



Università di Pisa
Dipartimento di Informatica

Corso di Laurea Triennale in Informatica
(classe L-31)

Relazione di Tirocinio presso l'azienda Zerynth S.R.L.

Integrazione di una piattaforma di vehicle tracking esistente con l'architettura cloud di Zerynth

Candidato:
Andrea Tufo
Matricola 578138

Tutore Aziendale:
Davide Neri
Tutore Accademico:
Gabriele Mencagli

Anno Accademico 2021–2022

Indice

1	Introduzione	5
2	Background e strumenti utilizzati	7
2.1	API REST	7
2.2	Docker e docker-compose	8
2.2.1	Utilizzo di docker e docker-compose	8
2.3	Code rabbitMQ	8
3	Studio del sistema e analisi dei requisiti	11
3.1	Studio della piattaforma Traccar	11
3.1.1	Come funziona Traccar	11
3.1.2	Analisi statica del codice di traccar	12
3.1.3	Architettura di ZDM	13
3.1.4	Caratteristiche di Traccar	13
3.1.5	Problematiche emerse	14
3.2	Struttura di zCloud	14
3.3	Parallelismo Traccar e zCloud	14
3.4	Personas e requisiti	17
4	Integrazione del sistema	19
4.1	Configurazione di Traccar in NGNIX	20
4.2	Autenticazione dell'utente in Traccar	21
4.2.1	Utilizzo dell'account-id in Traccar	21
4.2.2	Considerazioni sulla soluzione	22
4.3	Gestione della comunicazione con i device	22
4.3.1	Autenticazione e autorizzazione dei device	23
4.3.2	Storage dei dati dei device	25
4.3.3	Recupero dei dati da zStorage	26
4.3.4	Considerazioni finali sulla gestione dei dati dei device	26
4.4	Creazione dei device e delle workspace	26
4.4.1	Modifica del microservizio wfd	27
4.4.2	Filtro dei device	31
4.4.3	Osservazioni sulle complicazioni emerse	31

5	Integration test	33
5.1	Test del login	33
5.2	Test del device	34
5.3	Test dello storage	35
5.4	Test dell'autenticazione del device	36
5.5	Ulteriori modalità di testing	36
6	Conclusione	39
6.1	Demo finale	39
6.2	Critiche e possibili miglioramenti	41
6.2.1	Modularità	41
6.2.2	Gestione delle entità	41
6.2.3	Duplicazione delle istanze	42
6.3	Analisi dei punti di forza del lavoro svolto	42
6.4	Ultime osservazioni sugli obiettivi raggiunti	43
6.5	Ringraziamenti	43

Capitolo 1

Introduzione

Questo progetto aveva come scopo principale quello di integrare un sistema già esistente per il tracciamento veicolare, ovvero Traccar, con la piattaforma zCloud sviluppata dall'azienda Zerynth s.r.l.

zCloud, è a tutti gli effetti una piattaforma cloud che permette agli utenti registrati all'interno del servizio, di aggiungere device, fleet e workspace. Il device fisico ha un *digital twin* sulla piattaforma, esso possiede degli attributi, tra i principali abbiamo un identificatore, un nome, un identificatore dell'account a cui appartiene e un identificatore relativo al fleet in cui è contenuto. Il fleet invece rappresenta logicamente un insieme di dispositivi registrati, e anch'esso possiede un identificativo ed un nome. Infine abbiamo le workspace, ovvero, degli spazi di lavoro in cui l'utente può decidere di aggiungere fleet e device.

Traccar invece, è un sistema di tracciamento di dispositivi, possiede un'interfaccia grafica dove è possibile osservare il movimento real-time del dispositivo, i suoi attributi, dei report generati e customizzabili e anche delle geofence ovvero delle aree limitate che l'utente può decidere di tracciare sulla mappa. Il sistema permette inoltre di ricevere delle notifiche per qualsiasi evento l'utente decida di registrarsi, come ad esempio dispositivo che entra in una geofence (o esce dalla geofence), dispositivo che inizia a muoversi o che si ferma o ancora batteria del device bassa. Anche Traccar possiede le proprie entità come ad esempio i device, i gruppi, ma a differenza di zCloud non possiede il concetto di workspace.

Un primo passo molto importante è stato comprendere, tramite la documentazione e tramite alcune simulazioni effettuate in ambiente docker, il funzionamento di Traccar, delle API fornite e delle funzionalità offerte dal servizio, provando ad individuare punti di forza, possibili defezioni e relazioni con l'architettura di zCloud.

Dopo aver realizzato personas, use-cases e requirements, il passo seguente è stato quello di integrare Traccar con i servizi di zCloud in locale, modificando quindi il codice sorgente di traccar. Per integrare correttamente il sistema all'interno della piattaforma di zCloud, è stato necessario creare diverse interazioni con i microservizi di Zerynth, indispensabili per autorizzare e autenticare device ed utenti e accedere ai database di zCloud in lettura e in scrittura. Inoltre è stato anche essenziale apportare alcune modifiche ad un servizio di zCloud, ovvero il servizio che si occupa di gestire le diverse creazioni e modifiche delle workspaces, dei fleets e dei devices.

Capitolo 2

Background e strumenti utilizzati

Le conoscenze che ho acquisito durante il corso di studi mi hanno aiutato a comprendere le basi di partenza del lavoro che sarei andato a svolgere durante il tirocinio. Pur premettendo che buona parte degli argomenti erano quasi del tutto nuovi per me, come ad esempio la configurazione di un server nginx o l'utilizzo dei microservizi all'interno di una architettura cloud già esistente o l'utilizzo di docker e docker-compose, non ho avute grossissime difficoltà nell'apprendere tutto ciò che mi era nuovo.

2.1 API REST

L'utilizzo delle API è stato fondamentale per lo svolgimento del progetto di tirocinio. Le API (application programming interface) REST (Representational State Transfer) sono molto diffuse nel mondo della programmazione web e nella realizzazione di protocolli client-server. Il sistema REST è infatti basato su HTTP e quindi si basa sulla tipologia di comunicazione client-server. In questo sistema architetturale il server espone delle API, ovvero delle url specifiche, sulle quali vengono registrate delle funzionalità in base al tipo di richiesta (post, get, put) e al tipo di parametri o payload. Il client dunque ha la possibilità di effettuare delle richieste http a queste url particolari, realizzando quindi una vera e propria chiamata di funzione remota, in quanto il server una volta ricevuta la richiesta, eseguirà la funzione che è stata registrata per quella url, con quei parametri e per quel tipo di payload.

In un sistema come traccar ad esempio le api sono indispensabili per poter permettere al client di creare nuovi devices, nuovi gruppi, eliminare entità modificarle e ottenerle dal server. Uno dei formati più utilizzati per l'invio al giorno d'oggi è il formato JSON, un linguaggio di formattazione dei dati. Il linguaggio JSON è fondamentale per lo scambio client-server degli oggetti, infatti un punto di forza delle api è la possibilità del client di serializzare interi oggetti in json inviarli ad una api lato server in modo tale che il ricevente possa utilizzarli come parametri della funzione registrata a quell'endpoint. Ovviamente anche il viceversa è una possibilità concreta, anzi molto spesso il server restituisce nella response un json, come ad esempio di un'entità creata o di un report della richiesta fatta.

2.2 Docker e docker-compose

Il primo strumento utilizzato è una piattaforma che permette di eseguire in un ambiente sicuro applicazioni tramite codice sorgente o immagini, ovvero Docker. Docker infatti dà la possibilità di creare un *container*, ovvero un contenitore nel quale è possibile eseguire uno script o più in generale del codice. Ma la vera caratteristica utile di docker, è docker-compose, ovvero la possibilità di collegare più applicazioni e quindi più container tra di loro in una network. A tutti gli effetti viene realizzata una vera e propria rete docker, in cui le applicazioni che sono in esecuzione su essa possono comunicare tra loro scambiandosi dati e informazioni.

Ogni servizio di una rete di docker-compose ha essenzialmente due file principali: il `docker-compose.yml` ed il `Dockerfile`. Il primo è in pratica un insieme di regole che quel servizio deve seguire prima di avviare la propria esecuzione o durante la propria esecuzione, come ad esempio le porte che deve esporre, l'hostname all'interno della rete docker o anche le dipendenze che ha nei confronti degli altri servizi. Il `Dockerfile` presente anche in docker, è paragonabile ad un `makefile`, ovvero una sequenza di comandi che il sottosistema con kernel linux simulato deve eseguire.

Docker infatti è a tutti gli effetti un sistema virtuale, e come tale è implementata una simulazione di un kernel Linux, sulla quale vengono eseguiti in modo del tutto isolato i diversi container. Per far questo docker isola a livello di sistema i container limitando risorse come per esempio la CPU o quantità di memoria usufruibile da un container, rendendo così ogni servizio indipendente dagli altri a livello di risorse utilizzate. Docker riesce a far ciò utilizzando librerie per interfacciarsi con il sistema linux ed utilizzare funzionalità come namespace che offre la possibilità di limitare ciò che un processore o un insieme di processori può vedere a livello di informazioni relative, ad esempio, agli utenti come il loro `Id`, e `cgroup` che permette di associare ad un gruppo di processi delle limitazioni di risorse hardware (quantità di CPU, memoria, rete) e software (permessi alle risorse).

2.2.1 Utilizzo di docker e docker-compose

Questi due strumenti li ho entrambi utilizzati in primis nella simulazione che ho effettuato per studiare il sistema ed il funzionamento di Traccar e delle sue API e successivamente per testare l'integrazione di Traccar con i microservizi di `zcloud` all'interno della architettura cloud in locale.

All'interno dell'architettura erano presenti alcuni servizi come "portainer" che permette di visualizzare graficamente i container, i logs di ogni container, ed è stato un utilissimo strumento di debug. Oltre a portainer, è possibile citare tra i servizi principali che presentano un ambiente grafico, "pgadmin" per monitorare il database in postgres e "rqm" che permette di visualizzare lo stato delle code rabbitMQ.

2.3 Code rabbitMQ

Durante la fase centrale del progetto ho dovuto affrontare la parte relativa alla gestione dei dati inviati dai device tracciabili a Traccar. Dunque ho dovuto gestire questi ultimi inviandoli a code rabbitMQ presenti su `zcloud` che avrebbero poi svolto il compito di

fare lo storage dei dati sul database di zcloud.

Le code rabbitMQ non sono altro che un ausilio per realizzare la struttura publish-broker-subscribe del protocollo mqtt. Infatti ogni coda rabbitMQ rappresenta un topic sul quale i device pubblicano, o meglio, ogni coda possiede una "binding-key" (o routing-key) ovvero una chiave univoca la che identifica. In realtà il funzionamento delle librerie di rabbitMQ è leggermente diverso ed un po' più complesso.

Partiamo col dire che una coda rabbitMQ possiede come entità un *publisher* che pubblica i dati su una coda, ed un *consumer* che prende i dati dalla coda a cui è iscritto. Inoltre è presente un'ulteriore entità, l'*exchange*, che rappresenta un intermediario tra il publisher e la coda. L'*exchange* ha il compito di scegliere come e dove pubblicare il dato ricevuto dal publisher secondo un algoritmo specifico. Le tipologie di algoritmi principali sono quattro: fanout che invia dati a tutte le code presenti, direct che inoltra il dato solo alla coda alla quale viene indicato il topic del publisher, e poi abbiamo header e topic. Nel mio caso la tipologia era quella di un exchange topic, ovvero molto simile alla tipologia direct ma con la differenza che in questo caso le routing-key non hanno una qualsiasi struttura, bensì sono costruite in un modo specifico. Una routing-key in questo caso quindi è a tutti gli effetti una lista di parole delimitate da punti come ad esempio *j.data.dev-87y.temp*. E' bene specificare che mentre il publisher ed il subscribe rappresentano delle entità che devono essere implementate, l'*exchange* è un oggetto che già esiste, è necessaria esclusivamente la sua configurazione.

