POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Studio e sviluppo di un cruscotto per il monitoraggio dell'inquinamento urbano

Relatore
prof.ssa Silvia Chiusano

Candidato
Matteo Rolando

Dicembre 2017

Indice

Capitolo 1

Introduzione

L'inquinamento atmosferico presente nel contesto urbano causa una sempre maggiore preoccupazione ai cittadini, in quanto può produrre effetti negativi sulla salute umana. Per questo motivo è fondamentale poter analizzare gli elementi che concorrono a modificare ed alterare condizioni e qualità dell'aria, con la speranza di poter agire in modo da prevenire eventuali situazioni critiche, sfruttando le conoscenze emerse dalle analisi effettuate. Grazie alle innumerevoli reti di sensori, dislocati in varie aree urbane, preposti alla raccolta di diversi tipi di informazione, è possibile porre in relazione dati appartenenti a domini diversi, in grado comunque di contribuire alla variazione delle caratteristiche qualitative dell'aria e pertanto fornire una visione più generale del fenomeno.

In questa tesi viene presentato il processo di studio e sviluppo di uno strumento di monitoraggio dell'inquinamento urbano basato su tecniche di business intelligence, grazie al quale è possibile comprendere quali siano le correlazioni tra l'andamento nel tempo di alcuni inquinanti e altri fattori di tipo antropico e climatico.

I dati relativi agli inquinanti sono stati integrati con quelli sul traffico urbano e con quelli inerenti al meteo, per valutare al meglio l'impatto di questi fattori sulla qualità dell'aria. È stato progettato un data warehouse, ossia di un sistema per l'immagazzinamento di dati tale che consentisse, in maniera semplice ed efficace, lo svolgimento di analisi su questi dati. Sì è definito anche un insieme di Key Performance Indicators (KPI) per il monitoraggio dell'inquinamento atmosferico e il raffronto dello stesso con le

informazioni riguardanti traffico e condizioni meteorologiche, con diversi livelli di granularità temporale. La visualizzazione grafica di tali KPI è stata resa tramite l'utilizzo di alcuni scenari appositamente strutturati. Si è proceduto allo studio, alla progettazione e allo sviluppo di un cruscotto che fosse in grado di implementare e rendere in maniera semplice tali scenari di analisi attraverso l'utilizzo di tecnologie all'avanguardia per sfruttare al meglio il potenziale dei mezzi moderni: per questo, in particolare, si è deciso di utilizzare un'architettura client-server che permettesse lo sviluppo di una applicazione web a pagina singola (Single Page Application), grazie ai frameworks AngularJS e Node.js.

La tesi è strutturata come segue: il capitolo 2 espone le tecniche di business intelligence e il processo di progettazione di un data warehouse, il capitolo 3 descrive il dataset utilizzato e le analisi effettuate mentre il capitolo 4 tratta la progettazione e lo sviluppo del cruscotto.

Capitolo 2

Business Intelligence

In questo capitolo vengono presentate le metodologie di business intelligence considerate in questa tesi. Viene quindi descritto il processo di studio e progettazione di un data warehouse, sia dal punto di vista concettuale, sia da quello logico.

Nel corso degli anni, la mole di dati creati e raccolti da parte delle organizzazioni è aumentata notevolmente grazie all'evoluzione tecnologica e alla diffusione di sistemi di immagazzinamento degli stessi. In precedenza questi dati dovevano essere raccolti manualmente, richiedendo tempi di acquisizione lunghi e mole di lavoro complessa e pertanto molti di questi venivano trascurati in quanto non significativi nell'immediato. Con lo sviluppo di supporti informatici, il processo di raccolta e memorizzazione dei dati è divenuto più rapito e agevole. Di conseguenza è sorta la necessità di riuscire ad analizzare una grande quantità di dati in tempi brevi con lo scopo di generare informazioni utili al contesto dell'organizzazione. Infatti, tramite l'analisi dei dati, si acquisiscono nuove conoscenze grazie alle quali è possibile effettuare migliori scelte strategiche, con benefici per tutta l'organizzazione stessa. Sono nate, pertanto, diverse tecniche per agevolare il processo analitico, ognuna con i propri approcci e sfaccettature, anche molto diversi tra loro, a seconda del dominio di utilizzo. Ciò che le accomuna tutte è il processo concettuale che porta dall'acquisizione dei dati alla conoscenza, passando attraverso alcune fasi quali pulizia, trasformazione e modellazione dei dati. Una volta ottenute le informazioni necessarie, come ultimo passo, è necessario rendere disponibili i risultati delle analisi attraverso rappresentazioni visive chiare ed efficienti.

Tutte queste tecniche di analisi dei dati vengono indicate con il termine “business intelligence”, che comprende sia i sistemi volti ad analizzare presente e passato per studiare e comprendere la fenomenologia dei processi, sia quelli rivolti ad una previsione degli eventi futuri, con lo scopo di individuare le strategie di volta in volta più idonee. Da un punto di vista prettamente pratico, questa tecnica viene resa tramite utilizzo di alcuni strumenti software come i data warehouse (DW) e i cruscotti.

Un DW è un archivio informatico ottimizzato per memorizzare i dati aziendali in modo da rendere efficienti le analisi sugli stessi e consentire all’azienda la scelta di migliori strategie.

Con il termine cruscotto si intende uno strumento in grado di presentare le informazioni rilevanti in maniera sintetica ed immediata, mettendo in luce efficacemente i risultati delle analisi effettuate utilizzando grafici e tabelle.

2.1 Data Warehouse

Il *data warehouse* è una base di dati ideata per supportare al meglio le decisioni aziendali. Solitamente è meglio separarlo dal database dell’organizzazione in quanto le analisi dei dati effettuate sul data warehouse sono complesse e rallenterebbero le transazioni operative. Rispetto ad altri sistemi per il sopporto alle decisioni, l’integrazione di dati provenienti da diverse fonti costituisce la principale caratteristica dei data warehouse [2]. La raccolta dei dati dovrebbe essere:

- *Integrata*. All’interno dei data warehouse confluiscono dati da molte fonti diverse, pertanto è fondamentale inserirli in modo che siano confrontabili.
- *Orientata al soggetto*. I dati devono essere memorizzati in modo da favorire la produzione di informazioni e renderle facilmente utilizzabili dagli utenti.

- *Variabile nel tempo*. Nel data warehouse vengono archiviati dati relativi ad un fenomeno con un'estensione temporale piuttosto ampia. Anche se non sempre tali dati sono aggiornati, sono in grado di fornire un quadro storico ampio e completo che favorisce l'analisi del fenomeno.
- *Non volatile*. Una volta raccolti e inseriti nel data warehouse, i dati non sono più modificabili e l'accesso ai dati avviene in sola lettura.

Il data warehouse, pertanto, descrive il processo di acquisizione, trasformazione e distribuzione di informazioni che saranno di supporto alle decisioni strategiche dell'organizzazione e che verranno rese disponibili attraverso tools per la visualizzazione, come i cruscotti.

L'architettura di un data warehouse ha come caratteristica la separazione tra l'elaborazione transazionale e l'analisi dei dati, in modo da renderlo maggiormente scalabile: si parla quindi di architettura a due o più livelli.

Dal punto di vista architetturale, il data warehouse è costituito da più elementi, come descritto nella Figura 2.1:

- *Sorgenti dei dati*. Insieme dei dati da utilizzarsi nell'analisi provenienti da diversi database, anche esterni all'azienda.
- *Strumenti di ETL*. Servono per preparare i dati da introdurre nel data warehouse attraverso un processo eseguito al primo popolamento del DW e durante gli aggiornamenti periodici del contenuto del DW. Con la sigla ETL si intende:
    - *Estrazione*: acquisizione dei dati dalle sorgenti, scelti in base alla loro qualità. Può essere statica, al primo popolamento del DW, o incrementale, per gli aggiornamenti successivi.
    - *Pulitura*: in questa categoria rientrano tutte quelle operazioni effettuate sui dati con lo scopo di migliorarne la qualità, in particolar modo in riferimento a correttezza e consistenza. Ogni problema rilevato sui dati richiede tecniche specifiche, come quelle basate su dizionari (per risolvere

errori di battitura o formato), quelle di fusione approssimata (in grado di riconoscere duplicati) e quelle adatte alla identificazione di outliers e valori errati.

  - *Trasformazione*: i dati vengono rimodellati e integrati per essere compatibili con il formato proprio del data warehouse.
  - *Caricamento dei dati*: i dati così processati vengono inseriti all'interno del data warehouse stesso. Questo passo richiede che vengano garantite le proprietà transazionali di affidabilità e atomicità.

- *Staging area*. È un'area di transito che può essere inserita tra la sorgente dei dati e il data warehouse, permettendo operazioni di trasformazione e pulizia anche molto complesse.
- *Metadati*. Raccolgono informazioni sui dati inseriti e sulla loro struttura, indicandone valore, funzione, modalità di utilizzo.
- *Data warehouse* propriamente detto e *data mart*. Il primo contiene informazioni riguardanti tutti gli aspetti da analizzare, pertanto il processo di modellazione e progettazione richiede molto tempo. Il data mart, invece, è un sottoinsieme dipartimentale focalizzato su un unico settore; è possibile realizzarlo più rapidamente sia alimentandolo a partire dal data warehouse primario, sia direttamente dalle sorgenti.
- *Strumenti di analisi*. Il data warehouse, una volta memorizzati i dati, si occupa di effettuare delle analisi appoggiandosi su altre tecnologie.

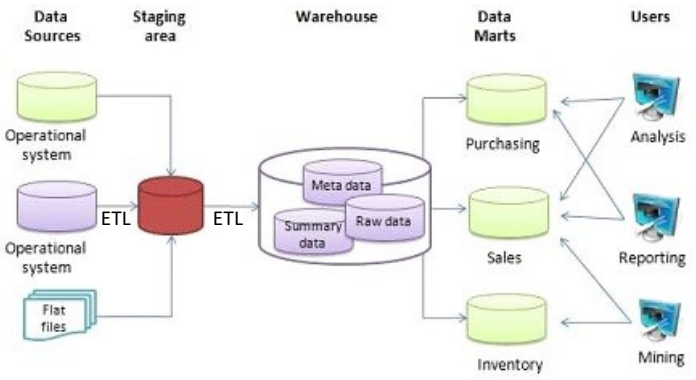

Figura 2.1 La struttura di un data warehouse [3]

Per progettare un data warehouse si possono seguire due differenti approcci: *top-down* o *bottom-up*. Mentre il primo prevede la realizzazione di data warehouse globale che fornisce una visione completa dell'azienda, con costi significativi e metodologie complesse, nell'approccio bottom-up, si inizia la costruzione dei singoli data mart, che potranno essere aggiunti in maniera incrementale al data warehouse riducendo il costo e il tempo di progettazione.

Il primo passo per la progettazione di un data warehouse è quello di analizzarne i requisiti applicativi, comprendendo le tipologie di analisi che dovranno essere soddisfatte dal sistema e raccogliendo i dati da fonti attendibili con lo scopo di ottenere un data mart che sia strategico per l'azienda. In seguito è necessario valutare alcuni requisiti strutturali come lo spazio disponibile e il tipo di architettura del sistema.

2.1.1 Progettazione concettuale di un data warehouse

Una volta definiti i requisiti, è possibile iniziare con la fase di progettazione concettuale. A differenza dei database operazionali, nei quali questa fase viene eseguita con la modellazione di un diagramma ER (entità-relazione), per un data warehouse non esiste un formalismo comunemente accettato. Tuttavia è frequente l'utilizzo di un modello creato ad hoc per la progettazione di data warehouse denominato "Dimensional fact model" (DFM) [4].

