Università politecnica delle Marche

Facoltà di ingegneria

Corso di Laurea Triennale in ingegneria informatica e dell'automazione

Progettazione e sviluppo di un sistema cloud per l'esecuzione di studi di fattibilità su reti tecnologiche

Design and development of a cloud system for feasibility studies on technological networks

Relatore
Chiar.mo Prof. Emanuele Frontoni

Correlatore
Chiar.mo Prof. Adriano Mancini

Candidato
Giovanni Alessandro Clini
1081850

Anno Accademico 2020-2021

"Sono convinto che l'informatica abbia molto in comune con la fisica. Entrambe si occupano di come funziona il mondo a un livello abbastanza fondamentale. La differenza, naturalmente, è che mentre in fisica devi capire come è fatto il mondo, in informatica sei tu a crearlo. Dentro i confini del computer, sei tu il creatore. Controlli – almeno potenzialmente – tutto ciò che vi succede. Se sei abbastanza bravo, puoi essere un dio. Su piccola scala."

Linus Torvalds

Indice

Capitolo 1

Introduzione

In questa tesi si vuole presentare l'attività svolta nell'ambito della progettazione e sviluppo di un progetto in seno all'azienda *EBWorld S.r.l. a socio unico* con sede in Pesaro.

Lo scopo di tale progetto è quello di progettare e sviluppare un'applicazione *as a service* che sia in grado di eseguire studi di fattibilità sulla maggioranza delle tipologie di reti tecnologiche. Faccio particolare riferimento alla rete della fibra ottica, quella idrica, ma anche la rete elettrica o fognaria. L'idea nasce da un prodotto *custom* costruito su misura per un cliente che ne aveva fatto in precedenza specifica richiesta. Visto l'enorme sviluppo della connessione in fibra e gli imminenti cambiamenti che le nuove tecnologie sono destinate ad introdurre nell'industria ma anche nella vita di tutti i giorni è sembrato opportuno investire su questo tipo di progettazione.

Il team di progetto è composto da alcuni referenti interni all'azienda, dal sottoscritto - che al momento della stesura di questo documento è studente e lavoratore part-time presso EBWorld - da un tesista dell'università di Urbino e un dottorando dell'università di Bologna.

È importante - prima di iniziare con la trattazione vera e propria - sottolineare che ciò che verrà in seguito illustrato non costituisce un prodotto finito e non pretende di essere esaustivo rispetto all'esigenza espressa dall'azienda ma vuole raccontare e formalizzare il lavoro svolto in questi mesi di studio e approfondimento e i risultati ottenuti.

Inizialmente verranno trattate le tecnologie utilizzate nel corso del progetto, approfondendo gli aspetti più importanti di ognuna. Dopodiché si affronterà la fase di progettazione esaminando modello dati e architettura *cloud* e infine verranno proposti alcuni approcci allo sviluppo dell'applicazione web vera e propria.

1.1 L'azienda

EBWorld dal 1983 opera nel settore dell'informatica applicata all'impiego di informazioni geografiche. È uno dei maggiori *player* in Italia a realizzare sistemi informativi geografici (Gis – *Geographical Information System*) e tecnologie correlate ai processi di acquisizione, elaborazione e gestione dei dati georiferiti. I mercati di riferimento dell'azienda sono quelli degli operatori di Telecomunicazione, le Utilities e, più in generale, tutte le realtà, pubbliche e private, che devono gestire e visualizzare i propri asset su

una mappa. Dal 1995 *EBWorld* è distributore e VAR per l'Italia dei prodotti *Smallworld* attualmente prodotti e distribuiti da *GE Energy* ed è partner di *Google, Here, Oracle*.

Capitolo 2

Tecnologie di progetto

In questo capitolo verranno elaborate da un punto di vista teorico le infrastrutture e le tecnologie di cui il team di progetto si è servito durante lo sviluppo, in modo da affrontare con più consapevolezza la successiva parte di analisi e progettazione.
È bene sottolineare che questo progetto punta a sviluppare un'applicazione *as a service*. Dunque verrà eseguita ed ospitata nella sua interezza sul *cloud*. Il *cloud provider* che è stato scelto in continuità con i progetti che l'azienda ha seguito finora è *Amazon AWS*.
Inoltre l'applicazione rientra all'interno di una numerosa famiglia di software che identifichiamo come *GIS: Geographic information system*, il cui sviluppo costituisce il *core business* dell'azienda EBWorld. Il punto centrale di questo progetto sarà infatti proprio una mappa, che prenderà - come vedremo - il 90% dell'interfaccia utente.

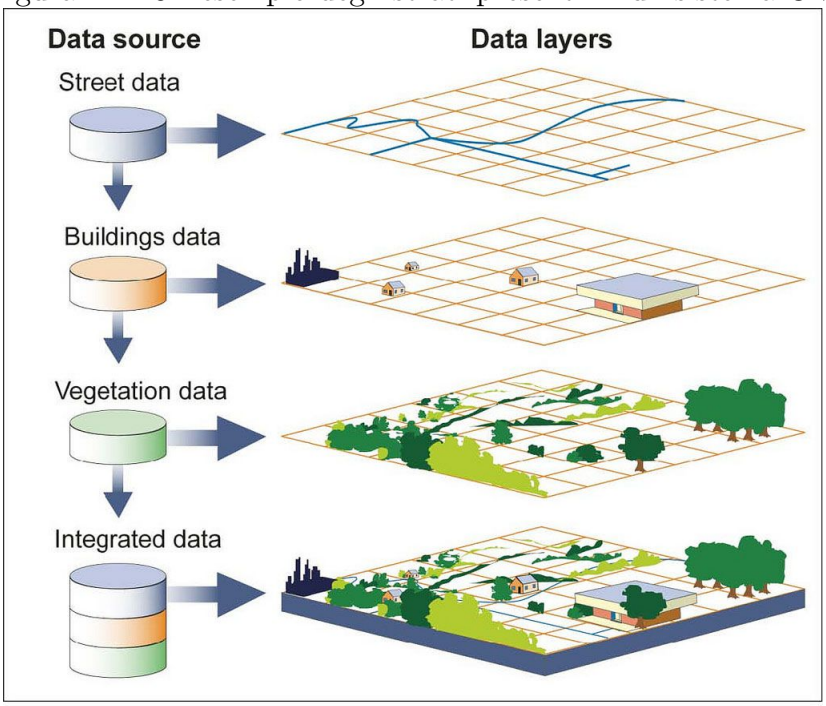
2.1 GIS: Geographic information system

Un sistema informativo geografico (GIS) è un sistema informatico per immagazzinare, verificare e visualizzare dei dati relativi a delle posizioni sulla superficie terrestre. Mettendo in relazione dati all'apparenza non correlati, un software GIS può aiutare gli individui e le aziende a meglio capire come oggetti o fenomeni di interesse si organizzano nello spazio, fornendo uno strumento fondamentale per una buona pianificazione industriale. Molti tipi di dati e informazioni possono essere confrontati usando un GIS. Il sistema può includere informazioni sulle persone, sulla vegetazione, o - come nel nostro caso - informazioni su di una qualsiasi rete tecnologica.

Per rappresentare dei dati in un software che elabori informazioni spaziali occorre definire un modello che si adatti agli input che ci arrivano da ciò che è reale. In un GIS abbiamo in generale tre tipi di informazioni: geometriche, topologiche, e informative. Le prime costituiscono l'ossatura di un sistema di questo tipo, rappresentano punti, linee, aree. Le informazioni topologiche invece descrivono le relazioni tra le varie informazioni geografiche, ed infine quelle informative comprendono tutto quello che non rientra nelle due categorie precedenti. Ogni tipologia di dato andrà importato in un ambiente GIS con l'obiettivo finale di sovrapporre tutte queste informazioni che spesso provengono da fonti eterogenee ed averle tutte in un unico luogo così da poterle visualizzare confrontandole tra loro e nello spazio.
I componenti fondamentali di un software GIS si riassumono in:

- **strumenti per l'input e gestione degli elementi geografici;**

Figura 2.1: Un esempio degli strati presenti in un sistema GIS



- **un geo-database relazionale;**
- **strumenti che supportano interrogazioni, analisi e visualizzazioni;**
- **Interfaccia grafica GUI.**

Si può dire - come si avrà modo di notare successivamente - che l'applicazione in oggetto rientra a pieno titolo nella famiglia dei sistemi GIS, seppur riferendosi ad alcune specifiche funzionalità che si applicano in un dominio preciso, quello delle reti tecnologiche.

2.2 Database e motore di routing

Le fondamenta di un sistema GIS e quindi della nostra applicazione sono composte dai **dati** che rappresentano le informazioni spaziali che vogliamo visualizzare, elaborare e dalle quali vogliamo ottenere delle precise informazioni. Affronto in questa sezione la scelta del tipo di database e del motore di *routing* in maniera contestuale poiché l'uno dipenderà necessariamente dall'altro.

Il punto di partenza del team che ha seguito e sta seguendo questo progetto è stato proprio questo. Il primo problema di progettazione affrontato è stato quello di trovare un tipo di Database che potesse rispondere alle esigenze del team di progetto. In particolare ci interessava un motore di *routing* che fosse facilmente adattabile a numerose ed eterogenee tipologie di dati e che supportasse una funzione di costo dinamica. Con *funzione di costo* intendo quel parametro che determina il peso di un nodo della nostra topologia, ovvero del grafo, composto da archi e nodi, su cui l'*engine* esegue l'algoritmo di ricerca. In primo luogo ci siamo soffermati su *OSRM (Open source routing machine)*, un motore di ricerca open source che lavora sul progetto *OpenStreetMap* e che si caratterizza per

le sue prestazioni elevate in termini di tempo e la sua estesa portabilità. Ciò che in un secondo momento ci ha fatto propendere per altro è dovuto al fatto che OSRM crea il grafo al primo inserimento dei dati, lo memorizza nella *cache* e con esso memorizza anche il parametro di costo che viene legato indissolubilmente al nodo, a patto di non creare la topologia ad ogni ricerca; un fattore che porterebbe a tempi troppo elevati rispetto ai nostri scopi.

2.2.1 PostgreSQL e PostGIS

Per quanto premesso sopra si è deciso di approfondire il noto strumento *open source* di gestione di database relazionali ***PostgreSQL*** il quale dispone di un motore di *routing* chiamato *PgRouting*.

PostgreSQL è un RDBMS che si contraddistingue per la sua elevata estensibilità e aderenza al linguaggio SQL. Dispone di transazioni che godono delle proprietà di Atomicità, Coerenza, Isolamento e Durabilità, note come proprietà **ACID**, di viste in grado di aggiornarsi automaticamente, ma anche di viste materializzate, *triggers*, chiavi esterne e *stored procedures*.
È in grado di supportare la maggior parte dei tipi di dati nativi e fornisce la possibilità all'utente di creare il proprio tipo di dato. Un esempio di questo è l'estensione che permette di elaborare dati con riferimenti spaziali, chiamata ***PostGIS***.

PostGIS come detto poco sopra è un estensione di *PostgreSQL* la quale permette ad oggetti di tipo GIS di essere immagazzinati in un database. Possiede inoltre funzioni specifiche per l'analisi e l'elaborazione di dati georiferiti.
Vediamo più nel dettaglio i paradigmi per la memorizzazione di dati spaziali. Lo *Open Geospatial Consortium* (OGC) ha sviluppato lo standard *Simple Features Access*, definendo il tipo spaziale fondamentale ***geometry***, assieme alle operazioni che manipolano e trasformano valori geometrici per eseguire analisi spaziali. *PostGIS* implementa questo standard attraverso i tipi di dati *geometry* e *geography*. Il tipo *geometry* è un tipo di dato astratto. Gli elementi che appartengono alla classe *geometry* appartengono ad uno dei sottotipi che rappresentano geometrie reali, i quali si dividono in due grandi famiglie: i tipi **atomici** tra cui rientrano *Point, LineString, LinearRing, Polygon* e quelli **collection** che comprendono *MultiPoint, MultiLineString, MultiPolygon, GeometryCollection*. Tutti i valori di questo genere sono associati con un **sistema di riferimento spaziale**, il quale indica il sistema di coordinate nel quale si configura la geometria. Il sistema di riferimento è identificato da un indice numerico chiamato SRID. Il tipo *geometry* è un tipo di dato che possiamo indicare come *opaque*, ovvero tutti gli accessi al dato vengono eseguiti invocando delle funzioni sui valori stessi.
PostGIS dispone anche di un tipo di dato che viene indicato con ***geography*** che provvede ad immagazzinare quei dati spaziali rappresentati su coordinate geografiche. Le coordinate geografiche sono coordinate sferiche espresse in gradi.

2.2.2 Il motore di routing: PgRouting

PgRouting estende le funzionalità di *PostGIS/PostgreSQL* fornendo funzioni per il routing spaziale. Come anticipato in precedenza, questo motore di ricerca non inserisce il parametro di costo nel grafo, ma lo computa ad ogni invocazione di funzione di routing.

La funzione che si occupa di creare la topologia è *pgr_createTopology*. Vediamola nel dettaglio; di seguito la firma:

```
varchar pgr_createTopology(text edge_table, double precision tolerance,
            text the_geom:='the_geom', text id:='id',
            text source:='source',text target:='target',
            text rows_where:='true', boolean clean:=false)
```

Accetta quindi i seguenti parametri:

- **edge_table:** Il nome della tabella che contiene la rete.
- **tolerance:** La tolleranza di *snapping*. Ovvero la distanza massima entro la quale due punti terminali della geometria lineare che compone la rete si considerano collegati tra loro.
- **the_geom:** Il nome della colonna della *edge_table* che contiene la geometria.
- **id:** L'id della *edge_table*.
- **source:** Il nome della colonna della *edge_table* contenente l'id del nodo considerato come punto iniziale della geometria lineare.
- **target:** Il nome della colonna della *edge_table* contenente l'id del nodo considerato come punto finale della geometria lineare.
- **rows_where:** Condizione per SELECT di un sottoinsieme di righe su cui eseguire la funzione.
- **clean:** Se impostato a *true* elimina tutte le topologie precedenti.

La funzione restituisce OK se la topologia viene creata correttamente. FAIL se c'è stato un errore durante il processo. In caso di successo la funzione crea una *vertices_table* contenente tutti i nodi che compongono il grafo risultante e popola i campi *source* e *target* della *edge_table* creando quindi la topologia vera e propria.
Non viene richiesto nessun parametro che possa servire da peso per ogni nodo ma sarà necessario fornirlo solo in un secondo momento, quando verrà eseguita sul nostro database una funzione di *routing* a scelta tra le tante messe a disposizione dall'*engine*. *PgRouting* mette a disposizione una serie di funzioni per eseguire ricerche nel grafo tra cui: *All Pairs Shortest Path, Johnson's Algorithm, All Pairs Shortest Path, Floyd-Warshall Algorithm, Shortest Path A*, Bi-directional Dijkstra Shortest Path, Bi-directional A* Shortest Path, Shortest Path Dijkstra, Driving Distance, K-Shortest Path, Multiple Alternative Paths, K-Dijkstra, One to Many Shortest Path, Traveling Sales Person, Turn Restriction Shortest Path (TRSP) etc.* L'applicazione web farà uso delle varie versioni dell'algoritmo di Dijkstra.

2.2.3 L'algoritmo di Dijkstra

Dato che l'applicazione web in oggetto farà un ampio uso dell'algoritmo di Dijkstra è bene soffermarsi sulla complessità computazionale di tale algoritmo anche in relazione alla complessità del grafo associato e alle varie modalità di implementazione.

L'algoritmo di Dijkstra è un algoritmo per ricercare all'interno di un grafo il cammino più breve tra più nodi. Esistono numerose versioni di tale algoritmo, ai fini della trattazione è sufficiente esaminare la versione originale proposta nel 1956 dall'informatico olandese Edsger Dijkstra che ricerca lo *shortest path* tra due dati nodi. Identifichiamo il nodo di partenza come **nodo iniziale** e con **distanza dal nodo Y** la distanza tra il nodo iniziale e Y. L'algoritmo segue i passaggi seguenti:

1. Tutti i nodi vengono segnati come non-visitati. Viene creato un set di tutti i nodi non-visitati.

2. Viene assegnato ad ogni nodo una distanza provvisoria: per il nodo iniziale viene impostata a zero e a infinito per tutti gli altri nodi. La distanza provvisoria di un nodo v è la lunghezza del percorso più breve trovato fino a quel momento tra il nodo v e il nodo iniziale. Dato che ancora il grafo non è stato esplorato, tutte le distanze vengono poste ad infinito. Il nodo iniziale è impostato come nodo corrente.

3. Per il nodo corrente, vengono considerati tutti i nodi vicini non-visitati e viene calcolata la loro distanza provvisoria attraverso il nodo corrente. Viene poi confrontata la distanza provvisoria appena calcolata con quella correntemente assegnata e viene quindi assegnata la minore. Ad esempio se il nodo corrente A ha una distanza pari a 5 e il vertice che lo collega al nodo vicino B ha distanza pari a 2 allora la distanza di B attraverso A sarà $5 + 2 = 7$. Se il nodo B è stato precedentemente segnato con una distanza maggiore di 7 allora gli viene assegnato il valore 7, altrimenti viene mantenuto il valore corrente.

4. Quando sono stati considerati tutti i nodi vicini al nodo corrente non-visitati, il nodo corrente viene segnato come visitato e rimosso dal set dei nodi non-visitati. Un nodo visitato non viene più controllato dall'algoritmo.

5. Se il nodo di destinazione è stato segnato come visitato oppure se la distanza provvisoria minore tra i nodi nel set dei nodi non-visitati è pari a infinito, l'algoritmo termina.

