MFA CORE

PROGETTAZIONE DI DETTAGLIO

PREMESSE.....2
Errori.....2
Inject.....3
SERVER.....3
ROUTES.....4
MetadataRoutes.....4
DevicesRoutes.....4
RequestRoutes.....7
ConfirmationRequest.....10
MIDDLEWARE.....13
GetMetadataMiddleware(...).....13
GetTokenDataMiddleware(...).....13
CheckTokenMiddleware(...).....14
CheckClientPermissionMiddleware(...).....14
CheckDeviceIdentityMiddleware(...).....15
ErrorHandlerMiddleware(...).....16
SERVICES.....16
TokenConverter.....16
OSMGeoConverter.....16
Notifier.....17
AESDecryptor.....17
RSADecryptor.....18
Hasher.....18
UUIDGenerator.....19
RandomCodeGenerator.....19
UTILS.....19
CheckScope(...).....19
StringifyNestedFields(...).....20
UnstringifyNestedFields(...).....20
RequestToFilter(...).....20
ApplyQueryAndFilter(...).....21
REPOSITORIES.....21
MongoDBMetadataRepository.....21
MongoDBDevicesRepository.....22
RedisTransactionsRepository.....23
MongoDBEventsRepository.....25

PREMESSE

Di seguito si troverà lo pseudocodice dei metodi coinvolti nel funzionamento del server. Ogni metodo, ad esclusione dei costruttori, è preceduto dall'indicazione del suo ruolo e, dove ritenuto necessario, anche da un diagramma di flusso che ne descrive il funzionamento. I metodi sono divisi per package e per classe e all'inizio di ogni classe si troveranno i suoi attributi e gli eventuali elementi importati.

Errori

Per alleggerire lo pseudocodice è stata omessa la gestione degli errori interni. Essa può essere infatti spiegata direttamente in questa sezione.
Tutte le funzioni racchiudono il loro corpo in un blocco try seguito da un catch.
Le funzioni delle classi Routes e le funzioni middleware gestiscono l'errore inoltrandono con la funzione return next(err).
Le funzioni delle altre classi invece inoltrano l'errore al chiamante usando reject(err).
In caso di errore quindi il messaggio raggiunge in ogni caso l'ultimo middleware ovvero errorHandlerMiddleware. Questo restituisce all'utente una risposta con codice 500 e testo "Internal error".
Per chiarire questa struttura seguono due esempi:

Questa funzione legge e restituisce lo "status" dai dati di una transazione

```
async getState(req: Request, res: Response, next: NextFunction) {
    try{
        transaction_id = req.params.transaction_id
        transaction = await this.transactionsRepo.getTransaction(transaction_id)
        if(transaction){
            return res.json({status: transaction.status})
        }else{
            return res.json(error: {'The specified transaction doesn\'t exist'})
        }
    } catch (err) {
        return next(err)
    }
}
```

Questa funzione converte le coordinate ricevute in una stringa rappresentante un indirizzo

```
getPlaceFromCoordinates(lat: float, lon: float): Promise<String> {
    try{
        return new Promise<String>(async (resolve,reject) => {
            response = await axios(
            {
                method: 'get',
                url: this.url,
                params: {
                    lat: lat,
                    lon: lon
                }
            })
            return resolve(response.data.address)
        })
    }catch (err) {
```

```
        return reject(err)
    }
}
```

Inject

Sono state omesse, tranne che nei middleware, le indicazioni di quando un oggetto viene iniettato da inversify.
In generale, ogni volta che una classe ha tra i suoi attributi una repository o un service, quest'ultimo viene provvisto da inversify.

SERVER

```
app: Express
```

```
host: string
```

```
port: number
```

```
constructor(host?: string, port?: number) {
    this.host = host || *legge host da un file di configurazione*
                || 'localhost'
    this.port = port || *legge la porta da un file di configurazione* || 8000
    this.app = Express()
}
```

Imposta nel server il middleware specificato. E' possibile specificare l'url e il metodo sui quali il middleware dovrà ascoltare. Sono accettatti tutti i metodi presenti alla pagina http://expressjs.com/en/5x/api.html#routing-methods.

```
addMiddleware(middleware: Function, url?: string, method?: string) {
    if(url){
        switch(method){
            case 'get':
                this.app.get(url,middleware)
                break
            case 'post':
                this.app.post(url, middleware)
                break
            ...
            default:
                this.app.use(url, middleware)
        }
        return
    }
    this.app.use(middleware)
    return
}
```

Imposta nel server il router dell'oggetto Route provvisto

```
addRoute(route: Route) {
    this.app.use(route.router)
}
```

Mette il server, con tutte le sue routes e middleware, in ascolto di richieste

```
listen() {
    this.app.listen(this.port, this.host)
}
```

ROUTES

MetadataRoutes

```
metadataRepo: MetadataRepository
```

```
constructor() {
    this.basePath = '/metadata/:domain_id'
    this.router = Router()

    this.router.get(this.basePath, this.getMetadata)
}
```

Gestisce la richiesta di metadata. Per ottenerli si affida alla repository preposta

```
async getMetadata(req: Request, res: Response, next: NextFunction) {
    domain_id = req.params.domain_id
    met = await this.metadataRepo.getMetadata(domain_id)
    if(met){
        return res.json({url: met.metadata_url})
    }else{
        return res.json({error: 'Metadata not found'})
    }
}
```