Il DFM è un modello grafico estremamente intuitivo, facilmente comprensibile sia dagli analisti che dagli utenti non tecnici. È basato sul "fatto", un concetto rilevante ai fini del processo di decision-making. Il "fatto" tipicamente modella un insieme di eventi che hanno luogo all'interno dell'organizzazione. Inoltre è dipendente dal tempo. Per ciascun fatto vengono definiti schemi per modellare:

- *Misura*. Descrive una proprietà numerica di un fatto, un attributo quantitativo interessante per lo scopo dell'analisi. (ad esempio la quantità di inquinante presente nell'aria).
- *Dimensione*. Proprietà con dominio finito che rappresenta le coordinate di analisi di un fatto. È caratterizzata da molti attributi. Tipicamente sono presenti più dimensioni. (ad esempio la data di raccolta della misura e il posto dove è stata effettuata).
- *Gerarchia*. Consiste in un grafo ad albero nel quale i nodi sono gli attributi della dimensione e gli archi modellano una relazione N:1 tra le coppie consecutive di attributi (ad esempio anno – mese – giorno, città – area – indirizzo).

2.1.2 Progettazione logica di un data warehouse

Dopo aver definito il DFM e quindi determinato lo schema concettuale del data warehouse, si procede con la progettazione logica. L'obiettivo di questa fase è quello di tradurre il DFM in uno schema ottimizzato per le operazioni di analisi da effettuare sui dati, ossia in uno schema che definisca l'insieme delle tabelle necessarie per la memorizzazione dei dati nel data mart. Anche in questo caso non è possibile utilizzare il modello relazionale come nelle basi di dati operazionali, in quanto la progettazione del data warehouse si basa su principi differenti, come la ridondanza dei dati e la denormalizzazione delle tabelle. Sono stati pertanto introdotti alcuni schemi ad hoc a supporto della progettazione logica di un DW:

- *A stella*. Questo schema prevede che ogni dimensione venga resa con una tabella contente tutti i suoi attributi. Le gerarchie non sono rappresentate esplicitamente e la chiave primaria è generata artificialmente. Si crea un'ulteriore tabella per ogni schema di fatto avente come chiave primaria la combinazione delle chiavi esterne delle dimensioni e come attributi della tabella le misure del fatto.

- *Snowflake*. Lo scopo di questo modello è di ridurre lo spazio necessario alla memorizzazione dei dati, con lo svantaggio di aumentare il costo della ricostruzione dell'informazione. Può essere utile solo quando porzioni di una gerarchia sono condivise da più dimensioni o in presenza di viste materializzate. Vengono separate le dipendenze funzionali tra attributi, frazionando i dati di una dimensione in più tabelle.

2.1.3 Viste materializzate

In alcuni casi è possibile aumentare l'efficienza delle interrogazioni eseguite sul DW, soprattutto se esse richiedono aggregazioni, con l'utilizzo di viste materializzate. Queste ultime sono sommari precalcolati della tabella dei fatti che vengono memorizzati esplicitamente nel data warehouse. La scelta delle viste da introdurre dipende dal tipo e dal numero di interrogazioni che verranno effettuate, in modo da minimizzare il costo dell'esecuzione di tutte le richieste. Tuttavia è necessario tenere in considerazione sia il tempo di aggiornamento della vista quando vengono aggiunti nuovi dati, sia la quantità di spazio necessaria a contenere tali viste.

Capitolo 3

Rappresentazione ed analisi dei dati

Per attuare il processo di analisi sull'inquinamento urbano, nel quale siano evidenti le correlazioni tra l'andamento di alcuni inquinanti e altri fattori, attraverso tecniche di business intelligence, è necessario reperire un'adeguata quantità di dati per poterli rappresentare e analizzare in maniera opportuna.

In questo capitolo sono presentate le tipologie di open data considerate in questo studio. Viene quindi descritto il modello di rappresentazione dei dati utilizzato e gli scenari di analisi realizzati mediante il cruscotto.

3.1 Sorgente dei dati

Dal momento che l'analisi da effettuare è inerente la qualità dell'aria all'interno delle aree urbane di una città, sono state prese in considerazione alcune categorie di dato, sia riguardanti direttamente la qualità dell'aria, sia relative ad altri aspetti, che possono comunque avere un'influenza sull'inquinamento atmosferico.

Nel dettaglio i dati utilizzati nel cruscotto sono i seguenti:

- *Concentrazione di un inquinante all'interno di un'area urbana*; in particolare sono stati presi in considerazione alcuni inquinanti, come materia particolata

(PM_{10} e $PM_{2.5}$), monossido di carbonio (CO), ozono troposferico (O_3), biossido di azoto (NO_2), benzene (C_6H_6) e biossido di zolfo (SO_2). Questi valori vengono raccolti tramite sensori presenti in stazioni di monitoraggio localizzate in diverse aree urbane. Ciascuna stazione è identificata da una posizione geografica e dispone di più sensori, identificati univocamente da un codice. Ognuno è in grado di misurare il valore di un inquinante con granularità oraria o giornaliera. Si tratta di dati open, i quali vengono resi disponibili da ARPA Lombardia [5].

- *Dati sul traffico dei veicoli*, determinati come il numero di veicoli in ingresso all'interno di una zona in un periodo temporale. Ogni ora, i sensori di una rete georeferenziata forniscono i dati relativi alla quantità di veicoli entrati nella zona, divisi per tipologia. Per meglio caratterizzare i dati relativi al traffico è stato scelto di categorizzare il traffico in base alla tipologia di carburante del mezzo (ad esempio benzina, gasolio, GPL) e al tipo di utilizzo dello stesso (ad esempio bus, taxi, trasporto merci, privati). Anche in questo caso i dati scelti sono open, raccolti e distribuiti dal comune di Milano [6].
- Misurazione di alcuni *aspetti meteorologici*, in particolare temperatura, umidità, pressione e intensità del vento. Tali dati sono stati importati dal web service "*Weather Underground*" [7], il quale raccoglie dati da stazioni meteorologiche private, registrate dai proprietari stessi. La granularità con cui vengono raccolti i dati è aleatoria, tuttavia è stata considerata una media con frequenza di 15 minuti.

3.2 Preparazione e modellazione

Per consentire un'analisi proficua sulla qualità dell'aria, i dati sull'inquinamento sono stati messi in relazione con altri fattori determinanti nel descrivere il contesto ambientale, quali il traffico e le condizioni meteorologiche.

Dal momento che la frequenza con cui vengono rilevati i diversi valori dai sensori varia a seconda della tipologia di dato, è stato necessario rielaborarli e, in particolare, scegliere

una granularità spazio-temporale comune a tutte le categorie. Considerando che le analisi di interesse si basano prevalentemente sui dati relativi alla concentrazione degli inquinanti, questi sono stati scelti come riferimento sia per la scala temporale sia per quella spaziale.

In particolare, siccome i dati sul meteo sono stati prelevati da un gran numero di stazioni private disposte in maniera non uniforme in tutte le aree della città, è stato necessario effettuare una trasformazione delle informazioni raccolte: ad ogni stazione di monitoraggio sono stati associati i dati meteorologici ottenuti calcolando la media ponderata basata sulla distanza dei valori forniti dalle tre stazioni private del servizio *"Weather Underground"* più vicine. Per quanto riguarda i dati sul traffico, per ogni zona la quantità di veicoli entranti sono stati associati a ciascuna delle stazioni di monitoraggio degli inquinanti presenti nella stessa. In entrambi i casi il timestamp dei dati è stato allineato a quello degli inquinanti: tra i timestamp dei dati relativi agli inquinanti, presenti nella stazione di monitoraggio precedentemente associata, viene scelto quello che presenta uno scarto minore e viene applicato al dato di partenza.

I dati, così rielaborati, sono stati organizzati a livello concettuale in una "*fact table*", secondo il *Dimensional Fact Model* descritto dalla Figura 3.1, e inseriti all'interno di un data warehouse. La misura di tale "*fact table*" è costituita principalmente dal valore della concentrazione oraria di un inquinante per ciascun sensore, al quale vengono aggiunte le informazioni relative alla situazione meteorologica e i dati relativi al traffico. Per poter effettuare analisi a diversi livelli di granularità spaziale e temporale, sono state prese in considerazione diverse dimensioni:

- Una *dimensione spaziale*, con informazioni legate alla posizione geografica del sensore quali indirizzo, tipo di sensore, latitudine e longitudine, area della città.
- Due *dimensioni temporali*, una legata all'orario della misurazione, l'altra al giorno della misurazione. La prima gerarchia aggrega l'ora in intervalli diversi e in base al periodo del giorno: mattino, pomeriggio, sera e notte. La seconda gerarchia mette in relazione il giorno, il mese e l'anno, aggiungendo dettagli riguardo al tipo

di giorno, come ad esempio di quale giorno della settimana si tratta e se feriale o festivo.

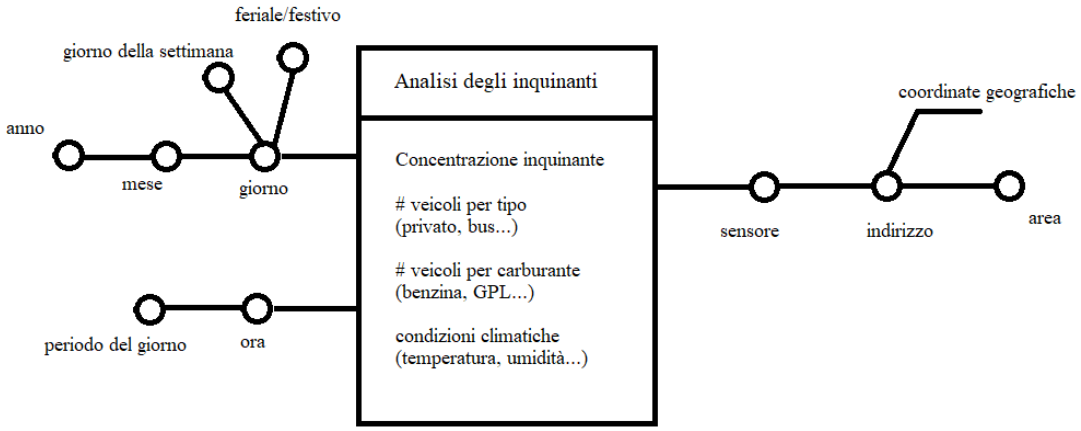

Figura 3.1 Schema concettuale del data warehouse

Terminata la fase di progettazione concettuale del data warehouse, si è potuto procedere con la fase successiva di progettazione logica. A partire dal DFM, sono state realizzate le tabelle corrispondenti seguendo lo schema a stella: per ciascuna dimensione è stata creata una tabella denormalizzata con tutte le informazioni legate alla dimensione; è stata poi aggiunta la tabella dei fatti, avente le informazioni relative alle misure del DFM.

Per rendere più efficienti le analisi, inoltre, sono state inserite e memorizzate alcune viste materializzate:

- Una prima vista raccoglie i dati delle misure aggregandole per giorno. Partendo dalle tabelle definite con la progettazione logica, viene calcolato il join in modo da avere all'interno della stessa vista tutte le informazioni riguardanti le dimensioni e in seguito il dettaglio dei risultati è stato ridotto raggruppando le informazioni sulla base della data (e calcolando la media aritmetica delle misure).
- Un'altra vista si occupa di mantenere le informazioni aggregate per ora. A differenza della vista precedente, in questo caso le misure vengono raggruppate sulla base dell'ora.

3.3 Key Performance Indicators

Una volta preparati, i dati sono pronti per essere analizzati allo scopo di ottenere informazioni circa la qualità dell'aria nelle aree urbane ed eventuali fattori correlati all'inquinamento atmosferico. A questo punto è necessario fornire all'utente feedback tramite la costruzione di alcuni scenari di analisi utilizzando *Key Performance Indicators* (KPI).

Per KPI si intende un valore numerico, un indice quantitativo tramite il quale è possibile valutare le prestazioni e la bontà di un processo o attività. Nel contesto dello sviluppo di questa applicazione, i KPI sono indici quantitativi che rappresentano e misurano la condizione dell'aria, con particolare riferimento alla quantità di inquinante presente nell'atmosfera. Per migliorare le analisi sull'inquinamento, sono stati introdotti altri KPI riguardanti traffico e meteo, volti a ricercare correlazioni tra questi ultimi e i primi.

