POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Applicazione web in cloud AWS per il monitoring di dispositivi LoRa

Relatore
prof. Giovanni Malnati

Candidato
Giuliano Portaro

Supervisore aziendale
STMicroeletronics
Ing. Davide Sergi

A.A. 2018-2019

Indice

Introduzione

STMicroelectronics, azienda leader nel mercato della produzione di componenti elettronici a semiconduttore, si è riposizionata negli ultimi anni principalmente su due mercati, *Smart Driving* e *IoT*:

- **Smart Driving:** rientra in questo ambito tutta l'elettronica dedicata ai clienti automotive, richiedenti controllo motore, sistemi avanzati di assistenza alla guida e guida autonoma, dispositivi di safety, dispositivi per electric charging etc.
- **Internet of Things:** data la vastità e la frammentazione del mercato IoT, rientrano in questo ambito la maggior parte dei clienti ST, dai key accounts alle decine di migliaia di SME/start-ups servite attraverso i canali di distribuzione. Rientrano nel segmento IoT tutte le famiglie di microcontrollori a 32 bit (STM32), sensori, attuatori e connettività low power.

Per sfruttare le potenzialità del mercato IoT e per incrementare l'utilizzabilità dei componenti, ST ha progressivamente introdotto dal 2014 delle schede general purpose di prototipazione: *STM32 Nucleo*, *expansion boards* e *IoT Discovery Kit*. Contestualmente, ST ha sviluppato e rilasciato pacchetti FW a corredo delle schede di prototipazione che permettono di implementare e testare in breve tempo le funzionalità on-board.

Motivazioni: Perché una dashboard web?

La necessità di un'applicazione in *cloud mass market*, fortemente personalizzata per i firmware ST, e generica nelle funzionalità, ha portato l'ST alla realizzazione di una dashboard web. La soluzione si pone come obiettivo la creazione di un servizio gratuito, anonimo e già pronto all'uso. Oltre alle motivazioni esplicitate, si aggiunge una motivazione interna all'azienda poiché, attualmente, ST non possiede nel settore marketing/vendite competenze tecniche specifiche sullo sviluppo di applicazioni cloud che permettano di promuovere efficacemente HW and FW.

Overview

La tesi è stata realizzata in collaborazione con *STMicroelectronics* e si pone come obiettivo la realizzazione di un'applicazione web per l'asset-tracking dei dispositivi ST connessi alla rete *LoRa* tramite protocollo di rete *LoRaWAN*.

Durante la prima fase del lavoro di tesi è stato condotto uno studio dettagliato del protocollo *LoRa*, focalizzando l'attenzione sulle caratteristiche di sicurezza da esso offerte. A fronte di alcune debolezze individuate, si è deciso di definire uno strato di sicurezza aggiuntivo in grado di offrire una comunicazione più sicura in termini di integrità dei dati trasmessi dagli end-device (prevenendo il fenomeno del *device impersonation)*. Nella seconda fase ci si è incentrati sul design e implementazione del back-end della web application. Di particolare interesse gode l'infrastruttura di comunicazione cloud-to-cloud messa in piedi tra AWS e Loriot al fine di instradare e storicizzare i dati inviati su rete LoRa dagli end-device. In ultimo, è stato progettato e sviluppato il font-end della web application con l'obiettivo di svincolarlo il più possibile dalle tecnologie back-end utilizzate.

Il nodo IoT oggetto dell'applicazione è un dispositivo STM32-based modulare composto da [1]:

- *B-L072Z-LRWAN:* board di sviluppo, ospita il firmware applicativo e fornisce connettività LoRa. Ha il set completo di funzioni disponibili nella serie *STM32L0* e offre funzioni *RF ultralow-power* e *LoRa*, inclusi LED, pulsanti, antenna, connettori *Arduino Uno v3* e connettore *USB 2.0 FS* in formato *Micro-B*.

- *X-NUCLEO-GNSS1A1: board di espansione* basata sul modulo Teseo-LIV3F GNSS. Rappresenta un modulo *GNSS (Global Satellite Navigation System)* economico e di facile utilizzo, che incorpora un ricevitore di posizionamento stand alone indipendente *TeseoIII*, utilizzabile in diverse configurazioni nel progetto *Nucleo STM32*.

- *X-NUCLEO-IKS01A2: board di espansione* che integra sensori ambientali e MEMS di movimento, permette di ottenere dati di telemetria del dispositivo. È compatibile con il layout del connettore *Arduino UNO R3* ed è progettato attorno all'*accelerometro 3D LSM6DSL* e *LSM303AGR* e al *giroscopio 3D*, al *magnetometro 3D*, al *sensore di umidità* e *temperatura HTS221* e al *sensore di pressione LPS22HB*.

Il function pack *FT-ATR-LORA1, sviluppato nell'ambito del programma STM32 ODE,* rappresenta il firmware del dispositivo abilitandolo alla lettura dei dati ambientali e di movimento, di GNSS e all'invio dei dati raccolti tramite protocollo *LoRaWAN*. Il firmware implementa profili a bassa potenza per garantire una lunga autonomia della batteria. La libreria middleware supporta le specifiche *LoRaWAN 1.0.2* e non quelle *1.1.x* e per tale motivazione è stato analizzato il *protocollo 1.0.2*, nonostante le funzionalità di sicurezza inferiori.

Al fine di testare le funzionalità di sicurezza aggiuntive che saranno proposte al termine del *primo capitolo*, è stato realizzato un ambiente virtuale che permette di simulare un dispositivo inviando dati di telemetria e di posizione generati randomicamente con l'aggiunta di una firma digitale al fine di ottenere le proprietà di autenticazione e di integrità dei dati stessi.

Per lo sviluppo dell'applicazione web è stato utilizzato il service network provider Loriot e il cloud provider *Amazon Web Service* (AWS). Il primo si occupa di comunicare direttamente con i dispositivi fisici e fare il forward ad AWS dei dati ricevuti. Il secondo ospita l'intera architettura e logica back-end dell'applicazione web.

1.0 LoRaWAN

LoRa (Long Range) è un protocollo di comunicazione wireless a lungo raggio che ad oggi compete con le altre grandi tecnologie *Low-Power Wide-Area Network* (*LPWAN*). L'idea che sta alla base di LoRa è quella di permettere ad un gran numero di sensori, alimentati a batteria e sparsi per il mondo, di poter effettuare comunicazioni a lungo raggio, con distanze che variano tra i 2 km negli ambienti urbani ad alta densità ai 20 km negli ambienti liberi. La velocità di trasmissione dei dati è compresa tra i 0.3 kbps ai 50 kbps, ma con un consumo della batteria del sensore molto contenuto che garantisce una messa in campo dei sensori ambientali anche per molti anni [2].

Long Range Wide Area Network (LoRaWAN) è un protocollo, sviluppato dalla *LoRa Alliance*, progettato per collegare in modalità wireless dispositivi a batteria ad Internet. L'architettura LoRaWAN è sviluppata tramite una topologia a stella dove i gateway inoltrano i messaggi ricevuti dagli end-device ad un network server centrale. I gateway sono collegati al server di rete tramite connessioni IP standard e fungono da bridge trasparente, semplicemente convertendo i pacchetti RF provenienti dai sensori in pacchetti IP diretti verso il network server centrale e viceversa. La comunicazione wireless sfrutta le caratteristiche dello strato fisico LoRa, consentendo un collegamento single-hop tra il l'end-device e uno o più gateway [3].

Le funzionalità di sicurezza del protocollo aggiungono la proprietà di confidenzialità dei dati effettuandone una cifratura AES in modo che nessuno durante la trasmissione, neanche il gateway, sarà in grado di poter leggere i dati inviati dal sensore. Inoltre, è anche offerta la proprietà di integrità dei dati, ovvero un attaccante non può modificare i dati in transito tra il gateway e il network server senza che il destinatario ne venga a conoscenza.

Le specifiche del protocollo sono state recentemente aggiornate al protocollo *1.1.x* (2017), ma a causa di mancata compatibilità con l'hardware a disposizione durante lo svolgimento della tesi è stato analizzato esclusivamente il protocollo *1.0.2*. Nei capitoli successivi saranno analizzati gli aspetti salienti della versione finale del protocollo *1.0.2* facendo riferimento alla documentazione ufficiale [4].

1.1 Formato dei messaggi

La terminologia LoRa distingue tra **messaggi** di **uplink** e **downlink**.

I *messaggi di uplink* (Fig. 1) sono inviati dall'end-device verso il network server e inoltrati da uno o più gateway. Tali messaggi utilizzano la modalità esplicita del pacchetto radio LoRa in cui il trasmettitore aggiunge l'intestazione fisica LoRa (*PHDR*), un CRC di intestazione (*PHDR_CRC*) e un *CRC del payload* che ne protegge l'integrità.

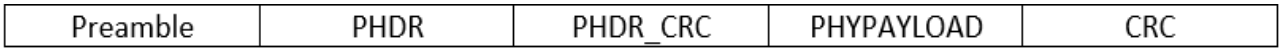


Figura 1: Struttura messaggio fisico di Uplink

I *messaggi di downlink* (Fig. 2) sono inviati dal network server verso un unico end-device ed inoltrato da un unico gateway. Tali messaggi utilizzano il pacchetto radio in modalità esplicita includendo l'header fisico LoRa (*PHDR*) e un header CRC (*PHDR_CRC*).

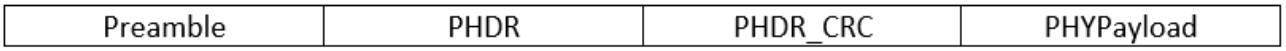


Figura 2: Struttura messaggio fisico di Downlink

Al fine di permettere la ricezione dei messaggi di downlink l'end-device apre due piccole finestre di ricezione a seguito di ogni trasmissione in uplink. L'inizio della prima finestra di ricezione è definito utilizzando la fine della trasmissione come referenza, ovviamente la lunghezza di tale finestra deve essere grande abbastanza da permettere all'end-device di rilevare il preambolo di downlink. Infatti, se un preambolo viene rilevato durante una delle finestre di ricezione, il ricevitore radio rimarrà attivo fino a quando il frame di downlink non sarà demodulato.

1.2 Formato dei messaggi MAC

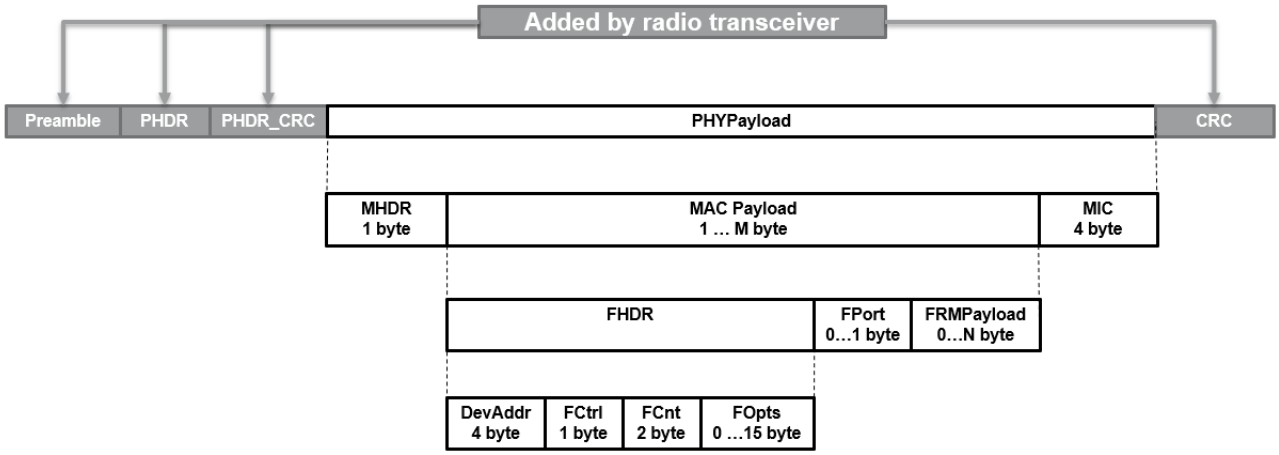


Figura 3: Formato dei messaggi MAC

Entrambi i messaggi, sia di uplink che di downlink, trasportano un payload composto da un header (*MHDR*), un payload MAC (*MAC Payload*) e un codice MIC, al fine di preservare l'integrità del messaggio. A sua volta, il MAC Payload è composto dal *frame header* (*FHDR*) che contiene l'indirizzo dell'end-device (*DevAddr*), un frame di controllo (*FCtrl*), un counter (*FCnt*) ed un frame che contiene le opzioni (*FOpts*) e viene utilizzato per trasportare i comandi MAC.

DataRate	*M*	*N*
0	19	11
1	61	53
2	133	125
3	250	242
4	250	242
5:7	Not defined	
8	41	33
9	117	109
10	230	222
11	230	222
12	230	222
13	230	222
14:15	Not defined	

Figura 4: Massima dimensione del MAC payload

Una nozione fondamentale è che c'è una massima quantità di dati che possono essere inviati in un singolo pacchetto. Come si evince dalla *figura 3*, la massima dimensione del campo MAC Payload varia in un range da $1 \leq length(payload) \leq M$, dove il valore di *M* è estratto dalla *figura* 4. Esso deriva dal tempo di trasmissione massimo consentito allo strato *PHY* tenendo conto di un eventuale incapsulamento del ripetitore. Il valore di *N* potrebbe essere inferiore se il campo *FOpts* non è vuoto. Le linee in grigio corrispondono alle velocità di dati che possono essere utilizzate da un end-device dietro un ripetitore, mentre se l'end-device non opererà al di sotto di un ripetitore la massima lunghezza del payload, in assenza del campo opzionale *FOpts*, risulterebbe maggiorata di 20 byte nei data rate compresi tra 8 e 13.

1.2.1 Frame Counter (FCnt)

Ogni end-device ha due frame counter: *FCntUp*, che tiene traccia del numero di frame di dati inviati in uplink e viene incrementato dall'end-device, *FCntDown*, che raccoglie il numero di frame di dati ricevuti dall'end-device in downlink e viene incrementato dal network server. Il network server tiene traccia del counter di uplink e genera i counter di downlink per ogni end-device registrato all'interno del network. Dopo uno scambio dei messaggi di *JoinRequest*/*JoinAccept* o dopo un reset, i contatori sull'end-device e sul network server vengono resettati a 0. Sequenzialmente, *FCntUp* e *FCntDown* sono incrementati dal mittente di un'unità per ogni nuovo frame di dati inviato nella rispettiva direzione. Il protocollo LoRaWAN permette l'utilizzo di contatori da 16 o 32 bit; il network server necessita di essere informato, out-of-band, circa la grandezza del contatore che viene implementato all'interno dell'end-device: se si utilizza un contatore da 16 bit, il campo *FCnt* può essere utilizzato direttamente come valore di contatore, mentre se viene utilizzato un contatore da 32 bit, il campo *FCnt* corrisponde ai 16 bit meno significativi dei 32 bit del contatore.

1.2.2 Cifratura del payload del pacchetto MAC

Se il frame dati contiene un payload, *FRMPayload* deve essere cifrato prima che il MIC venga calcolato. La cifratura utilizzata è basata sull'algoritmo descritto nell'*IEEE 802.15.4/2006 Annex B [IEEE802154*] utilizzando AES con una chiave da 128 bit. Inoltre, la chiave utilizzata per la cifratura dipende dal campo *Fport* che specifica se i dati sono destinati al network server e quindi devono essere cifrati con la chiave *NwkSKey*, porta 0, o sono destinati all'application server e quindi cifrati con la chiave *AppSKey*, porta nell'intervallo $[0, 255]$.

Size (bytes)	1	4	1	4	4	1	1
A_i	0x01	4 x 0x00	Dir	DevAddr	FCntUp o FCntDown	0x00	i

Figura 5: Cifratura del pacchetto

Per ogni messaggio, l'algoritmo definisce una sequenza di blocchi A_i con:

$$i = 1 \dots k$$
$$k = cell(len(FRMPayload)/16)$$

Il campo *Dir* rappresenta la direzione del messaggio ed assume valore *0* per i messaggi di uplink, mentre assume valore *1* per i messaggi di downlink.

I blocchi A_i sono cifrati per ottenere una sequenza *S* di blocchi S_i come segue:

$$S_i = aes128\_encrypt(K, A_i)$$

$$S = S_1 | S_2 | \dots | S_k$$

Mentre, la cifratura e la decifratura del payload sono ottenute effettuando una troncatura per i primi $len(FRMPayload)$ ottetti:

$$(FRMPayload \mid pad_{16}) \oplus \mathrm{S}$$

1.2.3 Message Integrity Code (MIC)

Lo scopo del MIC è quello di assicurarsi che i dati trasmessi non siano stati modificati da una terza parte in ascolto lungo il canale di comunicazione. Il MIC viene generato tramite un metodo che deve essere noto sia al trasmettitore che al ricevitore in modo che il trasmettitore dopo aver

generato il codice MIC possa allegarlo ai dati e il ricevitore, ripetendo la medesima operazione, possa verificare l'autenticità dei dati ricevuti. La sicurezza di tale procedura si basa sul fatto che nessun'altro possa generare un nuovo MIC, essendo la chiave utilizzata per generarlo nota esclusivamente ai due peer della comunicazione. Inoltre, il livello di sicurezza della soluzione viene incrementato operando sulla lunghezza del MIC, essendo la robustezza esponenziale al numero di bit utilizzati.

Nel caso del protocollo LoRaWAN, il MIC viene calcolato su tutti i campi del messaggio come segue:

$$msg = MHDR \mid FHDR \mid FPort \mid FRMPayload$$
$$cmac = aes128\_cmac(NwkSKey, B_0 \mid msg)$$
$$MIC = cmac[0 \dots 3]$$

Dove il blocco B_0 è definito come segue:

Size (Bytes)	1	4	1	4	4	1	1
B_0	0x49	4 x 0x00	Dir	DevAddr	FCntUp o FCntDown	0x00	len(msg)

Figura 6: MIC

Il campo *Dir*, come nel caso della cifratura, assume valore *0* nel caso di messaggi di uplink e valore *1* nel caso di messaggi di downlink.

1.3 Attivazione dell'end-device

Al fine di partecipare al network LoRaWAN ogni end-device deve eseguire una procedura di attivazione che può essere effettuata tramite una delle seguenti modalità:

- **Activation By Personalization (ABP).**
- **Over-The-Air Activation (OTAA).**

La modalità *ABP* permette di bypassare la procedura di join, memorizzando direttamente all'interno dell'end-device il *DevAddr* e le chiavi *NwkSKey* e *AppSKey* in fase di produzione o di deploy dell'end-device. Questa modalità è stata scartata a favore della modalità *OTAA* (che sarà descritta nel capitolo 1.3.1), a causa di una sicurezza inferiore offerta per l'impossibilità di refreshing delle chiavi di sessioni che risultano configurate manualmente sul dispositivo e quindi non modificabili una volta che l'end-device è in esecuzione.

Al termine della procedura di attivazione dell'end-device, le seguenti informazioni devono essere memorizzate all'interno dell'end-device: l'indirizzo del dispositivo (*DevAddr*), un identificatore dell'applicazione (*AppEUI*), una network session key (*NwkSKey*) e un'application session key (*AppSKey*).

1.3.1 End-device address (DevAddr)

Il *DevAddr* è composto da 32 bit e permette di identificare il dispositivo all'interno del network ed ha il seguente formato:

Bit#	[31…25]	[24…0]
DevAddr bit	NwkID	NwkAddr

Figura 7: Formato DevAddr

I 7 bit più significativi sono utilizzati come identificatore del network (*NwkID*) per separare l'indirizzo territoriale che può essere sovrapposto tra i diversi operatori andando a risolvere il problema del roaming. Mentre, i 25 bit meno significativi, che rappresentano l'indirizzo di rete, possono essere scelti arbitrariamente.

1.3.2 Application Identifier (AppEUI)

L'*AppEUI* è un identificativo globale dell'applicazione nello spazio degli indirizzi *IEEE EUI64* che identifica in maniera univoca l'entità in grado di elaborare il la *JoinRequest*. Si noti che l'*AppEUI* è memorizzato nell'end-device prima che la procedura di attivazione sia eseguita.

1.3.3 Application Session Key (AppSKey)

La *AppSKey* è una *application session key* specifica per l'end-device e che viene utilizzata sia dall'application server che dall'end-device per cifrare e decifrare il payload del messaggio. Il payload applicativo è cifrato in maniera end-to-end tra il dispositivo e l'application server **ma non protetto dall'integrità**. Questo significa che il network server può essere in grado di alterare il contenuto dei dati in transito senza che l'application server possa rilevarne l'alterazione. Proprio per questo motivo i network server sono considerati entità *trusted* all'interno del protocollo.

1.4 Over-the-Air Activation (OTAA)

Al fine di partecipare allo scambio dei dati con il network server, l'end-device deve eseguire la *join procedure*; tale procedura deve essere, inoltre, eseguita ogni qualvolta l'end-device perde le informazioni di contesto della sessione. La procedura di join richiede che l'end-device sia personalizzato con le seguenti informazioni prima di avviare la *join procedure*: un identificativo globalmente univoco del dispositivo (*DevEUI*), l'identificativo dell'applicazione (*AppEUI*) e una chiave AES-128 (*AppKey*).

1.4.1 Identificativo dell'end-device (DevEUI)

Il *DevEUI* è un identificativo unifico dell'end-device nello spazio degli indirizzi IEEE EUI64 e che permette di identificare univocamente il dispositivo.

1.4.2 Application key (AppKey)

La *AppKey* è una chiave AES-128 specifica per ogni end-device, questo comporta che la compromissione della *AppKey* per un end-device non vada a compromettere tutti gli altri end-device registrati nel network, essendo le varie *AppKey* differenti tra loro. Quando un end-device effettua la *procedura di join* con un network tramite attivazione *OTAA*, la *AppKey* è utilizzata per derivare le chiavi di sessione *NwkSKey* e *AppSKey*, risultando anch'esse specifiche per l'end-device e, inoltre, per la sessione corrente. Tali chiavi vengono utilizzate rispettivamente per verificare l'integrità della comunicazione con il network server e per la cifratura dei dati applicativi.

1.4.3 Join Procedure

Dal punto di vista dell'end-device, la *join procedure* consiste in due messaggi MAC scambiati con il network server; tali messaggi prendono il nome di **join request** e **join accept**.

La *join procedure* è sempre iniziata dall'end-device tramite l'invio di un messaggio di *join request*, e tale messaggio non viene cifrato, con il seguente formato:

Size (bytes)	8	8	2
Join Request	AppEUI	DevEUI	DevNonce

Figura 8: Messaggio di Join Request

Il messaggio di *join request* contiene l'*AppEUI* e il *DevEUI* dell'end-device seguiti da un *nonce* (*number used only once*) di due ottetti chiamato *DevNonce*, ovvero un valore random che deve essere utilizzato esclusivamente una volta. Per ogni end-device, il network server tiene traccia di un certo numero di *DevNonce* utilizzati in passato, e ignora messaggi di *join request* che possiedono lo stesso valore di *DevNonce* per un end-device. Questo meccanismo permette di prevenire parzialmente gli attacchi replay, ovvero attacchi basati sull'invio dello stesso messaggio inviato precedentemente da un dispositivo al fine di ottenere accesso alla rete, portando alla disconnessione dell'end-device stesso dalla rete. La protezione è parziale poiché il network server tiene traccia esclusivamente di un numero limitato di *DevNonce*, ovvero quelli utilizzati più di recente.

Il MIC, per il messaggio di *join request*, è calcolato come segue:

$$cmac = aes128\_cmac(AppKey, MHDR \mid AppEUI \mid DevEUI \mid DevNonce)$$
$$MIC = cmac[0 \dots 3]$$

A seguito del messaggio di *join request*, il network server risponderà con un messaggio di *join accept* se l'end-device ha il permesso di effettuare il join con la rete. Il messaggio di *join accept* è inviato come un normale messaggio di *downlink* ma utilizzando dei delay differenti. Se il messaggio di *join request* viene rifiutato nessuna risposta sarà generata dal network server.

Size (bytes)	3	3	4	1	1	(16)
Join Accept	AppNonce	NetID	DevAddr	DLSettings	RxDelay	CFList

Figura 9: Messaggio di Join Accept

Il messaggio di *join accept* contiene un *application nonce* (*AppNonce*) composto da 3 ottetti, un identificatore di rete (*NetID*), l'indirizzo dell'end-device (*DevAddr*), un delay tra invio e ricezione *(RxDelay*) e opzionalmente una lista di canali di frequenza per la rete a cui l'end-device sta partecipando (*CFList*), notando che tale parametro è specifico per la regione ed è presente una specifica documentazione rilasciata dalla *LoRa Alliance*.

L'*AppNonce* è un valore random fornito dal network server e utilizzato dall'end-device per derivare le due chiavi di sessione *NwkSKey* e *AppSKey* come segue:

$$NwkSKey = aes128\_ecb\_encrypt(AppKey, 0x01 \mid AppNonce \mid NetID \mid DevNonce \mid pad_{16})$$
$$AppSKey = aes128\_ecb\_encrypt(AppKey, 0x02 \mid AppNonce \mid NetID \mid DevNonce \mid pad_{16})$$

Il MIC per il messaggio viene calcolato come segue:

$$cmac = aes128\_cmac(AppKey, MHDR \mid Join\ Accept)$$
$$MIC = cmac[0 \dots 3]$$

A differenza del messaggio di *join request*, il messaggio di *join accept* viene cifrato con la chiave *AppKey* come segue:

$$M = AppNonce \mid NetID \mid DevAddr \mid DLSettings \mid RxDelay \mid CFList \mid MIC$$

$$aes128\_ecb\_decrypt(AppKey, M)$$

Il network server utilizza un'operazione di decifratura, anziché di cifratura, al fine di permettere all'end-device di implementare esclusivamente la funzionalità di cifratura e non quella di decifratura. Questo permette di ridurre il consumo di memoria dovuto alla memorizzazione di ulteriori funzionalità crittografiche.

1.5 Protocollo di Comunicazione

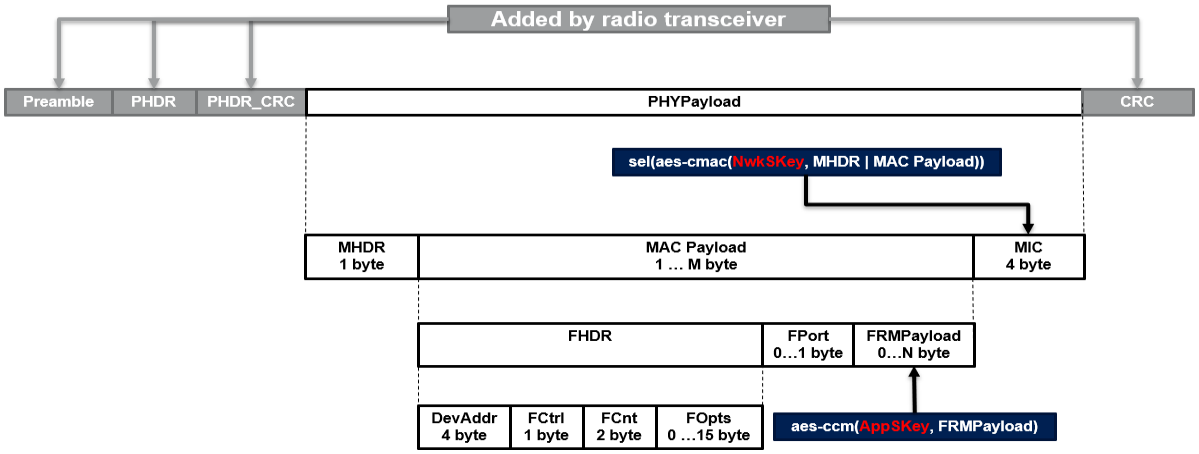


Figura 10: Formato pacchetto

Nella *figura 10* viene mostrato uno schema del pacchetto precedentemente descritto, mettendo in risalto le operazioni di cifratura che vengono effettuate sul pacchetto. All'interno del *PHYPayload* viene aggiunto il MIC al fine di garantire l'integrità dei dati in transito dall'end-device al network server: nel caso in cui il pacchetto non risulti integro, ovvero un attaccante ha manipolato i dati in transito, il network server provvederà immediatamente a scartarlo.

La procedura per calcolare il MIC segue:

$$cmac = aes\_cmac(NwkSKey, MHDR \mid MAC\ Payload)$$
$$MIC = cmac[0 \dots 3]$$

Come detto, il MIC permette di rilevare eventuali modifiche ai dati che transitano dall'end-device al network server, ma una volta esaminato viene scartato e non viene offerta alcune protezione d'integrità dei dati nel tratto tra il *network server* e l'*application server*.

Quando il valore di *FPort* è nell'intervallo $[1, \dots, 255]$, il campo *FRMPayload* contiene i dati applicativi, ovvero dati destinati all'application server, e il suo contenuto verrà cifrato utilizzando la *AppSKey* come chiave di cifratura; la procedura segue:

$$ciphertext = aes\_ccm(AppSKey, FRMPayload)$$

Mentre, nel caso in cui il campo *FPort* valga *0*, il campo *FRMPayload* contiene dati diretti al network server e viene cifrato utilizzando la chiave *NwkSKey*.

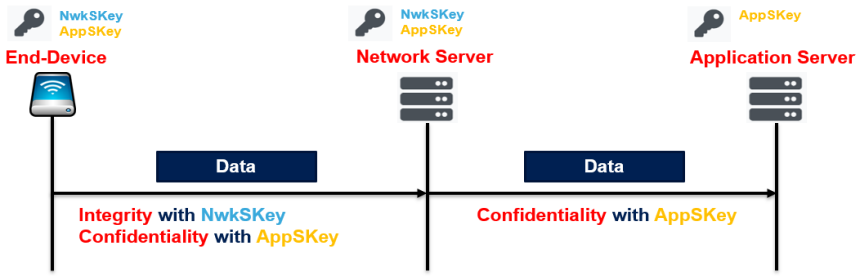


Figura 11: Proprietà di sicurezza

1.6 Problematiche di Sicurezza

Addentrandoci nell'analisi delle problematiche di sicurezza emerse dallo studio del protocollo *LoRaWAN 1.0.2,* andiamo a citare alcuni dei possibili attacchi e, nel capitolo successivo, le modifiche proposte al fine di mitigare le problematiche di sicurezza riscontrate.

