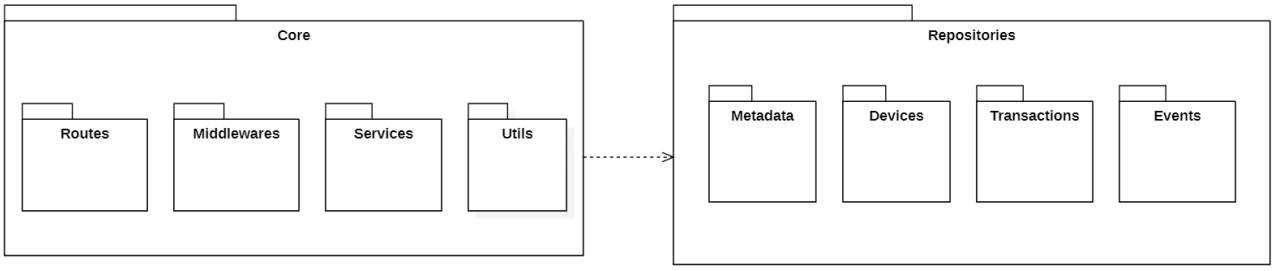
MFA Core – Architettura

Indice

Struttura generale

Il server Node sarà organizzato secondo la seguente struttura.



Il package Core contiene la definizione delle chiamate API e la logica del sistema, mentre Repositories contiene le classi per la comunicazione con i database.
Mentre il package Core verrà eseguito direttamente tramite uno script presente nella sua cartella base, il package Repositories costituirà un node module e verrà importato così nel Core.
Organizzando così il codice, le modalità di interazione con i database potranno essere facilmente riutilizzate da un altro sistema.

Core

I sotto-package di Core contengono:

- Routes: le definizioni degli URL della API e delle funzioni che ne sono in controllo

- Middlewares: le definizioni di alcune funzioni per elaborazioni comuni su diversi URL

- Services: la business logic del sistema, ovvero tutte le operazioni di elaborazione dei dati

- Utils: alcune funzioni che svolgono delle funzioni minori. Sono definite qui per rendere il codice più pulito o per evitare duplicazioni.

Inoltre è presente una classe Server che costituisce il nucleo del sistema. Esso contiene le configurazioni di base del server come la porta da esporre e permette di determinare quali Routes e quali Middleware verranno utilizzati e in che ordine. La classe server presenta due funzioni per inserire Routes e Middleware. L'ordine in cui le funzioni sono chiamate implica l'ordine in cui alle routes e ai middleware verrà ceduto il controllo delle richieste.

Routes

Il server esporrà la seguente lista di chiamate, suddivise in 4 gruppi gestiti da 4 classi, derivate da Routes.

Oltre ai valori forniti, ogni chiamata tranne metadata richiede che sia fornito un access_token nell'header Authorization e un domain_id nella query. I valori del body invece non sono mostrati visto che ricalcano i modelli rappresentati nella sezione Repositories.

METADATA	
GET metadata/:domain_id	Restituisce un URL che punta ai metadati utili all'utilizzatore
DEVICES	
GET /user/:user_id/devices/list	Restituisce la lista dei dispositivi dell'utente
GET /user/:user_id/device/:device_id/info	Restituisce le informazioni riguardanti il singolo dispositivo
POST /user/:user_id/device/add	Inserisce un nuovo dispositivo e ne restituisce l'identificativo
PUT /user/:user_id/device/:device_id/update	Sovrascrive il dispositivo specificato
PATCH /user/:user_id/device/:device_id/edit	Sovrascrive uno o più campi del dispositivo specificato
DELETE /user/:user_id/device/:device_id/remove?signed_device_id	Elimina il dispositivo specificato
GET /user/:user_id/device/:device_id/logs	Restituisce la lista degli eventi legati al dispositivo
GET /user/:user_id/devices/logs	Restituisce la lista degli eventi legati a tutti i dispositivi dell'utente
REQUEST	
POST /user/:user_id/transaction_target/:target_user_id/notify	Invia la notifica a tutti i dispositivi dell'utente indicato (target_user_id) e restituisce l'identificativo della richiesta di approvazione
GET /user/:user_id/transaction/:transaction_id/status	Restituisce lo stato della richiesta di approvazione specificata
CONFIRMATION	
POST /user/:user_id/transaction/:transaction_id/approve?device_id&signed_confirmation_code	Approva la transazione specificata
POST /user/:user_id/transaction/:transaction_id/deny?device_id&signed_confirmation_code	Rifiuta la transazione specificata

Nell'immagine sottostante si possono vedere le 4 classi e le funzioni che si occupano di gestire le chiamate. Ogni metodo è assegnato ad una singola chiamata.

Il metodo checkDeviceIdentity di DeviceRoutes e' eccezionale in quanto è sostanzialmente un middleware, che controlla le chiamate a