In sintesi, i KPI introdotti per l'analisi sono stati così classificati:

1. *KPI relativi alla concentrazione di inquinanti*. In questa categoria sono stati definiti tre diversi indicatori, ognuno riguardante i valori di inquinante:
    a. Media giornaliera per sensore.
    b. Media oraria per sensore.
    c. Media giornaliera calcolata tra i vari sensori.
2. *KPI sulla criticità della concentrazione di inquinante*. Questo KPI analizza la frequenza con cui ogni sensore rileva una concentrazione media giornaliera critica di un inquinante: dato un periodo di tempo, fornisce la percentuale di giorni nei quali la media giornaliera è superiore ad un valore, detto soglia, scelto dall'utente.
3. *KPI sul clima*. Ciascuno riporta il valore medio orario di una grandezza meteorologica.

4. *KPI inerenti il traffico*. Indicano, per ciascun giorno e zona urbana, il numero totale dei veicoli entranti e un parziale, calcolato in base alle seguenti categorie:
    a. Utilizzo (trasporto pubblico, merci, privato, taxi).
    b. Tipo di carburante.

3.4 Scenari di analisi

Dopo aver definito l'insieme di KPI di riferimento, vi è la necessità di renderli disponibili all'utente finale. Perciò sono stati definiti alcuni scenari di analisi, ognuno destinato alla visualizzazione di uno o più KPI. Ciascuno scenario ha lo scopo di affrontare un'analisi mirata:

- *Concentrazione spazio-temporale di un inquinante*. Tali scenari riportano il valore dell'inquinante per ciascun sensore su un grafico. In particolare sono stati definiti due scenari in base alla granularità temporale. Il primo scenario mostra la media giornaliera di un determinato inquinante per ciascun sensore (KPI 1.a), per un periodo di dieci giorni. Il secondo scenario visualizza la media oraria in un dato giorno, sempre per ciascun sensore (KPI 1.b). In aggiunta al grafico, viene inserito su una mappa geografica un marker in corrispondenza di ogni sensore con il valore del KPI in esame. Questo scenario è utile per comprendere se alcuni periodi o aree urbane siano più soggetti all'inquinamento atmosferico rispetto ad altri. Inoltre la possibilità di visualizzare la posizione delle stazioni sulla mappa, consente di conoscere la quantità di strutture impiegate.
- *Criticità della concentrazione di un inquinante*. Un altro scenario, sempre legato alla concentrazione degli inquinanti, è quello sulla percentuale di giorni sopra soglia (KPI 2). All'interno di un periodo temporale, per ciascun sensore, viene calcolato il valore del KPI e mostrato spazialmente su una mappa, in corrispondenza del sensore che ha effettuato le rilevazioni di quell'inquinante.

- *Clima*. Questo scenario di analisi mette in relazione l'andamento giornaliero di un inquinante (KPI 1.c) con la media giornaliera di valori meteorologici, quali temperatura, umidità e pressione (KPI 3). Tale scenario dà la possibilità di analizzare se e quale fenomeno meteorologico contribuisca maggiormente a determinare la qualità dell'aria. Dal punto di vista visivo, la relazione è resa con l'accostamento di grafici, uno per l'inquinante selezionato, gli altri per le condizioni meteo.
- *Traffico*. Per quanto concerne il traffico, sono stati definiti due scenari di analisi con lo scopo di evidenziare una possibile relazione tra il trend dell'inquinante e quello del traffico. In particolare l'andamento di un inquinante (KPI 1.c) nell'arco di alcuni giorni viene confrontato con quello dei veicoli suddivisi, nel primo scenario, per tipologia di utilizzo (KPI 4.a), nel secondo, per carburante (KPI 4.b). Analizzando e confrontando questi valori è inoltre possibile determinare se e quale categoria di veicoli incida maggiormente su un inquinante.

Capitolo 4

Cruscotto

Il processo di realizzazione del cruscotto si attua attraverso alcune fasi: (i) la *scelta architetturale*, che consiste nell'individuare la struttura concettuale più adatta all'esigenza; (ii) la scelta degli *strumenti software* che meglio si prestano e rendere tale architettura e (iii) lo *sviluppo* effettivo del cruscotto finalizzato a fornire all'utente la possibilità di ricavare informazioni relative all'inquinamento urbano.

In questo capitolo vengono descritti i modelli architetturali considerati per lo sviluppo del cruscotto, in particolare l'architettura di una applicazione web a pagina singola e i modelli relazionale e NoSQL per la base di dati. Vengono anche presentate le tecnologie utilizzate per la realizzazione del cruscotto stesso. Inoltre, vengono descritte le modalità di sviluppo del cruscotto.

4.1 Architettura

4.1.1 Single Page Application

L'architettura del cruscotto realizzato in questa tesi è basata su una struttura client-server che permetta lo sviluppo di applicazioni a pagina singola (Single Page Application o SPA), che siano flessibili, facilmente riutilizzabili e ampliabili [8].

Nelle applicazioni web tradizionali, il framework lato *server* deve occuparsi di molti compiti e principalmente:

- Genera dinamicamente, durante l'esecuzione, le pagine web da inviare al client in risposta ad ogni sua richiesta.
- Gestisce la connessione al un DBMS.
- Si occupa di implementare tutta la logica di business ossia l'insieme di regole specifiche di una applicazione che determinano il comportamento della stessa. È pertanto il cuore della applicazione.

Il *client*, invece, deve dedicarsi esclusivamente al rendering del codice HTML della pagina web richiesta al server.

Questo approccio presenta alcuni svantaggi: in particolare, dal momento che ogni cambiamento dello stato passa dal server, e quest'ultimo deve occuparsi di costruire ad ogni richiesta la pagina web corrispondente, è elevato il rischio di sovraccaricare il server stesso.

Per questo motivo è stato introdotto il *modello a pagina singola*: in tale modello parte della logica di business viene spostata sul client, soprattutto per la parte relativa al controllo del flusso, sfruttando le capacità computazionali dei moderni browser e mantenendo sul server solamente la logica di accesso ai dati, la parte relativa alla sicurezza e gli aspetti più generali della business-logic. Questa scelta architetturale permette una migliore modularizzazione e divisione dei compiti tra client e server. Inoltre il server viene sollevato dall'incarico di generare le pagine web rendendolo più efficiente. Il server si comporta quindi alla stregua di un servizio REST, fornendo, tramite API, i dati prelevati dal DBMS, detti risorse (Figura 4.1).

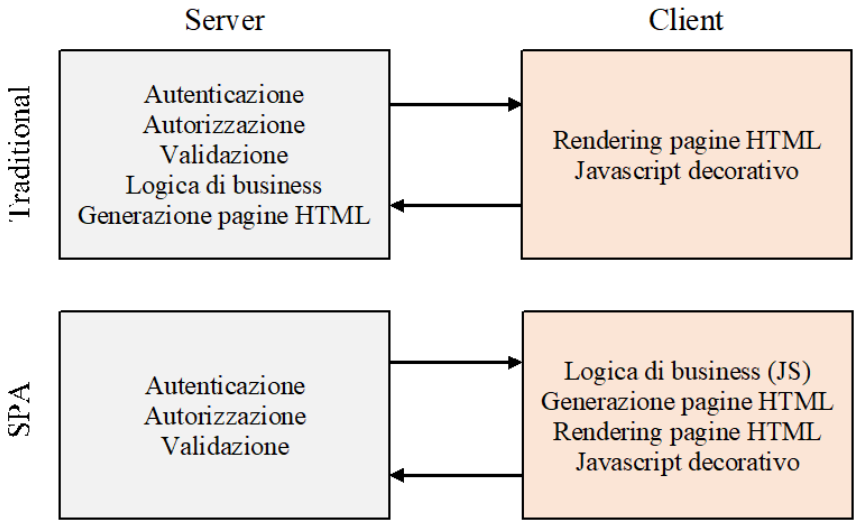

Figura 4.1 Confronto tra l'architettura tradizionale di una web application e quella di una single page application.

Il client, su richiesta dell'utente, interroga il server sfruttando tecniche AJAX. Una volta ricevuta la risorsa, il client non ricarica completamente la pagina, ma aggiorna solo la parte della vista interessata. A tal proposito la Figura 4.2 mostra un esempio di una pagina web formata da alcune componenti piuttosto comuni:

- Una barra superiore, detta *navigation bar*, utilizzata solitamente per gestire la navigazione tra le varie sezioni della applicazione web. Cambiando la sezione questa zona rimane pressoché invariata, quindi non necessita di essere ricaricata.
- Una barra sottostante, chiamata *footer*. Di solito contiene informazioni sul sito stesso e sull'azienda. Anche questa parte rimane invariata, qualunque sia la sezione del sito visitata.
- Un *menu laterale*. Di solito si tratta di una navigation bar secondaria, utile per navigare all'interno di una sezione specifica del sito. Varia a seconda della sezione e viene caricata ogni volta che si effettua l'acceso a tale sezione.

- La *parte centrale*, prevalentemente costituita dal contenuto mostrato dalla applicazione. È principalmente per questa parte che vengono sfruttate le tecniche AJAX. Navigando tra le entry del menu varia il contenuto senza bisogno di ricaricare le altre parti.

I concetti introdotti dalle SPA, tuttavia, fanno emergere alcune problematiche:

- la gestione delle richieste AJAX. Quando il client termina il download della risorsa, questa deve essere segnalata all'applicazione, la quale si deve occupare di interpretarla e aggiornare la vista.
- la necessità di simulare la navigazione tra pagine.

Dal momento che tali questioni sono indipendenti dal dominio della applicazione, esse vengono gestite da un framework javascript, con modalità diverse in base al produttore del framework.

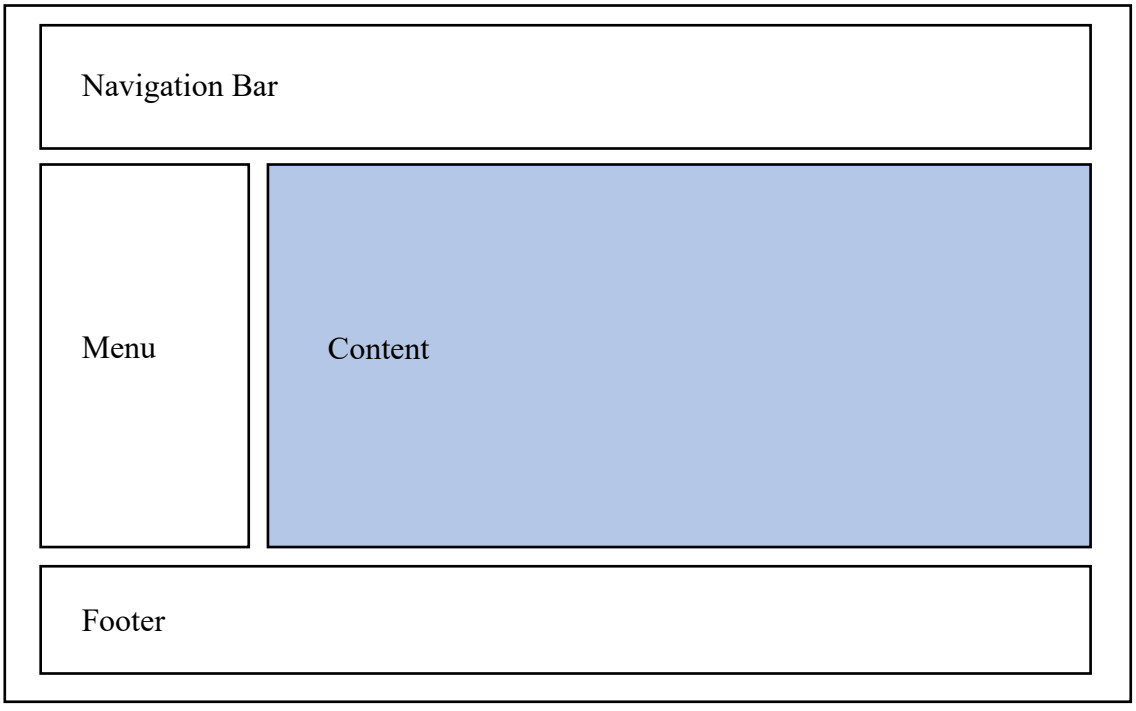

Figura 4.2 Esempio di composizione di pagina in una single page application

4.1.2 Database

Per la realizzazione del database sono stati presi in considerazione e confrontati i due modelli principali utilizzati nei sistemi moderni. Da un lato il classico modello relazionale, basato sul linguaggio SQL, dall'altro quello proposto in seguito alla nascita del movimento NoSQL con lo scopo di allontanarsi dal linguaggio SQL e superare alcune limitazioni presenti nel modello relazionale anche a costo di perderne dei benefici.

