POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Definizione, sviluppo e validazione di un sistema E2E per l'upgrade firmware remoto di dispositivi IoT

Relatore
prof. Guido Albertengo

Studente
Francesco Vernacchia
matricola: 257427

Supervisore aziendale
Santer Reply Spa
dott. ing. Marco Mastropasqua

Anno accademico 2020-2021

Sommario

Quest'opera di tesi si articola nel modo seguente: nel **capitolo 1** si introdurrà lo stato dell'arte dell'IoT e della tecnologia cloud e l'esigenza che si ha, a livello aziendale, di avere dispositivi sempre aggiornati e con alti livelli di performance. Nel **capitolo 2** si presenteranno i componenti e i servizi che sono stati studiati per rendere possibile la realizzazione di questo progetto di tesi. Nel **capitolo 3** si presenterà tutto il materiale elaborato per costruire le applicazioni client e server necessarie alla simulazione dell'intero sistema *Firmware Over The Air*. Nel **capitolo 4** si concluderà mostrando delle idee per migliorie future da apportare per l'architettura elaborata.

Ringraziamenti

La crescita umana ed intellettuale è un fatto strettamente personale e come tale difficilmente può essere intestato ad altri; volevo però ringraziare il Politecnico di Torino ed i suoi docenti che nel corso di questi anni mi hanno insegnato, oltre alla professione dell'Ingegnere, anche il pensiero scientifico ed analitico che mi ha permesso di diventare una persona migliore dal punto di vista dell'approccio al mondo ed i suoi problemi. Vorrei ringraziare il prof. Guido Albertengo per la disponibilità che mi ha dato nel farmi da relatore, Santer Reply SPA che mi ha permesso di apprendere numerosi concetti lavorativi e tecnologici che non conoscevo, il mio tutor aziendale dott.ing. Marco Mastropasqua per avermi seguito dal primo giorno in azienda e avermi insegnato ad utilizzare i "ferri del mestiere". Volevo inoltre ringraziare la città di Torino per avermi accolto per tanti anni e avermi sempre fatto sentire a casa.

Indice

Capitolo 1

Stato dell'arte

1.1 L'Internet of Things

Il numero di dispositivi Internet of Things (IoT) connessi alla rete Internet nel 2019 ammontava a 7.74 miliardi: nel 2020 questo valore è salito a 8.74 miliardi e le analisi prevedono che nel 2030 si arrivi a 25.44 miliardi di dispositivi connessi. La fetta più grande del mercato dei dispositivi IoT riguarda gli smartphone, che si stima possano raggiungere da soli la cifra di 8 miliardi nel 2030.[1] Un quantitativo così grande di dispositivi rende perfettamente sensato il neologismo coniato da Kevin Ashton nel 1999, il quale indicava con il termine IoT la rete di oggetti fisici che si interfacciava al web e che ricopriva tutto il pianeta.

IoT è spesso associato alla parola **smart** poiché entrambi identificano un oggetto fisico che può interfacciarsi con una rete, senza l'ausilio di cavi, per scambiare informazioni. Essi sono in grado di acquisire dati dall'ambiente circostante e farne un'elaborazione, che può essere una semplice formattazione numerica ma anche un'analisi **on the edge** di quegli stessi dati.

I campi di applicazione in cui l'IoT è già oggi una tecnologia dominante sono molteplici:

- Domotica e Smart Home.
- Smart Buildings.
- Monitoraggio industriale.

[1] `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/`

- Self Driving Car e industria automobilistica
- Smart Health in ambito sanitario.
- Smart City e Smart Mobility.
- Sicurezza e videosorveglianza.
- Agricoltura smart e zootecnia.[1]

Uno stesso dispositivo fisico può essere impiegato in più di uno degli ambiti sopracitati: ad esempio una sensore Radar di prossimità potrebbe essere utilizzato sia nell'ambito della Smart Home per rilevare quando accendere una fonte luminosa, sia nell'ambito della sicurezza per segnalare una presenza umana indesiderata.

La discriminante tra gli oggetti che compongono il mondo IoT è il **firmware**: impartire un compito ad un dispositivo IoT significa avere del firmware salvato su un chip elettronico (ROM, EEPROM, FLASH), che viene eseguito dal microprocessore del device e che costituisce il contatto tra l'architettura hardware e quella software. Ogni dispositivo elettronico in grado di eseguire un'ordine presenta del firmware al proprio interno, che lo abilita ad effettuare ogni azione, dall'acquisizione dati dai sensori alla connessione ad una rete wireless.

La velocità con cui si evolve il mondo IoT e la tecnologia in generale implica una costante necessità di aggiornamento delle funzionalità di un prodotto e quindi del suo firmware. Si supponga di avere una centralina per l'impianto di irrigazione di un terreno che ne verifica l'umidità tramite sensori che analizzano l'umidità media: si scopre che il raccolto non è stato abbondante come da previsioni poiché l'analisi dell'umidità media non è un dato sufficiente ma devono essere segnalati anche i picchi di umidità, in modo da diminuire/alzare immediatamente il flusso idrico. Sarà necessario un aggiornamento del firmware per andare a correggere le funzionalità del dispositivo. Nel caso in cui si abbia a che fare con un'unica centralina, sarebbe facile fare l'aggiornamento, andandolo a caricare fisicamente sulla memoria locale; se, però, il numero dei devices aumenta e raggiunge l'ordine di qualche decina, non è conveniente fare l'aggiornamento manualmente e si potrà adottare, invece, un sistema automatico, in grado di comunicare con un'enorme quantità di dispositivi in contemporanea e sopratutto in tempi brevi.

Lo scopo di questo lavoro di Tesi è proprio quello di andare a costruire un'applicazione in grado di gestire questa necessità: la tecnologia abilitante, su cui si basa questo progetto, è la tecnologia **cloud**.

1.2 La tecnologia cloud

La crescita del cloud negli ultimi quindici anni è dovuta alla necessità, da parte delle aziende, di avere sempre a disposizione, senza costi di manutenzione e su larga scala, un numero scalabile di risorse a costi prefissati. In base alla presenza geografica dei data center del provider, le risorse cloud possono - più o meno velocemente - essere chiamate e create in ogni parte del mondo e questa opportunità offre un'enorme versatilità di soluzioni. Inoltre, se presso la propria azienda si possiedono risorse come, ad esempio, un cluster di server, esse possono essere facilmente integrate col cloud attraverso soluzioni ibride che permottono di lavorare sia in locale sia in remoto, gestendo così eventuali problemi di sensibilità dei dati. Tali risorse possono includere sia prodotti hardware che software e sono etichettate dai provider dei servizi cloud in tre macrocategorie:

- **IaaS** (*Infrastracture as a Service*): dischi rigidi, memorie flash, server, macchine virtuali, sistemi HPC.
- **PaaS** (*Platform as a Service*): sistemi operativi, basi di dati, contenitori, API dedicate.
- **SaaS** (*Software as a Service*): applicazioni per l'e-commerce, sistemi di gestione delle risorse di archiviazione, software di gestione dei pagamenti

Le soluzioni IaaS prevedono la gestione delle risorse hardware da parte del provider che le affitta, e quindi implicano una spesa nulla per quanto riguarda i costi di gestione fisica della struttura IT: esse consentono di gestire on-demand la potenza di calcolo e le risorse di memoria necessarie, adattandole a progetti che altrimenti sarebbero enormemente dispendiosi per le finanze di una sola azienda come l'analisi di big data. Il pagamento del servizio, viene effettuato soltanto per le risorse che vengono utilizzate; questo rende l'IaaS ideale per costruire, ad esempio, un grosso sito di e-commerce il cui traffico varia sensibilmente dalle ore notturne alle ore diurne o in giorni particolari dell'anno. Inoltre anche processi particolarmente dispendiosi, come grossi backup di dati, vengono gestiti completamente dal provider. Di contro, la garanzia della sicurezza dei dati è in mano al fornitore del servizio cloud e ciò potrebbe implicare problemi legali sulla diffusione dei dati. Inoltre, una volta intrapresa una grossa fornitura da un provider, diventa molto complesso trasferire l'intera infrastruttura al cloud di un nuovo provider. Le risorse di tipo IaaS, pertanto, devono essere gestite dal cliente a livello di

configurazione, quindi aggiornamenti, antivirus, firewall, installazioni sono tutte a carico dell'utente finale.[2]

Le PaaS raggruppano una tipologia di servizi in cui la gestione del sistema a livello infrastrutturale viene fatta direttamente dal provider cloud, che quindi si occupa della gestione dei server, delle virtual machines, dell'organizzazione dell'archiviazione e dei middlewares, fornendo un'ambiente di sviluppo già pronto per la creazione di applicazioni web; risulta quindi un tipo di servizio particolarmente favorevole per le aziende che vogliono creare velocemente il software e metterlo a disposizione, a livello aziendale o a livello commerciale, direttamente dal cloud sotto forma di SaaS. Ogni provider configura le proprie piattaforme con i linguaggi che rietiene più concorrenziali e che generano più richiesta, per cui non tutti i linguaggi sono supportati dalle piattaforme SaaS. Inoltre, occorre fare a monte un'analisi del traffico che un'applicazione genererà: infatti in caso di alti picchi di traffico, il provider addebiterà costi maggiori che bisogna prevedere prima di rendere disponibile online la propria applicazione. [3]

Va da sé che i SaaS rappresentino le risorse ad astrazione più alta dell'ambiente cloud: sia l'hardware che la piattaforma software vengono gestiti dal provider mentre all'utente sono lasciate le funzionalità dell'applicazione software. L'utente dovrà solamente avere un account e provvedere alle spese per l'utilizzo ma tutta l'infrastruttra sottostante non sarà di sua competenza. Le soluzioni verranno eseguite in remoto sul cloud e utilizzate localmente tramite una connessione Internet. L'azienda non dovrà più pagare costose licenze per l'utilizzo del prodotto ma soltanto l'affitto dell'applicazione per ogni utente: questo fatto rende molto più facile e veloce l'integrazione di nuovi collaboratori. Vi è inoltre una retroazione positiva per quanto riguarda la sicurezza: gli standard di sicurezza dell'azienda provider del servizio cloud andranno ad applicarsi al servizio SaaS e questo renderà sicuri e affidabili i dati su cui l'azienda cliente (più piccola) andrà a lavorare, garantendogli standard che da sola farebbe difficoltà ad avere. I problemi per quanto riguarda i servizi SaaS sono legati alla **cessione dei dati** in quanto non si ha più il controllo diretto sui dati elaborati dall'applicazione, ma bisogna fare affidatemento sulla segretezza che garantisce il provider. [4]

Le PaaS sono le principali risorse che verranno usate in questo progetto di Tesi.

1.3 L'aggiornamento del firmware

Il firmware è la parte di codice responsabile di tutte le interazioni con l'hardware di un dispositivo e quindi anche il fulcro di tutte le funzionalità dello stesso. Un malfunzionamento o un bug nel firmware potrebbe portare anche alla non accensione di un device oppure all'impossibilità di andare ad eseguire il software applicativo: l'aggiornamento del firmware risulta dunque essere una parte cruciale della vita di dispositivo.

Quando si fa l'update di device, le informazioni fondamentali sono due: il numero di serie, che ne va ad identificare la classe a cui appartiene (ad esempio a FTT12N897A2020 corrisponde il sistema di infotainment di una Fiat Tipo con anno di costruzione 2020), e la versione del firmware correntemente installata (quando il device è stato appena rilasciato, avrà tipicamente al suo interno una versione 1.x).

Durante l'aggiornamento del firmware, l'alimentazione elettrica deve essere sempre disponibile, in caso contrario si può andare incontro ad una corruzione dei dati all'interno della ROM e quindi ad un grave malfunzionamento del dispositivo, che nei casi peggiori, può portare anche all'incapacità di eseguire le microistruzioni più basilari.

Si vanno quindi ad analizzare le due tipologie di *firmware update*, quello locale e quello remoto (di interesse per questo progetto di tesi).

1.3.1 L'aggiornamento manuale

Se si è presenti sul luogo di lavoro del dispositivo che si vuole aggiornare, si può effettuare manualmente un aggiornamento locale.

L'informazione fondamentale da acquisire è il codice seriale del dispositivo che si può ottenere in diversi modi:

- Dal package in cui era contenuto.
- Da un'etichetta apposta sul retro del device.
- Se il costruttore ne fornisce una, dalla dashboard d'interfaccia web per quella classe di dispositivi.
- Se il dispositivo è dotato di display, dalle impostazioni di sistema.

Se si è il costruttore del device, allora si disporrà di un database relazionale in cui ad ogni ID seriale corrisponderà la rispettiva classe di appartenenza e l'ultima versione del firmware installata, altrimenti bisognerà consultare la

pagina web dell'azienda produttrice per verificarne la classe e la disponibilità di un aggiornamento.

L'aggiornamento manuale risulta una scelta valida quando:

- Il dispositivo non ha una buona connessione ad Internet.
- Il dispositivo deve tornare in uno stato noto in seguito ad un malfunzionamento (ad esempio deve tornare in Power ON).
- Il dispositivo sta lavorando in situazioni particolarmente critiche (pacemaker, pompe ad insulina).[5]

Una volta scaricato l'aggiornamento su un supporto rigido quale un harddrive o su un PC, si deve collegare fisicamente il device a questi supporti. E' necessario che l'unità di archiviazione non presenti blocchi malfunzionanti che possano causare la corruzione del nuovo firmware e quindi il malfunzionamento del dispositivo da aggiornare. Se si utilizza un PC, bisogna disabilitarne temporaneamente l'antivirus che potrebbe interferire e bloccare il processo di aggiornamento verso il device. Il costruttore del dispositivo deve inoltre assicurarsi che, durante l'aggiornamento, una copia del vecchio firmware rimanga salvata in memoria fino all'installazione della nuova: in questo modo è possibile fare un reboot della vecchia versione in caso di mancato completamento dell'installazione della nuova. Dovrà essere presente anche un programma **boot loader** in grado di reindirizzare correttamente la memoria dopo un aggiornamento o in caso di fallimento dell'aggiornamento.

In sintesi, sull'aggiornamento manuale di un device si possono fare le seguenti affermazioni:

- Un individuo deve occuparsi personalmente e fisicamente dell'aggiornamento.
- E' necessario avere a disposizione una finestra temporale in cui il dispositivo non viene utilizzato in modo da poterlo acquisire fisicamente.
- Il device è spesso difficile da riacquisire una volta che non è più nella mani del produttore.

E' chiaro che l'aggiornamento locale del firmware è possibile soltanto in condizioni in cui si ha l'accesso diretto al dispositivo ed il numero dei dispositivi è estrememente ridotto: se non ci si trova in questa situazione, probabilmente la scelta migliore sarà effettuare un *Firmware Update Over the Air*.

1.3.2 Il FOTA

L'acronimo OTA (*Over The Air*) si riferisce ad una tipologia di aggiornamento del software che avviene in remoto, quindi con uno scambio di informazioni criptate tra i nodi che intervengono nella comunicazione, che permette un aggiornamento instantaneo di tutti i dispositivi collegati all'unità centrale che ordina l'aggiornamento. Quando l'aggiornamento viene richiamato, i dispositivi devono per forza effetturalo appena terminate le loro routine. Non necessitando di un accesso locale ai device, questo sistema permette un risparmio sia in termini di tempo sia di risorse economiche.

Nel 2015, Chrysler Automobiles venne criticata per aver inviato via posta ai propri clienti delle chiavette USB contenti degli aggiornamenti firmware per il GPS degli autoveicoli: una volta in possesso delle chiavette, i proprietari dovevano accendere la vettura e aspettare oltre mezz'ora per il completamento dell'aggiornamento. Appare chiaro quanto il metodo fosse inefficiente ed esposto a facili rischi di manomissione dei singoli dispositivi. Al contrario, Tesla nel 2016 sperimentò per la prima volta il FOTA (*Firmware Over The Air*) quando inviò un aggiornamento da remoto che installava una funzionalità di parcheggio automatico sui propri veicoli, senza che i proprietari dovessero installare manualmente l'aggiornamento. Nel 2018, però, la stessa Tesla venne fortemente criticata perché la sua vettura berlina Model 3 aveva una risposta in frenata inferiore a quella di un pick up di grosse dimensioni come il Ford F-130: come risposta alle critiche, pochi giorni più tardi, l'azienda eseguì un FOTA sulle proprie berline migliorandone la frenatura di 18 ft = 2,4384 m.[6]

Si può quindi affermare che attraverso il FOTA:

- I bug ed il comportamento di un prodotto possono essere costantemente ed automaticamente aggiornati.
- Le aziende possono testare nuove funzionalità su larga scala inviandole ai dispositivi.
- Le aziende ottengono un risparmio economico gestendo tutti gli aggiornamenti da un'interfaccia unificata.
- Gli sviluppatori possono scrivere in maniera più sicura in quanto i prodotti continueranno il loro funzionamento normale durante la fase di sviluppo.
- Aumenta la scalabilità di un dispositivo in quanto ne consente l'aggiunta di funzionalità direttamente sul campo.[7]

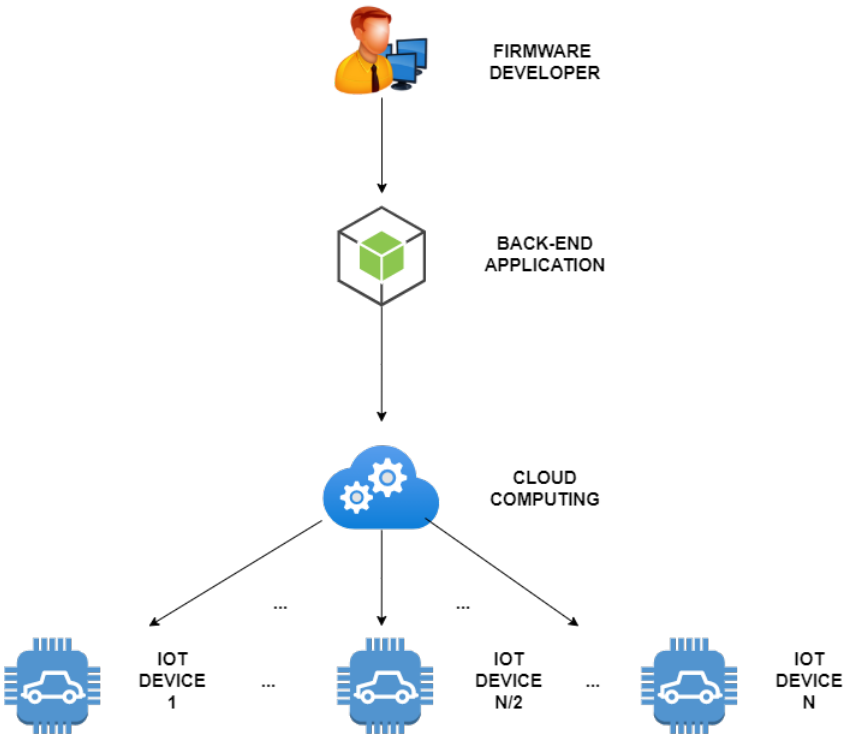

Figura 1.1. Lo sviluppatore firmware quando avrà a disposizione del nuovo codice, richiamerà un'applicazione back-end incaricata di parlare col cloud, che a sua volta invierà l'aggiornamento ai device selezionati.

L'aggiornamento FOTA presenta degli svantaggi da tenere in considerazione:

- Il dispositivo necessita di lavorare con una connessione ad internet.
- Il device ha bisogno di essere riavviato dopo un update; questo fatto può causare problemi di scheduling dell'aggiornamento.
- In generale, il *fail rate* di un aggiornamento FOTA è più alto di un aggiornamento manuale, in quanto il device deve effettuare da solo un controllo su sè stesso; può essere un fatto critico in caso il device abbia problemi di connessione oppure di power OFF e questi portino ad un fallimento del download o ad una corruzione del dispositivo. [5]

La sinergia naturale tra l'approccio di tipo FOTA al mondo IoT e il cloud computing è alla base dello sviluppo di questa di tesi.

Capitolo 2

Gli attori del FOTA

2.1 Microsoft Azure

Microsft Azure è una piattaforma di elaborazione cloud a cui il Chief Software Architect di Microsoft Ray Ozzie fece per la prima volta cenno in un memorandum aziendale del 2005. Ozzie teorizzava una nuova piattaforma in grado di rendere disponibili, via web, prodotti come Microsoft Office, l'architettura OS e le applicazioni .NET. Azure venne lanciato per la prima volta il primo febbraio del 2010[8] e attualmente include tutti e tre i livelli di soluzioni cloud IaaS, PaaS e SaaS. Le principali caratteristiche di Azure possono essere sintetizzate come segue:

- **Flessibilità**: la piattaforma consente lo spostamento di risorse in maniera rapida tra l'archiviazione locale e quella cloud.
- **Ampiezza**: Azure supporta ogni tipo di sistema operativo, linguaggio di programmazione, tool o framework.
- **Affidabilità**: Microsoft garantisce sempre, a prescindere dal pacchetto utilizzato, una disponibilità delle risorse pari almeno al 99.5% totale del tempo, dal momento dell'acquisto, e supporto tecnico 24/7.
- **Globalità**: Microsoft presenta strutture e data center che ospitano i suoi server in ognuno dei 5 continenti della Terra: questo fatto rende molto veloce la connessione con il cloud se si sceglie una risorsa geograficamente vicina.
- **Economicità**: il cliente paga soltanto il prodotto che sta effettivamente utilizzando. [9]

Microsoft offre una serie di documentazioni su ci ci si è basati per lo studio della piattaforma Azure, con appositi moduli di learning e lezioni tenute dagli esperti di Microsoft sull'approccio al mondo cloud.

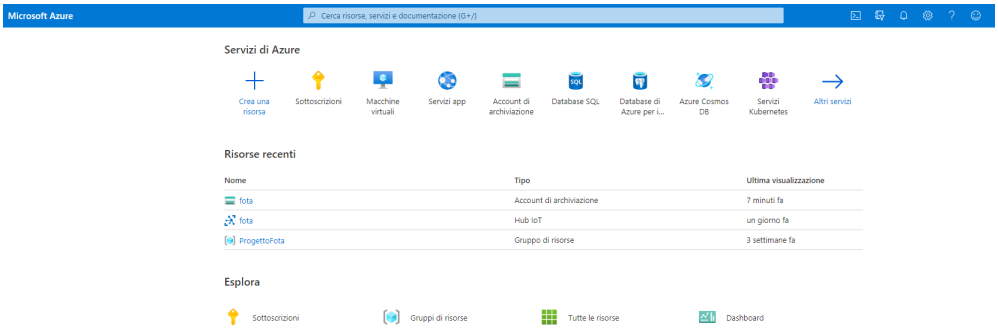

Figura 2.1. Homepage di Microsft Azure.