Nel mio caso specifico ho utilizzato le librerie di java per rabbitMQ, ed una volta individuato il punto del codice sorgente di Traccar in cui viene gestita la comunicazione con i device, ho implementato un publisher che pubblica i dati che aveva ricevuto dal device su una coda rabbitMQ presente sul cloud di zerynth.

L'*exchange* in zCloud era nominato *amq.topic*, mentre la binding key era nominata *data*. L'intero sistema di gestione delle publish dei device è stato gestito in modo tale che i device comunicassero direttamente a traccar, in cui tramite l'aggiunta di un apposito handler, dopo aver eseguito l'autenticazione e l'autorizzazione del dispositivo sfruttando microservizi interni allo zCloud, viene invocata l'esecuzione di un metodo presente all'interno di una classe, realizzata per gestire le interazioni con i vari servizi di zCloud, detto *publishOnRMQ* in cui viene effettuata la scrittura sulla coda precedentemente indicata con il tag *latlon*, che indica un dato relativo alla posizione del device. Infatti viene memorizzato un vero e proprio file json che corrisponde al marshalling di un entità detta *position* presente in Traccar.

Capitolo 3

Studio del sistema e analisi dei requisiti

La prima parte del tirocinio era composta da due fasi, ovvero uno studio preliminare della piattaforma, comprendere quali erano le entità coinvolte, come erano fatti gli oggetti che Traccar creava e gestiva (user, device, gruppi) e soprattutto uno studio mirato alle API fornite da Traccar stesso; la seconda fase invece è consistita nel trovare un parallelismo tra le entità di Traccar e le entità di zCloud, discutendone analogie, differenze e similitudini, e successivamente pensare a come si sarebbe potuto costruire il workflow di un utente generico che intendesse aggiungere su zCloud un device tracciabile, definendo quindi personas, user stories, use cases e requirements.

3.1 Studio della piattaforma Traccar

Per comprendere al meglio le funzionalità che Traccar offriva, dopo aver analizzato la documentazione, ho realizzato una simulazione utilizzando docker e docker-compose. La simulazione consisteva in un server costituito da una immagine di Traccar, un database postgres, e un client scritto in go che utilizzasse la maggior parte delle api fornite da Traccar. Il client infatti aveva una fase di inizializzazione in cui eliminava tutti gli oggetti creati da un'esecuzione precedente dal server nel caso in cui questi oggetti fossero presenti, altrimenti procedeva con l'inizializzazione dell'ambiente e quindi con la creazione dei devices dei groups e delle altre entità che Traccar offre.

Da questo studio delle API di Traccar sono emerse tutti i punti di forza e tutte le criticità di Traccar, e qui di seguito proverò a riassumerle in modo da capire meglio quali sono le problematiche che io ed il mio tutore aziendale abbiamo dovuto affrontare durante il corso del tirocinio.

3.1.1 Come funziona Traccar

La piattaforma da integrare ha essenzialmente la tipica struttura di una web application, costituita da un server, più precisamente un Jetty Web Server, che espone delle API e le websocket, un client e un database. Il server si occupa di gestire tutte le richieste

dei vari client che interagiscono con esso tramite web app, mentre le connessioni con i devices sono gestite da una Netty network pipeline che comunica indirettamente con il web server.

Questa piattaforma supporta molteplici tipi di protocolli per quanto riguarda i device e nel caso ad esempio nella mia simulazione ho utilizzato il protocollo OSMand (Open-StreetMap Automated Navigation Directions) poiché è davvero semplice da rispettare ed applicare. I device infatti pubblicano la loro posizione e i loro attributi direttamente al server utilizzando il protocollo scelto, compreso il loro uniqueid tramite il quale il sistema riesce ad identificare univocamente il device fisico associato al suo digital twin. Il database di default di Traccar è utilizzabile solo per quanto riguarda la fase di testing, mentre è consigliabile, da quanto riportato dalla documentazione di traccar di utilizzare un database esterno per la fase di sviluppo.

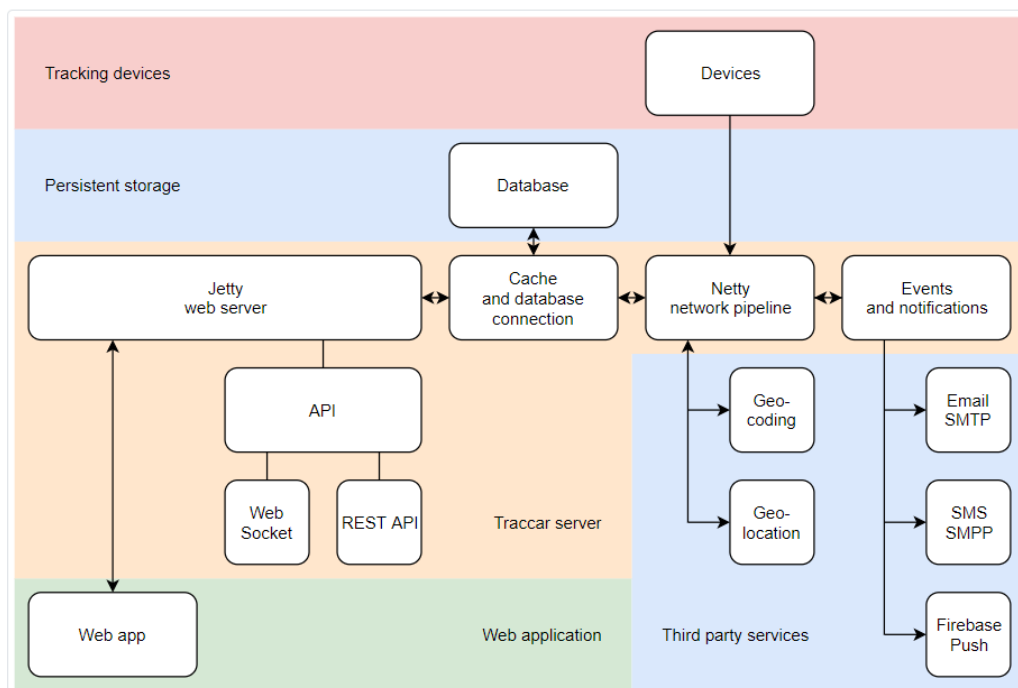


Figura 3.1.1: Architettura di traccar

3.1.2 Analisi statica del codice di traccar

Sebbene questa parte non è stata realizzata in un ordine cronologico definito, bensì è stato un processo che ho eseguito durante tutto il corso del progetto, è necessario descrivere in generale come è costituito il codice di traccar in modo da poter comprendere al meglio le sezioni successive di questo documento.

Il codice di traccar, in particolare lato server, è costituito da più parti divise in directory che svolgono ruoli differenti. Abbiamo la directory `api` e `api/resource` dove sono registrati tutti gli endpoint delle api di traccar. Poi tra le directory più importanti abbiamo i `models` dove si trovano tutte le classi che rappresentano gli oggetti presenti in traccar, come ad esempio `devices`, `groups` e `geofences` e la directory `protocols` che contiene tutti gli algoritmi di decodifica di tutti i protocolli supportati da traccar. Abbiamo inoltre la directory `handler` dove sono contenuti tutte le classi che hanno il

compito di handling di eventi come un device che inizia ad inviare dati al server o una notification da inviare al client. Infine è importante citare anche la directory database, dove si trovano tutte le classi che gestiscono i dati, e quindi forniscono tutti i metodi necessari per interfacciarsi al database di traccar a seconda dell'entità desiderata, e quindi abbiamo ad esempio il `DeviceManager()` che gestisce i dati riguardanti i device e così via. Infatti in questa directory è presente una classe detta `DataManager()` in cui sono eseguite le query necessarie per inserire, prelevare e modificare i dati dal database.

3.1.3 Architettura di ZDM

L'architettura del cloud di Zerynth è composta da tanti microservizi che cooperano per svolgere azioni e funzionalità specifiche. Nel mio caso ho dovuto spesso volte interfacciarmi con molti di questi microservizi per poter accedere ad esempio ai database di zerynth sia in scrittura che in lettura.

Giusto per citarne alcuni: *rpgbouncer*, *login-service*, *wfd-service*, *dev-authorization-service*, *dev-authentication-service*. Il primo permette di accedere solo in lettura ad un particolare database di Zerynth chiamato *zStorage*¹ composto a sua volta da sei partizioni, tre dette "master" e tre "repliche" dei relativi master.

Il servizio di *login-service* permette di autorizzare un utente registrato, e nel mio caso ho utilizzato questo servizio indirettamente, e più avanti nel documento verrà spiegato meglio nel dettaglio.

In seguito abbiamo il *wfd-service* che in realtà ho dovuto modificare, in quanto questo servizio gestisce la creazione, l'aggiornamento e la rimozioni delle entità all'interno dello *zCloud*, ed infine i servizi *dev-authentication* e *dev-authorization*, indispensabili per l'autenticazione del device e per autorizzare quest'ultimo ad utilizzare le risorse all'interno dell'architettura di zerynth.

3.1.4 Caratteristiche di Traccar

Traccar offre una caratterizzazione del proprio spazio di lavoro (anche se come vedremo dopo non esiste in realtà il concetto di "workspace" di un utente), infatti si ha la possibilità di aggiungere una "geofence" per limitare un'area specifica che può essere realizzata con qualsiasi tipo di forma, delle "notifications" per ricevere delle notifiche sugli eventi a cui si è interessati, aggiungere dei "commands" ovvero degli attributi calcolati sulla base degli attributi base che un device possiede. Inoltre un'altra cosa di rilievo è senza dubbio poter personalizzare ogni singolo device aggiungendo un set potenzialmente infinito di attributi custom. Da segnalare anche la possibilità di poter usufruire dei reports personalizzabili e anche di grafici riguardanti i device associati ad un profilo.

¹I database in zerynth sono principalmente tre *zCloud*, *ZDM* e *zStorage*, nell'ultimo in particolare vengono memorizzati per ogni workspace tutti i dati pubblicati dai device.

3.1.5 Problematiche emerse

Purtroppo sono emerse diverse problematiche durante il corso del progetto e molte scelte effettuate in fase di analisi sono state modificate in quanto nelle fasi più mature del progetto è migliorata la consapevolezza degli strumenti e la conoscenza del codice stesso di traccar.

Una delle problematiche più importanti è certamente la mancanza di accedere al servizio da terze parti utilizzando dei token d'accesso. O meglio, è possibile estendere le funzionalità di traccar per l'autenticazione in modo tale da sfruttare il protocollo LDAP per l'autenticazione da terze parti. Nonostante ciò questa ipotesi è stata scartata in quanto poco pratica da realizzare e quindi la strada presa è stata quella di estendere direttamente il codice sorgente di traccar.

Inoltre anche il claiming del dispositivo è stato un passo importante da realizzare in quanto traccar non supporta l'utilizzo di token firmati da parte dei device, bensì l'autenticazione dello stesso è effettuata solo tramite l'analisi del codice IMEI del device.

3.2 Struttura di zCloud

zCloud ha un'impostazione leggermente diversa da Traccar, in quanto permette di creare per ogni utente diversi spazi di lavoro, ma nonostante ciò alcuni oggetti sono molto simili. In zCloud ogni utente ha la possibilità di creare da una a più workspace, in ogni workspace da zero a più fleet e in ogni fleet da zero a più device. Analizzeremo adesso la struttura dei json di ogni elemento di zCloud, confrontandola con quella dell'oggetto che ho deciso di mappare in Traccar.

3.3 Parallelismo Traccar e zCloud

In questa fase era necessario trovare una soluzione ai problemi che erano sorti con questa fase di studio preliminare di Traccar. Come prima cosa bisognava trovare una relazione tra le varie entità e quindi capire come codificare un oggetto presente in zCloud in uno di Traccar nel modo migliore possibile. Come vedremo qui di seguito il problema più grande sarà rappresentato dalla workspace, entità che in zCloud esiste ma che in Traccar no.