Modello Relazionale

In un database relazionale [9] i dati vengono memorizzati in tabelle, composte da varie righe e varie colonne. Ciascuna riga, detta tupla, rappresenta un dato, mentre ogni colonna identifica un tipo di attributo del dato. La struttura è pertanto fissa, ogni dato inserito nella tabella deve sottostare ad uno schema predefinito: l'elenco delle colonne e le tipologie di dato ammesse da ciascuna vengono decisi alla creazione della tabella stessa e sono immutabili. L'insieme dei dati, invece, può mutare anche rapidamente nel tempo: possono essere aggiunte nuove tuple, rimosse altre, modificate completamente o solo relativamente al valore di qualche attributo. Inoltre è indispensabile permettere la selezione di un sottoinsieme proprio o improprio dei dati in modo da fornire i risultati corretti a ricerche effettuate sugli stessi. Queste operazioni di manipolazione dei dati avvengono tramite l'utilizzo del linguaggio SQL.

SQL (Structured Query Language) è un *linguaggio dichiarativo* che opera a *livello di set*: in quanto dichiarativo, descrive *cosa* fare e non *come;* ha un livello di astrazione alto, il che lo rende indipendente dalla struttura fisica del database; inoltre gli operatori non lavorano sui singoli dati, ma sulle relazioni e forniscono sempre, come risultato, una relazione. È possibile effettuare più operazioni di seguito che risultino parte di un'unica unità logica grazie alle *transazioni*.

Ogni transazione deve garantire alcune proprietà fondamentali per un database relazionale [10]:

- *Atomicità* (*Atomicity*): una transazione è un'unità indivisibile: o tutte le operazioni svolte vengono eseguite con successo e in questo modo la transazione avrà successo, oppure tutta la transazione verrà abortita.
- *Consistenza* (*Consistency*): la transazione deve mantenere i vincoli presenti sul database.
- *Isolamento* (*Isolation*): ogni transazione deve essere eseguita in maniera indipendente dalle altre. Esistono vari livelli di isolamento definiti dallo standard ANSI/ISO per permettere maggiore flessibilità nel caso fosse necessario un aumento prestazionale.
- *Durabilità* (*Durability*): una volta terminata con successo una transazione, i risultati devono essere memorizzati sulla base di dati senza essere persi, nemmeno in caso di guasto. Questa proprietà è ottenuta con la memorizzazione di metadati sulle operazioni in una zona di memoria robusta, in grado di resistere anche a guasti. I file che contengono queste informazioni sono chiamati *file di log*.

Queste proprietà, dette comunemente ACID, tuttavia, rendono il sistema complesso, meno performante e aumentano la difficoltà a scalare orizzontalmente: affinché un database relazionale abbia prestazioni migliori, infatti, la scelta più semplice è migliorare la qualità hardware del sistema, ossia scalare verticalmente; se invece si introduce un sistema distribuito, composto da più nodi, ciascuno contenente un sottoinsieme delle tabelle, mantenere le proprietà ACID risulta più oneroso.

Prima di comprendere come queste proprietà siano implementate su un sistema distribuito, occorre identificare classi diverse di transazioni in base alla tipologia di istruzioni SQL contenute al loro interno:

1. *Remote Request*. In questa classe rientrano tutte le richieste effettuate al database che contengono solo istruzioni di lettura (solamente "SELECT") e i dati di interesse risiedono su un unico server.
2. *Remote Transaction*: qualsiasi istruzione SQL è ammessa, anche quelle che andranno a modificare lo stato del database, ma i dati risiedono su un unico server.
3. *Distributed Transaction*. All'interno di questa classe vengono inserite tutte le transazioni, effettuate in un ambiente distribuito, contenenti qualsiasi tipologia di istruzione, ciascuna delle quali si riferisce ai dati di un solo nodo. In questo caso è necessario garantire l'atomicità globale.
4. *Distributed Request*. È la classe più generale: qualsiasi comando SQL è ammesso, anche se fa riferimento a più nodi. È necessario che il database si occupi di ottimizzare la transazione sulla base della dislocazione fisica dei dati. Il client non è a conoscenza della suddivisione degli stessi.

Inoltre, con dati replicati su più nodi, è possibile che delle transazioni distribuite (classe 3 e 4) vengano effettuate in contemporanea o quasi su nodi differenti, con possibile interessamento degli stessi dati. Quindi, senza un controllo su queste evenienze, il database potrebbe andare incontro ad inconsistenze nei dati o problemi con le altre proprietà ACID. Per poterli evitare bisogna introdurre delle tecniche apposite:

- Atomicity. Per garantirla in un ambiente distribuito si utilizza la tecnica del 2 phase commit, utile per consentire a tutti i nodi coinvolti nella transazione di implementare la decisione finale corretta (commit o abort).
- Consistency. È la più difficoltosa da mantenere. I vincoli vengono applicati solo localmente.
- Isolation. Richiede due tecniche: strict 2 phase locking e 2 phase commit.
- Durability. Nel caso di ambienti distribuiti bisogna estendere le procedure locali.

La tecnica del 2 phase commit [9] ha l'obiettivo di coordinare la conclusione della transazione nell'ambiente distribuito. Viene identificato un coordinatore, detto TM (transaction manager), e alcuni partecipanti, detti RMs (resource managers), tra le varie

parti interessate dalla transazione: client e i vari nodi interpellati dalla transazione stessa. Il protocollo prevede due fasi:

1. Nella prima fase il TM invia un messaggio a ciascun RM e setta un timeout, gli RM rispondono al messaggio indicando la loro disponibilità a concludere la transazione. Nel caso il TM riceva una risposta positiva da ciascun RM, viene presa come decisione finale il global commit. Nel caso un nodo non risponda o la risposta sia negativa, il TM decide per il global abort.
2. La seconda parte del protocollo prevede che il TM comunichi ad ogni RM la decisione finale e aspetti la loro risposta. Nel caso non ne arrivino alcune, viene inviata nuovamente la decisione finale agli RM dai quali non è stata ricevuta la risposta.

Grazie a questo protocollo è possibile evitare, in presenza di errori o guasti, che una transazione venga terminata con successo su un nodo, ma non su un altro.

Un'altra tecnica utile per garantire la proprietà I (isolation) è lo strict 2 phase locking [9]. Il lock è un meccanismo adottato per bloccare l'accesso ad alcune risorse. Ne vengono utilizzati di due tipi: uno per le letture e uno per le scritture. Quando viene richiesto l'accesso per effettuare operazioni di sola lettura su una risorsa, viene inserito un "read lock" su di essa. Analogamente, per le operazioni di scrittura, viene inserito un "write lock". Una richiesta di accesso ad una risorsa protetta da un read lock può essere consentita solo se in sola lettura. Nel caso di write lock, l'accesso non sarà consentito ad altri. La tecnica del 2 phase locking prevede che tutte le richieste di lock avvengano all'inizio, in una fase detta "growing" e le risorse vengano rilasciate tutte in una seconda fase, chiamata "shrinking". In questo modo nessun lock su una risorsa può essere acquisito successivamente ad un rilascio di qualunque risorsa. Lo strict 2 phase locking aggiunge un ulteriore vincolo: il rilascio delle risorse può avvenire solo in seguito alla decisione finale (commit o abort). Questa tecnica garantisce che le transazioni concorrenti siano serializzabili e non creino anomalie nel sistema.

NoSQL

Negli ultimi anni, visto l'aumento dei dati da memorizzare ed elaborare, è sorta la necessità di migliorare le prestazioni delle basi di dati. Tuttavia, come si è visto, nel caso di database relazionali è difficile scalare orizzontalmente, in quanto si presentano problematiche legate alle proprietà ACID in caso di database distribuito. Oltretutto i database relazionali hanno alcune limitazioni intrinseche al modello.

- I database relazionali non offrono molto supporto ai linguaggi object oriented: il linguaggio SQL non è facilmente adattabile a strutture dati quali gli oggetti.
- Il modello relazionale richiede che, prima di inserire i dati, venga definita una struttura, uno schema fisso delle relazioni. Inoltre, per poter modificare tale struttura in un secondo momento, viene richiesto un lavoro oneroso al database.
- Vi è inoltre la necessità di normalizzare i dati per eliminare ridondanze ed anomalie durante le operazioni di aggiornamento del database. Questo causa una notevole riduzione delle prestazioni in operazioni di lettura qualora fosse necessario effettuare il join di più tabelle per ricostruire tutta l'informazione.

Si è pensato quindi che in alcuni casi fosse meglio abbandonare il modello relazionale in favore di un altro meno limitante e più prestazionale, anche a costo di perdere alcuni tra i requisiti fondamentali del primo. È stato pertanto introdotto un nuovo modello, non relazionale, pensato per soddisfare alcuni requisiti:

- L'inserimento e le modifiche dei dati dovrebbero poter avvenire anche senza la definizione di uno schema apriori. In tal caso modifiche anche importanti sullo schema dei dati non causerebbero ritardi o interruzione dei servizi.
- Il linguaggio SQL viene parzialmente o completamente abbandonato per lasciare spazio ad altri linguaggi ottimizzati per la tipologia di database. Vengono anche introdotte semplici API utilizzabili dal client ed indipendenti dallo specifico database, che permettano un accesso rapido ai dati, demandando il compito di analisi complesse al client stesso.

- Il sistema deve essere in grado di scalare orizzontalmente con maggior facilità, e non è necessario che il client sia a conoscenza dell'ubicazione esatta dei dati divisi tra più macchine.

Tuttavia l'utilizzo di sistemi distribuiti porta con sé alcune problematiche: a tal proposito, nel duemila venne enunciato dal Prof. Eric Brewer dell'università di Berkeley al PODC (Principle of Distributed Computing) [11] un teorema dimostrato formalmente poi nel 2002 da Seth Gilbert e Nancy Lynch del MIT [12]. Questo teorema, noto come CAP, afferma che in un ambiente distribuito è impossibile garantire contemporaneamente tre proprietà: consistenza (Consistency), disponibilità (Availability) e tolleranza di partizione (Partition tolerance): (Figura 4.3)

- *Consistenza* (C): tutti i nodi dovrebbero essere in grado di fornire gli stessi dati all'applicazione.
- *Disponibilità* (A): dopo ogni richiesta fatta, il client dovrebbe ricevere una risposta, positiva o negativa che sia. Pertanto anche in presenza di problemi ad alcuni nodi, gli altri dovrebbero essere in grado di operare.
- *Tolleranza di partizione* (P): anche in presenza di errori di rete, il sistema distribuito dovrebbe essere in grado continuare ad operare.

Tenuto conto del fatto che, in tali sistemi, non è possibile garantire le tre proprietà allo stesso tempo, ne deriva che una di esse non venga soddisfatta completamente. Se si decidesse di garantire la consistenza e la disponibilità, a discapito della tolleranza alle partizioni, nel caso di problema di rete tra i nodi, verrebbero a crearsi sistemi indipendenti multipli che garantiscono le proprietà C e A solo localmente, senza interagire: un eventuale aggiornamento ricevuto da un nodo durante la partizione, viene perso. In un sistema distribuito la tolleranza alle partizioni è quindi obbligatoria.

Nel caso in cui si decidesse di mantenere le proprietà C e P, il sistema utilizzerebbe un protocollo per tenere sincronizzati i nodi. In presenza di una partizione della rete, i nodi, non potendo comunicare tra loro, sarebbero autorizzati a non rispondere a domande

dall'esterno finché non riusciranno a sincronizzarsi. Per poter garantire la consistenza globale e tollerare le partizioni, pertanto è necessario rinunciare alla proprietà A ogniqualvolta si verifichi un problema di rete. Un esempio è il lock utilizzato nelle transazioni.

Ultimo caso, se venissero garantite le proprietà A e P, ignorando la consistenza (in particolare la consistenza globale), ogni nodo potrebbe rispondere alle richieste fornendo le informazioni in proprio possesso, anche in caso di partizioni: nel caso di problemi di rete, i nodi non riceverebbero gli aggiornamenti e restituirebbero informazioni non consistenti tra loro.