1.6.1 Attacchi all'integrità

Il protocollo *LoRaWAN 1.0.2* offre integrità dei dati esclusivamente nel tratto che va dall'end-device al network server, non prevedendo ulteriori controlli nel tratto finale che va dal network server all'application server *(Fig. 11)*. Questo comporta che un'attaccante può apportare modifiche ai dati senza che l'application server possa rilevare tali modifiche. Nonostante i dati siano cifrati, ciò non preclude la possibilità di modifiche casuali nei dati trasmessi. Questo può creare possibili problemi che, in assenza di opportuni controlli e gestione di eccezioni, possono introdurre ulteriori vulnerabilità all'interno dell'application server.

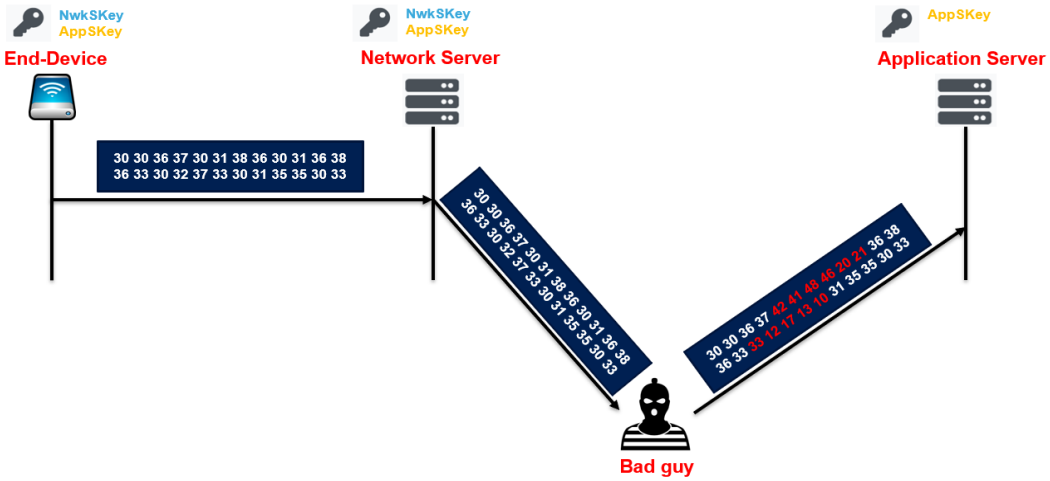


Figura 12: Esempio di attacco all'integrità dei dati

La *figura 12* mostra un possibile scenario in cui un attaccante, denominato *bad guy* in figura, effettua un attacco di *man-in-the-middle*, ovvero si interpone tra il mittente e il destinatario della comunicazione. Questo permette all'attaccante di entrare in possesso dei dati diretti all'application server, di modificarli e successivamente inoltrarli alterati alla destinazione finale, senza che la modifica possa essere rilevata per la mancanza di un meccanismo di sicurezza che garantisca la proprietà di integrità. L'attacco può diventare particolarmente pericoloso (*bit-flipping attack)*, quando l'attaccante conosce il formato del messaggio: in tale situazione, l'attaccante può modificare il messaggio non casualmente, ma in modo tale da alterarne il senso

dei dati trasmessi senza alterarne la struttura (ad esempio un valore di temperatura trasmesso può passare da 20° a 50°). Il danno apportato da questo genere di attacchi dipende notevolmente dalla tipologia di dati che vengono trasmessi dagli end-device all'applicazione.

1.6.2 Device Impersonation

In generale, un *impersonation attack* è un attacco ove una terza parte è in grado di assume l'identità di uno dei nodi coinvolti in una comunicazione di rete. Questo attacco può essere effettuato sia in un semplice sistema di comunicazione che durante lo svolgimento di un protocollo di autenticazione, così da permettere l'accesso a dati o servizi di cui non si ha l'autorizzazione d'accesso, come la lettura di informazioni o la gestione dei dispositivi associati ad un altro account utente. Si noti che un buon sistema di autenticazione deve possedere la proprietà di rendere trascurabile, o quasi del tutto assente, la possibilità che un terzo possa simulare l'identità di un altro nodo.

Nel caso del protocollo LoRaWAN, questo si traduce nella possibilità per un attaccante di emulare l'identità di un dispositivo *x*, comportando la possibilità di invio di dati falsi, l'esecuzione di azioni rivolte al dispositivo (ad esempio il caso banale di accensione o spegnimento di un led), di leggere eventuali statistiche riportate nel provider di riferimento, et cetera.

L'identificativo dell'end-device all'interno del network server è il *DevEUI*, un identifico globalmente univoco basato sull'IEEE EUI64. Tale identificativo viene univocamente assegnato al dispositivo al momento di produzione. Di conseguenza, la mancanza di un meccanismo non offerto dal protocollo che permetta di associare in maniera indissolubile i dati e il proprietario, rende possibili attacchi di *device impersonation*.

1.6.3 Attacchi fisici

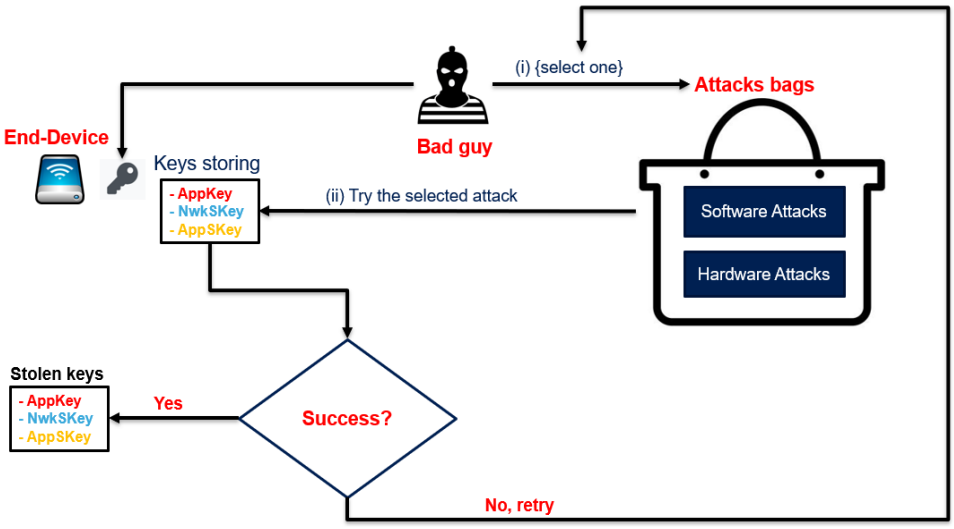


Figura 13: Attacchi fisici

Dopo aver analizzato i possibili attacchi che possono essere effettuati nel tratto di comunicazione tra il Network Server e l'Application Server vanno analizzate le possibili vulnerabilità presenti sull'end-device [5]:

- **Distruzione o Furto dell'End-Device:** la chiave principale (*AppKey*),da cui vengono generate le chiavi di sessione a seguito della *procedura di join*, è univocamente generata

per ogni dispositivo durante il tempo di fabbricazione del dispositivo o al tempo di deploy. Questo permette, nel caso di compromissione del singolo dispositivo, di non compromettere alcuna informazione all'interno del network, oltre che i dati memorizzati per lo specifico dispositivo;

- **Plaintext Key Capture:** se gli elementi di archiviazione sicura non vengono utilizzati e le chiavi sono archiviate in semplici file nell'end-device, questo attacco può rappresentare una grave minaccia per la riservatezza, l'integrità e la disponibilità della rete LoRaWAN;
- **Jamming Attack:** è un exploit che può essere utilizzato per compromettere l'ambiente wireless. Esso è basato sulla negazione del servizio agli utenti autorizzati poiché il traffico autorizzato è bloccato dal traffico illegittimo;
- **Impersonation Attack:** il gateway può essere impersonato per performare attacchi contro gli end-device, che possono essere monitorati e i loro indirizzi di rete determinati. Ancora più pericoloso è un metodo di triangolazione, che necessita di almeno 3 gateway per essere attuato, che può essere utilizzato per determinare la posizione fisica degli end-device.

1.6.4 Sicurezza delle chiavi

Le chiavi utilizzate nel protocollo vengono derivate, come già visto, dall'*AppKey* che deve essere nota sia all'end-device che al network server, il quale dovrà utilizzarla per cifrare il messaggio di *join accept*. La conoscenza della chiave root rende possibile, agli operatori della rete, la derivazione delle chiavi di sessione *NwkSKey* e *AppSKey*, rendendo di fatto trasparente per il network server i dati trasmessi nonostante le operazioni crittografiche apportate. Il *protocollo 1.0.2,* sotto questo punto di vista, implementa una politica di confidenzialità non completa e non perfettamente end-to-end. Di conseguenza, se i dati inviati dai sensori attraverso la rete risultassero sensibili, si consiglia di implementare un'ulteriore cifratura dei dati sull'end-device, in modo che la chiave utilizzata risulti nota esclusivamente al server provider di riferimento e non a terzi.

1.7 Soluzione proposta

In questa sezione andremo a discutere le problematiche di sicurezza affrontate nei capitoli precedenti:

1. *Problemi di integrità dei dati*
2. *Device Impersonation*
3. *Attacchi fisici all'end-device*
4. *Sicurezza delle chiavi*

I punti *(1)* e *(2)* sono affrontati nel capitolo *1.7.1,* il punto *(3)* nel capitolo *1.7.3*. Per quanto riguarda la sicurezza delle chiavi, si è deciso di non affrontare tale problematica in quanto il contesto applicativo non si presta bene a tal scopo. Infatti, significherebbe aggiungere ulteriore overhead all'invio dei dati e alle operazioni crittografiche da realizzare, che per un dispositivo con profilo low-power e banda di rete disponibile limitata (dell'ordine del centinaio di byte) rappresenterebbero un'importante criticità.

1.7.1 Certificato Digitale

Un certificato digitale è un documento elettronico che attesta l'associazione univoca tra una chiave pubblica e l'identità di un soggetto a cui è apposta la firma della CA che ha emesso il

certificato per evitarne modifiche. Grazie all'associazione tra chiave pubblica e identità del soggetto, in questo caso dell'end-device, è possibile evitare qualunque forma di *device impersonation*, andando ad aggiungere una firma digitale ai dati che l'end-device intende inoltrare all'interno della rete. Inoltre, la firma digitale permette di risolvere i problemi di integrità discussi nel capitolo *1.6.1*, poiché permette al destinatario di verificare se i dati hanno subito qualunque forma di alterazione da parte di un possibile attaccante lungo il tragitto. Sarebbe anche possibili offrire la possibilità di *non ripudio* nell'eventualità in cui si aggiungesse anche il certificato digitale ai dati trasmessi ma, poiché la dimensione del payload trasmissibile attraverso la rete LoRa è limitato, non sarà aggiunto il certificato ai dati, di conseguenza ci si limiterà alle proprietà fondamentali di *autenticazione* e *integrità dei dati*.

Al fine di generare la firma associata ai dati si utilizza una funzione di hash pubblica, che deve essere nota al destinatario al fine di ripetere la stessa operazione eseguita dal mittente, per ricavare l'impronta digitale del documento. Successivamente, tramite una funzione crittografica, si effettua la cifratura dell'hash crittografico utilizzando come chiave quella privata associata al certificato. Il prodotto di tale operazione sarà la firma, la quale potrà essere allegata ai dati in modo che il destinatario sia in grado di verificare l'autenticità e l'integrità dei dati trasmessi:

$$H = enc\left(S_{kpri}, hash(M)\right)$$

Poiché l'end-device viene univocamente identificato dal proprio *DevEUI*, a cui nel provider di riferimento vi si assocerà il certificato a chiave pubblica, neanche l'aggiunta della chiave pubblica è stata considerata ma si sfrutterà il *DevEUI* trasmesso al fine di ottenere la chiave pubblica associata all'end-device e quindi verificare la firma a destinazione. Questo permette di ridurre ulteriormente l'overhead di trasmissione introdotto dalla funzionalità di sicurezza descritta; inoltre, si rimanda al capitolo *1.8.1* al fine di valutare la differenza di prestazioni e di dimensioni dei dati inviati dall'end-device grazie a dei test effettuati in ambiente virtuale.

Nell'ambito IoT l'uso della crittografia asimmetrica sarebbe molto controproducente a causa della lunghezza della chiave, di almeno 2048 bit, causando un overhead di memoria, nonché di prestazioni, non trascurabile. La *crittografia a curve ellittiche (ECC)* offre lo stesso di livello di sicurezza di RSA, ma offrendo una dimensione della chiave ridotta di circa 1/10 e un'implementazione più efficiente, diventando di fatto lo standard moderno per l'implementazione della sicurezza e della privacy per l'emergente mondo IoT. La crittografia a curve ellittiche, come si evince dal nome, è basata sulle curve ellittiche, lavorando sulla superficie di una curva ellittica 2D e utilizzando un problema del logaritmo discreto molto più complesso rispetto a quello ad aritmetica modulare utilizzando per la crittografia asimmetrica, richiedendo per tale motivazioni chiavi di dimensione inferiore. La riduzione della chiave non offre solo un overhead inferiore ma permette anche di ottenere prestazioni in termini di velocità di esecuzione dell'algoritmo crittografico molto superiori, permettendo di risparmiare batteria nonché di ridurre la potenza di calcolo necessaria allo svolgimento dell'operazione.

Al fine di creare la firma digitale associata ai dati, si è deciso di utilizzare l'algoritmo **Elliptic Curve Digital Signature Algorithm (ECDSA)**, descritto nell'*RFC6979*, una variante per le curve ellittiche del noto algoritmo *DSA*. La lunghezza della coppia di chiavi scelta è di 256 bit, che equivale a una chiave RSA da 3072 bit, offrendo un livello di sicurezza adeguato e oggigiorno considerato sicuro.

Al fine di produrre il message digest è stato deciso di utilizzare *sha256*, ottenendo una sicurezza complessiva della firma da 128 bit, molto più che sufficiente per l'applicazione in campo IoT di sensori volti all'invio di dati di telemetria e GNSS.

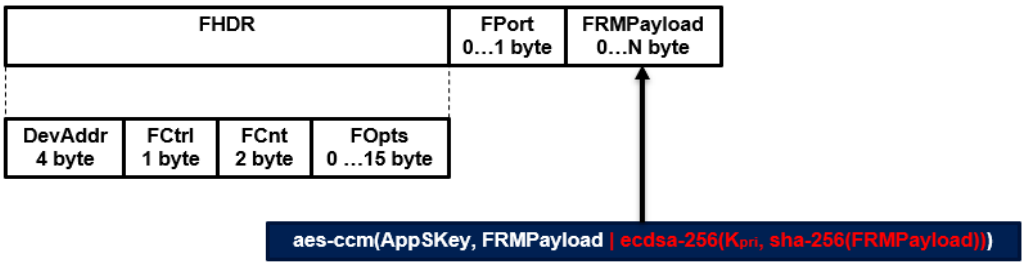


Figura 14: Formato dati applicativi proposto

La *figura 14* mostra la modifica al campo *FRMPayload* che verrà effettuata dall'end-device prima di spedire i dati raccolti:

$$FRMPayload = FRMPayload \mid ecdsa256(K_{pri}, sha256(FRMPayload))$$

La firma digitale può essere sfruttata pure per i messaggi di downlink, ovvero i messaggi provenienti dal server provider di riferimento e diretti verso l'end-device, in modo da assicurarsi che il messaggio non possa essere modificato da terzi alterando l'azione che debba essere eseguita sul dispositivo, come ad esempio l'accensione di un led. Naturalmente, la chiave da utilizzare dovrebbe essere quella del server e l'end-device dovrebbe avere il possesso della corrispettiva chiave pubblica.

Infine, la tipologia di certificati che sarà utilizzata per lo sviluppo della tesi è del tipo X.509 v3, per maggiori dettagli fare riferimento allo standard *rfc5280*.

1.7.2 Personal Security Environment (PSE)

Un PSE è un componente che consente di proteggere la chiave privata e i certificati delle CA fidate di un utente. Può essere implementato via software come un file protetto da passphrase o hardware il quale a sua volta si suddivide in:

5. **Passivo:** non dispone di capacità crittografiche autonome e di conseguenza, ogni qualvolta il materiale crittografico deve essere utilizzato, viene fornito al sistema.
6. **Attivo:** dispone di capacità crittografiche autonome, eseguendo qualunque operazione che richieda l'uso della chiave al suo interno.

La massima sicurezza ottenibile con tale soluzione è quella di installare un *PSE hardware attivo* sull'end-device in modo che le chiavi utilizzate per la cifratura dei dati e per l'integrità (*AppKey, AppSKey* e *NwkSKey*), nonché la chiave privata che sarà utilizzata per aggiungere l'integrità end-to-end, siano memorizzate e utilizzate esclusivamente all'interno di un ambiente protetto, in modo da resistere alla maggiorparte degli attacchi eseguibili sull'end-device, al fine di compromettere le chiavi descritti nel capitolo *1.6.3*. Questa soluzione aggiunge un ulteriore costo alla produzione del dispositivo, ma aumentandone la sicurezza e accelerandone tutte le operazioni crittografiche che il dispositivo deve compiere durante la procedura di join e di invio dei dati.

Un'ulteriore azione che potrebbe essere eseguita è l'aggiunta di integrità al firmware o ai software che vengono installati sul dispositivi, sempre tramite firma digitale. In questo modo si ha la sicurezza che eventuali patch o aggiornamenti installati, o anche l'immagine caricata al boot del sistema, siano validi, in arrivo da una fonte affidabile non alterati da un attaccante, andando così a proteggere il dispositivo da altre tipologia di attacchi attuabili.

1.7.3 Refresh del Contesto di Sessione

A causa della sicurezza ridotta delle chiavi, che decresce ulteriormente con il numero di utilizzi e l'arco temporale d'utilizzo, si consiglia di associare in fase di creazione delle chiavi, un campo *expiration* che segnala all'end-device fino a quando potrà utilizzarle. Inoltre, si necessiterà di effettuerà nuovamente la procedura di *join*, ovvero lo scambio di messaggi *join request* e *join accept* tra l'end-device e il network server, al fine di ottenere le nuove informazioni di contesto. Naturalmente, a seguito della nuova procedura di join, si andranno a resettare i contatori di uplink e downlink, poiché la loro unicità deve essere garantita esclusivamente per il contesto utilizzato, ovvero per una certa coppia di chiavi, e non globalmente. Il refresh delle chiavi di sessione non è implementato all'interno del *protocollo 1.0.2,* ma si consiglia fortemente di provvedere ad implementare tale modifica considerando, nel calcolo del nuovo campo *expiration*, la durata della batteria del dispositivo. Questo permetterà di prevedere un numero adeguato, ma non troppo grande, di *rejoin;* il rischio, altrimenti, è di generare la stessa coppia di *nonce*. Poiché generati da dispositivi diversi, la probabilità che un nonce si ripeta è indipendente dalla probabilità che l'altro nonce si ripeta. Quindi, se il calcolo è veramente casuale, per avere una buona possibilità di ripetizione si necessitano di circa 2^{20} tentativi (2^{12} per l'*AppNonce* e 2^{8} per il *DevNonce*).

1.8 Overview Architettura

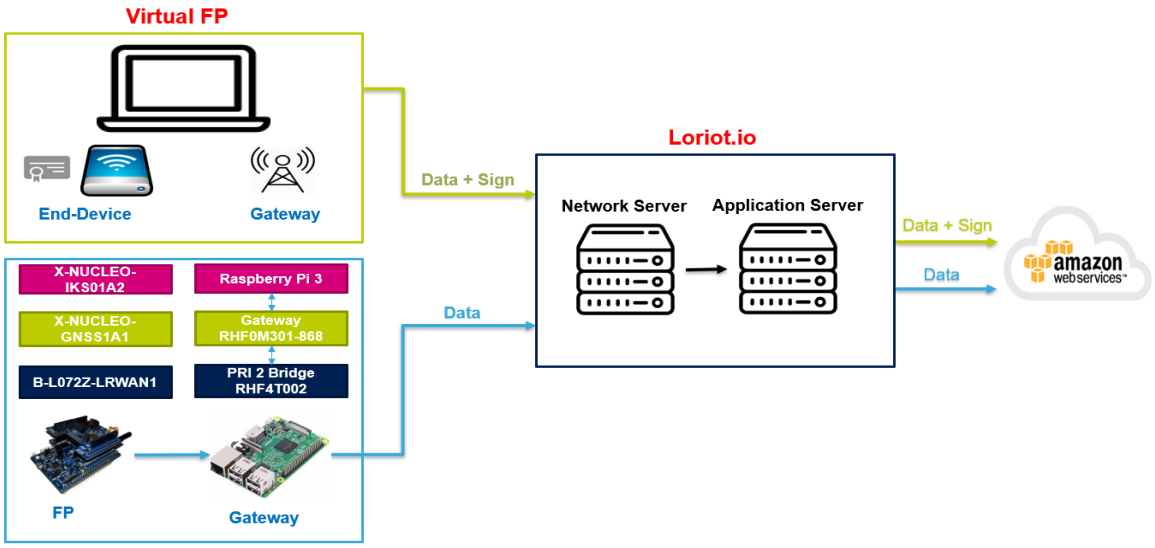


Figura 15: Overview Architettura

La *figura 15*, mostra una overview sull'architettura implementata con le considerazioni finora esposte. A causa del numero limitato di dispositivi a disposizione e delle mancanza delle funzionalità di sicurezza aggiuntive presenti nell'hardware a disposizione, è stato realizzato un ambiente virtuale che si andrà a contrapporre a quello fisico dato dal *FP* e da un gateway

realizzato con un *Raspberry Pi 3*, un modulo *Gateway RHF0M301-868* e un modulo *PRI 2 Bridge RHF4T002*.

L'architettura centrale, ovvero il *Network Server* e l'*Application Server*, sono stati realizzati sfruttando l'infrastruttura a lungo raggio offerta da *Loriot* (https://www.loriot.io) che permette, con un account gratuito, di associare un gateway al network server già preimpostato e di aggiungere fino a 10 dispositivi sull'Application Server. È, inoltre, possibile settare un connettore che permetta di inoltrare i dati ricevuti dagli end-device al provider di riferimento scelto. Tale parte sarà attenzionata in seguito, andando a specificare le politiche di sicurezza e le scelte progettuali che sono state eseguite. L'unica pecca è rappresentata dalla mancata possibilità di attivare il downlink per l'account *loriot* gratuito, comportando l'impossibilità nell'implementazione di tale feature durante lo svolgimento di questo lavoro.

L'ambiente virtuale è stato realizzato sfruttando le librerie *packet-forwarder* e *packet-forwarder-json-builder*. Il funzionamento dell'ambiente virtuale prevedere la realizzazione di un file di configurazione *deviceList* dove vengono aggiunte tutte le informazioni necessarie al funzionamento dei dispositivo simulati. Inoltre, vengono aggiunte le informazioni circa le chiavi

```
The virutal gateway will send packes for:
 - BE-7A-00-00-00-00-00-05
 - BE-7A-00-00-00-00-00-0A
N.B: The packets will be send each: 1m
-----------------------------------------------------

Sending data for BE-7A-00-00-00-00-00-05 ... done at 2019-5-21 13:43:27
Sending data for BE-7A-00-00-00-00-00-0A ... done at 2019-5-21 13:43:27
```

Figura 16: Esecuzione ambiente virtuale

private associate a questi dispositivi, in modo da poter produrre la firma da aggiungere ai dati. Il comportamento, ovvero la tipologia e il formato dei dati scambiati, è il medesimo del FP in modo da poter mettere a risalto esclusivamente le differenze in termini di sicurezza e funzionalità che possono essere aggiunti grazie agli studi precedentemente effettuati sul protocollo LoRaWAN e di cui si è già discusso ampiamente.

La *figura 16* mostra la scherma di avvio dello script che permette di eseguire l'ambiente virtuale: una peculiarità dello script è la possibilità di inserire la cadenza con cui i pacchetti devono essere inviati dall'end-device virtuale, che deve essere espressa in millisecondi.

1.8.1 Overhead

Al fine di valutare **l'overhead** e l'aumento di tempo d'invio dovuto all'aggiunta della firma digitale è stato effettuato uno studio su un campione di *10000 pacchetti* inviati, in modo da ottenere una buona stima. Lo studio è stato effettuato sull'ambiente virtuale e di conseguenza potrebbe non essere fedele con i risultati di tempo ottenibili riproducendo lo stesso comportamento sul dispositivo fisico, avente prestazioni nettamente inferiori rispetto al laptop utilizzato per il test.

La **dimensione dei dati** inviati dal FP è di 64 byte, mentre la firma ha una dimensione di 70 byte, per una dimensione totale del pacchetto dati di 134 byte, rientrando pienamente nella dimensione massima dei pacchetti dati che possono essere inoltrati tramite LoRaWAN. Inoltre, viene ottenuto un overhead di poco più del doppio del reale payload trasmesso che viene largamente ripagato dalle funzionalità di sicurezza che la trasmissione acquisisce.

Per quanto concerne la **velocità di invio,** è stato evidenziato un netto peggioramento dai $0{,}0082147\ ms$ nel caso di invio di dati sprovvisti di firma, ai $0{,}1740437\ ms$ nel caso di invio dei dati con firma. L'aumento dei tempi d'invio potrebbe essere inferiore se nel dispositivo fisico venisse implementato un *PSE* che permetta di ridurre le elaborazione crittografiche grazie ad un hardware specializzato, permettendo di aumentare i tempi di durata della batteria che, sennò, risulterebbero ridotti a causa dell'overhead temporale.

2.0 Amazon Web Services – AWS

Con il termine cloud computing si intende quel modello di business avente come obiettivo principale la fornitura di servizi IT, ad esempio: potenza di elaborazione, storage, applicazioni e altre risorse on demand. Indipendentemente dalla finalità d'uso, che sia l'esecuzione di applicazioni per la condivisione o il supporto di operazioni aziendali *mission critical*, una piattaforma cloud fornisce accesso rapido alle risorse IT flessibili ed a basso costo. Con il cloud computing, non è necessario effettuare grandi investimenti in infrastruttura hardware o dedicare molto tempo ad attività impegnative di gestione dell'hardware. Al contrario, è possibile effettuare il provisioning delle risorse di elaborazione in base ad esigenze specifiche. Il cloud computing consente di accedere alla quantità di risorse necessarie in modo quasi istantaneo, pagando solo in base all'uso effettivo [6].

In questo scenario, negli ultimi anni, sono nati decine di servizi cloud che prendono parte alla cosiddetta "*Industry 4.0*". Tra questi emerge **Amazon Web Services (AWS)**, una piattaforma che fornisce servizi di clouding che vengono distribuiti su più 20 regioni, con un numero sempre crescente e con un gran numero di prodotti differenti, oltre alla possibilità di selezionare il prodotto su misura in base alle esigenze specifiche del progetto che si sta realizzando. In questo modo viene dato notevole spazio alla customizzazione e alla costruzione mirata, diminuendo drasticamente i costi e la difficoltà di messa in piedi di un applicativo.

I servizi AWS utilizzati nella soluzione sono i seguenti:

- **DynamoDB:** servizio di database NoSQL che supporta modelli di dati di tipo documento e chiave-valore. Nel nostro caso è stato utilizzato per memorizzare le seguenti tabelle:
    - **Telemetry:** tabella utilizzata per la memorizzazione dei dati di telemetria ricevuti dagli end-device. Ogni oggetto possiede un attributo *TTL (Time To Live)*. La chiave di partizione primaria è l'identificativo univoco dell'utente all'interno del *Cognito Identity Pool*, mentre la chiave di ordinamento primaria è il *timestamp* espresso in millisecondi; campo che per motivi di sicurezza viene aggiunto dal server al momento di ricezione dei dati.
    - **GNSS:** tabella utilizzata per la memorizzazione dei dati GNSS ricevuti dagli end-device. Ogni oggetto possiede un attributo *TTL*. La chiave di partizione primaria è l'identificativo univoco dell'utente all'interno del *Cognito Identity Pool*, mentre la chiave di ordinamento primaria è il *timestamp* espresso in millisecondi; campo che per motivi di sicurezza viene aggiunto dal server al momento di ricezione dei dati.
    - **Thresholds:** tabella utilizzata per la memorizzazione delle soglie sui dati riportati dagli end-device; verrà utilizzata per la gestione delle notifiche in real-time (capitolo *2.3.2)*. La chiave di partizione primaria è il *DevEUI* dell'end-device, mentre la chiave di ordinamento primaria è il nome della threshold. Il *TTL* non è presente poiché il numero di oggetti che saranno inseriti all'interno della tabella è piccolo, dipendendo dal numero di end-device che gli utenti hanno registrato all'interno dell'applicativo, considerando che all'interno dell'account *loriot* è possibile memorizzare fino ad un massimo di 10 end-device se si dispone di un account gratuito.

