UNIVERSITÁ POLITECNICA DELLE MARCHE

FACOLTÁ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione

Sviluppo di un'architettura serverless per la la gestione di sensori in contesti di Orti botanici in rete

Development of a serverless Architecture to mange sensors In the context of a Botanic Gardens Network

Relatore:
Ch.mo Prof. Mancini Adriano

Laureando:
Spada Claudio

Anno Accademico 2018-2019

Indice

Capitolo 1

Introduzione

1.1 Prefazione

La tesi consiste nella descrizione dettagliata del progetto di tirocinio svolto in collaborazione con il mio collega Roberto Broccoletti [10] che ha lavorato con me allo sviluppo dell'applicazione. Il progetto consiste nella realizzazione di un sistema automatico che lavora nel campo degli orti botanici *smart*; in particolare il sistema deve acquisire quotidianamente da un server dei dati forniti da dendrometri e sensori di temperatura, presenti in un orto botanico, elaborarli, creando dei grafici e successivamente caricarli su un servizio di *storage cloud*, per poi pubblicare un *tweet* con il grafico allegato.

1.2 Obiettivi

L'obiettivo del progetto è di creare il sistema completamente automatico senza avere necessità di un server, quindi con un approccio innovativo, detto appunto *"serverless"*. L'utilizzo di un server classico è efficiente e funzionale e risulta ancora il mezzo più diffuso per creare applicazioni; tuttavia presenta alcuni svantaggi:

- elevata spesa iniziale per l'acquisto dell'hardware;
- il server ha bisogno di una continua manutenzione, sia hardware che software (eventuali riparazioni dovute all'usura, oppure numerosi aggiornamenti da applicare al software);
- comporta costi elevati per tenerlo sempre acceso;
- richiede di implementare una elevata protezione del software.

Per ovviare ad alcuni di questi svantaggi, c'è la possibilità di affittare una *virtual machine* da un *provider* di servizi (ad esempio *Amazon Web Services, Azure*), eliminando così i problemi di gestione del server, ma una *virtual machine* resterebbe sempre in esecuzione, comportando costi anche quando non viene utilizzata; inoltre, rischierebbe di intasarsi nei momenti in cui vengono effettuate troppe richieste da parte degli utenti, rendendo necessaria una maggiore spesa per l'aumento delle risorse della macchina virtuale. L'innovativo approccio *serverless*, invece, elimina tutti gli svantaggi del classico server e della macchina virtuale, dato che, oltre a non dover gestire il server, presenta i seguenti vantaggi:

- Non è necessario gestire il *software* del *server*; aggiornamenti e configurazioni sono interamente eseguite dal fornitore del servizio
- Non si hanno a disposizione delle risorse fisse, ma vengono calibrate automaticamente in base al numero di richieste effettuate all'applicazione in un determinato istante.
- Si paga solo il tempo di esecuzione e le risorse effettivamente consumate;
- Il servizio risulterebbe sempre disponibile e resistente ai guasti.

Quindi, l'approccio *serverless*, risulta effettivamente efficace e affidabile. Tuttavia richiede un modo di strutturare il software completamente diverso; non basterà sviluppare l'applicazione in locale e poi passare tutto direttamente sul server. Occorre invece organizzare il progetto in una serie di piccole funzioni, che lavorano in cascata attraverso dei trigger che vengono scatenati al verificarsi di determinati eventi.

1.3 Struttura tesi

La tesi è strutturata nel seguente modo:

- in una prima parte vengono descritte tutte le tecnologie usate, ad esempio eventuali *software* necessari o i linguaggi di programmazione utilizzati per sviluppare il progetto
- nella seconda parte vengono descritte tutte le funzioni necessarie al funzionamento del sistema e viene descritto il *workflow* dell'intero progetto
- nell'ultima parte viene spiegata dettagliatamente l'implementazione delle funzioni e la spiegazione delle parti principali delle funzioni stesse.

Infine sono presenti delle considerazioni personali, incentrate sulle difficoltà incontrate durante lo sviluppo e il *debug* del *software*, dovute principalmente all'approccio completamente nuovo.

Capitolo 2

Tecnologie usate

2.1 Servizi

Di seguito i servizi utilizzati nel progetto

2.1.1 AWS

Amazon Web Services (AWS) è una piattaforma *cloud* che comprende diversi servizi, ad esempio, comprende servizi per lo *storage*, per i *database* e per il calcolo; è la piattaforma adottata per il *deploy* dell'intero progetto [1]. Il vantaggio di avere a disposizione una piattaforma *cloud* vasta come quella di *Amazon* è che si hanno a disposizione tutti i servizi necessari e sono tutti collegati tra loro, attraverso *trigger* che si scatenano al verificarsi di determinati eventi. In particolare, per il sistema creato, sono stati usati i seguenti servizi:

- **S3 (Simple Storage Service):** è un semplice servizio di *storage* che è stato usato come appoggio per i file di configurazione, i pacchetti delle *lambda-function* e per i grafici generati. *S3* presenta una serie di vantaggi, come l'elevata scalabilità e durabilità dei dati e le notifiche di eventi, che permettono di far attivare delle funzioni lambda. Lo spazio di archiviazione è diviso in *bucket*, che sono dei contenitori in cui è possibile inserire/cancellare/leggere un numero qualsiasi di oggetti di qualsiasi tipo. Su S3, all'interno del *bucket*, non esistono vere e proprie cartelle, ma può essere usato il nome dell'oggetto come se fosse il percorso di un file, che può essere quindi interpretato come se fosse un normale percorso all'interno di un classico file *system* [17].

- **Lambda:** è il servizio che esegue il codice delle funzioni. Permette quindi la creazione di funzioni che saranno sempre pronte per essere eseguite in caso di richiesta; queste funzioni sono eseguite in ambienti isolati, ogni funziona avrà a disposizione a *runtime* delle risorse configurabili, che ne

permettono l'esecuzione. Dato che le funzioni sono *stateless* e vengono quindi inserite in ambiente di esecuzione solo a *runtime*, *AWS* può avviare contemporaneamente diverse istanze della stessa funzione, in base al numero di richieste ricevute [16].

- **DynamoDB:** è un *database NoSQL* che supporta dati di tipo documento e di tipo chiave-valore, molto scalabile e con tempi di risposta molto rapidi ed è stato usato per salvare i dati provenienti dai dendrometri [15].
- **API Gateway:** è il servizio che permette la creazione di *API*, per permettere l'uso di applicazioni dall'esterno di *AWS*; in questo progetto sono state usate sia per permettere il salvataggio di dati inviati direttamente sul *DB*, sia per creare una interfaccia per la modifica delle configurazioni dell'applicazione [13].
- **Cognito:** permette la gestione di un *pool* di utenti; si occupa di gestire le registrazioni e le sessioni di *login* e permettere quindi di accedere ad una funzione *lambda* [14].
- **CloudWatch:** è un servizio di monitoraggio delle applicazioni, creato da *Amazon* per consentire agli sviluppatori di monitorare le *performance* delle proprie applicazioni. In questo progetto è stato usato per notificare alle funzioni *lambda* un evento temporale [19].

2.1.2 Sentry

Sentry è una piattaforma *cloud open source* per il monitoraggio degli errori; è risultata molto utile in fase di *debug*, utile in particolar modo con l'approccio *serverless*, poiché non si ha modo di monitorare l'esecuzione dell'applicazione e di capire quindi gli errori che si sono verificati. Per usarla è necessario scaricare l'*SDK*, disponibile per diversi linguaggi di programmazione e poi inserire nelle funzioni del codice che permette a *Sentry SDK* di catturare le eccezioni non gestite e inviarle sul *cloud* di *Sentry* [11].

2.2 Software

Di seguito si presentano i vari strumenti software utilizzati durante l'esecuzione del progetto di tesi.

2.2.1 Postman

Postman è un *software* per semplificare e migliorare lo sviluppo di *API*. Per questo progetto è stato usato solo come *client*, per il *test* delle *API* create e per capire meglio come viene effettuata la gestione delle richieste da parte di *API Gateway* e di *Twitter*. In particolare, è stato utile per capire come settare correttamente gli *header* per le richieste, dato che permette di effettuare in modo semplice richieste *POST* o *GET*, completando così la fase di *test* delle *API* prima di sviluppare il codice [9].

2.2.2 Docker

Docker è uno strumento *software* che permette l'esecuzione delle applicazioni all'interno di un *container*. Un *container* è un'unità *software* che comprende codice e dipendenze, in modo da eseguire l'applicazione da un ambiente all'altro, senza cambiare nulla. I *container* sono distribuiti in immagini, che contengono tutto il necessario per l'esecuzione, comprese anche le librerie di sistema. Un altro vantaggio risiede nel fatto che il container è uno spazio isolato dal sistema operativo e quindi un software al suo interno girerà allo stesso modo su tutte le piattaforme. In questo progetto è stato usato per eseguire l'immagine dell'ambiente in cui *AWS* esegue le *lambda*, in questo modo è stato possibile recuperare, attraverso lo strumento di *python (pip)*, le librerie per *python* in grado di funzionare con le *lambda function*. In alternativa sarebbe stato necessario compilare le librerie attraverso le *EC2* di *AWS*, che si sarebbe rivelato un superfluo dispendio di risorse economiche [2].

2.2.3 AWS-CLI e AWS Console

AWS-CLI è uno strumento *software* con una interfaccia a linea di comando per gestire i servizi *AWS* da linea di comando o da *script*; è possibile ad esempio caricare file su *S3*, oppure configurare i vari servizi [18].
AWS Console è invece uno strumento *software online* con interfaccia grafica, disponibile sul sito *AWS*, che consente una comoda gestione dei servizi *AWS* [20].

2.3 Linguaggi di programmazione

2.3.1 Python

Python è un linguaggio di programmazione ad alto livello, multi-paradigma, con supporto al paradigma orientato agli oggetti ed è un linguaggio semplice e flessibile. Utilizza variabili non tipizzate, ma a *runtime*, il tipo viene assegnato alla variabile e risulta ben definito, quindi la variabile non potrà contenere tipi diversi da quello assegnato. Una sua caratteristica distintiva è l'uso dell'indentazione per definire le specifiche, quindi il codice risulterà molto più pulito e leggibile. Ha una gestione automatica per la liberazione della memoria. In fase di *runtime*, il sorgente viene prima convertito in *bytecode* e poi eseguito; c'è quindi una precompilazione che permette di non interpretare il linguaggio ad ogni esecuzione, migliorandone quindi le prestazioni. Ha il vantaggio di poter integrare estensioni scritte anche in C o C++, per migliorare i tempi di esecuzione di alcune parti del programma e ad oggi gode di una vasta gamma di librerie che rendono il *Python* utilizzabile per innumerevoli scopi [29].
In questo progetto è stato usato per scrivere il codice di tutte le *lambda function*, quindi il core del progetto è basato interamente su *Python*.

2.3.2 HTML

HTML (HyperText Markup Language) [7] è un linguaggio di *markup*, usato nel *W3 (World Wide Web)* per la creazione di ipertesti. Un linguaggio di *markup* è un insieme di regole che permettono di descrivere la struttura di un testo, tralasciando tutta la parte di formattazione del documento, che viene affidata ad altri linguaggi, come il *CSS (Cascading Style Sheets)* [8], che è creato appositamente per definire delle regole di stile del documento. Le pagine *HTML* sono quindi dei semplici file di testo con dei *TAG* al proprio interno, che vengono scaricate e interpretate dai *browser* durante la navigazione di un sito *web*. In questo progetto è stato usato l'*HTML* sia per creare le *Twitter Card*, sia per creare le pagine dell'applicazione web per la configurazione dell'applicazione.

2.3.3 JavaScript/JSON

JavaScript è un linguaggio di *scripting*, orientato agli oggetti e agli eventi e l'utilizzo più diffuso è nelle applicazioni web lato *client*, per ottenere effetti visivi dinamici o implementare delle vere e proprie funzioni e permette il controllo completo del *DOM (Document Object Model)*. Negli ultimi anni si è ampiamente diffuso anche lato *server*, attraverso la *runtime Node.js*. Le funzioni *JavaScript* possono essere inserite direttamente nelle pagine *HTML*, oppure scritte in file esterni e poi collegate alle pagine *HTML*. Un aspetto interessante è la compatibilità con la tecnica *AJAX (Asynchronous JavaScript and XML)*, una tecnica per

lo scambio di dati *client-server* asincrona [5], utilizzata in questo progetto per lo sviluppo dell'applicazione *web*.
JSON (JavaScript Object Notation) è invece un formato, basato sul linguaggio *JavaScript*, per l'interscambio di dati tra applicazioni *client/server*. Questo formato è stato utilizzato per la scrittura dei file di configurazione e per lo scambio di dati *client/server* dell'applicazione *web* [22].