DevicesRoutes

```
devicesRepo: DevicesRepository
```

```
eventsRepo: EventsRepository
```

```
uuidGen: UUIDGenerator
```

```
constructor() {
    this.basePath = '/user/:user_id'
    this.router = Router()

    this.router.get(this.basePath + '/device/:device_id', this.getDevice)
    this.router.get(this.basePath + '/devices', this.getDevices)
    this.router.post(this.basePath + '/device/', this.addDevice)
    this.router.put(this.basePath + '/device/:device_id', this.updateDevice)
    this.router.patch(this.basePath + '/device/:device_id', this.editDevice)
    this.router.delete(this.basePath + '/device/:device_id',
                       this.removeDevice)
    this.router.get(this.basePath + '/device/:device_id/logs',
                       this.getDeviceEvents)
    this.router.get(this.basePath + '/logs', this.getUserEvents)
}
```

Gestisce la richiesta dei dati di un dispositivo, richiedendolo semplicemente alla repository preposta

```
async getDevice(req: Request, res: Response, next: NextFunction) {
    device_id = req.params.device_id
    dev = await this.devicesRepo.getDevice(device_id)
    if(dev){
        return res.json(dev)
    }else{
        return res.json({error: 'Device not found'})
    }
}
```

Gestisce la richiesta dei dati dei dispositivi, richiedendoli semplicemente alla repository preposta

```
async getDevices(req: Request, res: Response, next: NextFunction) {
    user_id = req.params.device_id
    filter = requestToFilter(req)
    devs = await this.devicesRepo.getDevices(user_id, filter)
    if(devs){
        return res.json(devs)
    }else{
        return res.json({error: 'User not found'})
    }
}
```

Gestisce la richiesta di aggiunta di un dispositivo, inoltrando i dati ricevuti alla repository dedicata. Inoltre memorizza l'evento

```
async addDevice(req: Request, res: Response, next: NextFunction) {
    user_id = req.params.user_id
    dUuid = this.uuidGen.getUUID()
    newDev: Device = {
        _id: dUuid,
        user_id: user_id,
        *leggi dati dal body della chiamata*
    }
    await this.devicesRepo.addDevice(newDev)
    eUuid = this.uuidGen.getUUID()
    event: Event = {
        _id: uuid,
        user_id: user_id,
```

```
        device_id: dUuid,
        type: 'device_added',
        timestamp: new Date(),
    }
    await this.eventsRepo.addEvent(event)
    return res.json({result: 'Added successfully'})
}
```

Sovrascrive il dispositivo indicato con i nuovi dati forniti, inoltrandoli alla repository dedicata

```
async updateDevice(req: Request, res: Response, next: NextFunction) {
    device_id = req.params.device_id
    device: Device = req.body
    await this.devicesRepo.editDevice(device_id, device)
    return res.json({result: 'Updated successfully'})
}
```

Modifica il dispositivo specificato solo nei campi forniti, inoltrandoli alla repository dedicata

```
async editDevice(req: Request, res: Response, next: NextFunction)  {
    device_id = req.params.device_id
    device: Partial = req.body
    await this.devicesRepo.editDevice(device_id, device)
    return res.json({result: 'Updated successfully'})
}
```

Rimuove il dispositivo specificato attraverso l'azione della repository dedicata. Inoltre memorizza l'evento

```
async removeDevice(req: Request, res: Response, next: NextFunction) {
    device_id = req.params.device_id
    user_id = req.params.user_id
    await this.devicesRepo.removeDevice(device_id)
    eUuid = this.uuidGen.getUUID()
    event: Event = {
        _id: uuid,
        user_id: user_id,
        device_id: device_id,
        type: 'device_removed',
        timestamp: new Date(),
    }
    await this.eventsRepo.addEvent(event)
    return res.json({result: 'Removed successfully'})
}
```

Restituisce gli eventi del dispositivo specificato, prendendoli dalla repository dedicata

```
async getDeviceLogs(req: Request, res: Response, next: NextFunction) {
    device_id = req.params.device_id

    filter = requestToFilter(req)

    events = await this.eventsRepo.getDeviceEvents(device_id, filter)
    if(events.length == 0){
        return res.json({error: 'The specified device is not registered or
                        has no events'})
    }else{
        return res.json(events)
    }
```

```
}
```

Restituisce gli eventi dell'utente specificato, prendendoli dalla repository dedicata

```
async getUserLogs(req: Request, res: Response, next: NextFunction) {
    user_id = req.params.user_id
    filter = requestToFilter(req)

    events = await this.eventsRepo.getUserEvents(user_id, filter)
    if(events.length == 0){
        return res.json({error: 'The specified user is not registered or
                        has no events'})
    }else{
        return res.json(events)
    }
}
```

RequestRoutes

```
transactionsRepo: TransactionsRepository
```

```
eventsRepo: EventsRepository
```

```
notifier: Notifier
```

```
geoConv: GeoConverter
```

```
uuidGen: UUIDGenerator
```