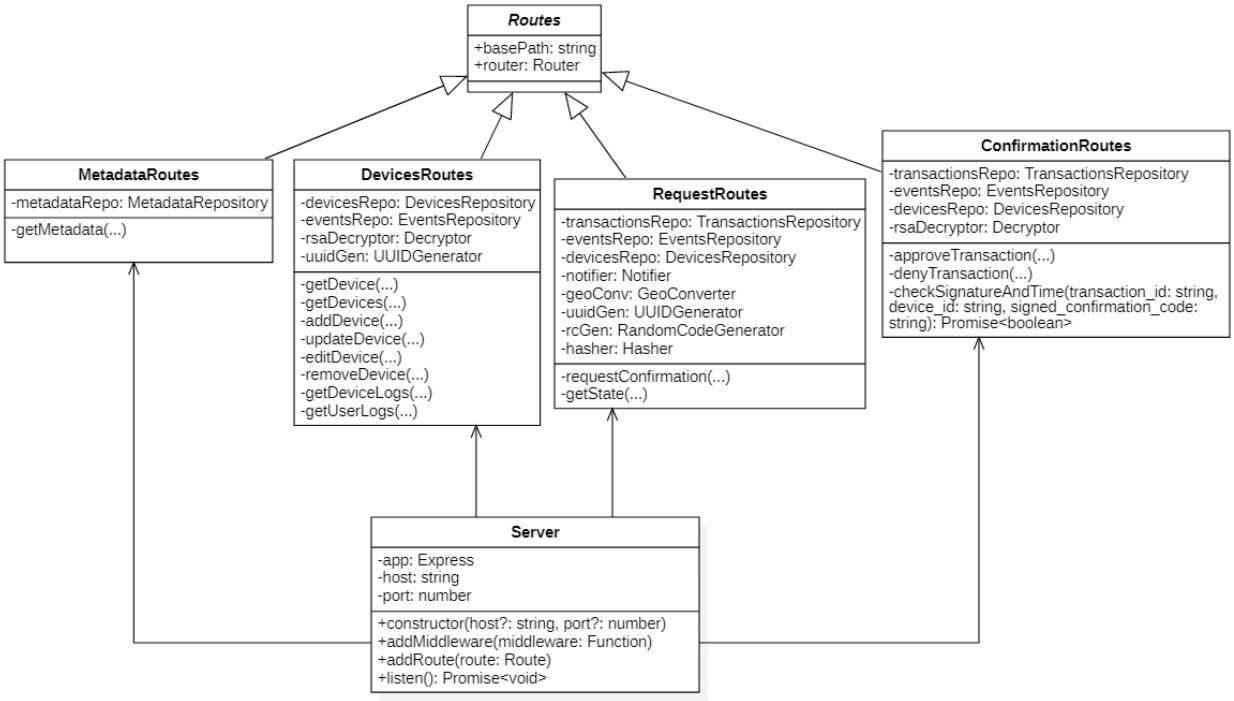
```
/user/:user_id/device/:device_id/remove?signed_device_id
```

prima che il loro controllo venga passato a removeDevice. Esso è stato inserito in questa classe perché non è utilizzato in altre. Questa funzione viene utilizzata per verificare l'identità dichiarata da un dispositivo nella query. L'espressione della propria identità da parte del dispositivo avviene

indicando il proprio codice di dispositivo firmato con la propria chiave privata. Questo middleware verifica il codice decriptandolo con la chiave pubblica del dispositivo. Per capire quale dispositivo sta effettuando la chiamata, e quindi quale chiave pubblica usare, si fa riferimento a device_id. Infatti un dispositivo può essere eliminato o da se stesso o dal client corrispondente al frontend. Nel secondo caso signed_device_id non è richiesto.

I metodi nel diagramma seguente sono dichiarati per semplicità senza argomenti ma nella realtà tutti ricevono tre oggetti:

- req: Request ovvero l'oggetto che rappresenta la richiesta,
- res: Response ovvero l'oggetto che rappresenta la risposta
- next: NextFunction ovvero la funzione da chiamare per passare il controllo alla funzione successiva su quell'URL.



Middlewares

Le funzionalità trasversali, ovvero utilizzate in più di una chiamata, sono racchiuse in funzioni chiamate middleware.

L'unica finora individuata è:

- **getMetadataMiddleware**: ottiene i metadati del dominio specificato nella richiesta e li memorizza perché possano essere utilizzati da funzioni successive

- **getTokenDataMiddleware**: ottiene i dati del token specificato nella richiesta e li memorizza perché possano essere utilizzati da funzioni successive

- **checkTokenMiddleware**: verifica che il token provvisto sia valido, che dia i permessi necessari e che l'utente che l'ha richiesto sia lo stesso dell'utente dichiarato nella richiesta

- **checkClientPermissionMiddleware**: verifica che il tipo del client che ha effettuato la richiesta sia ammissibile. Con tipo si intende uno tra: app mobile, web app e software utilizzatore

- **checkDeviceIdentityMiddleware**: verificha che, se il client è di tipo app_mobile, il dispositivo che ha richiesto l'operazione sia lo stesso di quello che la subisce

- **errorHandlerMiddleware**: se avviene un errore non ancora mostrato a schermo, questa funzione provvede ad avvisare l'utente con un messaggio che c'è stato un errore interno

Services

La business logic è definita nelle classi derivate da Service. Quelle individuate finora sono:

- **TokenConverter**: a partire dai Metadata di un dominio e da un access_token restituisce le informazioni legate a quest'ultimo. In particolare il codice che indica il suo richiedente e i suoi scope. Per farlo comunica con l'introspection API specifica del dominio specificato.

- **GeoConverter**: converte longitudine e latitudine nel nome di una località. GeoConverter è un'interfaccia che può essere realizzata da diverse possibili implementazioni. Quella scelta attualmente utilizza il servizio di OpenStreetMaps.