2.1.1 Le SDK

Con *Software Development Kit* (SDK) si intende una raccolta di strumenti forniti dal provider di un servizio (in questo caso Azure e tutte le sue risorse correlate), che consenta allo sviluppatore una più agevole costruzione del software. E' infatti di interesse del provider che applicazioni di terze parti vengano rilasciate sul mercato, in modo da dare più visiblità alla sua risorsa. La risorsa può essere anche un nuovo linguaggio di programmazione, si ricordi infatti che Microsoft nel 2002 introdusse **C#** in questo modo. [12] Le SDK di Azure includono:

- Driver.
- Protocolli di comunicazione.
- Librerie.
- Ambienti di sviluppo.
- Documentazione ed esempi.

In questo progetto l'ambiente di sviluppo è stato **Microsoft Visual Studio Code**, scelto per la sua integrazione integrazione naturale con i servizi Azure e per la capacità di eseguire rapidamente operazioni di Push/Pull del codice attraverso la piattaforma **github**. Azure possiede una pagina su github da

cui è possibile accedere a tutte le SDK con i relativi pacchetti in maniera gratuita. [1] Le SDK di Azure sono sottoposte a licenza MIT rilasciata dal *Massachusetts Institute of Technology*, che ne permette il libero utlizzo al solo costo di preservarne il copyright. I linguaggi supportati dalla piattaforma Azure sono diversi:

- .NET.
- Java.
- Javascript/Typescript.
- Python.
- Go.
- C.
- C++.
- Android.
- iOS.
- Ruby.
- PHP.

Il linguaggio scelto per lo sviluppo dell'applicazione relativa a questo progetto è stato Javascript assieme al suo framework Node JS.

Il framewok Node JS

Node Js è un ambiente di sviluppo run-time e open source di Javascript, introdotto nel 2009 sul motore di elaborazione V8 di proprietà di Google. E' un ambiente fullstack, ovvero offre la possibilità di lavorare sia lato client che lato server, con lo stesso linguaggio di programmazione, che viene tradotto dal motore direttamente in linguaggio macchina: questo implica l'assenza di compilatori che ne rallentino l'esecuzione. Essendo basato su Javascript è un linguaggio molto semplice, che non necessita di una stretta tipizzazione degli oggetti: ciò consente di avere un codice molto più snello dal punto di vista

[1] `https://azure.github.io/azure-sdk/releases/latest/index.html`

formale e anche estrememente leggero da far girare, andando quindi ad incrementarne le prestazioni in real-time . A livello di paradigma, basandosi sull'OOP (*Object Oriented Programming*) già presente in Javascript, consente di costruire applicazione estrememente modulari costituite da diverse API e microservizi, quindi una modalità di sviluppo software più leggera rispetto ad avere un unico nucleo multifunzionale. Node JS risulta ideale per un'applicazione come quella FOTA perchè possiede un numero elevatissimo di pacchetti, costantemente aggiornati dalla community di Node e installabili tramite il package manager **npm** (per la costruzione ad esempio delle API del server è stato usato il pacchetto **restify**). Avendo inoltre la necessità di interagire con un gran numero di dispositivi IoT, Node permette di gestire tramite il *non-blocking I/O* diverse richieste concorrenziali in contemporanea. Inoltre le applicazioni vengono salvate nella cache del motore Google, quindi non vengono rieseguite ogni volta che vengono chiamate e ciò permette di avere una latenza estrememente bassa. [13]

2.1.2 L'Iot Hub

L'hub IoT di Azure è un servizio PaaS che funge da gateaway tra un qualsiasi device IoT e un'applicazione back-end, consentendone una comunicazione sicura e affidabile. Può connettere milioni di dispositivi contemporaneamente ed è abilitato per la comunicazione bidirezionale, quindi integra il supporto per l'invio di dati di telemetria da dispositivo a cloud, il caricamento di file da dispositivo e metodi diretti per il controllo del device. Ogni volta che un dispositivo si connette all'hub, il suo ID viene registrato nel **registro delle identità** che permette di tenere traccia dell'intera rete di connessioni. Inoltre la dashboard dei log dell'hub consente di ottenere tutte le statistiche necessarie al monitoraggio dei dispositivi; è inoltre disponibile un'interfaccia per la visualizzazione dei dispositivi attualmente registrati all'IoT hub.

Quando un device risulta connesso all'hub, ci sono due modi per per impartirgli un comando:

1. Utilizzare un metodo diretto.

2. Utilizzare i *Device Twin* (Dispositivi Gemelli)

I metodi diretti sono una funzionalità dell'IoT hub che permette ad un utente di effettuare una chiamata di tipo richiesta/risposta (modello **HTTP**) verso il device. Questa funzionalità risulta particolarmente utile quando si deve prendere una decisione basata sulla capacità che il dispositivo ha di

elaborare una risposta. Le richieste per i metodi diretti sono sincrone, ovvero prima di effettuare una nuova richiesta ad un device bisognerà attendere un timeout impostabile da 5 a 300 secondi. Risulta una tipologia di azione molto utile anche nel caso in cui si necessiti che il dispositivo esegua un'azione interattiva, ad esempio l'accensione di un impianto di condizionamento. Per richiamare un metodo diretto si può utilizzare un URI del tipo *iot hub name/twins/device id/methods/*[10]: nel nostro caso si è scelto *iot hub name* = **"fota"**. I metodi diretti non saranno oggetto d'interesse di questo lavoro di Tesi: si utilizzeranno invece i dispositivi gemelli per l'interazione con i devices.

2.1.3 I dispositivi gemelli

I dispositivi gemelli sono degli oggetti JSON[2] che vengono creati ogni volta che un device si registra sull'IoT hub: essi contengono informazioni sui metadati e sulle configurazioni del nuovo dispositivo. Possono essere utilizzati dal dispositivo e dall'applicazione di back-end per sincronizzare il proprio stato e comunicare operazioni da eseguire, oppure riportare il proprio status.

Oltre alla sezione **radice**, in cui sono salvate le opzioni correlate al registro di sistema, come l'ID del dispositivo oppure la data dell'ultimo accesso all'hub, il dispositivo gemello è dotato di tre sezioni particolari:

1. **Tag**: è una sezione in cui solo l'applicazione back-end può scrivere e che risulta invisibile al dispositivo. Consente di associare un tag per indirizzare le operazioni soltanto al tipo di dispositivo contraddistinto da una determinata classe.

2. **Desired Properties** (Proprietà Desiderate): è un campo in cui solo l'applicazione di back-end può scrivere e che può essere letto dall'applicazione del dispositivo. E' il meccanismo fondamentale che si userà per notificare al dispositivo la disponibilità di un nuovo firmware per la sua classe.

3. **Reported Properties** (Proprietà Segnalate): è un campo in cui può scrivere soltanto l'applicazione del device e che può essere letto o interrogato (tramite opportune query in linguaggio pseudo-SQL) dall'applicazione back-end. Insieme alle proprietà desiderate, permette la sincronizzazione dello stato del dispositivo. Nel progetto veranno utilizzate per

[2]Javascript Notation Object

segnalare il completamento dell'aggiornamento del firmware.[11]


```
{
    "deviceId": "audiQ5_2018_tdi_30_4ngh5",
    "etag": "AAAAAAAAAAI=",
    "deviceEtag": "ODkyODYwNDEx",
    "status": "enabled",
    "statusUpdateTime": "0001-01-01T00:00:00Z",
    "connectionState": "Disconnected",
    "lastActivityTime": "0001-01-01T00:00:00Z",
    "cloudToDeviceMessageCount": 0,
    "authenticationType": "sas",
    "x509Thumbprint": {
      "primaryThumbprint": null,
      "secondaryThumbprint": null
    },
    "modelId": "",
    "version": 3,
    "properties": {
      "desired": {
        "firmware": {
          "fwVersion": "1.1"
        },
        "$metadata": {
          "$lastUpdated": "2021-03-05T08:32:08.376789Z",
          "$lastUpdatedVersion": 2,
          "firmware": {
            "$lastUpdated": "2021-03-05T08:32:08.376789Z",
            "$lastUpdatedVersion": 2,
            "fwVersion": {
              "$lastUpdated": "2021-03-05T08:32:08.376789Z",
              "$lastUpdatedVersion": 2
            }
          }
        },
        "$version": 2
      },
      "reported": {
        "$metadata": {
          "$lastUpdated": "2021-03-05T08:31:22.6771251Z"
        },
        "$version": 1
      }
    },
    "capabilities": {
      "iotEdge": false
    }
```

```
46  }
```

Listing 2.1. Codice JSON per un device Twin preso dal portale di Azure

Il campo *deviceId* indica l'Id con cui si è registrato il device all'IoT hub, mentre *autenthicationType* indica il tipo di connessione utilizzata. Si noti come tra i metadati compaiano sempre dei timestamp che riportano la data dell'ultimo aggiornamento in formato UTC [3] e ISO8601[4].

2.1.4 Protocolli di comunicazione Client-IoT hub

Facendo riferimento all'**application layer** del modello **ISO/OSI**, un device generico può utilizzare i seguenti protocolli per comunicare con l'IoT hub:

- **MQTT**, PORTA 8883.
- **MQTT su WebSocket**, PORTA 443.
- **AMQP**, PORTA 5671.
- **AMQP su WebSocket**, PORTA 443.
- **HTTPS**, PORTA 443.

Il protocollo binario MQTT è una scelta ideale quando si vuole realizzare una connessione **end-to-end** con un dispositivo che non condivide le sue credenziali con altri devices e che non opera sulla stessa **TLS** (Transport Layer Security)[5]. MQTT supporta il push da parte del server, questo significa che i dispositivi non dovranno effettuare un polling periodico delle richieste per verificare la presenza di un messaggio, come avviene invece con HTTPS. MQTT rappresenta inoltre una soluzione ideale quando il dispositivo ha poca memoria RAM a disposizione poichè richiede una **footprint**[6] piccola.

Come MQTT, anche AMQP è un protocollo binario, quindi ha un payload[7] più compatto rispetto ad HTTPS ma sfrutta il multiplexing dei dispositivi su uno stesso livello di presentazione TLS e può quindi sfruttare un pool

[3]Coordinated Universal Time: il riferimento globale da cui si calcolano tutti i fusi orari

[4]YYYY-MM-DDTHH:MM:SS.mmmZ

[5]Protocollo crittografico di presentazione che fornisce integrità dei dati alle reti IP

[6]Spazio occupato dall'applicazione in esecuzione nella RAM.

[7]Carico utile disponibile per il messaggio con un dato protocollo.

di connessioni per singola identità verso l'IoT hub. Come MQTT, anche AMQP supporta il push dei messaggi dal server; ne consegue che il protocollo possiede una latenza molto bassa. Il payload, come per l'MQTT, è molto compatto a livello di pacchetto, ne conseguono meno dati da trasferire, minore rischio di perdita di dati e minore banda utilizzata: pagando le risorse di Azure anche in base al consumo di banda e al traffico, un pacchetto più compatto coincide anche con una riduzione dei costi. Di contro il protocollo AMQP ha una footprint maggiore di HTTPS e MQTT e questo può essere un problema quando si lavora con dispositivi a bassa disponibilità di RAM (minore di 1MB).

Se si ha la necessità di costruire un'applicazione real-time per browser oppure smartphone, una scelta ideale può essere quella di incapsulare i protocolli MQTT o AMQP in un frame Websocket[8] e utilizzare i protocolli MQTT su Websocket e AMQP su Websocket: questa scelta consente di alleggerire ulteriormente la comunicazione e di utilizzare un sistema di **handshaking**[9] per ridurre drasticamente le perdite d'informazione.

Vista l'elevata latenza ed il polling periodico da effettuare da parte del dispositivo, HTTPS è da utilizzare soltanto nel caso in cui un device non supporti gli altri protocolli. [14]

Nel caso di questo progetto di tesi, la scelta per l'interazione dei devices è stata l'MQTT, in quanto si ha un set di credenziali unico per ogni singolo dispositivo e si è voluta avere la massima flessibilità sulla disponibilità di risorse di memoria dei dispositivi, oltre che una bassa latenza in fase di comunicazione con l'hub. L'implementazione è stata fatta attraverso il pacchetto *Mqtt* delle SDK di Azure per Node JS.

2.1.5 Il Blob Storage

Il Blob Storage di Azure è un servizio di tipo PaaS che consente l'archiviazione di oggetti non strutturati sul cloud. Per oggetti non strutturati si intendono formati di dati che non seguono una struttura predefinita ma hanno struttura casuale, come ad esempio file di testo o binari. Sono archivi pensati specificatamente per:

- L'archiviazione file ad accesso distrubuito.
- File audio e video.

[8]Protocollo web Full-Duplex, simile ad HTTP.

[9]Scambio di pacchetti client-server prima dell'inizio della comunicazione.

- File di log.
- Dati di backup.

Per inizializzare una risorsa di archiviazione, bisogna prima creare un account di archiviazione che fornirà gli endpoint univoci in cui trovare le risorse. In fase di creazione, sarà possibile scegliere il tipo di account, se per archiviazione di risorse specifiche (solo file, solo blob, blob a blocchi di accodamento) o generiche, scegliere se adottare una ridondanza dei dati a livello locale (più copie all'interno dello stesso server), a livello di zona (**ZRS**, più copie all'interno di più server dello stesso data center), a livello geografico (**GZRS/RA-GZRS**, più copie all'interno di più server di molteplici data center in aree geografiche differenti). Inoltre Microsft permette di scegliere tra un livello di prestazioni standard per l'archiviazione di BLOB, file, tabelle e un livello di prestazioni premium, per l'archiviazione di dischi di macchine virtuali: nel caso di questo progetto, la scelta è ricaduta sull'archiviazione standard. Successivamente si è creato il contenitore Blob Storage denominato *apifota*, al cui interno verranno archiviati i codici relativi ai nuovi firmware disponibili. Al momento della creazione, Azure propone tre scelte sul tipo di velocità di accesso allo storage:

1. **Accesso frequente:** l'archiviazione è ottimizzata per applicazioni critiche che richiedono un rapido accesso ai blob. In questo caso, i costi di archiviazione saranno maggiori.

2. **Accesso sporadico** l'archiviazione è ottimizzata per enormi quantità di dati che vengono tenuti nello storage per almeno 30 giorni. I costi di archiviazione saranno minori ma quelli per l'accesso saranno maggiori rispetto all'accesso frequente.

3. **Accesso archivio:** ottimizzato per l'archiviazione di dati a lunga durata e per applicazioni in cui la latenza non è un fattore importante. L'accesso ai dati sarà costoso, ma i costi di archiviazione bassi.

Vista la frequenza di accesso al Blob Storage dovuta all'elevato numero differente di dispositivi e alla frequenza del rilascio degli aggiornamenti, la scelta per questo progetto è stata l'accesso frequente. [15]

Gli endpoint sono accessibili mediante i protocolli HTTP e HTTPS e tramite le librerie client del Blob Storage: in particolare saranno utilizzate le librerie **azure-storage** e **azure/storage-blob**, compatibili con Node JS.

L'endpoint conterrà il nome dell'account di archiviazione che perciò non potrà avere un doppione all'interno del cloud: in questo progetto il nome scelto per l'account di archiviazione è *fota*.

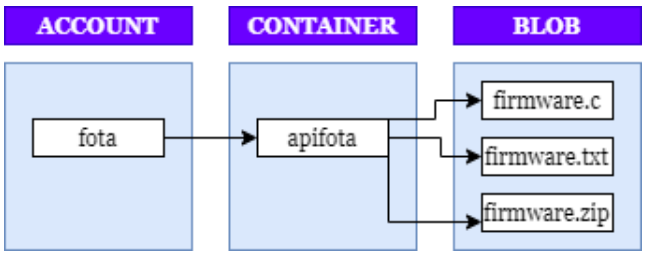

Figura 2.2. Struttura dell'account di archiviazione fota

Gli endpoint avranno una struttura del tipo: *http://mystorageaccount.blob.core.windows.net/mycontainer/myblob*. Per effettuare un accesso sicuro al blob storage, l'utente o il device che richiamano l'endpoint avranno bisogno di autenticarsi al cloud in uno dei tre modi seguenti:

1. **Azure Active Directory.**
2. **Autorizzazione con chiave condivisa.**
3. **Firma di accesso condiviso.**

Nel primo caso, si può concedere, come proprietario dell'account di archiviazione, un ruolo di gestione dello storage ad un utente/applicazione a cui verrà fornita la possibilità di generare un 2.0 OAuth token con cui potrà autenticarsi al servizio. Per utilizzare questo metodo, l'account di archiviazione deve essere stato creato con **Azure Resource Manager**.

Nel secondo caso, si utilizza una stringa di connessione e che viene fornita direttamente dal proprietario dell'account di archiviazione, generandone una dal portale di Azure. Il comportamento di una stringa di connessione è simile a quella di una password root, quindi una volta generata, l'utilizzatore dovrà salvarla in un archivio sicuro ed evitare di farla circolare al di fuori del suo ambiente di lavoro in locale. Le stringhe di connessione devono essere rigenerate a cadenza periodica per garantire la sicurezza dell'archivio.

Nel terzo caso, si può utilizzare un SAS token per generare una firma di accesso condiviso ed avere un controllo più preciso sui permessi da fornire agli utenti del servizio, è infatti possibile specificare a quali risorse specifiche può accedere il client, quali autorizzazioni hanno per le risorse e quanto tempo è

valido il token. La firma di accesso condiviso è un URI [10] contenente un token, al cui interno sono specificate le modalità di accesso alla risorsa da parte del client. Uno dei parametri del token è la firma che viene generata partendo dai privilegi SAS e a sua volta firmata con una chiave di delega utente oppure con una chiave dell'account di archiviazione. Una volta che un utente dispone del privilegio di creazione della firma ad accesso condiviso, non è più possibile controllare la generazione dei token dei client da parte dell'amministratore dello storage. Il token viene generato lato client utilizzando una delle librerie delle SDK fornite da Microsoft ed è possibile crearne in numero illimitato; una volta creato, è possibile condividerlo con le altre applicazioni client per accedere ad una risorsa all'interno del blob storage. L'URI finale sarà formato nel seguente modo: *Storage Resource URI + ? + SAS token*.

La scelta per questo progetto di tesi è stata quella di creare un file *config.js* dove memorizzare le stringhe di connessione dell'account di archiviazione e farvi accedere l'applicazione di back-end; in questo modo, l'applicazione di back-end possiederà tutti i privilegi da amministratore dell'account e potrà firmare gli URI che saranno poi forniti ai device client, come si vedrà più nello specifico durante la discussione del codice sorgente.

Figura 2.3. Pagina di generazione per le chiavi di accesso e e per le stringhe di connessione dell'account fota.

[10] Uniform Resource Identifier.

2.2 Le API

Per API (*Application Program Interface*) si intende un'applicazione server capace di valutare richieste su un set di URLs, elaborarle attraverso l'architettura server a più basso livello e fornire una risposta a quelle richieste. Per cercare un prodotto nel catalogo del sito, ad esempio, si può aprire, da smartphone o da PC, un browser client con cui si navigherà nel sito dell'e-commerce fino alla barra di ricerca dei prodotti: a questo punto il browser invierà una richiesta ben precisa al server con un body contente il nome del prodotto cercato, il filtro sul prezzo, la categoria specificata, la valutazione media del prodotto e ogni altro parametro impostato dall'utente. Quando la richiesta giungerà all'endpoint corrispondente, l'API si occuperà di effettuare delle query sul database dell'e-commerce per verificare l'effettiva presenza in magazzino del prodotto richiesto, controllare se c'è più di un prodotto corrispondente a quella ricerca, prendere le immagini relative a quei prodotti, il tutto senza far conoscere al client nulla rispetto all'elaborazione in corso. Una volta che l'elaborazione è stata completata, essa potrà avere un esito positivo, quindi il browser stamperà la lista dei prodotti con cui l'API ha risposto, oppure negativa stampando un messaggio di errore. [16]

In particolare per questo progetto si è fatto riferimento ad un tipo particolare di API, le **API REST.**

2.2.1 Le API REST

L'acronimo REST sta per *Representational State Transfer* e fu coniato dall'informatico Roy Fielding nella sua tesi di dottorato alla *University of California* di Irvine nel 2000. Fielding definiva 6 vincoli necessari ad un'interfaccia per essere considerata di tipo RESTful:

1. **Client-Server:** questo vincolo implica la separazione tra il lavoro del client sull'interfaccia utente e il lavoro del server sul salvataggio dei dati. Se si applica questo vincolo, le componenti del server risulteranno semplificate aumentandone la scalabilità e si aumenterà la flessibilità dell'interfaccia utente adattandola ad un utilizzo cross-platform.

2. **Stateless:** ogni richiesta da parte del client deve contenere tutto il contenuto necessario alla comprensione della richiesta stessa. Ogni richiesta del client non dovrà dipendere dalla precedente ma dovrà essere analizzata singolarmente dal server.

3. **Cacheable:** le risposte del server a una richiesta devono essere etichettate come **cacheable** oppure **non-cacheable**. Se una risposta è di tipo cacheable, allora il client potrà salvarne i dati nella propria cache in modo da velocizzare future richieste dello stesso tipo.

4. **Uniform Interface:** per avere un'interfaccia quanto più semplice e chiara possibile bisogna rispettare i principi dell'ingegneria del software e avere messaggi auto-descrittivi, dei link *ipermedia* che descrivano lo stato dell'applicazione, un'identificazione chiara delle risorse e chiarezza sullo stato delle risorse da parte del server. Se questi principi sono rispettati, il sistema sarà analizzabile e modellabile a blocchi, disaccoppiando facilmente il blocco client e quello server.

5. **Layered System:** organizzare un sistema a livelli permette all'architettura sovrastante di non vedere quello che succede a livello più basso, permettendo anche in questo caso un disaccoppiamento che fornisce flessibilità all'intero sistema.

6. **Code on Demand:** si tratta di un vincolo opzionale, il quale afferma che il server può effettuare trasferimenti di script al client affinché ne aumenti e personalizzi le funzionalità.

Oltre a questi sei vincoli, la visione chiave di un'architettura REST è quella di identificare ogni informazione come una risorsa. Questo fatto vale per un'immagine, un file di testo, un catalogo, un servizio e, in generale, per ogni oggetto a cui si può attribuire un nome.

La **rappresentazione della risorsa** è un set di informazioni che ne identifica lo stato e che permette al client di andare ad effettuare una richiesta specifica sulla risorsa stessa. Ogni unità atomica di informazione in una API REST possiede, in maniera implicita o esplicita, un link ad una risorsa ed è quindi definibile come **ipertesto**.

I **Metodi della risorsa** sono un set di azioni disponibili per una determinata risorsa presente su un URI dell'API. Per rispettare la definizione RESTful, questi metodi (comunicati ad esempio attraverso il protocollo HTTP) devono essere uniformati e il server deve comunicarli in una maniera comprensibile al client, in modo che il client stesso sappia come utilizzarli e come andrebbero a cambiare lo stato della risorsa.[17] Generalmente questo set di azioni si basa, come già accennato, sugli standard del protocollo HTTP:

- **GET:** si tratta di un metodo di sola lettura con cui il client comunica al server che vuole leggere le informazioni collegate ad un determinato URI. La risposta del server a questo metodo sarà **200, OK!**.