6. Altrimenti, viene selezionato un nodo non visitato con la distanza provvisoria minore, viene preso come nodo corrente, e l'algoritmo ricomincia dal passo 3.

Complessità computazionale La complessità computazionale dell'algoritmo di Dijkstra può essere espressa in funzione di $|V|$ ed $|E|$, rispettivamente il numero di nodi e degli archi che compongono il grafo su cui viene eseguito l'algoritmo. L'altro parametro che influenza l'algoritmo è la tipologia di struttura dati utilizzata per l'implementazione della coda di priorità da cui l'algoritmo attinge per selezionare il nodo corrente successivo. In generale l'algoritmo di Dijkstra è limitato superiormente da:

$$T_D(G) = \Theta(|V|) + T_B(|V|) + |V| * T_E(|V|) + |E| * T_U(|V|)$$

dove $T_B(|V|)$, $T_E(|V|)$, $T_U(|V|)$ sono le complessità delle operazioni di costruzione, estrazione del minimo e riduzione di un valore di una coda con $|V|$ elementi. Come anticipato, tali valori possono variare a seconda della struttura dati utilizzata. Ad esempio se scegliamo di utilizzare le *Heap binarie* le rispettive complessità valgono:

- $T_B(|V|) : \Theta(|V|)$

- $T_E(|V|) : O(log_2|V|)$
- $T_U(|V|) : O(log_2|V|)$

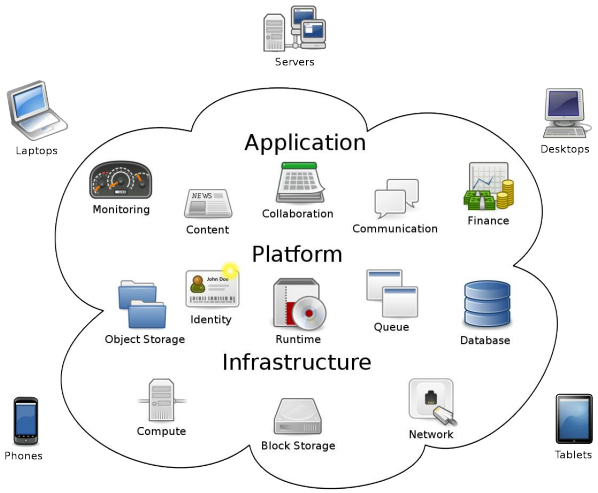
e dunque la complessità computazionale dell'algoritmo è pari a:

$$O((|V| + |E|)log_2|V|)$$

2.3 Cloud computing e Amazon web services

Per l'architettura *cloud*, come anticipato in precedenza, sono stati utilizzati i servizi offerti da *Amazon web services, Inc.* la nota azienda statunitense del gruppo Amazon. AWS fornisce agli sviluppatori una piattaforma per servizi di *cloud computing* e lo sviluppo di applicazioni *as a service*. Questi *web services* forniscono numerose infrastrutture astratte per la creazione di applicazioni sul *cloud*. Tra i tanti servizi e prodotti forniti troviamo database, machine learning, sviluppo mobile, servizi di storage e tanto altro. Prima di

Figura 2.2: Cloud computing



approfondire gli strumenti utilizzati dei tanti offerti da AWS, viene definito cosa si intende quando si parla di ***cloud computing***: con *cloud computing* ci si riferisce solitamente alle applicazioni che vengono erogate come *services* attraverso Internet e ai componenti hardware e sistemi software che vengono messi a disposizione in *data server* che forniscono quel tipo di servizi. Da lungo tempo - e se ne fa uso anche nel corso di questa trattazione - questi sistemi vengono chiamati software *as a service* (SaaS). Alcuni fornitori li chiamano *Infrastructure as a Service* (IaaS) o *Platform as a Service* (PaaS); in realtà, la differenza tra l'infrastruttura di basso livello e la piattaforma di alto livello è sottile ed è difficile stabilire una netta distinzione tra i due. Si preferisce quindi considerarli due concetti simili che necessariamente devono essere considerati nel loro insieme. Il software e l'hardware dietro al *data center* che ospita servizi che ci vengono erogati sono ciò che noi chiamiamo ***cloud***. Quando ciò che si trova nel *cloud* viene fornito come servizio *pay-as-you-go* lo chiamiamo *public cloud*; il servizio che viene venduto invece lo identifichiamo come *utility computing*. Il termine *private cloud* invece lo riferiamo a *data center* che sono interni di

aziende o privati quando sono sufficientemente grandi da beneficiare dei vantaggi dell'elaborazione in *cloud*. Dunque, con *cloud computing* facciamo riferimento all'insieme di *SaaS* e *utility computing*, ma senza includere piccoli o medi *data center*.
Da un punto di vista dell'*hardware provisioning and pricing* tre aspetti sono di centrale importanza e hanno determinato un'esponenziale diffusione delle tecnologie di *cloud computing*:

- La disponibilità di infinite risorse *on demand* e con una allocazione sufficientemente veloce da seguire l'andamento del carico di lavoro.
- L'eliminazione di un quantità iniziale di risorse allocate sufficientemente grande anche per sviluppi futuri. Le aziende ora possono iniziare con poche risorse ed aumentare in base alla necessità.
- La possibilità di effettuare pagamenti a breve termine e a necessità variare l'importo dovuto.

Vengono ora approfondite le tecnologie di AWS utilizzate nel progetto oggetto di questo elaborato.

2.3.1 Amazon RDS

Amazon Relational Database Service or ***Amazon RDS*** è servizio *on demand* che offre la possibilità di ospitare un database relazionale. Viene progettato con lo scopo di semplificare le operazioni di elaborazione e *setup* e di offrire un elevata scalabilità. Come *engine* per il database offre supporto per *MySQL, MariaDB, PostgreSQL, Oracle,*

Figura 2.3: Management models in Amazon EC2 and Amazon RDS

Feature	Amazon EC2 management	Amazon RDS management
Application optimization	Customer	Customer
Scaling	Customer	AWS
High availability	Customer	AWS
Database backups	Customer	AWS
Database software patching	Customer	AWS
Database software install	Customer	AWS
OS patching	Customer	AWS
OS installation	Customer	AWS
Server maintenance	AWS	AWS
Hardware lifecycle	AWS	AWS
Power, network, and cooling	AWS	AWS

Source: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html

Microsoft SQL Server. Come precedentemente anticipato si farà uso del motore per un database *PostgreSQL*.

2.3.2 AWS Lambda

Verrà utilizzato il servizio **AWS Lambda** per creare delle RESTful API che costituiscono - assieme al database - il *back-end* dell'applicazione e che verranno approfondite in seguito. Lambda è un servizio di elaborazione che permette allo sviluppatore di eseguire codice senza preoccuparsi della gestione delle risorse hardware o di gestire i server. Lambda si occupa infatti di tutte le operazioni di amministrazione delle risorse per l'elaborazione, gestione di server e sistema operativo. Il codice viene organizzato in funzioni lambda, le quali vengono eseguite solo se necessario, da una richiesta al giorno, a migliaia di richieste al secondo. Il prezzo viene stabilito in base al tempo di elaborazione che viene effettivamente utilizzato.

2.3.3 Amazon EC2

Amazon Elastic Compute Cloud o ***Amazon EC2*** è la parte centrale della piattaforma di *cloud computing* AWS. Permette agli utenti di noleggiare macchine virtuali nelle quali eseguire i propri applicativi. EC2 supporta implementazioni scalabili di applicazioni disponendo di un *web service* attraverso il quale è possibile avviare una *Amazon Machine Image (AMI)* per configurare una macchina virtuale che Amazon chiama *Instance*. L'utente può creare, avviare e terminare istanze in base alla necessità.

Si farà uso di una istanza Amazon EC2 per ospitare un *web server Apache* contenente il *front-end* al quale l'utente finale potrà collegarsi e usufruire delle funzionalità messe a disposizione dalla nostra applicazione.

2.4 Tecnologie di sviluppo

Per lo sviluppo del *back-end* come del *front-end* è stato utilizzato il linguaggio **Javascript**. Per la parte *server-side* - come già accennato in precedenza - è stato utilizzato il *runtime system open source* ***Node.js*** assieme al *framework* ***serverless***, mentre per la parte *client-side* è stato utilizzata la libreria ***jQuery*** e la libreria ***OpenLayers*** per visualizzare e interagire con la mappa, assieme ai linguaggi **HTML** e **CSS** e il ***Web Server Apache***.

2.4.1 JavaScript

JavaScript è un linguaggio di programmazione che - assieme ai linguaggi HTML e CSS - costituisce le fondamenta del *World Wide Web*. È un linguaggio leggero, interpretato o compilato *just-in-time*. Possiede inoltre funzioni di primo livello, ovvero una funzione è di primo livello se viene gestita come una qualsiasi altra variabile. Se la funzione principale è quella di linguaggio da *scripting* per le pagine web è utilizzato anche in ambienti *non-browser*, come *Node.js, Apache CouchDB a Adobe Acrobat*. Alcune delle principali caratteristiche strutturali del linguaggio, oltre al fatto di essere un linguaggio interpretato, sono:

- Nella sintassi si possono trovare diverse similitudini con altri linguaggi come ad esempio C, C++ e Java.
- Sono definite le strutture dei linguaggi ad alto livello come cicli, strutture di controllo etc.
- Consente l'uso del paradigma *object oriented*, ma si può definire debolmente orientato agli oggetti.
- È un linguaggio debolmente tipizzato.

2.4.2 HTML e CSS

Il ***HyperText Markup Language*** **è definito sullo standard ISO** ***Standard Generalized Markup Language***. Lo standard SGML è un sistema per definire diverse tipologie di documentri strutturati e il linguaggio di *markup* per definire tali documenti. Un istanza HTML è simile ad un file di testo ad eccezione che alcuni caratteri sono interpretati come *markup* (si potrebbe tradurre con *marcatori*). Il *markup* definisce la struttura del documento. Ogni istanza rappresenta una gerarchia di elementi. Ogni elemento ha un nome, degli attributi e del contenuto. Ad esempio:

```
<HTML>
 <TITLE>
  A sample HTML instance
 </TITLE>
 <H1>
  An Example of Structure
 </H1>
 Here's a typical paragraph.
 <P>
 <UL>
```

```
  <LI>
  Item one has an
  <A NAME="anchor">
   anchor
  </A>
  <LI>
  Here's item two.
 </UL>
</HTML>
```

Il linguaggio nasce principalmente con lo scopo di impaginare documenti ipertestuali nel web 1.0 ma oggi è utilizzato principalmente per il disaccoppiamento della struttura logica di una pagina web.

Cascadind Style Sheets o ***CSS*** è un linguaggio usato per descrivere la presentazione delle pagine web, inclusi colori, layout e font. Permette di adattare il modo in cui la pagina viene visualizzata a differenti tipi di dispositivi. CSS è inoltre indipendente dal linguaggio HTML e può essere usato con qualsiasi altro linguaggio di *markup*.

2.4.3 Node.js

Node.js è un ambiente *runtime open-source* e *cross-platfrom* per il linguaggio JavaScript. *Node.js* esegue il *V8 JavaScript engine*, il *kernel* di Google Chrome, al di fuori del browser.
Un'applicazione costruita con *Node.js* viene eseguita in un unico processo, senza instanziare un nuovo *thread* per ogni richiesta; inoltre provvede ad una serie di primitive asincrone di I/O nella sua libreria standard, il che previene un comportamento di tipo bloccante che può caratterizzare il linguaggio JavaScript. Quando viene eseguita una operazione di I/O infatti, al posto di bloccare il processo e di perdere cicli della CPU, *Node.js* continuerà ad eseguire le operazioni successive in attesa che ritorni il *response*. All'interno dell'applicazione in oggetto viene utilizzato per costruire le RESTful API che compongono il *back-end*.

2.4.4 Serverless framework

Per il *deploy* delle funzioni su AWS Lambda ci siamo serviti di un *framework* chiamato *serverless*. Il ***framework serverless*** è un *framework open-soruce* per creare applicazioni *as a service* scritto con *Node.js*. Dispone di una gestione completa del ciclo di vita dell'applicazione e può operare con diversi fornitori di *cloud services* come Microsoft Azure, AWS, Apache OpenWhisk, Google Cloud Platform, Cloudflare Workers.

2.4.5 Web Services e RESTful API

Per quanto riguarda la costruzione della *Web API* che assieme al database costituisce il lato *back-end* dell'applicazione, si è fatto riferimento all'architettura **RESTful API**.
Con *Web API* (o *Web Service*) ci si riferisce ad un server in esecuzione in ascolto su una porta specifica all'interno di una rete che risponde alle richieste che gli arrivano servendo dei documenti web (HTML, JSON, XML, immagini). Una *Web API* conforme all'architettura **RESTful API** è una **REST API**. REST è un acronimo che sta per

***RE**presentational **S**tate **T**ransfer architectural style for distributed hypermedia systems.* Si distingue dalle altre tipi di architetture per cinque punti fondamentali:

- ***Client-Server.*** La prima regola riguarda la separazione delle specifiche del lato client da quelle del lato server con richieste gestite tramite HTTP. Così facendo si migliora la *portability* dell'interfaccia utente attraverso diverse piattaforme e la *scalability* del lato server.

- ***Stateless.*** L'assenza di stato implica che ogni richiesta dal lato client al lato server contenga tutte le informazioni necessarie per portare a termine ciò che viene richiesto.

- ***Cache.*** L'architettura richiede che i dati restituiti vengano esplicitamente dichiarati come *cacheable* o *not cacheable*. Se una *response* è dichiarata come *cacheable* il client può salvare il risultato per usarlo con una successiva richiesta.

- ***Uniform Interface.*** La caratteristica centrale che differenzia l'architettura RESTful dalle altre è quella che riguarda l'interfaccia comune tra i componenti. Applicando i principi dell'ingegneria del software il sistema generale risulta semplificato e inoltre le implementazioni sono disaccoppiate dai servizi che forniscono, il che incoraggia lo sviluppo indipendente dei singoli componenti.

- ***Layered System.*** L'architettura è composta da strati organizzati in maniera gerarchica imponendo delle precise regole ai componenti in modo tale che ogni strato possa interagire solamente con quello adiacente. Restringendo la conoscenza del sistema ad un singolo strato viene messo un limite alla complessità del sistema e viene promossa l'indipendenza dei substrati.

- ***Codice on demand*** **(facoltativo).** Con *codice on demand* si fa riferimento alla capacità di inviare codice eseguibile dal server al client quando richiesto, estendendo la funzionalità del client.

2.4.6 Web server Apache

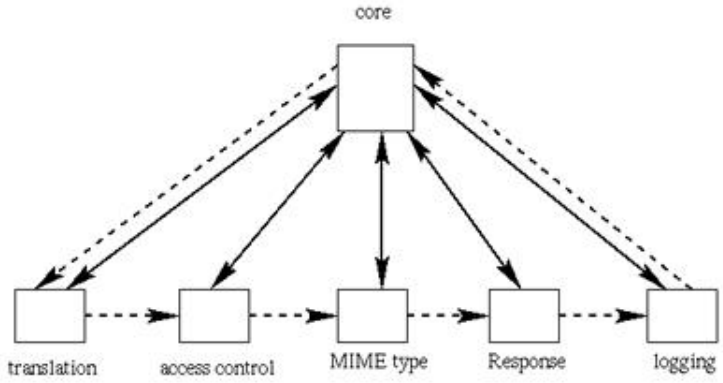
Per l'implementazione del *front-end*, una pagina web che ospita la mappa su cui l'utente potrà eseguire le operazioni che definiremo in seguito, abbiamo utilizzato il *web server apache* su un'istanza EC2 con Amazon Linux. ***Apache HTTP Server*** è un *web server open source* sviluppato dalla *Apache Software Foundation*. È la piattaforma modulare più diffusa in grado di operare su una grande varietà di sistemi operativi. È un software che realizza le funzioni di trasporto delle informazioni, di *internetwork* e di collegamento, ed ha il vantaggio di offrire funzioni di controllo per la sicurezza come quelle effettuate da un *proxy*.

Le linee continue in Fig. 2.4 rappresentano il flusso dei dati reale, le linee tratteggiate il flusso dei dati astratti che formano la pipeline. I moduli che compongono il *web server* Apache sono:

- ***Core.*** Il programma principale composto da un ciclo sequenziale di chiamate ai moduli.

- ***Translation.*** Traduce le richieste del client.

Figura 2.4: Architettura modulare di Apache web server



Source:
https://it.wikipedia.org/wiki/Apache_HTTP_Server

- ***Access control.*** Controlla eventuali richieste malevoli.
- ***MIME Type.*** Identifica il tipo di contenuto e decide quali moduli possono contribuire a servire la richiesta.
- ***Response.*** Invia la risposta al client e attiva eventuali procedure.
- ***Logging.*** Tiene traccia di tutto ciò che avviene.

2.4.7 Python

Abbiamo pensato di utilizzare Python per realizzare il componente che si occuperà del processo di ***ETL (Extract, transform, load)*** per via della sua semplicità di utilizzo e per le numerose librerie presenti nel campo dei *data analytics*.

Python è un linguaggio interpretato di alto livello. La filosofia con cui è stato implementato tende a enfatizzare la leggibilità del codice come punto cardine attraverso l'uso del *significant indentation* assieme all'approccio orientato agli oggetti. In realtà Python è un linguaggio multi-paradigma, il che significa che supporta pienamente diversi tipi di programmazione. Supporta inoltre il *dynamic typing* e per la gestione della memoria utilizza una tecnica che prende alcuni aspetti del *reference counting* e del *garbage collector* a cicli. Inoltre Python è stato progettato per essere estensibile modularmente. Dispone infatti di numerose librerie che possono essere aggiunte al core del linguaggio.