Capitolo 3

Descrizione e progettazione delle funzioni

3.1 Progettazione dell'applicazione

L'applicazione è stata progettata partendo dai requisiti, analizzandoli e capendo al meglio come realizzare l'applicazione utilizzando l'approccio *serverless*. Lo scopo dell'intero progetto è rendere *smart* un orto botanico, inserendo anche una forte componente *social*. In particolare, oltre ai *tweet* di grafici relativi alla crescita delle piante (grazie ai dendrometri), in una parte del progetto, sono stati creati anche dei *tweet* contenenti dei video, che mostrano la crescita delle piante. I requisiti iniziali erano i seguenti:

1. Periodicamente vi è da verificare se vi sono eventi relativi ad una data specie relativamente ad uno stato fenologico, pubblicando, al verificarsi di un evento, un *tweet* contenente: specie, varietà, periodo iniziale, periodo finale e sito;

2. Creazione di un sistema che consenta di ricevere in ingresso mediante una POST su *API Gateway* di *Amazon* dei dati relativi a sensori di umidità, temperatura e crescita di piante;

3. Sistema che crea un grafico per ogni dato e lo pubblica su *Twitter*, con *tweet* giornalieri, settimanali, bisettimanali e mensili.

Per realizzare questi primi tre requisiti, è stato necessario:
Per il punto 1, la creazione di due funzioni lambda:

- una funzione che preleva i dati relativi alla pianta dal *DB*, crea il testo del *tweet*, partendo da un testo presente in un file di configurazione e inserendo all'interno del testo i dati della pianta attraverso dei *placeholder*, come da specifiche;

- una ulteriore funzione che ha solo il compito di creare delle regole per l'attivazione della funzione che pubblica il *tweet* (funzione descritta poc'anzi); in particolare, per ogni pianta contenuta nel *DB*, crea due regole di *CloudWatch Events* (una per pubblicare un *tweet* il giorno di inizio periodo e un'altra per il giorno di fine periodo), che attivano la funzione all'arrivo del giorno previsto.

Si è scelto di utilizzare questo approccio per avere il vantaggio di evitare di interrogare quotidianamente tutto il *DB*, cercando una pianta che avesse un evento da pubblicare. In questo modo si evitano i costi giornalieri sia dell'esecuzione della funzione lambda, sia dell'interrogazione al *DB*. La funzione che crea le regole di schedulazione verrà chiamata solo in caso di inserimento, modifica o cancellazione di un elemento sulla tabella del *DB* contenente le piante.
Per il punto 2, è stato necessario creare una funzione *lambda* in grado di acquisire i dati forniti in *JSON* dalla richiesta effettuata su *API Gateway* e poi inserirli sul *DB* in una tabella che contiene tutti i dati prelevati dai vari sensori. Per lo sviluppo di questa specifica non sono stati necessari particolari accorgimenti.
Il terzo requisito ha richiesto la creazione di una funzione *lambda* che, in sequenza, svolge le seguenti operazioni:

- Attivazione attraverso un evento *CloudWatch*, che può essere giornaliero, settimanale, mensile, ..
- Prelievo dati relativi al periodo indicato nel *trigger* di attivazione, dal *DB*
- Creazione di tutti i grafici (uno per ogni variabile)
- *Upload* dei grafici su *S3*

Per evitare la creazione di troppe funzioni lambda che assolvono allo stesso scopo e per renderle più semplici e snelle possibile (evitando così di incentrare tutto il lavoro su un'unica lambda, ma rendendole appunto delle funzioni che svolgono un solo tipo di operazione), è stata usata la funzione creata in precedenza per la pubblicazione dei *tweet*, richiamandola, attraverso un *trigger*, ad ogni caricamento di un grafico su *S3*. La funzione si occuperà quindi, in caso di immagine da pubblicare, di creare una *Twitter card* (sono delle pagine *HTML* che permettono il link di qualsiasi contenuto al loro interno) e poi pubblicare il *tweet*.

Successivamente, si sono manifestate altre due necessità:

- Acquisire dati anche da un server *FTPS (File Transfer Protocol Secure)*, oltre che da *API Gateway* (è stato inserito un *datalogger* che carica due volte al giorno i dati su un server di appoggio in formato *CSV (Comma-separated values)*)

- Possibilità di modificare le impostazioni dell'applicazione e di inserire una nuova pianta, senza passare direttamente dal *DB* o da *S3*, rendendo quindi possibile la configurazione anche ai "non addetti ai lavori"

Per l'implementazione di queste ulteriori specifiche, è stata creata una funzione che preleva quotidianamente i dati dal *server FTPS* e il salva sul *DB* (per soddisfare il primo requisito). Per il secondo requisito è stato necessario creare una applicazione web, di cui il *back-end* è stato realizzato con una *lambda function* mentre il *front-end* è stato realizzato tramite *HTML e JavaScript*, usando *AJAX* per scambiare dati con il *server*. *AWS* comprende anche la gestione degli utenti, attraverso *Amazon Cognito*, in modo da poter configurare direttamente dalla console *AWS* l'accesso alla funzione lambda di *back-end*.

3.1.1 Workflow preliminare

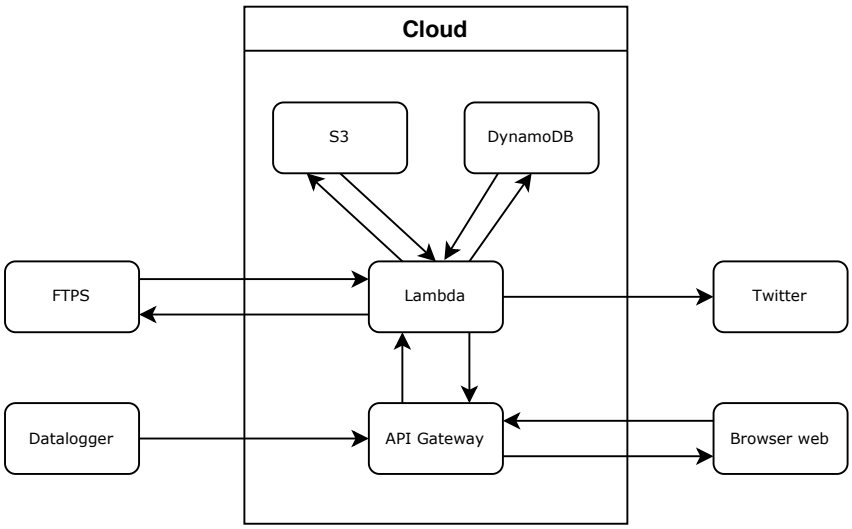


Figura 3.1: Workflow preliminare del funzionamento dell'applicazione

In questo schema è descritto in modo preliminare il funzionamento del progetto; è stato utile per iniziare a progettare le varie funzioni lambda e capire come farle interagire tra loro.
Dallo schema si può notare come le funzioni lambda sono il cuore di tutta l'applicazione. Tali funzioni si occupano infatti di creare grafici e anche di fare da ponte tra i servizi. In particolare, le funzioni dovranno svolgere i seguenti compiti:

- consentire al *datalogger* di caricare dati sul *DB*; si usa *API Gateway* per permettere alla funzione di ricevere la richiesta;
- prelevare dati dal *server FTPS* e caricarli sul *DB*;
- creare i grafici con i dati presenti sul *DB*;
- pubblicare *tweet* di stato e *tweet* con grafici;
- consentire all'applicazione *web* di accedere ai file di configurazione e al *DB*; anche in questo caso si userà *API Gateway* per gestire le richieste.

Partendo da questo schema preliminare, sono state definite dettagliatamente tutte le funzioni (come accennato nella sezione precedente):

- *Graph creator*
- *Publish event tweet*
- *Store data from FTPS*

- *Store data from API*
- *Fix DynamoDB plant table*
- *Rule schedule creator*
- *Setting config*

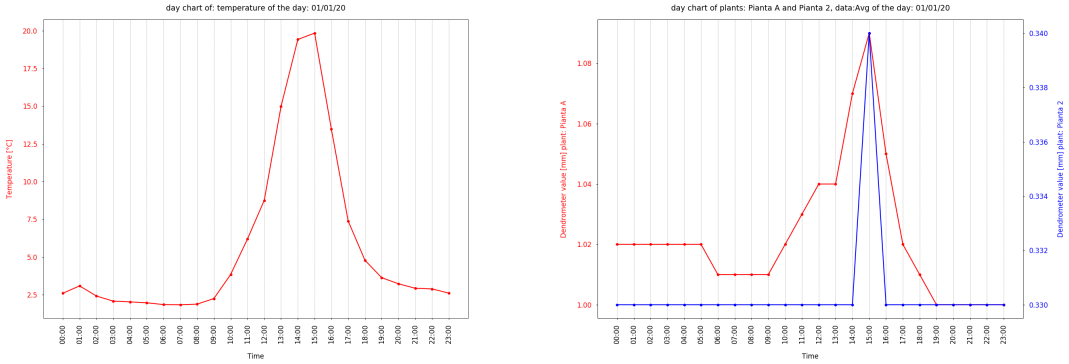
Per il **front end** dell'applicazione *web*, è invece stata creata una interfaccia *HTML* e una applicazione *JavaScript*, che consente di di scambiare dati con la funzione *SettingConfig* che fa da *back end*.

3.2 Workflow dettagliato

Le funzioni lambda elencate nella precedente sezione, non sono altro che codice (in *Python*; possono comunque essere scritti in diversi altri linguaggi, come *Node.js* e *Java*); è quindi necessario eseguire il codice, inserendolo in un ambiente di esecuzione creato da *AWS* solo quando è necessario eseguire la funzione. *AWS* assegna quindi alla funzione memoria, memoria di massa e un processore.
L'esecuzione viene scatenata al verificarsi di un evento. Alla funzione vanno quindi associati dei *trigger* che ne lanciano l'esecuzione. I *trigger* possono essere scatenati da vari servizi di *AWS* e in questo caso sono stati usati *trigger* di *S3*, *DynamoDB*, *API Gateway* e *CloudWatch Events*, che sono completamente configurabili tramite la *console* di *AWS*. I *trigger* sono delle notifiche che *Amazon* usa appunto per eseguire le funzione; al verificarsi dell'evento specificato, verrà quindi attivata la funzione, alla quale viene passato un file *JSON* contente i dettagli dell'evento che si è verificato.
Le funzioni lambda devono essere strutturate in modo da avere obbligatoriamente una funzione che funge da gestore (*handler*) che riceve due parametri: *event* e *context*.
Il parametro *event* non è altro che il file *JSON* che viene generato dal *trigger*; contiene tutte le informazioni relative all'evento che si è verificato, ad esempio, nel caso di caricamento di un file su S3, conterrà le informazioni dell'oggetto caricato (chiave, dimensione e *tag*), le informazioni relative al *bucket* e una serie di altre informazioni legate ad *AWS*. Il parametro *event* è fondamentale, in quanto consente di capire da cosa è stata attivata la funzione e quindi iniziare l'elaborazione relativa all'evento verificatosi.
Dopo queste note sulle funzioni *lambda*, si può procedere alla descrizione dettagliata delle funzioni create per la realizzazione dell'applicazione.

Graph creator

La funzione *Graph creator* è deputata a creare i grafici dei dati provenienti dai dendrometri (sensori che monitorano l'accrescimento delle piante), dai sensori di temperatura e di carica delle batterie. Dopo aver creato il grafico, ne carica il file *.png* su *S3* con una *key* (nome dell'oggetto) che è costituita dal percorso *cartella-grafici/AAAA/MM/GG/nome-del-grafico.png*.
Permette la creazione di grafici relativi a una variabile o a due variabili e ne permette la configurazione estetica di vari aspetti, come spessore linee e colore delle linee. Queste configurazioni sono presenti all'interno di un file *JSON* caricato in una specifica cartella di *S3*.



(a) Esempio di grafico a singola variabile (b) Esempio di grafico a due variabili

Figura 3.2: Esempi di grafici generati

Trigger: Regole di *CloudWatch Events* per eseguire la funzione ogni giorno, ogni settimana, ogni due settimane e ogni mese. Le regole sono state create con espressioni *cron*.

Input: file di configurazione *<root>/config/chart_ config.json*

Output: file *.png* dei grafici caricati su S3 dopo la creazione.