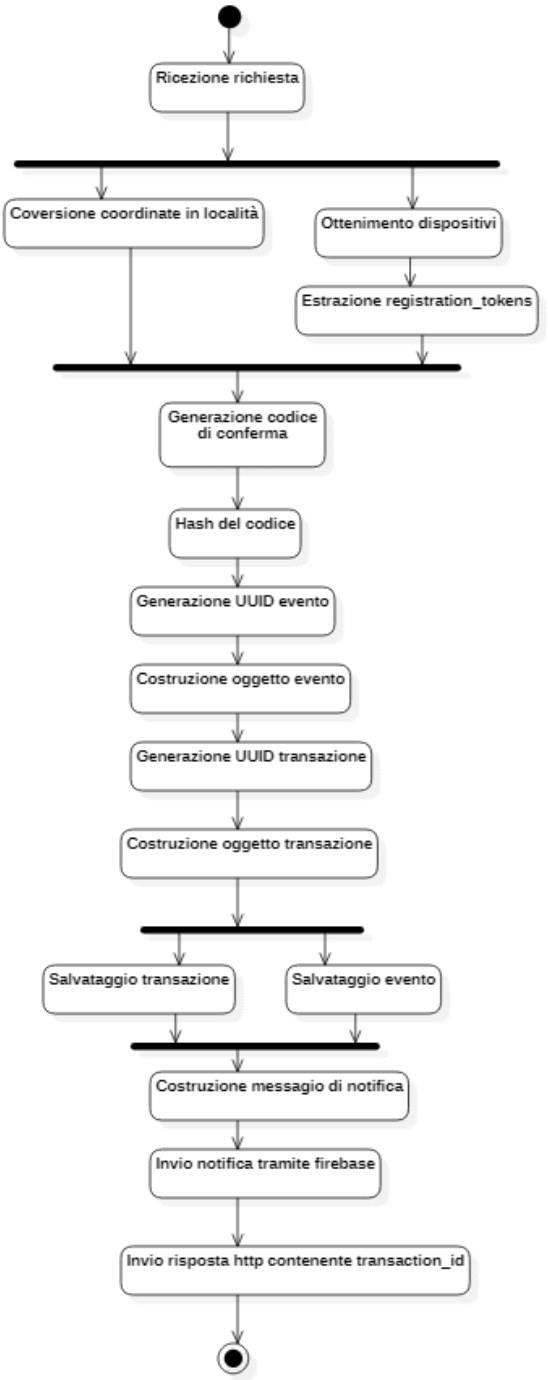
```
codeGenerator: RandomCodeGenerator
```

```
hasher: Hasher
```

```
constructor() {
    this.basePath = '/user/:user_id'
    this.router = Router()

    this.router.post(this.basePath + '/target_user/:target_user_id',
                     this.requestConfirmation)
    this.router.get(this.basePath + '/transaction/:transaction_id/state',
                     this.getState)
}
```

Ottiene i dispositivi dell'utente specificato e i loro registration_token. Poi invia ad ognuno di essi la notifica per la richiesta di approvazione, contenente un codice generato casualmente. Il codice viene inviato ai dispositivi senza modifiche ma viene memorizzato dal server solo dopo avervi applicato un hash. Ciò riduce i pericoli di un eventuale accesso non autorizzato al database Redis.




```
async requestConfirmation(req: Request, res: Response, next: NextFunction) {
    target_id = req.params.target_user_id
    data = req.body
    coordinates = data.coordinates
    requester_id = req.params.user_id

    promisesToExecute = []
    // ottenimento registration_tokens
    p1 = this.devicesRepo.getDevices(target_id).then((devs) => {
        registration_tokens = []
        devs.forEach((item, index) => {
            registration_tokens.append(item.registration_token)
        })
```

```
        return registration_tokens
    })
    promisesToExecute.push(p1)
    if(coordinates){
        // conversione coordiante in località
        p2 = this.geoConv(coordintes).then( (location) => {
            return location
        })
    }
    promisesToExecute.push(p2)

    results = await Promise.all(promisesToExecute)
    registration_tokens = results[0]
    location = results[1]

    if(registration_tokens.length == 0){
        res.json({error: 'The target user is not registered or
                    has no devices associated'})
    }

    // generazione e hash del codice di conferma
    confirmationCode = this.codeGenerator.getCode()
    hashedCode = this.hasher.hash(confirmationCode)

    // creazione e salvataggio evento e transazione
    eUuid = this.uuidGen.getUUID()
    event: Event = {
        _id: eUuid,
        user_id: target_id,
        type: 'request',
        timestamp: new Date(),
        transaction_id: eUuid,
        coordinates: coordinates,
        location: location,
        extra_info: data.extra_info
    }
    tUuid = this.uuidGen.getUUID()
    ttl = *prendi il tempo di attesa standard da un file di configurazione*
    transaction: Transaction = {
        transaction_id: tUuid,
        user_id: target_id,
        requester_id: requester_id,
        confirmation_code: hashedCode,
        status: 'pending',
        ttl: ttl,
        coordinates: coordinates,
        location: location,
        extra_info: data.extra_info
    }
    results = await Promise.all([
        this.transactionsRepo.addTransaction(transaction),
        this.eventsRepo.addEvent(event)
        ])

    // costruzione e invio della notifica
    toSendData: NotificationData = {
        transaction_id: uUuid,
        confirmation_code: confirmationCode,
        location: location,
        coordinates: data.coordinates,
        extra_info: data.extra_info
    }

    await this.notifier.sendNotification(registration_tokens, toSendData)

    // risposta all'utente
    return res.json({transaction_id: transaction.id})
}
```