Esempi

Programma che stampa la stringa "Hello, world!".

```
print('Hello, world!')
```

Programma che calcola il fattoriale di un intero positivo.

```
n = int(input('Type a number, and its factorial will be printed: '))

if n < 0:
    raise ValueError('You must enter a non-negative integer')

factorial = 1
for i in range(2, n + 1):
    factorial *= i

print(factorial)
```

Capitolo 3

Analisi e progettazione

In questa sezione affronterò la fase di analisi dei requisiti e progettazione dell'applicazione Web. Lo scopo che si prefigge questa parte del documento è quello di formalizzare ciò che dovrà essere sviluppato e implementato nelle fasi successive.

3.1 Analisi dei requisiti

Un requisito è una descrizione rigorosa e formale di un servizio che un sistema software deve fornire, insieme ai vincoli da rispettare sia in fase di sviluppo che durante la fase di operatività. La gestione di tali requisiti (acquisizione, analisi, negoziazione, specifica e validazione) è il primo passo del processo di sviluppo e una delle fasi più critiche poiché influirà direttamente sul buon esito della progettazione e del successivo sviluppo.

3.1.1 Descrizione del progetto

Si vuole realizzare un'applicazione web erogata come servizio il cui scopo è quello di fornire al cliente una piattaforma in cui eseguire ed archiviare studi di fattibilità per la propria rete tecnologica (rete in fibra ottica, rete idrica, fognaria etc.) e dove poter ottenere report specifici che rispondono a varie necessità pratiche. L'applicazione finale conterrà dei componenti comuni per tutti clienti ed indipendenti dal tipo di rete tecnologica sottoposta a studio di fattibilità e componenti *on premise* e specifici per ogni cliente che si cercherà di limitare in numero per fornire un'applicazione che sia leggera e facilmente scalabile. In questa prima fase di progettazione si intende produrre il *core* dell'applicazione, ovvero tutte quelle funzionalità di base che prescindono dal dominio specifico e che costituiscono le fondamenta per futuri sviluppi. Successivamente, si prevede di aumentare le funzionalità a disposizione dell'utente finale in base alle esigenze specifiche che verranno presentate da chi deciderà di usufruire di questa applicazione. Il nome ipotizzato finora è ***Smart Network Builder (SNB)***.
L'interfaccia utente presenterà una mappa che occuperà quasi l'interezza dello schermo e costituirà la schermata principale dell'applicazione dalla quale sarà possibile accedere a tutte le funzionalità che descriveremo in seguito.

3.1.2 Glossario

TERMINE	DESCRIZIONE	SINONIMI
web based	di software accessibili via internet attraverso browser	(applicazione) web
as a service	di un'applicazione messa a disposizione via internet, solitamente previo pagamento a seconda dell'uso	(applicazione) in cloud
operatore	azienda o singolo che usufruisce del servizio	azienda, cliente, utente
rete tecnologica	insieme delle infrastrutture che compongono un sistema che serve un determinato scopo	infrastruttura
studio di fattibilità	analisi e valutazione sistematica dei costi di un progetto sulla base di una preliminare idea di massima.	//
on premise	di software installati direttamente su macchina locale	//
dominio	ambito di interesse dell'applicazione	ambito d'interesse, argomento
SNB	nome abbreviato della *webapp*	//
admin	utente con privilegi da amministratore	amministratore, super user
routing	operazione di ricerca percorso con determinate caratteristiche su una rete predefinita	ricerca
algoritmo di Dijkstra	algoritmo di ricerca su un grafo composto da archi e nodi	//
cache	parte di memoria in cui viene memorizzata la rete tecnologica su cui si esegue la ricerca	memoria

3.1.3 Requisiti funzionali e non funzionali

Si definiscono requisiti funzionali l'insieme delle caratteristiche che la nostra applicazione dovrà implementare, mentre quelli non funzionali rappresentano l'insieme dei vincoli realizzativi, di tipo tecnico, che il sistema dovrà necessariamente rispettare.

Requisiti Funzionali

Lo scopo di questa prima progettazione è far si che la nostra applicazione garantisca le funzionalità essenziali all'operatore che voglia eseguire in maniera agile degli studi di fattibilità sulla propria rete. Dunque i requisiti funzionali che vogliamo assicurare all'utente sono: **Esegui studio di fattibilità**, **Archiviazione esito**, **Storico studi di fattibili-**

tà, **Aggiornamento esito** e **Cancellazione esito da storico** oltre all'operazione base di **Login** che permetterà di accedere alla piattaforma Web come utente Admin o come utente senza privilegi; un utente Admin potrà eseguire l'operazione di **Caricamento dati di rete in modo asincrono** accedendo all'applicazione Admin; inoltre verrà data la possibilità di attivare la **Visualizzazione della rete** su mappa e la **Visualizzazione della rete pianificata**.

Login L'utente o il gruppo di utenti che hanno acquisito il diritto ad accedere all'applicazione potranno inserire i dati nella schermata di Login ed accedere ai vari servizi a seconda che il loro profilo sia Admin o senza privilegi. Il sistema mostra all'utente i campi in cui inserire nome e password e verifica la correttezza dei dati inseriti confrontandoli con i dati presenti nel database. Se i dati inseriti risultano corretti, l'utente viene autenticato nel sistema e può usare l'applicazione, se i dati di autenticazione non sono corretti il sistema propone all'utente di modificarli. Una volta effettuato l'accesso, l'utente rimane connesso per trenta minuti.

Esegui studio di fattibilità Questa è la funzionalità principale. Permette all'utente di ottenere una stima dei costi necessari per attivare o installare nuova infrastruttura da un punto A ad un punto B. Verrà presentata una mappa all'utente attraverso la quale è possibile scegliere un punto di partenza ed uno di arrivo. Dopodiché, in base allo specifico dominio su cui lavora il singolo operatore, sarà possibile immettere dei parametri che influenzeranno direttamente l'algoritmo di routing. Di base, la ricerca considera un listino prezzi che associa ad ogni tipologia di rete un prezzo specifico al metro. L'algoritmo ipotizzato considera cinque punti di accesso alla rete, assieme ai cinque punti di uscita, calcola i venticinque percorsi migliori in base all'algoritmo di Dijkstra sulla lunghezza delle tratte, associa ad ogni percorso un prezzo in base al listino prezzi fornito dal cliente e ritorna il risultato più conveniente. L'algoritmo inoltre, nella definizione del prezzo del nuovo tracciato, considererà le tratte già pianificate assegnandoli un prezzo nullo in quanto le spese per quei tratti in comune sono già state considerate nei precedenti studi di fattibilità. Viene quindi visualizzato sulla mappa il tragitto migliore e assieme al costo - che viene mostrato all'utente sia nel suo totale, sia diviso in base agli elementi che lo compongono - viene restituita la lunghezza del percorso. Oltre a queste informazioni di base, verranno mostrate delle informazioni specifiche di dominio su richiesta del cliente.

Archiviazione esito Al termine dell'esecuzione dello studio di fattibilità, una volta restituiti i risultati, verrà data la possibilità all'utente di salvare l'azione appena eseguita. Comparirà una schermata in cui poter inserire delle informazioni aggiuntive specifiche di dominio da salvare assieme al costo, alla lunghezza e al tragitto. Se l'utente sceglie di salvare le informazioni appena ottenute, verranno immagazzinate assieme a quelle precedentemente elaborate e sarà possibile utilizzarle per successivi studi.

Storico studi di fattibilità Viene data all'utente la possibilità di visualizzare gli studi di fattibilità che in precedenza ha scelto di salvare. Per ognuno è possibile reperire le informazioni di base come costo, lunghezza e tragitto e le informazioni specifiche di dominio. Sarà inoltre possibile, attraverso operazioni CRUD, accedere al database ed eliminare gli studi che non si ritiene siano più necessari oppure modificare le informazioni di altri.

Aggiornamento esito Una volta elencati gli studi di fattibilità sarà possibile selezionarne uno in particolare e modificare le informazioni specifiche di dominio che l'utente desidera.

Cancellazione esito da storico Una volta elencati gli studi di fattibilità sarà possibile selezionarne uno in particolare ed eliminarlo.

Caricamento dati di rete in modo asincrono Viene data la possibilità all'utente Admin di aggiungere in maniera asincrona nuovi dati a quelli già inseriti. Una volta effettuata l'operazione di login come utente con privilegi, si aprirà l'applicazione Admin che non presenterà una mappa. L'operatore potrà procedere con l'inserimento di un nuovo set di dati. I nuovi dati seguiranno un processo di tipo ***ETL, extract, transform, load***, ovvero verranno estratti dalla sorgente di origine, trasformati per renderli interrogabili dal motore di routing e caricati nella destinazione finale assieme ai dati precedentemente inseriti. Il caricamento di nuovi dati non aggiornerà istantaneamente le funzioni di ricerca e i dati visibili dalla mappa presentata dall'applicazione utente, ma sarà necessario attendere il tempo di creazione della nuova cache per poterne usufruire. Durante la creazione della cache di rete sarà possibile utilizzare il sistema sui dati non aggiornati. Terminato il processo di creazione, i dati saranno automaticamente aggiornati anche sulla mappa e i report e il *routing* opereranno direttamente sui nuovi dati.

Visualizzazione della rete Attraverso una *checkbox* sarà possibile attivare la visualizzazione della rete su cui viene eseguita la ricerca.

Visualizzazione della rete pianificata Attraverso una *checkbox* sarà possibile attivare la visualizzazione della rete risultante dall'esecuzione degli studi di fattibilità precedenti che l'utente ha deciso di memorizzare nel database.

Requisiti non Funzionali

Responsive web design Nello sviluppo dell'applicazione è richiesto l'utilizzo della tecnica del *Responsive web design*, un approccio che mira a rendere il *layout* delle pagine web adattabile all'ambiente nel quale vengono utilizzate: PC, tablet, smartphone etc.

Durata sessione Se l'utente, Admin o senza privilegi, non esegue alcuna azione nel sistema per un tempo continuativo di trenta minuti, la sessione si interrompe e l'utente viene scollegato.

Elencazione degli studi di fattibilità Gli studi di fattibilità memorizzati devono essere elencati in una lista in ordine di inserimento e con tutte le informazioni fornite dall'utente.

Database Le informazioni da salvare nel database riguardano gli account degli utilizzatori del sistema, i dati georiferiti che compongono la rete e la rete pianificata.

Lingua L'applicazione deve essere in lingua italiana, ma si deve comunque prevedere la possibilità di cambiare lingua su specifica richiesta del cliente.

Sicurezza Il sistema deve prevedere delle norme di sicurezza relative alla gestione degli utenti e alla protezione dei dati sensibili che i clienti immettono nel sistema.

Concorrenza L'applicazione deve prevedere l'utilizzo di più account contemporaneamente e uno stesso account deve poter essere utilizzato da più utenti allo stesso momento.

Estensibilità L'applicazione deve poter essere espandibile con ulteriori funzionalità su richiesta di uno specifico cliente.

Resilienza L'applicazione deve portare sempre ad un risultato nonostante i dati sottostanti.

3.1.4 Attori e casi d'uso

Un caso d'uso specifica cosa ci si aspetta da un sistema (*"what?"*) ma nasconde il suo comportamento (*"how?"*). È la specifica di una sequenza di azioni (incluse eventuali sequenze alternative e sequenze di errore) che un sistema può eseguire interagendo con attori esterni. Descrive il comportamento del sistema quando un attore gli invia un particolare stimolo. L'attore rappresenta un soggetto o un'entità che non fa parte del sistema, ma interagisce in qualche modo con esso. Individua un ruolo che l'utente ricopre nell'interagire con il sistema. Gli attori eseguono i casi d'uso.

Attori

Vengono definite due diverse tipologie di utenze. Gli attori che interagiscono direttamente con la nostra applicazione sono:

- **Utente admin.** È colui che si collega all'applicazione Admin sostanzialmente per l'inserimento asincrono di nuovi dati.
- **Utente senza privilegi.** È colui che si collega all'applicazione utente per eseguire e visualizzare studi di fattibilità.

Diagramma dei casi d'uso

I diagrammi dei casi d'uso sono diagrammi dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso. Nelle Fig. 3.1 - 3.2 troviamo il diagramma relativo all'utente senza privilegi e quello relativo all'utente admin.

Di seguito vengono descritte le specifiche dei casi d'uso con gli scenari principali. Uno scenario è una sequenza di passi che descrivono l'interazione tra un sistema e un attore.

Login

- **Nome caso d'uso:** Login
- **id:** UC1
- **Attori:** Utente admin e utente senza privilegi
- **Precondizioni:**

Figura 3.1: Diagramma dei casi d'uso per l'utente senza privilegi

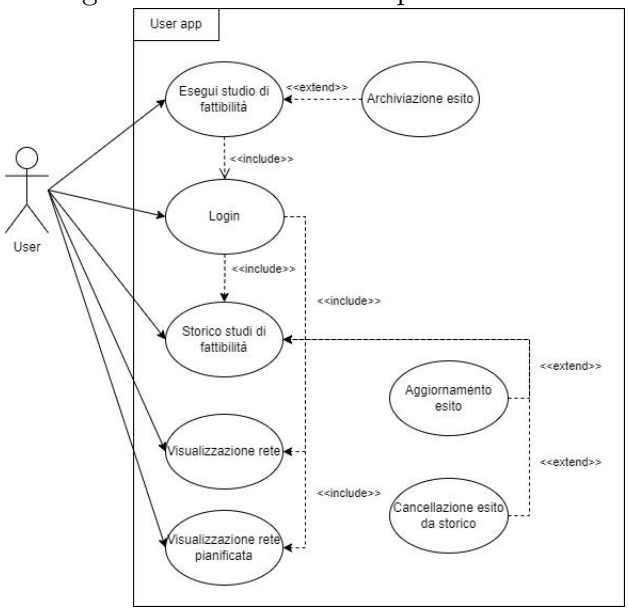
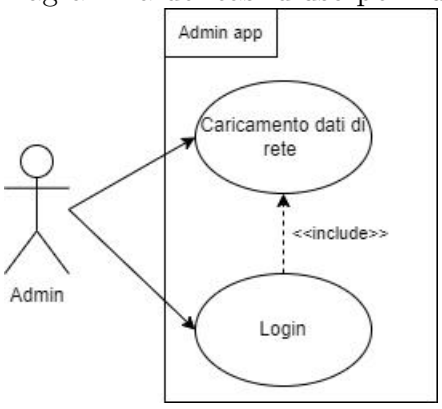


Figura 3.2: Diagramma dei casi d'uso per l'utente admin



1. L'utente ha ottenuto le credenziali di accesso per accedere al sistema

- **Scenario principale:**

1. Il sistema mostra la schermata di login all'applicazione fornendo la possibilità di scegliere tra autenticazione admin e senza privilegi
2. L'utente seleziona il tipo di login
3. L'utente inserisce i dati richiesti (username e password)
4. Il sistema verifica la correttezza dei dati. Se viene trovato un riscontro in base al tipo di utente selezionato l'utente viene indirizzato all'applicazione (admin o per utente senza privilegi), altrimenti viene richiesto dal sistema all'utente di modificare i dati

- **Postcondizioni:**

1. L'utente non deve effettuare l'accesso nuovamente per trenta minuti

Esegui studio di fattibilità

- **Nome caso d'uso:** Esegui studio di fattibilità
- **id:** UC2
- **Attori:** Utente senza privilegi
- **Precondizioni:**

1. L'utente ha effettuato il login nel sistema

- **Scenario principale:**

1. Il sistema mostra la schermata principale contenente una mappa della zona di interesse
2. L'utente seleziona due punti (partenza e arrivo) o uno soltanto sulla mappa in base alla tipologia di ricerca
3. L'utente apre il tool per effettuare gli studi di fattibilità ed inserisce i parametri di dominio specifici
4. L'utente avvia la procedura di analisi
5. Al termine della procedura viene mostrata una schermata con le informazioni di base e specifiche di dominio

- **Postcondizioni:**

1. L'utente può scegliere se memorizzare lo studio di fattibilità appena elaborato

Archiviazione esito

- **Nome caso d'uso:** Archiviazione esito
- **id:** UC3
- **Attori:** Utente senza privilegi
- **Precondizioni:**
  1. L'utente ha effettuato il login nel sistema
  2. L'utente ha eseguito uno studio di fattibilità
- **Scenario principale:**
  1. Il sistema mostra la schermata con le informazioni risultanti dallo studio appena richiesto in cui viene richiesto se si desidera salvare o meno quanto elaborato
  2. L'utente compila i campi disponibili con informazioni specifiche di dominio
  3. L'utente sceglie di salvare lo studio corrente
- **Postcondizioni:**
  1. Lo studio appena salvato potrà essere visualizzato tra quelli che in precedenza l'utente ha scelto di memorizzare

Storico studi di fattibilità

- **Nome caso d'uso:** Storico studi di fattibilità
- **id:** UC4
- **Attori:** Utente senza privilegi
- **Precondizioni:**
  1. L'utente ha effettuato il login nel sistema
  2. L'utente ha eseguito e memorizzato almeno uno studio di fattibilità
- **Scenario principale:**
  1. L'utente accede al servizio che elenca i vari studi di fattibilità
  2. Il sistema mostra all'utente gli studi di fattibilità in memoria con relative informazioni di base e specifiche di dominio
- **Postcondizioni:**
  1. L'utente può selezionare un singolo studio di fattibilità per modificarlo o eliminarlo dalla memoria