File di configurazione chart_config.json

```
{
    "labelPadding":20,
    "labelValuePadding":10,
    "labelSize":15,
    "labelValueSize":15,
    "titleSize":17,
    "figureWidth":17,
```

```
    "figureHeight":12,
    "figureDPI":80,
    "color": "red",
    "color2":"blue",
    "marker": "o",
    "linestyle": "solid",
    "linewidth": "2",
    "markersize": "5",
    "scalex": "linear",
    "scaley": "linear",
    "xlabel": "Time",
    "ylabel": {
        "temperature": "Temperature [\u00b0C]",
        "umidity": "Umidity [\%]",
        "dendrometer": "Dendrometer value [mm]",
        "battery": "Battery level [V]"},
    "hourStep": 1,
    "plotTwoDendrometer":1
}
```

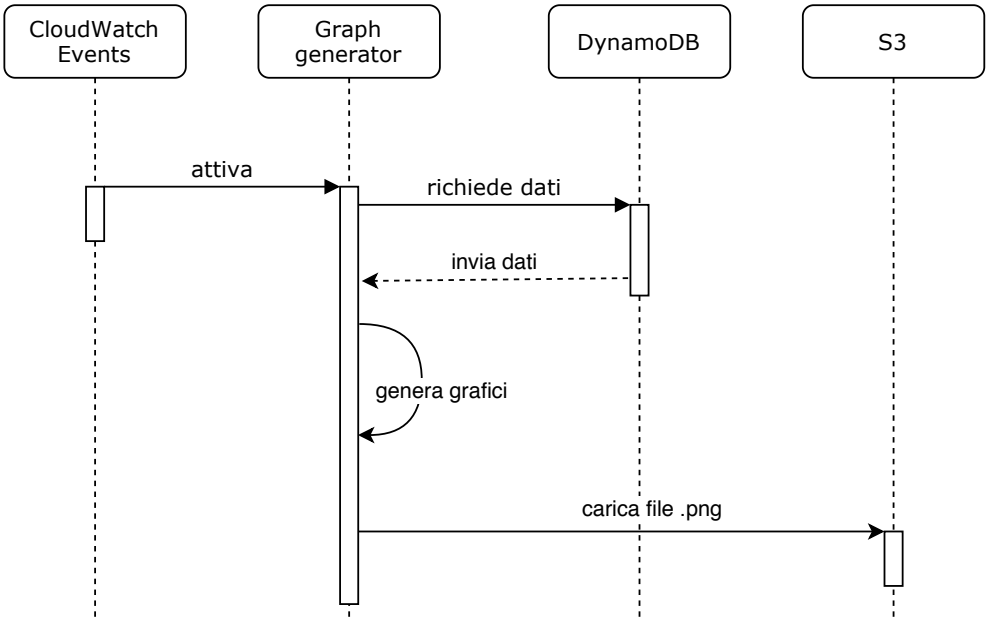


Figura 3.3: Diagramma esecuzione Graph creator

Publish event tweet

Publish event tweet è la funzione che si occupa della pubblicazione dei messaggi e dei grafici su *Twitter*. Dato il grande numero di *tweet* che sarebbero stati pubblicati (uno per ogni variabile, con frequenza giornaliera, più altri con frequenze plurigiornaliere) si è deciso di eliminare il vecchio *tweet* prima di pubblicare il nuovo; in questo modo si ha un profilo *Twitter* molto più ordinato e con un numero di *tweet* contenuti. Per implementare questa funzione è stato necessario creare una tabella sul *DB*, in modo da salvare l'ID del *tweet* corrispondente al grafico, per poterlo poi eliminare prima di caricare il nuovo *tweet*. Si potrebbe pensare che siano sufficienti le *Twitter Card* per evitare un eccessivo numero di *tweet*, dato che è sufficiente modificare l'*URL* all'interno della pagina della *card* per cambiarne l'immagine associata, ma con questo approccio si avrebbero solo *tweet* obsoleti, quindi il profilo non risulterebbe mai aggiornato. Le *Twitter Card* sono delle semplici pagine *HTML* che permettono di collegare al *tweet* un qualsiasi *link* contenente una immagine o un video, per evitare compressioni, perdite di qualità e limiti su lunghezza del video o dimensioni dell'immagine.
Questa lambda, oltre ai *tweet* con grafici, pubblica anche *tweet* relativi allo stato fenologico delle piante, estraendo dal *DB* i dati della pianta per cui di verifica l'evento e costruendo un *tweet* sulla base di un testo situato su *S3*.

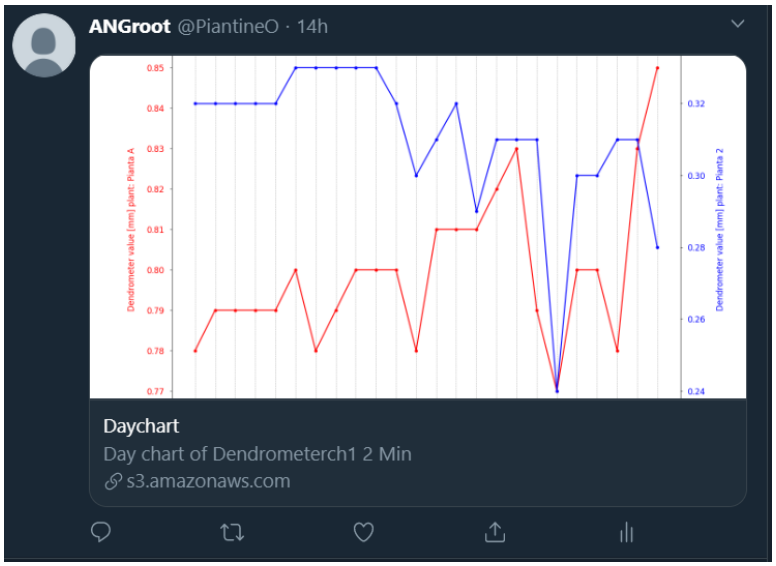


Figura 3.4: Esempio di una Card su Twitter

Trigger: Eventi di creazione oggetti su S3, con prefisso *<root>/chart/month/* e suffisso *.png*; Regole *CloudWatch Events*, due per ogni pianta (inizio e fine periodo), con nome evento: *<id-pianta>_<begin/end>*.

Input: *template* per il testo del *tweet* di inizio e fine periodo, *template* per

le *cards*. Presenti su *S3* in *<root>/template*.

Output: file *HTML* delle *cards* situati in:
<root>/summarycard/<day|month|twoWeeks|week>

Template Twitter cards

```
<!DOCTYPE html>
<html>
    <head>
        <meta content='text/html; charset=UTF-8'
        ↪  http-equiv='Content-Type' />
        <meta name="twitter:card" content="summary_large_image" />
        <meta name="twitter:site" content="@PiantineO" />
        <meta name="twitter:title" content="{{title}}" />
        <meta name="twitter:description" content="{{description}}"
        ↪  />
        <meta name="twitter:image" content="{{url}}" />
    </head>
    <body>
        <img src="{{url}}" alt="{{description}}">
    </body>
</html>
```

Come è possibile notare, la card è una semplice pagina *HTML*; le parole contenute all'interno di doppie parentesi graffe sono dei *placeholder* che verranno riempiti dalla funzione. Conterrà quindi un titolo, una descrizione e l'*URL* dell'immagine che si vuole allegare. Il *body* della pagina è formato solo da una immagine, che funge da anteprima della *card*.

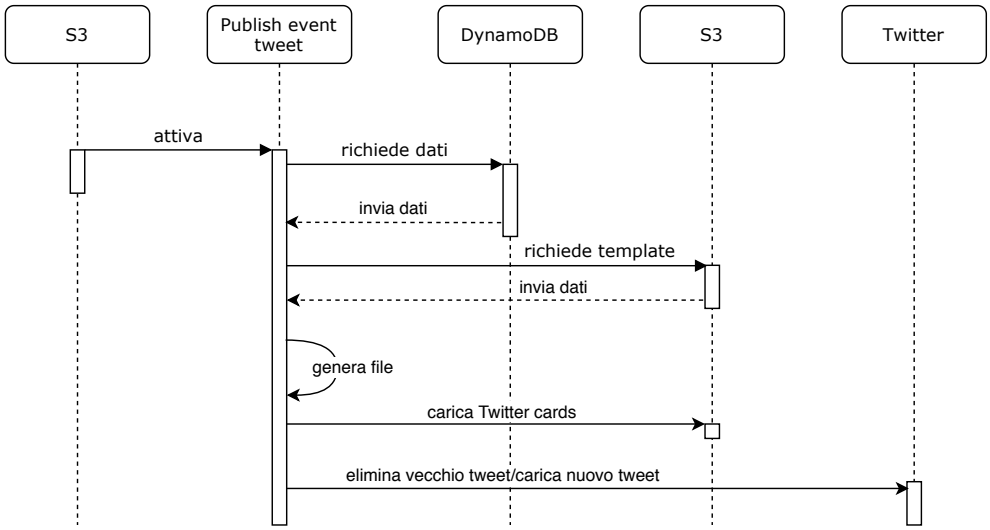


Figura 3.5: Diagramma esecuzione della funzione, dopo attivazione con *trigger* S3

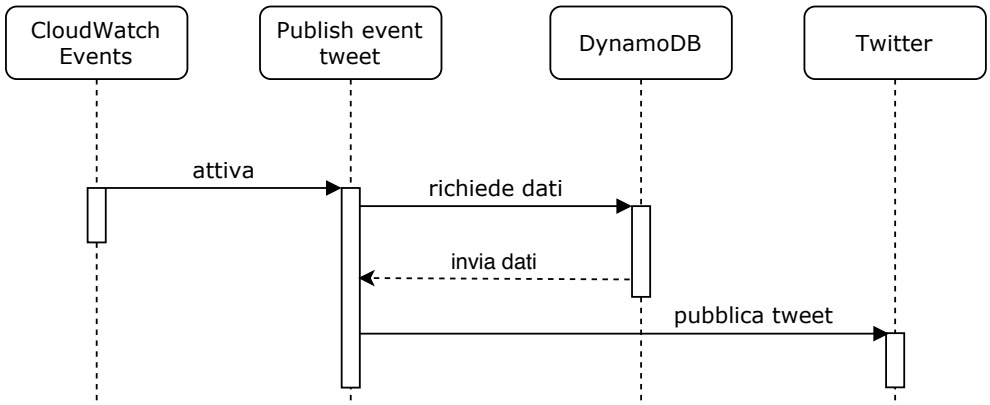


Figura 3.6: Diagramma esecuzione della funzione, dopo attivazione con *trigger CloudWatch Events*

Store data from FTPS

É la funzione che quotidianamente si occupa di prelevare dati dal *server FTPS* e caricarli su *DynamoDB*. Sul *server* sono presenti file in formato *CSV*, la funzione, dovrà quindi leggere tutto il file per estrarne i dati e inserirli in una tabella del *DB*.

Trigger: Regola *CloudWatch Events* per l'attivazione giornaliera.

Input: file di configurazione *<root>/config/ftpsParameters.json*

Output: nessun *output* necessario

File di configurazione ftpsParameters.json

```
{
    "host":"",
    "port":21,
    "timeout":40,
    "username":"",
    "password":"",
    "deviceUUID":""
}
```

Nel file, oltre alle credenziali e alle impostazioni per il *server*, è contenuto anche l'*UUID* del *device* che carica i dati sul *server FTPS*.

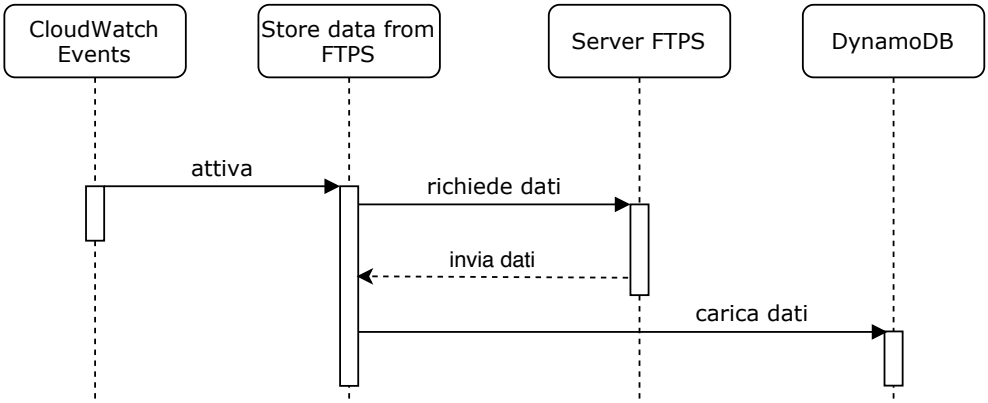


Figura 3.7: Diagramma di esecuzione della funzione Store data from FPTS

Store data from API

É stata creata questa funzione con lo scopo di consentire a un eventuale *datalogger* di poter caricare dati sul *DB* attraverso una richiesta *POST*. L'*API* è stata implementata attraverso *API Gateway* di *AWS*. La richiesta *POST* contiene i dati dei sensori in formato JSON:

```
{
    "deviceUUID": <uuid>,
    "timestamp": <linux_epoch>,
    "temperature": <temp_value>,
    "umidity": <temp_value>,
    "dendrometerCh1": <float_value>,
    "dendrometerCh2": <float_value>,
    "dendrometerCh3": <float_value>,
    "dendrometerCh4": <float_value>,
    "dendrometerCh5": <float_value>
}
```

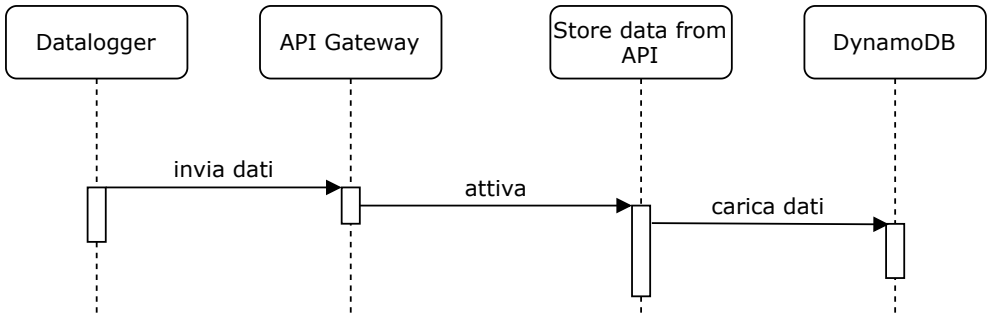


Figura 3.8: Diagramma di esecuzione della funzione *Store data from API*

Trigger: *trigger Gateway API* con l'*endpoint* configurato in *API Gateway*

Input: non sono necessari file di *input*

Output: non sono prodotti file di *output*

Fix DynamoDB plant table

Questa funzione è stata implementata per mantenere la coerenza tra la tabella delle piante e la tabella dei sensori. *DynamoDB* è un database non relazionale, quindi non si occupa di mantenere la coerenza tra le tabelle. I *DB NoSQL* andrebbero progettati in modo totalmente differente da un *DB* relazionale, evitando di avere dipendenze tra tabelle; in questo caso, tuttavia, si è rivelato conveniente avere una tabella che funge solo da collegamento tra la tabella piante e la tabella dispositivi. La struttura del *DB* verrà approfondita durante la fase di implementazione. La funzione *Fix DynamoDB plant table* viene attivata ogniqualvolta viene effettuata una operazione sulla tabella delle piante. La funzione riceve, attraverso il *trigger*, lo *stream* delle modifiche effettuate sulla tabella del *DB*; la funzione dovrà quindi aggiornare, sulla base delle modifiche rilevate, la tabella che funge da tabella di collegamento tra la tabella piante e la tabella dispositivi.

