PeakLand

Sistemi Distribuiti e Cloud Computing 2021/2022

Mattia Antonangeli
Mat. 0311183
Facoltà di Ingegneria Informatica
Università degli studi di Roma Tor Vergata.

Federica Villani
Mat. 0309716
Facoltà di Ingegneria Informatica
Università degli studi di Roma Tor Vergata.

Abstract

Questo documento ha l'obiettivo di descrivere l'architettura dell'applicazione *PeakLand* e le fasi di progettazione e implementazione che hanno portato al suo sviluppo.

1 Descrizione dell'applicazione

PeakLand è un'applicazione web creata per gli appassionati di sentieri di montagna e trekking. Essa, infatti, offre all'utente la possibilità di ricercare nuovi sentieri da percorrere, sia attraverso il nome che tramite l'inserimento di filtri (per esempio *livello di difficoltà*, *regione*, *adatto a famiglie* ecc..), e di visualizzarne le informazioni (*altitudine*, *lunghezza*, *località* ecc..), le recensioni inserite dagli altri utenti e le previsioni meteo. L'utente può, inoltre, creare un proprio account per poter usufruire di servizi aggiuntivi: inserire nuovi percorsi all'interno dell'applicazione, effettuare recensioni di sentieri percorsi, pianificare visite ad un sentiero scelto, invitare amici registrati nell'applicazione a tali visite e accettare o rifiutare gli inviti.

2 Struttura dell'applicazione

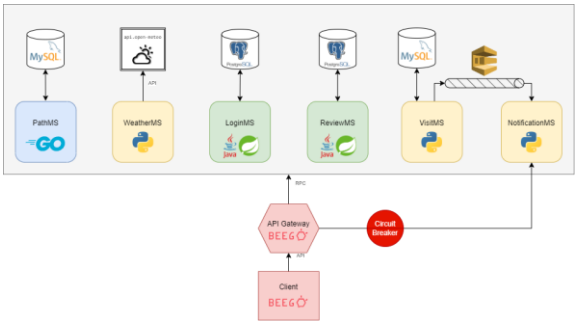
Come da specifiche, l'applicazione presenta un'architettura basata su microservizi, ovvero su diversi componenti indipendenti tra loro, ciascuno dei quali offre una specifica funzionalità all'utente e comunica con il resto del sistema attraverso un'interfaccia ben definita. Questo stile architetturale è vantaggioso rispetto ad una applicazione monolitica perché ciascun servizio può essere aggiornato, distribuito e ridimensionato per rispondere alla richiesta di funzioni specifiche dell'applicazione.

Per *PeakLand*, in particolare, sono stati individuati i seguenti microservizi:

- *Login:* che offre le funzionalità di login e signin e tiene traccia degli utenti correntemente loggati nell'applicazione.
- *PathManager:* che gestisce i sentieri presenti nell'applicazione e permette di eseguire ricerche e aggiungere nuovi percorsi.
- *ReviewManager*: che gestisce le recensioni associate ai sentieri, permettendo di visualizzare quelle già presenti e di aggiungerne di nuove.
- *WeatherForecast:* che permette di recuperare le previsioni meteo relative ad un particolare percorso
- *VisitManager:* che gestisce le visite programmate e permette di mandare gli inviti agli altri utenti.
- *NotificationManager:* che si occupa di notificare gli utenti che hanno ricevuto un invito.

Oltre a questi componenti, che costituiscono il cuore dell'applicazione, ci sono il *client*, che si occupa di comunicare con l'utente e di gestire l'interfaccia grafica, e il servizio *API-gateway*, che svolge il ruolo di orchestratore, facendo da tramite tra le richieste del client e i microservizi. Quest'ultimo componente fornisce un unico punto di accesso all'applicazione, facendo in modo che il client non debba conoscere i microservizi per soddisfare le proprie richieste, garantendo un disaccoppiamento spaziale.