Aggiornamento esito

- **Nome caso d'uso:** Aggiornamento esito
- **id:** UC5
- **Attori:** Utente senza privilegi
- **Precondizioni:**
  1. L'utente ha effettuato il login nel sistema
  2. L'utente ha eseguito e memorizzato almeno uno studio di fattibilità
  3. L'utente ha aperto la schermata che visualizza lo storico degli studi di fattibilità
- **Scenario principale:**
  1. L'utente seleziona un singolo studio di fattibilità
  2. Il sistema mostra all'utente le informazioni relative a quel determinato studio di fattibilità
  3. L'utente modifica le informazioni che desidera
  4. L'utente salva le modifiche appena apportate
- **Postcondizioni:**
  1. L'utente potrà visualizzare lo studio con le modifiche insieme agli altri memorizzati

Cancellazione esito da storico

- **Nome caso d'uso:** Cancellazione esito da storico
- **id:** UC6
- **Attori:** Utente senza privilegi
- **Precondizioni:**
  1. L'utente ha effettuato il login nel sistema
  2. L'utente ha eseguito e memorizzato almeno uno studio di fattibilità
  3. L'utente ha aperto la schermata che visualizza lo storico degli studi di fattibilità
- **Scenario principale:**
  1. L'utente seleziona un singolo studio di fattibilità
  2. Il sistema mostra all'utente le informazioni relative a quel determinato studio di fattibilità
  3. L'utente sceglie di eliminare lo studio di fattibilità
  4. Il sistema chiede conferma della volontà dell'utente
  5. Se la risposta è positiva, lo studio viene eliminato
- **Postcondizioni:**
  1. L'utente non potrà più visualizzare lo studio insieme agli altri memorizzati

Visualizzazione della rete

- **Nome caso d'uso:** Visualizzazione della rete
- **id:** UC7
- **Attori:** Utente senza privilegi
- **Precondizioni:**
  1. L'utente ha effettuato il login nel sistema
- **Scenario principale:**
  1. L'utente attraverso una *checkbox* chiede al sistema di mostrare la rete su cui viene eseguita la ricerca
  2. Il sistema mostra all'utente la rete contenuta in memoria sulla mappa che si sta visualizzando
- **Postcondizioni:**
  1. L'utente per nascondere la rete deve necessariamente spuntare nuovamente la *checkbox*

Visualizzazione della rete pianificata

- **Nome caso d'uso:** Visualizzazione della rete pianificata
- **id:** UC8
- **Attori:** Utente senza privilegi
- **Precondizioni:**
  1. L'utente ha effettuato il login nel sistema
  2. L'utente deve aver eseguito e memorizzato almeno uno studio di fattibilità
- **Scenario principale:**
  1. L'utente attraverso una *checkbox* chiede al sistema di mostrare la rete risultante dagli studi di fattibilità eseguiti
  2. Il sistema mostra all'utente la rete pianificata sulla mappa che si sta visualizzando
- **Postcondizioni:**
  1. L'utente per nascondere la rete deve necessariamente spuntare nuovamente la *checkbox*

Caricamento dati di rete in modo asincrono

- **Nome caso d'uso:** Caricamento dati di rete in modo asincrono
- **id:** UC9
- **Attori:** Utente admin
- **Precondizioni:**
  1. L'utente ha eseguito il login come utente admin
- **Scenario principale:**
  1. L'utente admin - attraverso apposita interfaccia - fornisce al sistema il dump dei dati nel formato desiderato
  2. Il sistema invia il pacchetto dati ai componenti che si occupano di ETL (extract, transform, load)
- **Postcondizioni:**
  1. Dopo una sufficiente quantità di tempo l'applicazione user eseguirà la ricerca sui nuovi dati

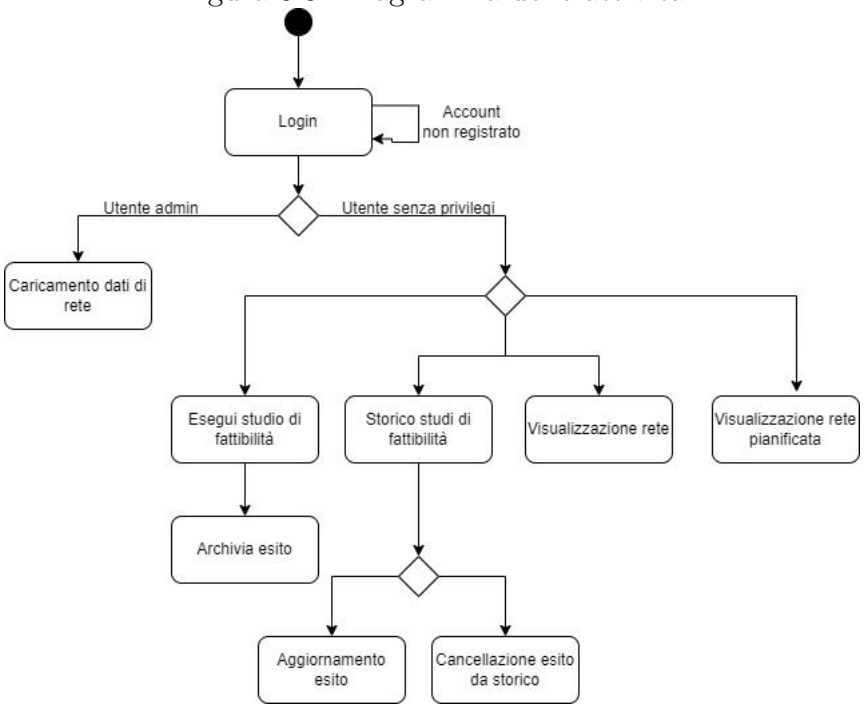
3.1.5 Diagramma delle attività

Il **diagramma delle attività** è una rappresentazione grafica del flusso di lavoro composto da attività e azioni, il quale supporta scelte multiple, iterazione e concorrenza. Nel linguaggio UML *Unified Modeling Language*, tale diagramma è pensato per modellare i processi computazionali e di organizzazione, assieme al flusso dei dati che va ad intersecarsi con le attività correlate. Nel diagramma:

- Le ellissi rappresentano le azioni
- I quadri rappresentano le decisioni
- Le barre rappresentano l'inizio e la fine di attività concorrenti
- Un cerchio nero rappresenta l'inizio del workflow
- Un cerchio bianco cerchiato rappresenta la fine del workflow

Di seguito il diagramma delle attività dell'applicazione web:

Figura 3.3: Diagramma delle attività



3.2 Progettazione

In questa fase si definisce l'architettura software su cui si baserà l'implementazione della *webapp* in oggetto.

3.2.1 Architettura Web

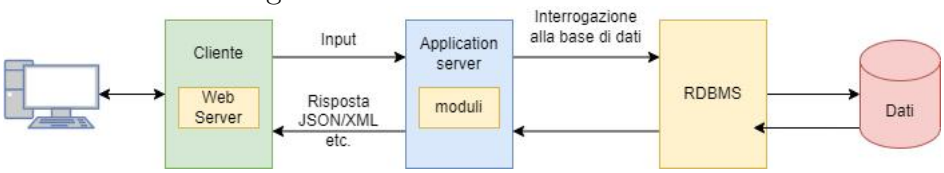
L'architettura scelta si basa su tre livelli logico-funzionali ed è chiamata architettura *three-tier*. Tale architettura prevede la suddivisione dell'applicazione web in tre diversi moduli dedicati rispettivamente alla interfaccia utente, alla logica funzionale e alla gestione dei dati persistenti. Tali moduli interagiscono tra loro seguendo le linee generali del paradigma *client-server*; in questo modo ciascuno dei tre moduli può essere modificato o sostituito conferendo scalabilità e manutenibilità all'applicazione. Nel caso di un'applicazione web i tre strati sono:

- L'interfaccia utente rappresentata da un *web server* e da eventuali contenuti statici (pagine HTML);
- Lo strato logico corrisponde ad una serie di moduli integrati in una *application server* per la gestione di contenuti dinamici;
- I dati sono depositati in maniera persistente in un DBMS.

Come anticipato nella sezione introduttiva sulle tecnologie di progetto le tecnologie utilizzate per ogni livello sono rispettivamente:

- Per la **presentazione** si farà uso di HTML come linguaggio di markup, CSS per la formattazione dei documenti HTML e JavaScript per gestire gli elementi dinamici dell'interfaccia utente; il tutto all'interno di un *web server Apache*.

Figura 3.4: Architettura three-tier



- Per l'**applicazione** vera e propria abbiamo invece si è scelto di utilizzare il runtime system Node.js e il linguaggio JavaScript per creare delle RESTful API che possono essere richiamate dal *web server* per interrogare il database e restituire di conseguenza dei risultati.

- Per la parte **dati** infine come RDBMS si è optato per *PostgreSQL* e il suo motore di *routing PgRouting*.

Un ulteriore componente che non viene compreso nell'architettura è quello che si occupa del processo di ***ETL (Extract, transform, load)***. Questo componente dovrà:

1. Prendere in input i dati che vengono consegnati dal cliente,

2. renderli consoni al modello dati in modo tale che possano essere interrogati correttamente con l'utilizzo delle API che metteremo a disposizione del *web server*,

3. e infine dovrà procedere a caricare i dati opportunamente elaborati nella *cache* di lavoro della nostra applicazione web.

Sarà inoltre l'unico componente specifico per ogni cliente e *on premise*. L'acquisizione di un nuovo operatore dunque implicherà una fase di analisi dei dati e sviluppo di un nuovo componente ETL. Si prevede di implementare tale componente attraverso uno **script Python**.

Architettura cloud

Come premesso, l'applicazione vuole essere erogata *as a service*, ovvero disponibile - a fronte di una sottoscrizione di un servizio - attraverso il web browser. Per rendere possibile ciò è necessario affidarsi ad un fornitore di servizi *cloud* se non si dispone di uno privato sufficientemente grande. Si è scelto di appoggiarsi al *cloud provider* Amazon AWS, anche per coerenza con i progetti seguiti in precedenza dall'azienda. Per ogni livello andremo ora a definire il corrispettivo servizio di cui usufruiremo in AWS.

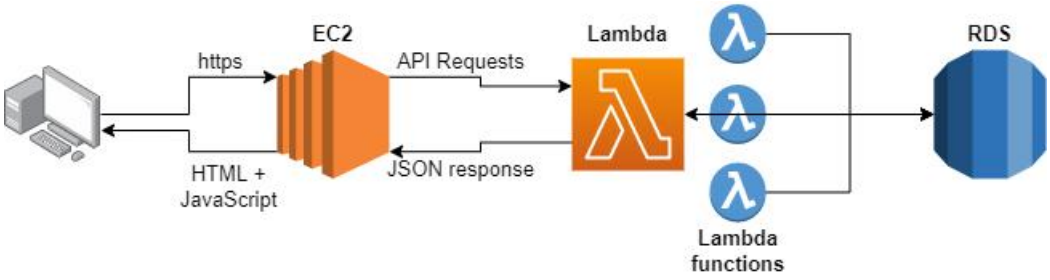
- Per l'implementazione del *web server* abbiamo scelto di utilizzare un'**istanza EC2**, ovvero un'istanza del servizio di AWS che permette di affittare macchine virtuali, sulla quale andremo a configurare il *web server Apache*.

- Per la costruzioni delle RESTful API che svolgono un ruolo di interfaccia tra la base di dati e l'interfaccia utente abbiamo scelto di utilizzare **AWS Lambda**, una piattaforma per l'esecuzione di codice *serverless*.

- Per la gestione dei dati sottostanti all'applicazione web abbiamo scelto di utilizzare **Amazon RDS**, un servizio *on demand* che offre la possibilità di ospitare un database relazionale.

Prevediamo inoltre - qualora dovesse presentarsi la necessità - di aggiungere all'architettura un componente che si occupi di ***load balancing***; con *load balancing* o bilanciamento del carico intendiamo quel processo atto a distribuire il carico di elaborazione di uno specifico servizio, ad esempio la fornitura di un sito web tra più server, aumentando in questo modo scalabilità e affidabilità dell'architettura nel suo complesso. Amazon AWS dispone di un servizio specifico chiamato ***Elastic Load Balancing (ELB)***. Possiamo quindi schematizzare l'architettura cloud su Amazon AWS in questo modo:

Figura 3.5: Architettura cloud su AWS



3.2.2 Modello dati

Lo studio sulla base di dati da implementare ha richiesto la maggioranza del tempo dedicato fino a questo momento. Si ricorda infatti che l'obiettivo è quello di progettare un'applicazione che possa eseguire studi di fattibilità su diverse reti tecnologiche. Il tentativo che si è fatto è stato quindi quello di riunire sotto un unico modello tutte le varie casistiche di rete, senza escludere però che si possa eventualmente procedere con delle eccezioni nel caso in cui un particolare cliente non rientri pienamente nel modello generale ideato.

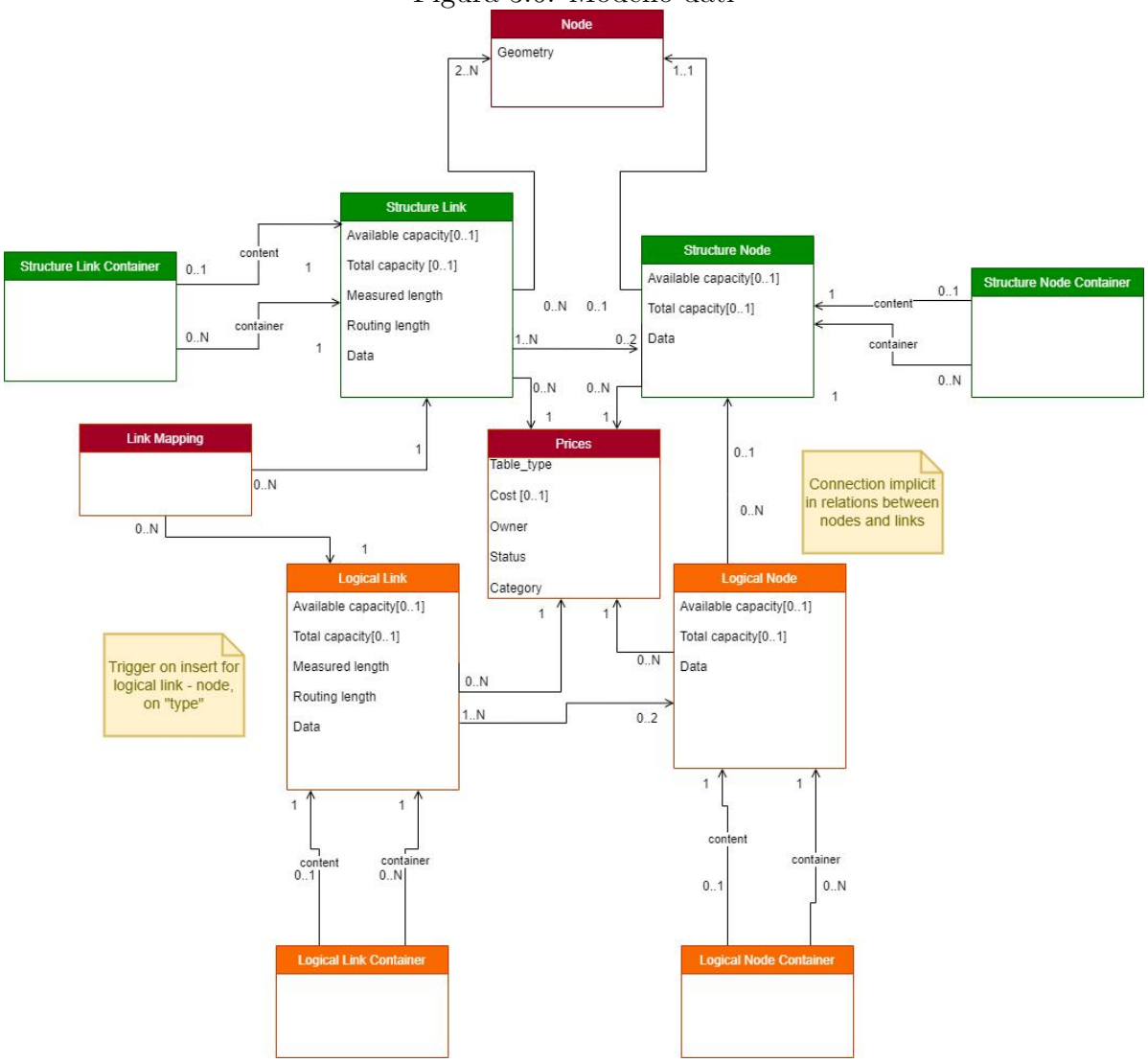
Per ideare il nuovo modello dati siamo partiti dallo studio dell'organizzazione dei dati nel dominio della fibra ottica. Da quei concetti, aggregandone alcuni ed eliminandone altri, siamo giunti al modello della nostra applicazione; il principio di base è stato quello di aggregare dati con alcune caratteristiche in comune in modo da formare poche tabelle generali scremando tutte quelle informazioni che non risultano utili allo studio di fattibilità. Il modello risultante ha la forma raffigurata nella figura 3.6.

Il primo concetto da sottolineare è che abbiamo mantenuto la distinzione tra **Rete civile** e la parte che chiameremo **Rete logica** (nel caso della fibra viene chiamata Rete ottica):

- Con rete civile intendiamo tutte quelle infrastrutture in cui scorre l'infrastruttura proprietaria, ovvero gli scavi effettivi nel terreno o le linee aeree.
- Con rete logica invece intendiamo tutti quegli elementi che appartengono solitamente all'operatore, ovvero ad esempio i cavi, guaine, tubature etc.

La rete civile è rappresentata in verde ed è la componente che contiene la geometria. La rete logica invece è rappresentata in arancione e non contiene nessun campo geometrico ma solo campi che aggiungono informazioni. Vengono ora esaminate nel dettaglio le entità che compongono il modello; le relazioni tra le entità vengono lasciate implicite e verranno realizzate con id univoci, ad eccezione della tabella *Prices*.