- **API Gateway:** servizio completamente gestito che semplifica la creazione, la pubblicazione, la manutenzione, il monitoraggio e la protezione delle API su qualsiasi scala [7]. All'interno del lavoro è stato utilizzato al fine di creare le API REST che verranno esposte per la comunicazione con gli endpoint. Le API sono protette grazie all'aggiunto di uno strato di sicurezza fornito dall'API Gateway; la sua trattazione verrà approfondita nel *capitolo 2.4*.
- **AWS IoT:** AWS IoT Core è una piattaforma cloud gestita che consente a dispositivi connessi di interagire in modo semplice e sicuro con applicazioni nel cloud e altri dispositivi [8]. È il servizio principale che permette, grazie all'utilizzo di un *Message Broker MQTT*, l'interazione con il cloud *Loriot*, garantendo che i dati inviati dagli *end-device* siano elaborati e memorizzati all'interno del database. Inoltre, fornisce la memorizzazione degli end-device, l'apertura di topic MQTT e la creazione di regole ad hoc per gestire i vari eventi, come ad esempio la pubblicazione di nuovi dati all'interno di un topic MQTT da parte degli end-device. Nella terminologia AWS gli end-device registrati all'interno dell'IoT Core vengono indicati come *things*; la sua trattazione verrà approfondita nel *capitolo 2.3*.
- **Identity and Access Management (IAM):** consente di gestire in sicurezza l'accesso ai servizi e alle risorse AWS creando e gestendo utenti e utilizzando le autorizzazioni per verificare l'accesso alle risorse AWS [9]. Permette di assegnare ad ogni utente delle chiavi d'accesso che saranno utilizzare per la creazione del connettore *Loriot-AWS*; inoltre, permette di assegnare dei ruoli, a cui vengono associate delle policy, sia agli utenti che alle funzioni Lambda in modo da concedere l'autorizzazione necessaria ad eseguire esclusivamente le operazioni di cui necessitano. La sua trattazione verrà approfondita nel *capitolo 2.1*.
- **Simple Storage Service (S3):** servizio di storage utilizzato per la creazione di un *bucket* per ospitare il *Front End* sviluppato per la gestione della dashboard.
- **CloudWatch:** servizio di monitoraggio utilizzato per la creazione di file di logging al fine di visualizzare il funzionamento e l'esecuzione delle funzioni *Lambda* eseguite.
- **Cognito:** servizio per la gestione del processo di autenticazione degli utenti, il quale offre supporto sia per l'accesso web che per quello mobile. Offre diverse funzionalità d'accesso, quali Facebook, Google e Amazon. L'autenticazione è basata sulla password e lo username dell'utente al fine di ottenere un *token d'accesso*, della durata di un'ora, e un *token di refresh*, di una durata di 7 giorni, al fine di non ripetere l'intera procedura di login per ottenere nuovamente un *token d'accesso*; la sua trattazione verrà approfondita nel *capitolo 2.2*.
- **Lambda:** servizio che consente di eseguire funzioni che possono essere invocate da servizi AWS, come ad esempio dall'API Gateway, senza gestire il provisioning o un server. Inoltre, permettono di aggiungere *Layer* in cui è possibile caricare librerie esterne o funzionalità condivise tra più funzioni in modo da migliorare la leggibilità e la manutenibilità del codice.

Prima di analizzare la gestione dei dispositivi e il funzionamento delle API REST esposte, verranno discussi i seguenti topic: *gestione delle policy*, connettore *Loriot-AWA* e la gestiste della sicurezza e delle autorizzazioni fornite agli utenti.

2.1 Gestione della Sicurezza: IAM e IoT Policy

Come già accennato, IAM viene utilizzato per la gestione degli accessi e delle autorizzazioni alle risorse cloud. Nella soluzione sviluppata, questo servizio è stato utilizzato principalmente per realizzare data segregation per ogni utente applicativo. Le funzionalità principali sono gli **Utenti IAM**, che permettono di rappresentare gli utenti all'interno del contesto IAM, e i **Ruoli IAM**, i quali permettono di specificare le autorizzazioni che un *principal* (ad esempio un utente o una funzione Lambda) possiede. Ad entrambi, viene associata una **Policy**, ovvero un documento JSON costituito da vari elementi che permettono di specificare quali azioni possono essere eseguite e su quali risorse. Il vantaggio principale nell'utilizzo delle *policy* è la versatilità e l'indipendenza dal metodo che si sceglie per eseguire l'azione specificata: la policy verrà sempre verificata nel momento in cui un principal chiede di accedere alla risorsa a cui essa fa riferimento.

La *figura 17* mostra un esempio di policy, dove:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Elemento1",
      "Effect": "Allow",
      "Action": [
        "iot:UpdateThingShadow",
        "iot:GetThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "arn:aws:iot:*:*:thing/BE7A000000000005",
        "arn:aws:iot:*:*:thing/BE7A000000000000A"
      ]
    }
  ]
}
```

Figura 17: Policy di Esempio

- **Version**: indica la versione del linguaggio utilizzato per la scrittura della policy.
- **Statement**: è un array di elementi di policy ove ogni elemento ha i seguenti attributi:
  - **Sid**: campo opzionale che identifica univocamente l'elemento di policy.
  - **Effect**: indica se l'elemento di policy deve permettere o negare le azioni che seguono sulle risorse specificate.
  - **Action**: array di azioni che devono essere permesse o negate.
  - **Resource**: array di risorse su cui l'azione deve essere eseguita.
  - **Condition Block**: campo opzionale che indica in quale circostanza l'elemento di policy deve essere considerato.

2.1.1 Utenti IAM

A seguito della registrazione di un utente viene creato un **utente IAM** a cui vengono associate una coppia di chiavi necessarie alla creazione del connettore *Loriot-AWS,* e una *policy* che permette di specificare a quali *thing* e a quali topic MQTT l'utente ha autorizzazione a collegarsi, ottenere informazioni e pubblicare nuove informazioni. Il connettore abilita l'*Application Server* ad inoltrare i dati ricevuti dal *Network Server*, e rispettivamente dagli end-device, ad un *Broker MQTT* che, in base al *DevEUI* dell'end-device, provvederà a pubblicare i dati ricevuti nell'opportuno topic MQTT che sarà rappresentato dallo **Shadow** della *thing*.

Uno *Shadow* è un documento JSON che viene utilizzato per memorizzare le informazioni sullo stato corrente dell'end-device. Il documento viene univocamente identificato tramite un identificativo che corrisponde al nome della thing, e quindi al *DevEUI*, a cui viene associato;

questo permette di inoltrare, lato *Loriot*, le informazioni ricevute al topic MQTT corretto in modo del tutto automatico grazie al ruolo svolto dal broker MQTT. Lo shadow viene suddiviso in due parti ben distinte:

- **Desired:** contiene lo stato desiderato per il device. Viene attualmente sfruttato per inserire informazioni di contesto che saranno utilizzate dalla regola associata ai topic MQTT per elaborare le informazioni ricevute dagli end-device. Potrebbe essere utilizzato dall'end-device, in fase di boot, per prelevare le informazioni relative alla sua configurazione iniziale. Le informazioni contenute all'interno del *desired* vengono settate dall'utente mentre l'end-device ha esclusivamente il compito di leggerle e interpretarle.
- **Reported:** contiene lo stato corrente inviato dal device sulla rete durante l'ultima rilevazione effettuata, cioè: valore di temperatura, pressione, umidità e posizione. Questi vengono elaborati da una regola in ascolto su tutti gli shadow delle thing associate ai device. Le informazioni contenute all'interno di tale sezione sono relativamente inserite dagli end-device e lette dal backend al fine di elaborare i dati, l'utente non dovrebbe modificarne il contenuto.

Autorizzazioni associate agli Utenti

Le autorizzazioni relative agli utenti sono state gestite tramite il meccanismo delle policy. Ogni utente deve poter accedere esclusivamente agli end-device che sono stati registrati tramite il proprio account; a seguire sarà analizzata la policy associata ad ogni utente in modo da comprendere come il suo ciclo di vita possa gestire l'intero meccanismo degli accessi e delle relative autorizzazioni, sfruttando interamente le potenzialità offerte da AWS.

Figura 18: Policy associata agli Utenti

La *figura 18* mostra la policy che viene associata ad ogni *IAM user* a seguito della sua creazione. Lo statement non numerato permette di effettuare l'operazione $iot{:}\,Connect$, ovvero l'autorizzazione a connettersi al broker di messaggi MQTT di AWS IoT con qualunque connection ID.

A differenza, i rimanenti elementi applicano autorizzazioni fine-grained. Lo statement *(1)* definisce le autorizzazioni relative alla possibilità di modificare, ottenere e cancellare le informazioni contenute all'interno dello shadow di una o più thing specificate all'interno dell'array delle risorse dell'elemento. Lo statement *(2)* definisce le autorizzazioni relative alla pubblicazione e la ricezione di informazioni per uno o più topic MQTT descritte nell'array di

risorse dell'elemento. In ultimo, lo statement *(3)* definisce le autorizzazioni relative alla sottoscrizione ad uno o più topic MQTT specificati all'interno dell'array delle risorse dell'elemento. In sostanza la policy appena descritta nega di default ad un qualsiasi utente privo di device registrati l'accesso a tutte le risorse IoT Core.

Alla **registrazione** di un device, da parte di un utente, gli statement numerati verranno automaticamente modificati al fine di autorizzare l'utente ad accedere alla risorsa appena creata:

- Per quanto concerne (1) le risorse aggiunte assumeranno il seguente formato:

  $$arn:aws:iot:region:id-account:thing/\boldsymbol{DevEUI}$$

  Thing rappresenta l'insieme di tutti gli oggetti, ovvero gli end-device, che sono contenuti all'interno dell'account identificato dall'*id-account*, nella regione specificata da *region*. Il *DevEUI* permette di effettuare un filtering e di agire esclusivamente su una specifica risorsa, ovvero l'end-device creato.

- Per quanto concerne (2) le risorse aggiunte assumeranno il seguente formato:

  $$arn:aws:iot:region:id-account:topic/\boldsymbol{DevEUI}/*$$

- Per quanto concerne (3) le risorse aggiunte assumeranno il seguente formato:

  $$arn:aws:iot:region:id-account:topicfilter/\boldsymbol{DevEUI}/*$$

  Topic rappresenta il path di uno specifico topic MQTT e abilita la possibilità di ricevere aggiornamenti e di pubblicare un determinato payload all'interno del topic specificato, mentre *topicfilter* rappresenta un filtro di topic MQTT e permette di effettuare la subscribe al path specificato, che ha come root l'identificativo dell'end-device creato, seguito dal simbolo "*", per permettere le azioni specificate su qualsiasi subpath dopo specificato. I valori di *region* e *id-account* saranno sempre valorizzati con il simbolo "*" al fine di mantenere la portabilità della policy tra account e regioni differenti di AWS; mentre il *DevEUI* sarà l'identificativo univoco dell'end-device per cui si è effettuata la registrazione.

Nel caso di **cancellazioni** di device si effettuerà un'operazione di modifica opposta a quelle descritte precedentemente: si procederà a rimuovere dai rispettivi array di risorse l'elemento che rappresenta l'end-device che si sta provvedendo a cancellare. Nel caso in cui l'array contenga esclusivamente quell'elemento, si procederà a ripristinare anche l'effetto degli elementi di policy che passeranno da *Allow* a *Deny*, ritornando ad assumere la stessa struttura e contenuto della policy descritta nella *figura 18*.

Come detto, la policy descritta permette di effettuare un controllo fine-grained sulle autorizzazioni di accesso allo shadow e ai topic relativi agli end-device, dando autorizzazione ad effettuare le azioni descritte esclusivamente sui device che sono stati registrati con un determinato account utente. Dal punto di vista della sicurezza, ciò evita eventuali modifiche apportate da un attaccante agli shadow o ai topic MQTT di end-device di altri utenti. Questo potrebbe interessante semplicemente la lettura delle informazioni di stato degli end-device di altri utenti. Inoltre, un altro dei vantaggi principali è la possibilità di non dover scrivere alcuna

linea di codice per implementare la gestione delle autorizzazioni, essendo direttamente integrate nell'ambiente utilizzato per lo sviluppo del backend, ovvero AWS. Questo, come già discusso precedentemente, offre la possibilità di gestione dell'autorizzazione indipendentemente dal metodo scelto dall'utente per effettuare una delle azioni discusse, nonché una maggiore velocità nella messa in piedi dell'applicativo finale.

La possibilità di non dover implementare del codice ausiliario per la gestione delle autorizzazioni, dal punto di vista della sicurezza, offre un'ulteriore vantaggio, ovvero quello di ridurre al minimo la superficie d'attacco lasciata ad un possibile utente malintenzionato: la superficie d'attacco dipende dagli strati software sviluppati, che nel nostro caso sarà di molto inferiore poiché basata esclusivamente sulle policy, riducendo non di poco i possibili bachi che possono essere introdotti tramite la scrittura di codice non correttamente testato.

2.1.2 Ruoli IAM

Le funzioni Lambda vengono eseguite per gestire le richieste provenienti da un endpoint HTTP REST, e, così come gli utenti Cognito necessitano di ottenere le autorizzazioni per effettuare una determinata azione su una risorsa, anche le funzioni necessitano di autorizzazioni per accedere alle risorse cloud. Al tal fine si è sfruttato il meccanismo dei *Ruoli IAM* per la gestione delle autorizzazioni. Questo perchè i permessi associati devono essere il più stringenti possibili per adempiere al principio del privilegio minimo (*least privilege*) secondo il quale un'entità deve avere accesso esclusivamente alle informazioni e alle risorse che sono strettamente necessarie al raggiungimento dello scopo della loro esecuzione.

Non entreremo nel dettaglio delle policy che vengono associate ai *Ruoli IAM* per la gestione delle autorizzazioni delle funzioni Lambda poiché contengono esclusivamente l'autorizzazione nell'effettuare tutte quelle azioni che vengono invocate durante l'esecuzione della funzione, che di conseguenza dipendono dal compito eseguito, ma ci limiteremo ad analizzare la policy che viene associata ad un utente che ha effettuato la procedura di *autenticazione*.

Al fine di mantenere una generalità più ampia possibile, è stato creato un singolo ruolo che viene associato ad ogni utente che ha correttamente portato al termine la procedura di autenticazione. L'associazione del ruolo viene effettuata in automatico dal *Cognito Identity Pool* a seguito della richiesta di credenziali temporanee di sessione da parte dell'utente al fine di poter controllare, ad ogni sua richiesta effettuata, le autorizzazioni di cui dispone prima di erogare il servizio richiesto. In questa sezione non si approfondirà il protocollo di autenticazione, si rimanda dunque al capitolo *2.2.2* per una trattazione completa.

Nei successivi capitoli sarà analizzata la policy che viene associata al ruolo IAM connesso all'utente al termine della fase di autenticazione.

Elementi per i topic MQTT

La *figura 19* mostra gli elementi di policy che permettono di specificare le autorizzazioni relative ai topic MQTT, abilitando l'utente ad effettuare la connessione (*Connect*) esclusivamente con un client MQTT avente come identificativo il *Cognito Identity ID (*da ora definito come *Identity)*, ovvero l'identificativo univoco assegnato all'utente dal *Cognito Identity Pool* durante la prima autenticazione. In maniera analoga, permette di effettuare la sottoscrizione (*Subscribe*), la

```
{
    "Effect": "Allow",
    "Action": [
        "iot:Connect"
    ],
    "Resource": [
        "arn:aws:iot:*:*:client/${cognito-identity.amazonaws.com:sub}"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "iot:Receive",
        "iot:Publish"
    ],
    "Resource": [
        "arn:aws:iot:*:*:topic/${cognito-identity.amazonaws.com:sub}/*"
    ]
},
```

```
{
    "Effect": "Allow",
    "Action": [
        "iot:Subscribe"
    ],
    "Resource": [
        "arn:aws:iot:*:*:topicfilter/${cognito-identity.amazonaws.com:sub}/*"
    ]
},
```

Figura 19: Elementi di policy per le autorizzazioni sui topic MQTT

pubblicazione (*Publish*) e la ricezione (*Receive*), esclusivamente da topic aventi come path base l'*Identity* associata all'utente:

$$\$\{cognito - identity.amazonaws.com{:}sub\}/*$$

Le successivi componenti del path possono essere scelte arbitrariamente, in modo da fornire un'ampia generalità ma una segregazione completa dei topic MQTT accessibili dagli utenti. In particolare, i topic MQTT saranno utilizzati per le ricezione di notifiche che indicano il superamento, in una rilevazione inviata dall'end-device, di una delle soglie settate dall'utente sulle misura che l'end-device può catturare. È stato sfruttato il meccanismo dei topic MQTT offerto da AWS al fine di permettere una ricezione delle notifiche in real-time, rendendo possibile un tempestivo intervento dall'utente nel caso in cui lo ritenesse opportuno.

```
{
    "Effect": "Allow",
    "Action": [
        "execute-api:Invoke"
    ],
    "Resource": [
        "arn:aws:execute-api:*:*:*/*/GET/users/${cognito-identity.amazonaws.com:sub}/devices",
        "arn:aws:execute-api:*:*:*/*/POST/users/${cognito-identity.amazonaws.com:sub}/devices",
        "arn:aws:execute-api:*:*:*/*/GET/users/${cognito-identity.amazonaws.com:sub}/devices/*",
        "arn:aws:execute-api:*:*:*/*/PUT/users/${cognito-identity.amazonaws.com:sub}/devices/*",
        "arn:aws:execute-api:*:*:*/*/DELETE/users/${cognito-identity.amazonaws.com:sub}/devices/*",
        "arn:aws:execute-api:*:*:*/*/GET/users/${cognito-identity.amazonaws.com:sub}/telemetry",
        "arn:aws:execute-api:*:*:*/*/PUT/users/${cognito-identity.amazonaws.com:sub}/thresholds/*",
        "arn:aws:execute-api:*:*:*/*/GET/users/${cognito-identity.amazonaws.com:sub}/gnss"
    ]
},
```

Figura 20: Elemento di Policy per l'API Gateway

Elemento per l'accesso all'API Gateway

La *figura 20* mostra l'elemento di policy designato per la gestione dell'autorizzazione relativa all'accesso alle API REST esposte dall'API Gateway. Ogni utente può contattare esclusivamente le API che sono state inserite all'interno dell'array di risorse dell'elemento di policy mostrato ed in particolare, come nel caso precedente, la presenza dell'elemento $\$\{cognito - identity.amazonaws.com{:}sub\}$, che verrà sostituito in fase di analisi della policy da parte dell'ambiente di esecuzione, permette di effettuare una prima segregazione delle informazioni relative all'utente. L'accesso alle API REST sarà pilotato dall'*Identity* associato alla richieste

inviate da un utente e che, per l'appunto, descrive l'identità dell'utente che sta effettuando la richiesta. Questo significa che un utente avente come *Identity* il seguente:

$$us - west - 2{:}\,0b0c5b73 - a63d - 4887 - beb1 - fe22a89cc0c0$$

Potrà accedere esclusivamente a tutte le API REST che hanno il seguente formato:

$$users/us - west - 2{:}\,0b0c5b73 - a63d - 4887 - beb1 - fe22a89cc0c0/*$$

Inoltre, viene anche specificato il metodo HTTP con cui è possibile contattare una determinata API REST, in modo da offrire un controllo completo sulle tipologia di richiesta che può essere effettuata da un utente all'API Gateway. Nonostante ciò, non vi è nessun controllo di autorizzazione che permetta di concedere all'utente un accesso esclusivamente ai propri *end-device*. Tali informazioni non sono state aggiunte all'interno della policy così da creare una singola policy che potesse essere valida per tutti gli utenti registrati e non specifica in base all'utente. Questa scelta è stata presa sia poiché il numero di policy che possono essere create all'interno del servizio IAM risulta fortemente limitato, sia perché in tal modo è possibile ottenere una migliore manutenibilità delle autorizzazioni relative agli utenti. Il controllo sull'accesso all'end-device sarà sviluppato all'interno delle funzioni Lambda che verranno richiamate a seguito della chiamata dell'API REST opportuna, sfruttando l'associazione che viene creata in fase di registrazione di un end-device tra *Identity* e *Thing*; tale associazione con i rispettivi vantaggi, sia in termini di risorse che di prestazioni, sarà discussa successivamente nel *capitolo 2.3.1*.

Elemento per l'accesso a DynamoDB

La *figura 21* mostra l'elemento di policy designato per la gestione dell'autorizzazione relativa all'accesso alle tabelle presenti all'interno del database *DynamoDB*. Come sarà discusso nel *capitolo 2.4*, al fine di migliorare le performance e ridurre i tempi di elaborazione di una richiesta

```
{
    "Effect": "Allow",
    "Action": [
        "dynamodb:Query"
    ],
    "Resource": [
        "arn:aws:dynamodb:*:*:table/Telemetry",
        "arn:aws:dynamodb:*:*:table/GNSS"
    ],
    "Condition": {
        "ForAllValues:StringEquals": {
            "dynamodb:LeadingKeys": [
                "${cognito-identity.amazonaws.com:sub}"
            ]
        }
    }
}
```

Figura 21: Elemento di Policy per DynamoDB

proveniente dall'endpoint, dando maggior peso alle richieste più onerose e comuni, si dà la possibilità di contattare direttamente alcuni specifici servizi come DynamoDB. Ciò implica la possibilità da parte dell'utente di manipolare, cancellare o semplicemente leggere tutti i dati contenuti all'interno della specifica tabella se non vengono effettuate delle azioni di filtering dei dati tramite l'esecuzione di una funzione. Al fine di ovviare questo tragico scenario, è stato creato un elemento di policy opportuno che permetta di assegnare le autorizzazioni d'accesso a livello di riga e non solo a livello di tabella. Come mostrato nella *figura 21*, viene concesso all'utente l'autorizzazione di effettuare un'unica azione, ovvero *Query*, che abilita esclusivamente il

retrieve delle informazioni contenute all'interno della tabella escludendo la possibilità di manipolazione dei dati. La possibilità di effettuare tale azione è concessa solo sulle tabelle *Telemetry* e *GNSS*. A differenza dei casi precedenti, è presente il campo *Condition* che assume un ruolo cruciale per la gestione delle autorizzazioni al fine di permettere all'utente di accedere esclusivamente ai propri dati e non a quelli di altri utenti: la condizione permette di poter svolgere le azioni sulle risorse specificate solamente nel caso in cui la riga che si vuole leggere dalla tabella abbia come chiave di partizionamento principale un valore che coincide con l'*Identity* associata all'utente che sta effettuando la richiesta.

L'implementazione di quest'ultimo elemento di policy permette di aggiungere una **segregazione a livello di row** per tutte le tabelle specificate nella policy come risorse, invalidando qualsiasi tentativo di un utente malintenzionato di leggere dei dati relativi agli end-device di un altro utente. Ovviamente, per qualsiasi tentativo da parte dell'utente di accedere a servizi o dati che non rientrino tra quelli permessi dalla policy ad esso assegnata, la risposta da parte del server sarà *403 Forbidden*, impedendo a monte l'esecuzione della funzione Lambda e/o del servizio richiesto.

2.1.3 IoT Policy

Un ultimo tassello che riguarda le policy utilizzate all'interno di AWS sono le *IoT Policy*. Le **IoT Policy** sono documenti JSON che seguono la stessa convenzione delle *IAM Policy*, e con lo scopo di permettere o negare la connessione con il *message broker AWS IoT*, ricevere o inviare messaggi MQTT e leggere o modificare lo shadow associato ad uno specifico end-device. Tale policy risulta di fondamentale importanza per permettere l'apertura dei canali di comunicazione MQTT per la ricezione delle notifiche. Al fine di abilitarne l'utilizzo, in fase di registrazione dell'utente viene associata all'*Identity* generato dal *Cognito Identity Pool*. La policy mantiene un aspetto generico (*figura 22*) specificando esclusivamente le azioni che l'utente è autorizzato ad effettuare ma non specificando le risorse, ove è presente il simbolo "*" che permette di indicare

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect",
        "iot:Subscribe",
        "iot:Receive",
        "iot:Publish"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

Figura 22: IoT Policy

tutte le risorse. Questa generalizzazione è resa possibile in modo da utilizzare esclusivamente un'unica *IoT* Policy, grazie all'affiancamento delle policy IAM associate ai ruoli e agli utenti IAM. In tal contesto le due policy saranno confrontate dall'ambiente di esecuzione *runtime* e verrà presa in considerazione esclusivamente la policy più stringente tra le due in termini di risorse, abilitando la possibilità d'accesso all'IoT Core. Si noti che nel caso in cui la IoT Policy non venga

dichiarata e associata non si avrà l'autorizzazione, indipendentemente dalla presente della IAM policy, alle funzionalità IoT Core specificate nell'array *Action*.

2.2 Autenticazione e Registrazione degli Utenti

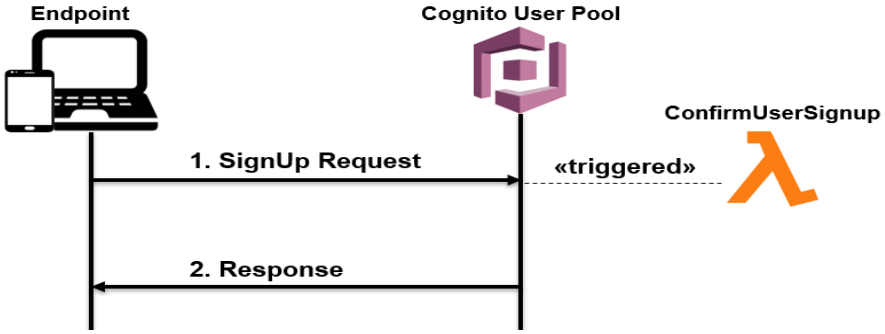
In questa sezione andremo ad analizzare il processo di **registrazione** dell'utente, analizzando passo dopo passo tutte le operazioni che vengono effettuate lato backend, e il processo di **autenticazione** dell'utente soffermandoci sulla potenzialità dei *token di refresh* e di *sessione* offerti da *Cognito* per la gestione dell'autenticazione degli utenti, sia mobile che desktop.

2.2.1 Registrazione dell'Utente

Il processo di registrazione dell'utente è suddiviso in due fasi:

1. Interazione con il *Cognito User Pool* per l'effettiva registrazione dell'utente all'interno del pool di utenti con la relativa memorizzazione delle informazioni relative all'utente.
2. Interazione con una funzione *Lambda* al fine di completare gli ultimi step delle registrazione, quali la creazione dell'utente IAM e delle relative chiavi d'accesso necessarie alla realizzazione del connettore *Loriot-AWS* e l'assegnazione della IoT Policy all'*Identity*.

Interazione con il Cognito User Pool



La *figura 23* mostra il primo step del processo di registrazione basato sull'interazione con il *Cognito User Pool*. Il *messaggio (1)* da inizio al processo di registrazione dell'utente tramite l'invio, da parte dell'endpoint, delle informazioni necessarie al processo di registrazione ovvero lo *username* che l'utente intende assumere e la *password* che sarà utilizzata per i tentativi di autenticazione al fine di effettuare l'accesso alle funzionalità offerte dal backend. Non sono state previste ulteriori informazioni, come ad esempio la mail, al fine di mantenere il più alto possibile l'anonimato dell'utente. Non essendo previsto l'ottenimento di informazioni utili per il processo di conferma di registrazione, come la mail o il numero di telefono, è stato associato un *trigger* al processo di registrazione dell'utente che prende il nome di *PreRegistrazione*, il quale permette di invocare una funzione Lambda, ovvero la *ConfirmUserSignup*, che permette di effettuare una convalida personalizzata, accettando la richiesta di registrazione senza l'utilizzo di una mail di conferma né di una conferma tramite MFA. Il *messaggio (2)* contiene l'esito della registrazione; in caso di successo si procederà ad eseguire la seconda fase di registrazione.

Interazione con la funzione Lambda

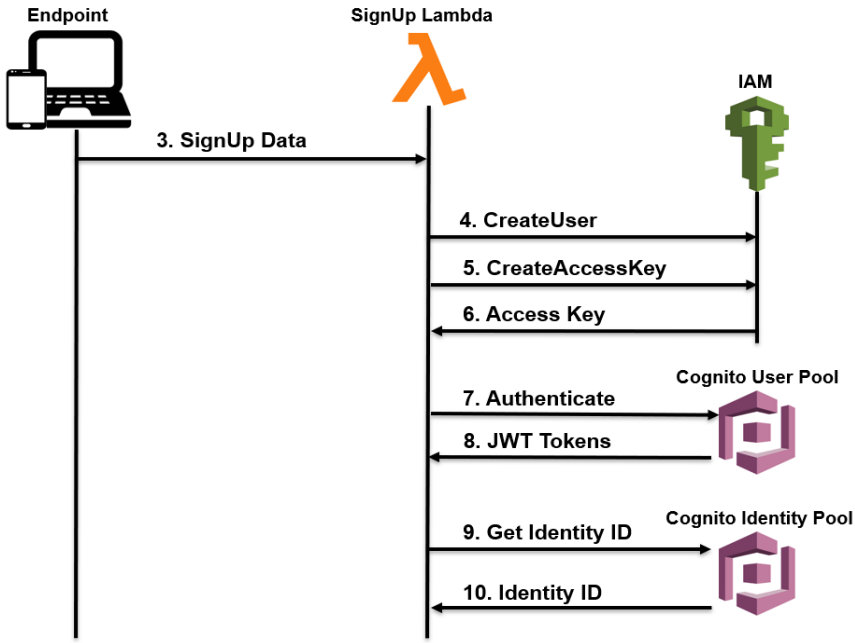
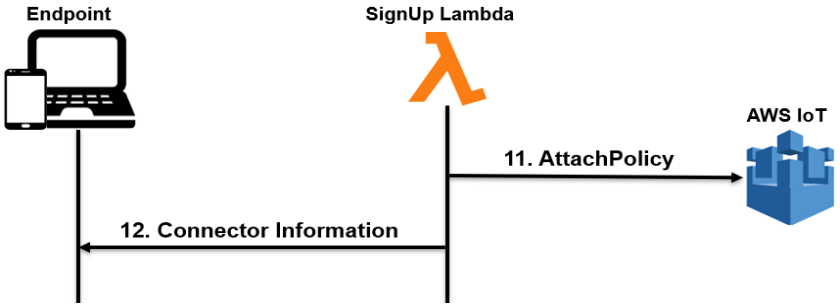


Figura 23: Flow di Registrazione 2/3

La *figura 24* mostra la prima parte del secondo step di registrazione in cui si effettua la chiamata della funzione Lambda passando lo *username* e la *password* utilizzati in fase di registrazione, *messaggio (3)*. La prima operazione effettuata dalla funzione Lambda è l'interazione con il servizio IAM al fine di effettuare la creazione di un *Utente IAM*, operazione rappresentata dal *messaggio (4)* Qui viene specificato che lo *username* che l'*Utente IAM* deve assumere sarà composto da un prefisso rappresentato dallo *username* passato dall'utente e dal suffisso "*LoRaWAN*". Questa modalità è stata scelta in modo da poter distinguere le tipologie di utenti IAM in base al cloud che gli end-device utilizzeranno per inviare i dati al cloud AWS; ciò permette di poter effettuare un'assegnazione delle autorizzazioni differenziata in base alla tipologia di cloud utilizzato dagli end-device per comunicare, effettuando anche operazioni differenti in fase di registrazioni nell'eventualità in cui sia necessario. Successivamente, tramite il *messaggio (5)*, si procede alla generazione delle chiavi passando come parametro il nome dell'Utente IAM creato al passo precedente; in caso di successo si otterranno in risposta, *messaggio (6)*, le chiavi d'accesso create ed associate all'account IAM, ovvero la *AccessKeyID* e la *SecretAccessKey*.

Terminata la fase di creazione dell'utente IAM si procederà, con i *messaggi* dal *(7)* al *(10)*, alla creazione dell'*Identity* associata al nuovo utente simulando parte del processo di autenticazione dell'utente. Questa operazione risulta necessaria al fine di collegare la *IoT Policy* al nuovo utente



una sola volta e senza dover verificare se essa sia già stata collegata all'*Identity* o meno, andando a ridurre i tempi di esecuzione di specifiche azioni che potranno essere eseguite successivamente dall'utente. Tali messaggi non saranno approfonditi in questa sezione ma nella successiva (capitolo *2.2.2)*, che è interamente dedicata al processo di autenticazione dell'utente. In caso di successo di tutti i passaggi, con il *messaggio (10)* si avrà a disposizione l'*Identity* associato all'utente all'interno del *Cognito Identity Pool* e si potrà procedere con la seconda ed ultima fase del secondo step di registrazione.

La *figura 25* mostra l'ultimo step del processo di registrazione in cui viene effettuata l'operazione finale tramite il *messaggio (11)* con il quale si associa all'*Identity* la *IoT Policy*. Infine, con il *messaggio (12)* si ritornano all'utente tutte le informazioni necessarie alla creazione del connettore *Loriot-AWS*, ovvero le due chiavi d'accesso ottenute tramite il *messaggio (6)*, la *AWS Region* e l'*EndpointRandomString* che dipendono dalla regione da cui si stanno sfruttando i servizi AWS. Tali informazioni saranno aggiunte all'interno dell'account *Loriot* dell'utente per realizzare il connettore. Al fine di aumentare la user friendly, il nome dei parametri ritornati è uguale a quello richiesto dal cloud per la creazione del connettore. È stata, inoltre, sviluppata una guida step by step da consultare così da poter configurare il tutto con facilità e nel minor tempo possibile, anche nel caso in cui l'utente non abbia familiarità con queste tecnologie.

2.2.2 Autenticazione dell'Utente

Il processo di autenticazione dell'utente è basato sull'interazione con il *Cognito User Pool (CUP)* al fine di ottenere un tripletta di token e successivamente con il *Cognito Identity Pool (CIP)* al fine di creare identità univoche per gli utenti e federarli con i provider di identità scelti. Il *CIP* risulta fondamentale al fine di ottenere le credenziali temporanee ed associare ad esse dei privilegi limitati al fine di accedere ai servizi offerti da AWS.

Interazione con il Cognito User Pool

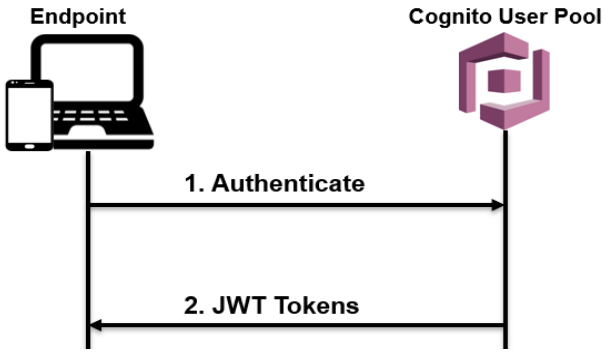


Figura 24: Autenticazione con il Cognito User Pool

La *figura 26* mostra la prima fase del processo di autenticazione, svolta tramite l'interazione con il *CUP*. Il *messaggio (1)* contiene la scelta del flusso di autenticazione da utilizzare, che nel nostro caso è "*USER_PASSWORD_AUTH*", e i dati di autenticazione richiesti dal flusso scelto, che nel caso specifico sono relativi allo *username* e la *password* utilizzati dall'utente in fase di registrazione. Nel caso in cui il *messaggio (1)* venga elaborato correttamente e porti ad un'autenticazione corretta, il *messaggio (2)* conterrà la seguente tripletta di token:

- **Refresh Token:** utilizzato per richiedere il refresh dell'ID Token in modo da evitare di dover ripetere l'intera operazione di autenticazione dell'utente. Questo permette di

ridurre i tempi di esecuzione e di non dover più inoltrare la password dell'utente per le successive autenticazioni fino alla espirazione del *refresh token* che ha una durata di 7 giorni.

- **Access Token:** utilizzato per ottenere e/o aggiornare le informazioni dell'account dell'utente con durata di un'ora. A causa delle informazioni ridotte contenute all'interno dell'account dell'utente tale token non è stato sfruttato per la realizzazione dell'applicativo ma è stato comunque elaborato ed inviato all'endpoint al termine della procedura di autenticazione. La tipologia del token è JWT.
- **ID Token:** utilizzato per ottenere le credenziali temporanee di sessione, ha una durata di un'ora. Tale token avrà un ruolo fondamentale all'interno della soluzione poiché necessario per la successiva fase di autenticazione dell'utente. Inoltre, contiene tutte le informazioni necessarie ad identificare l'utente come ad esempio il ruolo che deve essere utilizzato per sostituire quello standard assegnato a tutti gli utenti di default nell'eventualità in cui non venga associato prima di richiedere le credenziali temporanee. Questo lascia spazio per la possibilità di condivisione di end-device tra più account, ad esempio account aziendali. La tipologia di token è JWT.

JWT è un mezzo compatto e URL-safe per rappresentare le *claim* da trasferire tra due parti. Le *claim* in un JWT sono codificate come un oggetto JSON che viene utilizzato come payload di una struttura JSON Web Signature (JWS) o come testo in chiaro di una struttura JSON Web Encryption (JWE), consentendo alle *claim* di essere firmate digitalmente o protette da integrità con un codice di autenticazione dei messaggi (MAC) e/o crittografato [10]. L'idea alla base del JWT è che dopo l'autenticazione dell'utente, il server prepara un token al cui interno racchiuderà un payload in cui vengono aggiunte tutte le informazioni necessarie alla gestione dell'utente. Oltre al payload, viene aggiunta un'ulteriore informazione che è la modalità di cifratura del payload con la chiave del server in codifica *hash 256*, in modo che il client possa leggere il payload contenuto all'interno del token ma mai modificarlo, poiché in tal caso si avrebbe l'invalidazione del token stesso, ovvero il server sarebbe in grado di rilevare la modifica, scartando il token poiché considerato invalido. Il client allegherà il token ad ogni richiesta in modo che il server possa ottenere tutte le informazioni necessarie all'autenticazione dell'utente senza interrogare nessun database, permettendo di ottenere un'ottima scalabilità.

Visto il ruolo cardine che ricoprirà l'*ID token* nella fase successiva di autenticazione, andremo ad analizzarne la struttura e i campi fondamentali che lo compongono. A tal proposito si farà riferimento all'esempio contenuto all'interno della *figura 27* in cui viene mostrata la decodifica del payload di un *ID token*:


```
{
  "sub": "e54c5081-9f2e-47de-b6cc-fbc181135854",
  "aud": "7cq5ip8uk60ku9g5ubdn5khpec",
  "event_id": "0b4fed09-78bd-11e9-9b7c-6f33a367d0e8",
  "token_use": "id",
  "auth_time": 1558108894,
  "iss": "https://cognito-idp.us-west-2.amazonaws.com/us-west-2_OAIaYm5rR",
  "cognito:username": "Giuliano",
  "exp": 1558618188,
  "iat": 1558614588
}
```


Figura 25: ID Token Payload

- **sub (Subject):** a chi si riferisce il token, è un identificativo univoco dell'utente.

- **aud (Audience):** a chi o a cosa si riferisce il token, ovvero il *client_id* dell'utente autenticato.
- **event_id:** una stringa che identifica uno specifico evento o una modifica dello stato a cui questo evento è correlato.
- **token_use:** scopo del token, in questo caso assumerà sempre il valore "id". Viene utilizzato per distinguere a quale tipologia di token afferisce il payload, avendo AWS le tre tipologie precedentemente descritte.
- **auth_time:** timestamp dell'istante in cui è avvenuta l'autenticazione.
- **cognito:username:** nome dell'utente all'interno del CUP. Il prefisso $cognito$: afferisce ad eventuali attributi, non standard, aggiunti dal CUP poiché associati all'utente. Viene utilizzato anche per passare un eventuale ruolo associato agli utenti di un gruppo cognito, tecnica che viene utilizzata per la condivisione delle informazioni tra più account.
- **iss (Issuer):** indica chi ha emesso e firmato il token. Tramite il *CIP* è possibile effettuare l'autenticazione con differenti provider che saranno riconosciuti tramite il campo *iss*; nell'applicazione sarà sempre utilizzato esclusivamente AWS come provider d'autenticazione; in successive implementazioni potrebbe essere fornita l'autenticazione anche tramite *Facebook* e *Google*.
- **exp (Expiration Time):** tempo fino al quale il token ha validità. Tale campo viene sempre analizzato al fine di verificare la validità del token confrontandolo con il tempo corrente al momento della ricezione della richiesta.
- **iat (Issued At):** momento in cui è stato rilasciato il token. Tale campo viene utilizzato per verificare se il token, al momento della richiesta, sia già stato rilasciato; questo controllo risulta essenziale per evitare possibili attacchi condotti da utenti malintenzionati.

Interazione con il Cognito Identity Pool

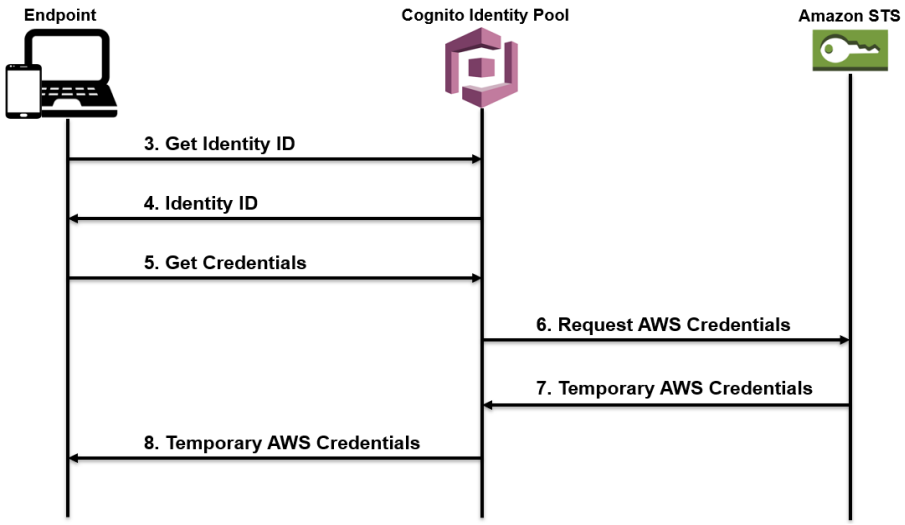


Figura 26: Autenticazione con il Cognito Identity Pool

La *figura 28* mostra la seconda ed ultima fase del processo di autenticazione dell'utente, la quale risulta pilotata dalle interazioni tra l'endpoint e il *CIP*, dove quest'ultimo dialoga anche con l'*Amazon STS* al fine di ottenere le credenziali temporanee. L'*Amazon Security Token Service (STS)*

è un servizio che consente al *CIP* di richiedere delle credenziali temporanee, a cui vengono associati dei privilegi limitati tramite il meccanismo dei ruoli con policy associate, per gli utenti autenticati.

Il *messaggio (3)* viene utilizzato dall'endpoint per la richiesta dell'*Identity* associato al proprio account, inserendo all'interno del payload della richiesta l'endpoint che ha rilasciato l'*ID Token* e il token stesso; l'endpoint nel caso specifico del protocollo utilizzato corrisponde all'identificativo del *CUP*. L'elaborazione della richiesta da parte del *CIP* risulta nella creazione dell'*Identity ID* nel caso in cui sia la prima volta che l'utente sta effettuando il processo di autenticazione o semplicemente il *retrive* per autenticazioni successive alla prima; nel *messaggio (4)* viene ritornato all'endpoint l'*Identity ID*. Il *messaggio (5)* viene utilizzato per richiedere le credenziali temporanee di sessione al *CIP* aggiungendo come payload della richiesta l'*ID Token* e l'*Identity ID* ottenuto nella fase precedente. Le credenziali temporanee di sessione permetteranno all'utente di effettuare le successive richieste per ottenere l'accesso ai servizi offerti da AWS o per contattare le API REST esposte dall'API Gateway fino alla loro scadenza, momento in cui dovrà richiederne delle nuove per continuare ad ottenere l'accesso ai vari servizi. Quando il *CIP* riceve la richiesta non genererà le credenziali temporanee di sessione ma sfrutterà l'*Amazon STS* per la loro generazione tramite il *messaggio (6)*. In risposta, *messaggio (7)*, l'*Amazon STS* fornirà le credenziali temporanee di sessioni a cui è stato associato un ruolo IAM al fine di limitare le autorizzazioni ad esse associate; il ruolo e la policy ad esso associata sono state discusse nel capitolo *2.1.2*. Ogni tentativo di utilizzo scorretto dei token, come ad esempio accedere alle informazioni di end-device associati ad un altro account utente, si tradurrà in una risposta "*403 Forbidden"* da parte del servizio a cui si sta richiedente l'accesso.

Infine, con il messaggio *(8)* si conclude il processo di autenticazione tramite l'invio da parte del *CIP* delle informazioni di sessione all'endpoint. Le informazioni di sessione seguono:

- **IdentityID:** l'identificativo univoco dell'utente che dovrà essere utilizzato per accedere alle API REST esposte dall'API Gateway e per pubblicare e/o ricevere informazioni dai topic MQTT.
- **AccessKeyID:** identificativo delle credenziali d'accesso.
- **SecretAccessKey:** chiave segreta utilizzata per derivare la chiave con cui firmare le richieste da inviare all'API Gateway.
- **SessionToken:** token che andrà passato ad ogni richiesta effettuata al fine di provare la propria identità.
- **Expiration:** indica quando le credenziali ottenute scadranno; viene utilizzato per effettuare, alla scadenza delle credenziali, il processo di *refresh* in modo da evitare di dover ripetere la richiesta nel caso in cui le credenziali fossero scadute e ridurre il numero di richieste necessarie al fine di ottenere le risorse richieste.

2.3 IoT Core

L'**AWS IoT Core** rappresenta il punto nevralgico della soluzione cloud sviluppata, dove tutti i dati di telemetria e di GNSS inoltrati dagli end-device vengono accolti grazie all'interazione cloud-to-cloud (definita dal connettore Loriot-AWS) tra il *broker MQTT* di IoT Core e il cloud LoRaWAN. Prima di analizzare la funzione Lambda che si occupa di come gestire i dati in arrivo dagli end-device andremo ad analizzare come essi vengono creati e memorizzati all'interno dell'*IoT Core* e

di come le funzionalità di AWS sono state sfruttate al fine di attingere al meglio da tutte le potenzialità offerte dalla piattaforma, permettendo di ridurre sia i costi che i tempi d'esecuzione.

2.3.1 Creazione e Memorizzazione dei Dispositivi

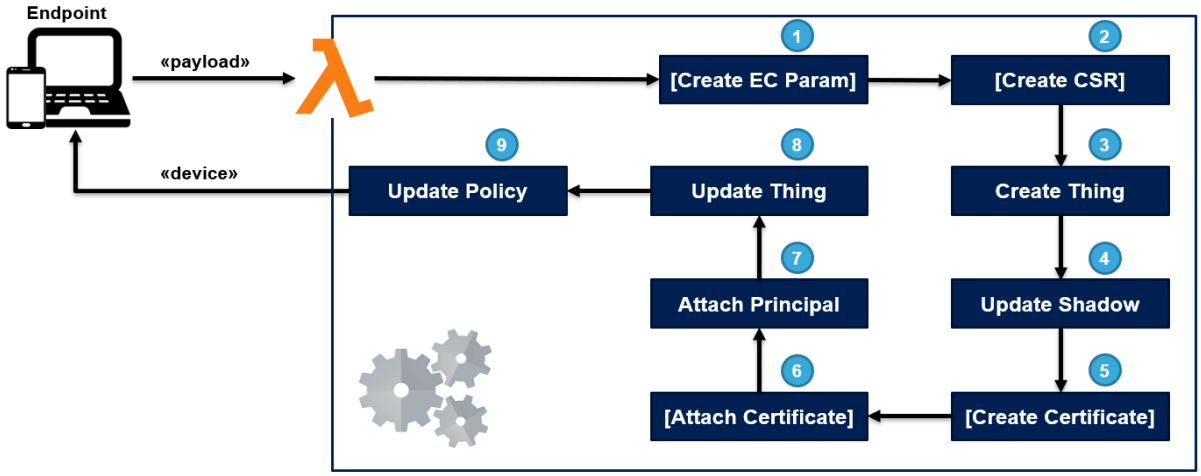


Figura 27: Creazione di un Dispositivo

La *figura 29* mostra il processo di creazione di un end-device all'interno di AWS, dove gli step indicati tra parentesi quadre sono facoltativi e la loro esecuzione dipende da uno dei parametri passati dall'endpoint nel payload. Al fine di iniziare il processo di creazione di un end-device, che nel mondo AWS assume la nomenclatura di *thing*, l'endpoint deve associare alla richieste un payload con la seguente struttura:

$$\{\text{eui}: string, name: string, model: string, certificate: boolean, user: string\}$$

L'*eui* è il *DevEUI* dell'end-device, *name* è un nome custom da assegnare all'end-device, *model* è la tipologia di rete a cui l'end-device si collega (LoRaWAN, SigFox, …), *certificate* indica se durante la creazione della *thing* è richiesta l'associazione di un certificato, e *user* è lo username dell'utente che sta richiedendo la creazione della *thing*.

Come già detto, gli step tra parentesi quadre sono opzionali e verranno eseguiti esclusivamente se l'endpoint ha valorizzato il campo *certificate*, presente nel payload della richiesta, a *true* in modo da indicare la volontà di voler associare un certificato digitale a chiave pubblica (X.509) all'end-device di cui si sta richiedendo la registrazione. A seguire la descrizione dei vari step:

1. Tale step, opzionale, si traduce in una chiamata alla libreria *openssl* al fine di generare i parametri e la coppia di chiavi necessarie alla generazione del certificato a chiave pubblica. La curva scelta per la generazione della coppia di chiavi è *prime256v1*, che genera una curva ellittica da 256 bit, come descritto nel primo capitolo di questa trattazione. Per maggiori informazioni sulla tipologia di curva utilizzata, o per maggiori dettagli, si rimanda all'*rfc5480*.
2. Tale step, opzionale, si traduce in un'ulteriore chiamate alla libreria *openssl* al fine di generare il *Certificate Signing Request (CSR)* che dovrà essere inviato alla *Certificate Authority (CA)*, gestita da AWS, al fine di creare il certificato a chiave pubblica. Un *CSR* è un messaggio che contiene le informazioni del richiedente, in questo caso l'end-device, la chiave pubblica per cui si sta richiedendo la generazione del certificato e una firma sul messaggio, generata tramite la relative chiave privata, al fine di provare il possesso della

coppia di chiavi per cui si sta richiedendo la generazione del certificato. Il campo di maggior rilievo è il *Common Name*, ovvero un campo che indica chi è il possessore del certificato; in questo caso sarà valorizzato con il *DevEUI* dell'end-device.

3. Tale step comporta la creazione di una *thing* all'interno dell'IoT Core in cui si necessita di specificare il nome della *thing* e il nome del *tipo* che si intende associare. Il *tipo* permette di creare una serie di attributi ricercabili sulla *thing* in modo da velocizzarne la ricerca tra più *thing* presenti all'interno dell'IoT Core, assumendo un ruolo molto simile a quello di un database ma riducendo i tempi di ottenimento delle informazioni.
4. Tale step viene realizzato al fine di aggiungere un payload all'interno della sezione *desired* dello shadow associato alla thing. Il payload conterrà il solo campo *status*, un intero che può assumere esclusivamente i valori *1* e *0*. Il valore *1* indica che la *thing* è attiva e l'eventuale ricezione di nuovi dati all'interno del campo *reported* dello shadow comporterà la loro elaborazione al fine di essere processati e inseriti all'interno della corretta tabella del database *DynamoDB*. Il valore *0* indica che la *thing* è stata disabilitata dall'utente e di conseguenza le eventuali modifiche ricevute all'interno della sezione *reported* dello shadow non dovranno essere processate. Questa funzionalità permette all'utente di poter disabilitare temporaneamente una *thing* senza doverla necessariamente cancellare dall'AWS IoT, nel caso in cui l'utente non abbia associato un certificato alla thing, ed inoltre il campo viene analizzato prima della reale esecuzione della funzione Lambda. Dal punto di vista della sicurezza permette di poter bloccare una thing nel caso in cui l'end-device sia manomesso o rubato, garantendo all'utente un controllo maggiore sugli end-device registrati.
5. Tale step, opzionale, comporta l'invio di una *CSR* al server al fine di ottenere l'associazione tra la *thing* e il certificato X.509 creato. In caso di successo, la risposta da parte del server conterrà le seguenti informazioni:
    a. **certificateArn:** l'*Amazon Resource Name (ARN)* del certificato. L'*ARN* di un oggetto all'interno di AWS può essere utilizzato come *principal* e di conseguenza può essere associato alla thing e verrà utilizzato come parametro nello step *(6)*.
    b. **certificateId:** identificativo univoco del certificato. Verrà associato come *attributo* alla thing, al fine di svolgere velocemente tutte le operazioni di verifica o modifica del certificato.
    c. **certificatePEM:** il certificato stesso in formato PEM.
6. Tale step, opzionale, comporta l'associazione del certificato alla *thing*. L'associazione del certificato al dispositivo, oltre per le motivazioni discusse nel capitolo *1*, permette di offrire all'utente un controllo maggiore sulla *thing*, abilitandolo a rendere *inattivo* il certificato o a *revocarlo* definitivamente. Tali funzionalità permettono di bloccare temporaneamente o definitivamente una *thing* per motivi di sicurezza, come la manomissione da parte di un attaccante o il furto. In entrambi i casi, durante l'elaborazione dei dati, il certificato sarà considerato invalido e i dati scartati.
7. Tale step comporta l'associazione dell'*Identity* alla *thing*. Questa operazione risulta di vitale importanza per l'architettura realizzata poiché, grazie alle funzionalità offerte da AWS, è possibile ricercare ed ottenere tutte le *thing* associate ad uno specifico *Principal*, che nel nostro caso è il l'*Identity* dell'utente. Ciò permette di effettuare il *retrive* di tutte le *thing* associate ad un'utente, avendo la sicurezza di ottenere esclusivamente quelle

che sono state registrate tramite uno specifico account, permettendo la segregazione dei dispositivi e rendendo impossibile l'accesso, da parte di un altro utente, ai dati relativi ai dispositivi non propri. Il vantaggio principale, oltre quello legato alla sicurezza, è quello di non dover creare e memorizzare un'associazione *utente-dispositivo* all'interno di una tabella in un database. Questa operazione avrebbe comportato la necessità di interagire, ad ogni richiesta di ottenimento dei dispositivi da parte di un endpoint, con un database, effettuando una query di ricerca e lettura dei dati, un'operazione che avrebbe causato un aumento dei tempi di ottenimento delle informazioni e dei costi, a causa di un maggiore traffico diretto al database. Notando che il retrieve dei dispositivi associati ad un account è un'operazione molto frequente e che nel caso del *Front End* necessita di essere eseguita ad ogni avvio dell'applicazione Web al fine di avere a disposizione le informazioni necessarie per la creazione della vista da mostrare all'utente.

8. Tale step permette di associare tutti gli attributi alla *thing* creata e che saranno utilizzati dalle varie funzioni Lambda per ottenere delle velocità, nella generazione della risposta, molto più rapide. Gli attributi che vengono associati ad una thing seguono:
    a. **createAt:** indica il momento di creazione della thing. Utilizzato per storicizzare il momento di creazione di tutte le thing presenti all'interno dell'IoT Core e può essere sfruttato per operazioni di monitoraggio delle attività.
    b. **eui:** identificativo univoco del dispositivo e che corrisponde al nome della thing (*DevEUI*).
    c. **owner:** username dell'utente che ha creato la thing.
    d. **certificateID:** identificativo univoco del certificato. Tale attributo viene utilizzato, in fase di ricezione dei dati, per ottenere rapidamente le informazioni di validità del certificato senza la necessità di creare una tabella nel database per avere le associazioni *certificato-dispositivo*, ottenendo gli stessi vantaggi discussi nello step *7*. Se il campo *certificate* della richiesta è posto a *false*, l'attributo non sarà presente.
    e. **name:** custom name assegnato dall'utente al dispositivo. Verrà sfruttato dal *Front End* al fine di permettere una user friendly maggiore nell'esperienza di navigazione e di visualizzazione delle informazioni.
    f. **updateAt:** indica il tempo di ultima modifica della thing. Fino a quando la thing non verrà modificata avrà lo stesso valore dell'attributo *createAt*.
    g. **model:** indica la tipologia di rete a cui l'end-device si appoggia. Questo attributo può essere sfruttato per effettuare operazioni differenti, ad esempio in fase di ricezione di nuovi dati, in base al modello di rete dell'end-device.
    h. **publicKey:** chiave pubblica associata alla thing. Viene inserita come attributo per motivazioni analoghe a quelle dell'attributo *certificateID*. A causa delle limitazioni sulla lunghezza e sui caratteri ammessi per il valore dell'attributo, la chiave pubblica viene memorizzata in *base64* ma modificandone leggermente il valore: i byte di padding (indicati con il simbolo *"="*) vengono rimossi dalla stringa e sostituiti con un numero, indicante i byte di padding presenti nella codifica. Se il campo *certificate* della richiesta è posto a *false*, l'attributo non sarà presente.

9. L'ultimo step della creazione della thing consiste nella modifica della policy associata all'utente, che è stata largamente descritta nel capitolo *2.1.1*, al fine di aggiungere l'autorizzazione per interagire con la thing creata e con i topic MQTT ad essa associati.

La procedura di creazione della thing si completa con la costruzione del payload della risposta da inoltrare all'endpoint; il formato della risposta è descritto nella *figura 30*, dove $userData.body$ è il payload ricevuto con la richiesta inviata dall'endpoint.