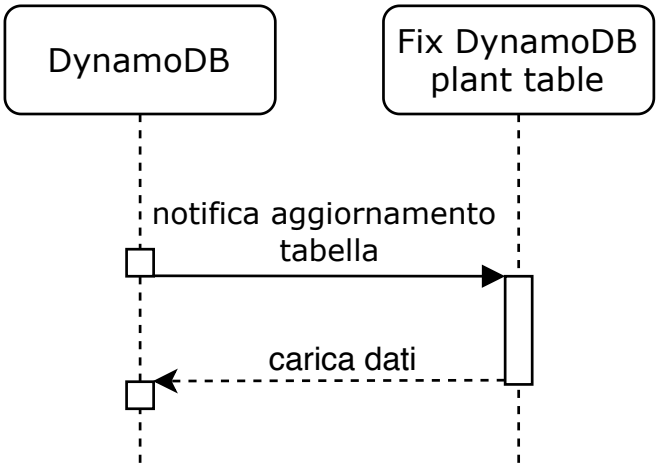


Figura 3.9: Diagramma di esecuzione della funzione *Fix DynamoDB plant table*

Trigger: *trigger* di *DynamoDB* per modifiche alla tabella *plant*

Input: non sono necessari file di *input*

Output: non sono prodotti file di *output*

Rule schedule creator

La funzione *Rule schedule creator* è stata creata allo scopo di evitare di effettuare una ricerca giornaliera sul *DB*, per trovare piante che fossero nel giorno di inizio o fine di un periodo fenologico. Si è pensato quindi di sfruttare il *trigger CloudWatch Events*, creando una regola per ogni data di inizio o fine periodo e di associarli alla funzione *publish event tweet*.

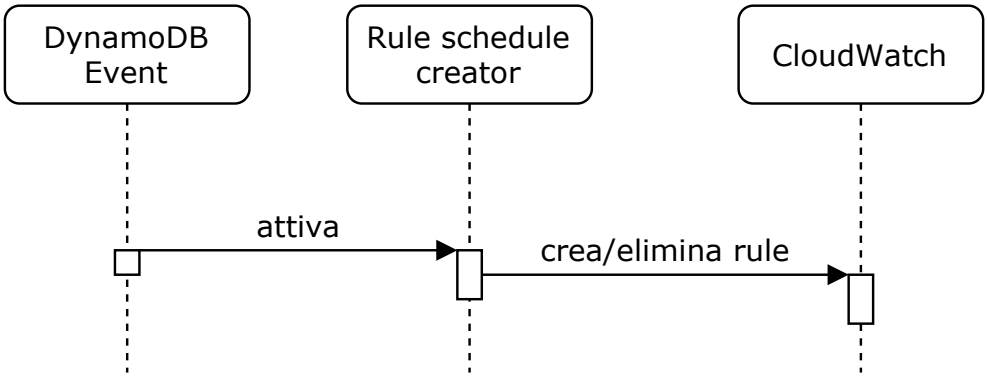


Figura 3.10: Diagramma di esecuzione della funzione *Rule schedule creator*

Trigger: *trigger* di *DynamoDB* per modifiche alla tabella *plant*

Input: non sono necessari file di *input*

Output: non sono prodotti file di *output*

Setting config

Setting confing è la funzione che funge da *backend* dell'applicazione *web* per la modifica dei file di configurazione e per l'inserimento e la modifica delle piante sul *DB*. Questa funzione riceve, attraverso *API Gateway*, le richieste dal *frontend*, recupera i dati richiesti da *DynamoDB* o da *S3* e risponde alla richiesta inviando i dati recuperati.
Lato *frontend* invece, è stato scritto uno *script JavaScript*, che gestisce le richieste e le risposte ricevute tramite *AJAX*. Il *frontend* verrà approfondito successivamente.

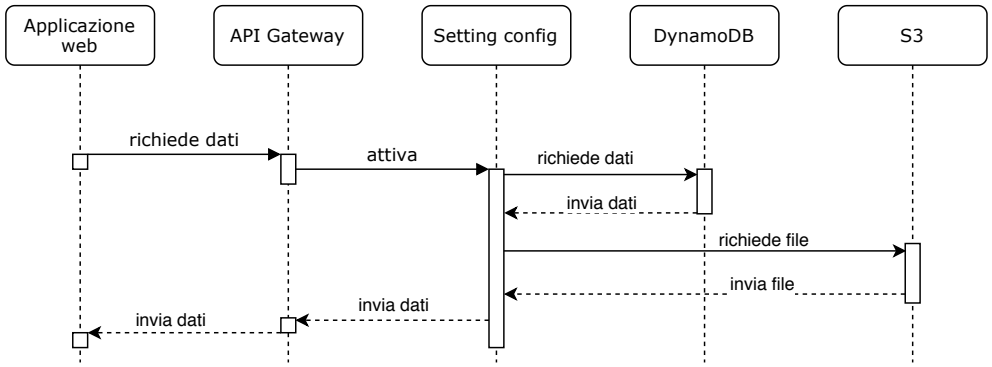


Figura 3.11: Diagramma di esecuzione della funzione *Setting config*

Trigger: *API Gateway* con l'*endpoint* configurato

Input: non sono necessari file di *input*

Output: non sono prodotti file di *output*

3.3 File system S3

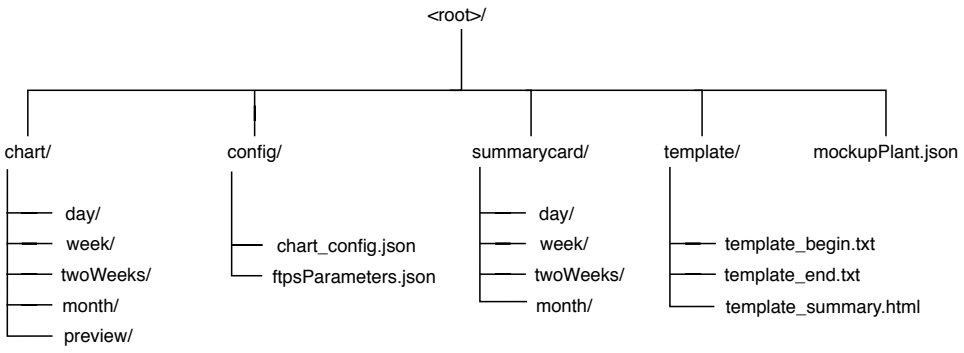


Figura 3.12: Organizzazione file su S3

Lo spazio su S3 è suddiviso in *Bucket*, cioè dei contenitori, che inglobano tutti gli oggetti. *AWS* non pone un limite al numero di oggetti che può contenere un *bucket*, impone solo il limite sulla dimensione del singolo oggetto (max 5TB). *Amazon S3* vanta molte caratteristiche utili, che lo rendono un'ottima scelta in caso si necessiti di un *cloud* professionale. Permette ad esempio di scegliere la classe di *storage*, ha una gestione integrata degli accessi, in modo da tenere i dati al sicuro e supporta le notifiche di eventi, usate per attivare le funzioni *lambda*.
Il *bucket* non è suddiviso in cartelle; è tuttavia possibile usare un sistema di cartelle "fittizie" attraverso le *key* (chiavi) degli oggetti. Si usa quindi, come chiave dell'oggetto, un percorso all'interno di un *file system* classico (ad es. la chiave: *<root>/config/chart_ config.json* indica che il file *chart_ config.json* si trova all'interno della cartella *<root>/config/chart/*).
Il *bucket* è stato organizzato in modo da avere una cartella *<root>*, configurabile all'interno delle *lambda*, che contiene tutti i *file* di *input* e *output* delle funzioni. Al secondo livello del *file system*, è situato il file *mockupPlant.json* che è la struttura di una pianta sul *DB*. Sono inoltre presenti le seguenti cartelle:

- **chart/** è la cartella in cui la funzione *Graph creator* carica i file *.png* dei grafici appena creati. La cartella è ulteriormente suddivisa nelle sottocartelle *day, week, twoWeeks e month*, una per ogni tipo di grafico generabile. I file sono inseriti all'interno della sottocartella corrispondente, con una chiave avente il seguente formato: *AAAA/MM/GG/nome_ grafico.png*, sono quindi inseriti all'interno di una sottocartella corrispondente alla data di creazione del grafico. Nella cartella *chart/* è presente anche la cartella *preview/*, che contiene solo il grafico creato dalla funzione *Setting config*, in caso di richiesta dell'anteprima del grafico;

- **config/** che contiene i file di configurazione delle funzioni *lambda*;

- **summarycard**/, suddivisa anch'essa in *day, week, twoWeeks e month*, contiene al suo interno le *card HTML* create per essere pubblicate su *Twitter*;
- **template**/ ha al suo interno i *template* per il testo dei *tweet* di inizio e fine periodo fenologico e il *template* per le *summary card*;

3.4 Struttura database

Come già anticipato, per questo progetto è stato usato il database di *Amazon DynamoDB*; è un *DB* non relazionale, con prestazioni e scalabilità elevate. Data la sua versatilità, può essere usato in progetti di qualsiasi dimensione, evitando di dover gestire il database in autonomia, con dispendio di risorse in termini di tempo (per le configurazioni, aggiornamenti, sicurezza, ..) e di costo (*provisioning* dell'*hardware*). Altre caratteristiche sono:

- le tabelle non hanno limiti ne sul numero di *item*, ne di *byte*;
- possibilità di creazione di indici secondari, per effettuare *query* su indici diversi e ugualmente veloci come le chiavi principali della tabella;
- elevata durabilità dei dati, con possibilità di backup dei dati inattivi;
- crittografia dei dati integrata.

DynamoDB, come di consueto, memorizza i dati in tabelle; all'interno delle tabelle, sono contenuti gli item, che sono set di attributi identificabili in modo univoco tra tutti gli altri item (sono l'equivalente delle tuple di un classico *RDMBS (Relational Database Management System)*). *DynamoDB* supporta due tipi di chiavi primarie:

- la chiave primaria: una semplice chiave primaria, composta da un attributo noto come *chiave di partizione*;
- chiave di partizione e chiave di ordinamento: è detta anche *chiave primaria composita*, perché è costituita da due attributi: chiave di partizione e chiave di ordinamento. *DynamoDB* usa la chiave di partizione per accedere alla partizione dello *storage* fisico in cui verranno salvati tutti gli *item* aventi la stessa chiave di partizione; usa invece la chiave di ordinamento per identificare lo specifico *item*.

Oltre alle *query* sulle chiavi primarie, è possibile effettuare *query* su eventuali indici secondari creati. Gli indici sono l'equivalente delle chiavi primarie, ma sono diversi dalle chiavi primarie specificate per la tabella.
In questo progetto, il *database*, è costituito da quattro tabelle:

- *plant*
- *devicedata*
- *plant_ device*
- *tweetIDs*

3.4.1 Tabella plant

Questa tabella contiene le informazioni delle piante monitorate dal sistema; all'interno della tabella sono salvate le seguenti informazioni:

- Nome della pianta;
- *UUID* della pianta;
- Date di inizio e fine periodo fenologico;
- *UUID* del *device* e numero del dendrometro collegato alla pianta;
- Coordinate del luogo in cui si trova la pianta;
- Specie e varietà della pianta

Queste informazioni, vengono inserite nel database (attraverso l'applicazione web o direttamente su *DynamoDB*) con la seguente struttura dati (la struttura dati è contenuta anche nel file *json* *<root>/mockupPlant.json*):

```
{
    "plantUUID": "",
    "name": "",
    "species": "",
    "variety": "",
    "period_begin": "",
    "period_end": "",
    "device": {
    "dendrometerCh": "-1",
    "deviceUUID": ""
    },
    "site": {
      "geometry": {
        "coordinates": {
          "0": "",
          "1": ""
        },
        "type": "Point"
      },
      "properties": {
        "name": ""
      },
      "type": "Feature"
    }
}
```

La tabella ha solo una chiave di partizione primaria, data dall'attributo *plantUUID*.

