarati dapprima i servizi (sotto la dicitura *services*) e conseguentemente la configurazione per ognuno di essi. Il numero di servizi è pari al numero di container. Il campo *image* contiene il nome della repository da cui prendere l'immagine. Il campo *ports* specifica le porte dell'host e del container da esporre. Il campo *volumes* indica la mappatura tra la directory dell'host e quella remota del container per gestire la persistenza del dato. Infatti, questa dicitura appartiene solo ai servizi relativi ai database e a Kafka. In altri servizi vengono specificate le dipendenze, ovvero di quali microservizi ha bisogno un altro servizio prima di avviarsi.

Per quanto riguarda Kubernetes, il modello preso come riferimento è il *One Container Per Pod*. In questo modo ogni Pod conterrà esattamente un container e questo porta, come vantaggi, un isolamento netto in termini di gestione delle risorse da parte del Pod stesso. Inoltre, nel campo *kind* viene settato a "Deployment" e questo permette di poter gestire più di una replica (infatti il campo è settato a 2, cioè 2 pod con dentro un container). Il file *Discord.yaml* è di tipo *kind=Secret*, ovvero è i file contenuti sono catalogati come confidenziali, quindi ad esempio le credenziali del database o il token per poter avviare il bot da Discord.

Le custom resource, installate tramite Helm, riguardano l'installazione dell'operatore Confluent per gestire i Pod relativi a Kafka. Infatti, nel file *confluent-platform.yaml* sono elencate tutte le risorse prima descritte.

Ogni microservizio ha due repliche di se stesso, eccetto il bot. I due database hanno ciascuno due repliche, per un totale di quattro. Anche l'interfaccia grafica di MongoDB possiede una replica. Sono stati creati appositamente dei file .bat per eseguire l'avvio dei Pod in maniera più semplice.

PROMETHEUS

Prometheus viene utilizzato nel progetto per il white-box monitoring, ovvero il monitoraggio delle richieste effettuate al bot. In generale, utilizziamo questa piattaforma per simulare, con un numero di campioni arbitrario, le richieste effettuate al bot, in particolare al Bet Data Microservice. Questo lavoro viene fatto per testare il load balancing tra i vari Pod (ovvero le varie repliche) dello stesso microservizio e/o per predire l'andamento delle richieste nel futuro. Ovvero viene simulato il numero di richieste, attraverso una predizione, che graveranno sul microservizio al fine di testarne il corretto funzionamento sia dal punto di vista del bilanciamento del carico di lavoro che della scalabilità (che viene valutata anche con il black-box monitoring).

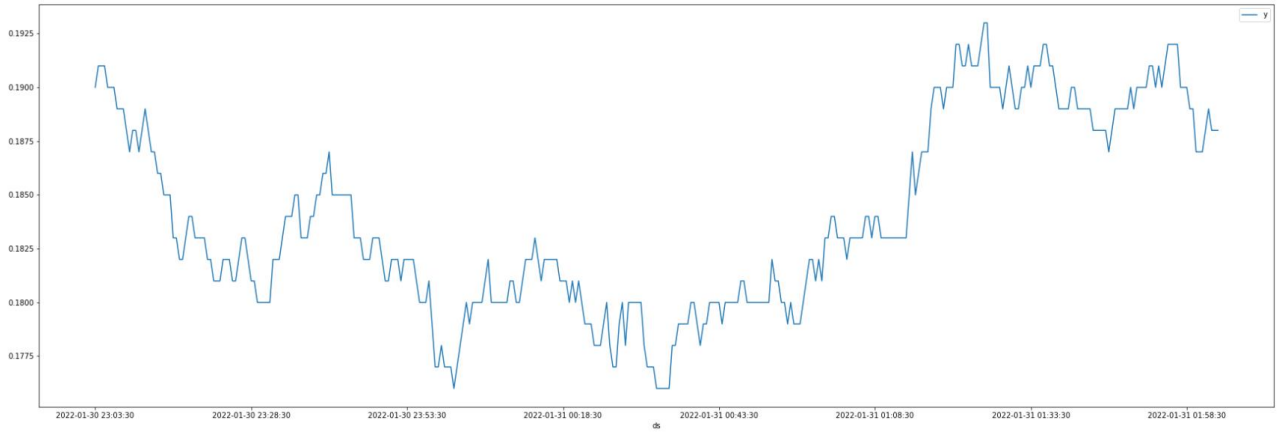
È stato utilizzato un predittore chiamato *PROPHET*, di Facebook. È stata fatta questa scelta in quanto la serie temporale da analizzare è caratterizzata da:

- Forte stagionalità (anche multipla), ovvero i cambi di trend della serie temporale sono ben visibili e ricorrono con una certa frequenza.
- Possibili valori isolati, cioè vi sono punti del grafico che si distaccano molto dal trend e per questo vengono detti isolati. Una serie di valori isolati da luogo a quello che si chiama *effetto holiday*, in cui sono raggruppati una serie di valori non affini al trend.