```
{
    "device": {
        'shadow': {
            'desired': { "status": 1 },
            'reported': null
        },
        'certificateInfo': (userData.body.certificate) ? {'status': 'ACTIVE'} : null,
        'thresholds': {},
        'attributes': attributesToSend
    },
    "privateKey": (userData.body.certificate) ? certificateData.privateKey : null,
    "certificatePEM": (userData.body.certificate) ? certificate.PEM : null
}
```

Figura 28: Risposta della Creazione del Dispositivo

Nel caso in cui si verifichi un errore durante l'esecuzione della funzione Lambda verrà attivata una *procedura di rollback* che permetterà di annullare tutte le operazioni effettuate all'interno dell'IoT Core in modo tale da rendere possibile un secondo tentativo di creazione della thing da parte dell'endpoint. Al fine di rendere possibile la procedura di rollback si tiene traccia, all'interno di un oggetto JSON, dell'identificativo del certificato e della thing nell'eventualità in cui vengano creati durante l'esecuzione dei vari step intermedi: ciò permette di rilevare quali modifiche sono state effettuate in modo da poter eseguire il procedimento di rollback indipendentemente dal punto in cui la funzione Lambda è fallita.

2.3.2 Elaborazione dei Dati

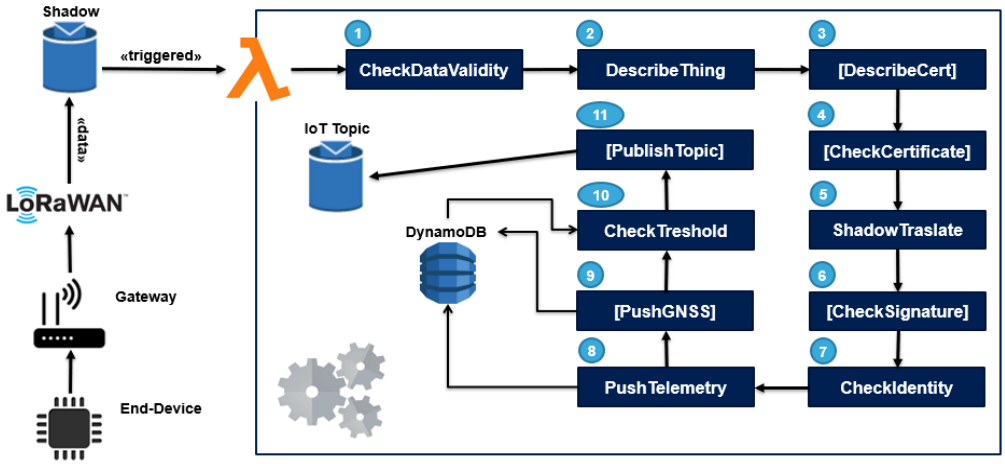


Figura 29: Elaborazione dei Dati

La *figura 31* mostra il processo di elaborazione dei dati inviati da un end-device, dove le operazioni tra parentesi quadre sono opzionali, grazie all'ausilio di un gateway con un modulo *LoRa*, al cloud LoRaWAN. Una volta ricevuti i dati, grazie alla presenza del connettore *Loriot-AWS*, i dati verranno spediti ad un *Message Broker MQTT* che provvederà ad aggiornare lo shadow dell'end-device. Una volta che lo shadow è stato modificato, grazie alla presenza di una *Regola AWS*, viene triggerata una funzione Lambda ad essa associata. Nella regola viene impostata la

tipologia di topic MQTT per la quale deve essere invocata la funzione Lambda associata. Nel caso dell'elaborazione dei dati, la regola agisce sui seguenti topic:

$$SELECT * FROM '\$aws/things/+/shadow/update \\ /documents' WHERE\ current.state.desired.status\ =\ 1$$

La clausola *FROM* permette di specificare un filtro sui topic MQTT. Il filtro creato permette di selezionare tutti gli shadow delle thing, tramite il simbolo *'+'*; inoltre, la parte $update/documents$ indica che al momento della pubblicazione di un nuovo payload all'interno dello *shadow* di una thing dovranno essere selezionate sia lo shadow precedente che quello nuovo. Infine, la clausola *WHERE* permette di specificare una restrizione sulla tipologia di dati che devono essere contenuti all'interno dello shadow. La clausola specificata permette di analizzare la parte *desired* dello stato e, nello specifico, analizza il campo *status*; come già descritto, la regola avvierà l'esecuzione della funzione Lambda esclusivamente se il campo *status* sarà posto ad *1*.

I dati pubblicati all'interno dello shadow saranno inseriti all'interno della sezione *reported* ed hanno i seguenti campi fondamentali:

$$\{cmd: string, fcnt: number, data: string, EUI: string\}$$

Dove *cmd* indica la tipologia di pacchetto, *fcnt* è stato descritto nella sezione *1.2.1*, *data* è una stringa in esadecimale che rappresenta i dati trasmessi dagli end-device e *EUI* rappresenta il *DevEUI* dell'end-device che ha inviato i dati.

Nel caso in cui la regola venisse triggerata, l'esecuzione della funzione Lambda ha il seguente flusso:

1. La prima operazione svolta è il controllo sulla validità del formato dei dati ricevuti. In tal senso, vengono effettuate le seguenti operazioni:
   a. Viene verificato che il campo *data* sia presente all'interno della sezione *reported* dello shadow.
   b. Viene verificato che il campo *cmd* abbia valore *'tx'*, questo viene effettuato per selezionare il corretto set di informazioni da analizzare.
   c. Viene verificato che il campo *fcnt* sia differente dal valore del campo *fcnt* della sezione *reported* del vecchio shadow. Questo viene effettuato per evitare la ricezione di pacchetti duplicati o di possibili attacchi replay condotti da un attaccante.

   Nel caso in cui uno dei seguenti controlli dovesse fallire, la funzione Lambda terminerà immediatamente, scartando silenziosamente i dati.
2. Tale operazione permette di selezionare gli attributi associati alla *thing*. Questo risulta fondamentale al fine di selezionare l'identificativo del certificato (attributo *certificateID*) e la relative chiave pubblica (attributo *publicKey*). Se i due attributi citati dovessero essere assenti significherebbe che la *thing* non ha nessun certificato associato, comportando la non effettuazione delle operazioni racchiuse tra parentesi quadre.
3. Tale operazione, opzionale, viene eseguita per richiedere, tramite l'attributo *certificateID*, le informazioni circa il certificato associata alla thing all'IoT Core.

4. Tale operazione, opzionale, permette di verificare lo stato del certificato associato alla *thing*. A fine di verificarne la validità vengono effettuare le seguenti operazioni:
    a. Viene verificato lo *status* del certificato che deve essere uguale ad *ACTIVE*, in caso contrario i dati vengono silenziosamente scartati poiché non considerati validi. Lo *status* può assumere valore *ACTIVE*, *INACTIVE* e *REVOKED*.
    b. Viene verificato il campo *validity* del certificato ove la proprietà *notBefore* non deve essere maggiore del timestamp attuale e la proprietà *notAfter* non deve essere maggiore del timestamp attuale. Questa operazione viene effettuata per verificare se il certificato è espirato o se sta venendo utilizzato prima del suo periodo di validità. Se uno dei due controlli non viene soddisfatto, i dati vengono silenziosamente ignorati poiché la firma associata è stata prodotta con un certificato non valido.
5. Tale operazione permette la traduzione dei dati ricevuti dall'end-device dal formato esadecimale al formato JSON. Al fine di effettuare la conversione viene utilizzata una struttura JSON d'appoggio per la corretta elaborazione del formato esadecimale utilizzato dai dispositivi ST connessi; la struttura utilizzata ha il seguente formato:

```
{
  "type": {
    "skip": "boolean",
    "size": "number",
    "resolution": "number",
    "component": "number",
    "name": ["name"]
  }
}
```

Figura 30: Formato Dati da Processare

Dove *type* è un identificativo da 2 byte che indica il tipo di misura che è stato registrato dall'end-device; *skip* indica se la misura registrata debba essere analizzata o ignorata; *size* indica il numero di caratteri da estrarre dalla stringa per essere convertiti in formato decimale, tenendo conto che i dati sono rappresentati in complemento a due; *resolution* indica l'esponente, base 10, del divisore del valore estratto dalla stringa al fine di portarli nell'unità di misura corretta; *component* indica il numero di dimensioni in cui viene rappresentato il dato. Si noti che il valore di *size* fa riferimento al singolo componente. Infine, *name* è un array dei nomi delle singole componenti della misura catturata. Se avviene un errore durante la conversione dei dati, i dati verranno scartati silenziosamente poiché considerati invalidi.

6. Tale operazione, opzionale, permette di verificare la firma associata ai dati. Al fine di verificare la firma, si calcolerà il *message digest* sui dati ricevuti, si decifrerà la firma ricevuta tramite la chiave pubblica associata alla thing ed infine si procederà a confrontare la firma prodotta con quella ricevuta. Se le due firme non coincidono, i dati vengono silenziosamente ignorati poiché hanno subito un'alterazione durante il tragitto e quindi considerati non validi. Rigorosamente vengono effettuate le seguenti operazioni:

$$sha256(data) == decrypt(K_{pub}, signature)$$

7. Tale operazione verifica la presenza dell'*Identity* come *principal* della thing. Questo viene effettuato al fine di inserire i nuovi dati raccolti all'interno della rispettiva tabella nel

database *DynamoDB* utilizzando come chiave di partizionamento principale tale identificativo, per i motivi di autorizzazione discussi nel capitolo *2.1.2*. Se non viene trovato l'*Identity* associato alla thing il pacchetto viene silenziosamente scartato poiché la *thing* viene considerata non attendibile.

8. Tale operazioni effettua la costruzione di un payload adeguato ad effettuare una query nella tabella *Telemetry* del DynamoDB. Per motivi di sicurezza, nel caso in cui non sia presente la firma sui dati, il *timestamp*, che ricordo essere la chiave di ordinamento primaria della tabella, viene aggiunta dalla funzione Lambda e non prelevata dai dati.
9. Tale operazione, opzionale, effettua la costruzione di un payload adeguato ad effettuare una query nella tabella *GNSS* del DynamoDB. Per motivi di sicurezza, nel caso in cui non sia presente la firma sui dati, il *timestamp*, che ricordo essere la chiave di ordinamento primaria della tabella, viene aggiunta dalla funzione Lambda e non prelevata dai dati.
10. La fase finale del processo di elaborazioni consiste nel verificare se per il dato *eui* siano presenti soglie sui dati di telemetria. Nell'eventualità in cui vengano trovate soglie settate si procederà a confrontarle con i corrispettivi dati ricevuti e, se eccedono, sarà costruito un nuovo oggetto JSON contenente le informazioni circa il superamento della soglia settata dall'utente. Se nessuna soglia viene trovata o nessuna di esse viene superata, l'oggetto JSON avrà valore *null*. La notifica costruita, per ognuna delle soglie superate, avrà il seguente formato:

    $$\{timestamp: string, threshold: string, exceeded: number, value: number\}$$

    Dove il campo *timestamp* viene espresso in secondi, *threshold* è il nome univoco della soglia che è stata superata, *exceeded* è il valore numerico che indica di quanto la soglia è stata superata. Si noti che nel caso in cui venga superata una soglia inferiore tale valore sarà negativo. Infine, *value* indica il valore della misura che è stato catturato dall'end-device.
11. Tale operazioni viene effettuata esclusivamente nel caso in cui l'oggetto JSON costruito nella fase (*10*) sia diverso da *null*. In tal caso, si procederà ad effettuare una pubblicazione di un nuovo payload all'interno del topic MQTT relativo alla thing di cui si sono ricevuti i dati. Il topic MQTT su cui verrà pubblicato il nuovo payload ha il seguente formato:

    $$cognitoIdentityID/eui$$

    Dove *cognitoIdentityID* è l'*Identity* dell'utente e *eui* è il *DevEUI* dell'end-device che ha inviato i dati.

Come è possibile vedere dall'operazione *11*, il topic MQTT utilizzato per la pubblicazione è univoco per ogni device e non vengono utilizzati più topic in base alla misura cattura al fine di ridurre al minimo i topic su cui l'endpoint dovrà effettuare la *subscribe* per ricevere notifiche relative ad un determinato end-device.

2.4 API Gateway

L'**API Gateway** garantisce diverse funzionalità che risultano utili nella realizzazione rapida e sicura di API REST. Nelle API esposte, che richiedono l'autenticazione dell'utente, si è sfruttato il meccanismo *AWS IAM* che permette di far verificare all'API Gateway la firma sulle richieste e di verificare il *session token* aggiunto nella richiesta dall'utente: nel caso in cui una delle operazioni

dovesse fallire verrà tornato un messaggio "*403 Forbidden*" per informare l'utente che non ha l'autorizzazione per accedere alle risorse richieste. Inoltre, viene anche verificata la policy IAM associata all'*Identity* dell'utente in modo da appurare che l'utente abbia l'autorizzazione necessaria per poter contattare una determinata API REST o di contattare un dato servizio.

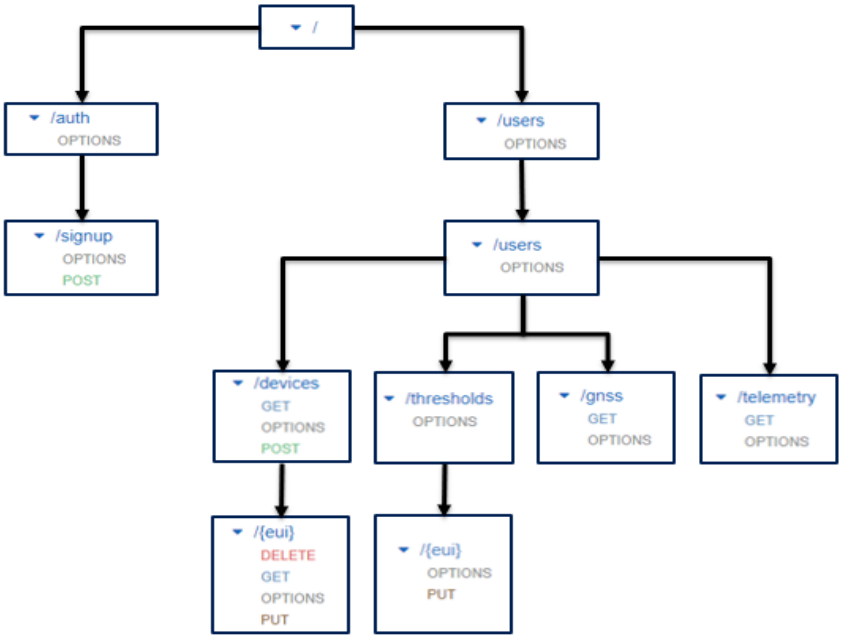


Figura 31: Architettura API REST

Inoltre, è possibile specificare il tipo di contenuto della richiesta (nel caso di POST e PUT), e il modello JSON che deve possedere la richiesta, permettendo di eseguire un *prefiltering* del payload prima dell'esecuzione della funzione Lambda, in modo da evitare che venga eseguita nel caso in cui il payload non fosse valido. Inoltre, tale controllo viene interamente eseguito dall'API Gateway in maniera del tutto automatico a seguito della ricezione di una richiesta, integrando perfettamente i controlli all'interno dell'API Gateway e non della funzione Lambda, sfruttando ancora una volta le funzionalità offerte dall'ambiente di sviluppo.

Quando la richiesta viene considerata valida avviene il processo di integrazione in cui si può specificare la tipologia di backend su cui opera il metodo esposto dall'API, ovvero una funzione Lambda o un servizio AWS, come i dati della richiesta devono essere trasformati prima che vengano inviati al backend scelto.

Nella *figura 33* vengono rappresentate tutte le API REST esposte dall'API Gateway. Tutte fanno riferimento ad una specifica funzione Lambda, ad eccezione delle API *gnss* e *telemetry* che fanno

```
{
   "eui": {
      data: [
         {
            "measure": "number",
            "timestamp": "number"
         }
      ],
      metadata: {
         "UserIdentity": "string"
      }
   }
}
```

Figura 32: Response Integration

riferimento ad un servizio AWS, ovvero DynamoDB. Come già visto, l'accesso diretto a *DynamoDB* è controllato tramite una *segregazione a livello di row,* permettendo un accesso esclusivo alle informazioni relative alle *thing* registrate dall'utente che effettua la richiesta grazie all'utilizzo delle policy IAM associata all'*Identity.* Ciò permette di velocizzare l'operazione di lettura dal database senza dover istanziare una funzione Lambda che aumenterebbe i tempi di esecuzione e i costi della soluzione. Al fine di permettere la comunicazione diretta viene effettuata una mappatura della richiesta dell'utente in modo da essere compatibile con il payload della richiesta atteso da *DynamoDB*, aggiungendo eventuali *query parameters* presenti all'interno della richiesta.

In maniera del tutto analoga, dopo l'elaborazione della richiesta da parte del backend, l'API Gateway intercetterà la risposta estraendone il payload e riadattandolo secondo un modello generalizzato avente la struttura mostrata nella *figura 34.* Qui l'oggetto JSON di risposta ha tante proprietà quante sono le *thing* di cui sono state prelevate le rilevazioni e tali proprietà avranno come chiave il *DevEUI* della thing e come valore un altro oggetto JSON. Quest'ultimo ha due proprietà: *data*, un array di oggetti aventi come proprietà tutte le misura che sono state catturate in un dato timestamp, e *metadata* un oggetto avente come unica proprietà *UserIdentity* che ha come valore una stringa che rappresenta l'*Identity* dell'utente. Infine, il nuovo modello viene imbustato all'interno di una risposta HTTP correlato dei necessari header, quindi settato il codice di risposta opportuno per poi essere inoltrata all'endpoint che ha contattato l'API.

L'API sotto la root *auth* sono API che non effettuano alcun tipo di controllo di autorizzazione e di conseguenza vengono utilizzate dagli endpoint non autenticati, sprovvisti di una policy IAM e delle informazioni di sessione. In particolare, l'unica API esposta in questa root è quella necessaria per la terminazione del processo di registrazione discussa precedentemente.

3.0 Front End

La realizzazione del Front End è basata sull'utilizzo del *Framework Javascript Vue.js*, un **progressive framework** per costruire interfacce utente sfruttando il *dual-binding* tra modello dati e vista [11]. A differenza degli altri framework monolitici, come ad esempio *Angular*, Vue è progettato per essere incrementalmente utilizzabile, offrendo una grande flessibilità basata sulla possibilità del programmatore di scegliere quali librerie utilizzare, ottenendo una maggiore libertà e semplicità di sviluppo.

Vue è basato sul pattern *MVVM* che, a differenza del pattern *MVC* specificamente designato per la creazione della separazione tra il *controller* e la *vista* tramite il data-binding, è designato per consentire alla vista e al controller di comunicare tra loro, definendo di fatto il comportamento dell'applicazione a runtime e non eseguendo logiche di business prima del rendering della vista:

- **Model:** rappresenta l'implementazione del dominio dati gestito dall'applicazione;
- **View:** rappresenta la parte grafica che viene mostrata all'utente;
- **ViewModel:** rappresenta il ponte tra il model e la view, fornendo alla view i dati in un formato consono alla presentazione.

Tramite l'utilizzo del framework è stato possibile realizzare una *single page application* in maniera molto articolata. Inoltre, sono state sfruttate le seguenti librerie principali per la costruzione dell'architettura e dell'aspetto grafico:

- **Vue-Router:** router ufficiale di Vue.js. Risulta strettamente integrato con il core Vue per semplificare la creazione di single page application offrendo la possibilità di creare route annidate, abilitare le transizioni di navigazione, un controllo a granularità fine sulla navigazione e la possibilità di gestire parametri nel passaggio tra una route e la successiva [12].
- **Vuex:** le applicazioni di grandi dimensioni possono crescere in complessità a causa di una frammentazione dello stato che risulta sparso tra molti componenti rendendo l'interazione tra i vari componenti articolata e complessa. *Vuex* è la soluzione offerta da Vue per risolvere tale problematica. Il concetto su cui si basa Vuex è lo *store*, ovvero un magazzino dove memorizzare tutte le informazioni di stato dell'applicazione in modo che tutti i componenti possano prelevare le informazioni di cui necessitano da un unico punto, essere informati a seguito di una modifica della parte di stato a cui sono interessati. Inoltre, negando la possibilità di apportare modifiche dirette allo stato, dove adesso sarà necessario invocare una *action* che si occuperà di effettuare una certa logica applicativa, riesce a verificare se le informazioni richieste dall'utente sono disponibili o effettuando una chiamata asincrona per soddisfare l'esecuzione di un'azione dell'utente, per poi invocare una *mutations* con il compito di effettuare l'aggiornamento della parte di stato interessata al cambiamento. L'architettura descritta è basata sul pattern *Flux*, dove è definito un flusso unidirezionale in cui il componente non può mai interagire direttamente con lo store ma dovrà contattare un'*actions* o un *getters*.
- **Vuetify:** framework per la realizzazione del design dei component molto simile al *framework Bootstrap,* il quale sarà comunque utilizzato al fine di ereditare le

impostazioni di stile dei principali tag HTML. Vuetify offre svariati componenti per il design con uno stile semplice, elegante e facilmente customizzabile.

La trattazione di quest'ultimo capitolo seguirà una prima overview sull'architettura utilizzata per la realizzazione del *Front End*, esplicitando le best practices e i vantaggi derivati dalle scelte intraprese, dando maggior rilievo ai componenti generici e riutilizzabili che sono stati creati per offrire un supporto molto ampio all'estendibilità e la manutenibilità del codice. Successivamente saranno analizzati il comportamento del *Router*, dando maggior attenzione ai controlli di sicurezza che sono stati implementati per gestire l'accesso alle viste, e dello *Store,* al fine di mostrare le potenzialità del modulo di configurazione realizzato che, come vedremo, permetterà di customizzare tramite la semplice modifica di un documento JSON la struttura dell'applicazione. Infine, sarà mostrata la fase di build dell'applicativo per mettere in risalto le

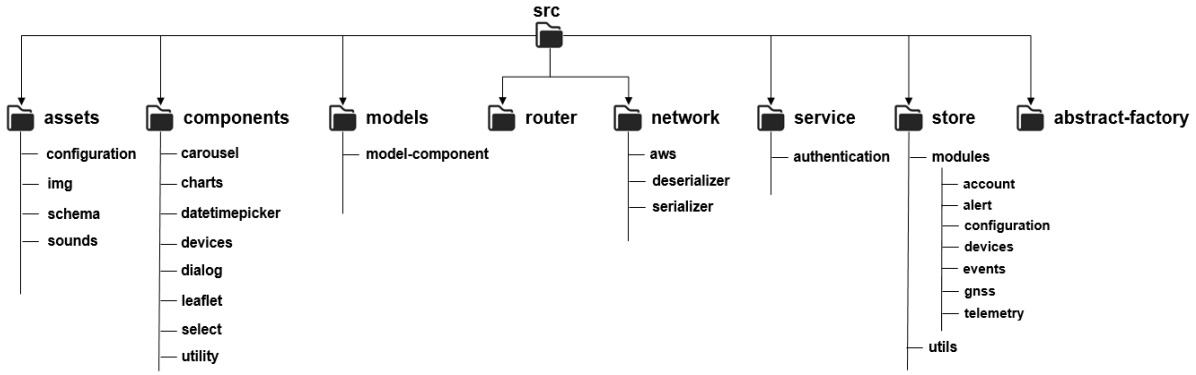


Figura 33: Struttura ad Albero delle Directory

funzionalità che sono state introdotte.

La *figura 35* mostra la struttura ad albero delle directory utilizzate all'interno della soluzione. Come è possibile vedere, si è fatto largo uso della suddivisione in directory delle varie funzionalità in modo da migliorare la leggibilità del codice, la suddivisione delle funzionalità e dell'architettura ottenendo una più semplice comprensione della logica sviluppata. Segue una breve descrizione delle directory:

- **Assets:** directory utilizzata per contenere tutto il materiale sprovvisto di logica ma utile per la corretta implementazione del *Front End,* come ad esempio le varie immagini utilizzate, suoni e lo schema per la validazione del file di configurazione;
- **Components:** insieme di directory dei componenti utilizzati all'interno dell'applicativo;
- **Models:** modelli di dati utilizzati per rappresentare gli oggetti o le *props* dei componenti;
- **Router:** directory dedicata all'istanza del *Vue Router*;
- **Network:** directory contenente la logica di comunicazione tra il *Front End* e il *Back End*;
- **Service:** directory contenente i servizi, utilizzati dalle *actions*, o in alcuni casi direttamente dai componenti, al fine di effettuare le chiamate asincrone;
- **Store:** insieme di directory che rappresentano i moduli dello store;
- **Abstract Factory:** insieme di *Abstract Factory* utilizzate per astrarre la tipologia di backend, il sistema di autenticazione e il gestore della firma, sulle richieste da inviare al backend, utilizzati.

3.1 Overview Applicazione

Prima di entrare nel dettaglio delle singole componenti che formano l'architettura, andremo ad effettuare una panoramica sull'architettura e sui moduli che sono stati costruiti al fine di permettere la realizzazione di una dashboard il più possibile indipendente dalla tipologia di provider. Questa generalizzazione, come vedremo, è stata resa possibile spostando all'interno di

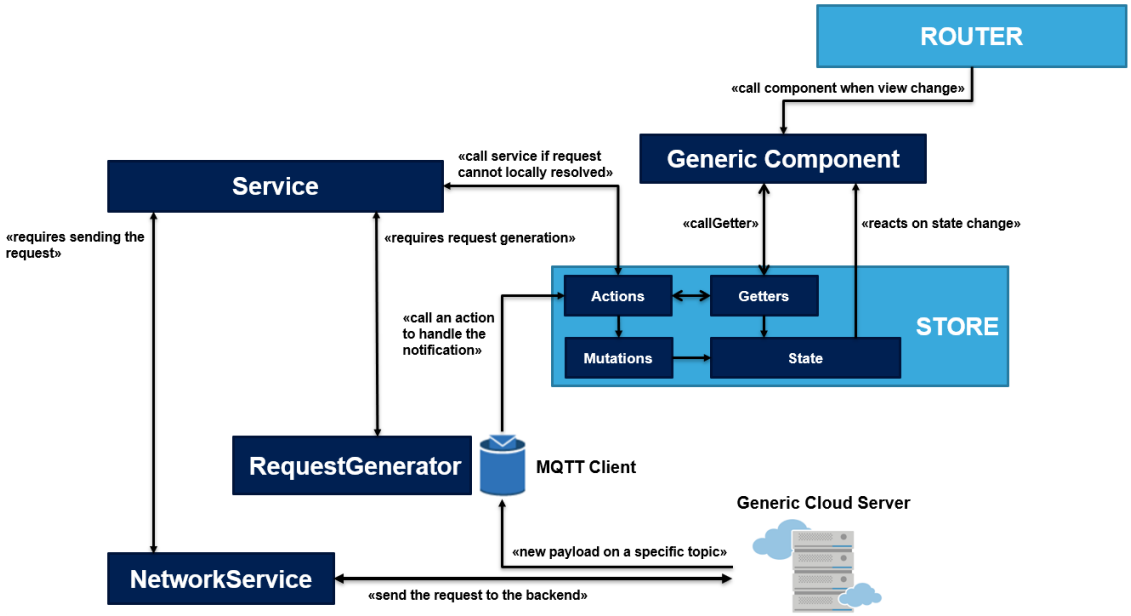


Figura 34: Schematico Architettura Front End

poche classi tutta la logica necessaria al fine di comunicare correttamente con il backend. Questo permette anche di rendere semplice e rapida l'implementazione di nuove classi che siano in grado di gestire correttamente un backend differente, rendendo di fatto lo strato applicativo gestito dai componenti del tutto indipendente dal provider, con alcune forzature sulla tipologia di metodi da esporre in queste speciali classi, al fine di rendere possibile l'integrazione senza modifiche al codice sorgente.

Lo schema nella *figura 36* rappresenta l'architettura utilizzata per la realizzazione del Front End. Quando l'utente, navigando all'interno dell'applicazione, modifica la URI entra in azione il modulo di routing che effettuerà il rendering del componente associato alla nuova URI, modificando dinamicamente la vista mostrata all'utente. Quando all'interno di un componente si necessita di reperire informazioni di stato, ad esempio per mostrare i device registrati dall'utente, si effettuerà in alcuni casi la chiamata ai *getters* esposti dal modulo, o in alternativa si utilizzerà un *action* al fine di ottenere le informazioni da visualizzare. Le *actions* hanno il compito di verificare se la richiesta del componente può essere soddisfatta localmente o se è necessario invocare un opportuno metodo esposto dallo strato dei *servizi* al fine di richiedere al backend le informazioni necessarie a soddisfare la richiesta del componente.

Lo strato dei servizi è composto da classi, ognuno che gestisce un insieme di richieste differenti, suddivise in base ai moduli messi a disposizione dallo store. All'interno delle classi sono esposti dei metodi statici che hanno come unico compito quello di modificare leggermente la richiesta pervenuta dalla *action*, effettuando un mapping tra metodo e corrispettiva API, per poi contattare l'istanza concreta del *RequestGenerator* per la generazione delle informazioni

necessarie per l'effettuazione di una richiesta. Una volta ottenute le informazioni sarà invocato, all'interno del metodo esposto dal servizio, l'unico metodo pubblico a disposizione della classe *NetworkService*, ovvero $handlerRequest$, passando le informazioni inerenti alla richiesta ritornate dal *RequestGenerator*. A questo punto il *NetworkService* invierà la richiesta, utilizzando il corretto protocollo su cui sono basate le informazioni prodotte dal *RequestGenerator*, attenderà una risposta, ne gestirà la ricezione applicando un opportuno *deserializzatore* o gestendo l'eventuale errore pervenuto all'interno della risposta. In entrambi i casi, la risposta sarà ritornata al servizio che la ritornerà al chiamante. La *action* provvederà a chiamare un opportuna *mutation*, che si occuperà di memorizzare i nuovi dati all'interno dello store. Infine, la *action*, chiamando un *getters* opportuno, ritornerà i dati al chiamante.

Inoltre, vi è un'ulteriore interazione con il backend data dalla possibilità di aprire dei canali MQTT al fine di ricevere notifiche se una soglia settata dall'utente sulle misure catturate dai device venisse superata. La pubblicazione, ricezione e iscrizione a uno dei topic MQTT a disposizione viene gestita tramite l'utilizzo di un client MQTT annegato all'interno dell'istanza concreta della classe *RequestGenerator*, che ha il compito di triggerare una *actions* all'interno del modulo *events* dello store al fine di permettere la memorizzazione delle nuove informazioni ricevute. In maniera del tutto duale, può essere sfruttato per pubblicare un nuovo payload all'interno del topic MQTT grazie all'abilitazione del canale *bidirezionale*.

3.1.1 App.vue

Il componente **App** è il wrapper principale dell'applicazione, ovvero il componente che viene richiamato all'avvio dell'applicazione. Al suo interno viene gestita la costruzione dell'applicazione, ovvero l'instanziazione di tutti i componenti a livello globale e l'inserimento della *router-view* che permetterà al *router* di caricare dinamicamente i nuovi componenti quando la URI viene modificata, sostituendo quelli precedentemente istanziati.

I componenti che necessitano di essere sempre presenti all'interno dell'applicativo e che vengono gestiti all'interno del componente *App* seguono:

- **Snackbar:** componente offerto da V*uetify* ed utilizzato per mostrare i messaggi di successo o di fallimento all'utente, ad esempio pervenuti da una richiesta HTTP o dalla ricezione di una notifica da uno dei topic MQTT a cui l'applicativo è iscritto. Il componente rimane in attesa di modifiche nel modulo *alert* dello store, modulo ove vengono pubblicati gli esiti di un'operazione svolta dall'utente.
- **AsyncCallDialog:** componente che abilita la visualizzazione di un *messaggio di waiting* che viene mostrato all'utente per forzare l'attesa, bloccando l'intera interfaccia grafica, durante l'esecuzione di una chiamata asincrona. Il componente rimane in ascolto su una parte dello stato contenuta nel modulo *alert*, proprietà *waiting*, al fine di ricevere informazioni sull'inizio e il termine della chiamata asincrona.
- **Sidebar:** componente che gestisce la barra di navigazione laterale che permetterà all'utente di muoversi all'interno dell'applicativo. Il componente rimane in ascolto sullo stato dell'utente per permettere la visualizzazione di due menù differenti in base allo stato di autenticazione dell'utente. La sidebar è stata costruita in modo da essere responsive e cambiare dimensione quando riceve un input dall'utente, tramite il pulsante di collapse collocato in basso all'interno della sidebar o, nel caso in cui le

dimensioni dello schermo siano troppo piccole, per permettere una corretta visualizzazione del contenuto centrale mostrato dal *router-view*.

In fase di avvio, prima che il componente principale sia costruito, avviene la verifica dello stato di autenticazione dell'utente: se l'utente è autenticato si effettuerà la richiesta di tutti i device associati all'account in modo da essere resi disponibili a tutti, effettuando tale operazioni una sola volta durante il ciclo di vita dell'applicazione. Successivamente, si procederà ad aprire un canale MQTT verso i topic associati al device registrato dall'utente in modo da permettere, fin dall'avvio dell'applicativo, la ricezione di notifiche: le notifiche saranno disponibile indipendentemente dalla pagina che l'utente sta visualizzando e alla ricezione sarà avvisato tramite un segnale acustico e un messaggio che lo informerà della disponibilità di una nuova notifica e da quale device è stata inviata.

3.2 Modelli dei dati

La modellizzazione dei dati è una pratica fondamentale per la corretta costruzione di un applicativo, poiché permette di astrarre le informazioni che devono essere modellate, aggiungere proprietà comuni ad un oggetto nonché offrire un grado di manutenibilità del codice molto maggiore, potendo sfruttare il riutilizzo dei modelli all'interno di più componenti.

Gli unici modelli dati utilizzati sono *user*, che permette di rappresentare l'utente e i rispettivi dati di autenticazione dipendenti dalla tipologia di provider utilizzato, e *device,* che rappresenta il cuore dell'applicativo essendo il fulcro delle informazioni e delle features messe a disposizione. Inoltre, sono stati costruiti due modelli interni, ovvero degli oggetti da utilizzare come *props* di altri componenti, che sono *chart-options* e *form-payload*.

Le **props** sono un utile meccanismo offerto da Vue.js per abilitare il passaggi dei dati da un *parent* a un *child*. Il vantaggio preponderante è la possibilità di specificare la tipologia di dati che corrisponde ad una *props*, un *nome* che dovrà essere utilizzato per il passaggio dei dati, una *descrizione* per esplicitare l'utilizzo all'interno del componente e un attributo (*required*) che permette di specificare se la *props* è obbligatoria o meno, permettendo inoltre di specificare un eventuale valore di *default* nel caso in cui non venga fornita dal componente *parent*. La composizione di una *props* con gli attributi citati permette, in *development mode*, di segnalare eventuali errori di gestione delle *props*, ad esempio nel caso in cui una *props* obbligatoria non venga fornita o se il tipo passato non corrisponde a quello atteso.

3.2.1 Modello Utente