Figura 3.6: Modello dati



Node

È l'entità cardine che compone il grafo e quindi la topologia. È l'elemento che contiene il campo geometrico *geometry* che sarà immagazzinato nel database grazie all'estensione di *PostgreSQL* per elementi spaziali *PostGIS*.

Structure Node

È l'entità che rappresenta gli elementi puntuali della topologia, compone la rete civile ed è legata a uno e un nodo (*node*) soltanto. Possiede come attributi:

- *Available capacity[0..1]*: è un parametro opzionale e rappresenta la capacità residua di quella determinata infrastruttura.
- *Total capacity[0..1]*: anche questo è un parametro opzionale e rappresenta la capacità totale di quella determinata infrastruttura.
- *Data*: è un campo di tipo JSON e contiene tutte quelle informazioni che non vengono comprese nei campi precedenti e di cui il cliente necessita.

Structure Link

È l'entità che rappresenta gli elementi lineari della topologia, compone la rete civile ed è legata da due a n nodi. Possiede come attributi:

- *Available capacity[0..1]*: è un parametro opzionale e rappresenta la capacità residua di quella determinata infrastruttura.
- *Total capacity[0..1]*: anche questo è un parametro opzionale e rappresenta la capacità totale di quella determinata infrastruttura.
- *Measured length*: si tratta della lunghezza misurata della tratta, è un parametro fondamentale per la funzione di *routing*.
- *Routing length*: è il parametro su cui effettivamente lavora la funzione di *routing*; se il campo *Measured length* è popolato allora conterrà lo stesso valore, altrimenti sarà popolato con la lunghezza calcolata (meno precisa di quella misurata).
- *Data*: è un campo di tipo JSON e contiene tutte quelle informazioni che non vengono comprese nei campi precedenti e di cui il cliente necessita.

Prices

Abbiamo immaginato una tabella *Prices* che costituisce il listino prezzi a cui l'algoritmo attingerà per assegnare i prezzi alle diverse tipologie di tratta. Possiede come attributi:

- *Table_type*: è la tabella di provenienza dell'elemento a cui viene associato un prezzo. La tipologia infatti può fare riferimento ad un tipo di scavo (minitrincea, trincea etc.) o ad un tipo di cavo.
- *Cost[0..1]*: Il costo al metro associato ad ogni tipologia di tratta. È opzionale perchè ipotizziamo che non tutte le tipologie di elementi puntuali o lineari abbiano un costo associato.
- *Owner*: è un parametro che serve a discriminare tra le tipologie di tratte/cavi propri e quelli di terzi.
- *Status*: è il campo che sta ad indicare la condizione della tratta e quindi ne influenza il prezzo (ad es. progettato, costruito, abbandonato etc.).
- *Category*: è il campo che indica la tipologia dell'elemento a cui viene associato un prezzo.

Logical Node

È l'entità che rappresenta un elemento puntuale della rete logica. Non contiene né campi né informazioni geometriche ma aggiunge informazioni. Possiede come attributi:

- *Available capacity[0..1]*: è un parametro opzionale e rappresenta la capacità residua di quella determinata infrastruttura.
- *Total capacity[0..1]*: anche questo è un parametro opzionale e rappresenta la capacità totale di quella determinata infrastruttura.
- *Data*: è un campo di tipo JSON e contiene tutte quelle informazioni che non vengono comprese nei campi precedenti e di cui il cliente necessita.

Logical Link

È l'entità che rappresenta un elemento lineare della rete logica. Non contiene né campi né informazioni geometriche ma aggiunge informazioni. Possiede come attributi:

- *Available capacity[0..1]*: è un parametro opzionale e rappresenta la capacità residua di quella determinata infrastruttura.
- *Total capacity[0..1]*: anche questo è un parametro opzionale e rappresenta la capacità totale di quella determinata infrastruttura.
- *Measured length*: si tratta della lunghezza misurata della tratta.
- *Routing length*: se il campo *Measured length* è popolato allora conterrà lo stesso valore, altrimenti sarà popolato con la lunghezza calcolata (meno precisa di quella misurata).
- *Data*: è un campo di tipo JSON e contiene tutte quelle informazioni che non vengono comprese nei campi precedenti e di cui il cliente necessita.

La tabella *Link Mapping* è una tabella di supporto che contiene le relazioni tra gli elementi lineari logici e civili. Si è reso necessario introdurla per facilitare l'algoritmo di routing e l'associazione tra i risultati forniti da *PgRouting* e le informazioni contenute nelle altre tabelle. Infine le entità *Link Container*, *Node Container*, *Structure Link Container* e *Structure Node Container* - rispettivamente per la parte logica e per quella civile - le abbiamo inserite immaginando che ci possano essere più strati all'interno di una tipica infrastruttura di rete e dunque vengono lasciate in questa fase vuote. Un esempio è quello della fibra ottica: all'interno dello scavo troviamo prima la guaina, poi il cavo ed infine la fibra. Può essere interessante inserire tutti questi elementi nel modello dati.

Capitolo 4

Un approccio allo sviluppo

In questo capitolo verranno descritti i passaggi che sono stati compiuti fino a questo momento nell'implementazione dell'applicazione. È bene precisare che questa parte di progetto non si è ancora conclusa ma - al momento della stesura di questo elaborato - è ancora in corso.

In questa prima fase sono stati utilizzati dei dati ancora grezzi, non conformi al modello dati elaborato nel capitolo precedente e limitati ad un'area precisa (verrà trattata solo la zona di Milano). Il dominio scelto per la realizzazione di una prima demo è quello della fibra ottica, in linea anche con i progetti precedenti dell'azienda. Lo scopo del team di progetto è stato quello di testare l'architettura e le funzionalità del motore di *routing* così da poter contare su di un primo prototipo su cui effettuare i primi test. Nei paragrafi che seguono verrà quindi affrontata l'implementazione dei tre livelli architetturali e il loro successivo *deploy* in cloud: **Base di dati**, **Applicazione server** e ***Web server***.

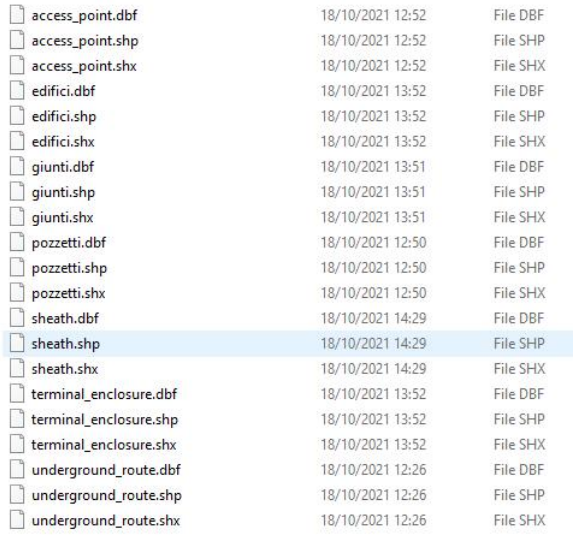
4.1 Base di dati

Il primo passaggio è stato quello di creare un *dump* in locale così da poter eseguire un successivo *deploy* sul cloud. La base di dati risultante dovrà essere pronta per essere interrogata dal motore di routing. I file di test sono stati forniti in formato *shape* (figura 4.1).
Il formato ***shapefile*** rappresenta un tipo di file vettoriale per immagazzinare informazioni geo-spaziali per software GIS. È sviluppato e regolato da *Esri* per l'interoperabilità tra *Esri* ed altri software GIS. Assieme al file .shp vengono forniti obbligatoriamente altri due file: .shx e .dbf. Il file .shp contiene le geometrie vere e proprie, quello con estensione .shx funziona da indice per ricercare velocemente tra le geometrie, mentre il file .dbf contiene gli attributi per ogni file shape. Esistono inoltre file opzionali che possono accompagnare quelli precedentemente citati: ad esempio il file .proj contiene informazioni riguardo le proiezioni con cui sono immagazzinati i dati geometrici.

Per il momento vengono inclusi solamente i dati che provengono dal file *underground_route*, che costituiranno la parte lineare della nostra topologia e dal file *pozzetti*, che costituiranno gli elementi puntuali.
Innanzitutto creiamo un nuovo database che andrà ad ospitare i dati provenienti dai due file:

```
CREATE DATABASE dati_prova
```

Figura 4.1: Dati di test



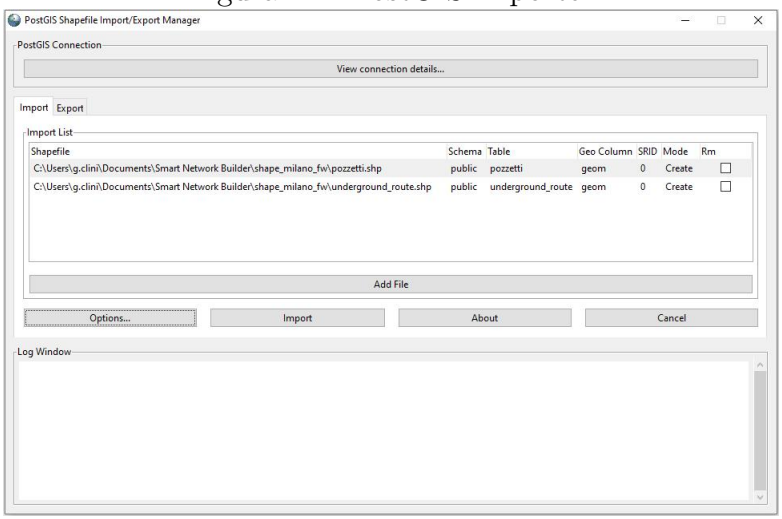

```
WITH
OWNER = postgres
ENCODING = 'UTF8'
CONNECTION LIMIT = -1;
```

e aggiungiamo le estensioni precedentemente installate che permettono di elaborare dati geometrici ed eseguire funzioni di routing:

```
CREATE EXTENSION postgis;
CREATE EXTENSION pgrouting;
```

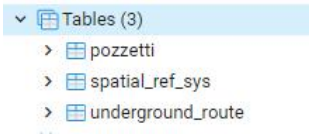
attraverso lo strumento fornito da *PostGIS* per importare file di tipo shape aggiungiamo i dati su cui vogliamo eseguire la ricerca. È importante accedere al pannello delle opzioni per selezionare *"Generate simple geometries instead of MULTI geometries"* poiché il motore di *routing* non lavora sulle geometrie multiple.

Figura 4.2: PostGIS importer



Inseriamo i dati di connessione al database e possiamo premere il tasto *Import*. Una volta terminata l'esecuzione avremo i file con tutti i campi di cui dispongono, compresi quelli geometrici, inseriti correttamente nel database appena generato (Figura 4.3).

Figura 4.3: Tabelle importate



Si noti che è stata creata un'ulteriore tabella *spatial_ref_sys*. Tale tabella contiene oltre 3000 sistemi di riferimento e le informazioni necessarie per eseguire trasformazioni tra una e l'altra proiezione.

Il prossimo passaggio sarebbe quello di creare la topologia così da poter eseguire le funzioni che *PgRouting* mette a disposizione; prima di fare questo dobbiamo però aggiungere due colonne alla tabella che contiene la geometria lineare, ovvero *underground_route*:

```
ALTER TABLE underground_route ADD source bigint;
ALTER TABLE underground_route ADD target bigint;
```

A questo punto siamo pronti per richiamare la funzione che crea il grafo, ovvero la topologia, illustrata nel primo capitolo di questo elaborato:

```
SELECT  pgr_createTopology('underground_route', 0.001, the_geom:='geom');
```

Definiamo esplicitamente il parametro *the_geom* in quanto il nome della colonna non è *the_geom* come previsto dalla funzione ma semplicemente *geom*.
Siamo ora pronti per interrogare la nostra base di dati. Eseguiamo un funzione di prova che ci fornisce *PgRouting*:

```
SELECT * FROM pgr_bdDijkstra( 'SELECT id,
    source,
    target,
    ST_Length(geom::geography) AS cost FROM underground_route',
    5, 23,
    FALSE)
```

Questa query restituisce il percorso più breve secondo l'algoritmo di Dijkstra dal nodo 5 al nodo 23 considerando la lunghezza della geometria lineare in metri. Tale funziona ritorna:

- **seq**: un id univoco per ogni nodo attraversato.
- **path_seq**: un intero che rappresenta la sequenza dei passi percorsi.
- **node**: identificatore del nodo nella tabella dei vertici.
- **edge**: identificatore dell'elemento (in questo caso dell'elemento *underground_route*) per andare da un nodo all'altro.
- **cost**: il costo - in questo caso i metri percorsi - per andare da un nodo ad un altro.
- **agg_costo**: il costo totale di tutti i passaggi precedenti.

Le altre funzioni di routing restituiscono risultati simili.

Figura 4.4: Risultato della funzione di routing

	seq integer	path_seq integer	node bigint	edge bigint	cost double precision	agg_cost double precision
1	1	1	5	6861882	24.29095846293892	0
2	2	2	11	6861881	24.38441366305362	24.29095846293892
3	3	3	12	6861880	23.92974277133457	48.67537212599254
4	4	4	14	6861879	24.28233603904432	72.60511489732711
5	5	5	13	6861878	23.903217806903207	96.88745093637144
6	6	6	15	6861877	24.003488965385742	120.79066874327464
7	7	7	18	6861876	23.82231263326311	144.79415770866038
8	8	8	19	6861875	21.538141048547846	168.6164703419235
9	9	9	20	6861874	11.948464255679779	190.15461139047136
10	10	10	22	5832749	6.575671305053453	202.10307564615113
11	11	11	21	5832750	12.713426154922761	208.67874695120457
12	12	12	16	5832751	23.36477144189257	221.39217310612733
13	13	13	17	5832752	10.619140268296404	244.7569445480199
14	14	14	23	-1	0	255.3760848163163

4.1.1 Deploy su RDS

Una volta effettuati questi passaggi preliminari abbiamo prima configurato un database su Amazon RDS e successivamente abbiamo importato nel *cloud* un *dump* della base di dati appena configurata.
Dopo aver effettuato il login con un account IAM fornito dall'azienda su Amazon AWS, il primo passaggio è stato quello di posizionarsi nella regione che potesse garantire le migliore prestazioni per i nostri scopi: abbiamo scelto la regione **Europa (Irlanda) eu-west-1**. Possiamo quindi accedere al servizio RDS e creare la nostra istanza. Vediamo alcuni parametri rilevanti.
La prima opzione da selezionare è la tipologia di motore e la versione che vogliamo utilizzare (Figura 4.5). Per quanto premesso sopra abbiamo scelto ***PostgreSQL 13.3-R1*** e come modello abbiamo scelto quello di **Sviluppo/Test**.

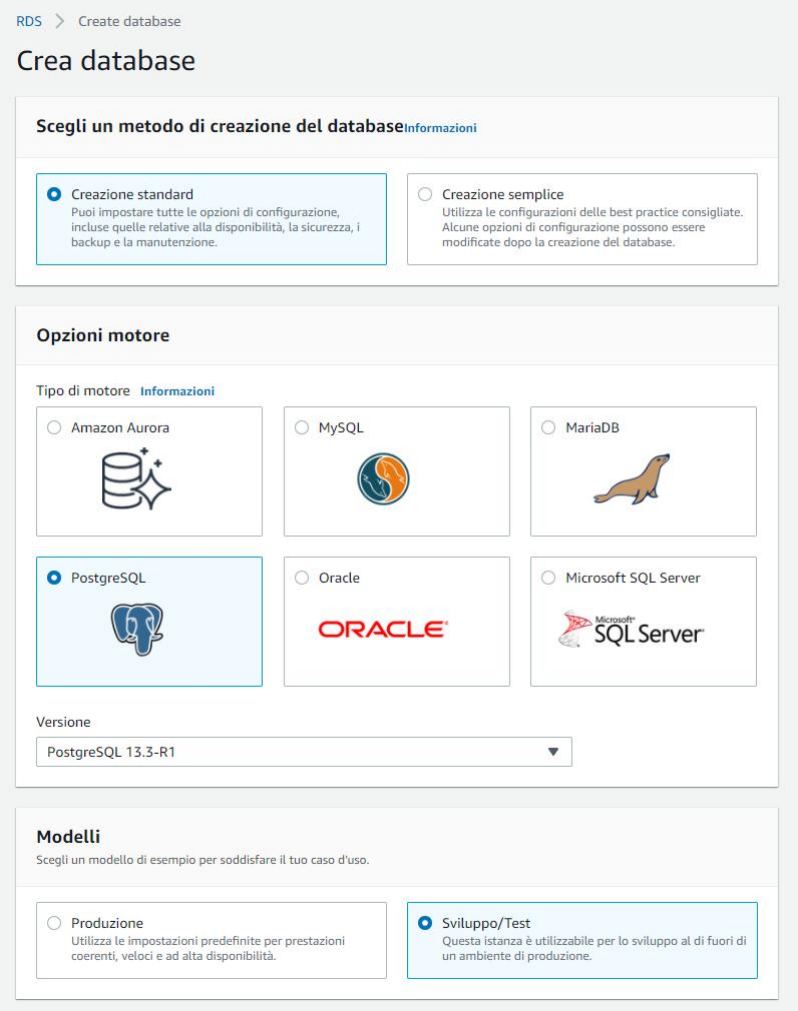
Un'altra impostazione importante che viene richiesta è la classe di database da instanziare. Preferiamo scegliere la classe più piccola ma espandibile per evitare spese al momento non necessarie. Scegliamo quindi la classe **db.t3.micro** (Figura 4.6).