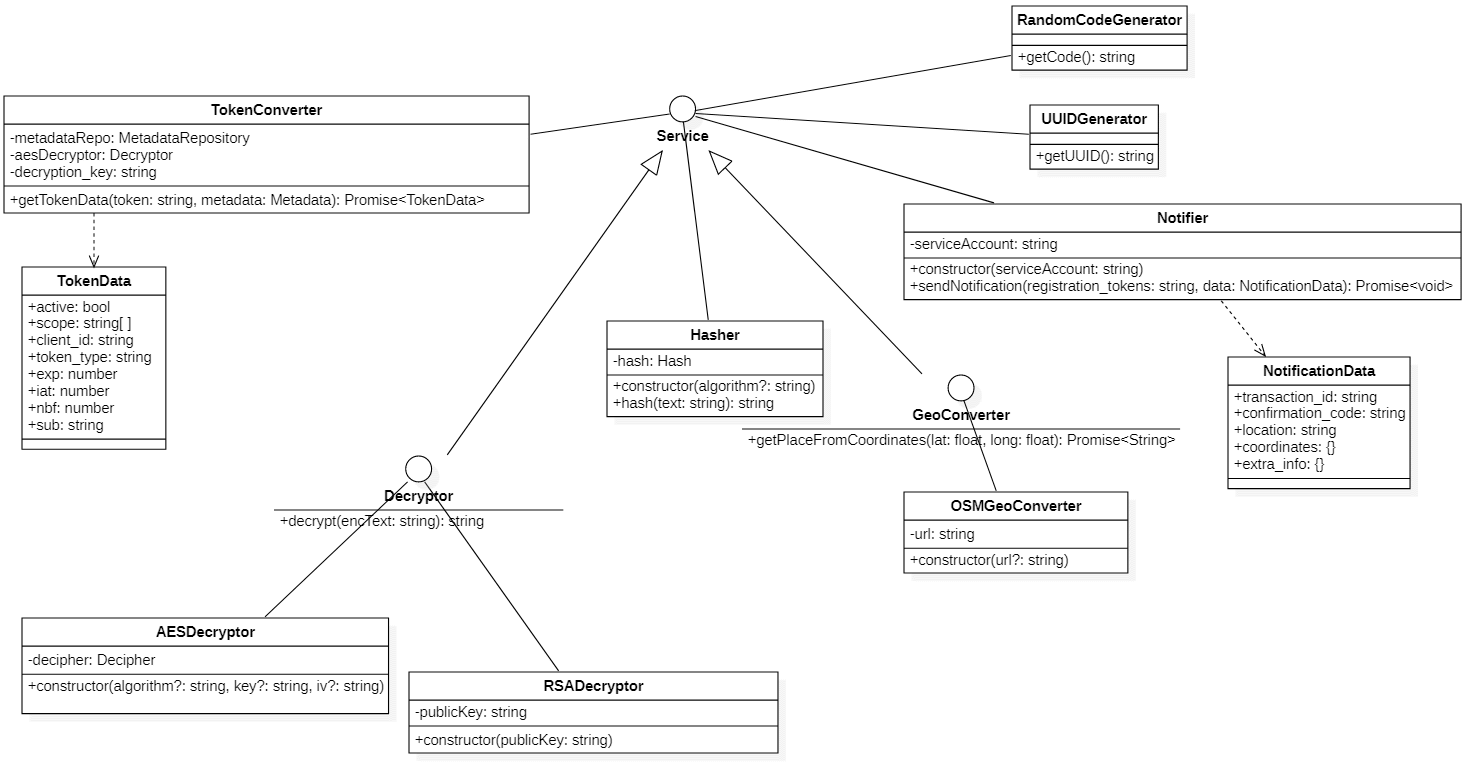
- **Notifier:** si occupa dell'invio delle notifiche ai dispositivi tramite il backend di Firebase. Per comunicare col backend deve certificare la propria identità con una stringa che identifica il suo Service Account.

- **AESDecryptor**: esegue la decrittazione di testi cifrati con AES. In particolare viene utilizzato per leggere i client_secret ottenuti dal database, che saranno poi necessari per la comunicazione con l'introspection API. Per eseguire la decrittazione deve essergli specificata la modalità di operazione tra ECB, CBC, CFB, OFB e CTR. Implementa l'interfaccia Decryptor.

- **RSADecryptor**: esegue la decrittazione di testi cifrati con una chiave privata. In particolare viene utilizzato per decriptare il codice ricevuto da un dispositivo durante la conferma o il rifiuto di un'operazione. Implementa l'interfaccia Decryptor.

- **UUIDGenerator**: fornisce degli identificatori univoci di tipo UUID che verranno poi assegnati ai dispositivi, alle transazioni e agli eventi memorizzati.

- **RandomCodeGenerator**: fornisce un codice generato in modo pseudo-randomico e non prevedibile, che può quindi essere utilizzato in ambiti crittografici.



Utils

Le funzioni finora individuate all'interno di questo package sono:

- **checkScope(url: string, scopes: string[]): boolean** : verifica che nella lista degli scope provvisti ci sia quello necessario per l'accesso all'url specificato. Lo fa leggendo le corrispondenze tra url e scope presenti in un file di configurazione.