Device Il device in zCloud ha una forte similarità con il device in Traccar, qui di seguito presento il json dell'oggetto di Traccar a quello di zCloud.

```
{
  "id":1105,
  "attributes":{},
  "groupId":0,
  "name":"name-of-dev",
  "uniqueId":"0",
  "status":"offline",
  "lastUpdate":"",
  "positionId":0,
  "geofenceIds":[],
  "phone":"",
  "model":"",
  "contact":"",
  "category":"helicopter",
  "disabled":false
}
```

JSON del device in Traccar

```
{
  "device": {
    "id": "dev-123456789",
    "name": "device name",
    "account_id": "acc-123456789",
    "fleet_id": "flt-123456789",
    "fleet_name": "fleet name",
    "workspace_id": "wks-123456789",
    "workspace_name": "wksp-name",
    "created_at": "",
    "phys_id": "devkit",
    "identities": null
  }
}
```

JSON del device in zCloud

Abbiamo infatti una corrispondenza quasi uno a uno con molti attributi, quali ad esempio *id*, *name*, che sono ovvie ed intuitive. Inoltre anche attributi come il *fleet_id*, il *workspace_id* ed il *phys_id* in zCloud hanno una corrispondenza implicita ma che comunque esiste. Tratterò l'argomento della workspace in seguito, perciò adesso mi concentrerò sul *fleet_id* e sul *phys_id*. Il primo può essere perfettamente mappato con il *group_id*, poiché come vedremo successivamente, il fleet è simile al group di Traccar, il *phys_id* invece è in relazione con lo *uniqueId* essendo entrambi due identificatori univoci del dispositivo fisico. Infine *account_id* in zCloud può essere in realtà ignorato dato che ogni dispositivo in Traccar è collegato al suo user, in quanto osservando lo schema relazionale, esiste un collegamento tra la tabella degli user e quella dei device. Per i restanti attributi, nel caso in cui dovessero essere necessari, si può ricorrere alla creazione di nuovi attributi nell'entità di Traccar, utilizzando il tag *attributes*.

Fleet Per quanto riguarda i fleet, come già accennato in precedenza, ho trovato una relazione con il group di Traccar, qui di seguito il confronto con i due JSON degli oggetti.

```
{
  "id":0,
  "attributes":{},
  "groupId":0,
  "name":"group-name"
}
```

JSON del group in Traccar

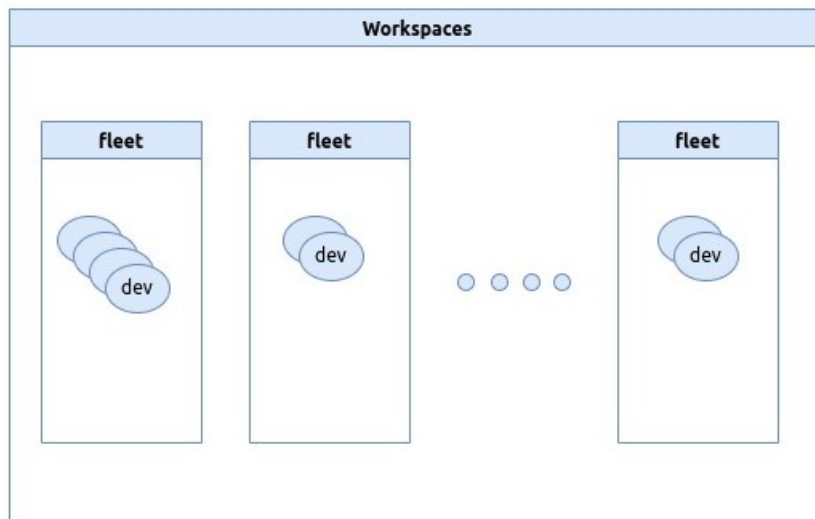
```
{
  "fleet": {
    "id": "flt-12345",
    "name": "fleet-name",
    "description": "fleet descr",
    "account_id": "acc-12345",
    "workspace_id": "wks-1234",
    "created_at": "data"
  }
}
```

JSON del fleet in zCloud

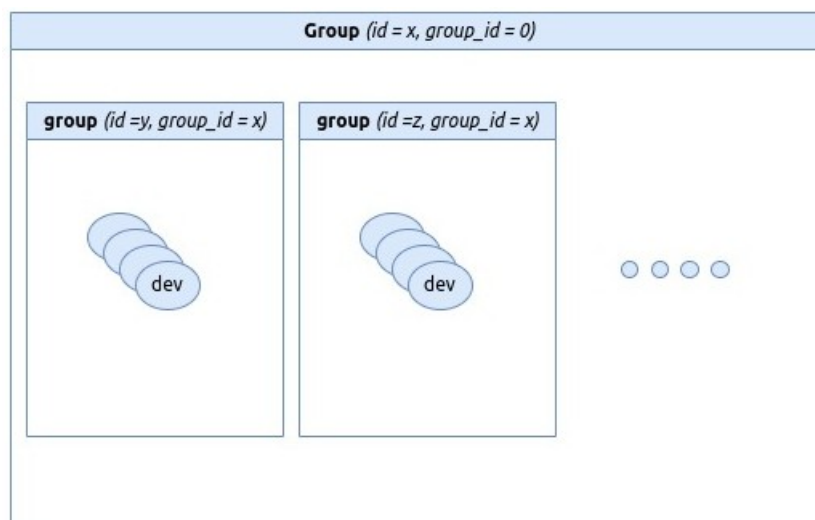
Vale anche qui il discorso fatto per i device, si vede molto chiaramente come le due entità sono molto simili. E' importante evidenziare come in traccar ci sia anche un *groupid* da non confondere con l'*id*, in quanto il primo indica l'indic del gruppo parent a cui appartiene, infatti in Traccar un gruppo può avere infiniti sottogruppi a cui fa da parent.

Workspace Terminando l'analisi con la workspace, dopo aver pensato a diverse idee su come modellarla in Traccar, la soluzione che ho preso è stata quella di mapparla come un group che non ha parent. Una workspace quindi è a tutti gli effetti un gruppo che ha l'*groupid* sempre a zero, in quanto non appartiene a nessun gruppo. Tutti i suoi gruppi figli invece, faranno essenzialmente le veci dei fleet.

Grazie a questa caratteristica di Traccar, ovvero che l'entità del group è qualcosa di generico, che rappresenta un insieme di elementi eterogenei, è stato possibile mappare con successo anche la workspace di zCloud, ecco qui di seguito una rappresentazione grafica della mappatura workspace e group.



Workspace di zCloud



Workspace mappata in group in Traccar

3.4 Personas e requisiti

L'Ultimo passo prima di iniziare l'implementazione e l'integrazione effettiva del servizio tracciar, è stato quello di descrivere alcuni archetipi di potenziali utilizzatori del servizio e quindi successivamente definire i possibili requirements.

In questa fase ho analizzato tre personas di cui due di queste appartenenti al mercato B to B e una sola personas che svolgeva il ruolo di consumer normale. Le aziende che potrebbero voler usufruire di questo servizio possono quindi essere legate al mondo dei trasporti, all'ambito militare o nel noleggio di veicoli come ad esempio tutte quelle realtà che si occupano di micromobilità.

Prima di passare ai requisiti ho realizzato alcuni use cases, azioni che l'utente potrebbe potenzialmente svolgere sul sistema in sviluppo. Prendendo ad esempio la creazione di un device tracciabile di un utente già registrato su zCloud, in figura 3.4 è rappresentato lo schema relativo di questa attività.

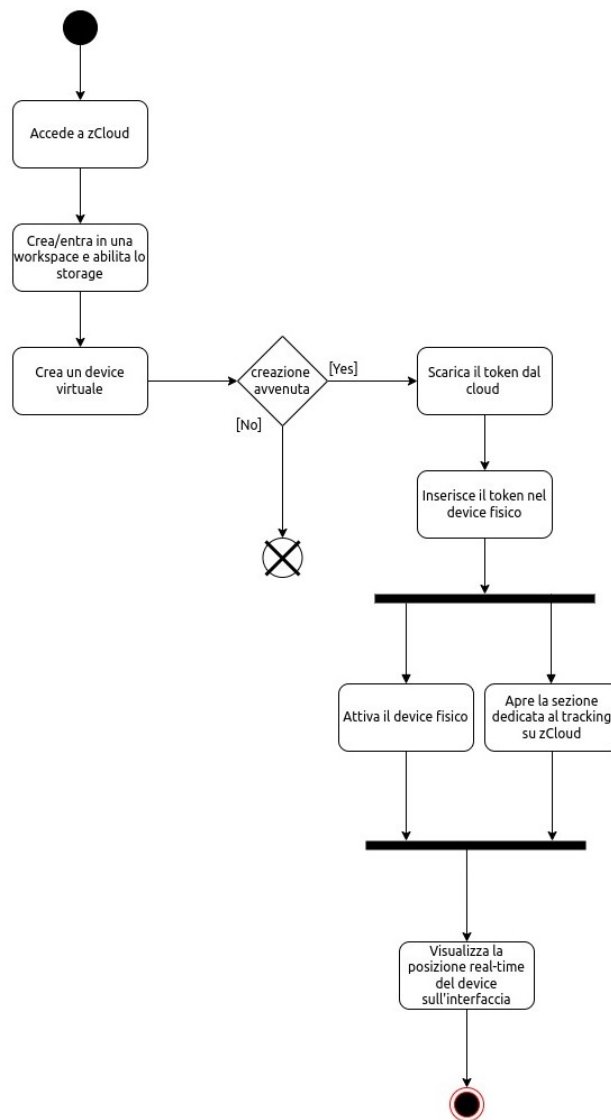


Figura 3.4: Diagramma dell'attività di un utente sulla creazione di un device e la sua visualizzazione della posizione real-time su interfaccia

E quindi sono infine passato ai requisiti tra i quali abbiamo ad esempio:

- L'utente non può aggiungere lo stesso device tracciabile due volte;
- Una workspace abilitata a tracciabile può contenere solo device tracciabili;
- Una workspace non abilitata a tracciabile non può avere device tracciabili;
- L'utente può avere lo stesso device su due o più workspace.

Capitolo 4

Integrazione del sistema

Inizialmente era ovviamente necessario poter accedere ai servizi di zerynth, e per fare ciò ho scaricato, tramite la repository aziendale, la loro architettura in locale, seguendo una determinata procedura che mi ha permesso di ottenere tutte le immagini dell'architettura di zerynth e di conseguenza inserire il codice sorgente di traccar come servizio proveniente non da un'immagine bensì da una vera e propria build del codice del software, modificando alcuni file di configurazione che avevano spesso il formato di file ".yml".

Per integrare correttamente il sistema di Traccar con la piattaforma di zerynth, è stato necessario mettere in collegamento il servizio di tracciamento dei dispositivi con i microservizi interni dell'architettura cloud di zCloud. E' stato dunque fondamentale capire come realizzare le chiamate api agli endpoint che i servizi di Zerynth espongono e codificare i dati ottenuti da essi per poi poterli utilizzare correttamente.

La fase iniziale dell'intero processo di implementazione è consistita nell'inserire traccar all'interno dei servizi di zCloud, connettendo il servizio all'intera architettura. Per far ciò ho dovuto implementare un file di configurazione di traccar e inserirlo nell'apposita directory, nella quale era situata tutta la configurazione di Nginx. Successivamente è stato necessario autenticare l'utente attraverso l'account id che il *login-service* di zCloud inviava come response nel caso in cui una autenticazione di uno user fosse andata a buon fine, e poi passare anche all'autenticazione e autorizzazione dei device che avrebbero dovuto poi comunicare con traccar.