- **PUT:** questa richiesta serve per creare un nuovo oggetto all'URI corrispondente. In particolare, le specifiche per la creazione della nuova risorsa sono specificate all'interno del *body* della richiesta, rendendo quindi la chiamata dell'API estrememante uniformata, qualunque sia l'applicazione back-end eseguita durante la richiesta.

- **POST:** questo metodo aggiunge una nuova risorsa ad una collezione già presente di risorse. Nella pratica PUT e POST hanno quasi lo stesso significato, quindi possono essere usati alternativamente oppure è possibile utilizzare PUT per l'aggiornamento di una risorsa e POST per la creazione. La risposta a questo metodo sarà il codice **201, Created!**.

- **DELETE:** questa azione elimina la risorsa associata all'URI richiamato.

Nel caso di questo progetto di tesi, si useranno i metodi GET e POST associati a delle richieste con protocollo HTTPS, nei cui **header**[11] sarà presente un campo *Content/Type* dentro cui sarà specificata la forma del body del messaggio.

Il body in generale può assumere diversi formati come HTML[12], XML[13], testo, immagini; in questo progetto è stato utilizzato il formato JSON per la sua naturale integrazione con il linguaggio Javascript. Se le API REST costituiranno il core dell'applicazione che si vuole costruire lato server, per simulare richieste dal lato client reali si è utilizzato il software **Postman**.

2.3 Postman

Postman è un tool open source che permette la creazione di richieste client senza dover scrivere codice per il testing delle proprie API. Postman permette di creare collezioni in cui inserire le richieste con metodi personalizzati permettendo di settare sia gli header che il body del messaggio. I body possono essere di diversi formati:

[11] Intestazione del massaggio.

[12] HyperText Markup Language.

[13] eXtensible Markup Language.

- **Codifica URL,** per inviare brevi e semplici messaggi.
- **Multipart/form-data,** per l'invio di messaggi di grandi dimensioni in caratteri diversi da quelli **ASCII**.
- **Raw,** per dati non codificati.
- **Binary,** per l'invio di audio, immagini e video.[18]

Si utilizzeranno in particolare i formati Multipart/form-data quando si caricheranno dei file e raw con rappresentazione JSON quando si necessiterà di inviare informazioni al server.

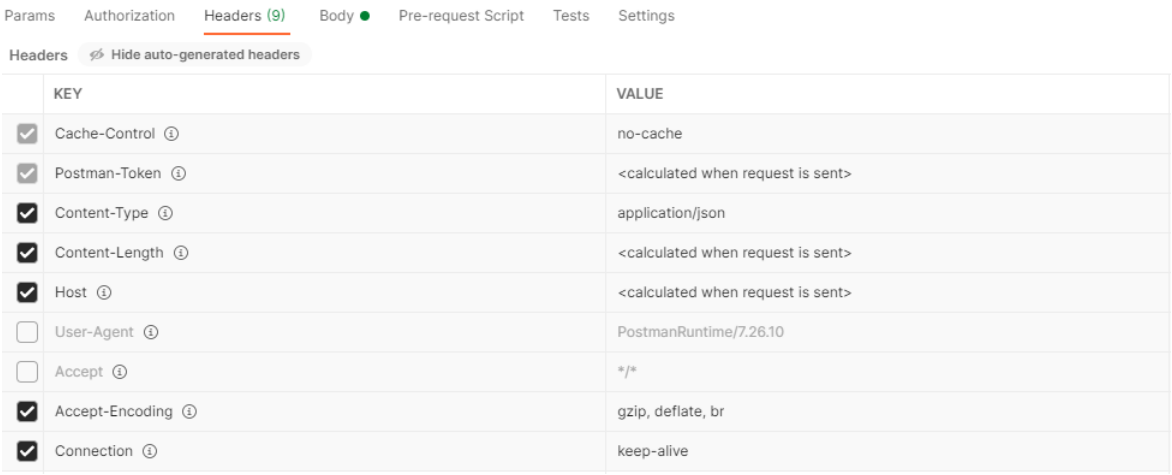

Figura 2.4. Headers utilizzati per le richieste con Postman.

Facendo riferimento all'immagine, si noti la disabilitazione del cache-control per rispettare le specifiche dell'architettura REST, l'abilitazione del Postman-Token che va ad aggiungere un ID ad ogni richiesta in modo che siano ben distinguibili da parte del server, il settaggio di Content-Type su application/JSON per segnalare che i messaggi inviati saranno in tale formato, l'abilitazione dell'accept-encoding per permettere al server, se necessario, di comprimere i file con gzip, deflate o br e inviarli a Postman e l'attivazione del keep-alive Connection per tenere aperto il canale di comunicazione client-server dopo l'esecuzione della richiesta in modo da velocizzare tante richieste consecutive alla stessa API.

Per tenere traccia di tutti i dati che verranno inviati al server, si è scelto di utilizzare dei database relazionali di cui si parlerà nella sezione successiva.

2.4 MySQL

MySQL è un *Relational Database Management System* di proprietà di Oracle ed è il sistema più diffuso al mondo per la gestione di basi di dati. MySQL permette di creare e gestire tabelle formate da enormi quantità di dati, ottimizzate per le velocità di accesso, creando un'istanza su un server dedicato MySQL chiamato **MySQL Server**. La gestione avviene seguendo la filosofia dei database relazionali: essa stabilisce che i dati non vadano salvati in unica tabella ma in tabelle differenti, più piccole e strutturate in modo che, nel caso in cui ci siano valori presenti in più tabelle, essi siano in relazione con i rispettivi campi di un'altra tabella. In questo modo si riesce ad avere un sistema organizzato ed ordinato, senza perdita di dati. La scelta di MySQL ha permesso la creazione di un database *fota* con host in locale per la gestione dei dati derivanti da un costruttore di dispositivi IoT e dei dati provenienti dal firmwarista. [19]

Per la gestione del server si è utilizzata l'interfaccia grafica **MySQL Workbench** che ha permesso una facile costruzione dello schema delle tabelle mediante un apposito Wizard, scegliendo la tipologia della colonna ed il suo ruolo all'interno della tabella (es. chiave primaria). Inoltre si è spesso utilizzata la sezione di scripting SQL della Workbench per l'esecuzione di statement sulle tabelle, in particolar modo per verificare, in maniera grafica, il buon esito di un'operazione su di esse. [20]

Capitolo 3

Il progetto

3.1 Elaborazione dell'architettura

Si vuole realizzare un'applicazione end-to-end in grado di gestire l'aggiornamento remoto del firmware dell'infotainment per gli autoveicoli di una grossa azienda cliente. Per la realizzazione del progetto, si è supposta la presenza di due aziende esterne che collaborano al progetto, una che produca fisicamente le schede dei dispositivi da andare poi ad installare sugli autoveicoli ed una che impieghi i propri sviluppatori firmware per studiare le specifiche tecniche e i report sul funzionamento dei devices in modo che sviluppino nuovo firmware da implementare. Si suppone che tutti i dispositivi abbiano integrata una sim per la connessione ad internet e che la versione di fabbrica del firmware installata su ogni dispositivo venga memorizzata:

A. Nella memoria non volatile di ogni dispositivo.

B. In un file che l'azienda partner dovrà inviare ogni volta che immette nuovi device in magazzino, da installare poi sui veicoli.

La prima esigenza è stata quella di possedere un'interfaccia che potesse essere richiamata, quando necessario, sia dai firmwaristi in possesso di nuovo codice che dal costruttore che vuole immettere nuovi device in magazzino: questo fatto implicava la necessità di avere almeno due path distinti. Era inoltre necessaria un'interfaccia per i dispositivi client che segnalavano la loro entrata in funzione in modo da registrarli correttamente. Infine era necessario un ulteriore path, richiamabile da ogni dispositivo, per andare ad acquisire, qualora fosse presente, nuovo firmware. La soluzione è quella presentata nella sezione **2.2.1** ovvero un'API REST che sia in grado di servire

queste quattro tipologie di richieste. Dall'altra parte, si è costruita un'applicazione front-end da installare nei devices quando entrano in funzione, in grado di registrarsi/interfacciarsi con l'IoT hub come client e, conseguentemente, col back-end, per procedere al download del firmware quando necessario. L'architettura dell'intero sistema, *back-end + front-end* si struttura come nell'immagine seguente:

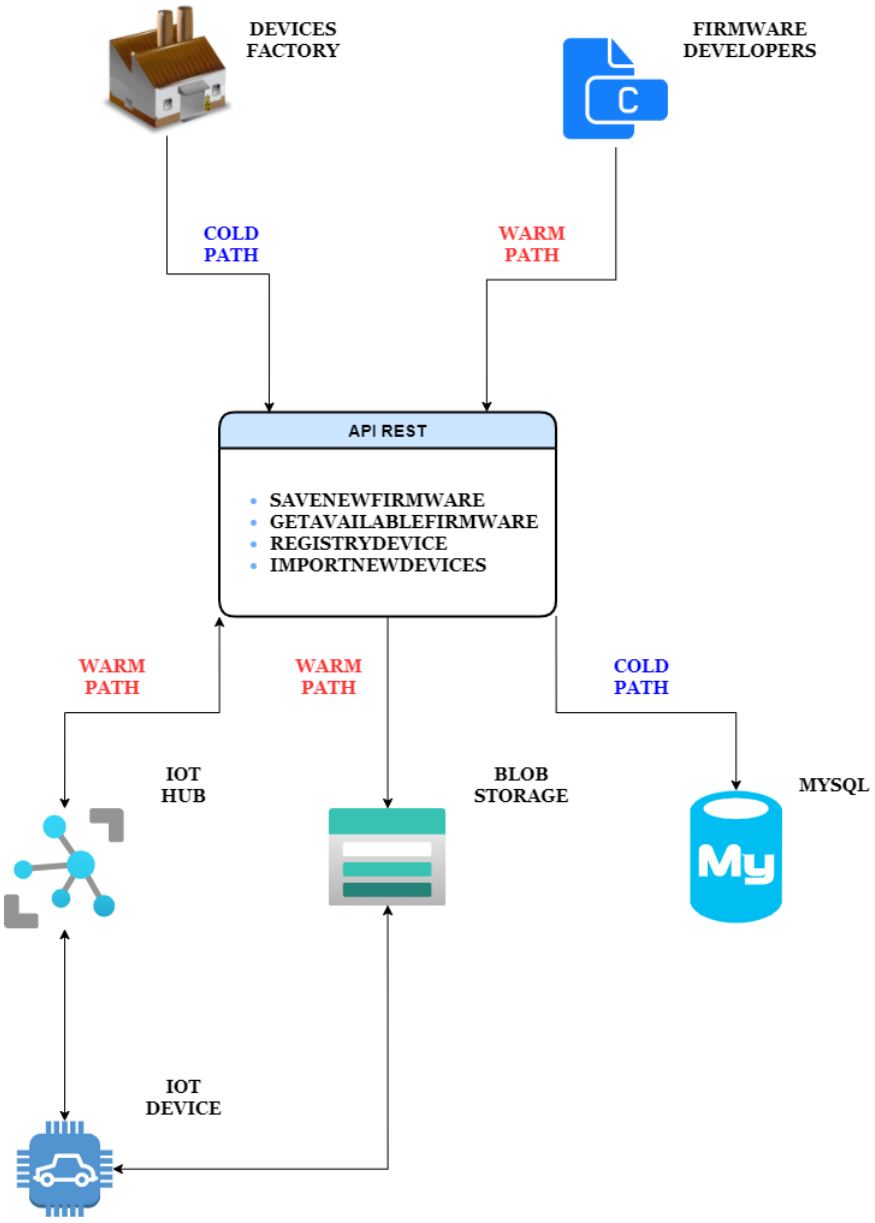

Figura 3.1. Architettura per l'aggiornamento del Firmware del dispositivo con API REST in evidenza.

Per **cold path** si intende un percorso dei dati non particolarmente critico, adatto quindi ad un processo di pura archiviazione come il caricamento dei dati sui nuovi dispositivi di fabbrica, in cui la latenza non rappresenta un problema; nel **warm path** invece i dati si muovono più velocemente e la latenza dovrà essere ridotta, in quanto bisognerà procedere con il download del firmware sui devices IoT. Si noti l'assenza di un **hot path** che sarebbe presente invece nel caso di applicazioni molto critiche che necessiterebbero di una costante **stream analysis** sui dati.

La progettazione è cominciata dal lato server, in particolare si è proceduto a creare le risorse Azure necessarie all'applicazione e a studiarne le SDK per Node JS.

3.2 Le risorse Azure

L'azienda per cui si deve creare l'applicazione, mette a disposizione un account Microsoft Azure: si vogliono dunque creare dei metodi con cui il back-end possa utilizzare le funzionalità di Azure ed in particolare utilizzare il PaaS IoT hub come canale comunicativo, da e verso i dispositivi, ed un account di archiviazione per andare a salvare tutto il nuovo firmware che viene caricato dai progettisti.

Per organizzare le risorse in maniera ordinata, si è proceduto a creare sul cloud, un **gruppo di risorse** che fungesse da contenitore per le soluzioni del progetto: in questo modo, i metadati di tutte le risorse appartenenti al gruppo vengono salvati ed archiviati nello stesso data center e risulta più efficiente l'aggiornamento o l'eliminazione delle risorse stesse. Il nome scelto per il gruppo di risorse è stato *ProgettoFota* e come luogo di creazione del gruppo si è scelto quello più vicino alla città di Torino ovvero **Francia Centrale**: Azure al momento del lavoro su questo progetto di Tesi, possedeva 60 data center in tutto il mondo, ognuno con un proprio sistema di networking, alimentazione, raffreddamento, in modo da non avere correlazione con un altro data center in caso di problemi tecnici. La scelta della località è fondamentale perché permette di avere una più alta disponibilità dei server e una riduzione della latenza di accesso se il data center scelto è fisicamente vicino al consumer che vuole usufruirne. [21]

Una volta costruito il gruppo di risorse, si è proceduto alla creazione di un account di archiviazione dal portale di Azure impostando le seguenti proprietà:

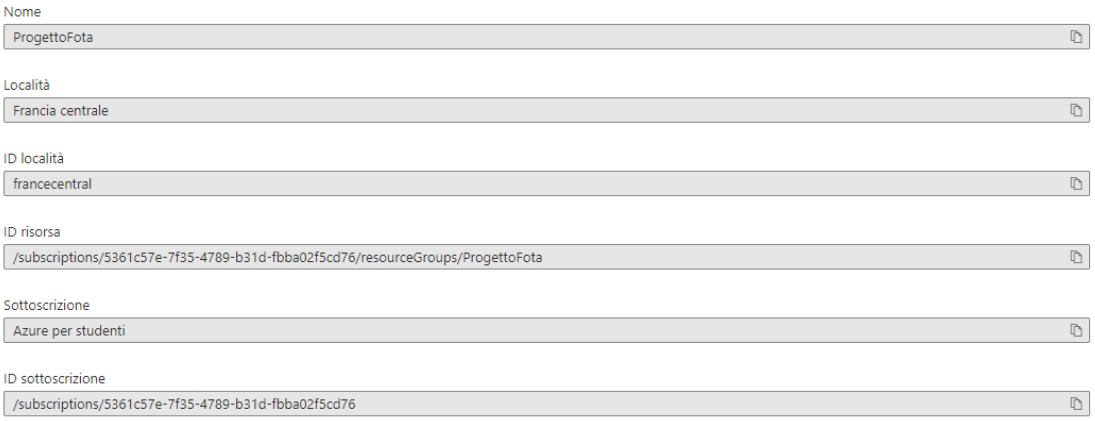

Figura 3.2. Overview delle proprietà del gruppo di risorse.

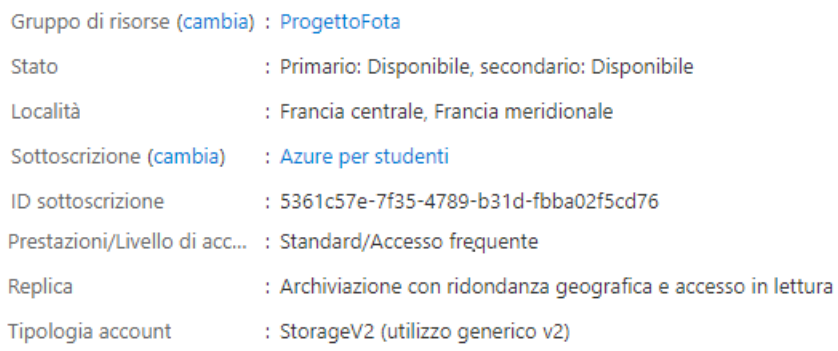

Figura 3.3. Informazioni generali sull'account di archiviazione *ProgettoFota*

Si notino le **prestazioni ad accesso frequente**, poiché lo storage farà parte del warm path dell'architettura, la tipologia di account per utilizzo generico dell'archiviazione delle risorse **StorageV2** e la ridondanza del tipo **RA-GRS** con cui dal data center Francia centrale viene effettuata una copia di backup asincrona nell'area secondaria selezionata, Francia meridionale; di seguito si può visualizzare la mappa con le zone appena citate.

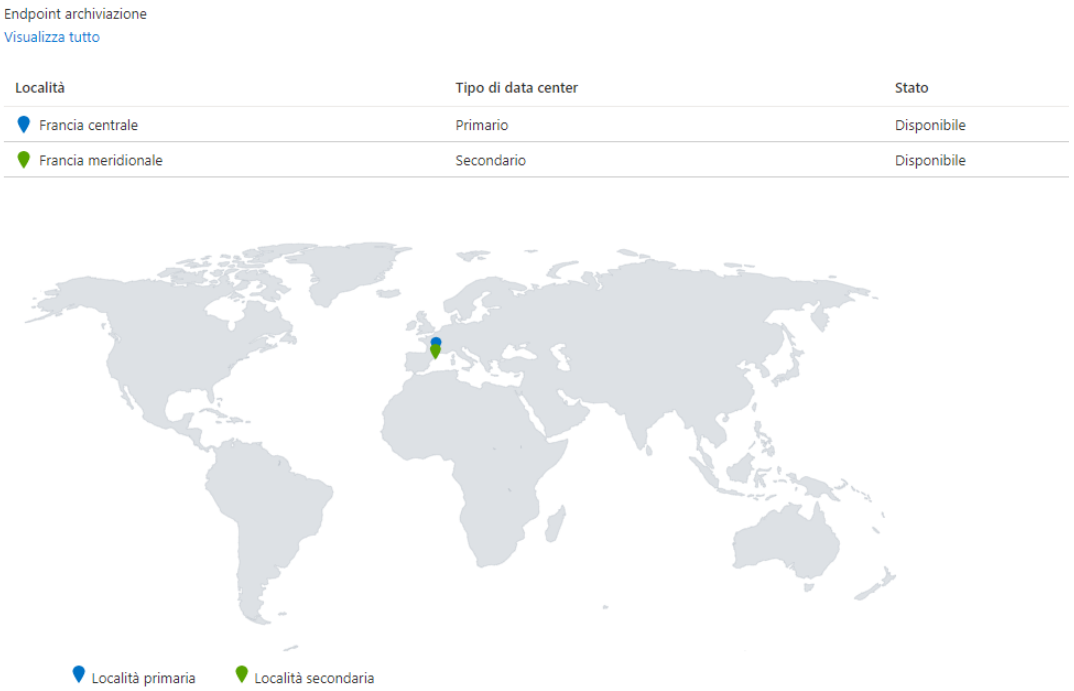

Figura 3.4. Mappa delle zone interessate dalla ridondanza geografica per l'account *ProgettoFota*

Dopo la configurazione dell'account di archiviazione, si è proceduto, dal portale di Azure, alla creazione di un contenitore per blob denominato *api-fota* in cui si andrà a lavorare attraverso l'applicazione di back-end, il cui funzionamento verrà spiegato nella sezione **3.4.2**.

In seguito si è creata la risorsa IoT hub, scegliendo come nome della risorsa *fota* e selezionando, come per lo storage, **Francia centrale** come posizione geografica del data center; si è inoltre impostato un piano tariffario che permetta la connessione simultanea di 500 dispositivi IoT per un totale di 8000 messaggi al giorno, condizioni sufficienti per la simulazione del progetto.

Poiché si vuole utilizzare l'hub come broker delle comunicazioni tra i devices ed il back-end, rivestono particolare importanza i criteri di accesso condiviso all'hub: un dispositivo, per comunicare col cloud, necessiterà di una chiave primaria riconoscibile dal servizio e dell'endpoint dell'hub a cui vuole accedere. Queste caratteristiche sono sintetizzate da una **stringa di connessione** generabile dal portale che, se posseduta dal device, ne consente l'autenticazione.

Esistono diversi permessi con cui un client generico può autenticarsi:

- **iothubowner**, permessi: lettura registro, scrittura registro, connessione servizio, connessione dispositivo.
- **service**, permessi: connessione servizio.
- **device**, permessi: connessione dispositivo.
- **registryRead**, permessi: lettura registro.
- **registryReadWrite**, permessi: lettura registro, scrittura registro.

Dove per "registro" si intende il registro delle identità dell'IoT hub in cui viene allocata una sezione ogni volta che un dispositivo si registra.

Si riporta di seguito un dettaglio della generazione delle chiavi per un servizio di tipo device.

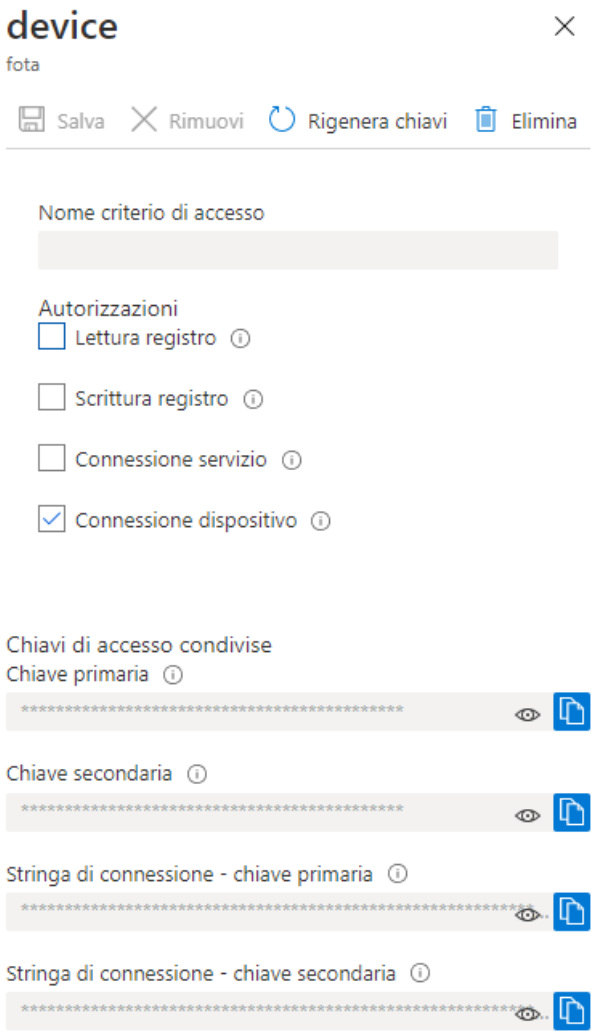

Figura 3.5. Criterio di accesso di tipo device per IoT hub.