Legge e restituisce lo "status" dai dati di una transazione

```
async getState(req: Request, res: Response, next: NextFunction) {
    transaction_id = req.params.transaction_id
    transaction = await this.transactionsRepo.getTransaction(transaction_id)
    if(transaction){
        return res.json({status: transaction.status})
    }else{
        return res.json({error: 'The specified transaction doesn\'t exist'})
    }
}
```

ConfirmationRequest

```
transactionsRepo: TransactionsRepository
```

```
eventsRepo: EventsRepository
```

```
devicesRepo: DevicesRepository
```

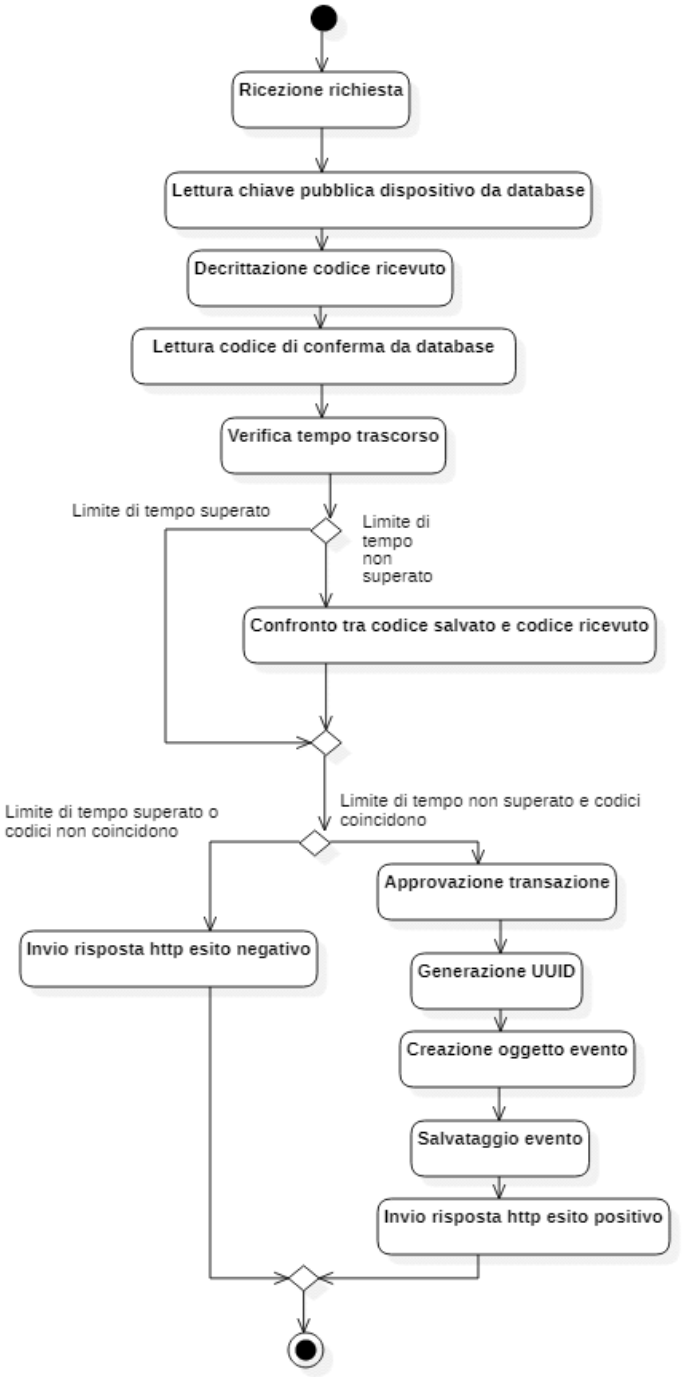
```
rsaDecryptor: Decryptor
```

```
uuidGen: UUIDGenerator
```

```
constructor() {
    this.basePath =
    '/user/:user_id/device/:device_id/transaction/:transaction_id'
    this.router = Router()

    this.router.get(this.basePath + '/approve', this.approveTransaction)
    this.router.get(this.basePath + '/deny', this.refuseTransaction)
}
```

Verifica la firma del dispositivo e se l'esito è positivo e il tempo non è scaduto approva la transazione specificata e memorizza l'evento sul database.
Il primo punto del flusso è eseguito dalla funzione approveTransaction, poi il controllo passa a checkSignatureAndTime e poi, a partire dalla seconda biforcazione, ritorna ad approveTransaction.




```
async approveTransaction(req: Request, res: Response, next: NextFunction) {
    transaction_id = req.params.transaction_id
    device_id = req.query.device_id
    user_id = req.params.user_id
    signed_confirmation_code = req.query.signed_confirmation_code
    verified = await this.checkSignatureAndTime(transaction_id,
                                                device_id,
                                                signed_confirmation_code)
    if(verified){
        result = await this.transactionsRepo
                           .approveTransaction(transaction_id)
        transaction = await this.transactionsRepo
                                .getTransaction(transaction_id)
```

```
        uuid = uuidGen.getUUID();
        event: Event = {
            _id: uuid,
            user_id: user_id,
            device_id: device_id,
            type: 'confirmation',
            timestamp: new Date(),
            transaction_id: transaction_id,
            coordinates = transaction.coordinates,
            location = transaction.location,
            extra_info = transaction.extra_info
        }
        await this.eventsRepo.addEvent(event);

        return res.json({result: 'Approval succeded'})
    }else{
        return res.json({error: 'Signature not valid or time exceeded'})
    }
}
```