Lo schema seguente descrive la struttura a microservizi dell'applicazione e fornisce una visione generale sulle tipologie di comunicazione, sui design pattern e su dettagli implementativi utilizzati, che verranno discussi in seguito:



3 Scelte progettuali

Comunicazione

La comunicazione tra il client e *l'API Gateway* è di tipo *RESTful,* mentre la comunicazione tra *l'API gateway* e i microservizi avviene adottando lo schema RPC, ovvero chiamata a procedura remota. Per quanto riguarda, invece, la comunicazione tra il microservizio *VisitManager* e il microservizio *NotificationManager* si è scelto di inserire una coda di messaggi, che memorizzi gli inviti alle visite effettuate dagli utenti e creati nel microservizio di gestione delle visite, i quali, poi, saranno ricevuti dal microservizio di gestione delle notifiche quando richiesti. Per ottenere le notifiche che competono all'utente autenticato, il client esegue un polling nei confronti dell'*API gateway*, il quale inoltra la richiesta al microservizio *NotificationManager*. Questo tipo di comunicazione permette di avere un disaccoppiamento spaziale, temporale e rispetto alla sincronizzazione.

Design pattern

I design pattern che sono stati applicati nell'applicazione in esame sono il *circuit breaker* e il *database-per-service*.

Il pattern *circuit breaker* è applicato tra l'*API gateway* e il microservizio di gestione delle notifiche ed esso permette di evitare che l'API gateway continui ad inoltrare le richieste provenienti dal polling del client verso il microservizio *NotificationManager* se questo è in down.

Il pattern *database-per-service* consiste nel definire un database differente per ogni servizio dell'applicazione, in modo da mantenere privati i dati e avere un basso accoppiamento tra i microservizi. In particolare, sono stati creati dei database privati per tutti i microservizi tranne *NotificationManager* e *WeatherForecast,* in quanto non necessitano di avere dati da mandare in persistenza.

Microservizi stateless e stateful

Escludendo il *LoginService*, tutti i microservizi sono di tipo *stateless,* ovvero il loro database è esterno al microservizio stesso, in modo da garantire una maggiore scalabilità del sistema. Per quanto riguarda il *LoginService,* invece, esso, oltre ad avere un database per memorizzare le credenziali degli utenti, mantiene all'interno di una struttura dati, presente nel microservizio stesso, la lista degli utenti che hanno effettuato il login nell'applicazione, al fine di mantenerne una sessione attiva senza il bisogno di effettuare nuovamente il login mentre si naviga tra diverse funzionalità. Questo conduce, nel caso in cui il servizio venga replicato per garantire tolleranza ai guasti e scalabilità, al partizionamento dello stato del microservizio tra le repliche. Tuttavia, nello specifico caso, questo non comporta un grande problema poiché, nel caso peggiore in cui un utente venga servito in una stessa sessione da repliche diverse, dovrà semplicemente ripetere la procedura di login.

Deployment

Per il deployment dell'applicazione, ogni microservizio, ogni database, il *client* e l'API G*ateway* sono stati incapsulati all'interno di container opportunatamente configurati e orchestrati da *Docker Compose*, quindi eseguiti tutti in locale.

4 Descrizione implementazione

Client

Il client dell'applicazione, che si occupa di gestire l'interfaccia grafica e di comunicare con l'API *gateway* per soddisfare le richieste dell'utente, è stato sviluppato nel linguaggio *Go,* utilizzando il framework *RESTful HTTP* open source *BeeGo,* il quale permette di sviluppare in Go API REST, applicazioni web e servizi backend. BeeGo è stato creato prendendo spunto da diversi framework che troviamo in altri linguaggi, come *Flask e Tornado*. L'interfaccia grafica è stata creata con HTML e CSS.

Un client BeeGo ha una struttura delle directory ben precisa:

- *Conf*: contiene i file di configurazione del server, come l'indirizzo della porta di deploy e la modalità di avvio (prod, dev, test);
- *Model*: contiene le definizioni delle struct utilizzate nell'applicazione;
- *Routers*: contiene le informazioni per eseguire il routing tra un URL e il relativo controller;
- *Controller*: contiene i controller che gestiscono le chiamate REST API eseguite dall'utente;
- *Static*: contiene i file statici usati per l'interfaccia grafica;
- *Views*: contiene i file HTML usati per l'interfaccia grafica;

All'interno della sessione di un utente, vengono mantenuti alcuni parametri, tra cui il più importante è il cookie che identifica l'utente una volta loggato. Questo viene inviato all'API gateway ad ogni richiesta in modo che, se l'utente sta cercando di accedere ad una funzionalità che necessita di autenticazione, possa essere chiamato il microservizio di Login per controllare se l'utente è correntemente autenticato all'interno del sistema. Nel caso in cui l'utente risulta non autenticato, il client verrà redirezionato sulla pagina di login, in modo che l'utente possa effettuare la procedura di login.