3.3 I database

Una volta create le risorse in Azure, si è proceduto con la preparazione dei database: si è quindi creato un server MySQL in *localhost*, sufficiente alle esigenze di questo progetto, con uno schema chiamato *fota*, dove costruire le due tabelle necessarie. Una tabella è stata utilizzata per il salvataggio delle informazioni di base provenienti dal costruttore sullo stato iniziale del dispositivo e può essere consultata nel caso in cui si dovesse risalire alla

versione iniziale del firmware installato sul device. La seconda tabella è stata utilizzata , a livello operativo, per andare a controllare, con apposite query, la disponibilità del firmware nello storage e la compatibilità delle versioni con i dispositivi. Le due tabelle sono state chiamate *productordevices* e *firmware*.

productordevices	
serialNumber	varchar(500) **PK**
classDevice	varchar(300)
fwVersion	varchar(45)

Tabella 3.1. Struttura di riferimento per la tabella SQL **productordevices** in cui viene indicato la formattazione di ogni colonna.

firmware	
fileNameID	varchar(500) **PK**
urlFirmware	varchar(500)
realeaseDate	datetime
uploadDate	datetime
version	varchar(45)
classDevice	varchar(300)

Tabella 3.2. Struttura di riferimento per la tabella SQL **firmware** in cui viene indicato la formattazione di ogni colonna.

La scelta della **chiave primaria** [1] nella tabella *productordevices* è stata il seriale del dispositivo, permettendo un'identificazione unica e senza ambiguità; viene inoltre comunicata dal costruttore la *classe* del dispositivo, poiché non esisterà un firmware update dedicato per singolo dispositivo ma soltanto per una classe di dispositivi; si ricorda che la versione del firmware iniziale sarà installata al momento dell'uscita del device dalla fabbrica.

Per la tabella *firmware* è stato scelto, come chiave primaria, l'ID del nome del file di aggiornamento, che verrà costruito all'interno del codice prendendo il nome del file caricato dal firmwarista ed accodandogli un codice generato casualmente; sono presenti un campo per evidenziare l'URL in cui viene salvata la risorsa all'interno del blob storage, le date di rilascio e di caricamento

[1] Attributo che permette l'identificazione univoca del record.

sul blob storage dell'aggiornamento, la versione del firmware e la classe di dispositivi a cui essa fa riferimento.

Le tipologie di dato sono espresse in *varchar*, la quale indica la lunghezza massima in byte che possono occupare i caratteri di quel campo. Il numero di byte è stato selezionato in maniera proporzionale alla lunghezza dei campi e sempre per eccesso per non avere troncamento dei dati. I campi *datetime* indicano una data nel formato *AAAA-MM-GG hh:mm:ss[.mmm]*.

3.4 Il codice - Lato server

Per la stesura del codice si è utilizzato l'editor Visul Studio Code di Microsoft. Utilizzando il manager di pacchetti **npm** di Node JS, si è provveduto, durante il progetto, all'installazione delle librerie necessarie all'elaborazione dell'applicazione. Le librerie di npm vengono create dalla community di sviluppatori di Node che le mettono a disposizione di npm, il quale le inserisce in un registro in cui associa ad ogni libreria le informazioni sulla versione e in seguito rende consultabile l'intera libreria sul proprio sito `https://www.npmjs.com/`. Per installare le librerie, si è provveduto ad utilizzare l'interfaccia client da linea di comando.

Per inizializzare il progetto, si è creato un file *package.json* standard di configurazione attraverso il comando *npm init -y*: ogni volta che si installerà una libreria, al file *package.json* verrà aggiunta una riga in cui verrà indicata la nuova installazione.

La prima scelta progettuale a livello di codice è stata quella di dividere l'applicazione in due sezioni: una che si occupi del **lato server** e l'altra, da installare sul dispositivo, che si occupi del **lato device**.

Per rendere i file del progetto più flessibili, si è scelto di andare a salvare alcuni parametri costanti e ricorrenti all'interno di due file *config.js*, uno per la parte server ed uno per la parte client: ogni volta che uno di questi parametri dovrà essere utilizzato, verrà richiamato il rispettivo file e con una notazione a punti si farà accesso al parametro desiderato.

Poichè il linguaggio Javascript si basa sul paradigma OOP [2], si è proceduto ad elaborare il codice con un approccio **bottom-up**. Il primo metodo ad essere stato elaborato è l'interfaccia col database MySQL.

[2]Oriented Object Programming

Osservazione 1 Per realizzare l'asincronismo in Javascript ci sono tre metodologie:

A **Funzioni di callback:** si tratta di una tipologia di funzioni che viene triggerata quando un'altra funzione, di cui la callback è un parametro, entra in una determinata condizione. La funzione esterna non verrà completata finchè non sarà completata la callback.

B **Promises:** si tratta di oggetti che possono acquisire tre stati (**pending, fullfilled, rejected**). Vengono create come fossero delle funzioni che acquisiscono due parametri, **resolve, reject** i quali descrivono il caso di successo della promise o della sua non realizzazione; si devono perciò descrivere i rami sia per il caso fullfilled che per il caso rejected. Questa tecnica permette di annidare diverse promise tra loro in maniera più ordinata rispetto alle funzioni di callback.

C **Costrutto async-await:** genera direttamente una funzione asincrona che ritorna un valore di tipo promise.

Nel corso del progetto sono state utilizzate tutte tre le soluzioni, prediligendo async-await nel caso in cui si abbiano diverse funzioni annidate, le promises nel caso si debbano sempre eseguire due funzioni una di seguito all'altra mentre le callback sono state utilizzate in prevalenza con i metodi che richiedevano una condizione di trigger.

3.4.1 dbManager.js

Si è installata attraverso npm la libreria *mysql* tra le dependencies del *package.json*. [22] La libreria si comporta da driver nei confronti di un server MySQL e permette di andare a lavorare sullo schema *fota* creato in precedenza.

```
1  const config = require('./config.js');
2  const mysql = require('mysql');
3  const formatDate = require('./utils/date.js');
4  //Create the pool
5  const pool = mysql.createPool({
6      host: config.POOL_CONFIG.host,
7      port: config.POOL_CONFIG.port,
8      user: config.POOL_CONFIG.user,
9      database: config.POOL_CONFIG.database,
10     connectionLimit : config.POOL_CONFIG.connectionLimit,
11     multipleStatements: config.POOL_CONFIG.multipleStatements
```

```
12 });
13
14
15 pool.on('release', function (connection) {
16   console.log('Connection %d released', connection.threadId);
17 });
18
19 pool.on('acquire', function (connection) {
20   console.log('Connection %d acquired', connection.threadId);
21 });
22
23 pool.on('connection', function (connection) {
24   console.log('Connection %d created', connection.threadId);
25 });
26
27 pool.on('enqueue', function () {
28   console.log('Waiting for available connection slot');
29 });
30
31 //Execute the query passed in input
32 function executeQuery(query) {
33   return new Promise(function(resolve, reject) {
34     pool.getConnection(function(err, connection){
35       if (err) {
36         // not connected!
37         console.log('Error during the opening of the
               connection with database: ' + err);
38         reject(err);
39       }
40       // Use the connection
41       connection.query(query, function (error, results,
            fields) {
42         // When done with the connection, release it.
43         console.log(query + ' executed');
44         connection.release();
45         // Handle error after the release.
46         if (error) {
47           console.log('Error during the query: ' + error);
48           reject(error);
49         } else {
50           resolve(results);
51         }
52       });
53     });
54   });
55 }
```

```
56
57 //Check the device table if there is an earlier version of
       the firmware
58 async function checkVersions(classDevice, versionInstalled){
59   try{
60     let query = "SELECT * FROM fota.firmware WHERE
         classDevice = '"+ classDevice + "' AND version > '"+
         versionInstalled +"' ORDER BY version DESC LIMIT 1;";
61     return executeQuery(query)
62     .then(function(results){
63         if(results.length != 0){
64         var latestFirmware = results[0].version;
65         var idFile = results[0].fileNameID;
66         var url = results[0].urlFirmware;
67         var update = {
68           latestFirmware,
69           idFile,
70           url
71         }
72         return update;
73       }
74       else{
75         console.log('Device is already updated');
76         return null;
77       }
78     })
79     .catch(function(error){
80       console.log('An error occured in the checking of
           firmware versions: ' + error.toString());
81     })
82   }catch(errors){
83     console.log(errors);
84   }
85 }
86
87 //When a new firmware is available
88 //Report the new firmware in the table
89 async function newFirmware(postRequest, url){
90   try{
91     let query1 = 'CREATE TABLE IF NOT EXISTS fota.firmware (`
         fileNameID` VARCHAR(500) NOT NULL,`urlFirmware`
         VARCHAR(500) NULL,`realeaseDate` datetime NULL, `
         uploadDate` datetime NULL, `version` VARCHAR(45) NULL,
          `classDevice` VARCHAR(300) NULL, PRIMARY KEY (`
         serialNumber`));'
92     executeQuery(query1)
```

```
93      .then(function(results){
94        console.log(results);
95        var today = new Date();
96        let query2 = "INSERT INTO fota.firmware(fileNameID,
              urlFirmware, releaseDate, uploadDate, version,
              classDevice) VALUES ( '" + postRequest.fileNameID +
              "' , '" + url + "' , '" + formatDate(new Date((
              postRequest.releaseDate))) + "' , '" + formatDate(
              today.toUTCString()) + "' , '" + postRequest.version
               + "', '" + postRequest.classDevice + "');";
97        executeQuery(query2)
98        .then(function(results){
99          console.log(results);
100       })
101       .catch(function(error){
102         console.log('An error occured during the insertion of
                new records inside firmware table: ' + error);
103       })
104     })
105     .catch(function(error){
106       console.log('Error during the creation of table
              firmware' + error)
107     })
108   }catch(error){
109     console.log('Error in the table firmware: ' + error);
110   }
111 }
112
113 //Create a new table for the devices provided from a
      productor
114 async function initializeTableDevices(data){
115   try{
116     let query1 = 'CREATE TABLE IF NOT EXISTS fota.
          productordevices (`serialNumber` VARCHAR(500) NOT NULL
          ,`classDevice` VARCHAR(300) NULL,`fwVersion` VARCHAR
          (45) NULL, PRIMARY KEY (`serialNumber`));'
117     executeQuery(query1)
118     .then(function(results){
119       console.log(results);
120       let query2 = 'INSERT INTO fota.productordevices(
              serialNumber, classDevice, fwVersion) VALUES("'+
              data.serialNumber + '", "'+ data.classDevice + '", "
              '+ data.fwVersion + '")';
121       executeQuery(query2)
122       .then(function(results){
123           console.log(results);
```

```
124         })
125       .catch(function(error){
126           console.log('An error occurred during the insertion
                  of new records inside table productordevices: '
                  + error);
127         })
128     })
129     .catch(function(error){
130       console.log('Error during the creation of table
             productordevices' + error);
131     })
132   }catch(error){
133     console.log('Error in the table productordevices' + error
          );
134   }
135 }
136
137 exports.executeQuery = executeQuery;
138 exports.checkVersions = checkVersions;
139 exports.newFirmware = newFirmware;
140 exports.initializeTableDevices = initializeTableDevices;
```

Listing 3.1. Codice Javascript per il file dbManager.js

Nelle righe **1-3** si vanno ad importare il file di configurazione in cui sono contenuti i parametri della connessione col server MySQL, la libreria *mysql* ed un modulo *date.js* scritto appositamente per la formattazione dei dati in un formato compatibile con il tipo SQL *datetime* di cui si parlerà in fondo al paragrafo.

Il primo metodo richiamato dalla libreria *mysql* è alla riga **5** ed è *createPool*: il **pooling della connessione** è una tecnica che viene utilizzata per non ristabilire ogni volta una nuova connessione col database, ma mantenere i dati salvati nella cache del server, permettendo un considerevole aumento della velocità di esecuzione dei comandi ricevuti dal client e riducendo drasticamente la latenza. Al metodo viene passata una configurazione JSON, andando ad attingere dal file *config.js*, che predispone il nome dell'host del database, la porta utilizzata per la comunicazione, l'username e la password per l'autenticazione al server, il nome dello schema a cui si vuole accedere (*fota*), il numero di connessioni contemporanee ammesse per questo pool per non doverne creare uno nuovo e infine un parametro *multipleStatements* che indica la possibilità di eseguire più query contemporaneamente sul database.

Quando si verifica uno dei quattro eventi nelle righe **15-29**, viene stampato a video un messaggio indicando l'azione che il client ha appena compiuto nei confronti del server:

- **release** indica che una richiesta è terminata e la rispettiva connessione è di nuovo libera per essere utilizzata da un nuovo processo.

- **acquire** indica che i dati per la connessione sono stati correttamente acquisiti.

- **connection** indica che la connessione del pool al server MySQL è avvenuta correttamente.

- **enqueue** indica che una richiesta è stata accodata perchè tutte le connessioni del pool sono occupate.

La funzione *executeQuery* al rigo **32** esegue la query SQL passata ritornando come valore una promise che avrà uno stato fullfilled in caso di esecuzione corretta sul database, *rejected* in caso di errore. La funzione richiama il metodo *getConnection* per connettere il pool al server MySQL ritornando un errore in caso di mancata apertura della connessione. Se la connessione è stata aperta correttamente, al rigo **41** la funzione *query* prende il parametro omonimo passato alla funzione e lo applica al database: in caso di successo la query eseguita verrà stampata a video e la connessione verrà liberata e reintrodotta nel pool e la funzione ritornerà i risultati della query; in caso contrario, la query non verrà eseguita e verrà stampato a schermo il relativo errore.

Al rigo **58** viene introdotta una nuova funzione chiamata *checkVersions* che andrà ad interrogare la tabella firmware per verificare se esista un aggiornamento più recente per la classe del dispositivo su cui si vuole intervenire: la funzione riceve in ingresso la versione del firmware comunicata dal device e la sua classe. I due parametri citati verranno introdotti all'interno della query SQL dichiarata al rigo **60**: si noti come il comando *ORDER BY version DESC LIMIT 1* intervenga sui risultati della query riportando soltanto il valore più grande della versione , corrispondente all'ultima release. A questo punto viene richiamata la *executeQuery* che eseguirà la query e ritornerà una promise: se lo stato della promise è fullfilled, la funzione controlla se la query ha prodotto un risultato che sarà unico per la formulazione della query ivi sopra e se ne acquisiranno i valori relativi alla versione dell'ultimo update disponibile, al nome del file d'aggiornamento e all'URL in cui si trova la risorsa nel blob storage. Se invece la query è andata a buon fine ma non ha prodotto risultati significa che il device è già aggiornato all'ultima versione. In coda alla funzione sono presenti i catch per la gestione degli errori generici.

Al rigo **89** viene dichiarata la funzione *newFirmware* la quale ha il compito di aggiornare il database con le specifiche sul nuovo firmware: la funzione accetta in ingresso un parametro corrispondente al body della richiesta inviata dal firmwarista e l'URL nel quale viene salvata la risorsa. Vengono quindi inserite due promise in cascata, ognuna con il compito di eseguire una query: la query al rigo **91**, se non ne esiste già una, crea una tabella firmware identica a quella discussa nel paragrafo **3.4**, mentre la seconda al rigo **96** prende i parametri in ingresso alla funzione e li inserisce nella tabella. Va da sé che qui la *executeQuery* viene richiamata due volte. Anche in questo caso i risultati delle operazioni vengono stampati a schermo in casso di fullfilled delle promise mentre vengono elaborati gli errori in caso di rejection.

Al rigo **114** si instanzia una funzione simile a *newFirmware* che andrà però ad occuparsi della tabella productordevices; anche in questo caso si costruisce una query che vada a creare, nel caso in cui non esista, una tabella productordevices identica a quella esposta al paragrafo xxx. Se non ci sono problemi e la prima query viene applicata, la prima promise ritorna un fullfilled permettendo di entrare nella seconda promise che costruisce una nuova query che inserisce i dati inviati dal costruttore nella tabella. Anche qui i catch in coda alla funzione vengono utilizzati per la gestione dei relativi errori.

Le righe **137-140** vengono utilizzate per esportare le funzioni appena discusse.

date.js

Il file trasforma la stringa passata come input in una stringa salvabile in formato *varchar*, andando a valutare se i campi anno, mese, giorno, ora, minuto e secondo esistono nella stringa passata in ingresso; nel caso in cui siano presenti, si formattano rispetto ad uno **0** di origine. Il parametro ritornato sarà la data formattata costituita dall'unione delle singole stringhe.

```
1   /**
2    * Get a date as string in format YYYY-MM-DDThh:mm:ssZ
3    */
4   function getFormattedDate(date) {
5       date = new Date(date);
6       var year = date.getFullYear();
7
8       var month = (1 + date.getMonth()).toString();
9       month = month.length > 1 ? month : '0' + month;
10
11      var day = date.getDate().toString();
```

```
12          day = day.length > 1 ? day : '0' + day;
13
14          var hours = date.getHours().toString();
15          hours = hours.length > 1 ? hours : '0' + hours;
16
17          var mins = date.getMinutes().toString();
18          mins = mins.length > 1 ? mins : '0' + mins;
19
20          var seconds = date.getSeconds().toString();
21          seconds = seconds.length > 1 ? seconds : '0' +
                seconds;
22
23          var formattedDate = year + '-' + month + '-' + day +
                'T' + hours + ':' + mins + ':' + seconds;
24          return formattedDate;
25      }
26
27      module.exports = getFormattedDate;
```

Listing 3.2. Codice Javascript per il file date.js

Si passerà adesso a discutere il file atto all'interlocuzione col blob storage.

3.4.2 blob.js

Per l'elaborazione del modulo si sono utilizzate due SDK fornite da Microsoft, *@azure/storage-blob* [23] e *azure-storage* [24]: le due librerie lavorano allo stesso modo e permettono di effettuare il CRLUD [3] delle risorse archiviate nel blob e di gestire l'account di storage ma hanno un'implementazione dei metodi leggermente diversa, per questo motivo sono state usate entrambe.

```
1  const {BlobServiceClient}  = require('@azure/storage-blob');
2  const azure = require('azure-storage');
3  const config = require('./config.js');
4  const AZURE_STORAGE_CONNECTION_STRING = config.
       AZURE_STORAGE_CONNECTION_STRING;
5
6  async function createContainer(name) {
7
8      // Create the BlobServiceClient object which will be used
           to create a container client
9      const blobServiceClient = BlobServiceClient.
           fromConnectionString(AZURE_STORAGE_CONNECTION_STRING);
10
```

[3] Create/Read/List/Update/Delete

```
11      // Create a unique name for the container
12      const containerName = name;
13
14      console.log('\nCreating container...');
15      console.log('\t', containerName);
16
17      // Get a reference to a container
18      const containerClient = blobServiceClient.
           getContainerClient(containerName);
19
20      // Create the container
21      const createContainerResponse = await containerClient.
           createIfNotExists();
22      console.log('Container was created successfully.
           requestId: ', createContainerResponse.requestId);
23
24      return containerClient;
25  }
26
27  //  Create a blob based on the file
28  //  That has been provided
29  async function createBlob(containerClient, name, path) {
30    // Create a unique name for the blob
31    const blobName = name;
32    const blobService = azure.createBlobService(
         AZURE_STORAGE_CONNECTION_STRING);
33
34    console.log('Uploading to Azure storage as blob:\n\t',
         blobName);
35
36    // Upload data to the blob
37    blobService.createBlockBlobFromLocalFile(containerClient.
         containerName, blobName, path, function(error){
38      if(error){
39        console.log('Error during the upload in the blob
             storage' + error);
40        return new Error(error);
41      } else console.log('Successfull update');
42    });
43
44    // Get the block blob client
45
46    const blockBlobClient = containerClient.getBlockBlobClient(
         blobName);
47
48    return blockBlobClient.url;
```

```
49 }
50
51 //Show the blobs available
52 //In the Azure Blob Storage
53 async function showBlobs(containerClient){
54   var blobList = [];
55
56   console.log('Listing the blobs...');
57
58 // List the blob(s) in the container.
59   for (const blob of containerClient.listBlobsFlat()) {
60       console.log('\t', blob.name);
61       blobList.push(blob.name);
62   }
63   return blobList;
64 }
65 // Create the token
66 async function generateSasToken(containerClient, container,
     blobName, permissions) {
67   const blobService = azure.createBlobService(
       AZURE_STORAGE_CONNECTION_STRING);
68
69   // Create a SAS token that expires in an hour
70   // Set start time to five minutes ago to avoid clock skew.
71   var startDate = new Date();
72   startDate.setMinutes(startDate.getMinutes() - 5);
73   var expiryDate = new Date(startDate);
74   expiryDate.setMinutes(startDate.getMinutes() + 60);
75
76   permissions = permissions || azure.BlobUtilities.
       SharedAccessPermissions.READ;
77
78     var sharedAccessPolicy = {
79         AccessPolicy: {
80             Permissions: permissions,
81             Start: startDate,
82             Expiry: expiryDate
83         }
84     };
85
86     var sasToken = blobService.generateSharedAccessSignature(
         container, blobName, sharedAccessPolicy);
87
88     return {
89         token: sasToken,
```

```
90          uri: blobService.getUrl(container, blobName, sasToken
                , true)
91      };
92  }
93
94
95  exports.createContainer = createContainer;
96  exports.createBlob = createBlob;
97  exports.showBlobs = showBlobs;
98  exports.generateSasToken = generateSasToken;
```

Listing 3.3. Codice Javascript per il file blob.js

Come si era detto in precedenza, per effettuare l'autenticazione al blob storage, si fornisce al modulo la stringa di connessione come avviene al rigo **4**, dal file *config.js* creato precedentemente. Al rigo **6** si instanzia una funzione asincrona *createContainer*, necessaria, nel caso non ne esista già uno, di creare un contenitore all'interno dello storage con il nome passato come paramentro d'ingresso. Al rigo **9** si utilizza il metodo *fromConnectionString* per autenticarsi al servizio tramite stringa di connessione ed al rigo **18** si chiama il metodo *getContainerClient* per costruire un'istanza del contenitore, che verrà fisicamente creato nel blob storage col successivo metodo al rigo **21** *createIfNotExists*. Se l'operazione è andata a buon fine, un messaggio viene stampato a video e la funzione ritorna l'istanza del contenitore che verrà riutilizzata nei prossimi metodi.

Il metodo *createBlob* corrispondente al rigo **29** va a creare il blob all'interno del contenitore costruito dalla funzione analizzata sopra: si forniscono infatti come parametri d'ingresso l'istanza *containerClient*, il nome dell'oggetto blob e il path in cui è salvato il file da caricare. Anche qui si effettua una connessione al servizio di archiviazione di Azure avendo però già creato l'istanza del contenitore; si passa quindi come parametro la proprietà *containerName* del *containerClient* ad un metodo chiamato *createBlockBlobFromLocalFile*, insieme al nome del blob e al percorso di archiviazione locale del file da caricare. In caso di successo del caricamento, si crea un oggetto *blockBlobClient* attraverso il metodo *getBlockBlobClient* che permette di estrapolare l'URL della risorsa appena archiviata sul cloud e ritornarlo dal metodo come stringa.