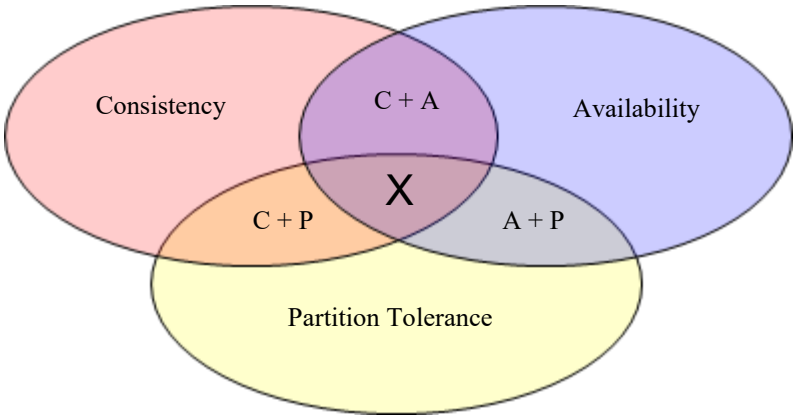

Figura 4.3 Teorema CAP

Dal momento che i sistemi distribuiti non possono fare a meno della tolleranza alle partizioni e fornire risposte immediate ha assunto un carattere dominante, molti sistemi NoSQL vengono realizzati cercando di soddisfare le proprietà A e P, diminuendo il livello di consistenza. Infatti, all'interno di un sistema distribuito la consistenza può essere più o meno forte. La scelta del livello di tale proprietà va ad incidere notevolmente sulle prestazioni del sistema stesso:

- *Eventual Consistency*. Si tratta del modello di consistenza più debole. Viene garantito soltanto che, prima o poi, tutte le repliche avranno tutti i dati aggiornati.

- *Session Consistency*. La consistenza è garantita a livello di sessione: l'ordine delle operazioni effettuate in una sessione viene mantenuta sicuramente.
- *Causal Consistency*. In questo modo vengono garantite le operazioni sul database in ordine 'causale': le operazioni effettuate in risposta ad un'altra vedranno tale ordine mantenuto su tutte le repliche. Nel caso di operazioni non correlate, non viene garantito l'ordine temporale tra le varie repliche.
- *Serializability*. È il modello più restrittivo e meno performante. Garantisce che su ogni replica venga rispettato un ordine globale sulle operazioni. Ogni lettura effettuata, su qualsiasi replica, restituisce il dato aggiornato. Ciò corrisponde all'impedimento di scritture concorrenti su repliche differenti.

Brewer pertanto proposte le proprietà BASE in contrapposizione a quelle ACID [13]. Mentre queste ultime si focalizzano sulla consistenza, in particolare sul modello più rigido, le proprietà BASE danno maggior enfasi alla disponibilità anche in presenza di partizioni:

- *Basic Available*. Il sistema deve garantire principalmente la disponibilità, com'è intesa dal teorema di Brewer.
- *Soft State*. Questa proprietà indica che lo stato del sistema può evolvere nel tempo, anche senza interventi esterni: questo a causa del modello di consistenza scelto.
- *Eventually Consistent*. Viene scelto il modello di consistenza più lasco: viene garantito che i nodi avranno i dati aggiornati solo dopo un certo intervallo di tempo senza aver ricevuto input.

Diverse sono le tipologie di database non-relazionali esistenti. In base alle modalità con cui vengono memorizzati i dati, si possono classificare in quattro categorie:

1. *Key-Value*. Sono database nei quali i dati vengono inseriti come coppia chiave-valore. La ricerca per chiave è molto efficiente mentre quella per valore molto difficile. Sono rapidi e scalano facilmente.

2. *Column-Oriented*. I dati vengono memorizzati per colonne in modo da favorire il calcolo di aggregati. Ogni colonna può essere un attributo anche complesso. Rispetto ai database relazionali sono meno adatti a transazioni quali lettura, aggiornamento e rimozione di un singolo record.
3. *Graph*. Sono basati sulla teoria dei grafi e sui concetti di nodi e archi. Sono molto utili per rappresentare le relazioni, le informazioni riguardanti reti e gestire dati mutevoli che presentano schemi in continua evoluzione. Risultano particolarmente adatti ad interagire con linguaggi orientati agli oggetti: i nodi rappresentano gli oggetti, mentre gli archi le relazioni tra gli stessi.
4. *Document*. Il database ha una struttura ad albero. Ogni record è memorizzato come documento, identificato da una chiave. All'interno di ciascun documento vi possono essere più coppie chiave-valore o documenti innestati: è possibile aggiungere campi di qualsiasi tipo e lunghezza al documento, senza bisogno di definire uno schema. Questi sistemi permettono, oltre a semplici ricerche per chiave, anche di recuperare i documenti in base ai loro campi attraverso la chiamata di API.

4.2 Scelte tecnologiche

4.2.1 Client

Per sviluppare la parte relativa al client si è scelto di utilizzare il framework AngularJS in quanto è in grado di fornire gli strumenti necessari alla costruzione di una applicazione web che sia modulare e soddisfi i principi delle applicazioni a pagina singola.

AngularJS

AngularJS [14] è stato sviluppato da Google per gestire le problematiche nate con l'introduzione delle SPA. È basato su una variante del pattern Model-View-Controller (MVC), ossia nella divisione dei compiti tra i vari componenti:

- Il *modello* (Model) fornisce i metodi per accedere ai dati utili all'applicazione.
- La *vista* (View) visualizza i dati del modello e gestisce l'interazione con l'utente
- Il *Controller* modifica lo stato dei componenti in risposta ad un evento.

Questa variante è chiamata MVVM (Model-View-View Model), nella quale il componente "View Model", oltre a svolgere il compito di controller, si occupa di mantenere un collegamento diretto tra la vista e il modello tramite un "Binder", che ha la funzione di sincronizzare gli altri due componenti.

Altre funzioni che caratterizzano AngularJS sono il "*Two-Way Data Binding*" e la "*Dependency Injection*".

A differenza della maggior parte dei sistemi di templating, in cui i dati sono legati in una sola direzione (One-Way Data Binding) e modello e template vengono combinati a generare la vista, AngularJS fornisce un meccanismo tramite cui, in tempo reale, ogni modifica effettuata sulla view si riflette sul modello e ogni modifica nel model sulla vista, come descritto dalla Figura 4.4.

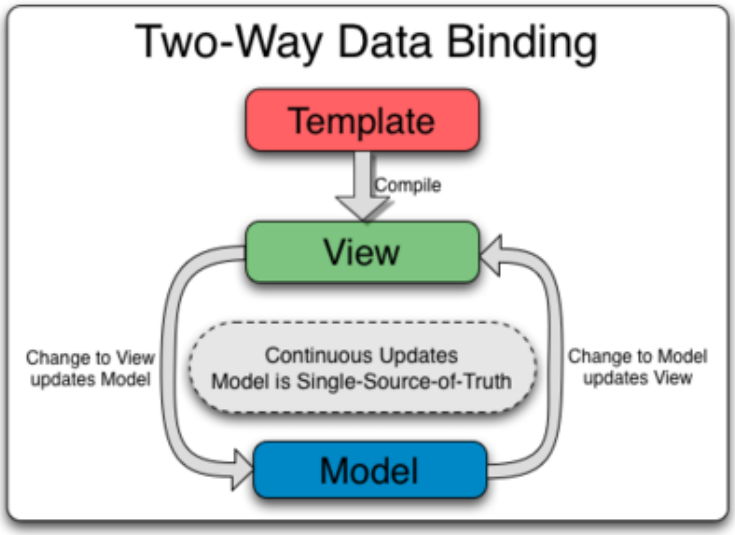

Figura 4.4 Two-Way Data Binding [15]

"Dependency Injection" è un pattern tramite il quale è possibile, all'interno di un modulo, dichiarare la necessità di utilizzare funzionalità offerte da altri moduli che verranno automaticamente richiamati quando necessario.

Il framework, per garantire queste funzionalità, introduce alcuni componenti (Figura 4.5), ciascuno con uno specifico compito:

- *View*. Ciò che viene visualizzato all'interno del browser a partire da un template HTML eventualmente arricchito con attributi personalizzato che viene elaborato da AngularJS.
- *Controller*. È una funzione javascript che si occupa di aggiungere funzionalità allo scope di una view, introducendo una serie di metodi e proprietà reperibile quando necessario dalla view stessa attraverso un oggetto chiamato "$scope".
- *Direttiva*. Si tratta di un componente volto ad estendere gli elementi HTML tramite attributi personalizzati. Quando il compiler di AngularJS effettua il parsing del template, associa ad ogni elemento personalizzato il comportamento specificato dalla direttiva corrispondente. Ne esistono alcune predefinite che

consentono di effettuare le operazioni più comuni, ma è possibile introdurne di nuove.

- *Servizio*. È un oggetto javascript istanziato una volta soltanto alla creazione della applicazione con il compito di gestire funzionalità indipendenti dall'interfaccia grafica e renderle disponibili agli altri componenti. I servizi vengono molto frequentemente utilizzati come interfacce per accedere ad un server con richieste AJAX.

Durante la navigazione di una pagina HTML l'applicazione può assumere vari stati, i quali vengono mappati in maniera univoca su diversi URL e possono corrispondere a più viste, ognuna legata ad un controller tramite lo $scope. L'applicazione è in grado di discernere quale view rappresentare grazie al routing: quando viene richiesto un particolare URL, l'applicazione intercetta la richiesta, controlla se questo path è associato ad una view e nel caso affermativo elabora il template, istanzia il controller associato e ricarica la parte della pagina relativa alla vista.

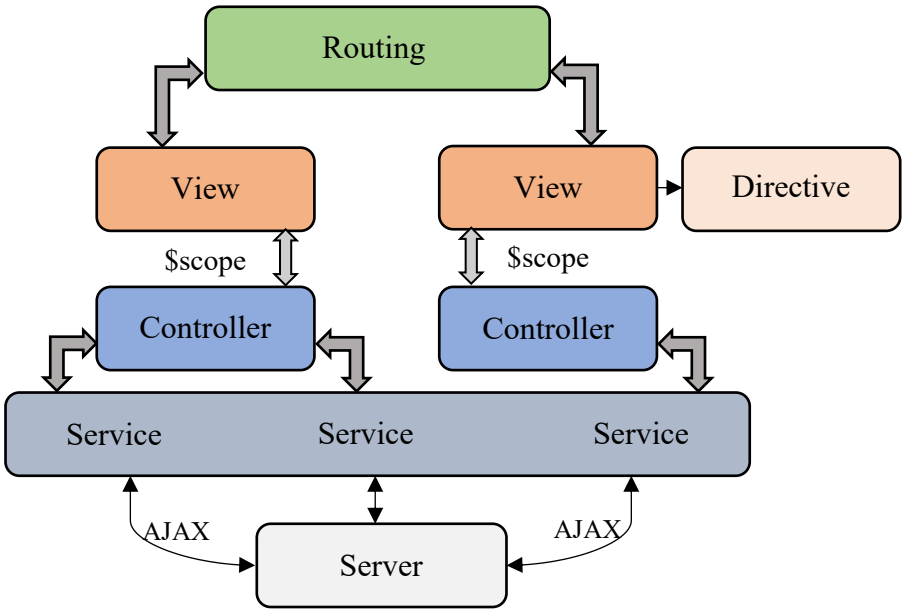

Figura 4.5 I componenti AngularJS

All'interno del cruscotto sviluppato è stata data molta enfasi all'utilizzo delle direttive. Dal momento che per la visualizzazione dei dati vengono utilizzati strumenti differenti quali mappe, grafici e menu, per ciascuno di questi elementi è stata introdotta una direttiva custom che si occupi di implementare il comportamento specifico, in modo da mantenere la SPA il più modulare possibile. Per poter creare una direttiva, è necessario fornire un nome e una funzione che ritorni un oggetto chiamato DDO (directive definition object) per definirne le impostazioni. Tale oggetto può contenere alcune proprietà, ciascuna con il proprio fine, ad esempio "restrict" serve per indicare in che parte del codice HTML la direttiva potrà essere usata, "template" o "templateUrl" descrive il template che deve sostituire o essere inserito all'interno della parte di codice HTML in cui è stata utilizzata la direttiva. Oltre a queste proprietà, ve ne sono altre che sono state utilizzate nello sviluppo del cruscotto, in particolare:

- "*scope*". Di default la direttiva eredita lo scope dal componente padre ossia il controller la cui vista associata utilizza la direttiva. Tuttavia in questo modo la direttiva risulta poco flessibile e riutilizzabile poiché rimane un legame stretto con il controller stesso. È quindi possibile modificare tale comportamento o creando un nuovo scope, dal quale è comunque possibile accedere a quello del componente padre, oppure generando uno scope isolato, ossia legato esclusivamente alla direttiva ed utilizzabile solo all'interno di questa e del template. Allo scope isolato è possibile mappare alcuni dati provenienti dall'esterno. In questo modo vengono costruite direttive riutilizzabili in ogni contesto e con dati differenti.
- "*link*". A questa proprietà può essere associata una funzione che verrà eseguita prima del rendering del template della direttiva. Tale funzione prevede dei parametri: lo scope della direttiva, l'elemento del DOM a cui è associata e l'elenco degli attributi. Con questa proprietà, quindi, è possibile fornire un metodo per eventuali inizializzazioni e definire comportamenti che non sarebbero attuabili con il solo template.