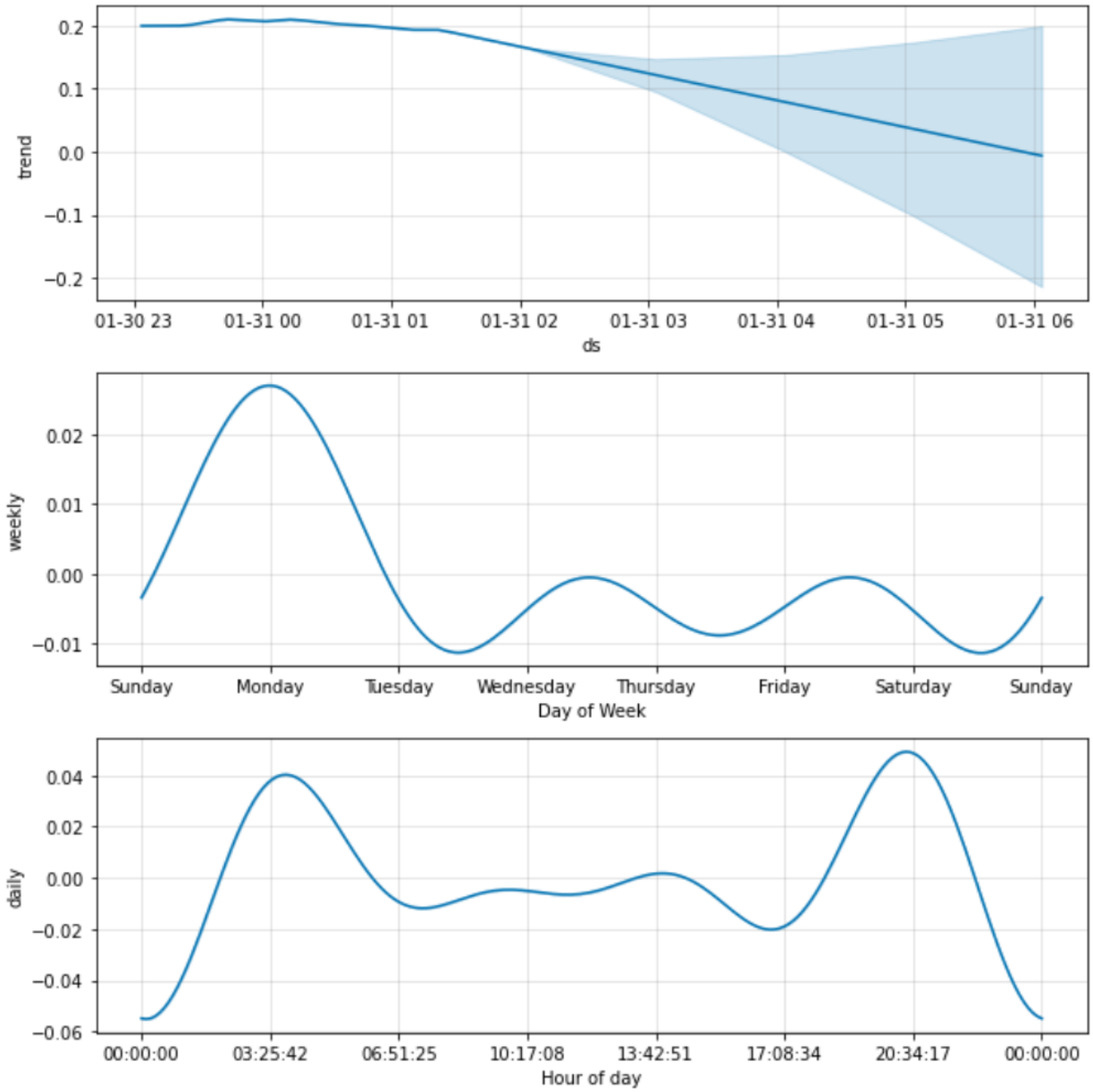
È stato scelto questo modello di predittore in quanto gli altri visti a lezione richiedono un livello di esperienza elevato per poter ottimizzare i parametri usati per la predizione. Inoltre, essi non sono scalabili, in quanto richiedono molte più risorse rispetto a Prophet, che risulta quindi essere più efficiente (lasciamo il link in merito al paper consultato: https://peerj.com/preprints/3190/).

Prophet trasforma un problema di analisi delle serie temporali in un semplice problema di regressione lineare a tratti. Questo permette di analizzare con più facilità i dati ottenuti dalla predizione. Lineare a tratti perché, come già detto, i trend hanno forte stagionalità; quindi, sono spesso soggetti a cambi di trend.

La stagionalità, nel progetto, viene implementata come componente additiva. Questo è lo stesso approccio utilizzato dall'*exponential smoothing*, implementato anche dal predittore Holt Winters.



Questo grafico mostra il rate di richieste per unità di tempo.



Per questo set di grafici sopra, invece, il primo mostra l'andamento della serie temporale, ovvero il trend. La serie storica parte dalle 23 e finisce alle 2 di notte, mentre il predittore stima l'andamento a partire dalle 2 di notte fino alle 6. Il secondo grafico mostra la stagionalità settimanale, ovvero come si comporta la serie temporale nell'arco di una settimana. Il picco è visibile solo di Lunedì in quanto i dati raccolti fanno riferimento solo a quattro ore. Invece, il terzo grafico, che mostra la stagionalità giornaliera, rivela che i picchi (ovvero il maggior numero di richieste al microservizio) avvengono intorno alle 20:30 e intorno alle 3:25 di notte.