3.4.2 Tabella devicedata

Nella tabella *devicedata*, le funzioni *Store data from API* e *Store data from FTPS* inseriscono i dati provenienti dai dendrometri; La tabella ha la seguente struttura:

```
{
    "deviceUUID": "",
    "timestamp": "",
    "battery": "",
    "dendrometerCh01_Avg": "",
    "dendrometerCh01_Max": "",
    "dendrometerCh01_Min": "",
    "dendrometerCh02_Avg": "",
    "dendrometerCh02_Max": "",
    "dendrometerCh02_Min": "",
    "temperature": ""
}
```

La tabella ha come chiave di partizione primaria l'*UUID* del dispositivo che effettua il caricamento dei dati (attributo *deviceUUID*) e come chiave di ordinamento primaria l'attributo *timestamp*. Nella tabella vengono storicizzati tutti i dati inviati da dispositivo, ossia, i dati dei dendrometri, la tensione della batteria del dispositivo e la temperatura. Come si può notare, il dispositivo ha due solo dendrometri (*CH01* e *CH02*); è possibile collegarne altri, senza dover effettuare modifiche all'applicazione, grazie alla grande versatilità di *DynamoDB*, le cui tabelle non hanno una struttura rigida (sono obbligatorie solo la chiave primaria e la eventuale chiave di ordinamento), ma può essere potenzialmente variata per ogni elemento. La funzione *Graph creator* è stata progettata in modo da non avere vincoli sul numero di dendrometri collegati.

3.4.3 Tabella plant_device

Questa tabella funge da collegamento tra la tabella *plant* e la tabella *devicedata*; ha come chiave di partizione primaria *deviceUUID* e come chiave di ordinamento primaria *dendrometerCh* in modo da ottimizzare la ricerca quando la funzione *Graph creator* deve accedere ai dati relativi alla pianta. Si è rivelato necessario crearla per i seguenti motivi:

- questo approccio permette di avere i dati delle piante separati dai dati dei sensori;

- *DynamoDB* impone dei limiti sulla dimensione dei singoli elementi; non sarebbe stato possibile inserire in un unico elemento sia i dati delle piante, sia lo storico dei dati provenienti dai sensori;
- non sarebbe risultato conveniente copiare i dati delle piante all'interno di ogni elemento della tabella *devicedata*;

è stato quindi deciso un approccio più simile a un *DB* relazionale, che rende più complesse le operazioni di inserimento e prelievo dei dati, ma permette di avere dati più ordinati e meno spreco di spazio e di tempo per copiare tutti i dati delle piante all'interno degli elementi della tabella *devicedata*. In questo modo, l'eventuale aggiornamento delle informazioni delle piante è molto meno oneroso (occorre semplicemente modificare un elemento della tabella *plant*).
La tabella è strutturata come segue:

```
{
    "deviceUUID": "",
    "dendrometerCh": 1,
    "plantUUID": ""
}
```

3.4.4 Tabella tweetIDs

La tabella *tweetIDs* è usata per salvare gli ID dei *Tweet* postati, potendo così essere recuperati per eliminare il *tweet* esistente e postare il nuovo. La tabella ha la seguente struttura:

```
{
    "device_UUID": "",
    "chart_period": "",
    "battery": "",
    "temperature": "",
    "dendrometerCh1_2_Avg": "",
    "dendrometerCh1_2_Max": "",
    "dendrometerCh1_2_Min": ""
}
```

La tabella ha come chiave di partizione primaria l'*UUID* del dispositivo (attributo *device_ UUID*) e come chiave di ordinamento il tipo di periodo del grafico (*day, week, twoWeeks o month*) contenuto nell'attributo *chart_ period*. Tutti gli altri attributi contengono l'ID del corrispondente *tweet*.

Capitolo 4

Implementazione delle funzioni

Dopo aver presentato la struttura del progetto e come esso è stato sviluppato, si procede con la descrizione dell'implementazione delle funzioni. Tutte le funzioni lambda sono state scritte in *python*, con il supporto di ulteriori moduli, di cui ne verrà spiegato lo scopo. I servizi *AWS* sono stati gestiti in *python* attraverso la libreria *boto3*, che fornisce un *client* per la gestione di ogni servizio della piattaforma di *Amazon* [4].

4.1 Implementazione funzioni lambda

4.1.1 Graph creator

Lambda handler

La funzione *Graph creator*, come detto in precedenza, viene attivata solo da *trigger* di *CloudWatch Events*; un esempio del file *json* passato all'*handler* della funzione (attraverso il parametro *event*) è il seguente:

```
{
    "id": "",
    "detail-type": "Scheduled Event",
    "source": "aws.events",
    "account": "{{account-id}}",
    "time": "2019-08-31T04:00:00Z",
    "region": "us-east-1",
    "resources": [
    "arn:aws:events:us-east-1:123456789012:rule/day_activation"
    ],
    "detail": {}
}
```

L'*handler* sfrutta il nome della regola, contenuta nell'*ARN (Amazon Resource Names)* cioè un identificatore di risorse usato da *Amazon* e può essere, in base al servizio, nei seguenti formati [3]:

```
arn:partition:service:region:account-id:resource-id
arn:partition:service:region:account-id:resource-type/resource-id
arn:partition:service:region:account-id:resource-type:resource-id
```

Per un evento di *CloudWatch Events*, il tipo di risorsa è *rule*, mentre il *resurce-id* è il nome assegnato alla regola. I nomi delle regole sono stati creati inserendo il tipo di attivazione nel nome *(day, week, twoWeek o month)*, seguito dal termine *activation*; in questo modo, la funzione riesce a capire che tipo di grafico generare. Attraverso la data contenuta in *time* calcola quindi l'intervallo di date che comprendono il periodo specificato. L'*handler* si occupa poi di reperire da *DynamoDB* l'elenco dei dispositivi e di chiamare la funzione *chart* che di occupa di creare il grafico.

Funzione graph

É la funzione richiamata dall'*handler*; si occupa di reperire da *DynamoDB* i dati da graficare, poi lancia la funzione *plot_ two_ dendrometer* se nel file *chart_ config* il parametro *plotTwoDendrometer* ha valore 1, lancia la funzione *draw_ chart* altrimenti.

Definizione della funzione:

```
def chart(dynamodb,device,chart_path,begin_date,end_date,
↪   bucket_name,s3_client,chart_path_S3,
↪   chart_config,activation_type):
```

Parametri:

- **dynamodb:** istanza *client DynamoDB* con *boto3*
- **device:** *UUID* del dispositivo
- **chart_path:** cartella locale di appoggio per il *file* del grafico
- **begin_date**, **end_date:** date di inizio e di fine del periodo del grafico
- **bucket_name:** nome del *bucket S3* in cui caricare il grafico
- **s3_client:** istanza *client* per *S3* di *boto3*

- **chart_path_S3:** percorso in cui si vuole caricare il *file* all'interno del *bucket* (corrisponde alla prima parte della chiave che avrà il *file* su *S3*)
- **chart_config:** è il *file chart_config* trasformato in *dict*
- **activation_type:** è il tipo di grafico da generare (*day, week, twoWeeks, month*)

Funzione draw_chart

La funzione *daw_chart* crea il grafico e lo salva in locale (nello spazio accessibile dalla *lambda* in fase di esecuzione), prima di essere caricato su *S3*. Il file generato ha due possibili nomi:

- *<device_UUID>_<nome_variabile>_<plant_UUID>_numeroRandom*
  nel caso in cui la variabile da graficare sia un dendrometro
- *<device_UUID>_<nome_variabile>_numeroRandom*
  per tutte le altre variabili

In entrambi i casi, è stato inserito un numero *random*, generato tramite il modulo *random* di *Python*, necessario per forzare l'aggiornamento dell'anteprima della *card* su *Twitter*; se il nome del *file* non varia, *Twitter* non aggiorna l'anteprima della *card*. Il grafico è creato con la libreria **matplotlib**, una libreria *Python* che permette la creazione di molte tipologie di grafici [26].

Definizione delle funzione:

```
def draw_chart(variable,time,variable_name,fig_path,bucket_name,
↪  s3_client,chart_path_S3,plants_info,device_UUID,chart_config,
↪  activation_type,begin_date):
```

Parametri:

- **variable:** è il vettore di dati da graficare (asse y)
- **time:** è il vettore del tempo (asse x)
- **variable_name:** nome della variabile graficata (batteria, temperatura, ..)
- **fig_path:** cartella locale di appoggio per il file del grafico
- **bucket_name:** nome del *bucket S3* in cui caricare il grafico
- **s3_client:** istanza *client* per *S3* di *boto3*
- **chart_path_S3:** percorso in cui si vuole caricare il file all'interno del *bucket* (corrisponde alla prima parte della chiave che avrà il file su *S3*)

- **plants_info:** *dict* contenente le informazioni relative alla pianta
- **device_UUID:** *UUID* del dispositivo
- **chart_config:** è il *file chart_config* trasformato in *dict*
- **activation_type:** è il tipo di grafico da generare (*day, week, twoWeeks, month*)
- **begin_date:** date di inizio del periodo del grafico

Nota: è presente anche la funzione *plot_two_dendrometer* che è molto simile a *draw_chart*, con l'unica differenza che crea grafici a doppia variabile; viene usata solo per coppie di dendrometri ed è attivabile settando il *flag plotTwoDendrometer* del *file* di configurazione.

Funzione upload_chart

Si occupa di caricare il *file* del grafico su *S3*; usa il *client* per *S3* della libreria *boto3*.

Definizione della funzione:

```
def upload_chart(bucket_name,s3_client,chart_name,chart_path_S3,
↪  chart_path_local):
```

Parametri:

- **bucket_name:** nome del *bucket S3* in cui caricare il grafico
- **s3_client:** istanza *client* per *S3* di *boto3*
- **chart_name:** nome che il grafico avrà su *S3*; la chiave completa sarà *chart_path_S3/chart_name*
- **chart_path_S3:** percorso in cui si vuole caricare il file all'interno del *bucket* (corrisponde alla prima parte della chiave che avrà il file su *S3*)
- **chart_path_local:** cartella locale di appoggio per il *file* del grafico

Funzione organize_device_info

É una funzione creata con lo scopo di organizzare i dati da inserire nei grafici in un unico *dict*, con una *list* per ogni campo. La struttura del *dict* è:

```
device_data{
    'timestamp':[]
    'temperature':[]
    'umidity':[]
    'battery':[]
    'dendrometerCh1':[]
    'dendrometerCh2':[]
}
```

Questa struttura rende il codice relativo al *plot* del grafico molto più semplice, dato che gli elementi del *dict* sono già pronti per essere "plottati"; *timestamp* è usato invece come asse dei tempi.

Funzione get_plant_info

Si occupa di estrarre dal *DB* le informazioni relative alla pianta; per farlo deve usare la tabella di collegamento *plant_device* per estrarre l'*UUID* della pianta, partendo dall'UUID del dispositivo. Successivamente la funzione organizza i dati in un *dict* che ha come indici il numero del dendrometro e come valori i dati delle piante.

4.1.2 Publish event tweet

Publish event tweet, come già anticipato, è la funzione che si occupa di pubblicare *tweet*. Per la gestione dell'*account* di *Twitter* e per effettuare le operazioni sui *tweet* è stata usata la libreria per *Python Twython*. *Twython* è un *wrapper* per le *API* di *Twitter*, scritta in *python* puro; consente quindi di usare direttamente le *API* di *Twitter* in *Python*. [30]

Lambda handler

L'*handler* della funzione *Publish event tweet* gestisce l'attivazione da parte di due tipi di *trigger*:

- *trigger* per creazione oggetto su *S3*
- *trigger* di *CloudWatch Events*

Nel caso di **trigger su un evento di S3**, cioè quando un nuovo grafico con prefisso *<root>/chart/<day|week|twoWeeks|month>* e suffisso *.png* viene caricato su *S3*, il parametro *event* risulterà così composto:

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
        ↪  mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "ortobotanico",
```

```
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::example-bucket"
        },
        "object": {
          "key": "test_2/chart/week/
          ↪  68c4cdd9-07e4-4b5c-bceb-fe5660617edc_umidity.png",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

Il *JSON* dell'evento, in questo caso, contiene informazioni su *S3*, quindi sul *bucket* e sull'oggetto caricato; l'*handler* estrae dall'*event* la chiave (parametro *key*), che servirà per ricavare l'*UUID* del *device*, il nome della variabile graficata e l'eventuale *UUID* della pianta, nel caso in cui la variabile sia un dendrometro. Dopo aver ricavato queste informazioni, chiama la funzione *chart_ tweet*.

Nel caso di attivazione tramite **trigger di CloudWatch**, il parametro *event* sarà così composto:

```
{
    "id": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",
    "detail-type": "Scheduled Event",
    "source": "aws.events",
    "account": "{{account-id}}",
    "time": "1970-01-01T00:00:00Z",
    "region": "us-east-1",
    "resources": [
    "arn:aws:events:us-east-1:123456789012:rule/
    ↪  156c03d5-dae1-435e-8c16-35b5960ef27f_begin"
    ],
    "detail": {}
}
```

L'*handler* estrae quindi il nome della regola (corrisponde al codice dopo *rule/*) per ricavarne l'*UUID* della pianta e il momento dell'evento fenologico (inizio o fine); quindi chiama la funzione *status_ tweet*.