Librerie di supporto

In aggiunta, sono state utilizzate alcune librerie javascript open source, per la gestione di alcuni aspetti della applicazione web. Nel dettaglio, quelle più interessanti, in virtù delle modalità di analisi da proporre all'utente, sono le seguenti:

- Lealflet [16]. Una libreria che permette l'utilizzo di una carta geografica interattiva e facilmente ampliabile. La mappa è costituita da più livelli sovrapposti, ognuno dei quali rappresenta oggetti differenti. Ad esempio, nel cruscotto è stato implementato un livello per la gestione e visualizzazione di marker.
- C3.js [17]. Uno strumento per la creazione e customizzazione di grafici basato su D3.js (libreria sviluppata per la manipolazione dei documenti basata sui dati, con enfasi sugli standard web).
- Bootstrap [18]. Utilizzata per migliorare l'esperienza utente, fornisce una libreria per semplificare la gestione dell'aspetto grafico attraverso l'utilizzo di classi CSS definite dalla libreria.

4.2.2 Server

Il server è stato sviluppato sfruttando la piattaforma Node.js. Si tratta di un framework javascript basato su modello di I/O non bloccante e ad eventi, caratteristica che lo rende più efficiente in situazioni di elevato traffico di rete.

Node.js

Node.js [19] consente la creazione di server Web e strumenti di rete utilizzando JavaScript e una raccolta di "moduli" che gestiscono varie funzionalità di base, come ad esempio i

protocolli di rete, l'accesso al file system e altre ancora. È possibile aggiungere nuovi moduli, anche creati dal programmatore stesso, in modo da ridurre la complessità di scrittura e manutenzione del server. A differenza di molti altri strumenti per lo sviluppo di web server, come ad esempio il linguaggio PHP, nei quali la maggior parte delle funzioni in PHP bloccano fino al completamento (i comandi vengono eseguiti solo dopo aver completato quelli precedenti), in Node.js le funzioni sono progettate per essere non-bloccanti, per poter essere eseguite in parallelo e per segnalare il loro completamento o fallimento tramite callback. Inoltre, Node.js sfrutta il paradigma della programmazione ad eventi: invece di attendere che sia un'istruzione ad impartire il comando di svolgere un compito o elaborare un'informazione, come avviene nella programmazione procedurale, il sistema esegue continuativamente una serie di istruzioni, alcune delle quali si occupano di controllare la comparsa di un'informazione (ossia un evento) e in tal caso lanciare l'esecuzione della procedura che si occuperà di gestire l'evento.

Moduli di supporto

In aggiunta ai moduli principali di Node.js e a quelli per il supporto alla gestione degli aspetti legati alla creazione di un web server, ci si è serviti di alcune librerie per consentire l'interazione del server con la base dati. Tali librerie si occupano di generare e mantenere una connessione con il database, tradurre le richieste effettuate dal server nel linguaggio SQL o in quello specifico della base di dati utilizzata, ricevere ed organizzare i risultati della query e renderli disponibili in maniera semplice (ad esempio attraverso il formato JSON) al server.

4.2.3 Database

In seguito al confronto effettuato tra i due modelli di database, si è deciso di utilizzare un prodotto che facesse uso di un schema non relazionale. Tale scelta è avvenuta in particolare per garantire una maggiore flessibilità. Questa scelta permette infatti di supportare future evoluzioni del cruscotto. Ad esempio si potrebbe in futuro valutare di introdurre nuovi dati, alcuni dei quali potrebbero avere una struttura differente. La scelta di un database NoSQL con struttura a documenti permette di inserire nuove informazioni senza il rischio di dover modificare i dati inseriti precedentemente. Inoltre, nel caso si debbano gestire grandi moli di dati il database NoSQL può ridurre i costi di interrogazione.

MongoDB

MongoDB [20] è un DBMS non relazionale che salva i dati all'interno di documenti in maniera flessibile, utilizzando un formato simile al JSON, il BSON. Con l'acronimo BSON, si intende JSON binario, si tratta di un formato di scambio dati progettato per essere più efficiente sia in quanto a velocità di lettura, sia per lo spazio utilizzato per l'archiviazione dei dati [21].

I componenti principali di MongoDB sono i seguenti:

- *Database*: contiene un insieme di strutture dati dette collezioni. Ne possono essere presenti più di uno per istanza di MongoDB server.
- *Collezione*: è l'equivalente di una tabella di un database relazionale. Contiene più documenti correlati tra loro, tuttavia, a differenza di una tabella, non forza lo schema dei dati.
- *Documento*: può essere considerato l'equivalente di una tupla di una tabella nel modello relazionale. Contiene le informazioni su un dato con una struttura BSON,

ogni campo può assumere qualunque valore ammesso da tale formato. Ogni documento è identificato univocamente all'interno della collezione da una chiave.

La flessibilità di MongoDB permette che i campi possano variare da documento a documento e la struttura stessa dei dati possa cambiare nel tempo. Inoltre, MongoDB offre un supporto alle applicazioni consentendo di mappare un documento ad un oggetto. Per poter rappresentare le relazioni, possono essere intraprese due scelte [22]:

1. *References*: questa tecnica è la più simile a quella dei database relazionali. All'interno di un documento vi è un collegamento ad un altro documento, tramite la chiave del secondo. Per poter costruire tutta l'informazione è necessario accedere ad entrambi i documenti. I riferimenti vengono generalmente utilizzati per mappare le relazioni molti a molti (M:N), o nel caso di grandi insiemi di dati organizzati gerarchicamente. In questo caso i dati tendono ad essere normalizzati.
2. *Embedded Document* (*sotto-documenti*): la relazione è espressa all'interno del documento stesso, il quale memorizza tutti i dati correlati. Per avere l'informazione completa è sufficiente accedere ad un unico documento. In questo modo i dati vengono de-normalizzati, le stesse informazioni possono essere ripetute in più documenti. Si utilizza per mappare relazioni uno ad uno (1:1) e uno a molti (1:N).

MongoDB presenta un'architettura distribuita che permette di scalare orizzontalmente attraverso lo *sharding*. Con questo termine si intende un metodo per distribuire i dati tra più macchine. Un cluster MongoDB è composto da *shard*, *mongos* e *congif servers*: [23]

- *Shard*: ciascuno shard contiene una parte dei dati, che possono essere replicati su più shards. Tali repliche sono utili in caso di malfunzionamenti.
- *Mongos*: si comporta come router, interfacciandosi con un client e indirizzando la query ricevuta da quest'ultimo allo shard opportuno utilizzando metadati. Possono essercene più di uno per ridurre il carico di lavoro.

- *Config server*: ne sono presenti minimo tre, per ridondanza. Si occupano di memorizzare le impostazioni di configurazione e i metadati riguardanti il cluster e la distribuzione dei dati all'interno dello stesso.

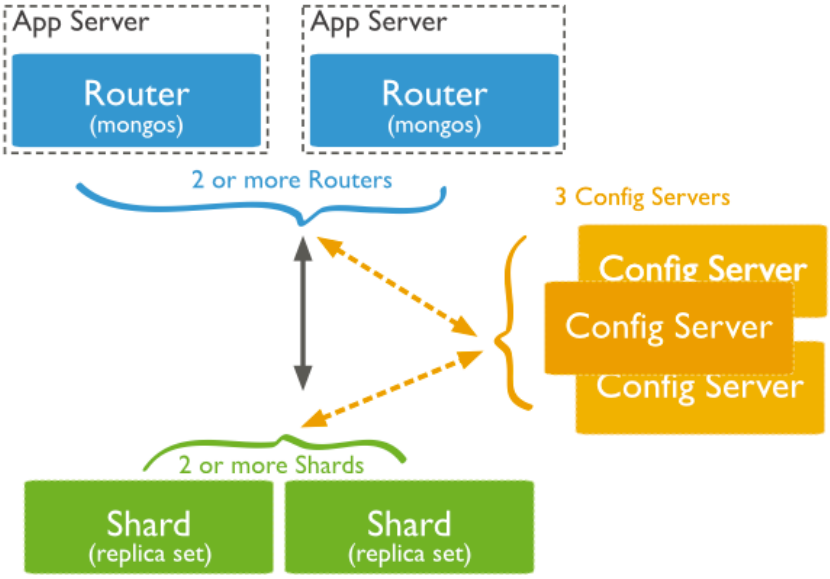

Figura 4.6 L'interazione dei componenti in un cluster MongoDB [24]

L'accesso ai dati e le loro modifiche sono possibili attraverso delle query che possono essere costruite ad hoc per ricercare campi specifici nei documenti, per selezionare range tra i valori di un campo e supportano persino le espressioni regolari. Inoltre, per migliorare le prestazioni di ricerca, è possibile definire degli indici su qualunque campo: questi memorizzano, in ordine di valore, il campo o i campi da indicizzare consentendo così operazioni rapide su tali attributi [23].

Oltre alle classiche CRUD (create o insert, read, update e delete), MongoDB supporta alcune operazioni di aggregazione, ossia raggruppa i dati di più documenti di una collezione, li processa e restituisce come risultato una nuova collezione. Esistono tre metodi per effettuare queste operazioni [23]:

- *Aggregation Pipeline*. Si tratta di un framework modellato sul concetto di *data processing pipeline*: la pipeline è composta da più fasi, dette stages, ciascuna delle quali trasforma i documenti in risultati aggregati. Per ogni documento in ingresso, lo stage può produrre zero o più documenti in uscita. Possono esserci fasi di filtraggio, riordino, raggruppamento e altre ancora.
- *Map-Reduce*. Questa operazione è composta da due parti:
    1. *Map*: ogni documento viene processato da una funzione che emette uno o più oggetti.
    2. *Reduce*: questa funzione prende come input l'output di quella di *map* ed effettua alcune operazioni su tali oggetti e li unisce.

  Le funzioni di *map* e *reduce* devono essere implementate dall'utente e scritte in javascript. Questo metodo, pur assicurando maggior flessibilità rispetto all'*Aggregation Pipeline*, ha prestazioni inferiori e presenta una complessità maggiore.
- *Single Purpose Aggregation Operations*. MongoDB offre supporto integrato per alcune tra le funzioni di aggregazione più utilizzate, come *count*() e *distinct*(). Queste operazioni sono ottimizzate, tuttavia hanno bassa flessibilità e vengono utilizzate solo nel caso di aggregazioni semplici.

4.3 Sviluppo

Il cruscotto è stato sviluppato, come descritto in precedenza, con un approccio client-server, seguendo i paradigmi della SPA e utilizzando i framework AngularJS e Node.js. A questa architettura è stato affiancato il database MongoDB per la gestione dei dati utilizzati.

4.3.1 Client

Quando la SPA viene caricata sul client, esso riceve anche un file di configurazione contenente alcune informazioni per rendere gli scenari.