Ovviamente per poter testare (ovviamente inizialmente tutto in locale) le singole fasi di implementazione, era necessario avvalersi di una simulazione di un device che invia dati al servizio traccar. Per far ciò ho realizzato un piccolo script in golang che invia dati al sistema utilizzando il protocollo OsmAnd ¹(scelta esclusivamente per una questione di praticità, il protocollo OsmAnd è molto semplice da realizzare) emulando quindi gli spostamenti di un dispositivo che si muove sulla mappa.

¹OsmAnd (OpenStreetmap Automated Navigation Direction protocol) è un protocollo di live tracking utilizzato dall'omonimo software di tracciamento OsmAnd, esso si può impiegare, lato device, inserendo nella url, alla quale il dispositivo intende mandare i dati, tutti i parametri necessari alla localizzazione e identificazione del device stesso. Un esempio di url strutturata in base al protocollo OsmAnd è il seguente: "http://tracking.org:5023/?id=23&lat=34.4&lon=2.4&speed=0&alt=2345"

4.1 Configurazione di Traccar in NGNIX

In questa parte ho realizzato il primo collegamento con traccar e la struttura di zCloud, collegando il servizio al proxy-server. Oltre a specificare il dominio con il quale Traccar sarebbe diventato raggiungibile dagli host esterni all'architettura di zCloud, la connessione ad NGNIX permette alla piattaforma di delegare le operazioni di default al cloud e quindi ai microservizi interni. Ad esempio, una delle principali azioni che in parte viene delegata ad uno dei servizi di Zerynth è l'autenticazione dello user quando intende accedere a traccar.

NGNIX infatti inoltra la richiesta di login da parte di un utente ad un servizio chiamato *login-service*, il quale si occupa di verificare l'esistenza di tale utente, e nel caso in cui l'autenticazione andasse a buon fine *login-service* restituisce un parametro al proxy-server, ovvero l'account id dell'utente appena verificato. NGNIX dunque inserisce questo identificativo all'interno di un campo dell'header, così da poter permettere a tutti i servizi interni di identificare l'autore delle richieste interne che si scatenano in cascata.

Qui di seguito una piccola rappresentazione grafica della connessione di traccar all'architettura cloud di Zerynth.

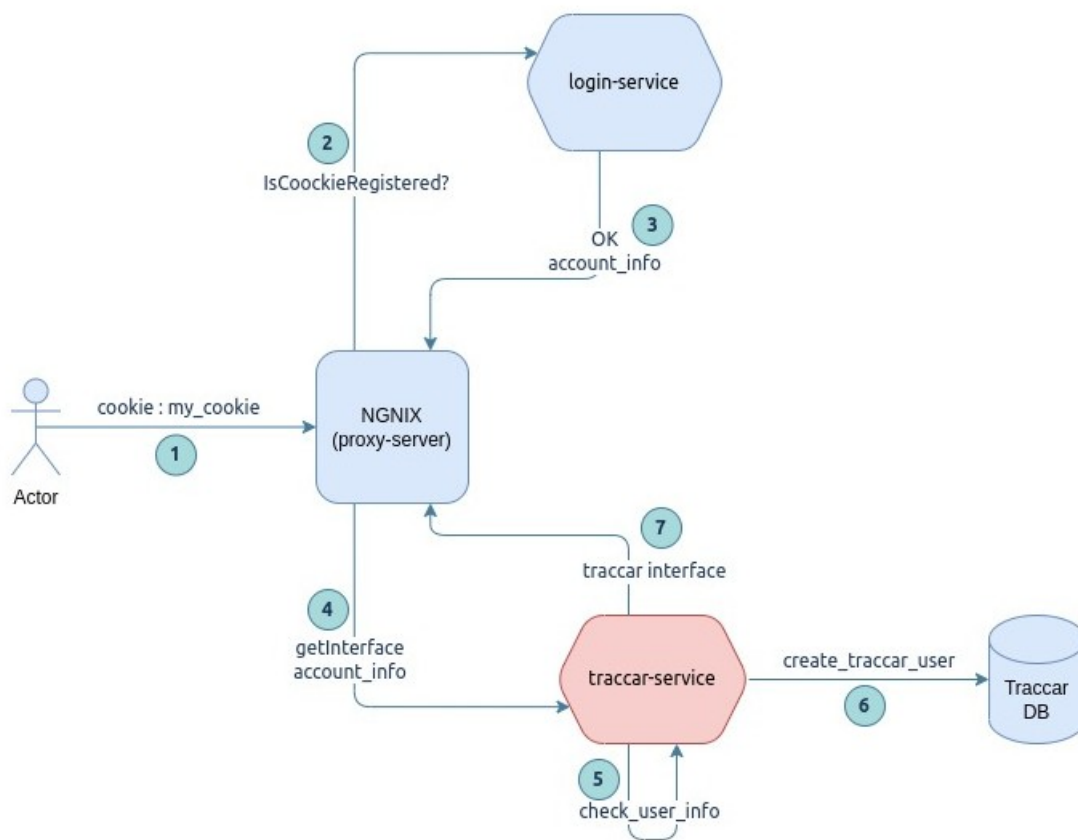


Figura 4.1: Prima fase di integrazione di traccar

Per completare la configurazione di traccar con NGNIX era fondamentale inserire anche l'inoltro delle connessioni per le web-socket, in quanto i client si connettono ad esse per aggiornare i dati riguardanti i device e in generale le entità presenti. Pertanto ho dovuto inserire una nuova location `/api/socket` in cui venivano inoltrate tutte le richieste effettuate a `http://traccar-service:8082/api/socket`.

4.2 Autenticazione dell'utente in Traccar

Il normale funzionamento di traccar prevede un login composto da username e password per accedere al servizio. Ma questa soluzione non era naturalmente adeguata al nostro scopo, era infatti necessario poter far accedere l'utente automaticamente una volta che si fosse registrato e avesse effettuato l'accesso su zCloud. In realtà buona parte di questa funzionalità è già realizzata in quanto il cloud possiede un servizio che autentica lo user, citato in precedenza il *login-service*. Questa piccola parte architetturale funziona nel seguente modo, seguendo lo schema in figura 4.1:

1. Il proxy-server NGNIX riceve la richiesta di accesso da parte di un client all'url in cui traccar è specificato come location (ad esempio `traccar.zerynth.localhost`) con il cookie dell'utente;
2. NGNIX inoltra la richiesta al *login-service* con il cookie che l'utente ha passato nella richiesta precedente;
3. Il servizio restituisce al server NGNIX la risposta, nella quale se l'utente risulta esistere, viene incluso l'*account-id* di tale utente all'interno di un campo header *x-user-info*;
4. NGNIX inoltra poi la richiesta iniziale, con l'aggiunta del nuovo campo dell'header a traccar;
5. Traccar viene eseguito un controllo per verificare che l'utente con quell'*account-id* sia già registrato nel database di traccar, se non dovesse essere così, allora viene creato un nuovo utente, e salvato nel database e poi l'interfaccia viene restituita al proxy-server NGNIX.

4.2.1 Utilizzo dell'*account-id* in Traccar

A questo punto il passo successivo è stato quello di istanziare una nuova sessione di traccar per l'utente. Il client che ha effettuato una richiesta di sessione a traccar dunque è già autenticato dal cloud, quindi non rimase altro che trovare un modo di gestire le informazioni nell'header che NGNIX inoltra a Traccar, più precisamente l'*account-id*. Nella classe dove viene gestita la richiesta di una sessione quindi ho dovuto modificare i vari controlli che traccar effettuava, infatti ecco come normalmente era strutturata una richiesta get di sessione all'endpoint² `/api/session`:

²Per endpoint si intende l'api esposta dal server, ovvero la parte di url che aggiunta alla base url del server garantisce delle funzionalità registrate, ad esempio nell'api `traccar.zerynth.localhost/api/session` l'endpoint è `/api/session`

HOST: "http://traccar_url/api/session"

HEADER: "Authorization Basic: user_code64:password_code64"

Dunque non rimaneva altro che modificare il punto in cui il server elabora la richiesta del client che vuole ottenere una sessione e quindi il punto in cui viene gestito lo scambio di dati all'endpoint `api/session`. La classe che si occupa di ciò è *SessionResource.java*, qui ho implementato una logica che ottiene dall'header della richiesta del client l'account-id dell'utente e controlla se è registrato nel db di traccar. Se è vero allora restituisce al client il formato Json con le informazioni relative all'utente, se invece non dovesse esistere nel database di traccar allora viene creata una nuova istanza e restituito il Json con le informazioni sull'utente appena creato, che possiede come nome l'account-id di zCloud.

Concludendo posso essere certo quindi, che gli utenti create siano tutti già stati autorizzati dal servizio di login-service e quindi non occorre fare aggiuntive verifiche, se non quelle di controllare che l'utente sia già registrato nel database di traccar.

4.2.2 Considerazioni sulla soluzione

Possono essere effettuati dei miglioramenti e delle ottimizzazioni a questa soluzione, però la scelta è stata quella di tenere questa implementazione in quanto essa rappresenta una realizzazione pratica e funzionale, infatti, uno dei problemi che avremmo dovuto affrontare se l'entità utente non fosse salvata nei db di traccar, sarebbe stato quello di mantenere tutti i link con gli oggetti come le geofence e le notification, ovvero tutti quegli oggetti che obbligatoriamente appartengono allo schema relazionale di Traccar. Più avanti vedremo come ho effettuato alcuni tentativi per evitare lo "sdoppiamento" delle risorse e quindi ad esempio avere per uno stesso device un'istanza sul database di traccar e un'altra sul database di zCloud.

4.3 Gestione della comunicazione con i device

La fase immediatamente successiva è stata quella di implementare tutta la fase di gestione dei dispositivi, codificando i dati che inviano a traccar in modo tale da ricavare informazioni da poter utilizzare per chiamare i servizi interni di zCloud. Le decisioni più importanti che sono state prese in questo caso sono state due, ovvero dove dovessero essere salvate le informazioni sulle varie posizioni di un device, e dove dovessero essere memorizzate le informazioni di un device appena creato.

Per quanto riguarda la prima parte, la scelta è ricaduta sul salvare le posizioni sullo zStorage di zCloud, in modo tale da permettere anche ad altri servizi all'interno dell'architettura dell'azienda di poter accedere a quei dati, senza dover modificare altri servizi. In generale si può schematizzare tutta la parte relativa ai dispositivi in una sequenza di azioni svolte da traccar, ecco qui di seguito una rappresentazione di questa parte architetturale che comprende solamente la parte di autorizzazione, autenticazione e memorizzazione dei dati.

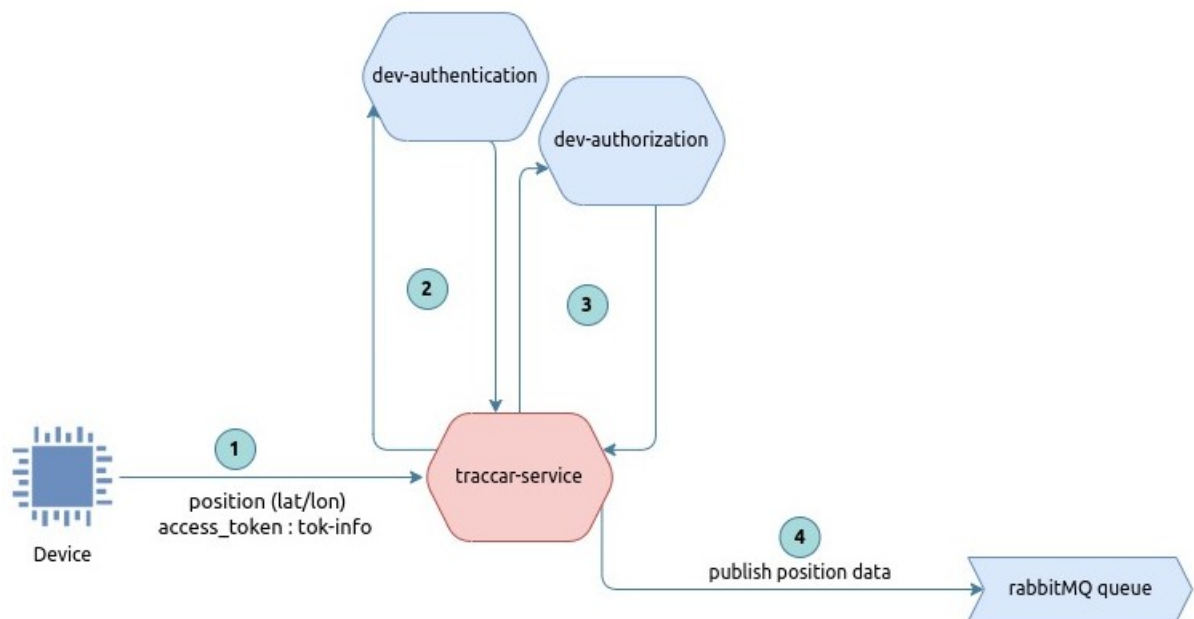


Figura 4.3: Integrazione dei device tracciabili

4.3.1 Autenticazione e autorizzazione dei device

Quando un device invia dei dati al sistema tramite uno dei protocolli che traccar supporta, le prime azioni che traccar deve effettuare sono autenticare e autorizzare³ il dispositivo, solo a quel punto può effettuare il salvataggio del payload all'interno di zStorage.