Dopo il login, il client effettua un redirect alla pagina a cui l'utente aveva tentato di accedere in precedenza. Inoltre, nel momento in cui l'utente completa la procedura di login o

registrazione, il client avvia una *goRoutine* che effettua il polling verso l'*API gateway* al fine di ricevere le notifiche di inviti diretti all'utente. In particolare, viene inviata una richiesta di ricezione di inviti ogni dieci secondi. Gli inviti ricevuti vengono poi salvati come attributo di sessione e recuperati nel momento in cui l'utente decide di visualizzare la pagina delle notifiche.

La comunicazione con l'*API gateway* avviene anch'essa tramite chiamate HTTP e i parametri vengono passati tramite stringhe in formato json.

API gateway:

L'*API gateway*, come il client, è stato sviluppato in *Go* con l'utilizzo di *BeeGo*, quindi presenta la stessa struttura dei package precedentemente descritta. Tuttavia, a differenza del Client, l'app è stata costruita come "API Beego application", poiché essa non deve avere una interfaccia grafica. Riceve, quindi, le richieste dal client tramite API REST e le gestisce andando a richiamare i microservizi tramite chiamate *goRPC*, nel caso di servizi scritti in *Go*, oppure tramite *gRPC*, per microservizi sviluppati in altri linguaggi di programmazione, comportandosi da client RPC. In particolare, per gRPC sono stati creati dei file *.proto*, poi opportunamente compilati, che definiscono i servizi offerti dai microservizi e la struttura dei messaggi di input e output. È stato, inoltre, creato un file di configurazione che specifica i parametri di connessione per ogni microservizio, ovvero indirizzo dell'host e numero di porta, necessari per la comunicazione. Dovendo fare da tramite tra il *Client* e il resto dei servizi, l'*API gateway* riceve e manda stringhe in formato *json*, senza effettuare l'unmarshaling. Non è quindi necessario che conosca la struttura del model e non è, quindi, impattato da eventuali suoi cambiamenti.

Come accennato nella precedente sezione, nell'*API gateway* è implementato il pattern del *circuit breaker* per la comunicazione con il servizio *NotificationManager*. Per fare ciò è stata utilizzata la libreria di Sony chiamata *GoBreaker*, che permette di creare un oggetto *CircuitBreaker* specificando i seguenti parametri di configurazione: il massimo numero di richieste che possono essere eseguite quando il circuito è half-open, il periodo dopo il quale i contatori delle richieste vengono azzerati, l'intervallo di tempo dopo il quale il circuito passa da open a half-open e la funzione da eseguire ad ogni cambiamento di stato. Per lavorare sempre con la stessa istanza di *CircuitBreaker* anche tra diverse chiamate dell'*API gateway* è stato applicato un pattern di *singleton*.

Login

Il microservizio di Login è stato sviluppato in *Java* con l'utilizzo del framework *Spring* integrato con *Hibernate.* Questi frameworks permettono di facilitare di molto la comunicazione con il database in quanto, tramite la definizione di classi contenenti specifiche annotazioni e la definizione di interfacce estendenti *JpaRepository*, costruiscono in autonomia lo schema del database e generano le query, permettendo di evitare al programmatore molto codice boilerplate. In particolare, per questo microservizio è stato creato il database Postgres *loginservice* con le tabelle *UserCredential*, che mantiene le credenziali degli utenti, e *LoggedUser*, che memorizza le informazioni del profilo degli utenti. Il microservizio è strutturato nelle seguenti directory:

- *Model*: che contiene le entities, che saranno poi mappate automaticamente nel database da Spring;
- *Controller:* che contiene le classi che si occupano di implementare la logica del microservizio;
- *Dao:* che contiene le interfacce per la comunicazione con il database;
- *Grpc:* essendo sviluppato in Java, la comunicazione con l'*api gateway* avviene tramite gRPC. Questa directory contiene, quindi, i file compilati a partire dal .proto che rendono possibile tale comunicazione;