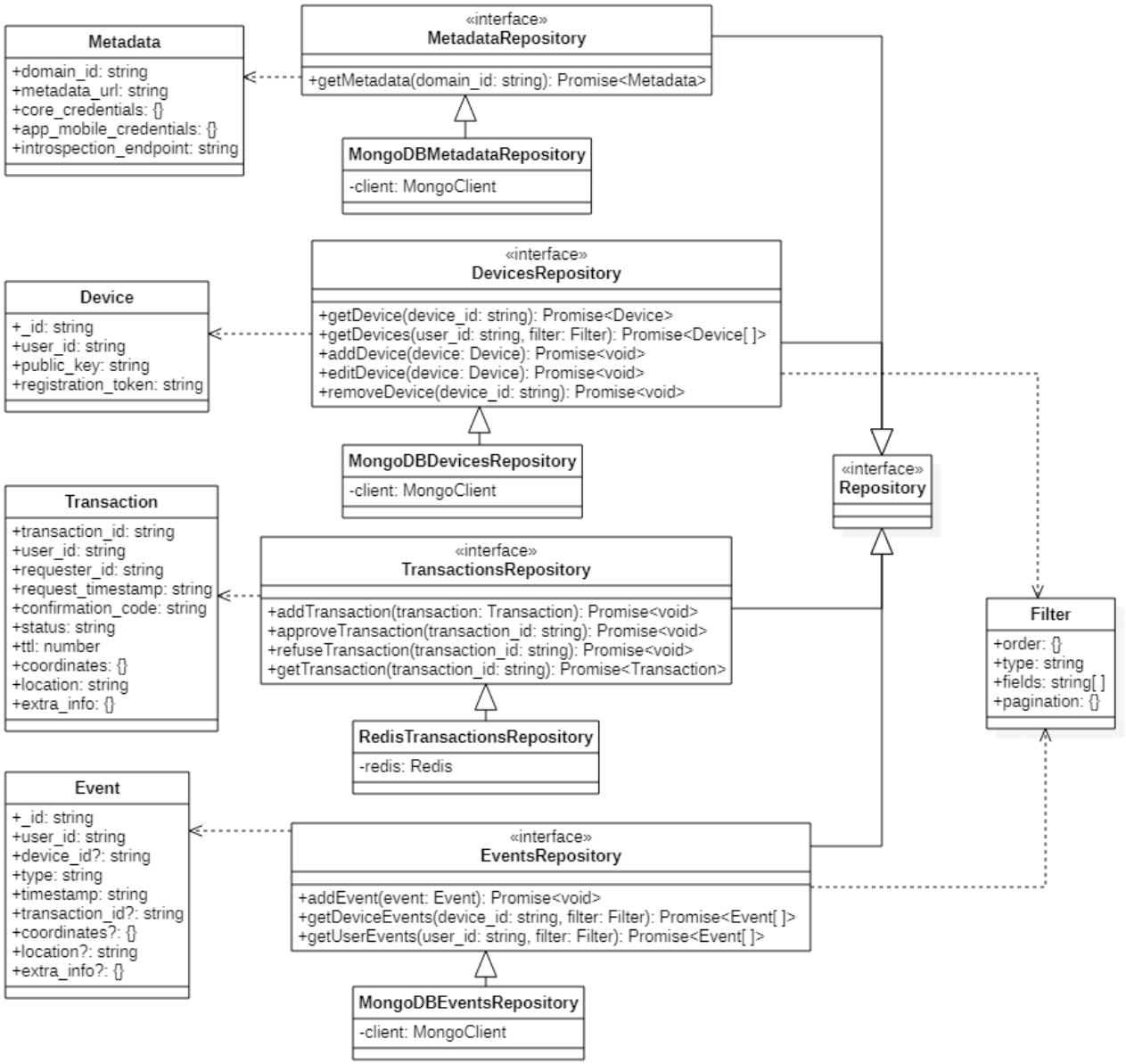
- **stringifyNestedFields(obj: any): any** : converte un oggetto a più livello in uno con profondità 1, trasformando in stringhe gli oggetti annidati

- **unstringifyNestedFields(obj: any): any** : converte un oggetto ad un solo livello e che contiene oggetti annidati sotto forma di strighe in un oggetto multilivello. Lo fa convertendo le stringhe rappresentanti oggetti in oggetti veri.

- **requestToFilter(req: Request): Filter** : riceve in input un oggetto Request, ricava i filtri specificati in esso e li racchiude nell'oggetto Filter che restituisce.

- **applyQueryAndFilter(collection: Collection, query: any, filter?: Filter): Promise<FindCursor>** : applica alla collezione la query e il filtro specificati.

Repositories



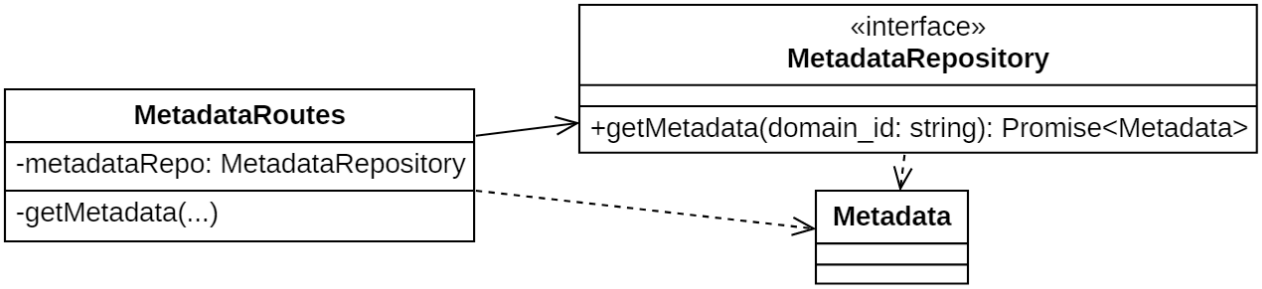
Il server lavora con 4 tipi di dati:
- metadati per l’autorizzazione
- dispositivi
- transazioni
- eventi

Per ognuno di essi sono stati definiti un modello e una repository. Ogni repository segue il pattern Strategy, ovvero i metodi vengono definiti in modo astratto in un’interfaccia e questa interfaccia viene concretizzata in una o più classi, ognuna focalizzata su un utilizzo concreto. In questo caso per ogni interfaccia è stata definita una sola classe concreta, specializzata su uno dei due tipi di database che saranno utilizzati: MongoDB e Redis. Questa astrazione rende un’eventuale cambiamento di database facile da integrare.