Funzione chart_tweet

É la funzione che crea la *card* da pubblicare su *Twitter*; prima di procedere alla creazione della *card*, la funzione deve effettuare il *download* del *template* della *card* da *S3*. Successivamente, crea la *card* e la carica su *S3*; è fondamentale che la *card* sia caricata con l'attributo *ContentType*, attributo relativo all'oggetto *S3* che dichiara il tipo di contenuto dell'oggetto, altrimenti il *file* viene salvato come *byte* e non come semplice testo, rendendo la pagina HTML illeggibile dai *browser*. Una volta pronta la *card*, viene recuperato l'ID del *tweet* preesistente dalla tabella *tweetIDs* di *DynamoDB* (tramite la funzione *getTweetID*) ed eliminato, con una chiamata alla funzione *delete_tweet*; viene quindi chiamata la funzione *publish_tweet* che pubblica il nuovo *tweet* con l'*URL* della *card* appena creata. Infine, l'*ID* del nuovo *tweet* viene salvato su *DynamoDB* con la funzione *uploadTweetID*.

Definizione della funzione:

```
def chart_tweet(bucket,object_key,s3,summary_path_S3,
↪  summary_template_S3):
```

Parametri:

- **bucket:** nome del *bucket* di destinazione
- **object_key:** chiave dell'oggetto caricato su *S3*; è la chiave completa del grafico
- **s3:** *client* per *S3* di *boto3*
- **summary_path_S3:** è il percorso su *S3* in cui si vuole salvare la *card HTML*
- **summary_template_S3:** è il percorso del file di *template* della *card*

Funzione status_tweet

É la funzione che crea il corpo del *tweet* nel caso si sia verificato un evento di inizio o fine del periodo fenologico. La funzione effettua il *download* del *template* da *S3* ed effettua anche il recupero dei dati relativi alla pianta, dalla tabella *plant* di *DynamoDB*, grazie alla funzione *get_plant_info*; quindi costruisce il corpo del *tweet*, chiamando la funzione *build_tweet* e pubblica il *tweet* tramite la funzione *publish_tweet*.

Definizione della funzione:

```
def status_tweet(bucket,root,s3,plant_uuid,event_type):
```

Parametri:

- **bucket:** nome del *bucket* di destinazione
- **root:** è la cartella principale del progetto (*root*) su *s3*
- **s3:** *client* per *S3* di *boto3*
- **plant_uuid:** *UUID* della pianta
- **event_type:** tipo di evento (inizio o fine)

Funzione publish_tweet

É la funzione che pubblica i *tweet*; è stata creata per pubblicare entrambi i tipi di *tweet*. La funzione crea un'istanza della libreria *Twython*, fornendo alla libreria le chiavi dell'account *developer* di *Twitter* (*consumer key* e *access token*). Viene poi usato il metodo *update_status* per pubblicare il *tweet*.

Definizione della funzione:

```
def publish_tweet(tweet,local_chart_path=None):
```

Parametri:

- **tweet:** è un *dict* contenente le informazioni da inserire nel *tweet* (testo e geolocalizzazione)
- **local_chart_path:** è il percorso di un eventuale file da allegare come immagine al *tweet*

Funzione build_summary

La funzione in esame sfrutta la libreria *jinja* [23] per costruire la *summary card*, sulla base del *template*. La funzione scarica da *S3* il *template* delle *card*, inserisce nei *placeholder* l'*URL* dell'immagine, il titolo e la descrizione; il *template* renderizzato (attraverso il metodo *render* di *jinja*) verrà restituito alla funzione chiamante sotto forma di semplice stringa *Unicode*.

Definizione della funzione:

```
def build_summary(summary_info,summary_template_S3):
```

Parametri:

- **summary_info:** è un *dict* contenente le informazioni da inserire nella *card* (testo, descrizione e *URL* dell'immagine)
- **summary_template_S3:** è il percorso del *file* di *template* della *card* su *S3*

Funzione build_tweet

É la funzione che genera il corpo del *tweet* a partire dal *template* caricato su *S3*; anche in questo caso si usa la libreria *jinja* per renderizzare il *template*. La funzione renderizza il *template* attraverso il metodo *render* di *jinja*, sostituendo ai *placeholder* le informazioni contenute in *plant_info*, che conterrà le seguenti informazioni: nome della pianta, specie, varietà, data di inizio e fine periodo fenologico, nome e coordinate del luogo in cui si trova la pianta. Infine viene restituito alla funzione chiamante il corpo del *tweet* in formato *str*.

Definizione della funzione:

```
def build_tweet(plant_info,template_local_path):
```

Parametri:

- **plant_info:** *dict* contenente le informazioni da inserire nel *tweet*
- **template_local_path:** è il percorso del file di *template* già scaricato nello spazio riservato alla *lambda*

Funzione get_plant_info

Recupera dalla tabella *plant* di *DynamoDB* le informazioni sulla pianta, attraverso l'*UUID* fornitogli. Restituisce un *dict* contenente tutti gli attributi relativi alla pianta.

Funzione uploadTweetID

Effettua il salvataggio dell'ID del *tweet* appena pubblicato nella tabella *tweetIDs* di *DynamoDB*.

Funzione getTweetID

Recupera l'ID del *tweet* esistente da *DynamoDB* e lo restituisce alla funzione chiamante.

Funzione delete_tweet

É la funzione che elimina il *tweet* corrispondente all'ID passatogli.

4.1.3 Store data from FTPS

Come già anticipato, per effettuare il prelievo dei dati da un *server FTPS* e salvarli su *DynamoDB* è stata creata la funzione *Store data from FTPS*; la funzione deve quindi collegarsi al *server FTPS*, effettuare il *download* del *file CSV* relativo al giorno precedente. É bene ricordare che la funzione viene attivata da eventi giornalieri generati da *CloudWatch*; ogni mattina dev'essere recuperato il file *CSV* dal *server*, che contiene i dati fino alla mezzanotte del giorno prima.

Lambda handler

L'*handler* di questa funzione preleva dal parametro *event* la data di attivazione, effettua il *download* del *file ftpsParameters.json* contenente i parametri per l'accesso al server ed effettua una chiamata alla funzione *downloadCsv* passando entrambi i parametri. Successivamente, conclusasi l'esecuzione di *downloadCsv*, effettua il salvataggio di tutte le righe del file *CSV* appena scaricato su *DynamoDB*. Un file *CSV* è un semplice file di testo, in cui i dati di una riga sono separati da virgole. I dati estratti dal file vengono quindi caricati nella tabella *devicedata*.

Funzione downloadCsv

É la funzione che si occupa di effettuare il download del file *CSV* dal *server FTPS*; utilizza la libreria *ftplib* per stabilire la connessione con il *server* ed effettuare il *download* del *file*. Il nome dei *file* sul *server* contiene la data di caricamento, quindi la funzione cerca il *file* con la data interessata all'interno del *server*.

Definizione della funzione:

```
def downloadCsv(cronDate,localFile,ftpsParams):
```

Parametri:

- **cronDate:** stringa della data in formato "AAAAMMGG"
- **localFile:** è il percorso in cui la funzione deve effettuare il *download* del file
- **ftpsParams:** *dict* contenente i parametri di connessione al *server*

4.1.4 Store data from API

L'implementazione di questa funzione è molto semplice; tutto il lavoro è svolto dall'*handler* e non sono necessarie particolari librerie (oltre a *boto3*)

Lambda handler

L'*handler* della funzione riceve, attraverso il parametro *event*, il *json* del *trigger API Gateway* che corrisponde al corpo di una richiesta effettuata all'*API*. In questo caso, il corpo della richiesta, è un *json* che contiene i dati da inserire all'interno della tabella *devicedata*; la funzione, organizza i dati prelevati dal parametro *event* in un *dict* e li inserisce nella tabella di *DynamoDB*.

4.1.5 Fix DynamoDB plant table

Lambda handler

La funzione viene attivata con *trigger* di *DynamoDB* che invia un evento in caso di inserimento, modifica o cancellazione effettuati sulla tabella *plant*. Il parametro *event*, in caso di inserimento, è cosi costituito:


```
{
  "Records": [
    {
      "eventID": "c4ca4238a0b923820dcc509a6f75849b",
      "eventName": "INSERT",
      "eventVersion": "1.1",
      "eventSource": "aws:dynamodb",
      "awsRegion": "us-east-1",
      "dynamodb": {
        "Keys": {
          "plantUUID": {
            "S": "156c03d5-dae1-435e-8c16-35b5960ef27f"
          }
        },
        "NewImage": {
          "device": {
            "M": {
              "dendrometerCh": {
                "S": "-1"
              },
              "deviceUUID": {
                "S": "68c4cdd9-07e4-4b5c-bceb-fe5660617edc"
              }
            }
          },
          "name": {
            "S": "Pianta A"
          },
          "period_begin": {
            "S": "30-11"
          },
          "period_end": {
            "S": "30-12"
          },
          "plantUUID": {
            "S": "156c03d5-dae1-435e-8c16-35b5960ef27f"
```

```
      },
      "site": {
        "M": {
          "geometry": {
            "M": {
              "coordinates": {
                "M": {
                  "0": {
                    "S": "13.52041482925415"
                  },
                  "1": {
                    "S": "43.61999974379523"
                  }
                }
              },
              "type": {
                "S": "Point"
              }
            }
          },
          "properties": {
            "M": {
              "name": {
                "S": "Ancona"
              }
            }
          },
          "type": {
            "S": "Feature"
          }
        }
      },
      "species": {
        "S": "margherita"
      },
      "variety": {
        "S": "margherita gialla"
      }
    },
    "ApproximateCreationDateTime": 1428537600,
    "SequenceNumber": "442158450000000000017450439091",
    "SizeBytes": 26,
    "StreamViewType": "NEW_AND_OLD_IMAGES"
```

```
      },
      "eventSourceARN":
       ↪  "arn:aws:dynamodb:us-east-1:123456789012:table/
       ↪  ExampleTableWithStream/stream/2015-06-27T00:48:05.899"
    }
  ]
}
```

L'*handler*, attraverso il parametro *eventName*, discrimina i casi tra inserimento/-modifica e rimozione.
Nel caso di inserimento o modifica, verrà chiamata la funzione *plant_ update*.
Nel caso di rimozione, l'*handler* chiama la funzione *delete_ old_ item*.

Funzione plant_update

La funzione preleva dalla tabella *plant_ device* del *DB* l'elemento corrispondente all'*UUID* del *device* e al canale del dendrometro associati alla pianta; se l'elemento è presente, lo aggiorna con i nuovi dati, prelevati dal parametro *event*; se l'elemento non c'è, viene inserito nella tabella.

Definizione della funzione:

```
def plant_update(dynamodb,ev_data):
```

Parametri:

- **dynamodb:** istanza del *client DynamoDB* di *boto3*
- **ev_data:** è il contenuto del parametro *dynamodb*, della variabile *event*, che contiene tutte le operazioni svolte sul *DB*

Funzione delete_old_item

É la funzione che elimina l'elemento dalla tabella *plant_ device* nel caso in cui venga eliminata la pianta dalla tabella *plant*.

Definizione della funzione:

```
def delete_old_item(dynamodb,ev_data):
```

Parametri:

- **dynamodb:** istanza del *client DynamoDB* di *boto3*
- **ev_data:** è il contenuto del parametro *dynamodb*, della variabile *event*, che contiene tutte le operazioni svolte sul *DB*

4.1.6 Rule schedule creator

Lambda handler

Anche in questo caso, la funzione *lambda* viene attivata da un evento di inserimento, modifica o rimozione, effettuati sulla tabella *plant* di *DynamoDB*. La struttura del *json* dell'evento è la stessa riportata per la funzione *Fix DynamoDB plant table*. L'*handler* distingue quindi i due casi: inserimento/modifica e rimozione e chiama la funzione *set_event_rule* nel caso di inserimento/modifica; chiama la funzione *del_event_rule* altrimenti.

Funzione set_event_rule

É la funzione che crea le regole *CloudWatch Events* e le associa alla funzione *publish_event_tweet*. Questa funzione crea due regole (una per il *trigger* di inizio periodo e una per quello di fine) tramite la funzione *create_rule*, che si occupa di estrarre il giorno e il mese di inizio o fine periodo dal *json* di *DynamoDB* e creare la regola *CloudWatch*. Successivamente, in caso di inserimento, la funzione *set_event_rule* crea i permessi per consentire ai *trigger* appena creati di eseguire la *lambda*. I permessi vengono creati con la funzione *add_lambda_permission*, che funge da semplice *wrapper* per il metodo *add_permission* del *client CloudWatch* di *boto3*.

Definizione della funzione:

```
def set_event_rule(ev_type,lambda_arn,lambda_client,event_client,
↪  ev_data):
```

Parametri:

- **ev_type:** tipo di evento (inserimento, modifica o rimozione)
- **lambda_arn:** è l'*arn* della funzione *publish_event_tweet*, destinataria della regola di attivazione *CloudWatch Events*
- **lambda_client:** *client* di *boto3* per *lambda*
- **event_client:** *client* di *boto3* per *CloudWatchEvents*
- **ev_data:** è il contenuto del parametro *dynamodb*, della variabile *event*, che contiene tutte le operazioni svolte sul *DB*

Funzione del_event_rule

Si occupa di rimuovere i permessi e le regole di *CloudWatch*, in caso di rimozione della pianta dal *DB*.