L'aspetto grafico che si presenta all'utente è mostrato dalla Figura 4.7 e include le seguenti componenti:

- Sullo sfondo viene caricata una *mappa geografica*. Sulla mappa compaiono inoltre alcuni markers e alcune linee che delimitano le varie zone della città. Su ogni marker compare, se lo scenario lo prevede, un valore e un colore.
- Sulla sinistra è presente un *menu* che permette di passare da uno scenario all'altro.
- A destra compare un pannello contenente una *legenda* con alcune informazioni sui colori assunti dai marker sulla mappa.
- Nella parte superiore è presente una barra che permette di *affinare i criteri di analisi*. In particolare è possibile effettuare la scelta della data, o del periodo, dell'inquinante e, se disponibile, di un valore soglia.
- In basso sulla destra vi è un pannello che si occupa di visualizzare alcuni *grafici* inerenti allo scenario selezionato.

Le operazioni che l'utente può effettuare e che causano richieste di dati al server sono le seguenti:

- Modificare la data, o il periodo di tempo, per cui si vogliono visualizzare le informazioni.
- Scegliere un inquinante tra quelli disponibili.
- Inserire un valore che identificherà la soglia di riferimento per l'inquinante.
- Cambiare lo scenario di analisi selezionando un elemento tra quelli presenti nel menu.

Molte delle componenti precedentemente elencate, coordinante da un controller che mantiene i dati necessari, sono state sviluppate con direttive AngularJS. In particolare:

1. *mapDirective*: si occupa di istanziare la mappa geografica, costruire i bordi delle zone della città e i markers. Ogni volta che viene selezionata una entry del menu, oppure viene modificato un parametro tramite la barra superiore, ricrea ogni marker con il valore e il colore corretti.
2. *menuDirective*: gestisce il menu. Quando l'utente seleziona una entry genera un evento per consentire agli altri componenti di reagire.
3. *legendDirective*: visualizza i valori di riferimento presenti nella legenda e li modifica al variare dello scenario, dell'inquinante o del valore soglia.
4. *chartDirective*: questa direttiva si occupa di visualizzare i dati tramite un grafico. Quando vengono modificati il periodo, l'inquinante o lo scenario ricrea il grafico utilizzando i valori corrispondenti.
5. *dateDirective*: permette all'utente di scegliere il periodo di interesse e comunica la scelta alle direttive interessate.
6. *pollutantDirective*: per ogni scenario mostra l'elenco degli inquinanti selezionabili. Ogni volta che l'utente ne sceglie uno, lo comunica alle altre componenti.

7. *thresholdDirective*: gestisce il campo della soglia. L'utente può inserire un qualunque valore numerico che verrà utilizzato come soglia per l'analisi corrispondente.

Quando l'utente effettua un'operazione con lo scopo di modificare o affinare i criteri di analisi, il client costruisce una richiesta http con i parametri selezionati e la invia al server. Una volta ricevuti i dati in risposta dal server, il client li rende visibili all'utente, sostituendo i grafici presenti in precedenza e ricreando i markers presenti sulla mappa tramite le direttive descritte.

L'utente, inoltre, ha la possibilità di visualizzare alcune informazioni aggiuntive riguardo alle stazioni di monitoraggio. Cliccando su un marker presente sulla mappa, viene aperto un popup contenente i dati relativi alla corrispondente stazione.

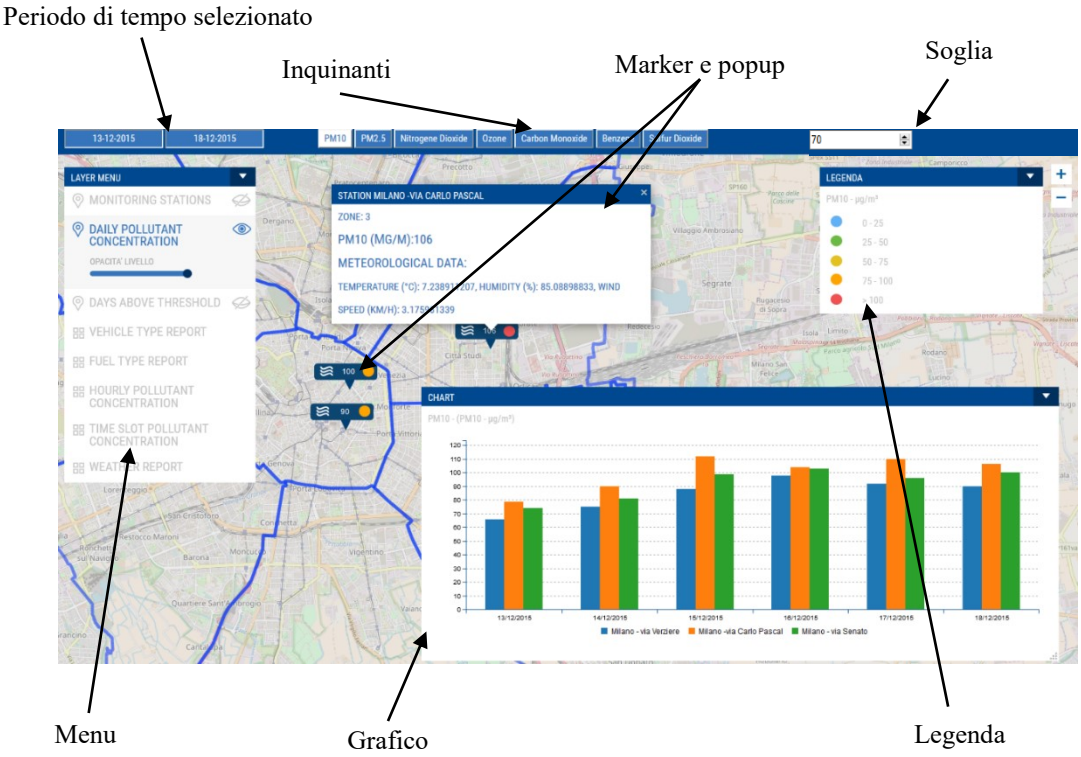

Figura 4.7 Cruscotto per il monitoraggio dell'inquinamento urbano

Gli scenari attualmente implementati sul client, dettagliati in seguito, sono: (i) *Monitoring Stations*, (ii) *Daily Pollutant Concentration*, (iii) *Days Above Threshold*, (iv) *Vehicle Type Report*, (v) *Fuel Type Report*, (vi) *Hourly Pollutant Concentration*, (vii) *Time Slot Pollutant Concentration*, (viii) *Weather Report*.

Monitoring Stations (Figura 4.7.1): vengono visualizzate sulla mappa geografica tutte le stazioni che compongono la rete di monitoraggio degli inquinanti. Per ciascuna stazione è possibile, grazie ad un popup, visualizzare l'elenco degli inquinanti rilevati da ciascun sensore presente nella stazione stessa. È indicata, inoltre, la zona della città in cui si trova la stazione.

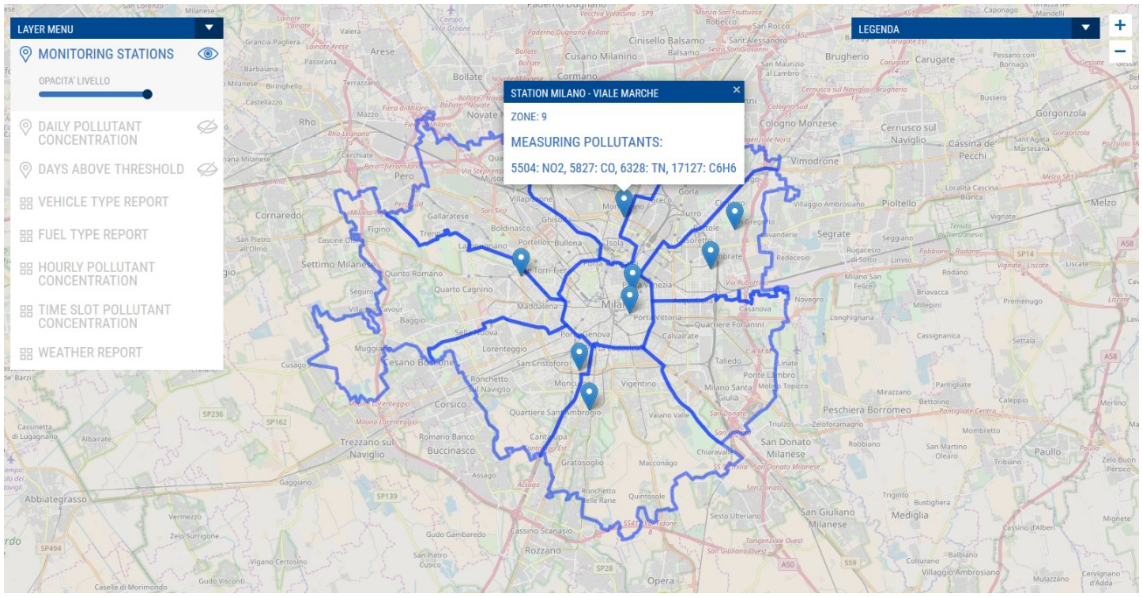

Figura 4.7.1 Scenario di analisi "Monitoring Stations"

Daily Pollutant Concentration (Figura 4.7.2): questa visualizzazione permette all'utente di osservare l'andamento di un inquinante nell'arco di qualche giorno. Ciò viene reso attraverso un grafico (presente nell'apposito pannello) che riporta in ascissa la data di misurazione e in ordinata il valore medio dell'inquinante, per ciascun sensore. Vengono inoltre visualizzati marker sulla mappa, uno per ciascuna stazione in grado di monitorare l'inquinante in esame. È possibile modificare il periodo di giorni da visualizzare e tipo di inquinante tramite la barra posta in alto.

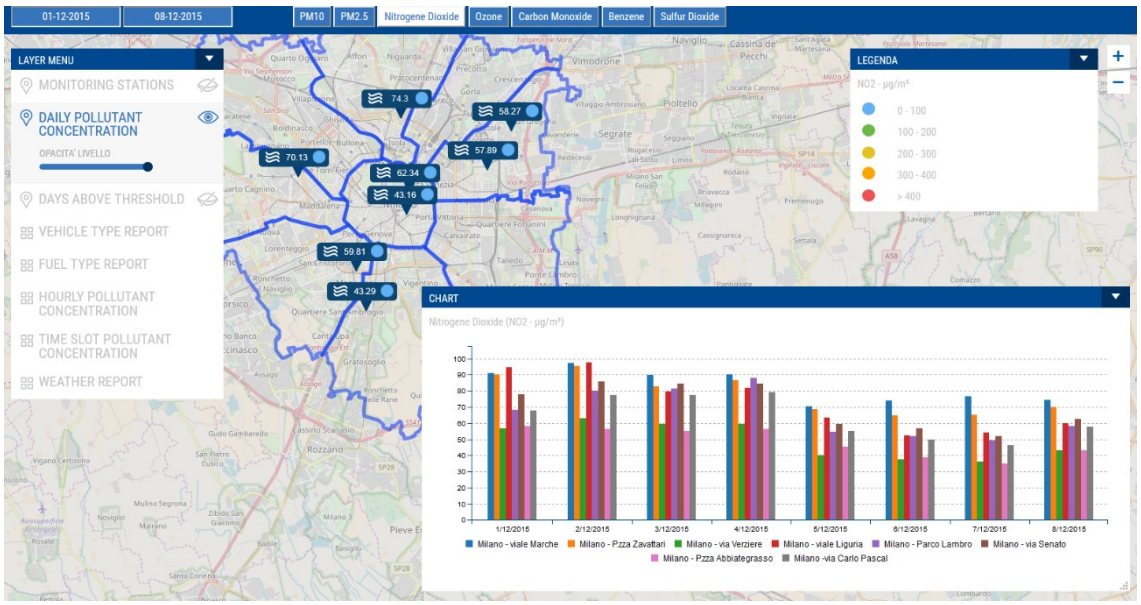

Figura 4.7.2 Scenario di analisi "Daily Pollutant Concentration"

Days Above Threshold (Figura 4.7.3): questa schermata permette all'utente di comprendere se in un periodo temporale, grande a piacere, i valori di un inquinante sono stati più o meno critici. La criticità è data dalla percentuale di giorni nei quali la concentrazione dell'inquinante è superiore alla soglia. Quest'ultima può essere modificata dall'utente, tramite un campo posto in alto sulla destra. Sulla mappa viene inserito un marker per ogni sensore in grado di rilevare l'inquinante selezionato. Il valore indicato sul marker rappresenta la percentuale di giorni, all'interno del periodo scelto, nei quali l'inquinante ha avuto una media giornaliera superiore alla soglia. Inoltre, su ogni marker l'indicazione della criticità è data anche dal colore, che spazia dal verde al rosso, mano a mano che la percentuale aumenta.