```
{
   "username": "string",
   "loginData": {
      "accessToken": "string",
      "idToken": "string",
      "refreshToken": "string",
      "principal": "string",
      "accessKeyId": "string",
      "secretAccessKey": "string",
      "sessionToken": "string",
      "expiration": "number"
   }
}
```


Figura 35: Modello Utente

Come già accennato, il **modello utente** permette di rappresentare le informazioni correlate ad un utente. La classe che rappresenta il *modello* utente ha due proprietà: *username* e *loginData*.

Lo *username* rappresenta il nome univoco dell'utente immenso in fase di registrazione e che sarà usato per effettuare la procedura di autenticazione, mentre *loginData* rappresenta tutti i dati di sessione che l'utente ha ottenuto al termine del processo di autenticazione. La proprietà *loginData* è un oggetto a cui non viene imposta una struttura ben definita al fine di rendere indipendente la classe *User* dal sistema di autenticazione utilizzato, il quale potrebbe cambiare o essere differente in future espansioni del lavoro. La mancata strutturazione dell'oggetto permette di aggiungere informazioni differenti in base alla tipologia di provider che si sta utilizzando. La *figura 37* mostra la struttura che avrà il modello utente nel caso in cui venga utilizzato AWS come provider di riferimento per il processo di autenticazione.

Le informazioni contenute all'interno della proprietà *loginData* sono tutte relative alle varie tipologie di token discusse nel *capitolo 2* e che verranno utilizzate per richiedere l'accesso ad uno qualunque dei servizi esposti dal provider, per effettuare la procedura di refresh delle credenziali di sessione, quando la proprietà interna *expiration* avrà raggiunto la scadenza, e per l'interazione con i topic MQTT.

3.2.2 Modello Device

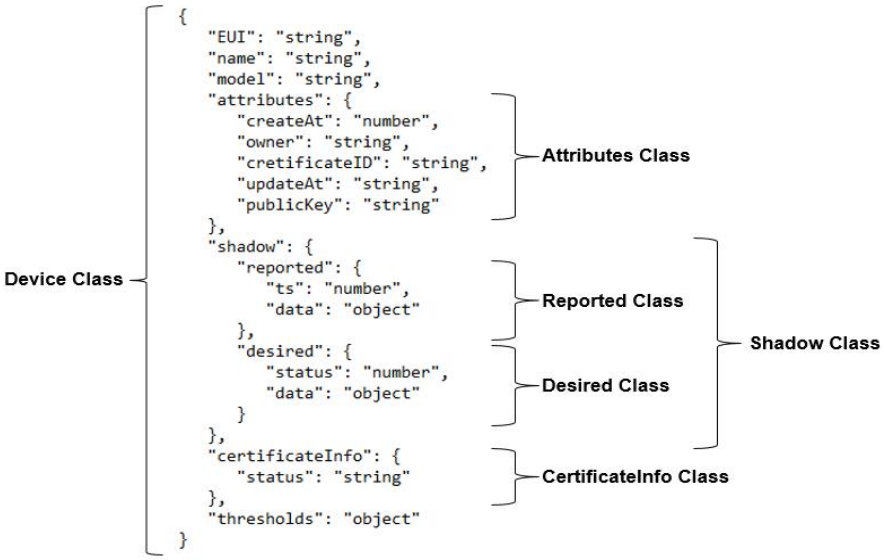


Figura 36: Modello Device

Come già accennato, il **modello device** permette di rappresentare tutte le informazioni relative ad un end-device. La classe utilizzata per rappresentare il modello dati, rappresentata nella *figura 38*, è composta da tre proprietà principali che permettono di identificare univocamente il device, ovvero l'*eui* che rappresenta il *DevEUI* dell'end-device, il *name* che rappresenta il nome custom assegnato dall'utente in fase di registrazione e il *model* che rappresenta la tipologia di rete a cui l'end-device si collega. Inoltre, sono presenti quattro proprietà complesse la cui trattazione segue:

- **Attributes:** classe degli attributi associati dal backend al device. Tali informazioni saranno visualizzate nei dettagli del device al fine di dare la possibilità all'utente di prendere nota della chiave pubblica ad esso associata nonché dell'istante temporale di ultima modifica e di creazione;
- **Shadow:** classe contenente i documenti JSON che descrivono lo stato del device. Lo stato del device, come già discusso nel *capitolo 2.1.1*, è suddiviso in una parte *desired,*

che permette all'utente di specificare la configurazione che l'end-device debba utilizzare, e una parte *reported,* che permette all'end-device di riportare il proprio stato, in cui vengono rappresentati anche gli ultimi dati rilevati:
    - **Desired:** classe che rappresenta lo stato dell'end-device impartito dall'utente. Ha una proprietà fissa, *status*, che rappresenta lo stato dell'end-device, discusso nel *capitolo 2.3.1*, e una proprietà *data*, un oggetto generico in cui non viene imposta nessuna struttura in modo da poter elaborare qualunque tipologia di informazione aggiuntiva. Questo permette di rendere indipendente la classe dalla tipologia di azioni che possono essere impartite all'end-device, operazioni che dipendono dalla tipologia di rete usata, dalla tipologia di end-device e anche dal provider;
    - **Reported:** classe che rappresenta lo stato corrente riportato dall'end-device. Ha una proprietà fissa, *ts* rappresentante l'istante temporale di ricezione dei dati, e una proprietà *data* contenente i dati di telemetria e GNSS ricevuti dall'end-device. Come nel caso precedente, non vi è imposta alcuna struttura dati al fine di mantenere una massima generalità nelle tipologia di dati che possono essere trasportati dall'end-device.
- **CertificateInfo:** classe che riporta le informazioni rilevanti sul certificato; ha un'unica proprietà *status* che rappresenta lo stato del certificato. Lo stato del certificato può essere *Active*, *Inactive* e *Revoked*, e può essere imposto dall'utente nella sezione dettagli del device;
- **Thresholds:** oggetto contenente le soglie che verranno settate dall'utente sul device. Tale proprietà verrà sfruttata per la ricezione di notifiche nell'eventualità in cui l'end-device rilevi una misura al di sopra, o al di sotto, di una delle soglie specificate nell'oggetto.

3.2.3 Modello Chart Options

Modello dati che rappresenta le opzioni da passare al *wrapper data-chart*, che sarà discusso nel *capitolo 3.7.5*, come *props* per la configurazione delle impostazioni del grafico. La classe risulta preimpostata in modo da essere pronta all'uso senza necessità di configurarla; ogni proprietà ha un valore di default che però può essere modificato una volta che la classe viene istanziata. La classe è stata costruita in maniera tale da essere compatibile con le impostazioni richieste dalla libreria *ChartJS*, di conseguenza per un maggior approfondimento sulle proprietà settabili all'interno della classe si faccia riferimento alla documentazione ufficiale di *ChartJS*. Le proprietà utilizzate all'interno della classe seguono:

- **Legend:** permette di configurare le impostazioni relative alla legenda associata al grafico.
- **Scales:** oggetto contenente le informazioni di configurazione degli assi *x* e *y*, ovvero la tipologia di dati che deve essere plottata, il formato delle label o informazioni di stile. È possibile configurare assi multipli;
- **Responsive:** booleano settato a *true* di default al fine di permettere il *resize* del grafico quando la dimensione dello schermo risulta differente, abilitandone la fruizione anche in ambienti mobile;

- **MaintainAspectRatio:** booleano settato a *true* di default al fine di mantenere sempre, indipendentemente dall'operazione di *resize*, lo stesso rapporto tra l'altezza e la larghezza del grafico;
- **Tooltips:** oggetto che permette di associare una callback all'evento *mouseover* di un punto plottato all'interno del grafico.

3.2.4 Modello Form Payload

Modello dati che rappresenta il valore della props da passare al *wrapper form dialog*, contenente gli elementi da costruire all'interno del form e le proprietà associate al form, il wrapper sarà discusso nel *capitolo 3.7.2*, a cui si rimanda per maggiori informazioni.

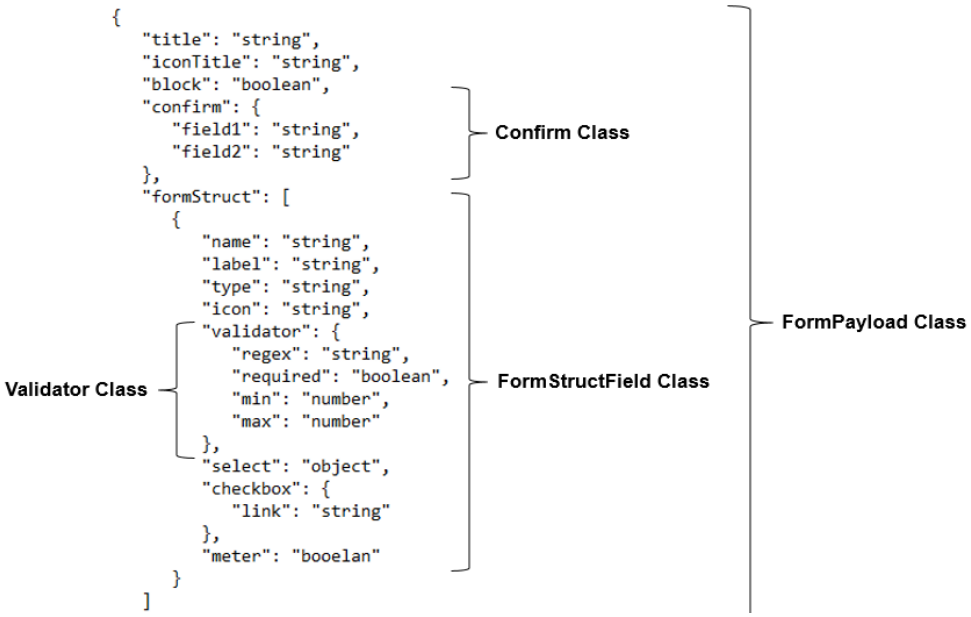


Figura 37: Modello Form Payload

La classe permette di configurare le impostazioni generiche del form, grazie alle proprietà *title* che rappresenta il titolo del form, *iconTitle* che rappresenta l'identificativo univoco dell'icona da associare al titolo, *block* che indica se il form deve essere chiudibile (*dialog*) o fisso (*view*), e *confirm* che indica la necessità di confrontare il valore di due elementi presenti all'interno del form che vengono identificati tramite la proprietà *name* della classe *FormStructField*, usualmente utilizzato per verificare se la password coincide con la password di conferma durante il processo di registrazione. Infine, è presente una proprietà *fromStruct*, un array di oggetti di classe *FormStrcutField* che rappresentano gli elementi costitutivi del form. Le proprietà della classe *FormStructField* permettono di specificare il nome (*name*) dell'elemento, la label (*label*) da associare all'elemento, la tipologia (*type*) di input che si desidera costruire basata sui tipi di input disponibili nell'HTML5, e la possibilità di mostrare un *password strength* (*meter*), ovvero un misuratore della robustezza della password inserita dall'utente che viene mostrato graficamente tramite l'ausilio di colori. Inoltre, nell'eventualità in cui si desideri un elemento differente da un normale *input,* si necessita di valorizzare o la proprietà *checkbox* o quella *select.* La prima trasforma l'elemento in una *checkbox* in cui è possibile associare un *link* da visualizzare al di sotto della checkbox; la seconda permette di trasformare l'elemento in una *select*, rappresentato da un array di oggetti *chiave-valore*, aventi come chiave il nome dell'elemento della select e come valore un oggetto avente come proprietà *icon*, ovvero l'identificativo univoco dell'icona da

associare all'elemento. Infine, è possibile valorizzare la proprietà *validator,* per associare un validatore all'elemento, abilitando i controlli sull'input dell'utente. Il valore è un oggetto di classe *Validator* e che permette di settare una *regex* (*regex*), che rappresenta i caratteri e la composizione che l'input può assumere, se il campo è obbligatorio (*required*) e il massimo (*max*) e il minimo (*min*) numero di caratteri che l'input può possedere.

3.3 Gestione dello Stato

Tutte le informazioni di stato dell'applicativo vengono memorizzate, grazie all'utilizzo di *vuex*, all'interno dello store che risulta, per motivi di maggior leggibilità del codice, suddiviso in moduli rappresentati sottoporzioni dello stato complessivo dato dall'unione dei moduli.

Ogni modulo, a sua volta, è scomposto nei seguenti file:

- un file relativo alle **actions**, ovvero funzioni Javascript con il compito di verificare se la richiesta avanzata dall'utente può essere risolta localmente o se necessita di un'ulteriore step dato da una chiama asincrona rivolta al backend. Al fine di effettuare la richiesta asincrona, le *actions* sfruttano lo strato dei servizi richiamando un opportuno metodo. Questa scelta è stata intrapresa in modo da non aggiungere logica dipendente dal backend all'interno dello store, operazione che avrebbe peggiorato la manutenibilità del codice;
- un file relativo ai **getters**, ovvero funzioni Javascript con il compito di effettuare operazioni di filtering, o semplicemente di retrieve, delle informazioni contenute all'interno dello store. Usualmente i *getters* vengono invocati dai componenti per ottenere una certa informazione o dalle *actions* per verificare se l'azione può essere risolta localmente;
- un file relativo alle **mutations**, ovvero funzioni Javascript con il compito di modificare lo stato contenuto all'interno dello store. Tali funzioni vengono esclusivamente invocate dalle *actions*. Si noti che la modifica dello store è attuabile esclusivamente dalle *mutations* e sarebbe un errore effettuarla al loro esterno: andrebbe contro il *pattern flux*;
- un file relativo all'**istanza del modulo** ove vengono dichiarate le varie componenti dell'oggetto che rappresenterà lo stato del modulo e l'associazione delle *actions*, *mutations* e *getters*, se presenti, con il modulo.

A seguire una descrizione dei moduli che compongono lo store, esplicitandone le funzionalità:

- **Account:** permette di memorizzare le informazioni relative all'utente tramite la classe *User*, prelevandole dal *localStorage,* se presenti, o impostandone il valore a *null* per indicare che l'utente non è autenticato;
- **Alert:** permette la gestione dei messaggi di errore o di successo che vengono emessi durante il ciclo di vita dall'applicativo. I messaggi possono essere prodotti dalla risposta dal backend, da una notifica proveniente da un topic MQTT o da errori sull'input dell'utente. Inoltre, permette di tenere traccia dell'inizio e della fine di una chiamata asincrona tramite una proprietà di stato *waiting*;
- **Configuration:** modulo di configurazione dell'applicativo, a causa dell'importanza vi sarà dedicato un capitolo (*3.3.1)*;

- **Devices:** permette di memorizzare le informazioni relative ai *devices* dell'utente tramite un array, avente come elementi un oggetto di classe *Device*. Inoltre, viene memorizzata anche una mappa che ha come chiave l'eui del *device* e come valore il suo *custom name*. Questa mappa verrà sfruttata per effettuare le conversioni *eui-name* o *name-eui* utili per mostrare all'utente un nome più chiaro per identificare il device rispetto ad un seriale esadecimale;
- **Events:** permette di memorizzare tutte le notifiche ricevute dai topic MQTT a cui l'applicativo è iscritto. La memorizzazione viene effettuata tramite un oggetto avente come proprietà il nome del topic e come valore un array di elementi che rappresentano le notifiche ricevute per il topic specificato dalla chiave;
- **GNSS e Telemetry:** permettono di memorizzare rispettivamente i dati GNSS e di telemetria di uno o più *device*. Entrambi contengono altre due informazioni circa l'intervallo temporale richiesto dall'utente per la visualizzazione delle informazioni, ovvero *startTimestamp* e *endTimestamp*. Tali informazioni vengono memorizzate per motivi di caching: quando l'utente richiederà un nuovo set di dati per uno o più *device* specificherà anche l'istante temporale per il quale è informato a visionare i dati, gli estremi dell'intervallo richiesto saranno confrontati con quelli memorizzati all'interno dello store per verificare se i dati presenti nello store sono validi per soddisfare la richiesta dell'utente o meno. Nel caso in cui l'intervallo temporale sia valido si verifica se tutti i devices richiesti sono presenti, condizione che permetterebbe di risolvere localmente la richieste senza generare una richiesta asincrona che aumenterebbe i tempi di risposta e sovraccaricherebbe il backend con una richiesta superflua. Si noti che, per un corretto funzionamento del sistema di caching, nell'eventualità in cui un *device* non abbia effettuato rilevazioni nell'intervallo temporale richiesto, si necessita di memorizzare ugualmente il *device* ma non associandovi alcun dato. La struttura memorizzata è quella discussa nel *capitolo 2.4* e rappresentata nella *figura 34*.

Inoltre, al fine di evitare ripetizioni di codice, è stato creato il file *utils* che verrà condiviso dai moduli dello store. Il file *utils* contiene un set di funzioni generiche per la gestione del fallimento di una richiesta, che comporta il logout forzato dall'applicativo qualora i token siano scaduti o la visualizzazione a schermo del messaggio di fallimento, e una funzione per la gestione del logout dello store a seguito del logout dell'utente, operazione che permetterà di cancellare tutte le informazioni contenute all'interno di ogni modulo che compone lo store e delle informazioni contenute all'interno del *localStorage*.

3.3.1 Modulo di Configurazione

Il modulo di configurazione viene utilizzato durante il ciclo di vita dell'applicazione al fine di reperire tutte le informazioni necessarie per la costruzione dei componenti e per la generazione delle richieste da inviare al backend. La potenzialità del modulo di configurazione è quella di offrire una grande customizzabilità delle funzionalità messe a disposizione dall'applicativo, nonché la possibilità di aggiungere, disattivare o rimuovere alcune caratteristiche del device come ad esempio le misure che devono essere analizzate, gli attributi o quali threshold possono essere settate dall'utente sul device. Le modifiche possono anche riguardare esclusivamente l'aspetto grafico, come ad esempio quale icona associare ad un attributo o ad una misura. Al fine

di descrivere al meglio tutte le funzionalità offerte andremo ad analizzare interamente il documento JSON che compone lo stato del modulo.

```
"providerEnum": {
   "provider": "enumerateValue"
},
"signatureEnum": {
   "signatureType": "enumerateValue"
},
```

Figura 38: Modulo di Configurazione 1/5

La *figura 40* mostra le due *enum* che verranno utilizzate delle *Abstract Factory* per la creazione delle istanze concrete delle classi per la gestione della generazione delle richieste e del servizio di autenticazione dell'utente tramite l'utilizzo di uno *switch* sui valori posseduti dalle enum, le *abstract factory* saranno approfondite nel *capitolo 3.5*. L'enum *providerEnum* è un oggetto avente come proprietà l'identificativo del provider (ad esempio AWS o Google) e come valore un intero incrementale. L'enum *signatureEnum* è un oggetto avente come proprietà l'identificativo della signature (ad esempio AWS_SIGNATURE_V4) e come valore un intero incrementale. Le enum permettono di specificare la tipologia di *backend* messi a disposizione e quali *signature* possono essere gestite.

```
"signatures": {
   "signatureType": { ... }
},
"backend": {
   "provider": {
      ...
      "dataConverter": {
         "nameServer": "nameClient"
      },
      "dataSerialize": {
         "nameClient": "nameServer"
      },
      "methodConverter": {
         "API-Name": {
            "httpMethod": "methodSerializer/methodDeserializer"
         }
      }
   }
},
```

Figura 39: Modulo di Configurazione 2/5

La *figura 41* mostra le due proprietà contenti rispettivamente le informazioni necessarie alla costruzione della firma sulle richiesta e la gestione delle richieste e delle risposte provenienti dal backend. La proprietà *signatures* è un oggetto complesso avente come proprietà il nome delle chiavi assegnate alla corrispettiva enum (*signatureEnum*) e come valore l'insieme di informazioni necessarie per la costruzione della firma sulla richiesta, come ad esempio i vari algoritmi utilizzati per la procedura di firma. La proprietà *backend* è un oggetto complesso avente come proprietà il nome delle chiavi assegnate alla corrispettiva enum (*providerEnum*) e come valore l'insieme di informazioni necessarie alla costruzione della richiesta da inoltrare al backend e tre metodi di maggior rilievo che devono essere presenti indipendentemente dal provider selezionato:

- **dataConverter:** permette di mappare il nome assegnato dal server ad una misura rilevata dall'end-device con il nome da assegnare all'interno del *Front End*. Viene

utilizzato ogni qualvolta devono essere deserializzate informazioni contenenti misure al fine di adattare i contenuti al formato richiesto dall'applicativo;

- **dataSerializer:** permette di mappare il nome assegnato dal *Front End* ad una misura rilevata dall'end-device con il nome assegnato dal backend. Viene utilizzato ogni qualvolta una misura deve essere inviata, all'interno del payload, al backend al fine di adattare il formato utilizzato nell'applicativo a quello atteso dal backend;
- **methodConverter:** proprietà che permette di mappare una API REST esposta dal backend ad un metodo da richiamare nel corrispettivo serializzatore, nel caso in cui la richiesta abbia un payload, e deserializzatore, quando viene elaborata la risposta inviata dal backend. *Api-Name* rappresenta il path dell'API REST, *httpMethod* è il metodo HTTP utilizzato per contattare l'API e *methodSerializer/methodDeserializer* rappresenta il nome del metodo da invocare per effettuare la trasformazione del payload in entrata e/o in uscita. Questa proprietà riveste un ruolo rilevante poiché permette di astrarre e rendere indipendente l'utilizzo dei serializzatori e dei deserializzatori quando viene creata la richiesta o elaborata la risposta in modo da ridurre al minimo le linee di codice ed aumentando notevolmente la leggibilità e la manutenibilità della soluzione.

```
"leaflet": { ... },
"profile": {
   "documentation": [
      {
         "name": "string",
         "href": "url"
      }
   ]
},
```

Figura 40: Modulo di Configurazione 3/5

La *figura 42* mostra le proprietà *leaflet* e *profile*. La proprietà *leaflet* ha il compito di memorizzare le configurazione di default da assegnare al *wrapper leaflet* nell'eventualità in cui non siano presenti informazioni GNSS su un dato device, permettendo una configurazione corretta del wrapper e una modifica delle informazioni di default senza la necessità di agire direttamente sul codice sorgente. La proprietà *profile* permette di aggiungere nuove documentazioni da visualizzare nel componente *Profile* al fine di fornire all'utente delle linee guida sull'utilizzo dei servizi offerti dall'applicativo, nonché degli approfondimenti su temi considerati maggiormente rilevanti. La proprietà è costituita da un oggetto complesso avente come unica proprietà *documentation*, un array di oggetti aventi un *name* che rappresenta il nome logico da mostrare all'utente e *href* che rappresenta il link alla documentazione.

La *figura 43* mostra la proprietà principale, ovvero il *device*. Le informazioni contenute all'interno della proprietà *device* permettono la costruzione di gran parte dell'applicazione grazie alla presenza di tutte le informazioni che devono essere elaborate per rappresentare graficamente le misure rilevate, per la gestione della pagina relativa alle informazioni del singolo device fino

Figura 41: Modulo di Configurazione 4/5

alle thresholds che l'utente può settare per uno specifico device. Andiamo ad esplicitare il senso e l'utilizzo di ognuna delle proprietà:

- **Measures:** oggetto complesso avente come proprietà i nomi univoci della misure rilevate dai device e come valore un oggetto avente come proprietà l'icona associata alla misura (*icon*), l'unità di misura (*units*) e lo stato della misura all'interno dell'applicativo (*active*). La proprietà *active* permette di abilitare o disabilitare la visualizzazione e l'elaborazione della misura, dando la possibilità allo sviluppatore di rimuovere o aggiungere una nuova misura semplicemente modificando il valore del booleano, ponendolo a *false* o aggiungendo una nuova proprietà nel caso in cui la misura non sia contenuta di default all'interno della proprietà;
- **Models:** oggetto avente come proprietà il nome univoco della rete a cui l'end-device si collega e come valore l'identificativo univoco dell'icona associata. Tale proprietà viene utilizzata, in particolare, in fase di registrazione di un nuovo *device*;
- **Attributes:** oggetto complesso avente come proprietà i nomi univoci degli attributi assegnati dal backend al device e come valore un oggetto avente come proprietà l'icona associata all'attributo (*icon*), lo stato dell'attributo (*active*) e l'espandibilità dell'attributo (*expand*). Come nel caso della proprietà *measures*, la proprietà *active* permette di attivare o disattivare la visualizzazione di un attributo, mentre la proprietà *expand* viene utilizzata nel caso in cui il valore contenuto all'interno dell'attributo abbia una lunghezza non trascurabile (ad esempio la chiave pubblica associata ad un device): se viene valorizzato a *true* il valore dell'attributo sarà mostrato all'interno del tag HTML5 *details* in modo da migliorarne la visualizzazione e sarà anche possibile copiarne il contenuto tramite il semplice click;
- **Thresholds:** oggetto complesso avente come proprietà i nomi univoci delle thresholds che possono essere settate e come valore un oggetto avente come proprietà l'icona associata alla threshold (*icon*), lo stato della threshold (*active*), il massimo valore (*max*) e il minimo valore (*min*) che possono essere settati come threshold. Come nei casi

precedenti, la proprietà *active* permette di abilitare o disabilitare l'utilizzo della threshold;

- **Certificate:** oggetto contenente le proprietà *active,* al fine di abilitare la gestione e visualizzazione del certificato all'interno dell'applicativo, *certificateStatus,* contenente l'insieme di stati del certificato, e *icons.* Quest'ultimo è un oggetto avente come proprietà i nomi univoci degli stati contenuti all'interno dell'array *certificateStatus* e come valore un oggetto con un'unica proprietà *icon,* contenente l'identificativo univoco dell'icona da associare allo stato del certificato.

La struttura del documento JSON, associato al modulo di configurazione, permette l'abilitazione o disabilitazione della maggior parte delle informazioni associate al device, in particolar modo la possibilità di disabilitare l'intera sezione riguardante il certificato, sezione comunque resa opzionale grazie alla possibilità di non associare il certificato al device al momento della registrazione. Inoltre, come visto, si dà la possibilità di modificare icone, unità di misura e nomi degli attributi, delle soglie e delle misura senza dover modificare il codice, operazione che avrebbe largamente rallentano l'implementazione di un nuovo dettaglio correlato al device. Al fine di ottenere questo risultato, il codice relativo ai componenti che si occupano di gestire le informazioni sul device sono resi indipendenti dalla tipologia di informazioni che si intende mostrare ma necessitano esclusivamente di conoscere la struttura dati da elaborare, riducendo il codice boilerplate che sarebbe stato introdotto dalla non generalizzazione delle informazioni sul device. Inoltre, tutte le modifiche vengono esclusivamente effettuate all'interno del file JSON e mai all'interno del codice.