Per l'autenticazione era necessario contattare il dev-authentication service di Zcloud inviando come payload un json specifico che contenesse precise informazioni riguardanti il device che dev'essere autenticato. Traccar riceve il device_id e l'access token del dispositivo⁴ e queste due informazioni saranno indispensabili per costruire il payload da inoltrare al servizio di Zerynth. Il json specifico da inviare come payload è rappresentato qui sotto.

³Per *autenticazione* si intende la verifica da parte di dev-authentication che il device esista correttamente per quell'user specifico. Mentre per l'*autorizzazione* il dev-authorization si occupa di assicurarsi che il device abbia i diritti necessari di lettura e/o scrittura all'interno della piattaforma.

⁴L'*access token* viene generato dal cloud ed attribuito al device per poter identificare univocamente lo stesso in fase di autenticazione.

```
{
  "client_id":"dev-9dozztcisbhj",
  "entity":"dev-9dozztcisbhj",
  "token":"bsbhJonvdTVFgvkBH98JJKnbl",
  "proto":"mqtt",
  "dcn":"",
  "clicert":"",
  "icn":"",
  "sha1":"",
  "error":"",
  "sni":"",
  "verified":"NONE",
  "result":"",
  "result_msg":""
}
```

Il payload viene quindi inviato al service di zerynth che restituisce uno status code per confermare che l'autenticazione è avvenuta correttamente.

Oltre all'autenticazione è necessario autorizzare il device ad utilizzare le risorse interne del cloud. Per far ciò è necessario contattare diversi endpoint di dev-authorization che il servizio stesso espone, in modo tale da eseguire tutti i controlli per ogni entità specifica che il dispositivo potrebbe utilizzare.

Gli endpoint da contattare sono quindi tre e qui di seguito elencati:

1. Exchange: endpoint dedicato all'autorizzazione di un device in relazione ad un particolare exchange, vanno indicati il nome di tale exchange, il virtual host sul quale è presente l'exchange, il device_id e il tipo di permesso, nel mio caso in scrittura;
2. Ip del device: in questo endpoint viene autorizzato l'indirizzo ip del device, e devono essere indicati l'ip del dispositivo, il device_id e il virtual host;
3. Topic: infine tramite questo endpoint si può autorizzare un device per uno specifico topic, indicandone il nome del topic, il device_id del dispositivo, il virtual host, il tipo di permesso, nel mio caso in scrittura e infine anche la routing-key specifica.

Nel caso in cui questa fase fallisca il device non può comunicare la sua posizione al sistema né ovviamente salvare i propri dati all'interno del cloud.

Queste chiamate ai due servizi vengono effettuate all'interno di un'unica funzione, invocata dalla classe BaseProtocolDecoder() che a sua volta possiede un metodo onMessage() che si occupa di gestire tutte le azioni di controllo che devono essere eseguite ogni qual volta un device comunica con Traccar.

Per ottimizzare il codice questa funzione che contatta i servizi di zerynth è implementata all'interno di una classe chiamata ZdataManager() che svolge un compito di "interfaccia" tra i dati di Traccar e i dati di zCloud. All'interno di questa classe viene inizializzata una variabile privata che memorizza tutti i dispositivi appena autenticati e autorizzati, in modo tale da evitare di fare eccessive richieste http ai microservizi del cloud.

4.3.2 Storage dei dati dei device

Dopo aver autorizzato i dispositivi, la fase successiva è stata quella di memorizzare i payload di dati riguardanti la posizione all'interno dello zStorage, come raffigurato dalla freccia 4 nella figura 4.3. Più precisamente all'interno di Traccar è stata implementata una funzionalità che effettua delle publish sulle code rabbitMQ presenti nell'architettura di Zerynth, questi dati verranno gestiti da altri microservizi che salveranno le informazioni all'interno dello zStorage di zerynth.

Questa funzionalità è fornita da un metodo all'interno della classe citata precedentemente ovvero `ZdataManager()`, che oltre ad effettuare la publish sulla coda rabbitMQ, si occupa di aggiornare l'ultima posizione nota del device che ha appena comunicato con il server, modificando una variabile `HashMap` locale che associa un device alla sua ultima posizione inviata.

E' stata dunque necessaria la registrazione di un nuovo handler⁵ custom, chiamato `ZdataHandler()`, all'interno della pipeline degli handler presenti in traccar e successivamente l'implementazione del metodo `handlePosition()` della classe. Il payload che viene pubblicato sulla coda rabbitMQ ha una struttura ben specifica, ovvero rappresenta in formato JSON la modellazione dell'entità "Position" presente in Traccar. Infatti il device invia al server la sua posizione e indipendentemente dal suo protocollo viene codificato tutto in un oggetto di tipo "Position". Nel mio caso dunque viene effettuata una deserializzazione di tale oggetto in un file JSON e successivamente viene effettuata la publish con questo payload. La struttura del JSON è rappresentata qui sotto:

```
{
  "protocol": "osmand",
  "server_time": "dd/mm/yyyy HH:MM:SS",
  "device_time": "dd/mm/yyyy HH:MM:SS",
  "fix_time": "dd/mm/yyyy HH:MM:SS",
  "outdated": false,
  "valid": true,
  "latitude": 38.6,
  "longitude": 42.1,
  "altitude": 2000,
  "speed": 12,
  "course": 2.2,
  "address": "via ... ..",
  "accuracy": 0.7,
  "network": "main_net"
}
```

Una volta effettuato il salvataggio del payload, viene aggiornata una variabile tenuta internamente alla classe `ZdataManager`, che memorizza le ultime posizioni note di ogni singolo device.

⁵All'interno del server di traccar ci sono diversi handler che si occupano di effettuare azioni diverse quando avvengono determinati eventi, come ad esempio un device che comunica la sua posizione, o un device che entra in una geofence. Ogni handler implementa una `BaseDataHandler()` o una `BaseEventHandler()`, classi che possiedono dei metodi abstract da implementare, come ad esempio `handlePosition()`.

4.3.3 Recupero dei dati da zStorage

Per preservare il corretto funzionamento di tutte le funzionalità di traccar come geofences, report e notifications era indispensabile poter recuperare i dati relativi alle posizioni salvate sullo zStorage. Anche in questo caso è stato fondamentale avere la possibilità di sfruttare un servizio interno al cloud per poter accedere più facilmente ai dati in lettura. Il servizio in questione è il *pgbouncer* che si occupa di ricevere le query da parte di un altro servizio, e gestire le richieste in lettura al database.

I metodi che implementano la parte relativa all'estrazione dei dati si trovano all'interno della classe *dataManager* di *traccar*. Sono stati implementati due metodi in particolare, uno per ottenere l'ultima posizione nota di ogni device, e l'altro per ottenere la lista delle posizioni di un dispositivo. Una volta recuperati i dati, essi possono essere o inviati al client che ha effettuato la richiesta tramite api oppure sono utilizzati per inizializzare alcune strutture dati.

4.3.4 Considerazioni finali sulla gestione dei dati dei device

La difficoltà principale riscontrata in questa fase, è stata quella di capire esattamente il funzionamento di *traccar* in quanto concerne la gestione dei dati inviati dai dispositivi, e quindi individuare i punti esatti in cui inserire le chiamate ai metodi nuovi e comprendere al meglio quali sarebbero state le conseguenze in seguito alle modifiche del codice.

La parte più complessa è stata quella di evitare di modificare in maniera eccessiva il codice di *traccar*, cercando altresì di estenderlo aggiungendo classi custom. Per buona parte di questa fase l'implementazione di questi nuovi moduli è avvenuta con successo, sebbene è doveroso sottolineare come in alcuni punti è stato del tutto inevitabile agire direttamente sul codice originale di *traccar*.

Anche per quanto riguarda la modifica della sorgente dei dati inviati dai device è stato importante il lavoro di code refactoring, in quanto è stato necessario sostituire del tutto alcuni metodi che già esistevano, come ad esempio il metodo che eseguiva il *fetching* dei dati dei dispositivi dal database di *traccar*.

4.4 Creazione dei device e delle workspace

Un altro aspetto fondamentale è la gestione della creazione dei device, e quindi garantire la sincronizzazione tra zCloud e *Traccar*. Il servizio di zCloud che ho utilizzato in questo caso è stato il *wfd-service*, un servizio interamente dedicato per la creazione, eliminazione, modifica e aggiornamento delle tre principali entità che esistono in Zerynth ovvero *workspaces*, *fleets* e *devices*.

In generale l'integrazione di questa sezione è logicamente abbastanza semplice. Il client crea un device sulla piattaforma di zcloud e immediatamente viene eseguita una post al servizio di *traccar* che salva il dispositivo sul proprio database come riportato in figura 4.4. Nelle sezioni seguenti entreremo maggiormente nei dettagli implementativi e delle problematiche riscontrate in questa fase.

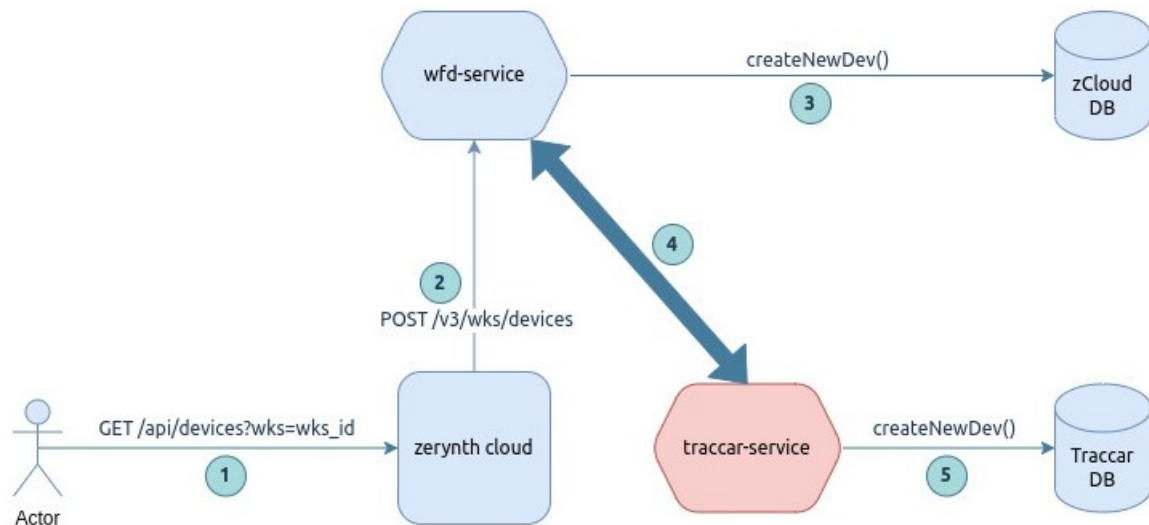


Figura 4.4: Creazione di un nuovo device

4.4.1 Modifica del microservizio wfd

E' stato necessario modificare il servizio di zerynth per poter creare un oggetto nel database in traccar, dunque inizialmente ho dovuto individuare il punto in cui veniva effettuata la creazione del device. Il servizio, interamente scritto in golang, era composto da una funzione `createNewDevice()` che invocava un'ulteriore funzione chiamata `storeDevice()`. In questa parte di codice veniva eseguita a tutti gli effetti la query per memorizzare i dati del nuovo device nel database.