Come *VPC, Virtual Private Cloud*, scegliamo quella predefinita e selezioniamo l'opzione che permette di creare un nuovo gruppo di sicurezza. Per il momento inoltre, impostiamo su "Si" l'opzione "Accesso pubblico": permetterà ad istanze ed utenze esterne alla VPC di default di connettersi al database. Poiché siamo ancora nelle prime fasi dello sviluppo abbiamo preferito questa opzione. Al termine della configurazione Amazon RDS ci propone una stima del prezzo che andremo a sostenere per l'affitto di questo servizio (Figura 4.7).

Per poter importare il *dump* abbiamo interesse a connetterci attraverso *PgAdmin4* al database appena instanziato. Per fare ciò è sufficiente recuperare l'endpoint fornito da AWS, username e password inseriti in fase di configurazione e potremmo avere l'accesso diretto al nostra istanza di database in *cloud* (Figura 4.8).

A questo punto ci sarà sufficiente utilizzare le funzionalità di import/export di *PgAdmin4* per avere una copia in *cloud* identica a quella in locale e perfettamente interrogabile della base di dati.

Figura 4.5: Configurazione istanza RDS (1)



4.2 Applicazione server

L'**applicazione server** sarà composta da una serie di RESTful API che permetteranno all'utente, attraverso l'interfaccia, di interagire con il database ed eseguire le varie funzioni che sono messe a disposizione. Per implementare tali funzionalità useremo i servizi offerti da **AWS Lambda** con l'aiuto del *framework* ***serverless***.
Per il momento vogliamo costruire un *endpoint* che cerchi il percorso più breve tra due punti e assieme al risultato ritorni anche il costo base, ovvero la lunghezza in metri del percorso.

Per prima cosa, dopo aver installato localmente *Node.js* abbiamo inizializzato un progetto *serverless*. Con il gestore dei pacchetti *npm* abbiamo installato la *serverless framework CLI*.

```
npm install -g serverless
```

Abbiamo usato l'opzione *g* per installarlo globalmente. Terminata l'installazione abbiamo avviato la procedura di inizializzazione di un progetto:

```
serverless
```

Figura 4.6: Configurazione istanza RDS (2)

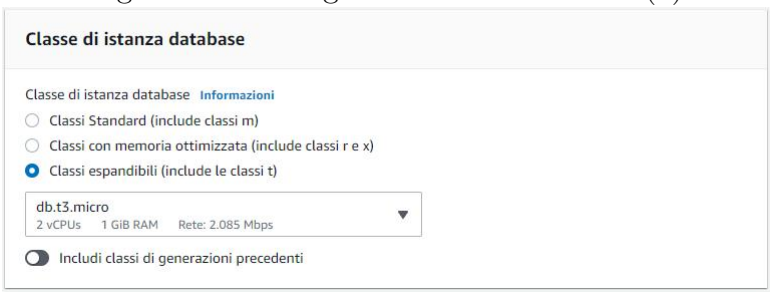
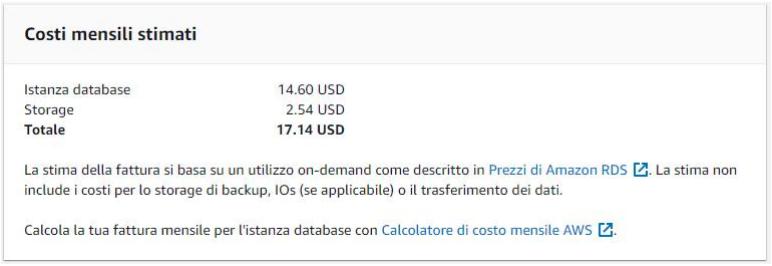


Figura 4.7: Configurazione istanza RDS (3)



Il comando, dopo aver richiesto alcuni parametri come il *template* di progetto e il nome, ha creato una nuova directory di progetto. Dopodiché ci siamo spostati nella nuova cartella e abbiamo lanciato il comando

```
npm init -y
```

per inizializzare un nuovo *package*. La struttura iniziale risultante da questa procedura ha la struttura raffigurata in Fig. 4.9.

Il file **serverless.yml** è dove definiamo le funzioni, ovvero gli *endpoint* e le loro proprietà, mentre il file **handler.js** è dove viene definita la logica sottostante.
Dunque, dopo aver instanziato il progetto, ci siamo autenticati alla console AWS utilizzando le credenziali fornite dall'azienda tramite il comando:

```
serverless config credentials
    --provider aws
    --key <youraccesskey>
    --secret <yoursecretkey>
```

Dopodichè abbiamo configurato il collegamento al database attraverso il file di configurazione db.js e il file db_connect.js che crea la connessione, importata successivamente nel file handler.js. Fatto questo abbiamo implementato la funzione che esegue la ricerca sul database spaziale. In Fig. 4.10 è riportata la parte del file *serverless.yml* che contiene la dichiarazione di tale funzione.

Abbiamo definito la funzione da richiamare nel file *handler.js* *getPathFromAtoB*, il tipo di chiamata, la struttura dei parametri in entrata e abbiamo abilitato *cors*. Per quanto invece riguarda la logica la possiamo ritrovare nel file *handler.js* (Figura 4.11) con lo stesso nome definito in precedenza.
La funzione riceve i parametri dall'url della chiamata e con essi costruisce la query SQL che poi invia al database per essere eseguita, aspetta il risultato in formato JSON in modalità asincrona e lo trasforma in una stringa pronta per essere facilmente elaborata.

Figura 4.8: Configurazione connessione PgAdmin4

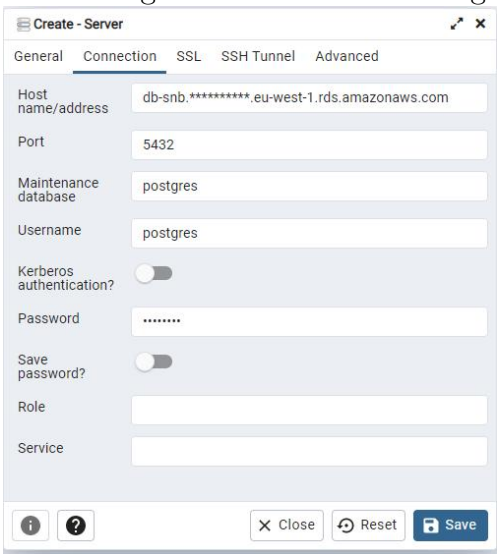
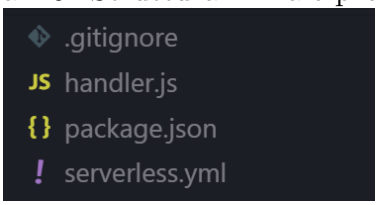


Figura 4.9: Struttura iniziale progetto



Viene aggiunto l'*header 'Access-Contro-Allow-Origin'*, un *CORS (Cross-Origin Resource Sharing) header* che comunica al browser che il contenuto del *response* è accessibile da alcune origini, nel nostro caso il simbolo '*' comunica che è accessibile da tutte. La struttura finale del progetto fino a qui definito è del tipo in figura 4.12.

Per eseguire il *deploy* su AWS Lambda sarà sufficiente eseguire il comando

```
serverless deploy
```

nella cartella di progetto. In questo modo il *framework serverless* si occuperà di impacchettare le funzioni per il *cloud* e renderle disponibili su AWS Lambda. Il *framework* seguirà i seguenti passi:

1. Viene creato un *AWS CloudFormation template* dal file *sevrerless.yml*.

2. Se non è stato creato uno *stack*, viene creato senza risorse eccetto per un S3 *Bucket* che conterrà il file .zip con il codice delle funzioni dichiarate.

3. Se si stanno usando localmente immagini ECR verrà creata una *repository* ECR.

4. Il codice delle funzioni viene impacchettato in file .zip.

5. Se si stanno utilizzando immagini ECR create localmente, vengono create e caricate su ECR. *Serverless* recupera gli *hash* di tutti i file della distribuzione precedente (se presenti) e li confronta con gli *hash* dei file locali.

6. Serverless termina il processo di *deploy* se tutti gli *hash* sono gli stessi.

Figura 4.10: Dichiarazione della funzione su serverless.yml

```
pathFromAtoB:
  handler: handler.getPathFromAtoB
  events:
    - http:
        path: pathfromatob
        method: get
        request:
          template:
            application/json: >
              {
                "x1" : "$input.params('x1')",
                "y1": "$input.params('y1')",
                "x2": "$input.params('x2')",
                "y2": "$input.params('y2')"
              }
        cors: true
```

7. I file .zip precedentemente creati vengono caricati nel S3 *Bucket*.
8. Qualsiasi ruolo IAM, funzione, evento o risorsa sono aggiunti al template AWS *CloudFormation*.
9. Lo *stack CloudFormation* viene caricato con il nuovo *template*.
10. Ogni nuovo *deploy* pubblica una nuova versione delle funzioni dichiarate.

Al termine del *deploy* il *framework Serverless* ci comunicherà se il processo è andato a buon fine e in caso affermativo l'indirizzo a cui effettuare le chiamate. Vediamo un esempio di una chiamata alla funzione di cui sopra; utilizzeremo il software *Postman*, lo stesso software (assieme alla sua versione CL *Newman*) che utilizzeremo in seguito per definire alcuni test.

Prendiamo in maniera casuale due paia di coordinate all'interno dell'area di test ed eseguiamo la chiamata al nostro endpoint appena creato (Figura 4.13). Notiamo che viene restituito un vettore contenente un elemento *cost* che ci indica che il percorso più breve - percorrendo la rete contenuta nel database tra le coordinate fornite - misura circa 9,570 chilometri e il secondo parametro *st_asgeojson* contiene in formato stringa il GeoJSON che rappresenta il percorso su mappa.

4.2.1 API Test

Per implementare dei test per verificare il comportamento degli *endpoint* che andremo via via creando, costruiamo una *collection* su Postman che andremo poi ad eseguire con la versione a riga di comando *Newman*. La *collection* è definita come nella figura 4.14.

Vengono definite 5 chiamate: la prima all'*endpoint* di test mentre le successive 4 all'*endpoint* descritto in precedenza con l'eccezione che all'ultima chiamata vengono fornite coordinate fuori dall'area di test e dunque il risultato dovrà essere nullo. Dopodiché, definisco alcuni test per ogni chiamata, come mostrato nella figura 4.15.

Figura 4.11: Dichiarazione della funzione su serverless.yml

```
module.exports.getPathFromAtoB = (event, context, callback) => {
    context.callbackWaitsForEmptyEventLoop = false;
    var start_x = event.multiValueQueryStringParameters.x1
    var start_y = event.multiValueQueryStringParameters.y1
    var end_x = event.multiValueQueryStringParameters.x2
    var end_y = event.multiValueQueryStringParameters.y2
    const sql = "WITH \
                dijkstra AS ( \
                    SELECT * FROM pgr_dijkstra('SELECT id, source, target, ST_Length(geom::geography) AS cost \
                    FROM underground_route', \
                                (SELECT source FROM underground_route \
                                 ORDER BY ST_Distance( ST_SetSRID(ST_StartPoint(geom),4326), \
                                     ST_SetSRID(ST_MakePoint(" + start_x + ", " + start_y + "),4326) ) ASC LIMIT 1), \
                                (SELECT source FROM underground_route \
                                 ORDER BY ST_Distance( ST_SetSRID(ST_StartPoint(geom),4326), \
                                     ST_SetSRID(ST_MakePoint(" + end_x + "," + end_y + "),4326) ) ASC LIMIT 1), \
                                FALSE)), \
                get_geom AS ( \
                    SELECT dijkstra.*, underground_route.id, \
                        CASE \
                            WHEN dijkstra.node = underground_route.source THEN geom \
                            ELSE ST_Reverse(geom) \
                        END AS route_geom \
                    FROM dijkstra JOIN underground_route ON (edge = id) \
                    ORDER BY seq) \
                SELECT (Sum(cost) + 1) AS cost, \
                ST_AsGeoJSON(ST_MakeLine(route_geom)) \
                FROM get_geom;";
    console.log(sql)
    db.query(sql)
        .then(res => {
            callback(null, {
                statusCode: 200,
                headers: {
                    'Content-Type': 'application/json',
                    'Access-Control-Allow-Origin': '*' },
                body: JSON.stringify(res)
            })
        })
        .catch(e => {
            console.log(e);
            callback(null, {
                statusCode: e.statusCode || 500,
                headers: {
                    'Content-Type': 'application/json',
                    'Access-Control-Allow-Origin': '*' },
                body: 'Error: ' + e
            })
        })
```

Una volta definita la *collection* ed esportata in formato JSON posso eseguirla con *Newman*. Per prima cosa installo il software con il *package manager npm*:

```
npm install -g newman
```

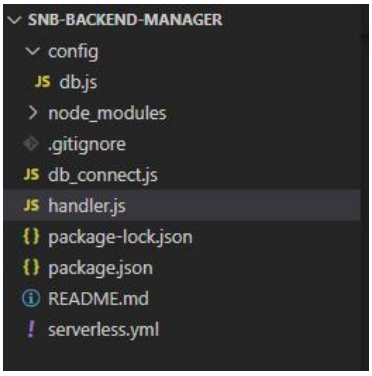
Dopodichè mi sposto nella directory dove ho esportato la *collection* ed eseguo il comando:

```
newman run \
api_tests.postman_collection.json
```

L'*output* è raffigurato in Figura 4.16.

Possiamo dunque concludere che l'*endpoint* appena definito stia correttamente funzionando. Inoltre eseguire con una certa frequenza dei test è uno strumento fondamentale per evitare recessioni nella scrittura di codice e per creare della "documentazione live", ovvero per rimanere sempre aggiornati sulle funzionalità che vengono aggiunte man mano che si procede con lo sviluppo. La *collection*, una volta esportata, potrebbe essere anche utilizzata all'interno del *source control* come base per la *pipeline* per eseguire alcuni test.

Figura 4.12: Struttura del back-end



4.3 Web server

Per implementare il *Web server* che conterrà la pagina web a cui accedere per eseguire le funzionalità messe a disposizione dalla nostra applicazione abbiamo bisogno di attivare un'istanza EC2 su Amazon AWS. Disponiamo già di un VPC e di una *subnet* - una sotto-rete all'interno della VPC - con accesso a internet poiché entrambi forniti dall'azienda. Dunque, per prima cosa abbiamo creato un gruppo di sicurezza su AWS che disponesse delle regole necessarie per il *web service* che vogliamo implementare; su AWS creiamo un nuovo gruppo di sicurezza all'interno della VPC di default e come regole in entrata inseriamo: **HTTP — TCP — 80 — Anywhere** per connettersi alla pagina web tramite il protocollo http e la regola **SSH — TCP — 22 — Anywhere** per collegarci tramite il protocollo ssh. Siamo ora pronti per lanciare l'istanza EC2. Seguiamo la procedura guidata: anche in questo caso, abbiamo scelto di selezionare le caratteristiche hardware più basse per evitare prezzi troppo alti rispetto alle nostre necessità, almeno in questa fase. Come sistema operativo abbiamo scelto **Amazon Linux** e come tipo di istanza la **t2.micro** con una CPU virtuale e un GiB di memoria RAM. Abbiamo quindi scelto la VPC di default, la subnet configurata dall'azienda e il gruppo di sicurezza appena creato. I restanti parametri sono rimasti quelli di default. Al termine della configurazione abbiamo creato la chiave SSH per accedere alla macchina virtuale e scaricata in locale. Una volta in esecuzione la macchina virtuale appena configurata ci siamo collegati al suo terminale attraverso il software *Putty* e la chiave in precedenza salvata localmente. Per installare il *web server Apache* abbiamo eseguito i seguenti comandi:

1. Per prima cosa abbiamo aggiornato la macchina virtuale attraverso il comando

   ```
   sudo yum update -y
   ```

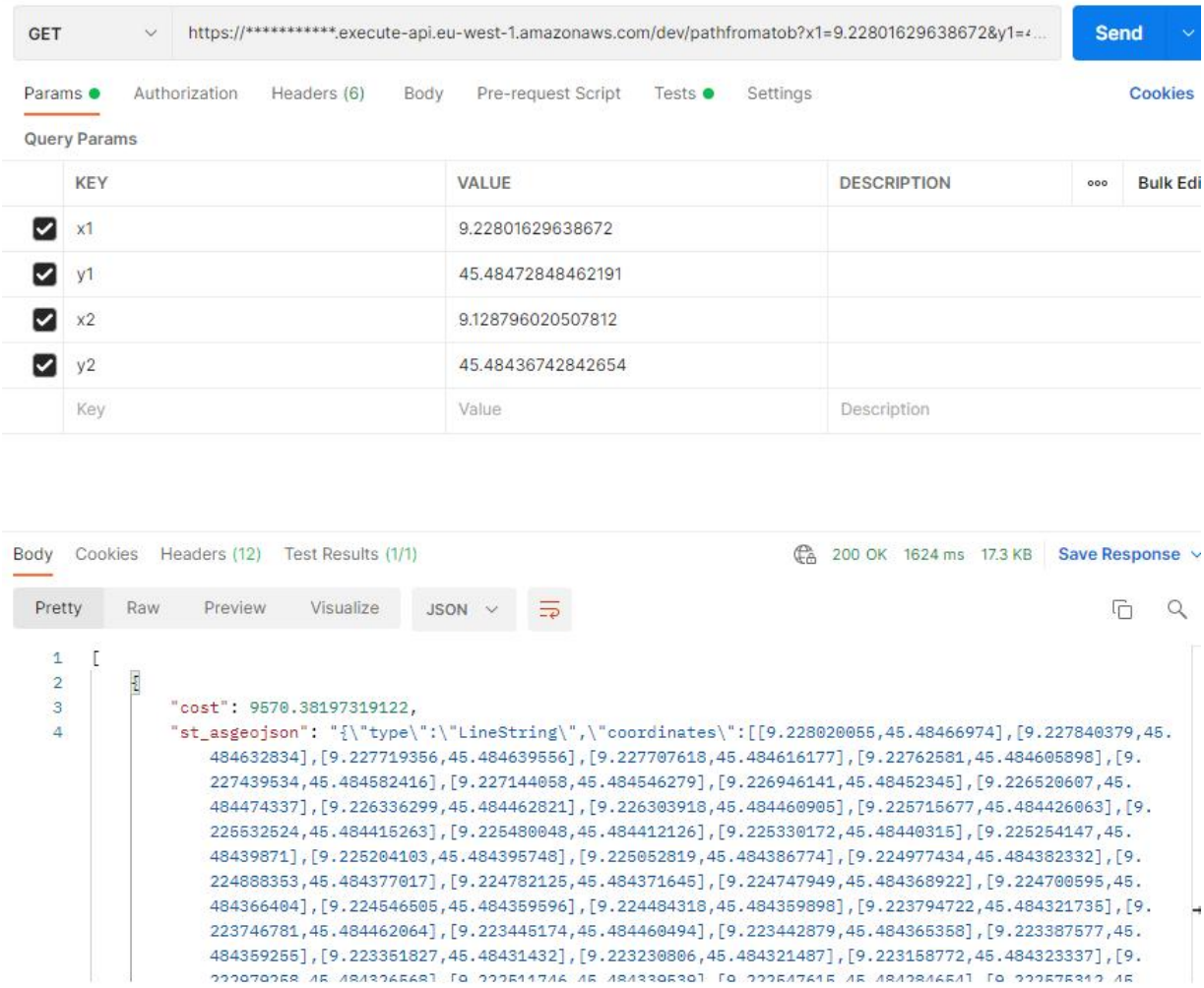
2. Dopodichè abbiamo installato il *web server Apache*