Al rigo **53** è presente un metodo di utility *showBlobs* che è stato molto utile nell'elaborazione di questo progetto di tesi ma che non è stato usato nella fase finale; il metodo permette, attraverso il passaggio come parametro d'ingresso dell'istanza *containerClient*, di iterare su tutto l'array di oggetti blob presenti nel contenitore, andando ogni volta ad aggiornare una lista con i rispettivi nomi e, ad iterazione conclusa, di stamparne a schermo i nomi.

Il metodo al rigo **66** è *generateSasToken* ed è di fondamentale importanza perché permette di autenticare e di abilitare al download della risorsa soltanto i client in possesso di un token che si andrà qui a generare; si notino sempre come parametri d'ingresso l'istanza del container, il nome del container, il nome del blob e i permessi di accesso. Come nelle precedenti funzioni del modulo *blob.js*, anche qui ci si autentica al servizio di archiviazione attraverso la stringa di connessione. Alle righe **71-74** si creano degli oggetti di tipo *Date* per catturare l'istante di chiamata del metodo; si è quindi scelto di settare l'inizio della validità a 5 minuti prima della chiamata del metodo, in modo da evitare qualsiasi skew a livello temporale, e a 60 minuti la validità del token. La dichiarazione al rigo **76** indica che se non vengono passati altri permessi in input, il permesso di default a cui si darà l'accesso sarà in lettura. Si procede quindi alla creazione di un JSON in cui vengono riassunte le regole discusse finora, che verrà passato al metodo *generateSharedAccessSignature* assieme al nome del contenitore ed al nome del blob. Infine la funzione ritorna un JSON con il token per l'autenticazione e l'URI già dotato di token per l'accesso diretto alla risorsa. Anche in questo file sono presenti gli *exports* per permettere l'accesso esterno alle funzioni.

3.4.3 hub.js

Per l'interfaccia con l'IoT hub si è utilizzata la SDK di Microsft Azure *azure-iothub* per Node JS [25]: in particolare, dopo aver aggiunto la libreria al progetto, si è importata la classe Registry per permettere l'accesso all'hub da parte del back-end tramite protocollo HTTPS.

```
1  const config = require('./config.js');
2  const Registry = require('azure-iothub').Registry;
3
4  var connectionString = config.IOT_HUB_CONNECTION_STRING;
5  var registry = Registry.fromConnectionString(connectionString
       );
6
7  // Check if the device that is performing the request
8  // Is already registerered to the iot hub
9  function checkID(deviceId){
10     let query = registry.createQuery("SELECT * FROM devices
           WHERE devices.deviceId = '"+ deviceId +"'" );
11     return new Promise(function(resolve, reject){
12       query.nextAsTwin(function onResults(err, results){
13         if (err) {
14           console.log('Error during the check ID phase ' +
               err);
```

```
15            reject(err);
16        } else {
17            var deviceIdFromTwin = null;
18            if(results == []){
19              resolve(deviceIdFromTwin)
20            }else{
21              results.forEach(function (twin) {
22                console.log('Device id in the twin ' , JSON.
                     stringify(twin.deviceId, null, 1));
23                deviceIdFromTwin = twin.deviceId;
24            });
25            }
26          }
27          resolve(deviceIdFromTwin);
28        })
29      })
30  }
31
32  //Send a notification of availability
33  //To all the devices that belong to that class
34  function checkClassToNotify(classDevice){
35    var query = registry.createQuery("SELECT * FROM devices
        WHERE devices.properties.reported.classDevice = '"+
        classDevice +"'" );
36    return new Promise(function(resolve, reject){
37      query.nextAsTwin(function onResults(err, results){
38        if (err) {
39          console.log('Error during the checkClass phase' + err
              );
40          reject(err);
41      } else {
42          if(results == []){
43            resolve(null)
44          }else{
45            let patch = {
46              properties: {
47                desired: {
48                  firmware: {
49                    status: 'available'
50                  }
51                }
52              }
53            }
54            results.forEach(function(twin) {
55              twin.update(patch, function(err) {
56                if (err) {
```

```
57                  console.log('Add configuration failed: ' +
                        err);
58                  reject(err);
59                } else {
60                  console.log('Add configuration succeeded for
                        device ' , twin.deviceId);
61                }
62              });
63          });
64            resolve(results);
65          }
66        }
67      })
68    })
69  }
70
71
72  exports.checkID = checkID;
73  exports.checkClassToNotify = checkClassToNotify;
```

Listing 3.4. Codice Javascript per il file hub.js

L'autenticazione viene effettuata sempre tramite la stringa di connessione, importata dal file di configurazione, e permette di costruire un oggetto *registry* dopo aver effettuato l'autenticazione. La funzione al rigo **9** è chiamata *checkId* e viene utilizzata per l'interrogazione dei dispositivi gemelli. La funzione riceve in ingresso, come parametro, l'id del dispositivo che si vuole controllare e lo inserisce in una query: la query controlla se tra i devices registrati nell'iot hub ce ne sia uno con l'id richiesto. La query viene poi inserita all'interno di una promise dentro cui viene richiamato il metodo *nextAsTwin* di *createQuery* che interroga i dispositivi gemelli. Quando i risultati dell'interrogazione sono disponibili, viene attivata la callback che gestisce l'eventuale errore oppure controlla se la query ha portato almeno un risultato; se il vettore dei risultati è vuoto, la promise ritorna un valore **null**, se invece il device esiste ed è registrato all'hub, la promise ritorna il suo *deviceIdFromTwin*. Al rigo **34** è presente una funzione *checkClassToNotify* il cui compito è quello di notificare a tutti i dispositivi registrati appartenenti ad una determinata classe, passata come parametro d'ingresso, la presenza di un nuovo firmware disponibile per il download. La query sui dispositivi gemelli ricerca tutti i dispositivi che nelle loro **reported properties** hanno segnalato l'appartenenza a quella classe. Si realizza, come nella funzione precedente, una promise e si richiama il metodo *nextAsTwin* di *createQuery*: la promise sarà in stato di rejection in caso di errore, altrimenti analizzerà i risultati

prodotti dalla query. Se il vettore dei risultati è vuoto, la promise ritornerà un valore **null** altrimenti preparerà una patch in formato JSON (rigo **45**) ponendo lo stato di disponibilità del firmware come **available**; al rigo **54**, la funzione itera sul vettore dei risultati della query e carica la patch nelle **desired properties** dei dispositivi gemelli di ognuno di essi. Se l'operazione ha avuto successo, la promise entra in stato di fullfilled e ritorna i risultati della query. Le due funzioni elencate, vengono rese disponibili tramite i metodi *exports* in coda al file.

3.4.4 Services e Validators

Si sono elencate fino ad ora le funzioni che dovrà gestire il server; si vuole adesso indicare un'architettura per rendere il server resiliente verso il traffico in ingresso andando a filtrare le richieste che arrivano. Soltanto se una richiesta soddisfa determinate condizioni allora potrà essere servita dal server:

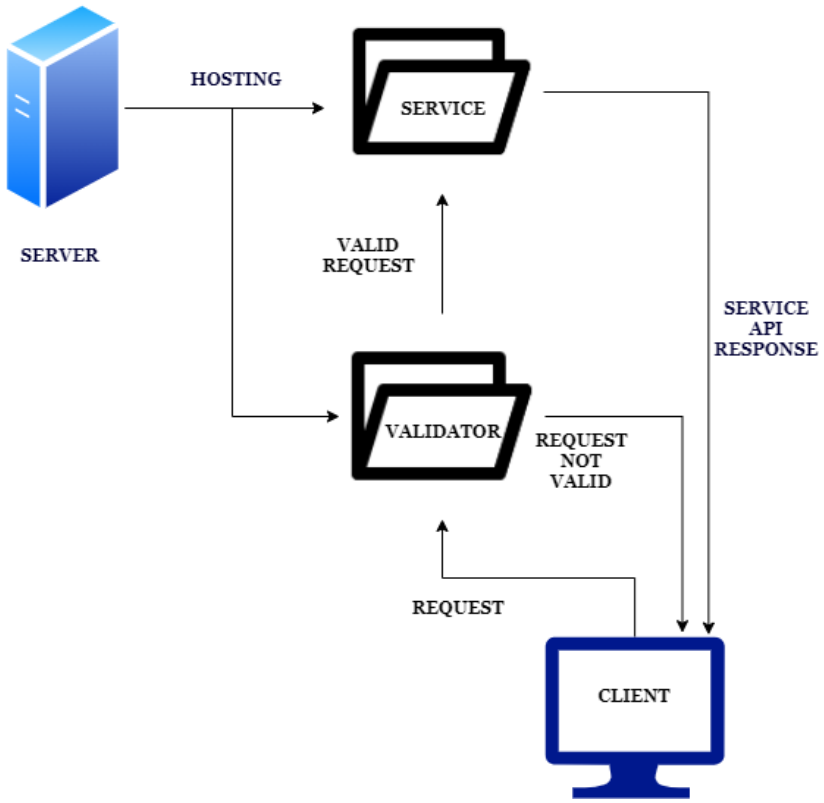

Figura 3.6. Schema architetturale della struttura dell'API con service e validator.

Seguendo la struttura sopra indicata, si sono costruite le quattro API di cui disporrà il server. La libreria che è stata usata per lo sviluppo dei file relativi ai validators è **tv4** [26] mentre i services sono stati strutturati come servizi puri dell'API.

3.4.5 initializeDevices

Si tratta dell'API addetta a rispondere alle richieste del costruttore dei devices IoT, quando questi ne immette di nuovi nel magazzino dell'azienda. L'API riceve dal costruttore un file costituito da una lista JSON al cui interno sono descritti i nuovi dispositivi disponibili. Viene richiamata con un metodo **POST** all'endpoint *http://localhost:3000/initializeDevices*.

initializeDevicesVal.js

```
1 const tv4 = require('tv4');
```

```
2  const errors = require('restify-errors');
3
4  class initializeDevicesVal {
5      async compile(req, res, next) {
6          try {
7              initializeDevicesVal.dataDevices = {
8                  'type': 'object',
9                  'properties':{
10                     'fileName': {
11                         'type' : 'string'
12                     },
13                     'newDevices' : {
14                         'type' : 'array'
15                     }
16                 },
17                 'required' : ['fileName', 'newDevices'],
18                 'additionalProperties' : false
19             }
20             if (tv4.validate(req.body, initializeDevicesVal.
                   dataDevices) == false) {
21                 console.log('Unable to process the request,
                       please retry ', (tv4.error).toString());
22                 console.log(JSON.stringify(req.body).red);
23                 return next(new errors.BadRequestError('File
                       format not valid.'));
24             } else {
25                 console.log('API initializeDevices,
                       validation passed');
26                 return next();
27             }
28         } catch (error) {
29             console.log('Error in the validator of
                   initializeDevices ' + error);
30         }
31     }
32 }
33
34 module.exports = initializeDevicesVal;
```

Listing 3.5. Codice Javascript per il file initializeDevicesVal.js

Alle righe **1-2** si importano il pacchetto **tv4** ,necessario alle validazioni, e il pacchetto **restify-errors** per la generazione dei messaggi d'errore. Viene in seguito dichiarato un oggetto *initializeDevicesVal* al cui interno è presente un metodo chiamato *compile* che si occuperà di validare eventualmente una richiesta, generare una risposta d'errore e saltare al successivo middleware dell'API.

Il validator procede a dichiarare un oggetto JSON del tipo di cui si aspetta la richiesta: in questo caso un oggetto con un *fileName* di tipo stringa e un *newDevices* di tipo array; entrambi al rigo **17** vengono classificati come **required** per far si che la richiesta sia validata. Quindi si procede a richiamare la libreria **tv4** ed il suo metodo *validate* per confrontare il body della richiesta col modello JSON preimpostato: se i campi richiesti non sono presenti nel body della richiesta, si genererà di risposta un errore **400, Bad Request** attraverso **restify-errors**. Se al contrario la richiesta viene validata, il validator procederà ad indirizzare la stessa richiesta al middleware per il service.

initializeDevicesSvc.js

```
1  const db = require('../dbManager.js');
2  const errors = require('restify-errors');
3
4  class initializeDevicesSvc {
5      async compile(req, res, next) {
6          try {
7              const {fileName, newDevices} = req.body;
8
9              const postRequest = {
10                 fileName,
11                 newDevices
12             };
13
14             //Create a new table if it is not existent
15             //And compile it with the provided info on
16             //the new devices
17             for(var i in postRequest.newDevices){
18                 await db.initializeTableDevices(postRequest.
                       newDevices[i]);
19             }
20
21             var response = {};
22             response['status'] ='created';
23             response['mexage '] = 'Database has been created
                   /updated with new devices';
24
25             res.send(201, response);
26             return next();
27         } catch (error) {
28             res.send(new errors.BadRequestError(error));
29             console.log('An error occured in the upload phase
                   ' + error);
30         }
```

```
31      }
32  }
33
34
35
36
37  module.exports = initializeDevicesSvc;
```

Listing 3.6. Codice Javascript per il file initializeDevicesSvc.js

Il service di *initializeDevice* dovrà interagire col database, per questa ragione viene importato il modulo *dbManager* oltre che la libreria **restify-errors** per la generazione del messaggio d'errore. Anche in questo caso viene costruito un oggetto al cui interno si troverà un metodo che processerà la richiesta; in particolare il metodo acquisirà il JSON della richiesta e ne salverà i campi all'interno di una costante. Poichè è necessario salvare tutti i nuovi dispositivi inviati dal costruttore, si è creato un for loop che iteri su tutto il vettore; per ogni posizione, il service chiama la funzione *initializeTableDevices* del *dbManager* ed inserisce i dati del singolo dispositivo nella tabella. Se l'inserimento è positivo, il service ritorna un messaggio **201, Created** e passa il controllo al server che rimarrà in ascolto della prossima richiesta; se ha incontrato un errore, invierà invece un messaggio **400, Bad Request**.

3.4.6 registryDevice

registryDevice è l'API addetta a raccogliere le richieste dei dispositivi IoT che sono intenzionati a registrarsi all'IoT hub.

registryDeviceVal.js

```
1  const tv4 = require('tv4');
2  const errors = require('restify-errors');
3
4  class registryDeviceVal {
5      async registry(req, res, next) {
6          try {
7              registryDeviceVal.status = {
8                  'type': 'object',
9                  'properties':{
10                     'deviceId' :{
11                         'type' : 'string'
12                     }
13                 },
14                 'required' : ['deviceId'],
15                 'additionalProperties' : false
```

```
16              }
17              if (tv4.validate(req.body, registryDeviceVal.
                    status) == false) {
18                  console.log('Unable to process the request,
                        please retry ', (tv4.error).toString());
19                  console.log(JSON.stringify(req.body).red);
20                  return next(new errors.UnauthorizedError('
                        Authentication failed.'));
21              } else {
22                  console.log('API registryDevice, validation
                        passed');
23                  return next();
24              }
25          } catch (error) {
26              console.log('Error in the registryDevice
                    validator ' + error);
27          }
28      }
29  }
30
31  module.exports = registryDeviceVal;
```

Listing 3.7. Codice Javascript per il file registryDeviceVal.js

La struttura è simile a quella utilizzata per *initializeDevicesVal* con l'importazione delle libreria e la dichiarazione di un oggetto al cui interno è presente un metodo per la validazione della richiesta. Questa volta il modello prevede che la richiesta abbia un body in cui sia presente l'ID del device in modo da porterlo salvare correttamente all'interno del registro dell'IoT hub. Di nuovo c'è un processo di validazione attraverso il metodo *validate* che in caso di responso negativo darà un codice di errore **401, Unauthorized**; se la validazione invece è stata positiva, il validator passa il controllo al service, col comando *next*, che si occuperà di servire la richiesta.

registryDeviceSvc.js

```
1  const errors = require('restify-errors');
2  const iothub = require('../hub.js');
3
4  class registryDeviceSvc {
5      async registry(req, res, next) {
6          try {
7              var response = {};
8              const {deviceId} = req.body;
9              console.log('Registry device service, device ID:
                   ' , deviceId);
```

```
10              //Check if the device is registered or not
11              //On the iot hub
12              iothub.checkID(deviceId)
13              .then(function(twinId){
14                  console.log('Twin id: ' + twinId);
15                  if(twinId == null){
16                      response['endpoint'] = 'fota.azure-
                            devices.net/devices/'.concat(deviceId)
                            ;
17                      console.log('Device is not present in the
                             iot hub, registration process started
                            , endpoint: ' , response.endpoint);
18                      res.send(201, JSON.stringify(response));
19                  }else{
20                      response['message'] = 'You\'re already
                            registered to iot hub!';
21                      console.log('Device is already in the iot
                             hub');
22                      res.send(202, JSON.stringify(response));
23                  }
24              })
25              .catch(function(err){
26                  console.log('An error occurred during the
                        query to device twins: ' + err);
27                  res.send(new errors.InternalServerError(err))
                        ;
28              });
29              return next();
30          } catch (error) {
31              res.send(new errors.InternalServerError(error));
32              console.log('An error occured in the registration
                    phase ' + error);
33          }
34      }
35  }
36
37  module.exports = registryDeviceSvc;
```

Listing 3.8. Codice Javascript per il file registryDeviceSvc.js

Al rigo **1** si importa il modulo *hub.js* per poter effettuare la comunicazione con l'IoT hub oltre che la libreria **restify-errors** per la gestione dell'errore; quindi si dichiara l'oggetto omonimo al file *registryDeviceSvc* con un metodo che riceverà la richiesta e si occuperà del suo servizio. In questo caso l'informazione utile sarà l'ID del device che sarà dunque salvato in una costante; l'informazione dell'ID viene passata come parametro al metodo *checkId* che

si occuperà di controllare se il dispositivo è già presente nel registro dell'hub attraverso una query ai dispositivi gemelli. A questo punto, se la ricerca non ha prodotto risultati, il device può procedere alla registrazione. Per stabilire una connessione e registrarsi, il device avrà però bisogno di un endpoint che viene strutturato nel seguente modo: *fota.azure-devices.net/devices/**deviceId***. Viene dunque inviato l'endpoint nel messaggio di risposta assieme al codice **201, Created**. Nel caso invece la ricerca produca come risultato che il device è già registrato, il service risponderà con un codice **202, Accepted** per indicare che la richiesta è stata accettata ma non terminata poiché il dispositivo è gia presente all'interno del registro dell'hub. In caso di errore, il service invierà un codice **500, Internal Server Error**.

3.4.7 getAvailableFirmware

Si è strutturata quest'API per servire le richieste da parte dei device di acquisire l'ultima versione disponibile del firmware.

getAvailableFirmwareVal.js

```
const tv4 = require('tv4');
const errors = require('restify-errors');

class getAvailableFirmwareVal {
    async list(req, res, next) {
        try {
            getAvailableFirmwareVal.valGetBlobList = {
                'type': 'object',
                'properties': {
                    'deviceID': {
                        'type': 'string'
                    },
                    'firmwareVersion':{
                        'type': 'string'
                    },
                    'classDevice':{
                        'type': 'string'
                    },
                },
                'required': ['deviceID', 'firmwareVersion', '
                   classDevice'],
                'additionalProperties': false
            };
```

```
23              if (tv4.validate(req.body,
                   getAvailableFirmwareVal.valGetBlobList) ==
                   false) {
24                  console.log('Unable to process the request,
                       please retry ', (tv4.error).toString().red
                       );
25                  console.log(JSON.stringify(req.body).red);
26                  return next(new errors.BadRequestError('id
                       not valid.'));
27              } else {
28                  console.log('API getAvailableFirmware,
                       validation passed');
29                  return next();
30              }
31          } catch (error) {
32              console.log('Error din the getAvailableFirmware
                   validator ' + error);
33          }
34      }
35  }
36
37
38
39  module.exports = getAvailableFirmwareVal;
```

Listing 3.9. Codice Javascript per il file getAvailableFirmwareVal.js

Si è utilizzata di nuovo la coppia di librerie **tv4** e **restify-errors** per la realizzazione del validator. La struttura del validator presenta un oggetto *getAvailableFirmware*, omonimo al file, dotato di un metodo per la validazione. Si struttura nuovamente un oggetto JSON: il modello richiede una proprietà in cui sia presente l'ID del dispositivo, una in cui sia presente la versione del firmware che il dispositivo ha installata al momento della richiesta ed una in cui viene indicata la classe del dispositivo. Nel caso in cui la richiesta venga validata, verrà passato il controllo al service attraverso il comando *next* al rigo **29**, altrimenti verrà ritornata una risposta di errore **400, Bad Request**.

getAvailableFirmwareSvc.js

```
1  const blob = require('../blob.js');
2  const db = require('../dbManager.js');
3  const errors = require('restify-errors');
4
5  class getAvailableFirmwareSvc {
6      constructor(containerClient){
```

```
7           getAvailableFirmwareSvc.containerClient =
                containerClient;
8       }
9
10      async list(req, res, next) {
11          try {
12              //Generate the list of files present on the cloud
13              //And process the request body
14              console.log(req.body);
15              var response = {};
16              var deviceID = JSON.parse(req.body).requestId;
17              var classDevice = JSON.parse(req.body).
                    classDevice;
18              var firmwareVersion = JSON.parse(req.body).
                    firmwareVersion;
19              response['message'] = deviceID + ' asked for a
                    fimware update';
20              response['status'] = 'ok';
21              const update = await db.checkVersions(classDevice
                    , firmwareVersion);
22              //Route the download only if is available a new
                    version of the firmware
23              if(update){
24                  var sasToken = await blob.generateSasToken(
                        getAvailableFirmwareSvc.containerClient,
                        getAvailableFirmwareSvc.containerClient.
                        containerName, update.idFile, "r");
25                  response['fwVersion'] = update.latestFirmware
                        ;
26                  response['url'] = update.url;
27                  response['sasToken'] = sasToken.uri;
28              }else{
29                  response['message'] = 'device already updated
                        ';
30              }
31              res.send(200, response);
32
33              return next();
34          } catch (error) {
35              res.send(new errors.BadRequestError(error));
36              console.log('An error occured during the access
                    to available firmware ' + error);
37          }
38      }
39  }
40
```

```
41 module.exports = getAvailableFirmwareSvc;
```

Listing 3.10. Codice Javascript per il file getAvailableFirmwareSvc.js

Per la creazione della classe di servizio per la *getAvailableFirmware* ci si è avvalsi dei moduli costruiti in precedenza *blob.js* e *dbManager.js* oltre che della libreria **restify-errors** per la gestione dell'errore. Questa classe è stata dotata anche di un costruttore che deve ricevere l'oggetto *containerClient* e settarlo come attributo della classe. Si è quindi creato il metodo addetto alla gestione della richiesta che va a prelevare dal body le informazioni sull'ID del device, sulla classe del device e sulla versione del firmware correntemente installata sul dispositivo. Il metodo richiama la funzione *checkVersions* del modulo *dbManager* per effettuare la ricerca sulle nuova versione del firmware. Se c'è un aggiornamento più recente disponibile, si provvede a generare un SAS token per permettere al dispositivo di accedere e scaricare la risorsa. E' stato quindi richiamato il metodo *generateSasToken* del modulo *blob.js*, di cui si era discusso in precendenza, fornendo l'oggetto *containerClient*, il nome del contenitore, l'id della risorsa a cui accedere e il permesso in lettura. Se la generazione è andata a buon fine, il service invierà come risposta un **200, OK** seguito dalla versione del firmware che è stata rilevata come più recente, il token per l'autenticazione e l'URI già incorporato del token per l'accesso diretto. Se la ricerca di nuovi aggiornamenti non ha prodotto risultati, il service segnalerà al device che è già aggiornato all'ultima versione. Se ci sono stati degli errori nel corso dell'operazione, il middleware risponderà con un **400, Bad Request**.