Una deduzione ovvia di questa fase è che se la query al database dovesse fallire per un qualsiasi motivo, allora la creazione del device in traccar non dovrebbe avvenire. Per implementare questo requisito, sia la query al database sia la richiesta post al servizio di traccar, sono state inserite all'interno di una *transaction*. In questo modo è garantito che se il salvataggio al database non dovesse andare a buon fine, allora la richiesta di creazione di un nuovo device all'interno di traccar non verrebbe stata effettuata.

```

err := p.db.RunInTransaction(func(tx *pg.Tx) error {
    err := tx.Insert(device)
    //controlli di errore
    wks, err := p.GetWksIdOfDevice(device.ID)
    //controlli di errore
    resp := traccar_post(device, wks)
    if resp == nil {
        return errorsZDM
    }
    return nil
})

```

Come si può osservare da questo stralcio di codice estratto dalla funzione `storeDevice()`, sia l'inserimento dei dati all'interno del database sia l'invocazione del metodo che esegue la richiesta `http post` del device appena creato sono all'interno della `RunInTransaction()`. E' importante sottolineare che in queste righe di codice sono state omesse alcune parti, come ad esempio i vari controlli sugli errori provocati dalle varie richieste.

Prima di effettuare la `post` viene eseguita una funzione che recupera il `workspace_id` relativo alla workspace alla quale il device in questione è stato aggiunto. Questa funzione che ho implementato esegue una query al database all'interno di `traccar` per ottenere un'entità di tipo `Fleet` che rappresenta il fleet in `zCloud` a cui il device appartiene. Un fleet ovviamente, deve appartenere obbligatoriamente ad una workspace e questo oggetto ottenuto dall'interrogazione del database di `zCloud` possiede dei metodi, tra i quali `getWksID()`: di conseguenza è possibile risalire alla workspace a cui il device è stato appena aggiunto. Adesso, per concludere questa sezione, entrerà maggiormente nel dettaglio della funzione `traccar_post()`, essa infatti si occupa di effettuare la chiamata al servizio di `traccar` per permettere il salvataggio del nuovo dispositivo e prende come parametri il device e il `workspace_id` di della workspace a cui esso appartiene. Questo servirà nella fase in cui l'utente accederà a `traccar`, e dovrà visualizzare sull'interfaccia solo i device che appartengono alla workspace che l'utente sta utilizzando in quel momento. Questa funzione infatti deve comporre il payload in JSON strutturato in modo adeguato con i dati che appartengono al dispositivo, come raffigurato in figura 4.4.1, ed effettuare la richiesta `post` all'api `http://traccar-service:8082/api/devices`.

```
{
  "id":0,
  "attributes":{},
  "uniqueId":"dev_id",
  "name": "name",
  "groupId": "group_id",
  "staus":"offline",
  "lastUpdate":"dd/mm/yyyy HH:MM:SS",
  "positionId":0,
  "geofenceIds": [],
  "phone":"",
  "model":"",
  "category":"",
  "contact":"",
  "disabled":false
}
```

Prima di eseguire questa richiesta però bisogna chiarire un'ulteriore passaggio, ovvero la gestione della creazione delle workspace. Infatti come è stato deciso in fase di analisi e studio dell'intero sistema, una workspace era rappresentabile in `traccar` come un gruppo che non possedesse gruppi parent. Per realizzare questo schema strutturale quindi, prima di inserire il dispositivo all'interno del database di `traccar`, era necessario assicurarsi di una condizione ovvero, l'esistenza di un gruppo in `traccar` che rappresentasse la workspace in cui il device era inserito all'interno di `zcloud`. A questo punto abbiamo due casi possibili, se il gruppo esiste allora si recupera il suo id e lo si inserisce all'interno

del campo "groupid" del JSON precedente, altrimenti è necessario creare un gruppo in traccar che possiede come campo "name" il workspace_id della workspace desiderata. Vediamo adesso nello specifico le parti di codice che eseguono questi passaggi.

```
func traccar_post(device *Device, wks string) *http.Response {
    id := exists_group(wks, device.AccountID)
    if id == 0 {
        id = create_group(wks, device.AccountID)
    }
    tdev := traccar_dev{
        Id:           0,
        Attributes:   nil,
        GroupId:      id,
        Name:         device.GetName(),
        UniqueId:     device.GetID(),
        Status:       "",
        LastUpdate:   "",
        PositionId:   0,
        GeofenceIds: nil,
        Phone:        "",
        Model:        "",
        Contact:      "",
        Disabled:     false}

    client := http.Client{}
    body, _ := json.Marshal(tdev)
    req, err := doRequest("POST", "/api/devices", payload(tdev))
}
```

Qui sopra è mostrato uno "sketch" del codice di questa funzione. In particolare, osservando meglio la prima parte, si può notare una chiamata ad `exists_group()`, una procedura che restituisce il valore dell'id del gruppo di traccar relativo alla workspace di zCloud, oppure 0 nel caso in cui il gruppo non esista. Per ottenere l'id del gruppo, la funzione esegue una richiesta post alla api di traccar `/api/groups?wks=wks_id`, in questo modo il device potrà essere inserito all'interno di tale gruppo. E' bene specificare però che lato traccar-service è stato necessario effettuare alcune modifiche, infatti l'endpoint `/api/groups` non elabora di norma il parametro "wks". Dunque la modifica che ho dovuto apportare ha interessato la classe `SimpleObjectResource()` in quanto essa è superclasse di `groupsResource`, ed inoltre possiede un metodo che ha il compito di risolvere le richieste get da parte dei client⁶. In questa parte del codice di traccar ho implementato una parte di codice che effettua un controllo sul parametro wks, nel caso

⁶In realtà non gestisce tutte le richieste get, bensì essa fornisce un metodo base per le richieste get, e tutte le classi che si registrano ad una api ed estendono questa classe hanno la possibilità di usufruirne. Ad esempio la classe `deviceResource()` non estende la classe `SimpleObjectResource()` di conseguenza possiede una gestione delle richieste get alla api `/api/devices` appropriata.

in cui sia non nullo, si procede con la chiamata ad un metodo, realizzato da me, presente in `dataManager()` che esegue una query al database di traccar, così da ottenere il gruppo che possiede il campo "name" uguale al `workspace_id` passato come parametro. Successivamente, osservando il codice, se il gruppo non dovesse esistere allora viene invocata la `create_group()` che effettua una richiesta post a traccar all'api relativa ai groups con questo payload:

```
{
  "id":0,
  "name": "wks_id",
  "group_id":0,
  "attributes": {}
}
```

Viene dunque inserito il `workspace_id` all'interno del campo "name", come accennato in precedenza.

Ecco in figura 4.5 la presentazione grafica della comunicazione tra `wfd-service` e `traccar-service`.

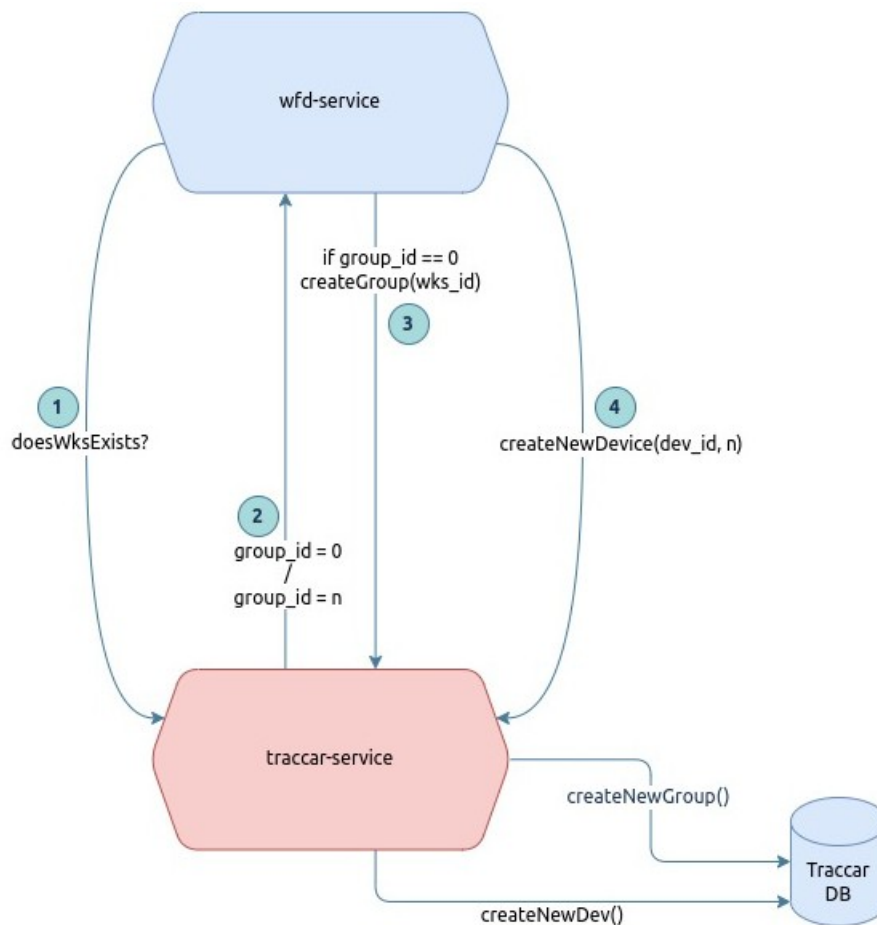


Figura 4.5: Interazione traccar e wfd

4.4.2 Filtro dei device

Per completare questa sezione correttamente, era ancora necessario affrontare un ultimo problema, ovvero come restituire le informazioni corrette al client che intende visualizzare i device di una specifica workspace. Per fare ciò, una volta individuata la classe che si occupava di gestire la richiesta alla api, chiamata `DeviceResource()`, era indispensabile capire come modificarla correttamente per adempiere al nostro scopo.

Quando un utente si collega a traccar, il client effettua le chiamate api per inizializzare l'interfaccia. E tra queste chiamate vi è la richiesta get all'endpoint `/api/devices`, che serve ad ottenere tutti i device che appartengono all'utente.

Il primo cambiamento dunque è stato lato client, infatti la base url della richiesta all'endpoint `/api/devices` doveva essere modificata, inserendo nella query un parametro che contenesse l'id della workspace desiderata. Per questioni relative al tempo l'id della workspace è stato inserito come costante, ma di fatto, non viene cambiata di molto la sostanza di questa funzionalità, in quanto basterebbe reperire il `workspace_id` tramite NGNIX, analizzando la query o inserendolo come una informazione aggiuntiva in un campo header custom, dopo la richiesta da parte dell'utente. Di conseguenza l'url risultante è:

```
http://traccar.zerynth.localhost/api/devices?wks=wks-hwb7hsoisa
```

A questo punto il server avrebbe dovuto codificare la query e ottenere l'id della workspace desiderata e ciò è implementato all'interno di `DeviceResource()`, in cui si ottengono tutti i device che appartengono all'utente⁷ e poco prima di restituire il risultato al client, in aggiunta alla funzionalità base, viene invocata una funzione di `ZdataManager()` che, dati una lista di device e una `workspace_id`, filtra i dispositivi restituendo solo quelli che appartengono a quella workspace. Per far ciò questa funzione, detta `filter_device()`, effettua una query al database in traccar realizzata nel seguente modo:

```
SELECT tc_devices.*  
FROM tc_devices INNER JOIN tc_groups ON tc_groups.id = tc_devices.groupid  
WHERE tc_groups.name = :wksId
```

restituendo così tutti i device corretti.