   ```
   sudo yum install -y httpd
   ```

3. e una volta installato, l'abbiamo avviato

   ```
   sudo systemctl start httpd
   ```

Figura 4.13: Chiamata di prova all'endpoint appena definito



4. Una volta avviato il *web server* lo abbiamo configurato affinche si avviasse ad ogni riavvio del sistema

```
sudo systemctl enable httpd
```

Una volta terminata questa procedura siamo passati alla creazione della pagina web. Per effettuare le modifiche ci siamo collegati al *web server* con il protocollo SSH attraverso il software *Visual Studio Code*. La cartella di default *root* del web server si trova in */var/www* e affinchè venga visualizzato ciò che desideriamo, la pagina HTML (*index.html*) deve essere salvata in questo percorso. Possiamo vedere la struttura in figura 4.17.

La cartella *images* contiene per ora solamente l'icona del puntatore su mappa nel momento in cui l'utente seleziona un punto. Il file *index.html* invece contiene la pagina html e dunque il contenuto statico, mentre il file *mapscript.js* contiene la parte dinamica della pagina web e viene incluso nella pagina html. Vediamo da vicino alcune parti rilevanti del codice contenuto in entrambi i file. Nella figura 4.18 viene definito il *body* della pagina html: viene definito un bottone, associato alla funzione *findPath()* (Figura 4.19) contenuta nel file .js che esegue la chiamata al *back-end* e restituisce il risultato che viene poi elaborato dalla funzione *addLayerFromQuery()* (Figura 4.20). Inoltre viene definita la mappa *OpenLayers* (Figura 4.21) e incluso lo script *mapscript.js*.

Figura 4.14: Collection di test

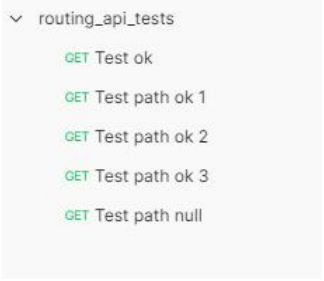


Figura 4.15: Test su una chiamata

```
pm.test("Access-Control-Allow-Origin is *", () => {
    // Controllo che l'header cors sia impostato su *
  pm.expect(pm.response.headers.get('Access-Control-Allow-Origin')).to.eql('*');
});

pm.test("ResponseJson[0].cost is a number", () => {
  const responseJson = pm.response.json();
  // Controllo che cost sia un numero
  pm.expect(responseJson[0].cost).to.be.a('number')
});

pm.test("ResponseJson[0].cost is as desired", () => {
  const responseJson = pm.response.json();
  // Controllo che il valore di cost sia quello atteso
  pm.expect(responseJson[0].cost).to.eql(9570.38197319122)
});

pm.test("ResponseJson[0].st_asgeojson is a string", () => {
  const responseJson = pm.response.json();
  // Controllo che st_asgeojson sia una stringa
  pm.expect(responseJson[0].st_asgeojson).to.be.a('string')
});

pm.test("ResponseJson[0].st_asgeojson is as desired", () => {
  const responseJson = pm.response.json();
  // Controllo che il valore di st_asgeojson sia quello atteso
  pm.expect(responseJson[0].st_asgeojson).to.eql("{\"type\":\"LineString\",\"coordinates\":[[9.228020055,45.48466974],..."


```

Vista dall'interfaccia utente l'esecuzione della funzione *findPath()* risulta come in figura 4.22.

Dalla console possiamo osservare che oltre al risultato in formato GeoJSON viene restituito anche il costo (in metri) per andare dal punto A al punto B (Figura 4.23) pari, in questo caso, a circa 1,510 kilometri.

Figura 4.16: Test eseguiti con Newman

```
C:\Users\g.clini\Documents\Smart Network Builder>newman run api_tests.postman_collection.json

newman

routing_api_tests

→ Test ok
  GET https://u35zesl19k.execute-api.eu-west-1.amazonaws.com/dev/test [200 OK, 1.23kB, 2.6s]
  √  Access-Control-Allow-Origin is *
  √  ResponseJson[0].st_asgeojson is a string
  √  ResponseJson[0].st_asgeojson is as desired

→ Test path ok 1
  GET https://u35zesl19k.execute-api.eu-west-1.amazonaws.com/dev/pathfromatob?x1=9.2280162963
8672&y1=45.48472848462191&x2=9.128796020507812&y2=45.48436742842654 [200 OK, 17.71kB, 2.3s]
  √  Access-Control-Allow-Origin is *
  √  ResponseJson[0].cost is a number
  √  ResponseJson[0].cost is as desired
  √  ResponseJson[0].st_asgeojson is a string
  √  ResponseJson[0].st_asgeojson is as desired

→ Test path ok 2
  GET https://u35zesl19k.execute-api.eu-west-1.amazonaws.com/dev/pathfromatob?x1=9.3186535034
17956&y1=45.537177378684646&x2=9.033668948644372&y2=45.427194280388505 [200 OK, 41.78kB, 1202
ms]
  √  Access-Control-Allow-Origin is *
  √  ResponseJson[0].cost is a number
  √  ResponseJson[0].cost is as desired
  √  ResponseJson[0].st_asgeojson is a string
  √  ResponseJson[0].st_asgeojson is as desired

→ Test path ok 3
  GET https://u35zesl19k.execute-api.eu-west-1.amazonaws.com/dev/pathfromatob?x1=9.3141227486
58233&y1=45.37550031410382&x2=9.151870971503735&y2=45.32266158971109 [200 OK, 17.4kB, 1150ms]

  √  Access-Control-Allow-Origin is *
  √  ResponseJson[0].cost is a number
  √  ResponseJson[0].cost is as desired
  √  ResponseJson[0].st_asgeojson is a string
  √  ResponseJson[0].st_asgeojson is as desired

→ Test path null
  GET https://u35zesl19k.execute-api.eu-west-1.amazonaws.com/dev/pathfromatob?x1=9.3589944477
03111&y1=45.133325508416306&x2=9.420892192084343&y2=45.04240855158062 [200 OK, 555B, 1082ms]
  √  Access-Control-Allow-Origin is *
  √  EesponseJson[0].cost is null
  √  ResponseJson[0].st_asgeojson is null

┌─────────────────────────┬────────────────────┬────────────────────┐
│                         │           executed │             failed │
├─────────────────────────┼────────────────────┼────────────────────┤
│              iterations │                  1 │                  0 │
├─────────────────────────┼────────────────────┼────────────────────┤
│                requests │                  5 │                  0 │
├─────────────────────────┼────────────────────┼────────────────────┤
│            test-scripts │                  5 │                  0 │
├─────────────────────────┼────────────────────┼────────────────────┤
│      prerequest-scripts │                  0 │                  0 │
├─────────────────────────┼────────────────────┼────────────────────┤
│              assertions │                 21 │                  0 │
├─────────────────────────┴────────────────────┴────────────────────┤
│ total run duration: 8.9s                                          │
├───────────────────────────────────────────────────────────────────┤
│ total data received: 76.07kB (approx)                             │
├───────────────────────────────────────────────────────────────────┤
│ average response time: 1684ms [min: 1082ms, max: 2.6s, s.d.: 666ms] │
└───────────────────────────────────────────────────────────────────┘
```

Figura 4.17: Root directory webserver

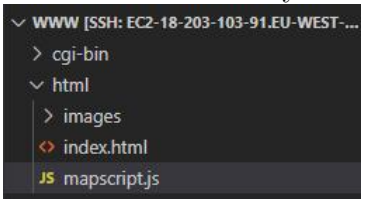


Figura 4.18: Definizione body html

```
<body>
  <h2>Smart Network Builder</h2>
  <button class="button button1" style="float: right;" onclick="findPath();" >Percorso</button>
  <div id="map" class="map"></div>
  <script src="mapscript.js"></script>
</body>
```

Figura 4.19: Definizione funzioe findPath()

```
function findPath(){

    x1 = start_and_end[0][0]
    y1 = start_and_end[0][1]
    x2 = start_and_end[1][0]
    y2 = start_and_end[1][1]

    var query_settings = {
        url: "https:/[redacted].execute-api.eu-west-1.amazonaws.com/dev/pathfromatob?x1=" + x1 + "&y1=" + y1 + "&x2=" + x2 + "&y2=" + y2,
        type: "GET",
        crossDomain: true
      };

    addLayerFromQuery(query_settings);
}
```

Figura 4.20: Definizione funzione addLayerFromQuery()

```
75  function addLayerFromQuery(query_settings){
76
77      $.ajax(query_settings).done(function (response) {
78          console.log(response)
79          var json_trail = JSON.parse(response[0].st_asgeojson)
80          console.log(json_trail)
81          var vectorSource = new ol.source.Vector({
82              features: new ol.format.GeoJSON().readFeatures(json_trail, {
83                  featureProjection: 'EPSG:3857'
84              })
85          });
86          var vector = new ol.layer.Vector({
87              source: vectorSource,
88              style: new ol.style.Style({
89                  stroke: new ol.style.Stroke({
90                      color: 'blue',
91                      width: 5,
92                  }),
93                  fill: new ol.style.Fill({
94                      color: 'rgba(0, 0, 255, 0.1)',
95                  }),
96              }),
97          });
98          //console.log("mappa", map);
99          console.log("source", vector);
100         map.addLayer(vector)
101         map.render()
102     });
103
104 }
```

Figura 4.21: Definizione della mappa

```
1   var map = new ol.Map({
2       target: 'map',
3       layers: [
4       new ol.layer.Tile({
5           source: new ol.source.OSM()
6           })
7       ],
8       view: new ol.View({
9           center: ol.proj.fromLonLat([9.2160, 45.4620]),
10          zoom: 13,
11          minZoom: 13,
12          maxZoom: 18
13      })
14  });
```

Figura 4.22: Parte dell'interfaccia utente

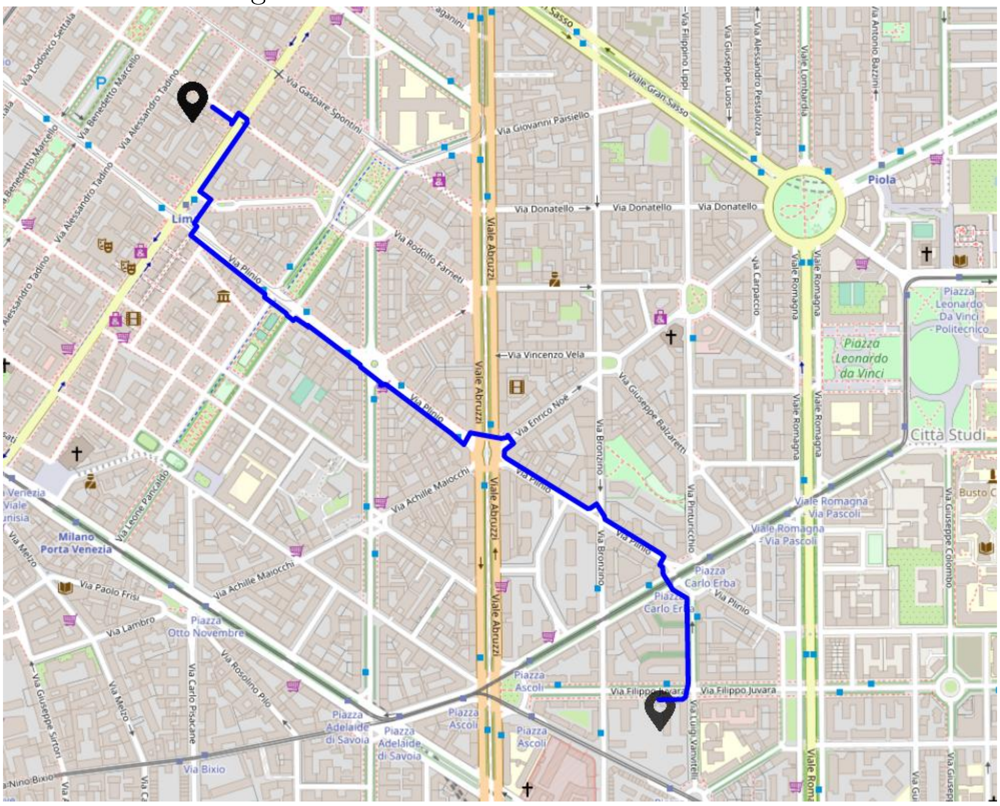
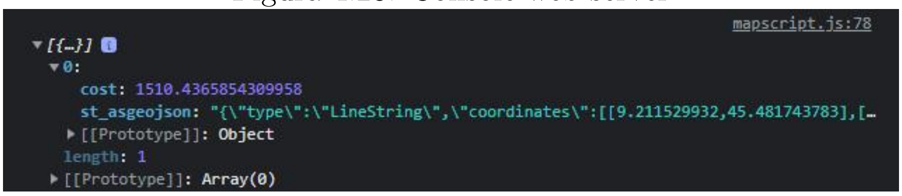


Figura 4.23: Console web server



4.4 ETL: Extract, Transform, Load

Con ***extract, transform, load (ETL)*** ci si riferisce generalmente al processo di estrazione di dati da una o più sorgenti verso un sistema di destinazione che rappresenta i dati in maniera differente rispetto al modello di partenza. Ne consegue che i dati estratti debbano attraversare un processo di elaborazione che li renda conformi al nuovo modello dati a cui segue la fase di caricamento nella destinazione. Come ricordato nella fase di analisi e progettazione l'unico componente *on premise* e costruito su misura in base al cliente è quello che si occupa appunto del processo di ETL. Ogni cliente che aderisce al servizio infatti avrà un proprio modello dati che fa riferimento ad una particolare rete tecnologica di interesse. Per poter utilizzare l'applicazione web sarà dunque necessario rendere conformi i dati di provenienza al modello definito in precedenza.

Abbiamo quindi sviluppato uno script servendoci del linguaggio Python e di alcune sue librerie che legge i dati di partenza da dei file CSV, li organizza nelle tabelle di destinazione e li carica direttamente su *PostgreSQL* con estensione *PostGIS*. Tale script sarà il medesimo che verrà attivato nel caso in cui un utente Admin, dopo aver effettuato il login all'applicazione, decida di effettuare un caricamento asincrono dei dati.
I dati ci sono stati forniti attraverso dei file CSV (Fig 4.24 - 4.25): alcuni file contengono le informazioni sugli elementi che compongono la rete con inclusi i campi geometrici, altri invece conterranno le relazioni tra i veri elementi, come il legame tra la parte logica della rete e quella civile.

Figura 4.24: files CSV con informazioni sulla rete

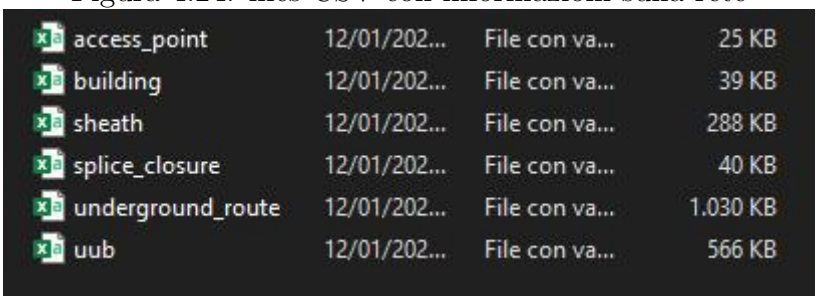
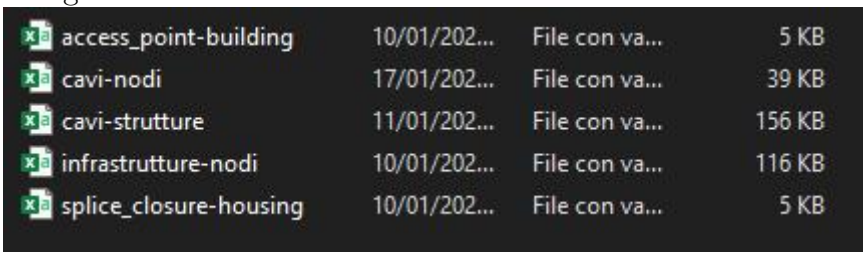


Figura 4.25: files CSV con relazioni tra elementi della rete



Nella figura 4.24 vengono elencati gli elementi fondamentali che compongono la rete di partenza, estratti per costruire il primo prototipo di applicazione. Nella tabella di seguito vengono messi in relazione gli elementi secondo il modello del cliente con quello dell'applicazione web.