3.4.8 saveNewFirmware

saveNewFirmware è l'API incaricata di andare a salvare nel blob storage il nuovo file di aggiornamento per una determinata classe di devices e contemporaneamente di andare ad aggiornare la tabella *firmware*

saveNewFirmwareVal.js

```
1 const tv4 = require('tv4');
2 const errors = require('restify-errors');
3
4 class saveNewFirmwareVal {
5     async upload(req, res, next) {
6         try {
7             saveNewFirmwareVal.uploadBlobVal = {
8                 'type': 'object',
```

```
9                 'properties':{
10                    'fileName': {
11                        'type' : 'string'
12                    },
13                    'classDevice':{
14                        'type' : 'string'
15                    },
16                    'version':{
17                        'type' : 'string'
18                    },
19                    'releaseDate':{
20                        'type': 'string'
21                    },
22                },
23                'required' : ['fileName', 'version', '
                      releaseDate','classDevice'],
24                'additionalProperties' : false
25            }
26            console.log('***************');
27            if (tv4.validate(req.body, saveNewFirmwareVal.
                  uploadBlobVal) == false) {
28                console.log('Unable to process the request,
                      please retry ', (tv4.error).toString().red
                      );
29                console.log(JSON.stringify(req.body).red);
30                return next(new errors.BadRequestError('file
                      format not valid.'));
31            } else {
32                console.log('API saveNewFirmware, validation
                      passed');
33                return next();
34            }
35        } catch (error) {
36            console.log('Error in the saveNewFirmware
                  validator' + error);
37        }
38    }
39 }
40
41
42
43
44 module.exports = saveNewFirmwareVal;
```

Listing 3.11. Codice Javascript per il file saveNewFirmwareVal.js

Le librerie usate sono la stesse utilizzate per le altre tre API ovvero **tv4**

per la validazione del body e **restify-errors** per la gestione dell'errore. L'oggetto *saveNewFirmwareVal* possiede un metodo *upload* che si occupa della validazione delle richieste. Il body che ci si aspetta per questa API deve avere un parametro in cui è dichiarato il nome del file, un parametro dov'è indicata la classe di dispositivi a cui è rivolto, un parametro in cui è indicata la versione del firmware che si sta caricando e un parametro in cui è indicata la data di rilascio dell'aggiornamento. Se i quattro parametri esistono ed il loro formato è corretto, la loro validazione viene completata ed il comando *next* passerà il controllo al service dell'API; in caso contrario, verrà ritornato un codice di errore **400, Bad Request**.

saveNewFirmwareSvc.js

```
1  const fs = require('fs');
2  const { v1: uuidv1} = require('uuid');
3  const errors = require('restify-errors');
4  const hub = require('../hub.js');
5  const blob = require('../blob.js');
6  const db = require('../dbManager.js');
7
8  class saveNewFirmwareSvc {
9      constructor(containerClient){
10         saveNewFirmwareSvc.containerClient = containerClient;
11     }
12     async upload(req, res, next) {
13         try {
14             const {fileName, classDevice, version,
                   releaseDate} = req.body;
15             const fileNameID = fileName + uuidv1();
16             const postRequest = {
17                 fileNameID,
18                 version,
19                 releaseDate,
20                 classDevice
21             };
22
23             //Save the file locally
24             var oldpath;
25             var newpath;
26             oldpath = req.files.file.path;
27             newpath = './' + req.files.file.name;
28             fs.rename(oldpath, newpath, function(err){
29                 if(err){
30                     console.log(err);
```

```
31                      return new Error(err);
32                  }
33              })
34              console.log(newpath);
35              var response = {};
36              //Create a blob taking the file specified
37              //At path
38              //With name filenameID
39              const url = await blob.createBlob(
                    saveNewFirmwareSvc.containerClient,
                    postRequest.fileNameID, newpath);
40              //Create new entry in the table firmware
41              await db.newFirmware(postRequest, url);
42
43              response['status'] ='created';
44              response['blob created at url: ']  = url;
45              res.send(201, response);
46              //Notify to device of that class
47              //That there is new firmware
48              hub.checkClassToNotify(postRequest.classDevice);
49              return next();
50          } catch (error) {
51              res.send(new errors.BadRequestError(error));
52              console.log('An error occured in the upload of
                    the firmware phase ' + error);
53          }
54      }
55  }
56
57  module.exports = saveNewFirmwareSvc;
```

Listing 3.12. Codice Javascript per il file saveNewFirmwareSvc.js

Il service di questa API deve interfacciarsi con tre moduli: *hub.js* per comunicare con l'IoT hub che a sua volta andrà a scrivere nei dispositivi gemelli e segnalare la presenza di un aggiornamento, *blob.js* per andare a salvare il nuovo file d'aggiornamento nel blob storage e con il *dbManager.js* per andare ad aggiungere un nuovo record alla tabella ogni volta che viene caricato un nuovo file.

Osservazione 2 Per la generazione casuale ed univoca dei nomi per i file da salvare nello storage e nella tabella *firmware*, è stata utilizzata una libreria chiamata **uuidv1** che permette la generazione di un codice casuale, a 128 bit, in formato esadecimale, chiamato **UUID** (*Universally Unique Identifier*), basando la generazione di questo codice sul timestamp al momento in

cui viene effettuata la richiesta di generazione e sull'indirizzo ethernet del dispositivo in cui è stato generato. Questa associazione di valori rende *de facto* impossibile avere due nomi identici. [27]

Come nel caso del service middleware dell'API *getAvailableFirmware*, anche qui è stato inserito un costruttore che setti un parametro *containerClient* per l'oggetto omonimo passato in ingresso. Il metodo asincrono *upload* processa la richiesta ed in particolare salva in tre costanti le informazioni ricevute riguardanti il nome del file, la classe del dispositivo e la data di rilascio dell'aggiornamento. Per rendere il nome del file univoco all'interno dell'intero sistema, viene accodato al nome del file comunicato un ID univoco generato con la libreria **uuidv1**. Si procede dunque a spostare il file ricevuto in un path locale per preparare l'upload al blob storage: al rigo **39** si richiama il metodo *createBlob* a cui vengono passati l'attributo *containerClient*, il nome del file univoco e il path locale in cui è salvato il file. Il metodo ritornerà, come già discusso nella sezione dedicata, l'URL in cui è stata creata la nuova risorsa. Il nuovo URL viene quindi passato al metodo *newFirmware* che lo andrà a caricare nel database assieme alle altre informazioni sul nuovo codice.

Viene quindi stampato un messaggio a video in cui si segnala la creazione della risorsa all'URL indicato e si invia di risposta al client un codice **201, Created**.

3.4.9 serverFota.js

La gestione logica e strutturale del server è stata elaborata basandosi sulle API create create in precedenza: il server quando sarà in esecuzione, gestirà le richieste che verranno effettuate sulla porta di ascolto **3000** e, nel caso in cui l'accesso all'API sia effettuato col metodo corretto, il server indirizzerà la richiesta al sistema di service-validator illustrato in precedenza.

La libreria con cui si è strutturato il server è **restify**: essa permette di inserire anche tutto il middleware necessario a rendere l'interfaccia RESTFul. [28]

Di seguito verrà illustrato il codice Node JS scritto per il server:

```
1 const restify = require('restify');
2 const blob = require('./blob.js');
3 const config = require('./config.js');
4 const chalk = require('chalk');
5 //Services and validators import
6 const registryDeviceVal = require('./Validators/
    registryDeviceVal.js');
```

```
7  const registryDeviceSvc = require('./Service/
       registryDeviceSvc.js');
8  const initializeDevicesVal = require('./Validators/
       initializeDevicesVal.js');
9  const initializeDevicesSvc = require('./Service/
       initializeDevicesSvc.js');
10 const getAvailableFirmwareSvc = require('./Service/
       getAvailableFirmwareSvc.js');
11 const getAvailableFirmwareVal = require('./Validators/
       getAvailableFirmwareVal.js');
12 const saveNewFirmwareSvc = require('./Service/
       saveNewFirmwareSvc.js');
13 const saveNewFirmwareVal = require('./Validators/
       saveNewFirmwareVal.js');
14
15
16
17 //Execute server
18 server();
19
20 async function server(){
21 try{
22   //Initialize the container in the blob storage
23   //If it not already exists
24   var containerClient = await blob.createContainer(config.
       CONTAINER_NAME);
25   //Create the service and validators for
26   //The APIs
27   this.server = restify.createServer({name : 'fota'});
28   this.registryDeviceVal = new registryDeviceVal();
29   this.registryDeviceSvc = new registryDeviceSvc();
30   this.initializeDevicesVal = new initializeDevicesVal();
31   this.initializeDevicesSvc = new initializeDevicesSvc();
32   this.saveNewFirmwareSvc = new saveNewFirmwareSvc(
       containerClient);
33   this.saveNewFirmwareVal = new saveNewFirmwareVal();
34   this.getAvailableFirmwareSvc = new getAvailableFirmwareSvc(
       containerClient);
35   this.getAvailableFirmwareVal = new getAvailableFirmwareVal
       ();
36
37   //Middlewares
38   this.server.use(restify.plugins.acceptParser(['application/
       json']));
39   this.server.use(restify.plugins.queryParser());
40   this.server.use(restify.plugins.requestLogger());
```

```
41    this.server.use(restify.plugins.bodyParser());
42
43    // Prevent response caching
44    this.server.use(function (req, res, next) {
45    res.header('Cache-Control', 'no-store');
46    res.header('Pragma', 'no-cache');
47    return next();
48  });
49
50  //Listen to port 3000
51  this.server.listen(config.PORT, async function(){
52    try{
53      console.log('Server is listening at port: ' , config.PORT
          );
54    }
55    catch(err){
56      console.log(chalk.red('Unexpected error: ' + err));
57    }
58  });
59
60    //Device needs to register to the iot hub API
61    this.server.post('/registryDevice',
62    await this.registryDeviceVal.registry,
63    await this.registryDeviceSvc.registry);
64
65    //New devices available API
66    this.server.post('/initializeDevices',
67    await this.initializeDevicesVal.compile,
68    await this.initializeDevicesSvc.compile);
69
70    //Save Firmware API
71    this.server.post('/saveNewFirmware',
72    await this.saveNewFirmwareVal.upload,
73    await this.saveNewFirmwareSvc.upload);
74
75    //Get the most recent firmware available API
76    this.server.get('/getAvailableFirmware',
77    await this.getAvailableFirmwareVal.list,
78    await this.getAvailableFirmwareSvc.list);
79
80    console.log('Server is running');
81      }
82      catch(err){
83       console.log(chalk.greenBright('Unexpected error in the
            server: ' + err.toString()));
84      }
```

```
85 }
```

Listing 3.13. Codice Javascript per il file serverFota.js

Oltre alla libreria restify, tra gli import si possono individuare il file di configurazione che è stato utilizzato anche per gli altri moduli, da cui si attingerà per prelevare il nome utilizzato per il contenitore sull'account di storage e la porta utilizzata per hostare la connessione, il file *blob.js* utilizzato per la creazione dell'oggetto contenitore, necessario per il service delle API *getAvailableFirmware* e *saveNewFirmware*, i quattro services ed i quattro validators delle API ed una libreria grafica **chalk** utilizzata per enfatizzare la scrittura dei messaggi a video all'interno del server.

Al rigo **17** avviene il lancio del server, ma si vuole prima mostrare la struttura interna definita alle righe successive. La funzione server viene dichiarata al rigo **19** in modo asincrono, così da poter eseguire altre funzioni asincrone al suo interno; la prima che si incontra è la *createContainer* a cui si passa il nome del contenitore e si avrà come ritorno l'oggetto contenitore associato.

Si passa dunque alla definizione delle proprietà del server (righe **26-34**). Attraverso il metodo di **restify** *createServer*, si restituisce un oggetto di tipo server con il nome passato come parametro al metodo (**fota** in questo caso); inoltre si è impostato un attributo per ognuno dei services e validators andando a richiamare gli oggetti omonimi e instanziandoli attraverso la keyword **new**. Per impostare gli altri middlewares necessari, si è utilizzato il metodo *use* (righe **37-40**) dell'oggetto server che verranno attivati soltanto quando le richieste saranno indirizzate ad una route disponibile:

- **restify.plugins.acceptParser('application/json')** è il middleware che imposta l'**accept header** che indica al client che tipo di contenuto il server si aspetta dalla richiesta (in questo caso un tipo JSON).
- **this.server.use (restify.plugins.queryParser())** permette di tradurre una query indicata nella richiesta.
- **restify.plugins.requestLogger()** registra l'ID di ogni client che effetua una richiesta al server.
- **restify.plugins.bodyParser()** si occupa di leggere e tradurre automaticamente il body della richiesta ricevuta.

Alle righe **44-47** si è scelto di utilizzare un sistema *non-cachable* per prevenire il salvataggio della risposta server nella cache del client; così facendo ogni nuova richiesta del client sarà scorrellata da quella precedente. Questo

fatto permette di soddisfare i requisiti RESTful e di non occupare cache nella memoria dei dispositivi, considerando la non critica frequenza degli aggiornamenti e quindi delle loro richieste. Vengono utilizzati due header per settare queste impostazioni: **Cache-Control** e **Pragma**. Entrambe le impostazioni prevengono il salvataggio nella cache del client e la scelta di utilizzarne due è dovuta alla retrocompatibilità con protocolli HTTP più vecchi che supporta l'header **Pragma**.

Si procede al rigo **51** a richiamare il metodo *listen* del server sulla porta 3000 a cui il server rimarrà in ascolto fino all'arrivo di una richiesta da servire.

Si sono quindi create le route alle quali il server può rispondere andando ad impostare i metodi accettati per ogni route. Al rigo **61** si è costruita una route */registryDevice* a cui un client può accedere con un metodo POST: se il routing avviene con successo, la richiesta dovrà passare attraverso il validator dell'API, richiamata come attributo della route dell'oggetto server, attraverso il metodo *registry*. Il metodo omonimo del service lavorerà per servire la richiesta in caso la validazione sia andata a buon fine.

La route */initializeDevices* viene costruita al rigo **66** attraverso un metodo POST dell'oggetto server: se il routing ha successo, il metodo *compile* del validator si occupa della validazione della richiesta. Se la validazione è andata a buon fine, il metodo omonimo del service servirà la richiesta.

Al rigo **71** si crea la route */saveNewFirmware*, la quale accetta un metodo POST: in caso di successo del routing, il metodo *upload* del validator si occuperà della validazione della richiesta. Se la richiesta è valida, verrà servita dal metodo omonimo del service.

Al rigo **76** viene costruita la route per l'API */getAvailableFirmware* attraverso un metodo GET: anche in questo caso, se l'instradamento avviene in maniera corretta, la richiesta sarà analizzata dal metodo *list* del validator. Se anche la validazione ha avuto successo, il metodo omonimo si occuperà di servire la richiesta.

Infine alle righe **80-83** si stampa a video un avviso di esecuzione del server e, in caso di criticità impreviste durante l'esecuzione, si gestiscono gli errori con un *catch* e si stampa a schermo il messaggio d'errore.

3.5 Il codice - Lato device

Si è supposto che i client IoT abbiano un'architettura almeno a 32 bit in grado di eseguire un'applicazione Node JS e che l'applicazione sia salvata all'interno della memoria non volatile del device prima di essere messo in

fuzione. Con queste premesse si è sviluppato un unico file Javascript client in grado di interfacciarsi efficacemente con l'applicazione back-end appena trattata, supportato solo da un file utility di configurazione ed un JSON reader; si vuole cominciare illustrando la struttura del file di configurazione elaborato per un device generico:

3.5.1 config.js

```
1  const factorySettings = {
2      deviceId: 'PG3084NFHMS',
3      classDevice: 'peugeot308_1.2_puretech_2018_allure',
4      firmwareVersion: '1.2',
5  }
6
7  module.exports = {
8      IOT_HUB_CONNECTION_STRING: process.env.
          IOT_HUB_CONNECTION_STRING || '******',
9      DEVICE: {
10         deviceId: factorySettings.deviceId,
11         classDevice: factorySettings.classDevice,
12         firmwareVersion: factorySettings.firmwareVersion,
13     },
14     OPTIONS_DEVICE : {
15         method : 'POST',
16         uri: 'http://localhost:3000/registryDevice',
17         json: true,
18         body : {
19                     'deviceId' : factorySettings.deviceId
20                 }
21       },
22      OPTIONS_FIRMWARE: {
23         method : 'GET',
24         uri: 'http://localhost:3000/getAvailableFirmware',
25         json: true,
26         body: {
27             'requestId': factorySettings.deviceId
28           , 'classDevice': factorySettings.classDevice
29           , 'firmwareVersion': factorySettings.
                firmwareVersion
30         }
31       }
32 }
```

Listing 3.14. Codice Javascript per il file config.js

Ogni device avrà un file di configurazione di questo tipo con le sue caratteristiche di fabbrica definite dal costruttore e salvate in una costante JSON. Una volta che il dispositivo passa dal magazzino dell'azienda per cui si sta elaborando questo progetto di tesi, si procederà a modificare il file inserendo la stringa di connessione dell'IoT hub (riga **8**) e le opzioni di accesso alle API, compresi gli attributi richiesti sul body.

3.5.2 device.js

Si procede ora ad illustrare l'applicativo del device:

```
1  const fs = require('fs');
2  const request = require('request');
3  const iothub = require('azure-iothub');
4  const Client = require('azure-iot-device').Client;
5  const Protocol = require('azure-iot-device-mqtt').Mqtt;
6  const crypto = require('crypto');
7  const config = require('./config.js');
8  const jsonManager = require('./jsonManager.js');
9  const { Console } = require('console');
10
11 //Setup the connection string
12 //For the IoT hub
13 const IOT_HUB_CONNECTION_STRING = config.
       IOT_HUB_CONNECTION_STRING;
14 const registry = iothub.Registry.fromConnectionString(
       IOT_HUB_CONNECTION_STRING);
15
16   // Ask for a new registration to iot hub
17   request(config.OPTIONS_DEVICE, async function(error,
         response, body){
18     try{
19       console.log(response.statusCode);
20       switch (response.statusCode) {
21         // If the response is a 201, the device is not
              registered yet
22         case 201:
23           registryNewDevice(error, response, JSON.parse(body)
               )
24           .then(function(client){
25             console.log('Device registered' , JSON.parse(body
                 ));
26             createTwin(client);
27           })
28           .catch(function(error){
29             console.log('Error in the request' + error);
```

```
30            return new Error(error);
31          })
32          break;
33
34        // If the response is a 202, the device is already
             registered
35        case 202:
36          console.log('Device already registered' , body.
               message);
37
38          // Read the local file where sas token is stored
39          jsonManager.readJson()
40          .then(function(keys){
41            var client = Client.fromSharedAccessSignature(
                 keys.sasToken, Protocol);
42            console.log('oggetto: ' , client._transport._fsm.
                 state);
43            if(client._transport._fsm.state.trim() == '
                 disconnected'){
44              console.log(keys);
45              console.log('Sas token has expired, generating
                   a new one ');
46              let sasToken = generateSasToken(String(keys.
                   endpoint), String(keys.SymmetricKey), '',
                   120);
47              var client = Client.fromSharedAccessSignature(
                   sasToken, Protocol);
48              keys.sasToken = sasToken;
49              console.log('Updating the new token');
50              jsonManager.writeJson(JSON.stringify(keys));
51              checkTwin(client);
52            }else{
53              console.log('Acquiring the twin');
54              checkTwin(client);
55            }
56          })
57          .catch(function(err){
58            console.log('Error during the reconnection with
                 the iot hub' , err);
59          })
60          break;
61
62        // Some problems occurred during the request
63        default:
64          console.log('Status code not recognized for the
               registration request ' , body);
```

```
65            break;
66          }
67        }catch(error){
68          console.log('An error occurred during the request for
               registration ' + error);
69        }
70      })
71
72  // Write the deviceID into the register of the iot hub
73  function registryNewDevice(error, response, body){
74    return new Promise(function(resolve, reject){
75      if(body.endpoint){
76        registry.create(config.DEVICE, function(err, deviceInfo
             , res) {
77          if (error){
78           console.log(' Error during the connection with the
                iot hub: ' + err);
79           return new Error(err);
80          }
81          else if (res && deviceInfo){
82            console.log('Status of the request: ' + res.
                 statusCode + ' ' + res.statusMessage);
83            console.log('Device info: ' + JSON.stringify(
                 deviceInfo));
84
85            // Get the symmetric key generated during the
                 registration
86            // In order to generate a sas token with a limited
                 time validity
87            const sasToken = generateSasToken(body.endpoint,
                 deviceInfo.authentication.SymmetricKey.
                 primaryKey, '', 120);
88            // Write the token into a local file
89            var keys = {
90              endpoint: body.endpoint,
91              primaryKey: deviceInfo.authentication.
                   SymmetricKey.primaryKey,
92              sasToken: sasToken
93            }
94            jsonManager.writeJson(JSON.stringify(keys));
95
96            const client = Client.fromSharedAccessSignature(
                 sasToken, Protocol);
97
98            //Check if an error occured during
99            //The authentication with SAS token
```

```
100           if (err) {
101             console.log('Could not open IotHub client');
102             reject(err);
103           } else if(client){
104             console.log('Client connected');
105             resolve(client);
106           }
107         }
108       })
109       }else{
110         console.log('An error occured while trying to
              registrate a new device' , error);
111         reject(error);
112       }
113   })
114   }
115
116 // Open the twin of the device in the iot hub
117 function createTwin(client){
118         client.open(function(err){
119
120           //Create the iothub client
121           if(!err){
122             console.log('Client opened');
123
124             // Acquire device Twin
125             client.getTwin(function(err, twin) {
126               if (err) {
127                 console.error('Could not get twin' + err);
128                 return new Error(err);
129               } else {
130                 // Initialize the twin of the device in the
                    iot hub
131                 initializeStatus(twin);
132                 console.log('Twin acquired');
133                   twin.on('properties.desired.firmware',
                      function(firmwareUpdate) {
134                     if (firmwareUpdate.status == 'available')
                        {
135                       console.log('Updating firmware version:
                          ');
136
137                       // Call the getAvailableFirmware API
138                       request(config.OPTIONS_FIRMWARE,
                          function (error, response, body) {
139                         if(response.statusCode == 200){
```

```
140
141                         //If a successfull request has
                               occurred
142                         //Download the update file
143                         console.log(body)
144                         var newFirmwareStream = fs.
                               createWriteStream('./lastUpdate.
                               zip')
145                         request(body.sasToken).pipe(
                               newFirmwareStream);
146                         newFirmwareStream.on('close',
                               function(){
147                           if(error){
148                             return new Error(error);
149                           }else{
150                             console.log('New version ' +
                                  body.fwVersion);
151                             twin.properties.reported.update
                                  ({firmware: {fwVersion :
                                  body.fwVersion}}, function(
                                  err) {
152                               if(err){
153                                 console.log('Error during
                                      the reporting of the
                                      properties ' + err);
154                                 return new Error(err);
155                               }
156                             });
157                           }
158                         });
159                       } else {
160                         console.log('Error during the
                               acquisition of the new firmware
                               on device side: '+ error)
161                         return new Error(error);
162                       }
163                     })
164                   }else{
165                     console.log('Error during the check of
                           the desired properties' + error);
166                     return new Error(error);
167                   }
168                 });
169               }
170           })
171         }
```

```
172       })
173 }
174
175 // Check the desired properties of the twin
176 function checkTwin(client){
177   client.getTwin(function(err, twin) {
178     if (err) {
179       console.error('could not get twin ' + err);
180       return new Error(err);
181     } else {
182         twin.on('properties.desired.firmware', function(
              firmwareUpdate) {
183           if (firmwareUpdate.status == 'available'){
184             console.log('Updating firmware version: ');
185
186             // Call the getAvailableFirmware API
187             request(config.OPTIONS_FIRMWARE, function (error,
                  response, body) {
188               if(response.statusCode == 200){
189                 //If a successfull request has occurred
190                 //Download the update file
191                 var newFirmwareStream = fs.createWriteStream(
                      './lastUpdate.zip')
192                 request(body.sasToken).pipe(newFirmwareStream
                      );
193                 newFirmwareStream.on('close', function(){
194                   if(error){
195                     return new Error(error);
196                   }else{
197                     console.log('New version ' + body.
                          fwVersion);
198                     twin.properties.reported.update({firmware
                          : {fwVersion : body.fwVersion}},
                          function(err) {
199                       if(err){
200                         console.log('Error during the
                              reporting of the properties ' +
                              err);
201                         return new Error(err);
202                       }
203                     });
204                   }
205                 });
206               } else {
207                 console.log('Error during the acquisition of
                      the new firmware on device side: '+ error)
```

```
208                 console.log(body);
209                 return new Error(error);
210               }
211             })
212           }else{
213             console.log('Error during the check of the
                  desired properties' + error);
214             return new Error(error);
215           }
216       });
217     }
218 })
219 }
220
221 var generateSasToken = function(resourceUri, signingKey,
      policyName, expiresInMins) {
222   resourceUri = encodeURIComponent(resourceUri);
223
224   // Set expiration in seconds
225   var expires = (Date.now() / 1000) + expiresInMins * 60;
226   expires = Math.ceil(expires);
227   var toSign = resourceUri + '\n' + expires;
228
229   // Use crypto
230   var hmac = crypto.createHmac('sha256', Buffer.from(
        signingKey, 'base64'));
231   hmac.update(toSign);
232   var base64UriEncoded = encodeURIComponent(hmac.digest('
        base64'));
233
234   // Construct authorization string
235   var token = "SharedAccessSignature sr=" + resourceUri + "&
        sig="
236   + base64UriEncoded + "&se=" + expires;
237   if (policyName) token += "&skn="+policyName;
238   return token;
239 };
240
241 // Send firmware update status to the hub
242 function initializeStatus(twin) {
243   var patch = {
244     classDevice: config.DEVICE.classDevice,
245     firmware: {
246       fwVersion: config.DEVICE.firmwareVersion,
247     }
248   };
```

```
249   twin.properties.reported.update(patch, function(err) {
250     if(err){
251       console.log('Error during the reporting of the
              properties ' + err);
252       return new Error(err);
253     }
254   });
255 }
```