4.4.3 Osservazioni sulle complicazioni emerse

Inizialmente il primo tentativo che è stato fatto aveva come obiettivo evitare di duplicare l'entità dei device sul database di traccar. Questa prima soluzione però ha fatto emergere diverse problematiche, in primis riguardanti le geofences e di conseguenza anche le notifications. Infatti le geofences sono collegate strettamente ai device nel database, tramite una chiave secondaria, e riuscire a sostituire questa connessione non risulta essere

⁷In realtà può essere anche invocata una richiesta per ottenere un solo o un sottoinsieme di device indicandone come parametro lo `uniqueId`, ma è bene notare che è stata modificata proprio l'url base della api, di conseguenza l'utente effettuerà la richiesta `/api/devices?wks=wks-shabi&uniqueId=imei`, ottenendo una risposta positiva solo nel caso in cui quei device appartengono alla workspace in cui sta lavorando in quel momento.

semplice. Dunque è stato optato per la soluzione attuale, in quanto il tempo necessario per implementare il sistema di connessione tra le varie entità non sarebbe stato poco, inoltre non era da escludere che le idee proposte per risolvere questo problema avessero comunque fallito con una buona percentuale.

In generale è comunque una soluzione buona, pur duplicando le entità su due database diversi, garantisce comunque un ottimo livello di stabilità del sistema a discapito di un overhead in termini di spazio e tempo.

E' doveroso evidenziare che in questa parte è stata volutamente tralasciata la gestione dei fleet, in quanto si mostra molto simile a ciò che è stato fatto per i device e per le workspaces. In generale è ovvio che per uno sviluppo futuro della piattaforma sarà necessario implementare anche questa parte.

Capitolo 5

Integration test

Per avere la certezza che tutto il sistema sia stato correttamente integrato e che tutte le funzioni funzionino correttamente, è stato necessario realizzare una suite di test per verificare questo punto. Per testare il sistema è stata utilizzata la libreria di pytest, e quindi l'intera parte di integration test è stata scritta in python.

I punti principali che ho dovuto verificare erano:

- Verificare che il redirect del login avvenga correttamente, e che quindi venga creato un utente su traccar;
- Verificare che i dati che i dispositivi inviano a traccar siano correttamente salvati all'interno di zStorage;
- Verificare che i device siano correttamente autorizzati e autenticati;
- Verificare che la creazione di un device avvenga correttamente, con la creazione di un device copia nel database di traccar.

La suite di test genera un report in html in cui vengono riportati tutti gli errori riscontrati nell'esecuzione dei test. Alcune funzioni erano già fornite da Zerynth e quindi nel corso di questo capitolo non entrerà troppo dentro i dettagli implementativi.

5.1 Test del login

Il test del login consiste nel verificare che un utente che è già registrato su zCloud riesca ad accedere ai contenuti di traccar correttamente. Per effettuarlo viene simulata una richiesta di login con le credenziali di un utente già registrato, e poi una volta ottenuta la conferma che l'utente esiste su zCloud bisogna controllare che sul database di traccar è stato creato un oggetto che contiene i dati di tale utente.

```
def test_user_auth(env_api, zcloud_client):
    res = zcloud_client.internal_auth()
    assert res.status_code == 200
    j = res.json()
    env_api._x_user_info_header = res.headers["x-user-info"]
    user = env_api._x_user_info_header.split(";")[0]
    r = requests.get("http://traccar.zerynth.localhost:8082/api/session",
        headers={"x-user-info":env_api._x_user_info_header},auth=(user, user))

    assert r.status_code == 200
    j = r.json()

    assert "name" in j
    assert j["name"] == users
```

Per analizzare al meglio il codice, è preferibile concentrarsi sugli "assert". Il primo infatti serve a verificare che l'utente è correttamente registrato sulla piattaforma di zCloud. Successivamente, abbiamo una richiesta ad un endpoint di traccar e subito dopo abbiamo tre assert. Il primo di questi tre si assicura che traccar restituisca correttamente l'utente richiesto, per verificare quindi che questo oggetto è stato adeguatamente salvato sul database di traccar. Gli ultimi due assert servono per una maggiore sicurezza, e per verificare che traccar abbia memorizzato in modo giusto i campi all'interno della tabella dell'istanza, andando a controllare gli attributi presenti all'interno del JSON restituito dal server di traccar.

5.2 Test del device

Per accertare che il salvataggio del device avvenga correttamente all'interno del database di traccar sono indispensabili due passaggi, la creazione del device all'interno di una workspace in zCloud e la richiesta dei dati del device appena creato da traccar passando come parametro il suo device_id. Ecco qui di seguito il codice realizzato per fornire questa funzionalità.

```
def test_device_in_traccar(zcloud_client, env_api, zdm_v3_client):
    r = zdm_v3_client._post("/workspaces/wks-6o7yvdo8inlt/devices",
        data={
            "name": "newDev"
        })

    assert r.status_code == 200
    j = r.json()
```

Qui sopra viene effettuata la creazione di un device di nome "newDev" all'interno di una workspace specifica.

```

res = zcloud_client.internal_auth()
r = requests.get("/api/devices?uniqueId="+dev_id,auth=(user, user))
assert r.status_code == 200
j = r.json()
dev = j[0]
assert "name" in dev
assert dev["name"] == "newDev"

```

Anche in questo caso abbiamo tre assert, in cui il primo ci garantisce che il device esista anche all'interno di traccar e che quindi sia stato correttamente creato anche nel database del servizio di tracking. Gli ultimi due sono essenziali per verificare che il device abbia i campi corretti, come ad esempio quello del nome.

5.3 Test dello storage

In questo test l'obiettivo è stato quello di assicurare che i dati che un device invia al servizio traccar, siano correttamente salvati all'interno di zStorage. Per far ciò viene simulata una device che invia dei dati tramite protocollo osmAnd a traccar, e successivamente viene prelevato l'ultimo dato inserito all'interno di zStorage da quel device e si controlla che il tag ed il payload siano quelli attesi.

```

r = zstorage_client._get("{0}/data?size=1&device={1}",
                        workspace_id, device_id)

assert r.status_code == 200
resp = r.json()
res = resp["result"]

assert res[0]["tag"] == "latlon"
assert res[0]["device_id"] == "dev-6oboztcifcox"
assert res[0]["payload"]['latitude'] == 41.897

t = my_time(res[0]["timestamp_device"])

assert t is not None
assert t > start

```

Questa parte di codice viene eseguita dopo che il device ha comunicato il payload a traccar, infatti una volta effettuata la richiesta dei dati del device in questione da zStorage, una serie di parametri vengono controllati, come ad esempio lo status code della richiesta, e che quindi essa sia andata a buon fine, il tag del device che deve corrispondere al tag di tipo "latlon", l'id del device che deve corrispondere al device che abbiamo selezionato in precedenza per mandare dati a traccar ed infine il timestamp, che dev'essere superiore al tempo di inizio del test, in modo tale da essere certi di non recuperare un dato che era già presente all'interno di zStorage in precedenza.

5.4 Test dell'autenticazione del device

Infine troviamo quest'ultimo test, in cui il compito è stato quello di confermare che il device fosse correttamente autorizzato prima di accettare la ricezione dei dati.

```
resp = zdm_v3_client._get(  
    "/workspaces/{0}/devices/{1}/events?type=authentication&size=1",  
    env_api.workspace_id, env_api.device_id)  
assert resp.status_code == 200
```

Viene effettuata la richiesta all'endpoint di zCloud per verificare se quel device sia stato correttamente autorizzato dalla piattaforma per utilizzare le risorse presenti sul cloud, e per far ciò si verifica lo status code della response della richiesta.

5.5 Ulteriori modalità di testing

Per testare ulteriormente l'integrazione del sistema ho realizzato un client in golang che simula un invio dei dati al server di traccar, con una frequenza stabilita. Questa simulazione che ho realizzato è stata utile in fase di testing ma anche e soprattutto in fase di sviluppo, per monitorare e debuggare i vari passaggi che venivano implementati. Il client è quindi costituito da un ciclo for che effettua una richiesta post, utilizzando il protocollo osmAnd al servizio di traccar. Esso invia dati riguardanti la sua posizione e il suo token indispensabile per l'autenticazione.

Si visualizza quindi sull'interfaccia il device in movimento, e nella demo finale ho preparato anche delle notification ed una geofence, in modo tale che nel momento in cui il device entri nella geofence, traccar invia all'utente una notifica dell'evento, come raffigurato in figura 6.1-2.



Figura 5.5: Dispositivo di test Teltonika

Il passo successivo sarebbe stato quello di testare l'intero sistema integrato non in locale bensì in ambiente di test e con un device reale ed in figura 5.5 vi presento un esempio di un possibile dispositivo di testing che utilizza un protocollo appartenente alla casa produttrice chiamato TeltonikaProtocol, compatibile con traccar.

Capitolo 6

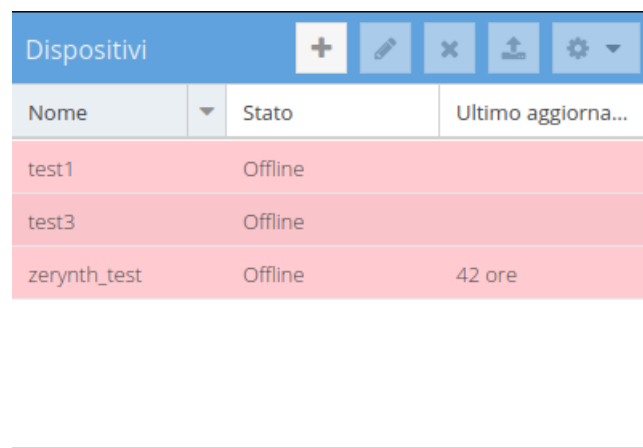
Conclusione

In conclusione l'intero progetto è stato terminato con un buon risultato, nonostante ci sono alcune scelte che potevano essere effettuate in modo diverso, mi ritengo comunque soddisfatto dell'integrazione del sistema, soprattutto per le numerose nozioni nuove che ho appreso.

Durante la simulazione finale si può osservare come tutte le componenti funzionano correttamente, qui di seguito illustro come ho eseguito la demo finale.

6.1 Demo finale

L'ultima parte del progetto era ovviamente dedicata a verificare il funzionamento di tutte le componenti tracciar integrate a zCloud. Per far ciò ho dovuto come prima cosa inizializzare l'ambiente, e quindi creare una workspace in zCloud e dopo creare all'interno di essa un device. Nella demo sono stati creati ben tre device come mostrato nell'immagine 6.1.



Dispositivi		
Nome	Stato	Ultimo aggiorna...
test1	Offline	
test3	Offline	
zerynth_test	Offline	42 ore

Figura 6.1: Lista dispositivi

Successivamente è stata creata una geofence ed una notification, entrambe poi associate al dispositivo "zerynth_test".



Figura 6.1-1: Creazione di una geofence

Come mostrato in figura 6.1-1 la geofence è stata posizionata sulla mappa in modo tale da poter controllare il corretto funzionamento delle notifications. Infatti una volta avviata la simulazione del movimento del device "zerynth_test" ecco qui il risultato.