Definizione della funzione:

```
def del_event_rule(lambda_arn,lambda_client,event_client,ev_data):
```

Parametri:

- **lambda_arn:** è l'*arn* della funzione *publish_event_tweet*, destinataria della regola di attivazione *CloudWatch Events*
- **lambda_client:** *client* di *boto3* per *lambda*
- **event_client:** *client* di *boto3* per *CloudWatchEvents*
- **ev_data:** è il contenuto del parametro *dynamodb*, della variabile *event*, che contiene tutte le operazioni svolte sul *DB*

4.1.7 Setting Config

La funzione *Setting config* è la funzione che funge da *backend* per l'applicazione web di configurazione. Essa viene attivata solamente da *trigger API Gateway*, che mette a disposizione l'*endpoint* su cui il *frontend* effettua le richieste.

Lambda handler

L'*handler* di questa *lambda* si occupa di chiamare la funzione corretta, a seconda del tipo di richiesta fatta dal *frontend*. Al termine dell'esecuzione della funzione selezionata, l'*handler* invia la risposta alla richiesta attraverso il *return*. Un esempio di richiesta effettuata attraverso *API Gateway* è dato dal seguente *file json*:

```
{
    "reqType":"write",
    "objType":"template",
    "data":{
        "S3TemplateContent":"La pianta {{name}} della specie
        ↪  {{species}}, varietà {{variety}}, è entrata nel suo
        ↪  periodo! Il suo periodo va da {{period_begin}} a
        ↪  {{period_end}}; la pianta è situata: {{site}}.",
        "S3FileKey":"test_2/template/template_begin.txt"
    }
}
```

Il parametro *reqType* indica il tipo di richiesta effettuata; può avere come valore: *write, read, insert o chartPreview*.
"write" si usa per salvare su *cloud* le modifiche effettuate all'oggetto specificato. Gli oggetti (parametro *objType*) possono essere:

- **plant** nel caso in cui si voglia scrivere le modifiche effettuate ad una pianta su *DB*; nel parametro *data* sarà quindi contenuto l'intero *json* della pianta da caricare su *DynamoDB*. Verrà quindi chiamata la funzione *put_plant_info* che effettuerà la modifica.

- **configFile** se da aggiornare c'è un file di configurazione (sarebbero i file contenuti su *S3* in *<root>/config/*). La funzione attivata è *put_configTemplateFile*.

- **template** se una modifica ad un *file* di *template* è da salvare; anche in questo caso, il contenuto del file si troverà nel parametro *data* e verrà chiamata la funzione *put_configTemplateFile* per effettuare il caricamento del *file* su *S3*.

Come è possibile notare dall'esempio, nel caso in cui l'oggetto da modificare sia un *file* di configurazione o un *template*, il parametro *data* è costituito dal parametro *S3FileKey*, che contiene la chiave del file su *S3* e dal parametro *S3TemplateContent* che contiene il testo del *template*.
"read" si usa invece per consentire all'applicazione web di leggere dati da *S3* o *DynamoDB*. Anche in questo caso il parametro *objType* può assumere come valore: *plant, configFile o template*. Nel caso in cui la richiesta sia una lettura, invece del parametro *"data"* sarà presente nella richiesta il parametro *objName* che contiene il nome del *file* o *UUID* della pianta che si vuole leggere. É bene notare che *"objName"* può assumere anche il valore *list*, che permette di ottenere una lista di tutti gli oggetti del tipo richiesto.
"insert" è usato per inserire una nuova pianta nel *DB*; in realtà la funzione attivata non inserisce la pianta nel *DB* ma recupera da *S3* il modello che deve avere un elemento della tabella *plant*, in modo da poter inserire tutte le piante con lo stesso modello. L'inserimento vero e proprio sul *DB* viene fatto attraverso una richiesta di *"write"*.
"chartPreview" si usa per richiedere un'immagine di anteprima dell'aspetto del grafico.

Funzione get_configTemplateFile

É la funzione che si occupa del *download* del file di configurazione o *template* richiesto; nel caso in cui il parametro *objName* ha valore *list*, la funzione genera una lista di tutti i file contenuti all'interno della cartella *config* o *template* di *S3*.

Definizione della funzione:

```
def get_configTemplateFile(objectType,bucket,folder,fileName):
```

Parametri:

- **objectType:** è il tipo di oggetto da scaricare (*template* o *config*)
- **bucket:** nome del *bucket* utilizzato su *S3*
- **folder:** percorso della cartella *config* o *template* su *S3*
- **fileName:** è il nome del *file* da scaricare, contenuto nel parametro *"objName"*

Funzione get_plant_info

É la funzione che recupera da *DynamoDB* i dati della pianta specificata. Se il valore di *"objName"* è *list*, vengono estratti dal *DB* tutti gli *UUID* e i nomi delle piante presenti nella tabella.

Definizione della funzione:

```
def get_plant_info(objName):
```

Parametri:

- **objName:** è l'*UUID* della pianta di cui recuperare i dati

Funzione get_plant_mockup

Recupera il file di *mockup* da *S3*.

Definizione della funzione:

```
def get_plant_mockup(bucket,fileName):
```

Parametri:

- **bucket:** nome del *bucket* di *S3* utilizzato
- **fileName:** è la chiave del file di *mockup* su *S3*

Funzione put_configTemplateFile

É la funzione che inserisce il file di configurazione o *template* su *S3*.

Definizione della funzione:

```
def put_configTemplateFile(bucket,objectType,data):
```

Parametri:

- **bucket:** nome del *bucket* utilizzato su *S3*
- **objectType:** è il tipo di oggetto da caricare (*template* o *config*)
- **data:** contenuto del *file* da caricare su *S3*

Funzione put_plant_info

Si occupa di caricare su *DynamoDB* i dati della pianta. La funzione viene chiamata sia in caso di inserimento di un nuovo elemento nella tabella, sia in caso di modifica; sarà la funzione *put_item* di *boto3* a inserire un nuovo elemento, se non presente oppure modificare l'elemento preesistente.

Definizione della funzione:

```
def put_plant_info(data):
```

Parametri:

- **data:** dati della pianta da inserire nella tabella

Funzione create_preview

É la funzione che crea un grafico, con la libreria *matplotlib*, utilizzando, per le variabili da graficare (ordinata del grafico), numeri *random* e come ascissa un vettore di numeri da 1 a 31. Il grafico creato serve da anteprima per prendere visione delle effettive modifiche effettuate al *file* di configurazione del grafico. Dopo aver creato l'immagine, la carica su *S3* nel percorso *<root>/chart/preview*. Nel *json* della richiesta, sarà presente il parametro *chartConfig*, che è il contenuto del nuovo *file* di configurazione del grafico.

Definizione della funzione:

```
def create_preview(chart_path_S3,bucket_name,chart_config):
```

Parametri:

- **chart_path_S3:** è il percorso su S3 in cui inserire il file *.png* del grafico
- **bucket_name:** è il nome del *bucket* di *S3* usato
- **chart_config:** è il contenuto del parametro *"chartConfig"*

4.2 Applicazione web

Come si può intuire, per modificare una impostazione contenuta in un *file* di configurazione o modificare dati relativi a una pianta è necessario agire direttamente sui file, modificarli e poi caricarli su S3, oppure, nel caso della modifica dati di una pianta, sarebbe necessario accedere a *DynamoDB* ed effettuare la modifica alla tabella. Per facilitare l'accesso alle configurazioni, considerando che l'applicazione non verrà usata dai programmatori ma da terze parti, è stata creata una interfaccia web per consentire operazioni sui *file* di configurazione. In particolare l'interfaccia web creata consente le seguenti operazioni:

- modifica di tutti i *file* di configurazione (presenti su *S3* in *<root>/config/*)
- modifica di tutti i *file* di *template* (presenti su S3 in *<root>/template*)
- modifica di tutte le piante presenti nella tabella *plant* del *DB*
- inserimento di una nuova pianta sul *DB*

Per lo sviluppo di queste *feature* è stata creata quindi un'applicazione web con una semplice interfaccia.
Come di consueto, le applicazioni web sono costituite da un *backend*, cioè il programma che resta in esecuzione sul server e gestisce le richieste effettuate dal *client* e da un *frontend*, cioè l'applicazione che viene eseguita sul *client* da un comune *browser web*. In questo caso, è stato usato sempre l'approccio *serverless*, quindi il *backend* non è sempre in esecuzione sul *server*, ma si attiva solo in caso di richiesta; per questa applicazione, il *backend* è costituito dalla funzione *lambda Setting config*, che, come descritto poc'anzi, viene attivata attraverso una notifica di *API Gateway* e gestisce la richiesta effettuata dal *frontend*. I servizi di *Amazon* includono anche la gestione degli utenti e quindi anche l'autenticazione e l'autorizzazione per l'esecuzione delle funzioni; il servizio che svolge queste funzioni è *Cognito*. Per la gestione del *login* e delle autorizzazioni, *Amazon* fornisce la libreria di *Cognito* per *JavaScript*, che si occupa di recuperare i *token* per l'accesso alla funzione *lambda*.
Il *frontend*, invece, è costituito da una classica interfaccia web, che è stata caricata su *S3*. Il *browser* esegue quindi i *file* che risiedono su *S3*.

Descrizione del frontend

Il *frontend* è caratterizzato, come anticipato, da una interfaccia web costituita da una pagina *HTML* statica e da uno *script JavaScript*.É presente inoltre una pagina *HTML* per il *login*.
La pagina di *login* si presenta così:

Figura 4.1: *Screenshot* della pagina di *login*

É quindi una semplice pagina per l'inserimento delle credenziali dell'utente. Dopo aver effettuato il *login*, si viene reindirizzati sulla pagina principale dell'applicazione, costituita da una pagina *HTML* che funge da "contenitore", formata da due sezioni: sezione di sinistra ha un menu per selezionare il tipo di dati da modificare; nella sezione sulla destra verranno caricate invece le informazioni richieste di volta in volta al *backend*.

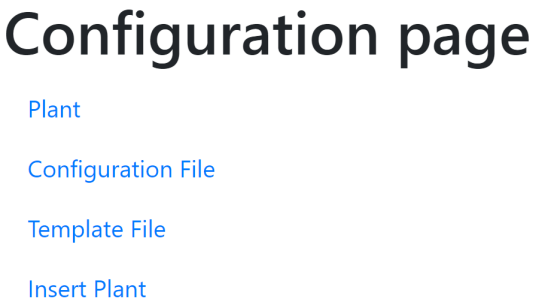


Figura 4.2: *Screenshot* del menu (sezione di sinistra)

Effettuando un *click* su un elemento del menu, verrà effettuata la richiesta della lista corrispondente, che verrà visualizzata nell'area sottostante al menu. Le possibili richieste sono: lista dei *file* di configurazione (figura 4.3), lista dei *template* (figura 4.4) e lista delle piante (figura 4.5).

Cliccando su un elemento della lista, verrà visualizzato, sulla sezione destra della pagina, un *form* per la modifica. In figura 4.6 un esempio di *form*.