Listing 3.15. Codice Javascript per il file device.js

Le righe **1-9** rappresentano gli import del progetto del device: **fs** è la libreria nativa di Javascript che permette l'interfaccia di lettura/scrittura con i file ed è utilizzata per creare uno streaming di **chunks** [4] durante la scrittura del file d'aggiornamento. La libreria **azure-iothub** è la stessa utilizzata nel blocco *hub.js* ma qui verrà utilizzata lato front-end. **azure-iot-device** è la SDK che permette di implementare la messaggistica e la comunicazione tra il device e l'hub: in particolare la classe **Client** specifica che si vuole mettere in comunicazione un dispositivo che avrà bisogno di un'autenticazione. [29] Come specificato nella sezione relativa ai protocolli per l'IoT hub, la scelta è ricaduta sul protocollo MQTT; si utilizzerà quindi la libreria **azure-iot-device-mqtt** con il suo modulo **Mqtt** che fornirà l'implemetazione del protocollo MQTT per il device. Si è utilizzata una libreria **crypto** per generare un SAS token in grado di fornire un accesso sicuro e veloce al dispositivo ogni volta che deve effettuare una connessione. [30] Oltre alle librerie, si importa anche il file di configurazione mostrato in precedenza e *jsonManager.js* che è stato scritto per facilitare il salvataggio e la lettura del file in cui viene archiviato il token necessario per la connessione.

Osservazione 3 Si è scelto di archiviare il SAS token dei dispositivi in un file locale differente da quello di configurazione per fare in modo che, in applicazioni reali, sia il dispositivo a scegliere una locazione della memoria sicura in cui salvare il token.

La prima azione che il device deve effettuare quando allaccia una comunicazione con l'IoT hub è quella di iscriversi al registro delle identità dell'hub: per realizzare questa operazione è necessario utilizzare il metodo *Registry* e passare la stringa di connessione dell'IoT hub. Se la connessione esiste e la stringa di connessione è valida, diventa possibile richiamare il back-end

[4] Blocchi di memoria ad allocazione dinamica.

attraverso le opzioni del file di configurazione dell'API *registryDevice*. All'interno della richiesta, viene analizzato lo status code della risposta (rigo **18**) e tramite un costrutto *switch* si analizzano le possibili opzioni:

- **201:** il dispositivo non esisteva nel registro delle identità e l'IoT hub può iscriverlo per la prima volta. Si lancia una funzione *registryNewDevice*, descritta al rigo **73**, la quale processa la risposta dell'API *registryDevice*; se la risposta del server contiene al suo interno un endpoint di connessione, allora il device può registrarsi all'iot hub. La registrazione avviene tramite il metodo *create* che indica all'hub di iscrivere il nuovo dispositivo nel registro delle identità. Una volta che la registrazione sarà completata, se l'hub invia una risposta ed è presente un parametro chiamato *deviceInfo*, allora la funzione va avanti. *deviceInfo* è un parametro generato dalla libreria **iot-hub** per il device al momento della registrazione e comunica il tag con cui è stato registrato il device, lo stato attuale della connessione tra dispositivo e cloud, il numero dei messaggi scambiati e, in particolare, due chiavi simmetriche generate al momento della registrazione. Firmando un endpoint di connessione attraverso una di queste chiavi, il device potrà autenticarsi all'hub e comunicare. Per questo motivo al rigo **87** si genera un SAS token basandosi su questi due parametri, fornendo una policy di default per la lettura e scrittura dei messaggi e un tempo di validità pari a 120 minuti; quest'ultimo fatto implica la necessità di salvare il token in un file locale e leggerlo se è ancora valido. Al rigo **89** si procede alla creazione di un file JSON locale impostando l'endpoint, la chiave primaria simmetrica ed il sas token appena generato: il tutto viene scritto attraverso il metodo *writeJson* del file *jsonManager* (il metodo *stringify* permette di trasformare il JSON in una stringa salvarlo come tale). A questo punto la funzione aprirà la connessione attraverso il metodo *fromSharedAccessSignature* fornendo come parametri il token appena generato ed il protocollo MQTT. Se la connessione è stata autenticata, la promise diventerà fullfilled e ritornerà un oggetto *client* da utilizzare per le operazioni di aggiornamento, altrimenti ritornerà gli errori.

  Tornando al rigo **23**, se la promise è stata risolta positivamente, si lancia una nuova funzione *createTwin* a cui viene passato il parametro *client*.

  La funzione *createTwin* viene definita al rigo **117** e apre la connessione client tramite il comando *open* e, se non c'è stato errore, acquisisce il modello del dispositivo gemello del device tramite il comando *getTwin*: se il

dispositivo gemello non è stato acquisito, si ritorna un errore, altrimenti si procede ad inizializzare il twin. La funzione *initializeStatus* viene definita al rigo **242**: essa riceve in ingresso l'oggetto *twin* e costruisce una patch d'inizializzazione partendo dalla classe del dispositivo e dalla versione del firmware installata dal costruttore. La funzione richiama il percorso delle **reported properties** dall'oggetto *twin* ed utilizza il metodo *update* per caricare la patch. Nel caso in cui il caricamento sia fallito, viene riportato un errore.

Dopo aver inizializzato le proprietà segnalate, al rigo **133** il device rileva se avvengono delle modifiche da parte del back-end sulle proprietà desiderate: se ciò avviene nel campo **firmware**, viene eseguito un controllo sullo **status**. Il controllo sullo **status** rileva se il messaggio corrisponde alla stringa **available**: se esiste corrispondenza, al rigo **138**, il device fa una richiesta all'API *getAvailableFirmware* attraverso i parametri specificati precedentemente nel file di configurazione. Se il codice di risposta dell'API è un **200**, il device potrà avere accesso alla risorsa necessaria per l'aggiornamento; viene quindi aperto uno stream per la scrittura del nuovo file attraverso il metodo *createWriteStream* della libreria **fs** e viene effettuata una richiesta al blob storage attraverso l'URI firmato col SAS token e attraverso un metodo *pipe* viene inserita all'interno dello stream di scrittura. Al rigo **146** si attende il salvataggio del file, quindi si controlla se è occorso un errore durante la fase di scrittura e, in caso di successo, viene stampato a schermo un messaggio con l'indicazione sulla nuova versione. Il device riporta a questo punto la notifica del ricevuto aggiornamento all'hub andando a modificare le **reported properties** tramite il metodo *update* indicando la versione appena scaricata. La funzione si chiude con la gestione degli errori dovuta al download del nuovo file e alla modifica delle proprietà del dispositivo gemello.

- **202:** il dispositivo era già presente nel registro delle identità e può quindi procedere direttamente con la comunicazione; tramite il metodo *readJson* al rigo **39**, si legge il file di testo in cui era stato archiviato il SAS token e, in caso di stato fullfilled della promise, si ricostruisce l'oggetto client tramite il metodo *fromSharedAccessSignature* passando nuovamente il token ed il protocollo come parametri. Nel caso in cui il token sia scaduto, bisogna generarne uno nuovo; si procede leggendo la chiave primaria dal file di testo e la si passa al metodo *generateSasToken* assieme all'endpoint, alle policy di default e al tempo di vita del token. Una volta rigenerato, si procederà a controllare il parametro *state* dell'oggetto

client per verificare se la connessione ha avuto successo: se il parametro riporta lo stato **disconnected**, il token è scaduto e bisogna generarne uno nuovo col metodo *generateSasToken* ed i parametri specificati in precedenza. Sia nel caso in cui il token sia scaduto, sia nel caso in cui sia ancora valido, si richiamerà una funzione *checkTwin* definita al rigo **176**.

La funzione prende l'oggetto *client* da cui chiama il metodo *getTwin* per individuare, anche in questo caso, la presenza di modifiche al campo **firmware** delle **desired properties**: nel caso in cui sia rilevata una modifica, si controlla lo status del campo **firmwareUpdate** e, se risulta come **available**, e si effettua una richiesta GET alla *getAvailableFirmware*. Se la risposta dell'API è un **200**, si procede anche in questo caso al download della risorsa dallo storage con la stessa metodologia illustrata nell'altro ramo. Quando il download è stato completato, il device scrive nelle **reported properties** la nuova versione. In coda alla funzione avviene, come nel resto del progetto, la gestione degli errori con i relativi messaggi.

- **Default:** il codice inviato dal server è sconosciuto e non interpretabile dal device e viene stampato un messaggio con il body della risposta.

La funzione definita al rigo **221** permette la realizzazione di una firma di accesso condiviso: l'URI passato come stringa di testo viene trasformato con codifica URI e il tempo in minuti, passato come parametro, viene convertito in secondi prendendo in considerazione come base dei tempi la data attuale ottenuta tramite *Date.now*. La data di validità viene quindi approssimata tramite il metodo *ceil* della libreria nativa **Math** di Javascript; si costruisce quindi la stringa da firmare con il token, unendo l'URI e il tempo di validità. A questo punto si richiama la libreria **crypto** ed il suo metodo *createHmac* che consente la creazione di un oggetto hmac [5], criptato da un algoritmo di hash che crea un'impronta non invertibile, a 256 bit, utilizzando la chiave primaria fornita come parametro d'ingresso. Al rigo **230**, tramite il metodo update dell'oggetto hmac, si unisce all'impronta già creata le stringhe dell'URI e del tempo di validità. A questo punto si può generare l'URI firmato tramite la struttura illustrata alle righe **235-236** e ritornare dalla funzione il valore ottenuto.

[5] eyed-Hash Message Authentication Code

3.5.3 jsonManager.js

Si tratta di un modulo che realizza la lettura e scrittura di un file JSON, per utilizzo in locale, come archiviazione delle chiavi primaria e secondaria e dell'ultimo SAS token generato.

```
1  const fs = require('fs');
2
3  function readJson(){
4      return new Promise(function(resolve, reject){
5          fs.readFile('../Device Side/keys.json', (err,
               dataToRead) => {
6              if (err){
7                  console.log('Error during reading of the keys
                       : ' + err);
8                  reject(err);
9              }else{
10                 console.log('Keys have been read');
11                 var keys = JSON.parse(dataToRead);
12                 resolve(keys);
13             }
14         });
15     })
16 }
17
18 // Parameter must be passed in string format
19 function writeJson(dataToWrite){
20     fs.writeFile('../Device Side/keys.json', dataToWrite,
           function(err){
21         if(err){
22             console.log('Error during the creation of the
                   json file with the keys: ' + err);
23             return new Error(err);
24         }else{
25             console.log('Keys written correctly!');
26         }
27     })
28 }
29
30 exports.readJson = readJson;
31 exports.writeJson = writeJson;
```

Listing 3.16. Codice Javascript per il file jsonManager.js

L'unico import è realizzato per **fs** per la lettura e scrittura dei file.

Si è definito come promise, un metodo *readJson* che prende il file locale denominato *keys.json*, lo legge e, se non occorrono errori, applica il metodo

parse per convertirlo in un formato facilmente accessibile con la notazione a punti di Javascript.

Il metodo *writeJson* riceve l'oggetto con le chiavi come parametro d'ingresso e lo scrive nel file locale *keys.json*: in caso di successo, viene stampato un messaggio a video.

3.6 Simulazione del flusso FOTA

Si vuole adesso simulare il flusso completo dell'architettura elaborata procedendo ad affrontare una situazione reale di utilizzo. Il produttore ha appena venduto i suoi dispositivi all'azienda incaricata di produrre questo progetto di tesi e vuole comunicare quali dispositivi ha appena immesso in magazzino e quali sono le loro caratteristiche. Per prima cosa, da un calcolatore locale situato nella sede dell'azienda, si lancia il server attraverso un'interfaccia **Powershell**.

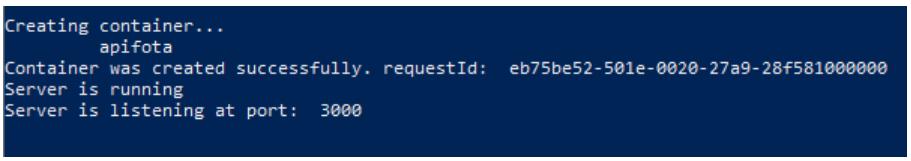

Figura 3.7. Lancio del server da interfaccia Powershell.

Come si nota dalla figura **3.7**, il parametro **apifota** è stato utilizzato per creare il contenitore omonimo sull'account di storage di Azure. Il server comunica anche che sta rimanendo in ascolto alla porta 3000.

Attraverso Postman si simula la seguente richiesta all'API *initializeDevices* attraverso un metodo POST.

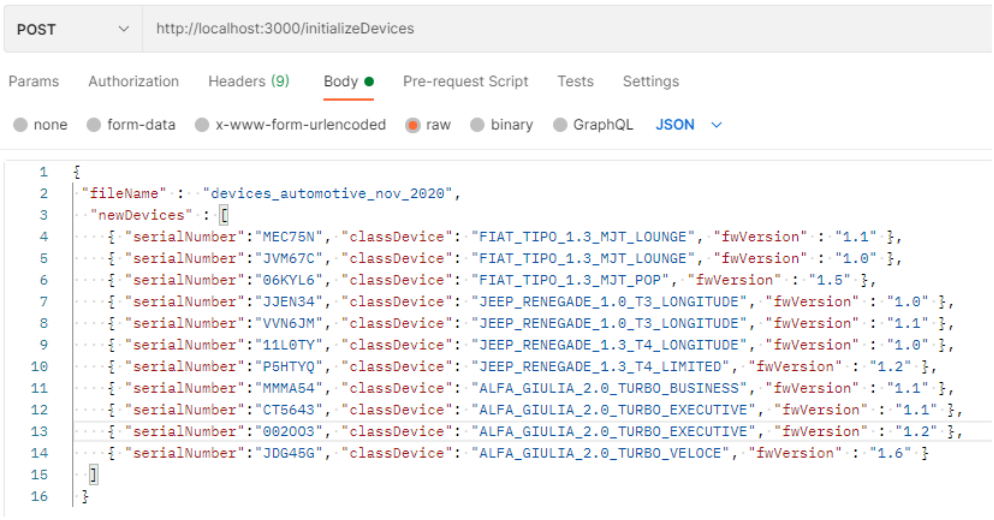

Figura 3.8. Nuovi dispositivi forniti dal produttore, in formato JSON, da finestra Postman.

Dalla figura **3.8** si può visualizzare il JSON all'interno del body, contentente i nuovi dispositivi che il produttore sta segnalando. Si procede dunque alla richiesta all'API *initializeDevices*, ottenendo la seguente risposta.

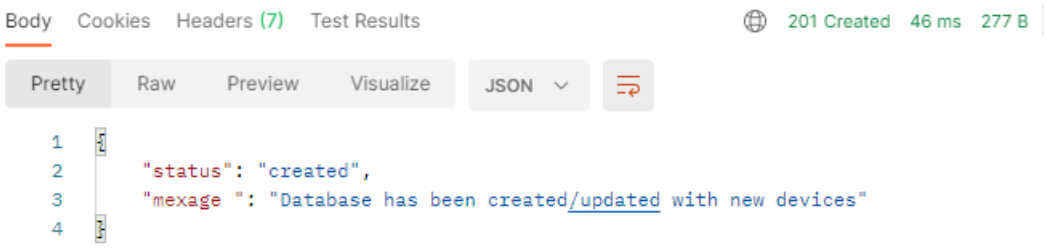

Figura 3.9. Risposta con codice 201 all'inizializzazione dei dispositivi.

Nel body della risposta è possibile visualizzare il messaggio di notifica di modifica del database con i nuovi dispositivi. Si controlla dunque l'effettiva presenza dei nuovi dispositivi nella tabella **productorDevices**.

Result Grid | Filter Rows: | Edit: | Export/Import:

serialNumber	classDevice	fwVersion
002OO3	ALFA_GIULIA_2.0_TURBO_EXECUTIVE	1.2
06KYL6	FIAT_TIPO_1.3_MJT_POP	1.5
11L0TY	JEEP_RENEGADE_1.3_T4_LONGITUDE	1.0
CT5643	ALFA_GIULIA_2.0_TURBO_EXECUTIVE	1.1
JDG45G	ALFA_GIULIA_2.0_TURBO_VELOCE	1.6
JJEN34	JEEP_RENEGADE_1.0_T3_LONGITUDE	1.0
JVM67C	FIAT_TIPO_1.3_MJT_LOUNGE	1.0
MEC75N	FIAT_TIPO_1.3_MJT_LOUNGE	1.1
MMMA54	ALFA_GIULIA_2.0_TURBO_BUSINESS	1.1
P5HTYQ	JEEP_RENEGADE_1.3_T4_LIMITED	1.2
VVN6JM	JEEP_RENEGADE_1.0_T3_LONGITUDE	1.1
NULL	NULL	NULL

Figura 3.10. Visualizzazione del database aggiornato con i nuovi dispositivi del produttore.

Si osservi come la tabella sia stata aggiornata correttamente. A questo punto, sono noti i device appena immessi in azienda, le loro versioni correnti dei firmware e si può caricare dentro ognuno di essi un file di configurazione appropriato.

Una volta settato il file di configurazione *config.js*, si lanciano gli applicativi dei dispositivi con ID **VVN6JM** e **JJEN34**, il primo con installata la versione firmware 1.1, per la classe **JEEP_RENEGADE_1.0_T3_LONGITUDE**, mentre il secondo con la versione 1.0, sempre per la classe **JEEP_RENEGADE_1.0_T3_LONGITUDE**.

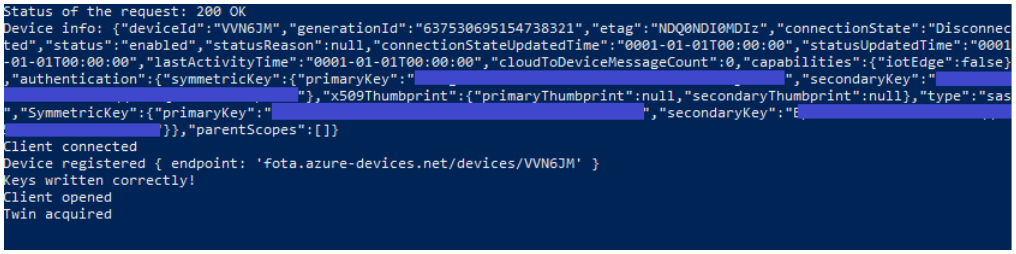

Figura 3.11. L'API registryDevice risponde con successo al dispositivo **VVN6JM.**

I dispositivi entrano in funzione e richiamano l'API *registryDevice* che, dopo aver validato le richieste, darà ordine all'IoT hub di iscrivere i nuovi dispositivi nel registro delle identità. L'IoT hub rilascia le chiavi simmetriche personalizzate per i due dispositivi, i quali le utilizzano per creare una chiave di accesso condiviso e salvano le informazioni in un file locale. Grazie al

SAS token, i due dispositivi aprono una connessione client e acquisiscono le informazioni sul dispositivo gemello. Dal portale di Azure si controlla che i due nuovi devices siano effettivamente registrati.