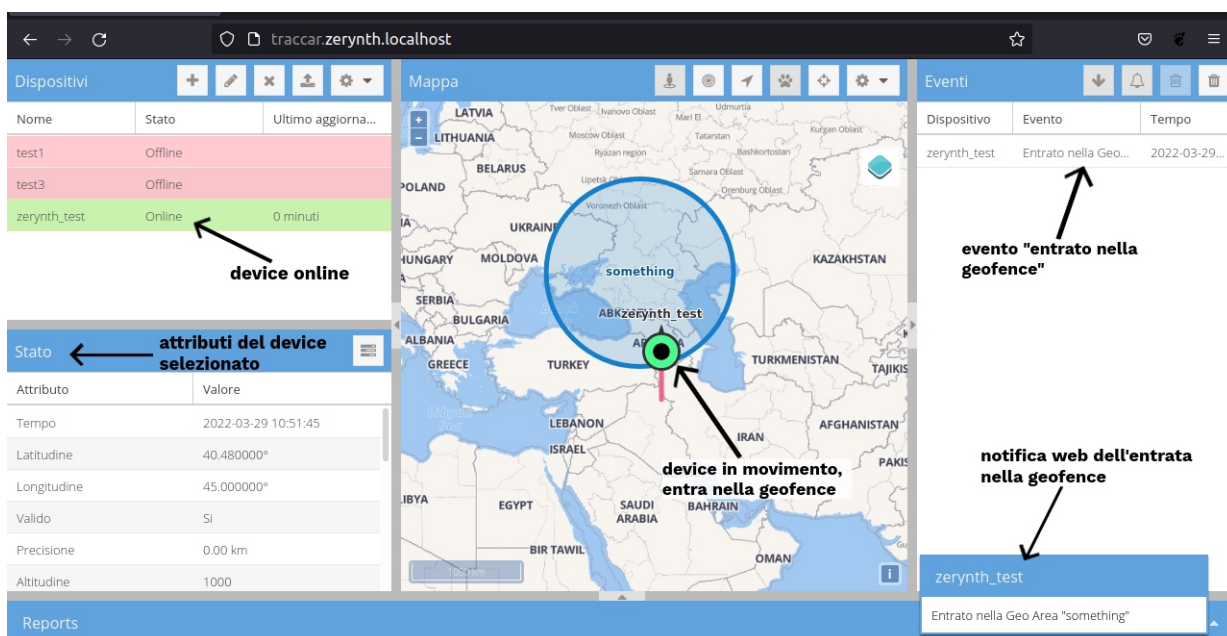


Figura 6.1-2: Interfaccia finale

In figura 6.1-2 è mostrata il risultato finale in cui tutte le componenti funzionano correttamente, a sinistra vengono elencati tutti i device presenti nella workspace richiesta dal client, ed in verde quelli che sono attualmente online. In basso a sinistra abbiamo

invece tutti gli attributi che il device selezionato invia al server di traccar, sia standard che custom mentre a destra abbiamo la sezione degli eventi e delle notifiche. Per quanto riguarda i report, nella figura 6.1-3 ne è visualizzato un esempio.

Tipo	Viaggi	Configura	Visualizza	Esporta	Email Report	Pulsici						
Nome Disp...	Ora di Part...	Odometro ...	Indirizzo di...	Ora di Arrivo	Odometro ...	Indirizzo di...	Distanza	Velocità M...	Velocità M...	Durata	Carburant...	Guidatore
zerynth_test	2022-03-22...	0.00 km		2022-03-22...	1224.51 km		1224.51 km	37021.5 kn	12.0 kn	0 h 1 m	0.0 l	
zerynth_test	2022-03-23...	2504.69 km		2022-03-23...	14193.24 km		11688.55 km	36106.9 kn	12.0 kn	0 h 10 m	0.0 l	
zerynth_test	2022-03-24...	25993.10 km		2022-03-24...	33340.19 km		7347.09 km	36582.0 kn	12.0 kn	0 h 6 m	0.0 l	
zerynth_test	2022-03-24...	40798.60 km		2022-03-24...	84213.20 km		43414.60 km	100353.8 kn	12.0 kn	0 h 14 m	0.0 l	
zerynth_test	2022-03-24...	87664.10 km		2022-03-24...	89111.26 km		1447.15 km	39482.9 kn	12.0 kn	0 h 1 m	0.0 l	
zerynth_test	2022-03-27...	90669.73 km		2022-03-27...	96235.70 km		5565.98 km	36865.0 kn	12.0 kn	0 h 4 m	0.0 l	

Figura 6.1-3: Report di un dispositivo

In questo caso vengono visualizzati i report riguardanti i viaggi del dispositivo, e quindi tutte le informazioni riguardanti la durata e le distanze.

6.2 Critiche e possibili miglioramenti

Il lavoro finale che ho realizzato presenta ovviamente delle criticità, che lasciano spazio a miglioramenti del codice, specialmente per renderlo maggiormente estendibile, e in generale per migliorarne ad esempio alcune caratteristiche come vedremo qui di seguito.

6.2.1 Modularità

Un punto essenziale per comprendere quanto un progetto sia realizzato ottimamente è la modularità, e di conseguenza quanto un software sia effettivamente estendibile con altri moduli ad esempio, o sia facile da mantenere. In questo probabilmente il sorgente che ho rilasciato come definitivo per la fase di testing presenta alcune lacune.

Prendendo come esempio la gestione dell'autenticazione dello user, è facile notare come sarebbe stato decisamente meglio sviluppare una nuova classe, chiamata ad esempio `ZsessionResource()` in cui si implementava un altro endpoint adatto all'autenticazione per gli utenti che provenissero da `zerynth`. Questo avrebbe decisamente portato ad una migliore modularità del codice e soprattutto lo avrebbe reso molto più leggibile. Invece uno dei problemi principali nella soluzione portata avanti nel progetto è quello di togliere una funzionalità base di traccar sovrascrivendo la mia parte di codice, e tale funzionalità magari avrebbe potuto rivelarsi utile per uno sviluppo futuro (ad esempio un superuser che vuole poter accedere come admin). Inoltre re-implementando questa classe, un eventuale aggiornamento del codice di traccar renderebbe molto pericoloso effettuare eventuali "merge".

Ovviamente la registrazione di un nuovo endpoint e quindi di una nuova classe, necessita di competenze adeguate, e avrebbe richiesto sicuramente molto più tempo di quello che era a nostra disposizione per realizzare un progetto funzionante.

6.2.2 Gestione delle entità

Un particolare difetto del mio progetto è l'isolamento delle api dall'esterno. Infatti il funzionamento corretto dell'interno sistema dovrebbe essere quello descritto nelle pagine

precedenti, più precisamente in figura 4.4, ovvero l'utente per poter creare un device ad esempio, effettua il login sulla piattaforma di zCloud e quindi crea il device virtuale. In conseguenza il device, se dovesse essere all'interno di una workspace tracciabile, verrà successivamente creata una sua copia sul database di traccar. Nella mia implementazione vi è però un problema in quanto le api fornite da traccar sono disponibili anche all'esterno. Pertanto nonostante all'interfaccia si può comunque impedire all'utente di creare device (o gruppi) direttamente dall'interno di traccar, rimane il problema delle api. Infatti basterebbe che l'utente realizzi le richieste strutturandole in un modo preciso (e quindi ad esempio inserendo il cookie) e potrebbe essere in grado di creare o eliminare i device dal database di traccar.

Sebbene questo potrebbe sembrare un problema lieve, in quanto un utente non vorrebbe mai modificare, alterare o in generale danneggiare un software che lui stesso utilizza, rimane comunque un problema che espone il sistema a possibili attacchi dall'esterno. Dunque potremmo considerare questa defezione come una leggera vulnerabilità che un malintenzionato potrebbe sfruttare.

6.2.3 Duplicazione delle istanze

Un problema che ho già trattato in precedenza è la creazione di istanze di supporto nel database di traccar. Ad esempio, con la creazione di un device all'interno di zCloud, comporta la creazione di un device all'interno del database di traccar. Questo ovviamente comporta una duplicazione delle risorse create, che in generale non è l'ideale. Limitare l'uso dello spazio è un ottimo obiettivo da porsi in applicazioni e servizi così grandi. Nel mio caso questa era la soluzione più veloce e semplice da implementare, e che allo stesso tempo rendesse l'intero sistema molto più stabile preservando tutte le altre funzionalità base che traccar fornisce.

6.3 Analisi dei punti di forza del lavoro svolto

Sebbene molte parti sono state realizzate sovrascrivendo o aggiungendo alcune parti nel codice di traccar, in altri punti l'integrazione delle nuove funzionalità è stata implementata con l'aggiunta di intere classi. Ne è un esempio la classe `ZDataManager()` in cui sono contenute tutte le funzioni di utility, per pubblicare sulla coda rabbitMQ, per autorizzare i device e per filtrare i device in base alla workspace; ma anche la classe `zDataHandler()` che come già accennato in precedenza ha il compito di implementare un handler, che a sua volta gestirà le posizioni che i device stanno comunicando.

Ma uno dei punti di forza del mio progetto è senza dubbio la possibilità da parte dei device di poter utilizzare fino a 127 protocolli diversi. Infatti tutte le funzionalità implementate in aggiunta sono state realizzate ad un "livello superiore" di astrazione, e quindi dopo che i vari decoder dei protocolli hanno effettuato la decodifica. Dunque questo permette di autenticare e autorizzare un device e permettere ad esso di scrivere i suoi dati nello `zStorage` a prescindere dal protocollo che viene utilizzato per inviare dati al server di traccar.

6.4 Ultime osservazioni sugli obiettivi raggiunti

In generale mi ritengo pienamente soddisfatto del lavoro effettuato sia per come sono stati raggiunti gli obiettivi preposti, sia del percorso che ho affrontato, nel quale ho imparato a comprendere al meglio il vero funzionamento di un'architettura cloud strutturata con microservizi.

Mi ha particolarmente colpito come è complessa la struttura e la composizione dei diversi servizi, inoltre la possibilità di analizzare il codice di un software esistente mi ha dato la possibilità di apprendere in modo più concreto come è realizzato un codice applicativo, in tutte le sue componenti, codice sorgente, codice compilato, e anche il lato front-end e quindi tutta la parte relativa all'interfaccia e in generale al lato client.

Di certo la fase implementativa è stata quella che mi ha interessato maggiormente, e quindi grazie alla curiosità e alla determinazione sono riuscito ad affrontare le non poche difficoltà che si sono poste lungo il processo. Durante tutto il progetto, posso affermare con certezza che la complessità maggiore è stata senza dubbio quella di comprendere davvero a fondo il funzionamento di traccar, e quindi analizzare il codice per comprendere al meglio le funzionalità di ogni classe in modo da essere consapevole cosa stessi modificando e quali conseguenze avrebbe potuto comportare. Infatti numerose volte è accaduto che intere fasi si bloccassero spesso o comunque subissero rallentamenti poiché piccole righe modificate all'interno di una classe provocavano grandi effetti collaterali su funzionamenti interni del software.

Durante questo tirocinio ho appreso come possono essere utilizzati strumenti come le code rabbitMQ o la piattaforma docker e docker-compose, le suite di test per controllare funzionalità specifiche di integrazione del sistema, l'utilizzo appropriato di query per prelevare o inserire dati in un database ed infine anche come sfruttare correttamente lo stile architetturale REST, utilizzando le chiamate api in modo corretto per implementare funzionalità o testare il sistema.

6.5 Ringraziamenti

Concludo il documento con i dovuti ringraziamenti, nei confronti dell'università che mi ha permesso di intraprendere questa opportunità, in particolar modo al professore Gabriele Mencagli che mi ha seguito durante tutto il processo di stesura della tesi. Allo stesso modo i ringraziamenti sono rivolti anche all'azienda che mi ha permesso, con le dovute precauzioni dovute alla pandemia da Covid-19, di svolgere questo tirocinio anche in presenza e a tutti coloro che ho avuto il piacere di conoscere durante questa mia piacevole esperienza. In modo particolare ringrazio Davide Neri, che in qualità di tutore aziendale, grazie alla sua disponibilità, mi ha aiutato a proseguire nel progetto superando alcune difficoltà incontrate e indicandomi la strada migliore lì dove avessi incertezze sulle scelte da realizzare.