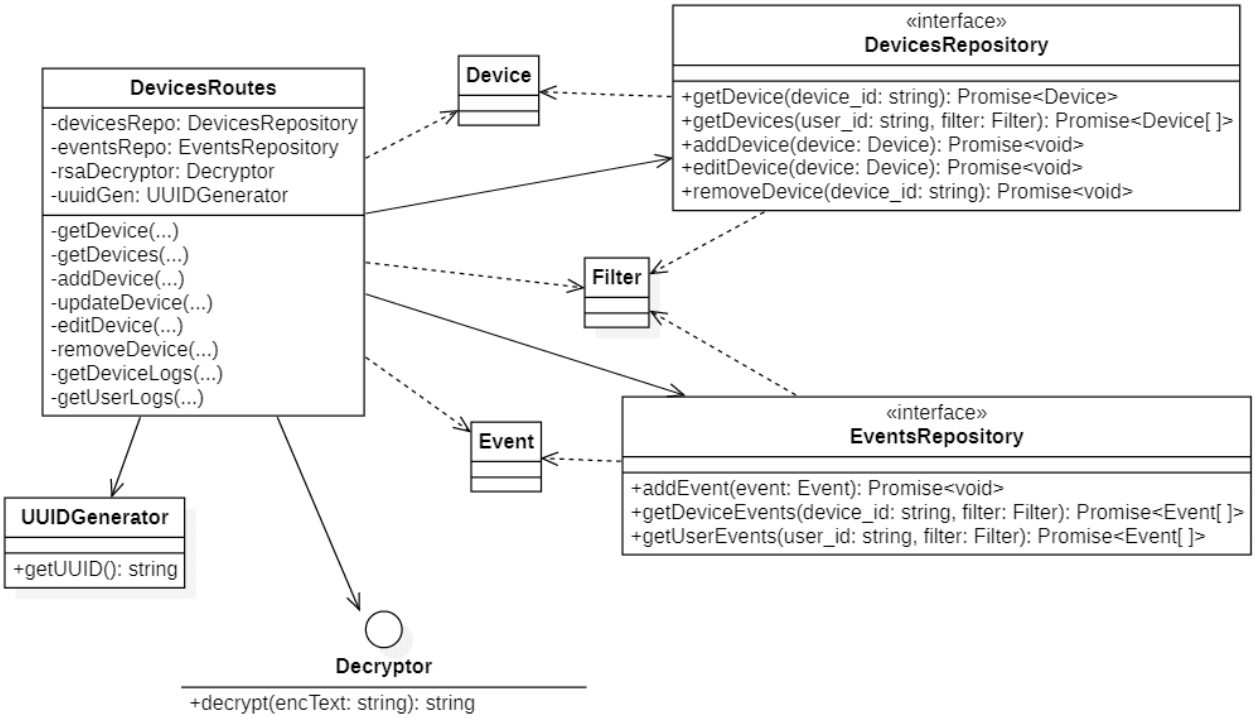
Relazioni

Di seguito sono rappresentate le relazioni fra le varie classi Routes, Repository e Service e i middleware. Per alcune entità non è presente un diagramma perché non presentano relazioni significative.