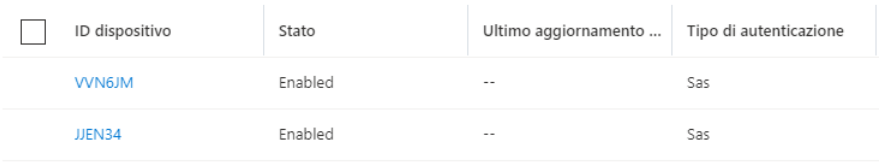

Figura 3.12. **JJEN34** e **VVN6JM** sono registrati e autenticati correttamente all'IoT hub.

Si supponga che il firmwarista abbia preparato tre aggiornamenti che ha archiviato in tre file **.zip** per risparmiare banda e permettere una connessione più veloce. I tre file in questione sono i seguenti:

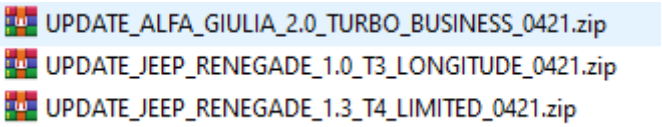

Figura 3.13. I tre aggiornamenti che il firmwarista ha preparato e archiviato nei file .zip.

A questo punto, si vogliono simulare tramite Postman le richieste del firmwarista per i tre aggiornamenti verso l'API *saveNewFirmware*.

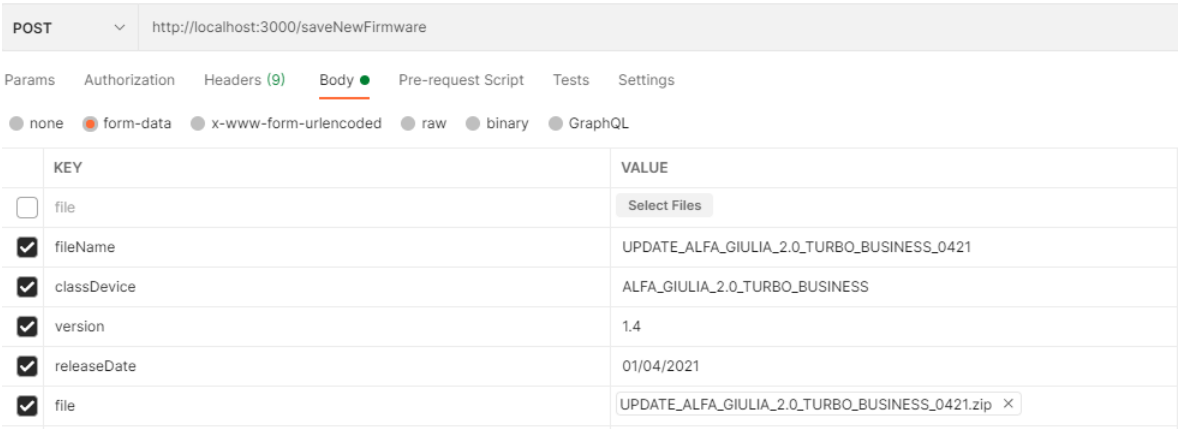

Figura 3.14. Preparazione della richiesta POST per il nuovo file **UPDATE_ALFA_GIULIA_2.0_TURBO_BUSINESS_0421.**

La risposta ottenuta dal server è l'URL in cui è stata archiviata la risorsa nel blob storage, come si può notare dalla figura **3.15**

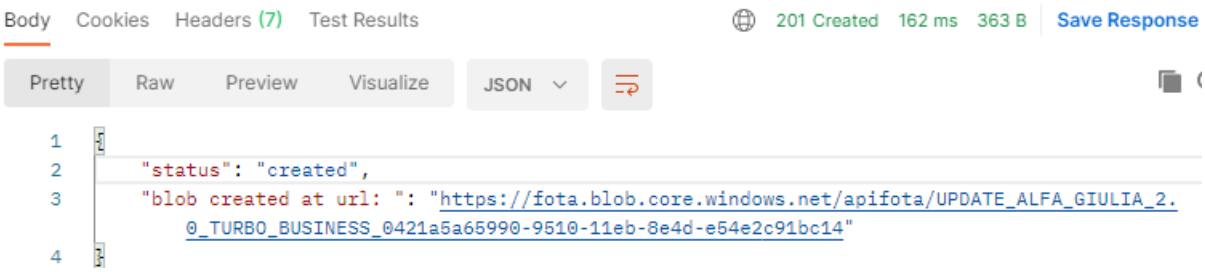

Figura 3.15. Risposta alla richiesta POST per saveNewFirmware per il file **UPDATE_ALFA_GIULIA_2.0_TURBO_BUSINESS_0421.**

Sempre tramite Postman si effettua una nuova richiesta per il caricamento del file **UPDATE_JEEP_RENEGADE_1.3_T4_LIMITED_0421**.

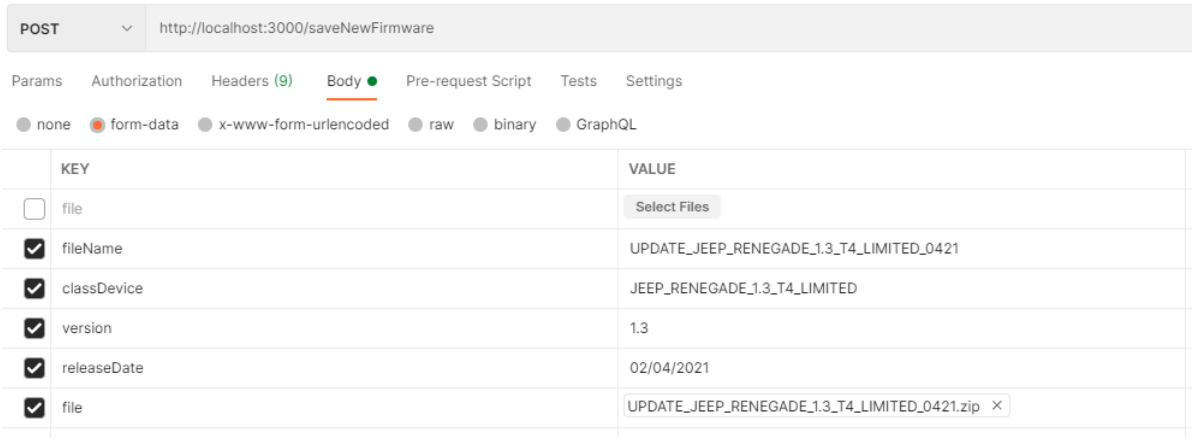

Figura 3.16. Preparazione della richiesta POST per il nuovo file **UPDATE_JEEP_RENEGADE_1.3_T4_LIMITED_0421.**

La risposta ottenuta è nuovamente l'URL di archiviazione della risorsa all'interno del blob storage.

Figura 3.17. Risposta alla richiesta di POST per saveNewFirmware per il file **UPDATE_JEEP_RENEGADE_1.3_T4_LIMITED_0421.**

A questo punto si procede a caricare il file d'aggiornamento per la classe **UPDATE_JEEP_RENEGADE_1.0_T3_LONGITUDE**.

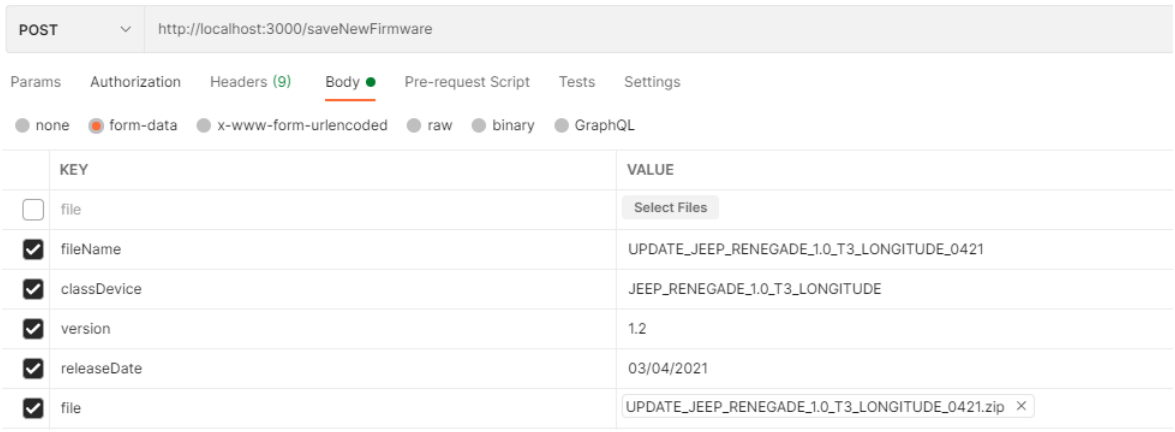

Figura 3.18. Preparazione della richiesta POST per il nuovo file **UPDATE_JEEP_RENEGADE_1.0_T3_LONGITUDE_0421.**

La risposta ritorna l'URL della locazione nello storage per quest'ultimo aggiornamento.

Figura 3.19. Risposta alla richiesta di POST per saveNewFirmware per il file **UPDATE_JEEP_RENEGADE_1.0_T3_LONGITUDE_0421.**

Se tutte e tre le richieste sono state servite correttamente, bisognerà trovare i tre nuovi record relativi ai file all'interno della tabella *firmware*.

fileNameID	urlFirmware	releaseDate	uploadDate	version	classDevice
UPDATE_ALFA_GIULIA_2.0_TURBO_BUSINESS_042...	https://fota.blob.core.windows.net/apifota/UP...	2021-01-04 00:00:00	2021-04-04 08:40:29	1.4	ALFA_GIULIA_2.0_TURBO_BUSINESS
UPDATE_JEEP_RENEGADE_1.0_T3_LONGITUDE_04...	https://fota.blob.core.windows.net/apifota/UP...	2021-02-04 00:00:00	2021-04-04 10:40:01	1.3	JEEP_RENEGADE_1.0_T3_LONGITUDE
UPDATE_JEEP_RENEGADE_1.3_T4_LIMITED_0421f...	https://fota.blob.core.windows.net/apifota/UP...	2021-02-04 00:00:00	2021-04-04 10:37:32	1.3	JEEP_RENEGADE_1.3_T4_LIMITED
NULL	NULL	NULL	NULL	NULL	NULL

Figura 3.20. Tabella firmware aggiornata con i dati sui nuovi file.

Si vuole ora controllare se i tre file siano effettivamente disponibili sul blob storage.

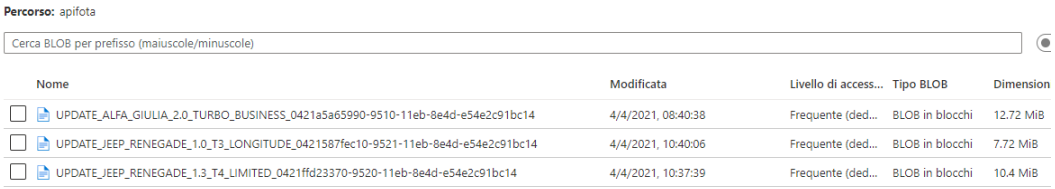

Figura 3.21. File zip archiviati correttamente sul blob storage.

Visto che le API *saveNewFirmware* e *registryDevice* si sono comportate nella maniera attesa, ci si aspetta che i due dispositivi che si sono registrati all'IoT hub, **VVN6JM** e **JJEN34**, abbiano scaricato correttamente il loro aggiornamento per la classe **JEEP_RENEGADE_1.0_T3_LONGITUDE**. Si va a guardare dall'interfaccia dei due dispositivi la risposta che hanno avuto dall'API *getAvailableFirmware*.

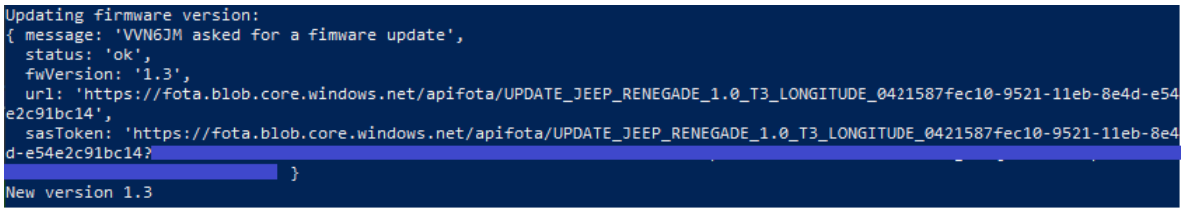

Figura 3.22. Download del file di aggiornamento per il dispositivo **VVN6JM** completato con successo.

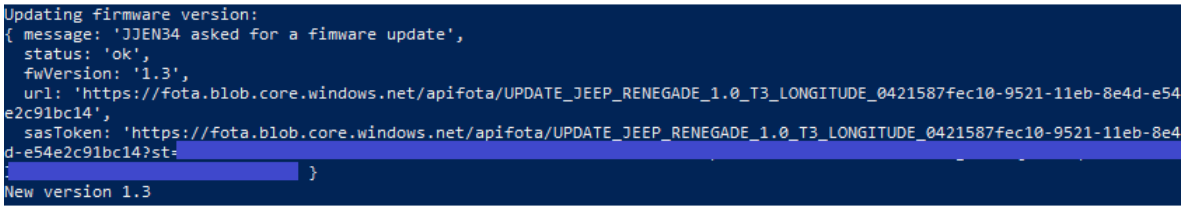

Figura 3.23. Download del file di aggiornamento per il dispositivo **JJEN34** completato con successo.

I file sono stati correttamente scaricati, inoltre l'applicazione di back-end ha agito contemporaneamente sulle richieste dei due device come si nota dall'interfaccia del server.

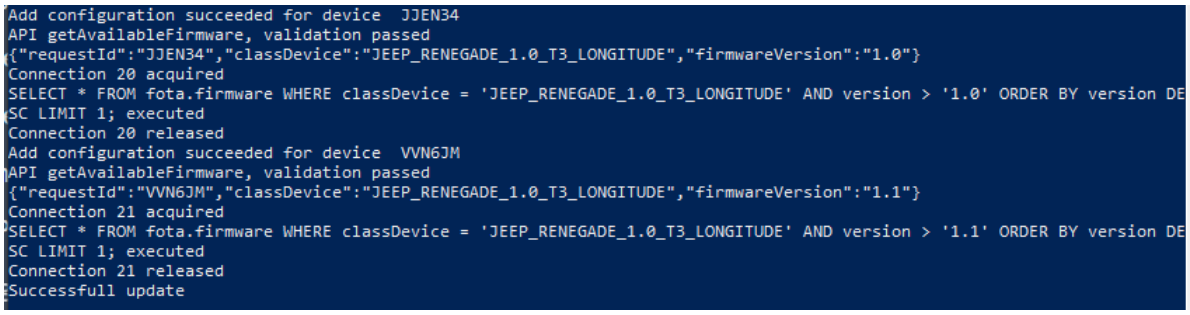

Figura 3.24. Risposta del server alle richieste di aggiornamento dei due device.

Si noti la risposta del validator dell'API *getAvailableFirmware* che valida le configurazioni dei due dispositivi e passa il controllo al *getAvailableFirmwareSvc* che effettua la query di controllo delle versioni sulla tabella SQL *firmware*. Considerando che **JJEN34** ha la versione 1.0 caricata al suo interno e **VVN6JM** ha la versione 1.1, ad entrambi viene notificata la presenza di un nuovo firmware per la classe **JEEP_RENEGADE_1.0_T3_LONGITUDE**, in particolare la versione più recente trovata nel database è la 1.3, di cui verrà ritornato l'URI firmato col SAS token che darà loro diritto al download. A sessione esaurita, le connessioni vengono rilasciate nuovamente nella cache del pool dove attenderanno una nuova chiamata.

Il flusso è stato provato solamente con dispositivi simulati, a causa della situazione emergenziale ancora in corso al momento dell'elaborazione di questo progetto di tesi, ma il funzionamento è il medesimo se gli applicativi Node JS vengono installati su dispositivi IoT fisici. Inoltre, potendo il numero di dispositivi connessi all'IoT hub crescere fino a 1.000.000, non si sono dunque riscontrati problemi relativi all'aumento del carico per l'IoT hub. Non avendo avuto dispositivi fisici a disposizione, si è presupposto che il sistema operativo dei devices fosse in grado di estrarre gli aggiornamenti dagli zip e completare l'update senza criticità.

Capitolo 4

Conclusioni

L'applicazione funziona bene ed è responsiva con i dispositivi simulati e può essere utilizzata efficacemente a livello aziendale per l'update remoto del firmware.

Lo studio del framework Node JS ha permesso di costruire il server e realizzare tutte le funzioni che ci si erano preposte all'inizio del progetto. Ciò nonostante, sono state ideate in corso d'opera alcune migliorie che, però, per motivazioni di tempo e di costi, non è stato possibile implementare all'interno del progetto. Tra queste:

- I file di aggiornamento possono essere anche dell'ordine delle decine di megabyte per dispositivi reali e non sempre si è dotati di una buona connessione, come per esempio può accadere con la sim di un dispositivo; per questo motivo, il firmwarista potrebbe generare un file di checksum attraverso una funzione crittografica di hash **md5** e caricarla insieme all'aggiornamento sullo storage. Si può quindi implementare la stessa funzione di hash anche sull'applicativo del device che, al termine del download, andrà a calcolare il checksum del file scaricato; se i due codici corrispondono, non c'è stata corruzione del file durante il download. Se i codici sono diversi, allora bisognerà rieseguire il download.

- Se il numero dei dispositivi cresce di alcuni ordini di grandezza, il calcolatore farà molta difficoltà a mantenere bassa la latenza dei confronti del server MySQL hostato in locale. Per questa ragione, l'implementazione del database può avvenire direttamente sul cloud attraverso un servizio PaaS come **Azure SQL** e lasciare gestire le risorse direttamente al cloud. Inoltre, il backup dei dati è sicuro e può avvenire efficacemente con ridondanza geografica.

- Il server può essere hostato su una macchina virtuale IaaS per implementazioni su larga scala: così facendo, si possono aumentare le prestazioni a livello di CPU e di memoria in maniera flessibile e pagando l'infrastruttura solamente in base al consumo.
- Le API possono essere implementate direttamente sul cloud attraverso un servizio PaaS chiamato **Azure Functions**; il servizio è ottimizzato per tutte quelle applicazioni che richiedono un trigger per essere richiamate, come nel caso delle API di questo progetto. In questo modo, è possibile mantenere tutto il sistema sul cloud.

Le migliorie sopra citate verranno con buona probabilità implementate in futuro, considerando che chi scrive continuerà sicuramente ad approfondire le tecnologie *Firmware Over The Air*.

Bibliografia

[1] Laura Zanotti, *Cosa significa IOT: come e perché così si rende il mondo (e il business) più smart*, https://www.digital4.biz/executive/digital-transformation/iot-cosa-significa-e-come-si-fa-a-rendere-il-mondo-ed-il-business-piu-smart/, feb. 2021

[2] *IaaS: l infrastruttura cloud altamente scalabile*, https://www.ionos.it/digitalguide/server/know-how/iaas-infrastructure-as-a-service/

[3] *PaaS: introduzione a Platform as a Service*, https://www.ionos.it/digitalguide/server/know-how/paas/

[4] *Che cos'è il Saas (Software as a service)?*, https://www.ionos.it/digitalguide/server/know-how/saas-software-as-a-service/

[5] Emily Maxie, *Manual vs. OTA Firmware Updates for IoT*, https://www.verypossible.com/insights/manual-vs-ota-firmware-updates-for-iot#:, Mag. 2019.

[6] Sean O'Kean, *Tesla can change so much with over-the-air updates that its messing with some owners heads*, https://www.theverge.com/2018/6/2/17413732/tesla-over-the-air-software-updates-brakes, giu. 2018

[7] Jeffrey Lean, *Over-The-Air Firmware: The Critical Driver of IoT Success*, https://hackernoon.com/over-the-air-firmware-the-critical-driver-of-iot-success-f4604bd0b881, nov. 2017

[8] Janakiram MSV, *A Look Back At Ten Years Of Microsoft Azure*, https://www.forbes.com/sites/janakirammsv/2020/02/03/a-look-back-at-ten-years-of-microsoft-azure/?sh=1067740b4929, feb. 2020

[9] Logan McCoy, *Microsoft Azure Explained: What It Is and Why It Matters*, https://ccbtechnology.com/what-microsoft-azure-is-

and-why-it-matters/

[10] *Comprendere e richiamare metodi diretti dall'hub IoT*, https://docs.microsoft.com/it-it/azure/iot-hub/iot-hub-devguide-direct-methods

[11] *Comprendere e usare dispositivi gemelli nell'hub IoT*, https://docs.microsoft.com/it-it/azure/iot-hub/iot-hub-devguide-device-twins

[12] *SDK: cosè un Software Development Kit?*, https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/software-development-kit/, set. 2019

[13] Kiran Malvi, *The Positive and Negative Aspects of Node.js Web App Development*, https://www.mindinventory.com/blog/pros-and-cons-of-node-js-web-app-development/, giu. 2018

[14] *Informazioni di riferimento: scegliere un protocollo di comunicazione*, https://docs.microsoft.com/it-it/azure/iot-hub/iot-hub-devguide-protocols

[15] *Introduzione all'archiviazione BLOB di Azure*, https://docs.microsoft.com/it-it/azure/storage/blobs/storage-blobs-introduction

[16] Petr Gazarov, *What is an API? In English, please.*, https://www.freecodecamp.org/news/what-is-an-api-in-english-please-b880a3214a82/, dec. 2019

[17] *What is REST*, https://restfulapi.net/

[18] *Postman API Client*, https://www.postman.com/product/api-client/

[19] *What is MySQL?*, https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html

[20] *MySQL Workbench Manual*, https://dev.mysql.com/doc/workbench/en/wb-intro.html

[21] Christoph Siegert, Debra Shinder ,David Burt, *Enabling Data Residency and Data Protection in Microsoft Azure Regions*, 2020

[22] mysql library, https://www.npmjs.com/package/mysql

[23] *Azure Storage Blob client library for JavaScript*, https://www.npmjs.com/package/@azure/storage-blob

[24] *Microsoft Azure Storage SDK for Node.js and JavaScript for Browsers*, https://www.npmjs.com/package/azure-storage

[25] *Microsoft Azure IoT service SDK for Node.js*, https://www.npmjs.com/package/azure-iothub

[26] *Tiny Validator (for v4 JSON Schema)*, `https://www.npmjs.com/package/tv4`
[27] *uuidv1*, `https://www.npmjs.com/package/uuidv1`
[28] *restify*, `http://restify.com/docs/plugins-api/#acceptparser`
[29] *Client class*, `https://docs.microsoft.com/en-us/javascript/api/azure-iot-device/client?view=azure-node-latest`
[30] *Crypto*, `https://nodejs.org/api/crypto.html#crypto_crypto`