TABELLA DI PARTENZA	SIGNIFICATO	TABELLA DI ARRIVO
access_point	Punto di accesso della rete nell'abitazione	Structure Node
building	Edificio, centrale	Structure Node
sheath	Cavo	Logical Link
splice_closure	Giunto ottico	Logical Node
underground_route	Scavo	Structure Link
uub	Pozzetti	Structure Node

Si può notare che vengono raggruppati elementi diversi con caratteristiche comuni nelle stesse tabelle di destinazione. Ad esempio oggetti di tipo *building* nel modello dati dell'applicazione web vengono equiparati con oggetti di tipo *uub* o *access_point* in quanto tutti costituisco un elemento puntuale della rete civile sebbene ognuno abbia le sue caratteristiche specifiche; per le funzioni che andremo ad implementare le diverse tipologie di provenienza possono essere trattate indifferentemente.

Vediamo ora lo script più da vicino:

Figura 4.26:

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://username:password@host:port/database')
mode_sql = "append"

import pandas as pd
from random import randint
import json
import csv
import collections
import numpy as np
import pickle

f_out_dir = r'absolute_path\SNB_tables'
f_base = r'absolute_path\exported_raw_tables'
f_join = r'absolute_path\join'
```

Nella figura 4.26 vengono definite le librerie che saranno poi utilizzate successivamente. In primo luogo viene importata e configurata la libreria ***sqlalchemy***: ci serviremo delle funzionalità messe a disposizione per effettuare direttamente i caricamenti nel database. Si può notare che la modalità di inserimento è definita come *append* in quanto come detto poco sopra l'algoritmo non sarà necessario solo per un primo inserimento ma anche per successivi caricamenti asincroni di dati. Tra le altre librerie utilizzate troviamo ***pandas*** che mette a disposizioni funzioni per la manipolazione e l'analisi dei dati e ***pickle*** che implementa un basilare ma potente algoritmo per serializzare e de-serializzare una struttura di oggetti Python. Infine vengono definiti i percorsi base da cui poi andremo ad estrarre i dati di cui abbiamo bisogno.
Di seguito vengono riportati due esempi di creazione di tabelle del modello dati finale.

Prices Nella figura 4.27 viene riportato la parte di script che dalle tabelle di partenza elabora i dati e crea la tabella *Prices*. Per prima cosa definiamo di ogni tabella di partenza i campi che vogliamo che siano presenti nella tabella finale, dopodiché vengono letti i file CSV da cui recuperare i dati di interesse, attraversano una fase di controllo ed elaborazione

e infine vengono salvati nel *dataframe pandas df_out*. Come ultimo passaggio, con l'ausilio della libreria *sqlalchemy* il dataframe viene direttamente inserito nel database desiderato in base alla stringa di configurazione definita in figura 4.26.

Structure Node La logica della parte di algoritmo che si occupa di creare ed esportare la tabella Structure Node è simile a quella precedente; lo script viene riportato in figura 4.28 - 4.29. In primo luogo viene creata - attraverso un id univoco - l'associazione con la tabella *Prices*. Ad ogni elemento della tabella Structure Node viene quindi assegnato un id da cui è possibile risalire al prezzo per quel tipo di infrastruttura. In seguito, per ogni tabella di partenza vengono estratti i dati di interesse tra cui il campo geometrico, i campi *available_capacity* e il campo *total_capacity* e il campo *data* che conterrà i campi rimanenti non strettamente necessari per l'algoritmo di routing in formato Json. Infine i risultati delle elaborazioni appena descritti vengono fusi in un unico *dataframe* e successivamente inseriti nel database.

Figura 4.27: Creazione della tabella Prices

```
f_names_suffixes = ["access_point.csv", "building.csv", "splice_closure.csv",
                    "uub.csv", "underground_route.csv", "sheath.csv"]
f_names = [f_base + '\\' + f for f in f_names_suffixes]

table_types = ["access_point", "building", "splice_closure",
               "uub", "underground_route", "sheath"]
cat_or_type = ["type", "type", "category", "type", "category", "category"]
constructi = ["Realizzato", "Realizzato", "constructi", "constructi", "constructi", "constructi"]
owner_or_costruttor = ["Inesistente", "Inesistente", "owner", "costruttor", "costruttor", "owner"]

df_list = []
for f_name in f_names:
    df_list.append(pd.read_csv(f_name))

type_list = set()
for i, df in enumerate(df_list):
    cat_list_tmp = df[cat_or_type[i]].to_list()

    if owner_or_costruttor[i] == "Inesistente":
        owner_list_tmp = ["Inesistente"] * len(cat_list_tmp)
    else:
        owner_list_tmp = df[owner_or_costruttor[i]].to_list()

    if constructi[i] == "Realizzato":
        constructi_tmp = ["Realizzato"] * len(cat_list_tmp)
    else:
        constructi_tmp = df[constructi[i]].to_list()

    table_type = [table_types[i]] * len(cat_list_tmp)

    for j in range(len(cat_list_tmp)):
        type_list.add((cat_list_tmp[j], owner_list_tmp[j], constructi_tmp[j], table_type[j]))

# Aggiungo costo dopo per non creare duplicati nel set "type_list"
t_outs = []
for t in type_list:
    if t[3] in ["underground_route", "sheath"]:
        cost = randint(1, 10)
    else:
        cost = 0
    t_out = (t[0], t[1], t[2], t[3], cost)
    t_outs.append(t_out)

t_outs.append(("base", "Inesistente", "Realizzato", "mit_bay", 0))
t_outs.append(("base", "Inesistente", "Guasto", "sheath", 99999))

df_out = pd.DataFrame(t_outs)
df_out.columns = ["category", "owner", "status", "table_type", "cost"]
df_out.head()

df_out = df_out.reset_index()

df_out.columns = ["id_tmp", "category", "owner", "status", "table_type", "cost"]

df_out.to_sql('prices', engine, if_exists=mode_sql)
engine.execute('ALTER TABLE prices RENAME COLUMN cost TO price')
```

Figura 4.28: Creazione della tabella Structure Node (1)

```
df_uub = pd.read_csv(f_base + r"\uub.csv")
df_ap = pd.read_csv(f_base + r"\access_point.csv")
df_building = pd.read_csv(f_base + r"\building.csv")

list_join_ids_uub = []
list_join_ids_ap = []
list_join_ids_building = []
df_type = df_out

for index, row_str in df_uub.iterrows():
    for index, row_type in df_type.iterrows():
        if row_str["type"] == row_type["category"] and
        row_str["costruttor"] == row_type["owner"] and
        row_str["constructi"] == row_type["status"] and
        "uub" == row_type["table_type"]:
            list_join_ids_uub.append(row_type["id_tmp"])
            break

for index, row_str in df_ap.iterrows():
    for index, row_type in df_type.iterrows():
        if row_str["type"] == row_type["category"] and
        "Inesistente" == row_type["owner"] and
        "Realizzato" == row_type["status"] and
        "access_point" == row_type["table_type"]:
            list_join_ids_ap.append(row_type["id_tmp"])
            break

for index, row_str in df_building.iterrows():
    for index, row_type in df_type.iterrows():
        if row_str["type"] == row_type["category"] and
        "Inesistente" == row_type["owner"] and
        "Realizzato" == row_type["status"] and
        "building" == row_type["table_type"]:
            list_join_ids_building.append(row_type["id_tmp"])
            break

df_uub["id_price"] = pd.Series(list_join_ids_uub)
df_ap["id_price"] = pd.Series(list_join_ids_ap)
df_building["id_price"] = pd.Series(list_join_ids_building)

#Estrazione UUB
main_columns_uub = ["id", "id_price", "geom"]
data_columns_uub = list(set(list(df_uub.columns)) - set(main_columns_uub) - set(['type', 'costruttor', "constructi"]))

uub_data_raw = df_uub[data_columns_uub]
list_data_uub = json.loads(uub_data_raw.to_json(orient="records"))
list_data_uub_str = [str(el) for el in list_data_uub]
df_uub["data"] = pd.Series(list_data_uub_str)

available_capacity = [-1] * len(list_join_ids_uub)
total_capacity = [-1] * len(list_join_ids_uub)
df_uub["available_capacity"] = available_capacity
df_uub["total_capacity"] = total_capacity

final_columns_uub = ["id", "id_price", "available_capacity", "total_capacity", "geom", "data"]
df_uub = df_uub[final_columns_uub]

df_uub.columns = ["id_tmp", "id_price", "available_capacity", "total_capacity", "geom", "data"]
```

Figura 4.29: Creazione della tabella Structure Node (2)

```
#Estrazione Access Point

main_columns_ap = ["id", "id_price", "geom"]
data_columns_ap = list(set(list(df_ap.columns)) - set(main_columns_ap) - set(['type']))

ap_data_raw = df_ap[data_columns_ap]
list_data_ap = json.loads(ap_data_raw.to_json(orient="records"))
list_data_ap_str = [str(el) for el in list_data_ap]
df_ap["data"] = pd.Series(list_data_ap_str)

available_capacity = [-1] * len(list_join_ids_ap)
total_capacity = [-1] * len(list_join_ids_ap)
df_ap["available_capacity"] = available_capacity
df_ap["total_capacity"] = total_capacity

final_columns_ap = ["id", "id_price", "available_capacity", "total_capacity", "geom", "data"]
df_ap = df_ap[final_columns_ap]

df_ap.columns = ["id_tmp", "id_price", "available_capacity", "total_capacity", "geom", "data"]

#Estrazione Building

main_columns_b = ["id", "id_price", "geom"]
data_columns_b = list(set(list(df_building.columns)) - set(main_columns_b) - set(['type']))

b_data_raw = df_building[data_columns_b]
list_data_b = json.loads(b_data_raw.to_json(orient="records"))
list_data_b_str = [str(el) for el in list_data_b]
df_building["data"] = pd.Series(list_data_b_str)

available_capacity = [-1] * len(list_join_ids_building)
total_capacity = [-1] * len(list_join_ids_building)
df_building["available_capacity"] = available_capacity
df_building["total_capacity"] = total_capacity

final_columns_b = ["id", "id_price", "available_capacity", "total_capacity", "geom", "data"]
df_building = df_building[final_columns_b]

df_building.columns = ["id_tmp", "id_price", "available_capacity", "total_capacity", "geom", "data"]

#Merge all structure node
df_structure_nodes = pd.concat([df_building, df_ap, df_uub], axis=0)

#Insert DB
df_structure_nodes.to_sql('structure_node', engine, if_exists=mode_sql)
engine.execute('ALTER TABLE structure_node ALTER COLUMN geom TYPE geometry')
```

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi sono stati presentati nel dettaglio i passaggi fondamentali compiuti fino ad ora dal team di progetto nella progettazione e sviluppo dell'applicazione oggetto di studio. In particolare, sono stati approfonditi gli aspetti progettuali e e i fattori che hanno portato alla scelta di alcune tecnologie a discapito di altre. Ripercorriamo nel dettaglio quanto fatto fino ad ora:

1. Individuazione di un progetto di applicazione a seguito di riscontri di mercato di necessità e interesse.
2. Definizione delle tecnologie necessarie per l'implementazione dell'applicazione.
3. Analisi delle necessità di un potenziale cliente e definizione dei requisiti.
4. Progettazione dell'architettura (architettura web).
5. Progettazione di un modello dati che risponda alle esigenze già definite.
6. Implementazione dell'architettura in *cloud* (implementazione su AWS) utilizzando dati grezzi, non conformi al modello dati.
7. Esecuzione di alcuni test su un primo prototipo di applicazione.
8. Sviluppo del componente di ETL con dati di prova.

Definito quindi lo stato attuale dei lavori è possibile immaginare i passaggi successivi per arrivare ad una demo completa e quindi al successivo inserimento del prodotto nel mercato:

1. Implementazione di una base di dati conforme al modello definito in questa trattazione, servendosi del componente di ETL implementato sui dati di prova.
2. Correzione dell'*endpoint* già definito e aggiunta di nuovi *endpoint*.
3. Modifica e aggiornamento dei test sul *back-end*.
4. Modifica al *web server* in modo tale che gli *endpoint* definiti trovino una corrispondenza anche lato client. Vengono messe a disposizione dell'utente finale le funzioni di **Visualizzazione storico, archiviazione esito, aggiornamento esito, visualizzazione Rete, visualizzazione Rete pianificata, e inserimento dati asincrono.**

5. Implementazione dell'interfaccia grafica in modo che sia accessibile, *user-friendly* e soddisfacente nell'utilizzo.

6. Aggiunta della schermata di Login e delle logiche sui permessi di utilizzo.

Si prevede di arrivare ad una prima demo completa entro almeno due mesi. Una volta che si dispone di un prodotto completo la parte commerciale dell'azienda valuterà se, quando, a chi e in che modalità proporre questo tipo di prodotto.

Per come è stato impostato il lavoro inoltre, bisogna infine sottolineare che un'applicazione di questo tipo richiede un lavoro intenso e costante di aggiornamento. Come sottolineato in precedenza, ogni nuovo cliente necessiterà di un componente ETL adatto al formato dati di cui dispone; inoltre, è un prodotto che ha l'ambizione di evolversi con il numero dei clienti serviti: maggiori gli utilizzatori dell'applicazione, maggiori saranno le richieste particolari che potranno pervenire; dunque, sarà necessario anche in questo senso un costante lavoro di aggiornamento.

Ringraziamenti

Mi sento in dovere di dedicare questa pagina del presente elaborato alle persone che mi hanno supportato nella redazione dello stesso.

Innanzitutto, ringrazio il mio relatore Chiar.mo Prof. Emanuele Frontoni e il mio correlatore Chiar.mo Prof. Adriano Mancini, sempre pronti a darmi le giuste indicazioni in ogni fase della realizzazione dell'elaborato. Grazie a voi ho accresciuto le mie conoscenze e le mie competenze.

Ringrazio l'azienda EBWorld S.r.l. e tutto il suo personale per avermi dato la possibilità di svolgere il mio lavoro di tesi in un luogo interessante e dinamico, che ha permesso di mettermi in gioco e fare un'esperienza che sarà preziosa per il mio futuro. In particolare ci tengo a ringraziare Luka, Maruska, Nicola, Roberto, Dimitri, Filippo e Matteo perché più di altri mi hanno accompagnato e sostenuto in questo percorso.

Un ringraziamento particolare va al team di progetto; ad Alessandro e Lorenzo un sentito grazie per avere condiviso con me questo percorso.

Ringrazio infine i miei genitori, mia sorella, mia nonna e tutta la mia famiglia perché senza di loro non avrei mai potuto intraprendere questo percorso di studi.

Bibliografia

[] *Amazon web services documentation*. URL: https://docs.aws.amazon.com/. (accessed: 19.12.2021).

[] *Apache HTTP Server*. URL: https://it.wikipedia.org/wiki/Apache_HTTP_Server. (accessed: 01.01.2022).

[] *Architettura three-tier*. URL: https://it.wikipedia.org/wiki/Architettura_three-tier. (accessed: 03.01.2022).

[] *Dijkstra's algorithm*. URL: https://en.wikipedia.org/wiki/Dijkstra%5C%27s_algorithm. (accessed: 02.02.2022).

[] *Geographic Information System*. URL: https://www.nationalgeographic.org/encyclopedia/geographic-information-system-gis/. (accessed: 15.12.2021).

[] *Geographic Information System*. URL: https://it.wikipedia.org/wiki/Geographic_information_system. (accessed: 15.12.2021).

[] *HTML & CSS*. URL: https://www.w3.org/standards/webdesign/htmlcss. (accessed: 26.12.2021).

[] *JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript. (accessed: 20.12.2021).

[] *Load balancing (computing)*. URL: https://en.wikipedia.org/wiki/Load_balancing_(computing). (accessed: 04.01.2022).

[] *Node.js documentation*. URL: https://nodejs.dev/learn. (accessed: 26.12.2021).

[] *OSRM, Open source routing machine documentation*. URL: http://project-osrm.org/docs/v5.24.0/api/#. (accessed: 16.12.2021).

[] *PgRouting documentation*. URL: https://pgrouting.org/documentation.html. (accessed: 17.12.2021).

[] *PostGIS documentation*. URL: https://postgis.net/docs/. (accessed: 17.12.2021).

[] *PostgreSQL documentation*. URL: https://www.postgresql.org/docs/. (accessed: 16.12.2021).

[] *Python documentation*. URL: https://docs.python.org/3/. (accessed: 03.01.2022).

[Arm+10] Michael Armbrust et al. «A view of cloud computing». In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[Dan93] Tim Berners-Lee e Daniel Connolly. «Hypertext Markup Language (HTML) - A Representation of Textual Information and MetaInformation for Retrieval and Interchange». In: *World Wide Web Consortium* (1993).

[Dij59] Edsger W. Dijkstra. «A note on two problems in connexion with graphs». In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[ESR98] ESRI ESRI. «Shapefile technical description». In: *An ESRI white paper* 39 (1998).

[Fie00] Roy Thomas Fielding. «Architectural styles and the design of network-based software architectures». Publication. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[Ope10] Open Geospatial Consortium. *OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture*. 2010. URL: http://www.opengeospatial.org/standards/sfa.