Verifica la firma del dispositivo e se l'esito è positivo e il tempo non è scaduto rifiuta la transazione specificata e memorizza l'evento sul database.

```
async denyTransaction(req: Request, res: Response, next: NextFunction) {
    transaction_id = req.params.transaction_id
    device_id = req.query.device_id
    signed_confirmation_code = req.query.signed_confirmation_code
    verified = await this.checkSignatureAndTime(transaction_id,
                                                device_id,
                                                signed_confirmation_code)
    if(verified){
        result = await this.transactionsRepo.refuseTransaction(transaction_id)
        transaction = await this.transactionsRepo
                                .getTransaction(transaction_id)
        uuid = uuidGen.getUUID();
        event: Event = {
            _id: uuid,
            user_id: user_id,
            device_id: device_id,
            type: 'denial',
            timestamp: new Date(),
            transaction_id: transaction_id,
            coordinates = transaction.coordinates,
            location = transaction.location,
            extra_info = transaction.extra_info
        }
        await this.eventsRepo.addEvent(event);
        return res.json({result: 'Refusal succeded'})
    }else{
        return res.json({error: 'Signature not valid or time exceeded'})
    }
}
```

Verifica i dati provvisti nella chiamata per certificare il dispositivo mittente. Oltre alla correttezza della firma viene verificato se il termine per l'approvazione non è ancora scaduto.

```
checkSignatureAndTime(transaction_id: string,
                      device_id: string,
                      signed_confirmation_code): Promise<boolean> {
    return new Promise<boolean>(async (resolve, reject) => {
        transaction = await this.transactionsRepo.
                              getTransaction(transaction_id)
        ttl = transaction.ttl
```

```
        min_ttl = *leggi il ttl minimo accettabile
                    da un file di configurazione*
        // verifica limite di tempo
        if(ttl>=min_ttl){
            conf_code = transaction.confirmationCode
            device = await this.devicesRepo.getDevice(device_id)
            pub_key = device.public_key
            dec = this.rsaDecryptor.decrypt(signed_confirmation_code)
            // verifica coincidenza codici
            if(dec == conf_code){
                return resolve(true)
            }else{
                return resolve(false)
            }
        }else{
            return resolve(false)
        }
    })
}
```

MIDDLEWARE

GetMetadataMiddleware(...)

Ottiene i metadata corrispondenti al domain_id specificato nella richiesta e li salva nell'oggetto res in modo che possano essere letti da funzioni successive

```
async getMetadataMiddleware(req: Request, res: Response, next: NextFunction) {
    domain_id = req.params.domain_id
    metadataRepo = @inject MetadataRepository()
    met = await metadataRepo.getMetadata(domain_id)
    if(met){
        // se vengono ottenuti i metadata
        res.locals.metadata = met
        return next()
    }else{
        return res.json({error: 'Domain metadata not found'})
    }
}
```

GetTokenDataMiddleware(...)

Ottiene i dati del token e li salva nell'oggetto res in modo che possano essere letti da funzioni successive

```
async getTokenDataMiddleware(req: Request, res: Response, next: NextFunction) {
    metadata = res.locals.metadata

    token = req.headers.get("Authorization")
    // questo oggetto viene iniettato da inversify
    tokenConv = @inject TokenConverter()
    tokenData = await tokenConv.getTokenData(token, metadata)
    if(tokenData){
        // se vengono ottenuti i dati del token
        res.locals.tokenData = tokenData
        return next()
```

```
    }else{
        return res.json({error: 'Unable to get token data'})
    }
}
```

CheckTokenMiddleware(...)

Verifica che il token provvisto sia valido, sia stato richiesto dall'utente specificato, e fornisca i permessi necessari per accedere all'url richiesto

```
async checkTokenMiddleware(req: Request, res: Response, next: NextFunction) {
    tokenData = res.locals.tokenData
    are_scopes_correct = checkScope(req.url, tokenData.scopes)
    if(tokenData
       && tokenData.active == true
       && tokenData.sub == req.params.user_id
       && are_scopes_correct) {
        return next()
    }else{
        if(!tokenData){
            return res.status(401).json({error: 'Unable to get token data'})
        }else if(!tokenData.active){
            return res.status(401).json({error: 'Token not valid'})
        }else if(tokenData.sub != req.params.user_id){
            return res.status(401).json({error: 'Token owner and specified user
                                        are not the same'})
        }else if(!are_scopes_correct){
            return res.status(401).json({error: 'Scopes not correct'})
        }
    }
}
```

CheckClientPermissionMiddleware(...)