```
"sidebar": {
   "width": "200px",
   "widthState": [
      "200px",
      "50px"
   ],
   "menuUnlogged": [
      {
         "header": "boolean",
         "title": "string",
         "visibleOnCollapse": "boolean",
         "href": "string",
         "icon": "string"
      }
   ],
   "menuLogged": [
      {
         "header": "boolean",
         "title": "string",
         "visibleOnCollapse": "boolean",
         "href": "string",
         "icon": "string"
      }
   ]
}
```

Figura 42: Modulo di Configurazione 5/5

La *figura 44* mostra l'ultima proprietà contenuta all'interno del documento, ovvero *sidebar*. La proprietà *sidebar* contiene tutte le informazioni necessarie ad una corretta costruzione della sidebar, ovvero le dimensioni che dovrà aver nella versione espansa e compressa, contenute all'interno della proprietà *widthState*, nonché la dimensione attuale contenuta all'interno della proprietà *width*. Inoltre, anche le informazioni che debbono essere visualizzate al suo interno sono configurabile tramite la proprietà *menuUnlogged* nel caso in cui l'utente non abbia effettuato l'autenticazione, e *menuLogged*, in caso contrario. Entrambe le proprietà contengono lo stesso set di proprietà: *header,* permette di specificare se l'elemento è l'header della sidebar o meno, *title* rappresenta il nome dell'elemento che sarà visualizzato, *visibleOnCollapse* permette di specificare se l'elemento deve essere mostrato nella versione compatta della

sidebar, *href* rappresenta il collegamento con la vista che sarà invocata (tramite il meccanismo del routing) a seguito del click dell'utente sull'elemento, e *icon* è l'identificativo univoco dell'icona che si intende associare all'elemento. Anche in questo caso, il contenuto della sidebar è completamente customizzabile effettuando le modifiche esclusivamente all'interno del documento JSON, senza necessitare di modifiche al codice sorgente.

3.4 Single Page Application – Routing

L'applicazione è stata costruita come *single page application*, ovvero la navigazione dell'utente attraverso viste differenti non richiede il caricamento dell'interno DOM ma viene effettuata una modifica del contenuto della pagina al fine di inserire i nuovi contenuti richiesti dall'utente, ottenendo delle performance notevolmente superiori rispetto alla navigazione a pagina singola, dove ad ogni richiesta viene generata una nuova vista. La gestione della storia di navigazione dell'utente e il caricamento dinamico dei nuovi contenuti è stato realizzato grazie all'utilizzo del *vue-router*: viene effettuato un mapping tra i componenti e le *routes*.

```
{
   "path": "/devices",
   "name": "Devices",
   "component": "Devices",
   "children": [
      {
         "path": ":eui",
         "name": "DeviceDetails",
         "component": "DeviceDetails"
      }
   ]
}
```

Figura 43: Elemento di Routing

La *figura 45* mostra un elemento dell'array di *routes* che andrà a comporre il *router*. La peculiarità dell'elemento è la presenza della proprietà *children,* che indica la presenza di una relazione *padre-figlio* tra il componente *Devices* e il componente *DeviceDetails*. Grazie alla relazione è possibile comunicare al modulo *vue-router* che se viene matchato un path del tipo:

$$/devices/BE7A00000000000A$$

Dovrà essere renderizzato il componente *Devices* e successivamente, al suo interno, il componente *DeviceDetails* a cui saranno passate, dal componente padre, le informazioni relative al device da mostrare all'interno della vista. Ciò permette di non dover renderizzare nuovamente il componente *Devices* nell'eventualità in cui si torni indietro dalla vista dei dettagli del device alla vista generica, aumentando la fluidità dell'applicazione ed evitando inutili render.

3.4.1 Router Guard

La navigazione all'interno della single page application è protetta tramite l'utilizzo di un *navigation guard* offerto dal modulo *vue-router*. Esso permette di verificare se l'utente ha già effettuato il processo di autenticazione, caso in cui può accedere a tutte le viste dell'applicazione ad esclusione della vista per effettuare la registrazione e il login o meno, caso in cui può accedere a una sezione limitata di viste. Al fine di verificare se l'utente ha già effettuata l'autenticazione o meno, si analizza il modulo dello store *account*, verificando se la proprietà *user* dello stato ha un valore differente da *null*, valore di default nel caso in cui l'utente non abbia effettuato l'autenticazione. Nel caso in cui l'utente tenti di accedere ad una pagina che richiede di aver già effettuato il processo di autenticazione, partirà un redirect automatico alla vista di login. Il

controllo descritto viene eseguito ogni qualvolta viene rilevato un cambiamento nella URL, di conseguenza prima di qualunque processo di rendering del componente associato alla nuova URI.

3.5 Abstract Factory Design Pattern

Lo scopo dell'**Abstract Factory** è quello di fornire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti tra loro senza dover specificare le loro classi concerete limitandone, di conseguenza, l'uso diretto. All'interno del progetto è stato sfruttato al fine di creare le opportune estensioni delle classi *AuthenticationService*, *SignatureRequest* e *RequestGenerator* durante l'avvio dell'applicazione effettuandone l'*injection* all'interno dell'istanza di *Vue* in modo da essere disponibili all'interno di ogni componente.

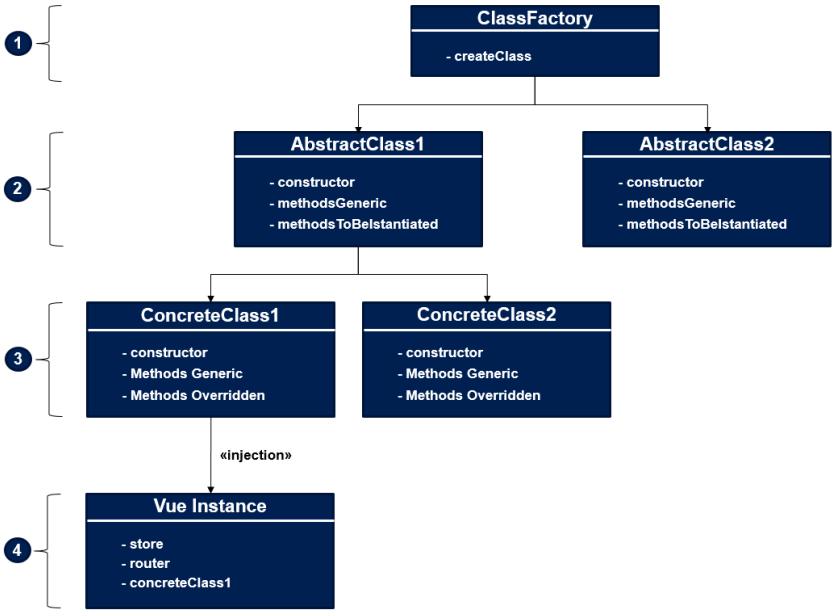


Figura 44: Abstract Factor Pattern

La *figura 46* mostra la struttura utilizzata per la costruzione del pattern *Abstract Factory* all'interno del *framework Vue.js*; a seguire una descrizione dei vari step:

1. Viene costruita una classe rappresentante la factory per le classi astratte. Al suo interno è presente un unico metodo che permette la creazione dell'istanza concreta passando come parametro il valore dell'enum, presente nel modulo di configurazione dello stato, corrispondente alla factory: il valore dell'enum viene analizzato tramite uno *switch case*, che una volta individuato il valore corrispondente procederà a creare la corretta istanza.
2. Viene definita una classe padre che rappresenta l'istanza astratta. La classe possiede i metodi che dovranno essere implementati nella classe *figlia*, ovvero l'istanza concreta; nel caso in cui tali metodi non verranno implementati, alla loro invocazione sarà lanciata un'eccezione. Con questa struttura si fa sì che qualsiasi classe concreta esponga sempre gli stessi metodi, garantendo una corretta integrazione della classe concreta sviluppata con l'applicativo già esistente. Questo permette ai servizi che andranno ad invocare i metodi di non conoscere nulla sull'implementazione concreta dei metodi, ma di conoscere esclusivamente i metodi esposti, garantendo un'adesione al principio della separazione delle responsabilità.

3. Vengono definite una o più classi concrete che estendono la classe astratta. Nella classe vengono implementati i metodi della classe astratta ed eventuali ulteriori metodi di supporto per le operazioni esposte, in tal senso non sono imposti vincoli. Nel costruttore della classe concreta viene invocato il costruttore della classe astratta, passando eventuali configurazioni necessarie per disporre di metodi globali definiti nell'istanza generica. Inoltre, le classi concrete utilizzato il *pattern Singleton*, che permette di assicurarsi che una classe abbia una sola istanza e di fornire un punto di accesso globale ad essa.
4. L'istanza concreta prodotta dalla factory viene iniettata all'interno dell'istanza di Vue.js.

L'adesione al pattern *Abstract Factory* permette di far apparire la classe concreta esclusivamente una volta nel codice, ovvero quando viene istanziata all'interno della factory, che è l'unico oggetto ad averne il controllo. Questo permette, oltre a semplificare il processo di cooperazione tra oggetti della stessa famiglia, di richiedere esclusivamente la modifica della factory al fine di aggiungere una nuova specializzazione della classe astratta o modificare un collegamento con una classe concreta già presente nel sistema.

Dopo questo excursus sull'implementazione del pattern andremo ad analizzare nello specifico le *AbstractClass* e le *ConcreteClass* utilizzate.

3.5.1 Authentication Service

La classe **AuthenticationService** rappresenta il servizio astratto per effettuare l'autenticazione, la registrazione, il logout dell'utente e il refresh del token. Al fine di creare un'istanza concreta di tale classe viene utilizzata la *Authentication Factory*.

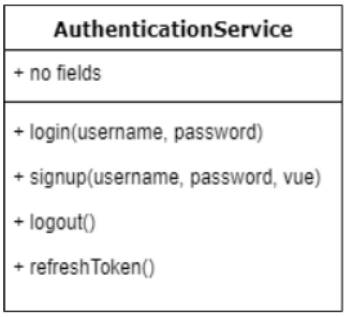


Figura 45: Authentication Service

La *figura 47* mostra la classe ove non sono presenti proprietà all'interno e quattro metodi che devono essere implementati all'interno della classe concreta, la cui descrizione segue:

- $login(username, password)$: metodo utilizzato per effettuare il processo di autenticazione con il backend. A seguito della procedura di autenticazione, se andato a buon fine, il metodo deve ritornare un oggetto di classe *User*, riempendo la proprietà *loginData* con tutte le informazioni sulle credenziali fornite dall'*authentication provider* per identificare la sessione dell'utente.
- $signup(username, password, vue)$: metodo utilizzato per avviare il processo di registrazione. L'oggetto *vue* rappresenta il contesto del componente che ha richiesto di avviare il processo di registrazione; questo risulta necessario al fine di invocare l'istanza concreta della classe *RequestGenerator* per eseguire la seconda fase del processo di registrazione.

- $logout()$: metodo utilizzato per avviare il processo di logout dell'utente.
- $refreshToken()$: metodo che permette di verificare se il token utilizzato è scaduto, caso in cui effettuerà il processo di *refresh* delle credenziali temporanee. Il metodo ritornerà un nuovo oggetto di classe *User* o un errore in caso di fallimento.

In tutti i metodi, l'oggetto ritornato dovrà essere una *promise* per permettere al chiamante di attendere l'esito dell'operazione.

Una **promise** rappresenta un'operazione che non è ancora completata, ma lo sarà in futuro. Consente di associare degli handlers di successo o fallimento a un'azione asincrona consentendo di utilizzare i metodi asincroni come se fossero sincroni: la funzione asincrona non ritorna un valore ma una *promise*, tramite la quale si potrà ottenere il valore una volta che l'operazione sarà terminata. Quando una *promise* evolve da *pending* a *fulfilled* o *rejected* vengono chiamati gli handler associati che sono stati accodati dal metodo *then* della *promise* in caso di successo, o a *catch* in caso di fallimento.

All'interno dell'applicazione è stata prodotta solo una *classe concreta*, ovvero l'*AuthenticationAWS*.

AuthenticationAWS

Classe concreta della classe astratta *AuthenticationService*, che permette di effettuare l'override dei metodi definiti nella classe astratta al fine di permetterne l'esecuzione sul provider AWS. L'aspetto più interessante è l'utilizzo degli *sdk* forniti da AWS al fine di effettuare le operazioni di registrazione, autenticazione e logout dell'utente, dove quest'ultimo permette di invalidare i token, e di refresh dei token. Vengono sfruttati gli *sdk* AWS al fine di contattare direttamente il *Cognito User Pool* e il *Cognito Identity Pool* piuttosto che contattarli tramite l'API Gateway.

3.5.2 Request Generator

La classe **RequestGenerator** rappresenta il servizio astratto per effettuare la generazione delle richieste HTTP e MQTT. Al fine di creare un'istanza concreta di tale classe viene utilizzata la *Request Generator Factory*.

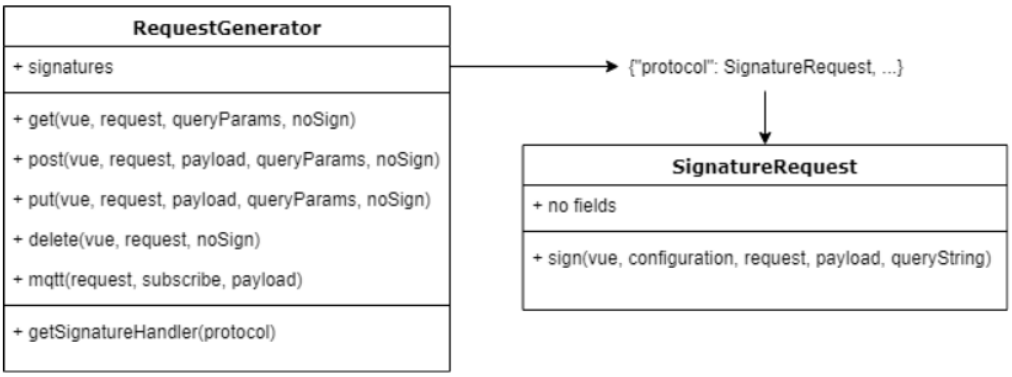


Figura 46: Request Generator

La *figura 48* mostra la classe ove è presente un solo attributo di classe, *signatures* che permette, in fase di costruzione della classe concreta, di iniettare all'interno dell'istanza un oggetto avente come proprietà il nome del protocollo e come valore l'istanza concreta della classe *SignatureRequest*, costruita anch'essa tramite l'utilizzo di una factory, precisamente la *Signature Request Factory*. Questo attributo viene inserito e valorizzato all'interno della classe astratta

RequestGenerator, poiché indipendente dalla tipologia di implementazione della classe concreta. La classe possiede i seguenti metodi su cui dovrà essere effettuato l'override nella classe concreta:

- $get(vue, request, queryParams, noSign)$: metodo utilizzato per richiedere la generazione di una richiesta HTTP con metodo GET;
- $post(vue, request, payload, queryParams, noSign)$: metodo utilizzato per richiedere la generazione di una richiesta HTTP con metodo POST;
- $put(vue, request, payload, queryParams, noSign)$: metodo utilizzato per richiedere la generazione di una richiesta HTTP con metodo PUT;
- $delete(vue, request, noSign)$: metodo utilizzato per richiedere la generazione di una richiesta HTTP con metodo DELETE;

Vue rappresenta il contesto del componente che ha richiesto la generazione della richiesta, *request* è un oggetto avente come proprietà *path* che indica il path relativo dell'API da contattare, *payload* rappresenta un eventuale payload che dovrà essere inserito all'interno della richiesta, *queryParams* rappresentano eventuali parametri che dovranno essere aggiunti alla richiesta e *noSign* permette di specificare se è necessario firmare la richiesta o meno.

```
{
   "protocol": "HTTP",
   "request": {
      "method": "httpMethod",
      "host": "string",
      "path": "string",
      "headers": "object"
   },
   "payload": "string",
   "deserializer": {
      "object": "deserializerClass",
      "method": "this.methodConverter[request.path.split('/')[0]][apiMethod]"
   }
}
```

Figura 47: Output dei metodi HTTP

L'output dei metodi ha la struttura rappresentata nella *figura 49*. Nello specifico, il campo *protocol* specifica quale protocollo è stato utilizzato per generare la richiesta, *request* è un oggetto contenente tutte le informazioni che devono essere correlate alla richiesta, in particolar modo gli headers che rappresentano le informazioni sulla signature se presente, *payload* rappresenta l'eventuale payload da agganciare alla richiesta e *deserializer* contiene una proprietà *object*, ovvero l'istanza del deserializzatore che si deve utilizzare una volta ricevuta la risposta dal backend, e una *method*, ovvero il metodo che dovrà essere invocato dall'istanza del deserializzatore. Come già spiegato nel *capitolo 3.3.1*, viene sfruttato il file di configurazione al fine di effettuare un mapping tra il metodo HTTP e il path utilizzato per la richiesta al fine di ottenere quale metodo invocare sul deserializzatore, permettendo di ottenere un mapping del tutto automatico. L'oggetto cosiffatto sarà utilizzato dal *Network Service* per inviare la richiesta al backend.

- $mqtt(request, subscribe, payload)$: metodo utilizzato per richiedere la *sottoscrizione* a un topic MQTT e/o la pubblicazione di un nuovo payload al suo interno.

R*equest* rappresenta un oggetto con un'unica proprietà *topic* che rappresenta il path a cui bisogna far riferimento, *subscribe* è un booleano che indica se bisogna effettuare la sottoscrizione al topic MQTT specificato nella *request* e *payload* rappresenta il payload da pubblicare all'interno del topic MQTT.