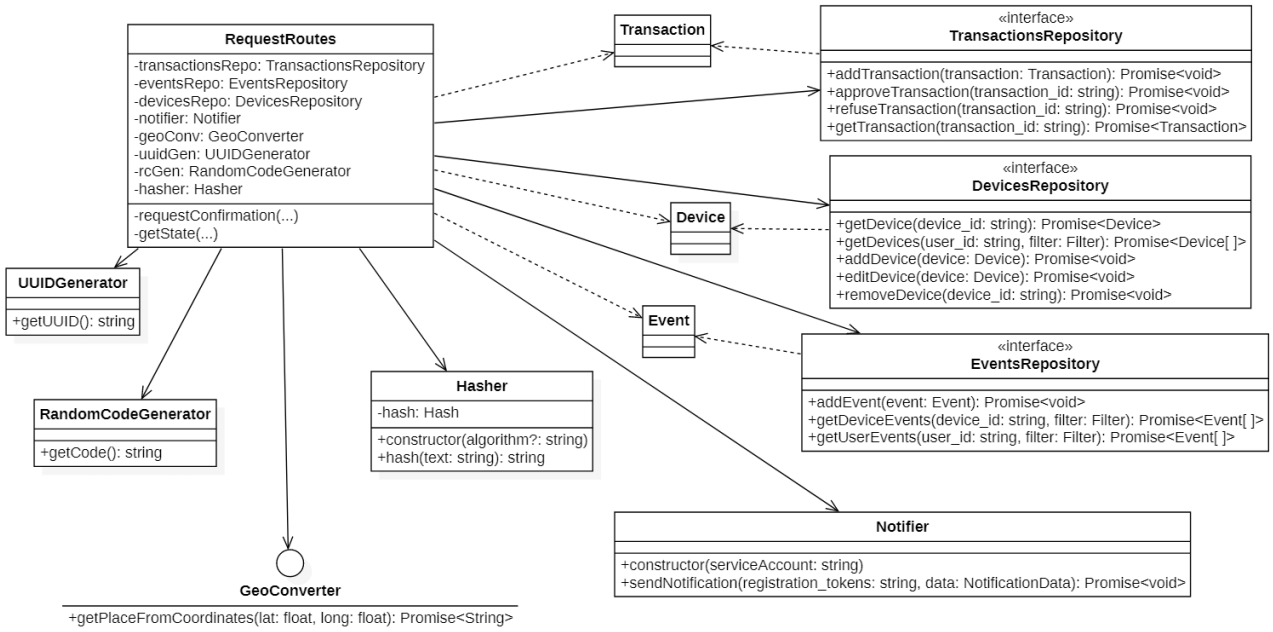
MetadataRoutes



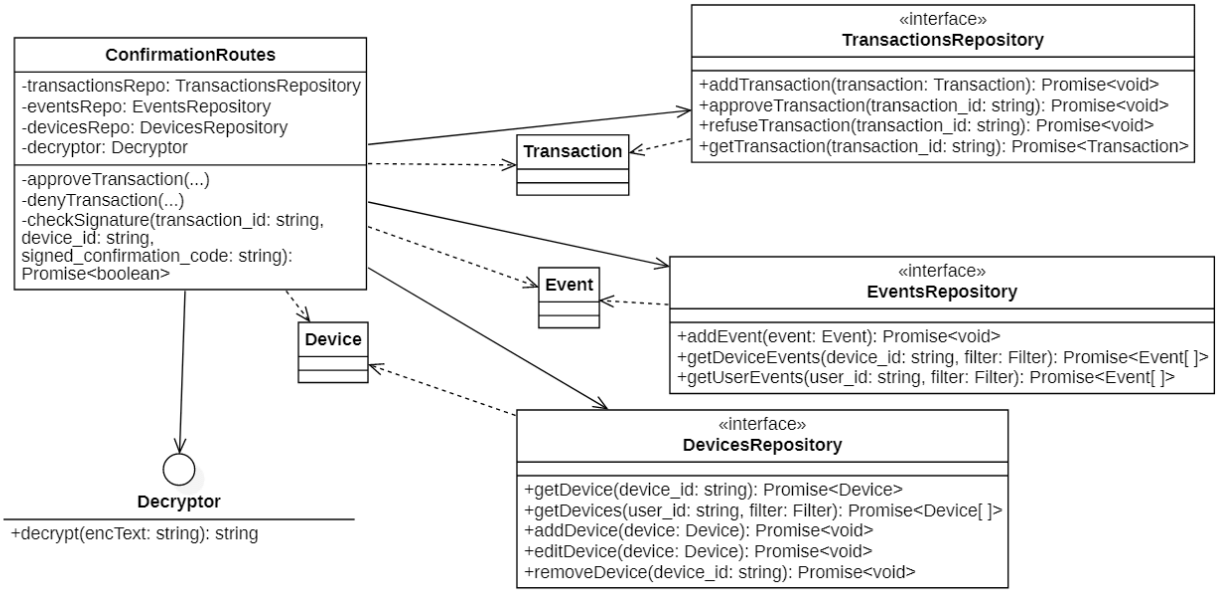
DevicesRoutes



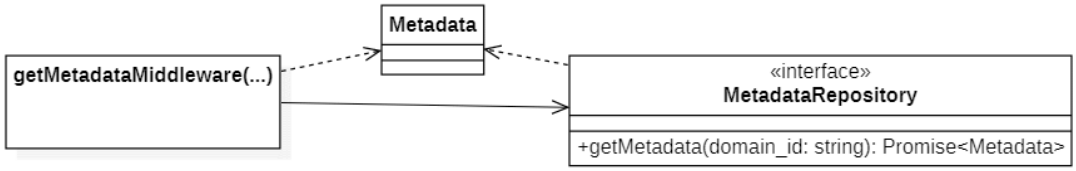
RequestRoutes



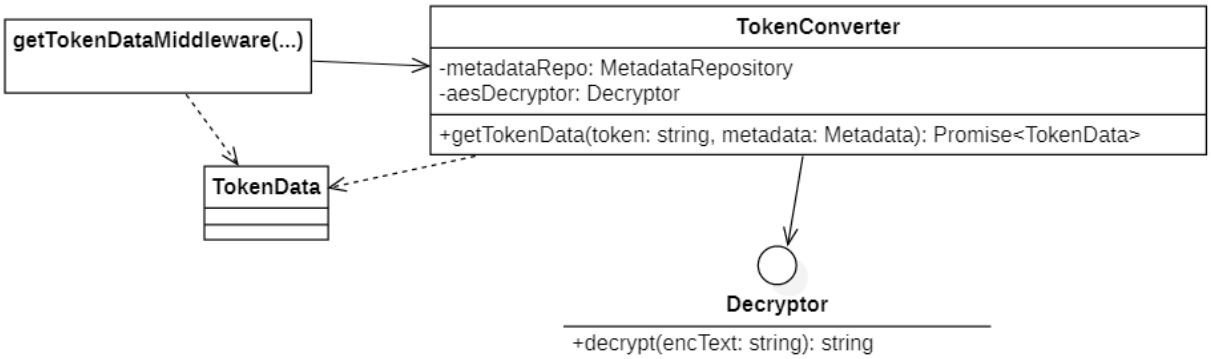
ConfirmationRoutes



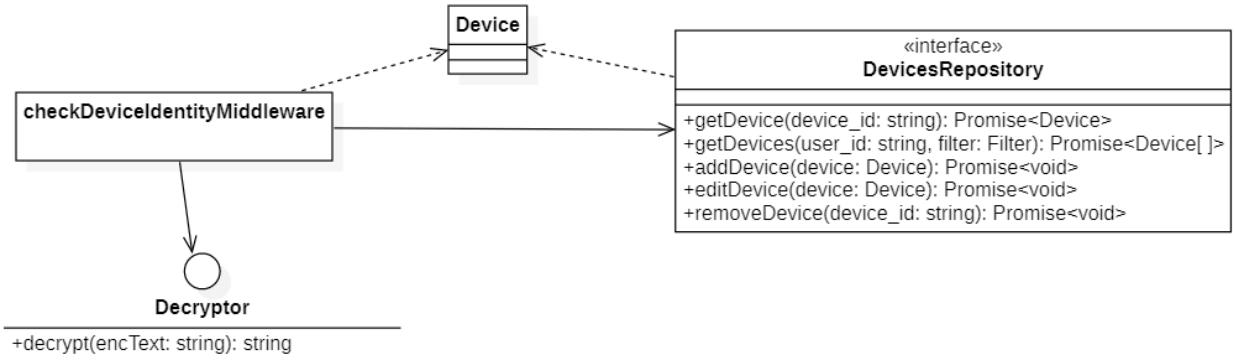
getMetadataMiddleware(...)



getTokenDataMiddleware(...)



checkDeviceIdentityMiddleware(...)



Funzionamento

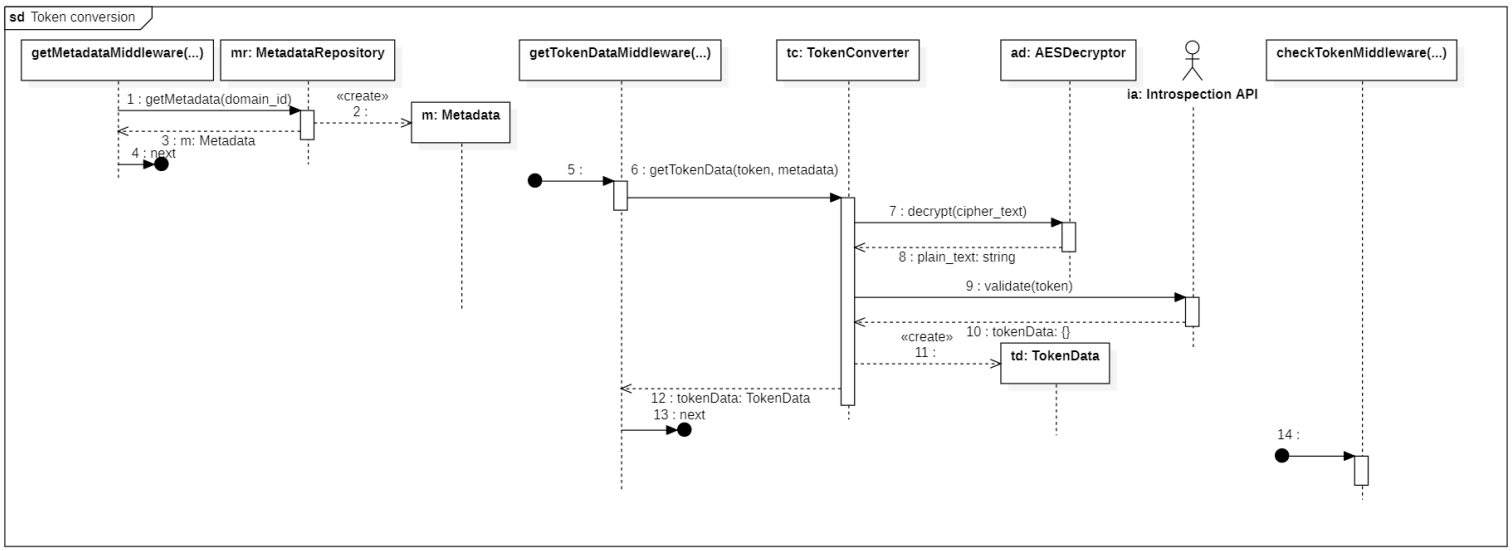
Le chiamate al server vengono processate a partire dalle Middleware e dalle Routes e in ogni funzione per cui passano sono elaborate attraverso i Services, mentre i dati vengono recuperati o memorizzati attraverso le Repository.

Per visualizzare questo schema di funzionamento, di seguito sono riportati due diagrammi che raffigurano rispettivamente

- la successione degli eventi che avviene durante la verifica di un token

- la successione degli eventi che avviene durante la richiesta da parte di un software utilizzatore della conferma di un'operazione.

Ai diagrammi è applicata una semplificazione: sono omesse le Promise. Al passaggio 2 del primo diagramma per esempio non viene restituito direttamente l'oggetto di tipo Metadata, ma viene restituita una Promise che una volta risolta fa proseguire il processo. Per semplificazione nei diagrammi tutte le risposte sono considerate dirette. Per vedere invece quali chiamate restituiscono effettivamente una Promise è possibile consultare i diagrammi delle classi più in alto.