Figura 4.3: Lista *file* di configurazione

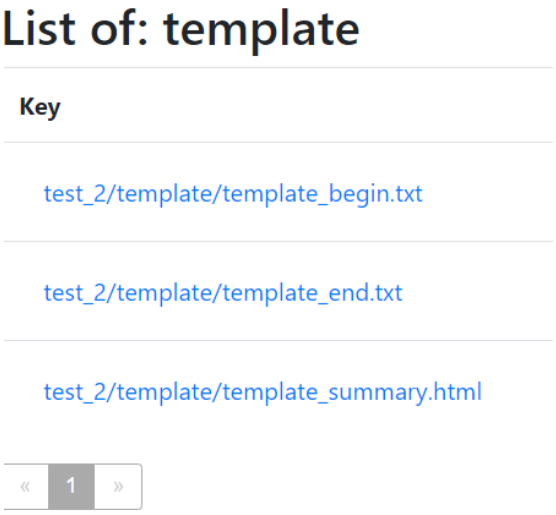


Figura 4.4: Lista *template*

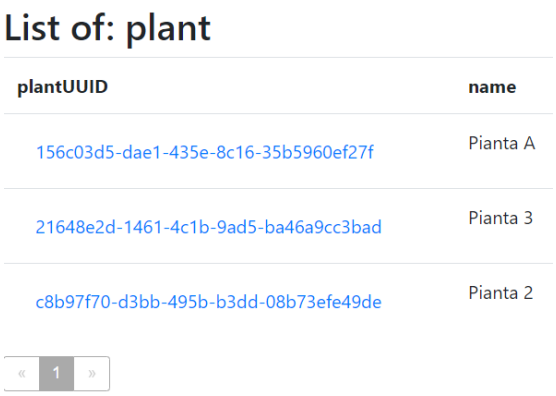


Figura 4.5: Lista delle piante

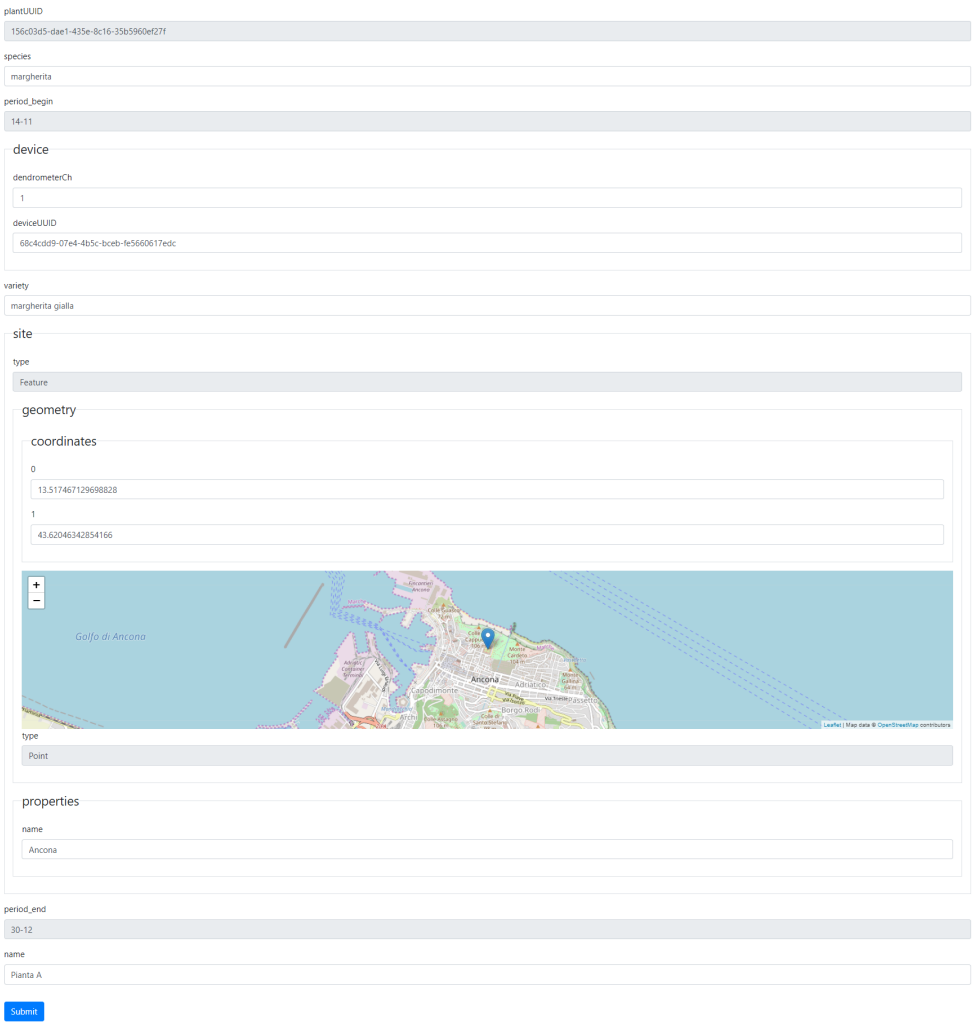


Figura 4.6: *Screenshot* del *form* di modifica della pianta

4.2.1 Implementazione HTML/CSS

Come già anticipato, la pagina *HTML* contiene solo due sezioni. La sezione di sinistra contiene un elenco puntato che costituisce il menu; la sezione di destra contiene un *"box"* che sarà la destinazione delle *form*. Per le regole di stile, invece di creare un foglio *CSS* da zero, è stato usato *bootstrap*, un *toolkit* che include un gran numero di elementi di stile creati con *jQuery* [24]. *Bootstrap* consente di velocizzare lo sviluppo della componente grafica della pagina *HTML*, dato che non sarà più necessario costruire tutti gli effetti manualmente, ma si potranno usare direttamente quelli inclusi in *bootstrap*. Il *toolkit* usato include inoltre regole *CSS* già pronte per creare un sito *responsive*[21].
La pagina *HTML* dell'applicazione è contenuta nel file *menu.html*; è presente anche il file *login.html* che contiene il *form* per eseguire il *login*.

4.2.2 Implementazione JavaScript

Lo *script JavaScript* è invece più complesso, perché deve gestire le richieste (effettuare la richiesta e inserire i dati ricevuti nella pagina *HTML*). É contenuto nel *file menu.js* ed è costituito da una serie di funzioni, create per effettuare le richieste e gestire le risposte provenienti dal server. Le funzioni sfruttano la comunicazione asincrona con il *server*, viene usata infatti la tecnica *AJAX (Asynchronous JavaScript and XML)*, i dati vengono quindi richiesti al *server* al momento del bisogno e non solo in fase di caricamento della pagina web. Grazie a questo approccio è stato possibile quindi costruire un sito web costituito da una sola pagina *HTML* e non da un insieme di pagine statiche collegate tra loro.
Appena ci si collega al sito, se non è già in atto una sessione, o la sessione attiva è scaduta, si viene reindirizzati alla pagina di *login*, nella quale inserire le credenziali per l'accesso al sito. Non è stata creata una pagina di registrazione, poiché gli utenti vengono creati dall'amministratore del sito direttamente tramite *Cognito*. La gestione del *login* viene fatta dallo *script cognito-auth.js* fornito da *Amazon*, che include tutte le funzioni necessarie alla verifica dell'identità e all'ottenimento dei *token* di autorizzazione per effettuare le richieste alla funzione *lambda*. Il file *cognito-auth.js* sfrutta la libreria di *Amazon Congito*, contenuta nei file *aws-cognito-sdk.min.js* e *amazon-cognito-identity.min.js* [12].
Se il *login* va a buon fine, viene caricata la pagina *menu.html*, che visualizza il menu di scelta del tipo di elemento da modificare; alle voci del menu è stata associata la funzione *requestList* che effettua alla funzione *lambda* una richiesta con il seguente *file json*:

```
{
    reqType: 'read',
    objType: 'plant'|'configFile'|'template',
    objName: 'list'
}
```

La risposta del *server*, in caso di successo, viene gestita dalla funzione *writeList*, che si occupa di inserire nella pagina *HTML*, nella sezione sottostante il menu, la lista inviata dal *server*. La lista viene visualizzata attraverso la libreria di paginazione *Pagination.js* [28] che si occupa di effettuare la paginazione degli elementi, in modo da non creare una lista troppo lunga e di difficile utilizzo.
Alla voce *"Insert Plant"* del menu è associata invece la funzione *requestInsert*, che invia al *backend* la richiesta del *file* di *mockup* della pianta (come descritto nella sezione relativa alla funzione *"Setting config"*). Il *json* del *mockup* ricevuto dal *server* verrà poi inserito all'interno della pagina *HTML* dalla funzione *writeInfo*.
Agli elementi della lista viene associata la funzione *readData*, che invierà una richiesta contenente il seguente *json*:

```
{
    reqType: 'read',
    objType: 'plant'|'configFile'|'template',
    objName: <nome oggetto>
}
```

Il parametro *"objName"* è dato dal nome dell'oggetto che compare sulla lista. La risposta ottenuta sarà gestita dalla funzione *writeInfo*.
La funzione *writeInfo* è la funzione che inserisce all'interno della pagina *HTML*, nella sezione di destra, la *form* di modifica dell'elemento. La scrittura del codice *HTML* della *form*, viene effettuata dalla funzione *createForm*. Dato che dalla funzione *lambda* viene inviato un *json* contenente i dati, sia nel caso di una pianta, sia nel caso di un *file* di configurazione (sono già dei *file json*), la funzione *createForm* crea il codice *HTML* scorrendo tutto il *json* e creando un campo di testo per ogni elemento del *json*. É stato tuttavia necessario creare la funzione in modo ricorsivo, in modo da poter scorrere tutti i livelli del *json* (in caso di dati annidati). Ogni elemento della *form* ha come nome il percorso all'interno del *file json*; in questo modo è possibile ricostruire il *json* grazie alla libreria *serializeToJson* [6]. Ad esempio, il *json*:

```
{
    "site": {
        "geometry": {
            "coordinates": {
                "0": "",
                "1": ""
}
```

avrà come elementi all'interno della *form* due caselle di *input* di testo con nome: *"site.geometry.coordinates.0"* e *"site.geometry.coordinates.1"*.
Inoltre, per semplificare l'inserimento o la modifica del luogo geografico di una pianta è stata inserita, all'interno della *form*, una mappa con la libreria *leaflet*

[25] che usa le mappe di *OpenStreetMap* [27] che consente di scegliere da una cartina geografica interattiva il luogo in cui è situata la pianta.
Al *submit* della *form* è associata la funzione *sendData*, che invierà una richiesta di *"write"* con i dati della *form* in allegato, che verranno salvati su *S3* o *DynamoDB*, a seconda del tipo di dato.

Capitolo 5

Conclusioni

In conclusione, come è possibile intuire dalla descrizione del progetto effettuata, l'approccio *serverless* dimostra di possedere i vantaggi che promette, dato che abbatte notevolmente i costi relativi alla gestione o affitto di un *server*.

Tuttavia, necessita ancora di alcune migliorie, soprattutto legate alla fase di sviluppo, dato che la fase di *debug* è davvero difficile, a causa dell'impossibilità di usare strumenti di *debug*. Si potrebbe pensare di usare strumenti come *Docker*, che consentono di ricreare l'ambiente di esecuzione delle funzioni *lambda* e quindi il *test* di esse, ma risultano abbastanza scomodi e non hanno la possibilità di avere collegamenti con tutti i servizi *AWS* esterni alle *lambda*. Per lo sviluppo si è infatti rivelato fondamentale il servizio *Sentry*, che ci ha permesso di scoprire gli errori che si verificavano durante l'esecuzione delle funzioni. Lo sviluppo su una piattaforma come *AWS* richiede inoltre una conoscenza approfondita dei servizi offerti da *Amazon*, oltre che dei linguaggi di programmazione e tecnologie per lo sviluppo delle applicazioni.

In un'ottica di un futuro sviluppo, si può ampliare ulteriormente il progetto potenziando l'applicazione web; in particolare si potrebbero implementare le seguenti migliorie:

- Implementazione di tutte le operazioni *CRUD (Create, Read, Update, Delete)* per tutte le tabelle del *database* in modo da avere una gestione più completa dall'applicazione web
- Implementazione di una autenticazione a più livelli, in modo da avere diversi ruoli, che consentono di aver accesso a funzioni diverse (ad esempio implementare un ruolo che consenta la cancellazione delle piante dal *database*)

Ringraziamenti

Mi è doveroso dedicare questo spazio del mio elaborato alle persone che hanno contribuito, con il loro instancabile supporto, alla realizzazione dello stesso.
Un sentito grazie al mio relatore Adriano Mancini per la sua infinita disponibilità.
Ringrazio il mio compagno di progetto, Roberto Broccoletti, per essermi stato accanto in questo periodo intenso e per gioire, insieme a me, dei traguardi raggiunti.
Ringrazio infinitamente i miei genitori e mio fratello, che mi hanno sempre sostenuto, appoggiando ogni mia decisione, fin dalla scelta del mio percorso di studi.
Infine, ringrazio tutti i miei amici e compagni di corso tra cui Francesco Augello, che mi hanno da sempre incoraggiato.

Bibliografia

[1] Amazon. *Amazon Web Services*. URL: https://aws.amazon.com/it/.

[2] *Docker website*. URL: https://www.docker.com/.

[3] *Documentazione ARN*. URL: https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html.

[4] *Documentazione di boto3*. URL: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

[5] *Documentazione XMLHttpRequest sul sito WHATWG*. URL: https://xhr.spec.whatwg.org/.

[6] *Pagina GitHub di serializeToJSON*. URL: https://github.com/raphaelm22/jquery.serializeToJSON.

[7] *Pagina relativa ad HTML sul sito W3C*. URL: https://www.w3.org/html/.

[8] *Pagina relativa al CSS sul sito W3C*. URL: https://www.w3.org/Style/CSS/.

[9] *Postman website*. URL: https://www.postman.com/.

[10] Broccoletti Roberto. "Progettazione e sviluppo di un sistema *serverless* a supporto di orti botanici in rete - Design and development of a serverless system to support a network of botanical gardens". 2018.

[11] *Sentry website*. URL: https://sentry.io/welcome/.

[12] *Sito GitHub Cognito SDK*. URL: https://github.com/aws-amplify/amplify-js/tree/master/packages/amazon-cognito-identity-js.

[13] *Sito web Amazon API Gateway*. URL: https://aws.amazon.com/it/api-gateway/.

[14] *Sito web Amazon Cognito*. URL: https://aws.amazon.com/it/cognito/.

[15] *Sito web Amazon DynamoDB*. URL: https://aws.amazon.com/it/dynamodb/.

[16] *Sito web Amazon Lambda*. URL: https://aws.amazon.com/it/lambda/.

[17] *Sito web Amazon S3*. URL: https://aws.amazon.com/it/s3/.

[18] *Sito web AWS CLI*. URL: https://aws.amazon.com/it/cli/.

[19] *Sito web di Amazon CloudWatch*. URL: https://aws.amazon.com/it/cloudwatch/.

[20] *Sito web di AWS Console*. URL: https://aws.amazon.com/it/console/.

[21] *Sito web di Bootstrap*. URL: https://getbootstrap.com/.

[22] *Sito web di JavaScript*. URL: https://www.w3.org/Style/CSS/.

[23] *Sito web di Jinja*. URL: https://jinja.palletsprojects.com/en/2.11.x/.

[24] *Sito web di jQuery*. URL: https://jquery.com/.

[25] *Sito web di Leaflet*. URL: https://leafletjs.com/.

[26] *Sito web di Matplotlib*. URL: https://matplotlib.org/.

[27] *Sito web di OpenStreetMap*. URL: https://www.openstreetmap.org/.

[28] *Sito web di Pagination.js*. URL: https://pagination.js.org/.

[29] *Sito web di Python*. URL: https://www.python.org/.

[30] *Sito web di Twython*. URL: https://twython.readthedocs.io/en/latest/.

Elenco delle figure