```
{
    "protocol": "MQTT",
    "mqttClient": "object",
    "subscribe": "boolean",
    "request": "string",
    "payload": "string"
}
```

Figura 48: Output del metodo MQTT

L'output atteso dal metodo viene rappresentato nella *figura 50*: il campo *protocol* specifica quale protocollo è stato utilizzato per costruire la richiesta, *mqttClient* rappresenta l'istanza del client MQTT, *subscribe* la volontà di voler effettuare la subscribe al topic MQTT specificato o meno, *request* rappresenta l'identificativo del topic MQTT a cui è diretta la richiesta e *payload* rappresenta l'eventuale topic da aggiungere all'interno del topic MQTT.

L'ultimo metodo, ovvero $getSignatureHandler(protocol)$ non necessita di override ma viene invocato dalla classe padre passando come parametro il protocollo per cui si vuole ottenere l'istanza corretta della classe *SignatureRequest*. Questo offre la possibilità di assegnare più gestore, e quindi tipologie di firme, differenti in base al protocollo scelto.

All'interno dell'applicativo prodotto è stata prevista solo una *classe concreta*, il *RequestGeneratorAWS,* a cui viene associato un unico handler delle firma, cioè *SignatureRequestAWSV4,* relativo al protocollo HTTP.

RequestGeneratorAWS

Al fine di gestire l'override dei metodi *post*, *put*, *get* e *delete* è stato creato un metodo privato *handlerHTTPRequest* che viene chiamato in tutti e quattro i metodi ma modificandone i parametri passati, in modo da discriminare il metodo HTTP usato e viene anche applicato il *serializzatore* se presente un payload.

Per quanto concerne il metodo *mqtt* si è deciso di creare il *client MQTT* alla invocazione del metodo e successivamente viene memorizzato come attributo di classe. Questo permette di mantenere tutte le sottoscrizioni effettuate durante l'attività svolta sull'applicazione. Inoltre, ad esso viene associato una callback per la gestione dei messaggi MQTT provenienti dai topic su cui è stata effettuata la sottoscrizione: alla ricezione di una nuova notifica verrà invocato il deserializzatore e successivamente sarà effettuato il *dispatch* della *action* incaricata della gestione della ricezione delle notifiche all'interno del modulo *events* dello store. Nel caso di invocazioni successive del metodo *mqtt*, verrà effettuato l'aggiornamento delle credenziali associate al client MQTT.

3.6 Network Service

La classe *Network Service* è una classe Singleton in cui viene iniettato il modulo *https* a seguito della creazione dell'istanza. Permette di interfacciarsi con i networks esterni effettuando la generazione di una richiesta HTTP o effettuando la *subscribe* e/o la *publish* in un topic MQTT. Le

informazioni necessarie alla creazione della richiesta vera e propria vengono fornite dall'output di uno dei metodi dell'istanza concreta del *RequestGenerator*.

Al fine di annegare la logica all'interno della classe, viene esposto un unico metodo $handlerRequest(request)$, dove la struttura di *request* viene mostrata nelle *figura 49* e nella *figura 50*, che in base al *request.protocol* passato come parametro, effettuerà l'opportuna chiamata al metodo privato corrispondente. Ciò permette, ancora una volta, di implementare liberamente i metodi all'interno della classe *NetworkService* senza necessitare che i servizi conoscano la tipologia di protocolli implementati. In fase di manutenzione dell'applicativo, di espansione di un nuovo protocollo su cui veicolare le richieste o in alternativa in caso di modificare del mezzo utilizzato per instradare la richiesta, sarà possibile esclusivamente operare sulla classe *NetworkService*, garantendo una velocizzazione delle attività citate.

Nel caso del **protocollo HTTP** sarà invocato il metodo privato $\_httpsModule(request)$ che effettuerà la richiesta HTTP con i parametri specificati da *request* ed attenderà una risposta di successo o di fallimento da parte del backend. Al fine di gestire al meglio la risposta ricevuta dal backend è stato analizzato il codice di risposta:

- Se il codice HTTP di risposta è pari a *200* la risposta sarà considerata valida ed elaborata. Una volta ricevuta per intero sarà invocato il metodo appropriato del *deserializzatore*, passato come parametro nella richiesta, e la risposta correttamente formattata sarà ritornata al chiamante.
- Se il codice HTTP di risposta è pari a *403*, la risposta sarà considerata di errore a causa dei *token* scaduti e di conseguenza si ritornerà come messaggio d'errore *"403"*, che sarà elaborato dagli handler degli errori al fine di procedere con il logout forzato dell'utente dall'applicativo.
- Se il codice HTTP di risposta è uguale o maggiore a *500* il messaggio contenuto sarà considerato di errore causato da un problema avvenuto durante l'elaborazione della richiesta e di conseguenza sarà tornato l'errore generico *"Internal Server Error"*.
- Se il codice HTTP di risposta non rientra nei casi precedentemente descritti, la risposta rivenuta sarà elaborata e considerata d'errore a causa di un errore nella richiesta inviata dall'utente o un errore di rete. Il messaggio contenuto all'interno del payload sarà ritornato all'utente come messaggio d'errore.

In tutti i casi, il valore ritornato dalla metodo sarà wrappato all'interno di una *promise* in modo che il chiamante possa attendere l'esito della chiamata asincrona.

Nel caso del **protocollo MQTT** sarà invocato il metodo privato $\_mqttModule(request)$ il quale non avrà un valore di ritorno poiché controllerà esclusivamente se il booleano *subscribe* contenuto nella *request* assume un valore positivo, caso in cui effettuerà la subscribe al topic MQTT specificato, e se il *payload* contenuto nella *request* è diverso da *null*, caso in cui effettuerà la *publish* sul topic MQTT specificato.

3.7 Componenti Generici

Dopo aver analizzato l'intera struttura dell'applicativo concentriamoci sull'analisi dei *wrapper* realizzati al fine di erogare le *features* che saranno analizzate nel *capitolo 3.8*. Lo scopo dei wrapper è quello di garantire una massima riutilizzabilità in svariate situazioni differenti,

garantendo però una grande customizzazione dell'input tramite l'utilizzo di *props* articolate che permettono di specificare come il wrapper dovrà essere costruito. Ciò permette di aumentare leggibilità, manutenibilità ed estendibilità della soluzione proposta.

3.7.1 Wrapper Select

Figura 49: Wrapper Select

Il **wrapper select** permette di creare una *select*, basata su *Vuetify*, ma aggiungendo l'icona al nome dell'elemento da visualizzare, come mostrato nella *figura 51*.

Le **props** richieste dal wrapper sono:

- *setIcons:* array di icone da visualizzare nella select. È un oggetto che ha come proprietà il nome dell'elemento e come valore un oggetto che ha una proprietà *icon* avente come valore l'identificativo univoco dell'icona.
- *elements:* array di elementi da mostrare all'interno della select.
- *default:* l'elemento che deve essere selezionato di default.
- *label:* stringa che rappresenta la label da associare alla select.
- *dark:* applica il tema *dark*, modificando i colori, se il valore è *true*.
- *name:* identificativo univoco da associare al wrapper.

Inoltre, il wrapper espone un evento *newSelection* che viene attivato quando l'elemento selezionato cambia. Il payload dell'evento sarà il nuovo elemento selezionato e l'identificativo univoco del wrapper (*name*).

3.7.2 Wrapper Form

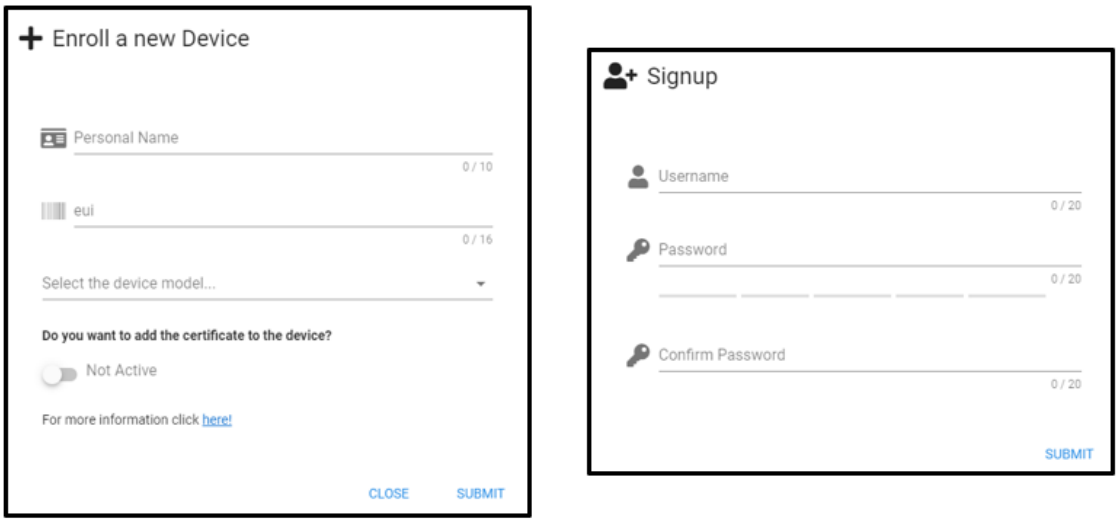


Figura 50: Wrapper Form

Il **wrapper form** permette di creare un form di qualunque tipologia passando esclusivamente le tipologie di elementi da costruire. Questo permette di concentrare tutta la logica relativa alla costruzione dei form all'interno di un unico componente, offrendo una grande riutilizzabilità in svariate situazioni come è possibile vedere nella *figura 52*. In figura il *wrapper* viene sia utilizzato per la costruzione di un *form per la registrazione di un nuovo device* sia per il *form di registrazione*, ove vengono richieste informazioni molto differenti tra loro.

I componenti che possono essere costruiti sono un normale input, in cui è possibile specificare controlli quali le regex o dimensione della stringa inserita, oltre alla funzionalità base HTML5, ovvero il *type*. Inoltre, è possibile costruire un *wrapper select* (*capitolo 3.7.1*) se viene valorizzata la proprietà *select* dall'oggetto di classe *FormStructField,* che rappresenta l'elemento del form, una *checkbox*, graficamente rappresentata tramite *switch* se viene valorizzata la proprietà *checkbox* dell'oggetto di classe *FormStructField* e un misuratore della robustezza della password inserita se viene posto a *true* la proprietà *meter* dell'oggetto di classe *FormStructField*. Inoltre, è possibile selezionare uno solo degli elementi specificati alla volta, di conseguenza le altre proprietà non dovranno essere valorizzate.

La possibilità di associare un validatore all'input dell'utente permette al wrapper di emettere l'evento *submitForm* esclusivamente quando l'input inserito dall'utente è valido, risolvendo localmente i controlli sulla sua validità.

Le **props** richieste dal wrapper sono:

- *form:* Un oggetto di classe *FormPayload*, discusso nel *capitolo 3.2.4*.

Inoltre, il wrapper espone due eventi:

- *rejectForm:* indica al componente padre che il dialog è stato chiuso e il wrapper sta procedendo con la distruzione. Tale evento non viene mai emesso se la proprietà block dell'oggetto *FormPayload* è posta a *true*.
- *submitForm:* ritorna un oggetto rappresentante l'input dell'utente. L'oggetto ritornato ha come proprietà i nomi degli elementi inseriti (proprietà *name* degli elementi *FormStructField*) e come valore l'input dell'utente.

3.7.3 Wrapper Carousel

Figura 51: Wrapper Carousel

Il **wrapper carousel** permette la creazione di un sistema di selezione dei device ove vengono mostrare le specifiche dei device al fine di velocizzare il processo di selezione.

Il wrapper offre la possibilità di selezionare uno o più device tramite l'utilizzo di una select o del carousel, dove è presente una checkbox. In entrambe le modalità vengono visualizzate le misure rilevate dal device, utile in particolare nella selezione dei device per mostrare graficamente i dati di telemetria rivenuti, monitorando quali misure sono disponibili per uno specifico device.

All'interno del carousel vengono anche mostrati il modello di rete a cui il device si collega e lo status del device, in termini di certificato e di stato riportato all'interno del *desired* dello shadow.

Al fine di permettere una maggiore suddivisione dei compiti, il *wrapper carousel* utilizza un ulteriore *wrapper* al suo interno che è dato dal *CarouselItem*. Questo wrapper costituisce l'elemento del carousel e permette al componente padre di ricevere eventi circa la selezione o deselezione di un elemento (rispettivamente *removeItem* e *addItem*). Le *props* attese dal wrapper *CarouselItem* sono *configuration*, ovvero il modulo di configurazione dello store al fine di abilitare la costruzione dinamica delle informazioni relative al device, ad esempio quali misure sono abilitate all'interno dell'applicativo o se il certificato è abilitato, il *device* da mostrare, che deve essere di classe *Device*, e il *model* che rappresenta un array avente come valori i nomi dei *device* che già sono stati selezionati. Questo permette di poter modificare il contenuto del *carousel*, o implementare una nuova tipologia di contenuto, senza dover operare sul *wrapper carousel*, migliorando la logica di separazione delle responsabilità a cui sono soggetti i componenti realizzati, aumentandone anche la riutilizzabilità.

Le **props** richieste dal wrapper sono le seguenti:

- *configuration*: oggetto che rappresenta il modulo di configurazione dello store. Viene utilizzato per la costruzione della select associata al carousel e per passarlo al wrapper interno.
- *devices:* Array di oggetti di classe *Device*. Rappresenta l'array di device che devono essere visualizzati all'interno della select e del carousel.

Il wrapper espone un unico evento *changeModel* che porta come payload l'array dei nomi dei device che sono stati selezionati tramite select o carousel. Il vantaggio principale nell'utilizzo del wrapper è la possibilità di gestire, con un unico evento, il cambiamento della lista di device selezionati indipendentemente dal mezzo utilizzato dall'utente per effettuare la selezione: la selezione può essere effettuata tramite select o carousel ma in ogni caso l'utilizzatore del wrapper riceverà nota, all'interno di un solo evento, del nuovo array di device selezionati, annegando all'interno del wrapper il tedioso lavoro di gestione delle selezioni. Infine, viene offerta la possibilità di selezionare o deselezionare tutti i device tramite un apposito pulsante presente all'interno della select.

3.7.4 Wrapper Datetimepicker

Il **wrapper datetimepicker** permette la selezione di un intervallo temporale, definito da *start date* e *end date*, e l'operazione di sincronizzazione con l'istante temporale attuale, che andrà ad aggiornare l'*end date*. Il wrapper è stato realizzato sfruttando il componente *vue-datetime* [13] che permette l'apertura di un *dialog* per la selezione dell'anno, mese e giorno tramite una prima interfaccia di selezione e successivamente di ore e minuti in una successiva interfaccia. La sua rappresentazione grafica è mostrata nella *figura 54*.

Il componente, grazie alla sua semplicità, non necessita di alcuna *props* ed espone i seguenti eventi:

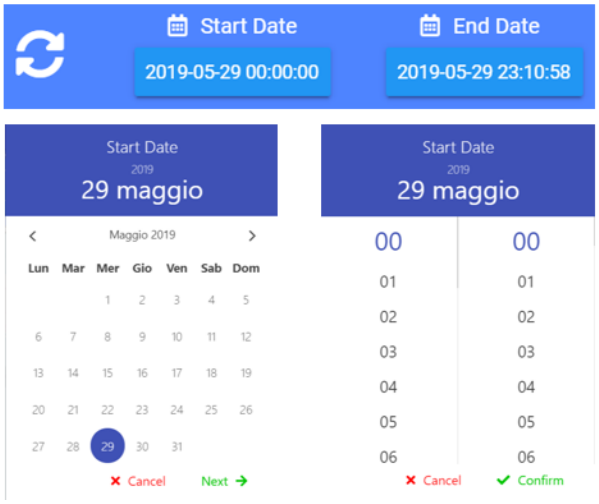


Figura 52: Wrapper Datetimepicker

- *changeStartDate:* permette di segnalare al componente padre un cambiamento sull'istante temporale iniziale. Il payload dell'evento rappresenta la nuova data di inizio intervallo selezionata.
- *changeEndDate:* permette di segnalare al componente padre un cambiamento nell'istante temporale finale. Il payload dell'evento rappresenta la nuova date di fine intervallo selezionata.
- *error:* permette di segnalare al componente padre che è stato rilevato un input errato da parte dell'utente. Il payload dell'evento è una stringa contenente i dettagli dell'errore. Un possibile errore è la selezione di un intervallo temporale in cui la *start date* è maggiore di *end date*.

Nonostante questo componente venga sempre utilizzato in congiunzione con il *wrapper carousel* sono stati comunque mantenuti separati poiché le azioni che svolgono sono differenti e unirli avrebbe causato il non adempimento al principio della *separation of concerns*.

3.7.5 Wrapper ChartJS

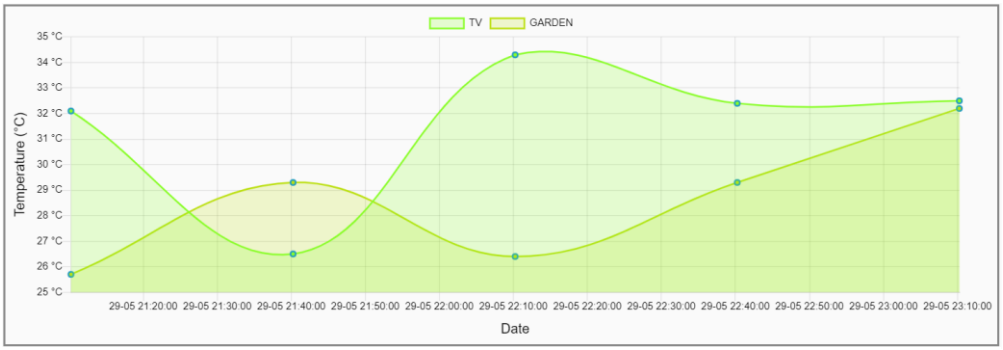


Figura 53: Wrapper ChartJS

Il **wrapper chartjs** è stato realizzato tramite l'utilizzo del componente *vue-chartjs* [14] ma aggiungendo una serie di funzionalità che permettono un facile confronto tra più misure di device differenti. La tipologia di grafico che viene costruita è del tipo *scatter*.

Le **props** richieste dal wrapper sono:

- *chartData:* un oggetto che rappresenta i punti da graficare all'interno del grafico. L'oggetto ha il seguente formato:

$$\{datasets: []\}$$

  Dove *datasets* è un array di dataset da mostrare all'interno del grafico. Un dataset descrive i punti da graficare relativi a un device, di conseguenza più dataset corrispondono a informazioni di device differenti. Al fine di ottenere maggiori dettagli sulle proprietà degli oggetti che compongono l'array *datasets* si faccia riferimento alla documentazione di *chartjs*.
- *chartOptions:* oggetto di classe *ChartOptions*, discusso nel *capitolo 3.2.3*, che permette di specificare le opzioni del grafico.

Considerato che il compito del wrapper è la sola costruzione dei punti all'interno di un asse cartesiano, non sono esposti eventi. Infine, ogni qualvolta il wrapper rileverà il cambiamento della *props chartData* effettuerà, in automatico, l'aggiornamento dei punti contenuti all'interno del grafico effettuando un nuovo rendering.

3.7.6 Wrapper Leaflet

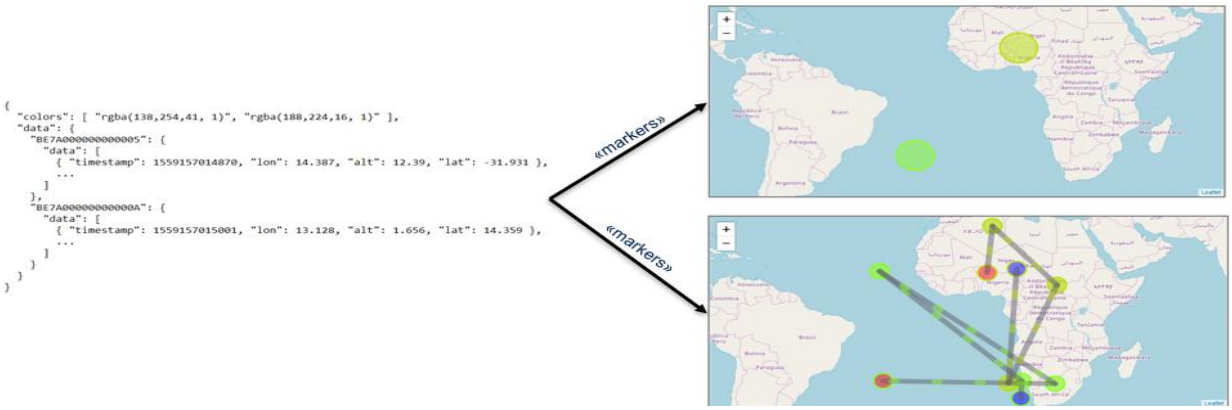


Figura 54: Wrapper Leaflet

Il **wrapper leaflet** permette la creazione di una mappa basata sul componente *vue2-leaflet* [15] e sul componente *leaflet-ant-path* [16]. Le funzionalità offerta dal wrapper permettono di modificare dinamicamente l'insieme di punti passati come *props* (*markers*), reagendo a modifiche dei punti da parte del componente padre. Inoltre, permette di effettuare due tipologie di rappresentazioni diverse, una relativa all'ultima posizione utile registrata dai device e una relativa allo storico degli spostamenti dei devices, andando ad evidenziare, come è possibile vedere in *figura 56*, il punto di partenza, rappresentato in *blu*, e il punto di destinazione, rappresentato in *rosso*.

Le **props** richieste dal wrapper sono:

- *coords:* oggetto contenente le proprietà *lat* e *lon*, che ha la funzione di abilitare il componente alla costruzione anche nel caso in cui non siano presenti *device* con coordinate geografiche da mostrare. Usualmente viene utilizzata la proprietà *leaflet,* contenuta all'interno del modulo di configurazione.

- *configuration:* oggetto contenente le proprietà *zoom*, *maxZoom* e *minZoom* per l'inizializzazione della mappa.
- *markers:* oggetto con due proprietà principali: *colors* e *data*. La proprietà *colors* rappresenta un array di stringhe che descrivono i colori che ogni *markers* deve assumere, per tale motivazione il numero di elementi deve coincidere con il numero di proprietà della proprietà *data*. La proprietà *data* è un oggetto che ha una proprietà per ogni device e il nome della proprietà è l'*eui* del device. Il valore di ogni proprietà è un array di oggetti, dove ogni oggetto ha le proprietà *lat*, latitudine, *lon*, longitudine, *alt*, altitudine e *timestamp*. Nella *figura 56* è mostrato anche un esempio della struttura di *markers*.
- *type:* tipologia di mappa che si intende costruire. Esistono due tipologie di mappe:
    - *markers:* permette di visualizzare le ultime informazioni GNSS per ogni device contenuto all'interno della props *markers*.
    - *roads:* permette di costruire uno storico delle posizioni e del tragitto percorso da uno o più device.

  In entrambi i casi, è offerta la possibilità di ottenere i dettagli sul singolo punto, o su un subpath, grazie ad un *tooltips* che sarà attivato al passaggio del mouse nella sezione d'interesse, in cui saranno mostrate le informazioni approfondite.

Essendo il compito del wrapper la sola costruzione dei punti all'interno della mappa, non sono esposti eventi.

Infine, ogni qualvolta il wrapper rileverà il cambiamento delle *props type* e *markers* andrà a modificare la mappa di conseguenza, reagendo in maniera dinamica. Nell'eventualità in cui non siano presenti markers da mostrare, verrà visualizzato un messaggio di default che sarà posizionato, indipendentemente dagli spostamenti nella mappa, al centro dello schermo per informare l'utente che non sono presenti informazioni geografiche da visualizzare.

3.8 Interfaccia Grafica e Funzionalità

Terminata l'analisi dei componenti principali costruiti per la fruizione delle funzionalità offerte dall'applicativo, non ci resta che mostrare, una per una, tutte le funzionalità che l'utente potrà sfruttare per interagire con i device registrati. L'analisi sarà effettuata in termine di viste esposte all'utente, quindi in termini di pagine visitabili. Concentrandoci esclusivamente su quelle significative per il lavoro svolto, ignorando ad esempio la *home* e il *profile* che mostrano rispettivamente una overview sulle *features* esposte dalla dashboard e della documentazione creata ad hoc per quanto concerne il protocollo LoRaWAN. Così facendo si guida l'utente, step by step, nel processo di creazione del connettore nonché di registrazione degli *end-device* all'interno del cloud *Loriot*, che dovrà essere interamente gestito dall'utente come già spiegato nell'introduzione di questo documento.

3.8.1 Device

La vista **device** permette all'utente di visionare tutti i device registrati all'interno dell'applicazione, di visionare i dettagli dei singoli dispositivi e di aggiungere un nuovo device.

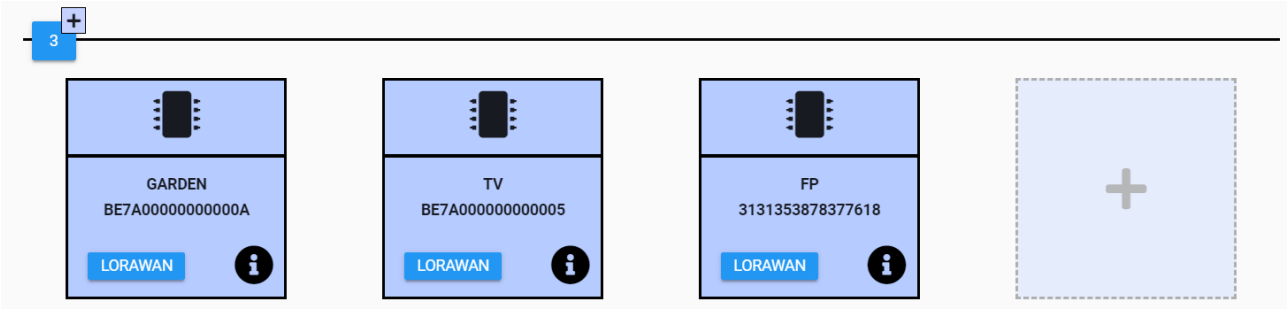


Figura 55: View Devices

La prima schermata mostrata nella *figura 57* permette di visionare tutti i device che sono stati registrati, mettendo in risalto esclusivamente l'*eui* del device, il nome custom assegnatovi in fase di registrazione e la tipologia di rete a cui gli *end-device* si collegano.

Inoltre, tramite due pulsanti posizionati in zone differenti, si dà la possibilità all'utente di aggiungere un nuovo device all'interno dell'applicativo, permettendo di specificare le informazioni base del device e la possibilità di associare un certificato X.509 al device. Quest'ultima evenienza è prevista nel caso in cui, al termine della registrazione, sarà ritornato il *PEM* rappresentante le informazioni del certificato e la chiave privata da inserire all'interno dell'end-device, in modo da abilitarlo all'utilizzo della soluzioni di sicurezza discusse durante il *primo capitolo* di questa trattazione.

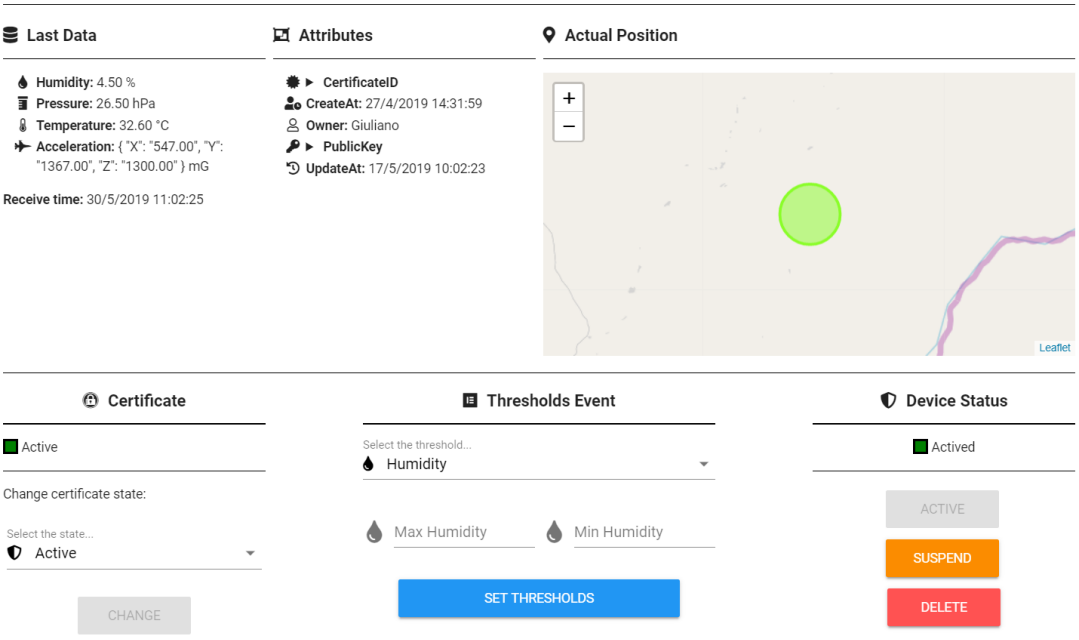


Figura 56: View Device Details

Infine, è possibile visionare i dettagli di uno specifico device cliccando sul pulsante informativo mostrato all'interno della *card* rappresentante il device, azione che permetterà di aprire un dialog contenente le informazioni del device (*figura 58)*. Le informazioni rappresentate descrivono interamente il device mostrando: gli ultimi dati ricevuti, specificando l'istante temporale di ricezione; gli attributi associati al *device*; l'ultima posizione registrata per il device, nell'eventualità in cui sia abilitato all'invio di questa tipologia di dati; la possibilità di modificare lo stato del certificato, al fine di bloccarlo temporaneamente o definitivamente nel caso in cui sia necessario; la possibilità di settare soglie su alcuni dati di telemetria, al fine di ricevere una notifica nell'eventualità in cui sia registrata una misura avente un valore maggiore o inferiore

rispetto alla soglia settata. Infine, è possibile modificare lo status contenuto all'interno del *desired* sospendendo temporaneamente il device o cancellandolo dall'applicazione. Nell'eventualità in cui l'utente stia per effettuare un'operazione non annullabile, ad esempio la cancellazione del device, sarà avvisato tramite un messaggio di warning che esplicita che l'azione non può essere annullata.

3.8.2 Telemetry

Figura 57: View Telemetry

La vista **telemetry**, mostrata nella *figura 59*, permette all'utente di analizzare le misure rilevate dai devices in un grafico cartesiano ove nell'asse delle ascisse viene rappresentato l'intervallo temporale, mentre nell'asse delle ordinate viene mostrato il valore della specifica misura, mostrando anche l'unità di misura relativa al valore. Grazie all'utilizzo dei *wrapper* precedentemente discussi sarà possibile selezionare uno o più devices al fine di confrontarne i valori e selezionare anche l'istante temporale a cui si è interessati, con una precisione di un minuto. Inoltre, sarà possibile aggiornare in tempo reale la fine dell'istante temporale tramite il tasto di refresh, avendo una precisione di un secondo.

3.8.3 GNSS

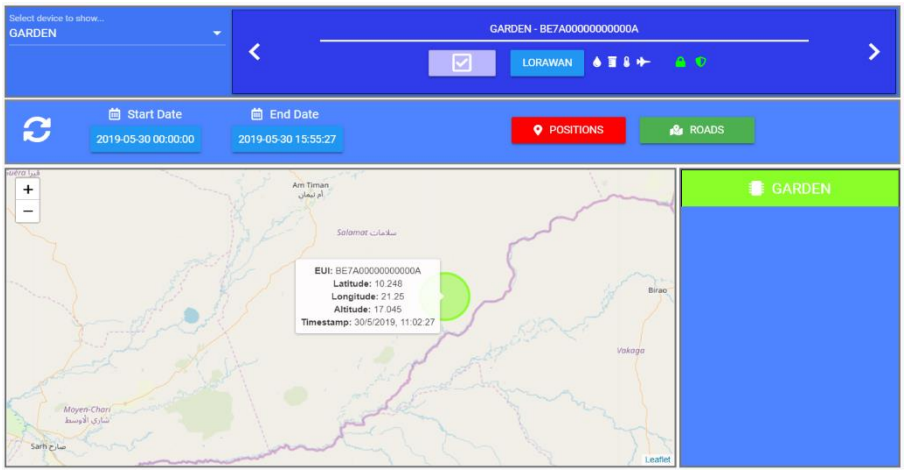


Figura 58: View GNSS

La vista **GNSS**, mostrata nella *figura 60*, permette di visionare gli ultimi dati relativi alla posizione geografica di uno o più devices all'interno del *wrapper leaflet* discusso in precedenza. Vengono offerte tutte le funzionalità viste nella view *telemetry* ed inoltre viene visualizzato,

adiacentemente alla mappa, una legenda ove vengono mostrate le relazioni *device-colore*, in modo da facilitare la comprensione dei dati mostrati sulla mappa.

3.8.4 Events

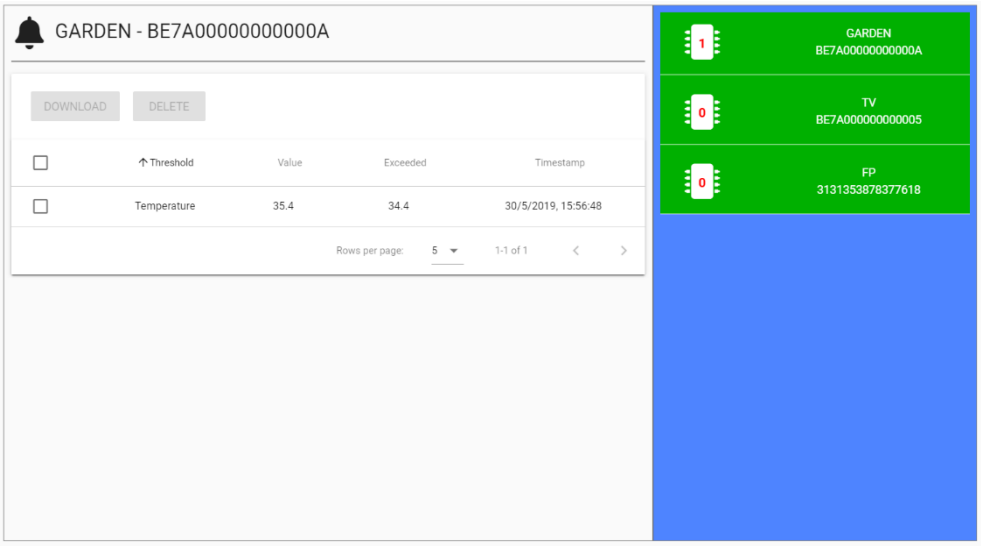


Figura 59: View Events

La vista **Events**, mostrata nella *figura 61*, permette di gestire le notifiche ricevute dai device, grazie all'utilizzo dei topic MQTT, a seguito del superamento di una soglia settata dall'utente. Le notifiche ricevute vengono memorizzate all'interno dello store e rese disponibili per tutto il ciclo di vita dell'applicativo, ma non saranno memorizzate all'interno del database a causa del numero non trascurabile di notifiche che possono essere ricevute. Al fine di compensare questa mancanza, viene offerta la possibilità, all'utente, di effettuare il download delle notifiche a cui è interessato in formato *".csv"*. Inoltre, viene anche offerta la possibilità di cancellare le notifiche che vengono considerate irrilevanti o che sono già state scaricate localmente.

Infine, tramite la legenda posta a destra della datatable, contenente le notifiche per uno specifico device, è possibile selezionare il device di cui si desidera analizzare le notifiche. Il numero di notifiche ricevute per un determinato device viene rappresentato dal numero all'interno dell'icona nella legenda.

3.9 Build dell'Applicativo

Terminata l'analisi dell'applicativo, andiamo ad analizzare la **fase di build**. Questa permette la messa in campo dell'applicativo e, grazie all'introduzione di ulteriore logica, di effettuare due azioni fondamentali per la corretta istanziazione dell'applicazione:

- **Validazione del documento JSON** rappresentate il modulo di configurazione dello store tramite un *JSON schema*. Questo permettere di individuare eventuali errori presenti all'interno del file di configurazione prima di avviare l'applicativo, assicurandosi, inoltre, che tutte le informazioni contenute all'interno del modulo di configurazione siano del formato richiesto al fine di integrarsi con i moduli software già preesistenti. Inoltre, il file di configurazione, una volta letto e validato, viene memorizzato all'interno di una variabile d'ambiente al fine di essere disponibile, direttamente, all'interno dello store senza l'aggiunta di ulteriore logica.

- **Lettura dei moduli che compongo lo store**; tale operazione permette di effettuare la lettura, tramite scansione delle varie directory che compongono lo store, di tutti i moduli che andranno a formare lo store una volta che l'applicativo sarà avviato. L'array contenente tutte le istanze dello store viene memorizzato all'interno di una variabile d'ambiente in modo da essere già disponibile in fase di avvio dell'applicazione. Prelevando il contenuto della variabile di sistema sarà possibile costruire l'oggetto che rappresenta l'istanza dello store.

Tutte le operazioni svolte vengono rappresentate nella *figura 62*.

```
> client@1.0.0 dev C:\Users\giuli\Desktop\Front End
> webpack-dev-server --inline --progress --config build/webpack.dev.conf.js

--- JSON schema successfully build ---

--- Start to read store modules ---
src/store/modules/account/index.js
src/store/modules/alert/index.js
src/store/modules/configuration/index.js
src/store/modules/devices/index.js
src/store/modules/events/index.js
src/store/modules/gnss/index.js
src/store/modules/telemetry/index.js
```

Figura 60: Build

Conclusioni

Lo scopo della tesi, rappresentato dalla realizzazione di una dashboard generica che permetta l'integrazione con qualsiasi tipologia di backend, è stato realizzato ottenendo buone prestazioni e offrendo una rapida espansione della dashboard nei confronti di qualsiasi backend. È stata, inoltre, richiesta esclusivamente la modifica delle tre classi principali in cui è contenuta l'intera logica applicativa, ovvero le classi concrete generate dalle *abstract factory* in fase di inizializzazione dell'applicazione per adattare la dashboard ad un nuovo backend. La velocità di implementazione di una nuova soluzione permette una grande riusabilità che ridurrà nuovi lavori sul progetto prodotto.

L'integrazione con gli aspetti di sicurezza ha permesso di gestire l'intero sistema di autorizzazioni, sfruttando le funzionalità messe a disposizione dal provider AWS, come il meccanismo delle policy, e riuscendo ad ottenere buoni livelli di segregazione delle informazioni. Inoltre, la *device impersonation*, problematica centrale nello studio, è stata risolta grazie all'introduzione della firma digitale sui dati inviati dall'end-device. La firma viene verificata sul backend di riferimento in modo da appurare se i dati ricevuti sono validi, e corrispondono all'end-device corretto.

Purtroppo, come già detto, non è stato possibile attenzionare la parte di downlink, ovvero la possibilità da parte dell'utente di inviare dei comandi ad uno specifico end-device, a causa delle limitazioni dell'account *free* offerto da *Loriot*, dove il downlink è disabilitato.

Proprio per questa ragione una futura espansione potrebbe trattare la realizzazione di uno studio dedicato esclusivamente alla comunicazione in downlink, dove si necessiterebbe di effettuare un'analisi simile a quella condotta nel *primo capitolo* per quanto concerne gli attacchi all'integrità ed autenticazione dei dati scambiati. Infatti, in maniera del tutto analoga, sarebbe possibile aggiungere una firma ai dati inviati dal *backend* all'*end-device,* al fine di guadagnare le due proprietà citate ma si necessiterebbe di creare un certificato, da assegnare al *backend*, e memorizzare la corrispondente chiave pubblica all'interno dell'end-device. Così facendo si potrebbe verificare se il contenuto dei dati è stato modificato durante la trasmissione e se il mittente è realmente il server che ha dichiarato di essere. Oltre alla memorizzazione della chiave pubblica, si necessiterebbe di implementare il codice relativo alla verifica della firma associata ai dati all'interno dell'end-device. Certamente questa introduzione aggiungerebbe un overhead di computazione e di memorizzazione, ma andrebbe ad aumentare la robustezza e sicurezza del canale di downlink utilizzato. Infine, se i dati scambiati dal server all'end-device dovessero essere sensibili, sarebbe opportuno effettuarne la cifratura in modo che nessuno sia in grado di leggerli.

Bibliografia

[1] ST, «User Manual,» [Online]. Available: en.DM00547531.pdf.

[2] Semtech, «Semtech,» [Online]. Available: https://www.semtech.com/lora/what-is-lora. [Consultato il giorno 21 05 2019].

[3] L. Alliance, «About LoRaWAN,» [Online]. Available: https://lora-alliance.org/about-lorawan. [Consultato il giorno 21 5 2019].

[4] M. L. T. E. T. K. O. N. Sornin, «LoRa Alliance,» Luglio 2016. [Online]. Available: http://wiki.lahoud.fr/lib/exe/fetch.php?media=lorawan102-20161012_1398_1.pdf.

[5] N. P. a. M. G. Ismail Buton, «Security Risk Analysis of LoRaWAN and Future Directions,» *Future Internet,* p. 22, 2018.

[6] AWS, «Cos'è il cloud computing,» [Online]. Available: https://aws.amazon.com/it/what-is-cloud-computing/. [Consultato il giorno 22 05 2019].

[7] AWS, «Amazon Api Gateway,» [Online]. Available: https://aws.amazon.com/it/api-gateway/.

[8] AWS, «Panoramica di AWS IoT Core,» [Online]. Available: https://aws.amazon.com/it/iot-core/.

[9] AWS, «Identity and Access Management (IAM),» [Online]. Available: https://aws.amazon.com/it/iam/.

[10] IETF, «RFC7519,» [Online]. Available: https://tools.ietf.org/html/rfc7519.

[11] Vue, «Introduzione a Vue,» [Online]. Available: https://vuejs.org/v2/guide/.

[12] Vue, «Vue Router Introduction,» [Online]. Available: https://router.vuejs.org/.

[13] mariomka. [Online]. Available: https://www.npmjs.com/package/vue-datetime.

[14] J. Juszczak, «vue-chartjs,» [Online]. Available: https://vue-chartjs.org/.

[15] korigan, «vue2-leaflet,» [Online]. Available: https://www.npmjs.com/package/vue2-leaflet.

[16] rubenspgcavalcante, «leaflet-ant-path,» [Online]. Available: https://www.npmjs.com/package/leaflet-ant-path.