Verifica che il client che ha richiesto il token sia di un tipo ammesso per questa richiesta. Se è così salva nell'oggetto di risposta il tipo di client

```
async checkClientPermissionMiddleware(req: Request,
                                      res: Response,
                                      next: NextFunction) {
    tokenData = res.locals.tokenData
    metaData = res.locals.metaData
    if(!tokenData){
        return res.status(401).json('Token data not available')
    }
    if(!tokenData){
        return res.status(401).json('Domain metadata not available')
    }

    client_id = tokenData.client_id
    // Da un file viene letta una mappa del tipo
    // url: string -> [dispositivi permessi]
    // I dispositivi permessi possono essere app, webapp e software
    permissions = *leggi da file di configurazione*
    app_id = metaData.app.client_id
    webapp_id = metaData.webapp.client_id
    software_id = metaData.software.client_id
    client_type = ''
    if(client_id == app_id){
```

```
        client_type = 'app'
    }else if(client_id == webapp_id){
        client_type = 'webapp'
    }else if(client_id == software_id){
        client_type == 'software'
    }
    currentUrl = req.url
    if(!permissions[currentUrl].contains(client_type)){
        return res.status(401).json({error: 'Device type not permitted'})
    }
    res.locals.verified_client_type = client_type
}
```

CheckDeviceIdentityMiddleware(...)

Verifica che, se la richiesta è partita da una app mobile, il dispositivo che sta eseguendo l'azione sia lo stesso del dispositivo che la sta subendo

```
async checkDeviceIdentityMiddleware(req: Request,
                                    res: Response,
                                    next: NextFunction) {
    // controllo che sia stata attuata la verifica del tipo di client
    if(res.locals.verified_client_type){
        if(res.locals.verified_client_type == 'app'){
            // se la richiesta viene dall'app verifico il dispositivo
            claimed_id = req.params.device_id
            signed_id = req.query.signed_device_id
            devicesRepo = @inject DevicesRepository()
            deviceData = await devicesRepo.getDevice(claimed_id)
            pub_key = deviceData.public_key
            rsaDecryptor = @inject RSADecryptor()
            decrypted = rsaDecryptor.decrypt(signed_id)
            if(claimed_id && decrypted == claimed_id){
                return next()
            }else{
                return res.status(401).json({error: 'The client which requested the acti
                                                    is not the same as the target device'})
            }
        }
    }else{
        return res.status(401).json({error: 'The device type is not verified'})
    }

}
```

ErrorHandlerMiddleware(...)

Questo middleware viene utilizzato per segnalare che c'è stato un errore interno al server. Può facilmente essere modificato per dare più informazioni. Per esempio mostrando il testo dell'errore e lo stack trace

```
async errorHandlerMiddleware(err: Error,
                             req: Request,
                             res: Response,
                             next: NextFunction) {
    return res.status(500).json({error: 'Internal error'})
}
```

SERVICES

TokenConverter

```
aesDecryptor: Decryptor
```

Restituisce i dati legati al token. Ha bisogno dei metadati con i quali comunicare coll'introspection API

```
getTokenData(token: string, metadata: Metadata): Promise<TokenData> {
    return new Promise<TokenData>(async (resolve,reject) => {
        encClientSecret = metadata.core_credentials.client_secret
        decryptedClientSecret = this.aesDecryptor.decrypt(encClientSecret)
        auth = base64(matadata.core_credentials.client_id
                      + ':'
                      + decryptedClientSecret)
        tokenData = await axios({
                            method: 'post',
                            url: metadata.introspection_endpoint,
                            data: 'token='
                                  + token
                                  + '&token_type_hint=access_token',
                            headers: {
                                'Authorization': auth
                            }
                        })
        return resolve(tokenData)
    })
}
```

OSMGeoConverter

```
import {axios} from 'axios'
```

```
url: string
```

```
constructor(url?: string) {
    this.url = url || *leggi da file di configurazione*
}
```

Converte, tramite una API pubblica, le coordinate ricevute in una stringa rappresentante un indirizzo

```
getPlaceFromCoordinates(lat: float, lon: float): Promise<String> {
    return new Promise<String>(async (resolve,reject) => {
        response = await axios(
        {
            method: 'get',
            url: url,
```

```
            params: {
                lat: lat,
                lon: lon
            }
        })
        return resolve(response.data.address)
    })
}
```

Notifier

```
import {admin}
```

from 'firebase-admin'

```
constructor(serviceAccount: string) {
    admin.initializeApp({
      credential: admin.credential.cert(serviceAccount)
    })
}
```

Invia il messaggio ai dispositivi identificati dai registration_token forniti. L'invio avviene tramite il backend di Firebase

```
sendNotification(registration_tokens: string[],
                 data: NotificationData): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        message = {
            data: data,
            tokens: registration_tokens
        }
        await admin.messaging().send(message)
        return resolve()
    })
}
```

AESDecryptor

```
decipher: Decipher
```

```
constructor(algorithm, key, iv) {
    _alg = algorithm || *leggi da file di configurazione*
    _key = key || *leggi da file di configurazione*
    _iv = iv || *leggi da file di configurazione*
    this.decipher = crypto.createDecipheriv(_alg,
                                            Buffer.from(_key),
                                            Buffer.from(_iv))
}
```