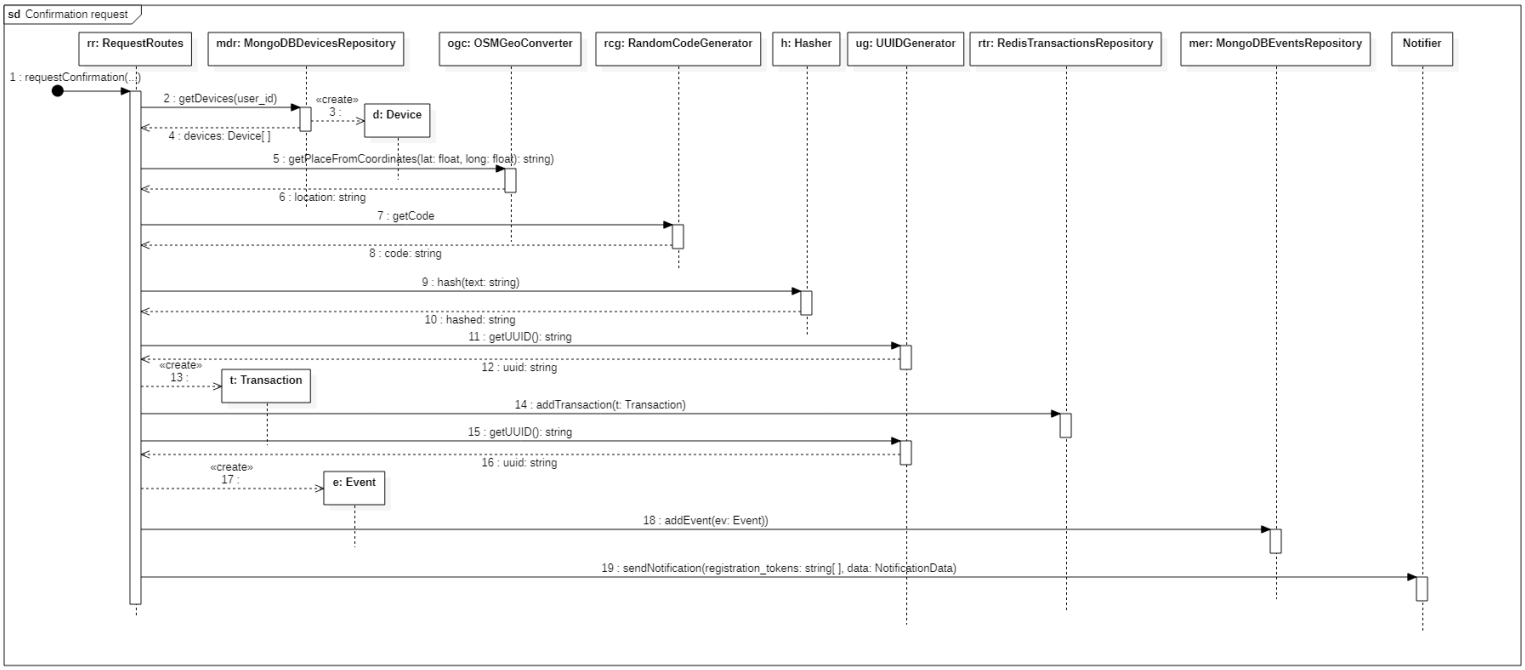


Per tutte le chiamate diverse da /metadata/:domain_id vi è un controllo del token. I tre middleware coinvolti in esso sono getMetadataMiddleware, getTokenMiddleware e checkTokenMiddleware.

Il primo si occupa di ottenere, dalla repository apposita, i metadati del dominio indicato nella richiesta e di salvarli nell'oggetto di risposta.

Il secondo legge il token provvisto nella richiesta e i metadati salvati dal precedente middleware. Comunica poi questi due dati al servizio TokenConverter che, comunicanzo con l'introspection API dedicata ottiene i dati del token e li restituisce al middleware. A questo punto anche questi dati vengono memorizzati nell'oggetto di risposta.

Il terzo ottiene i dati del token salvati e li usa per verificare se l'utente è autorizzato ad accedere alle risorse richieste.



La funzione requestConfirmation serve per inviare ai dispositivi di un utente una richiesta di conferma di un'operazione, sotto forma di notifica. Prima di inviare la notifica è necessario:
- individuare i dispositivi dell'utente e in particolare i loro registration_token
- convertire le eventuali coordinate ricevute in una località
- memorizzare l'evento di richiesta con un UUID
- memorizzare la transazione con un UUID
- creare un oggetto NotificationData con i dati da inviare, compreso il codice che il dispositivo dovrà reinviare firmato. Questo codice viene generato direttamente da requestConfirmation(), senza fare affidamento ad una classe apposita.

Successivamente è possibile inviare la notifica.

Inversion of Control

La gestione delle dipendenze tra le classi viene affidata al modulo Inversify, che applica il principio di design dell'inversion of control. In questo modo le classi non si devono occupare di costruire o procurarsi gli oggetti dai quali dipendono, ma essi sono forniti direttamente da Inversify.

Database

I database utilizzati dovranno memorizzare i tipi di dati definiti nella sezione Repositories. In particolare in MongoDB saranno create le seguenti collezioni.

Devices

```
{
   "_id": “”,
   “user_id”: “”,
   "public_key": "",
   “registration_token”: “”
}
```

Events

```
{
   "_id": “”,
   “user_id”: “”,
   “device_id”: “”,
   “type”: “”,
   “timestamp”: “”,
   “transaction_id”: “”,
   “coordinates”: {
      “latitude”: “”,
      “longitude”: “”
   },
   “location”: “”,
   “extra_info”: {
      “field_name”: “”
   }
}
```

I campi id rappresentano rispettivamente device_id e transaction_id, ma il nome è diverso perché prende il posto dell’identificativo utilizzato da MongoDB.

Negli oggetti di Events sono presenti dei campi annidati, ovvero i campi coordinates e extra_info, che contengono a loro volta dei campi. Queste è un possibilità offerta da MongoDB.

Gli oggetti di tipo Transaction saranno invece memorizzati in un database Redis in strutture dati Hash. In Redis non è possibile salvare campi dati annidati, quindi, visto che non si scende mai sotto il secondo livello di profondità, tutti i singoli dati saranno salvati su un singolo livello mantenendo l’informazione che consente, in lettura, di ricostruire la struttura dei livelli originale.

Gli oggetti di tipo Metadata sono memorizzati su un database MongoDB già esistente.