Come precedentemente accennato, questo microservizio è stateful, poiché mantiene all'interno di un attributo di tipo lista della classe *LoginController* i cookie associati ad utenti che sono correntemente loggati nell'applicazione, e che, quindi, non devono eseguire la procedura di login per un certo intervallo di tempo mentre utilizzano l'applicazione. I servizi offerti da questo microservizio sono:

- Login: controlla che le credenziali inserite dall'utente siano corrette e in caso affermativo aggiunge il cookie alla lista dei loggati.
- signIn: aggiunge le nuove credenziali al database e aggiunge il nuovo cookie alla lista dei loggati. Il cookie viene poi restituito all'*API gateway*, che a sua volta lo restituisce al *client* in modo che lo possa memorizzare nella sua sessione;
- logOut: rimuove il cookie dell'utente dalla lista dei loggati;
- checkLogin: controlla, dato il cookie associato all'utente, se è correntemente loggato nell'applicazione, andando ad accedere alla lista;
- checkUsername controlla se l'utente con lo username in input è presente all'interno del sistema;
- getProfile: restituisce le informazioni sul profilo dell'utente presenti nel database;

- updateProfile: permette di aggiornare il profilo dell'utente nel database;

PathManager

Il microservizio per la gestione dei sentieri è sviluppato in *Go* e utilizza il database MySQL *pathmanager* per la memorizzazione delle informazioni relative ai sentieri presenti all'interno dell'applicazione, salvandole nella tabella *Path.* Esso può, quindi, comunicare con l'*API gateway* utilizzando *goRPC*, comportandosi da server RPC. *Q*uindi, all'interno del file *main.go* vengono eseguite le operazioni di registrazione dei servizi offerti, di ascolto su una determinata porta configurabile e di accept. La struttura delle directory del microservizio è la seguente:

- *Conf:* contenente i file di configurazione per il servizio e la connessione al database
- *Model:* contenente le strutture di dati chiavi per i servizi, ovvero *MountainPath* e *AssistedSearchFilters*, per la ricerca assistita
- *Controller:* contenente i file che implementano la logica dei servizi offerti.

I servizi che offre questo microservizio sono la ricerca per nome (anche parziale) e per filtri di un determinato sentiero e l'aggiunta di un nuovo sentiero all'interno del database. Le prime due funzionalità sono disponibili per qualunque utente, mentre per l'ultima è necessario che l'utente sia autenticato.

ReviewManager

Anche questo microservizio, come il *Login*, è stato sviluppato in *Java* con i frameworks *Spring* ed *Hibernate*, pertanto presenta una struttura delle directory uguale a quella del *Login*. Per la persistenza dei dati è stato utilizzato un database Postgres con nome *reviewmanager* in cui è presente la tabella *Review*, che memorizza le recensioni inserite dagli utenti. Non essendo sviluppato in Go, anche per la comunicazione di questo microservizio con l'api gateway è necessario utilizzare gRPC. È stato, quindi, definito un file ".proto" che descrive i servizi messi a disposizione da *ReviewManager*. Questi, nello specifico sono la ricerca di tutte le recensioni relative ad un sentiero, disponibile anche per utenti non autenticati, e l'aggiunta di una nuova recensione, che necessita, invece, di autenticazione.

VisitManager

Il microservizio di gestione delle visite è stato realizzato in Python, e utilizza un database MySQL per mantenere in persistenza le visite organizzate dagli utenti e i partecipanti a tali visite, che costituiscono due tabelle distinte. In questo microservizio è stata utilizzata la libreria *Pandas*, una libreria di *Python* che permette di effettuare analisi e manipolazione dei dati. In particolare, essa viene utilizzata per la gestione delle date, poiché si è deciso di salvare nel database i timestamp delle visite in formato *JulianDay*. Non essendo implementato in *Go*, la comunicazione con l'*API gateway* avviene tramite gRPC. Inoltre, come accennato in precedenza, VisitManager comunica con il microservizio addetto alle notifiche tramite un sistema a code di messaggi in modo da inviare agli utenti una notifica nel caso in cui vengano invitati ad una visita. In particolare, la coda di messaggi utilizzata è quella offerta dal servizio *SQS* di *Amazon*. Per quanto riguarda le visite, il microservizio offre le seguenti funzionalità:

- GetAllVisits: restituisce una lista di tutte le visite relative ad un certo utente, e per ogni visita viene restituita anche una lista di tutti i partecipanti;
- GetVisitByID: restituisce una visita specifica e tutti i partecipanti ad essa in base a un ID;
- AddNewVisit: aggiunge una nuova visita nel sistema;

Invece, per quanto riguarda la gestione degli inviti, abbiamo le seguenti funzioni:

- InviteUserToVisit: permette di invitare un utente a una visita
- AcceptOrRefuseInvite: permette ad un utente di accettare o rifiutare un invito ad una visita

Questo microservizio, nel momento in cui viene effettuato un invito ad un utente per una visita, invia un messaggio alla coda SQS, in modo che, se richiesto, il microservizio delle notifiche possa ricevere il messaggio e inviare una notifica al client. Inoltre, nel momento in cui un invito viene accettato/rifiutato, *VisitManager* riceve il messaggio relativo a quell'invito in coda e lo elimina da essa. Altrimenti, fino a quando l'utente non compie un'azione sulla notifica, il messaggio, dopo un intervallo di tempo di invisibilità, ritorna visibile nella coda per poter essere riscaricato. Si è deciso di operare in questo modo sia per rendere il microservizio delle notifiche stateless, sia per fare in modo che, anche in caso di diverse connessioni dell'utente o in presenza di repliche del microservizio *Client*, non venga persa alcuna notifica.

NotificationManager

Il microservizio di gestione delle notifiche è stato implementato in *Python* e permette agli utenti di ricevere le notifiche relative agli inviti per le visite, effettuati da altri utenti. In particolare, esso implementa la funzione *receiveInviteRequestMessage* che, preso come parametro lo userId dell'utente, riceve messaggi relativi ad esso dalla coda *SQS* di cui sopra e ne restituisce il contenuto al richiedente. La coda ha una semantica di consegna di tipo

timeout-based: quando il messaggio viene ricevuto, esso viene reso invisibile in coda per cinque secondi, dopodiché, se prima non viene eliminato dalla coda, il messaggio torna ad essere visibile. Come detto prima, l'eliminazione del messaggio è a carico del microservizio *VisitManager*, dunque il microservizio di notifica riceve soltanto i messaggi per mandarli al client, senza eliminarli. La comunicazione con *l'API gateway* avviene tramite *gRPC*. Questo microservizio viene richiamato con periodicità poiché, come spiegato in precedenza, il *Client* effettua polling per ricevere le notifiche che gli competono.

WeatherForecast

Il microservizio *WeatherForecast*, implementato in Python, permette all'utente di ricevere informazioni relative alle previsioni del tempo di una data città. In particolare, questo microservizio, data la città in cui è presente il percorso, esegue due chiamate ad API pubbliche:

1. *geocoding-api.open-meteo.com*: recupera le coordinate geografiche del luogo definito dalla città passata come parametro
2. *api.open-meteo.com*: date le coordinate geografiche recuperate prima, permette di ricevere le previsioni del tempo. In particolare, vengono ricevute le previsioni del tempo per una settimana.

Le informazioni del tempo ricevute sono:

- Temperatura;
- Umidità;
- Precipitazioni;
- Nuvolosità;
- Velocità del vento;

5 Conclusioni

In questo progetto abbiamo sperimentato alcune delle metodologie (come i pattern di Circuit Breaker, Database per Service ed API Gateway) e tecnologie (come Amazon SQS e Docker) per la progettazione ed il deploy di applicazioni a microservizi.
Possiamo pensare che i microservizi che potrebbero essere più utilizzati sono quelli che offrono delle funzionalità che non necessitano di login, come il microservizio di PathManager usato per la ricerca di percorsi, il microservizio *ReviewManager* per la visualizzazione delle recensioni di un sentiero e *WeatherForecast*, per la visualizzazione delle previsioni meteo. Questo perché ci si aspetta che il numero di utenti che accederanno all'applicazione in modalità anonima sia nettamente superiore del numero di utenti che decideranno di creare un proprio account all'interno del sistema. Quindi, prendendo in considerazione quello che potrebbe essere il carico del sistema, questi microservizi avranno bisogno di più repliche per gestire le richieste degli utenti rispetto agli altri microservizi.