Restituisce il messaggio decrittato con la chiave e l'iv (initialization vector) forniti durante la costruzione

```
decrypt(encText: string): string {
    this.decipher.update(encText)
    decr = decipher.final()
    return decr.toString()
}
```

RSADecryptor

```
import {crypto} from 'crypto'
```

```
publicKey: string
```

```
constructor(publicKey: string){
    this.publicKey = publicKey
}
```

Restituisce il testo decrittato con la chiave precedentemente provvista al costruttore

```
decrypt(encText: string): string {
    dec = crypto.publicDecrypt(this.publicKey, encText)
    return dec.toString()
}
```

Hasher

```
import {crypto} from 'crypto'
```

```
hash: Hash
```

```
constructor(algorithm?: string) {
    this.hash = crypto.createHash(algorithm)
                || *leggi da file di configurazione*
                || 'sha256'
}
```

Restituisce il testo dopo avergli applicato l'algoritmo di hashing indicato durante la costruzione dell'oggetto

```
hash(text: string): string {
```

```
    this.hash.update(text)
    return this.hash.digest()
}
```

UUIDGenerator e RandomCodeGenerator sono separati perchè solo il secondo ha necessità di essere crittograficamente sicuro, ovvero randomico in un modo non prevedibile

UUIDGenerator

```
import {crypto} from 'crypto'
```

Genera e resituisce un UUID. Non necessariamente questo UUID deve essere randomico in un modo sicuro per l'uso crittografico

```
getUUID(): string {
    return crypto.randomUUID()
}
```

RandomCodeGenerator

```
import {crypto} from 'crypto'
```

Restituisce un codice randomico generato in modo che sia sicuro per l'uso crittografico

```
getCode(): string {
    return crypto.randomUUID()
}
```

UTILS

CheckScope(...)

Verifica che tra gli scope provvisti in token_scopes ci sia quello necessario per accedere all'url specificato

```
checkScope(url: string, token_scopes: string[]): true {
    // qui ogni regex rappresenta un formato di url e quindi un endpoint
    regexes = *ottieni una mappa del tipo
              (regex: regular expression -> scope: scope)
              da un file di configurazione*
    required_scope = null
    // certa il regex corrispondente all'url
    for entry in regexes{
        if(url.match(entry.regex)){
            required_scope = entry.scope
            break
        }
    }
    if(token_scopes.contains(required_scope)){
        return true
    }else{
        return false
    }
}
```

StringifyNestedFields(...)

Converte in stringhe gli oggetti annidati. In questo modo di ottiene un oggetto con un solo livello di profondità.

```
stringifyNestedFields(obj: any): any {
    converted_obj = obj;
    for(field in converted){
        converted_obj.field = JSON.stringify(converted_obj.field);
    }
    return converted_obj
}
```

UnstringifyNestedFields(...)

Se sono presenti campi che contengono la traduzione in stringhe di oggetti JSON, a questi campi viene assegnato l'oggetto JSON corrispondente

```
unstringifyNestedFields(obj: any): any {
    converted_obj = obj;
    for(field in converted){
        converted_obj.field = JSON.parse(converted_obj.field);
    }
    return converted_obj
}
```

RequestToFilter(...)

Restituisce un oggetto Filter con i filtri specificati nella richiesta req

```
requestToFilter(req: Request): Filter {
    // ordinamento
    order = req.query.order
    // campi
```

```
    fields = req.query.fields
    // tipo
    type = req.query.type
    // paginazione
    pageNum = req.query.pageNum
    pageElements = req.query.pageElements

    filter = new Filter(order,
                        type,
                        fields,
                        {pageNum: pageNum, pageElements: pageElements})
    return filter
}
```

ApplyQueryAndFilter(...)

Applica a collection la query specificata e, se presente, anche filter

```
async applyQueryAndFilter(collection: Collection, query: any, filter?: Filter): Promise<
    if(!filter){
        return await collection.find(query);
    }
    // tipo
    match = query
    if(filter.type){
        match.type = filter.type
    }
    // campi
    projection = {}
    if(filter.fields) {
        projection = {}
        for field in filter.fields{
            projection.field = true
        }
        if('_id' not in selected_fields){
            projection._id = false
        }
    }
    return await collection.find(match, projection)
                           .sort(filter.order)
                           .skip(filter.pagination.pageNum * filter.pagination.pageEleme
                           .limit(filter.pagination.pageElements)
}
```

REPOSITORIES

MongoDBMetadataRepository

client MongoClient

```
metadata: Collection
```

```
constructor(uri?: string, options?: any): {
    this.client = new MongoClient(uri
                                  || *leggi da file di configurazione*,
                                  options
                                  || *leggi da file di configurazione*)
    this.client.connect()
    database = this.client.db(*leggi da file di configurazione*
                              || 'mfa')
    this.metadata = database.collection(*leggi da file di configurazione*
                                        || 'metadata')
}
```

Restituisce i metadati corrispondenti al dominio specificato

```
getMetadata(domain_id: string): Promise<Metadata> {
    return new Promise<Metadata>(async (resolve,reject) => {
        met = await this.metadata.findOne( {domain_id: domain_id})
        return resolve(met)
    })

}
```

MongoDBDevicesRepository

```
client: MongoClient
```

```
devices: Collection
```

```
constructor(uri?: string, options?: any) {
    this.client = new MongoClient(uri
                                  || *leggi da configurazione*,
                                  options
                                  || *leggi da file di configurazione*)
    this.client.connect()

    database = this.client.db(*leggi da file di configurazione*
                              || 'mfa')
    this.devices = database.collection(*leggi da file di configurazione*
                                       || 'devices')
}
```

Restituisce i dati del dispoitivo specificato

```
getDevice(device_id: string): Promise<Device> {
    return new Promise<Device>(async (resolve,reject) => {
        dev = await this.devices.findOne( {device_id: _device_id})
        return resolve(dev)
    })
}
```

Restituisce i dati dei dispositivi specificati, filtrati secondo quanto indicato in filter

```
getDevices(user_id: string, filter?: Filter): Promise<Device[]> {
```

```
    return new Promise<Device[]>(async (resolve,reject) => {
        res = await applyQueryAndFilter(this.devices, {user_id: user_id}, filter)
        arr = await res.toArray()
        return resolve(arr)
    })
}
```

Memorizza il dispositivo provvisto

```
addDevice(device: Device): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        try{
            res = await this.devices.insertOne(device)
            return resolve(res)
        }catch(err) {
            return reject(err)
        }
    })
}
```

Modifica il dispositivo indicato secondo i campi indicati in device

```
editDevice(device_id: string, device: Partial): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        try {
            res = await this.devices.updateOne({_id: device_id},
                                               {$set: device})
            return resolve(res)
        }catch (err) {
            return reject(err)
        }
    })
}
```

Rimuove il dispositivo indicato

```
removeDevice(device_id: string): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        try {
            res = await this.devices.deleteOne({_id: device_id})
            return resolve(res)
        }catch (err) {
            return reject(err)
        }
    })
}
```

RedisTransactionsRepository

```
import Redis from 'ioredis';
```

```
redis: Redis
```

```
constructor(host?: string, port?: number, options?: any) {
    redis = new Redis( host ||
                       *leggi da file di configurazione*
                       || 'localhost',
                       port
                       || *leggi da file di configurazione*
                       || 6379,
                       options
                       || *leggi da file di configurazione*)
}
```

Memorizza nel database Redis la transazione provvista

```
addTransaction(transaction: Transaction): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        plain_transaction = stringifyNestedFields(transaction)
        delete plain_transaction._id
        delete plain_transaction.ttl
        result = await this.redis.hmset(transaction_id, plain_transaction)
        await this.redis.ttl(transaction_id, transaction.ttl)
        return resolve()
    })
}
```

Imposta lo stato della transaione specificata ad "approved"

```
approveTransaction(transaction_id: string): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        result = await this.redis.hset(transaction_id, 'status', 'approved')
        return resolve()
    })
}
```

Imposta lo stato della transazione specificata a "refused"

```
refuseTransaction(transaction_id: string): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        result = await this.redis.hset(transaction_id, 'status', 'refused')
        return resolve()
    })
}
```

Restituisce i dati della transazione specificata

```
getTransaction(transaction_id: string): Promise<Transaction> {
    return new Promise<Transaction>(async (resolve,reject) => {
        result = await this.redis.hgetall(transaction_id)

        if(Object.keys(result).length == 0 ){
            // se la transazione indicata non è presente
            return reject(new Error('Transaction not found'))
        }else{
```

```
            tra: Partial = unstringifyNestedFields(result)
            tra._id = transaction_id
            ttl = await this.redis.ttl(transaction_id)
            if(ttl == -2){
                return reject(new Error('Transaction not found'))
            }
            tra.ttl = ttl
            return resolve(tra)
        }
    })
}
```

MongoDBEventsRepository

```
client: MongoClient
```

```
events: any
```

```
constructor(uri?: string, options?: any) {
    this.client = new MongoClient(uri
                                  || *leggi da file di configurazione*,
                                  options
                                  || *leggi da file di configurazione*)
    this.client.connect()

    database = this.client.db(*leggi da file di configurazione*
                              || 'mfa')
    this.events = database.collection(*leggi da file di configurazione*
                                      || 'events')
}
```

Memorizza l'evento nel database MongoDB

```
addEvent(event: Event): Promise<void> {
    return new Promise<void>(async (resolve,reject) => {
        res = await this.events.insertOne(event)
        return resolve(res)
    })
}
```

Restituisce gli eventi rigurdanti l'utente specificato filtrati secondo quanto indicato da filter

```
getUserEvents(user_id: string, filter?: Filter): Promise<Event[]> {
    return new Promise<Event[]>(async (resolve,reject) => {
        res = await applyQueryAndFilter(this.events, {user_id: user_id}, filter)
        arr = await res.toArray()
        return resolve(arr)
    })
}
```

Restituisce gli eventi rigurdanti il dispositivo specificato filtrati secondo quanto indicato da filter

```
getDeviceEvents(device_id: string, filter?: Filter): Promise<Event[]> {
```

```
    return new Promise<Event[]>(async (resolve,reject) => {
        res = await applyQueryAndFilter(this.events, {device_id: device_id}, filter)
        arr = await res.toArray()
        return resolve(arr)
    })
}
```