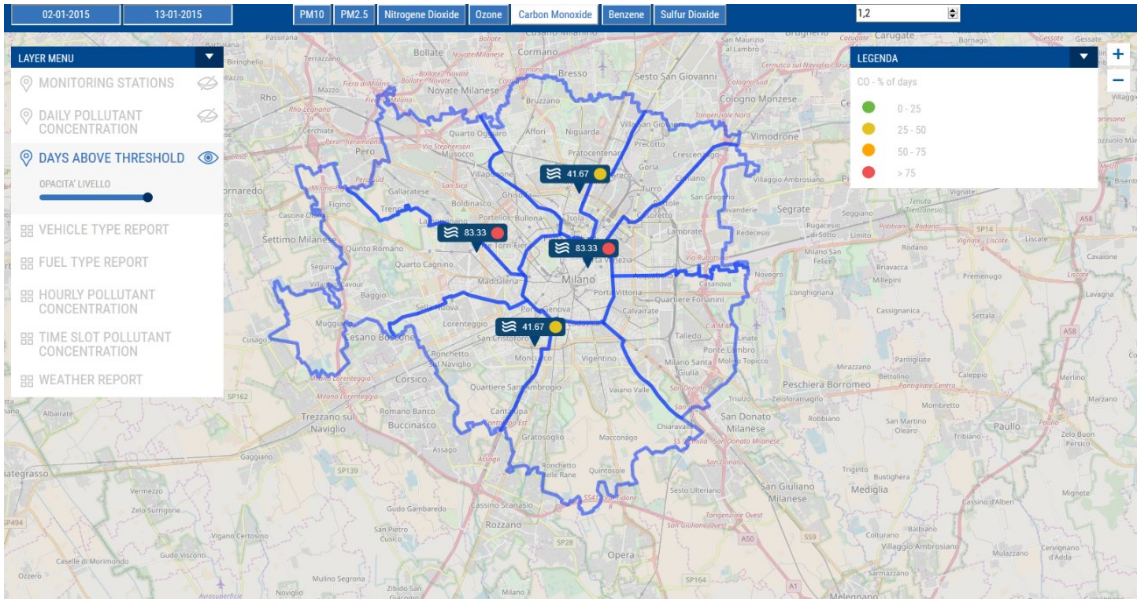

Figura 4.7.3 Scenario di analisi "Days Above Threshold"

Vehicle Type Report (Figura 4.7.4): questo scenario offre il confronto tra due grafici di cui il primo riporta la quantità giornaliera di veicoli transitati, divisi per tipologia di utilizzo (trasporto pubblico, merci, privato, taxi), il secondo il valore medio (tra tutti i sensori) di un inquinante nell'arco di qualche giorno. È possibile scegliere il tipo di inquinante tramite la barra posta in alto. Inoltre, grazie ai due campi in alto a sinistra vi è la possibilità di selezionare il periodo per il quale osservare l'andamento dei valori riportati dai grafici.

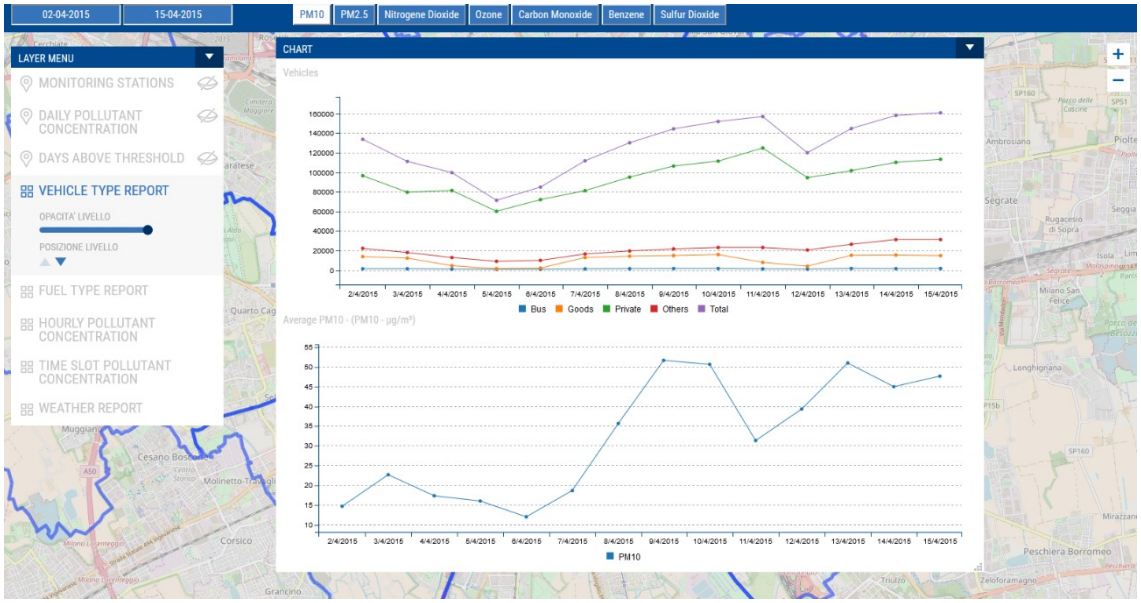

Figura 4.7.4 Scenario di analisi "Vehicle Type Report"

Fuel Type Report (Figura 4.7.5): questo scenario offre il confronto tra due grafici di cui il primo riporta la quantità giornaliera di veicoli transitati, divisi per tipologia di carburante, il secondo il valore medio (tra tutti i sensori) di un inquinante nell'arco di qualche giorno. È possibile modificare il periodo e il tipo di inquinante tramite la barra posta in alto. Le tipologie di carburante riportate sul grafico sono: benzina, gasolio, elettrico, GPL e ibrido. A questi viene aggiunto il valore totale.

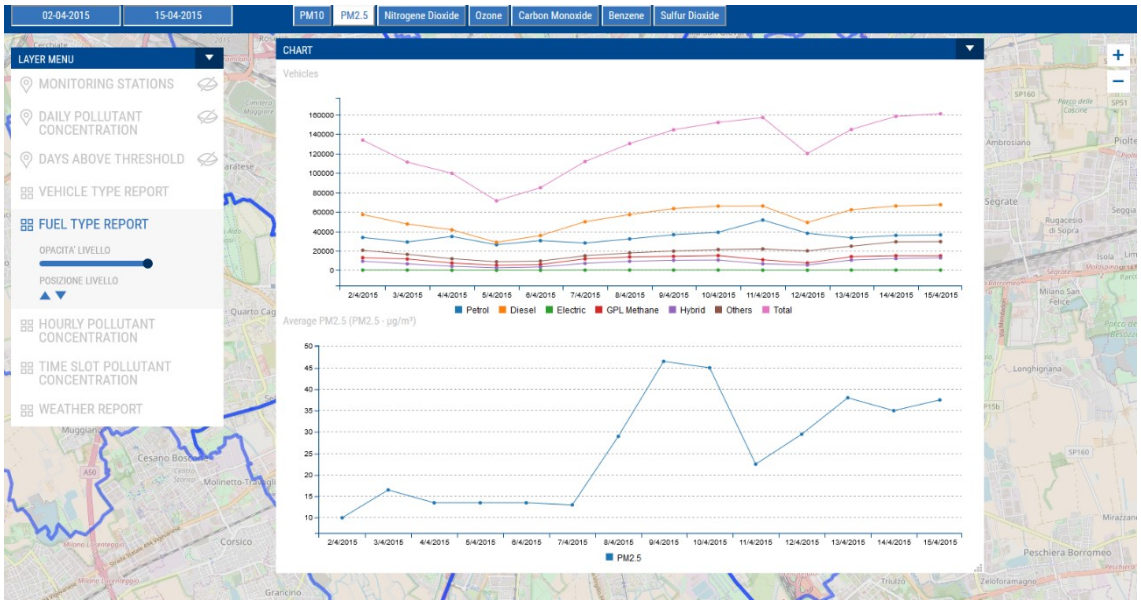

Figura 4.7.5 Scenario di analisi "Fuel Type Report"

Hourly Pollutant Concentration (Figura 4.7.6): questa visualizzazione permette all'utente di osservare l'andamento orario di un inquinante in un dato giorno. Ciò viene reso attraverso un grafico (presente nell'apposito pannello) che riporta in ascissa l'ora di misurazione e in ordinata il valore medio dell'inquinante, per ciascun sensore. È possibile modificare la selezione del giorno e del tipo di inquinante tramite la barra posta in alto.

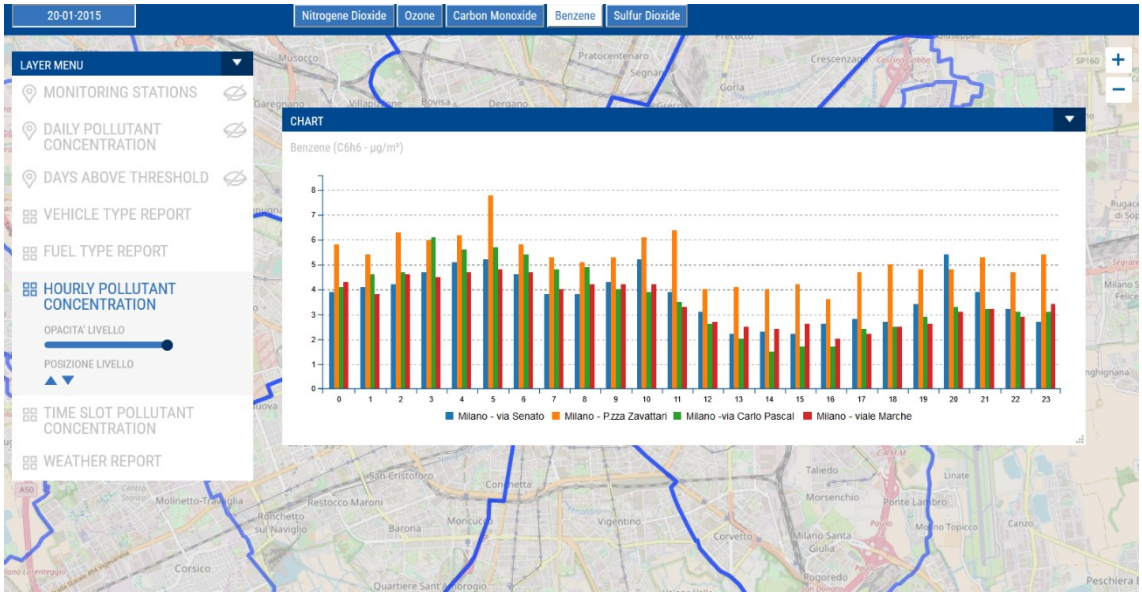

Figura 4.7.6 Scenario di analisi "Hourly Pollutant Concentration"

Time Slot Pollutant Concentration (Figura 4.7.7): questo scenario mette in mostra, per ciascun inquinante, la sua concentrazione per ciascun sensore. Questa viene ottenuta a partire delle informazioni sull'andamento medio orario, calcolando la media per ogni parte del giorno: mattino, pomeriggio, sera e notte. È possibile modificare giorno e inquinante grazie alla barra posta in alto.

Figura 4.7.7 Scenario di analisi "Time Slot Pollutant Concentration"

Weather Report (Figura 4.7.8): in questo scenario viene messo a confronto un grafico che mostra l'andamento orario di un inquinante in un dato giorno per sensore con più grafici ciascuno riportante la variazione oraria di un fattore meteorologico (temperatura, umidità, velocità del vento e pressione atmosferica).

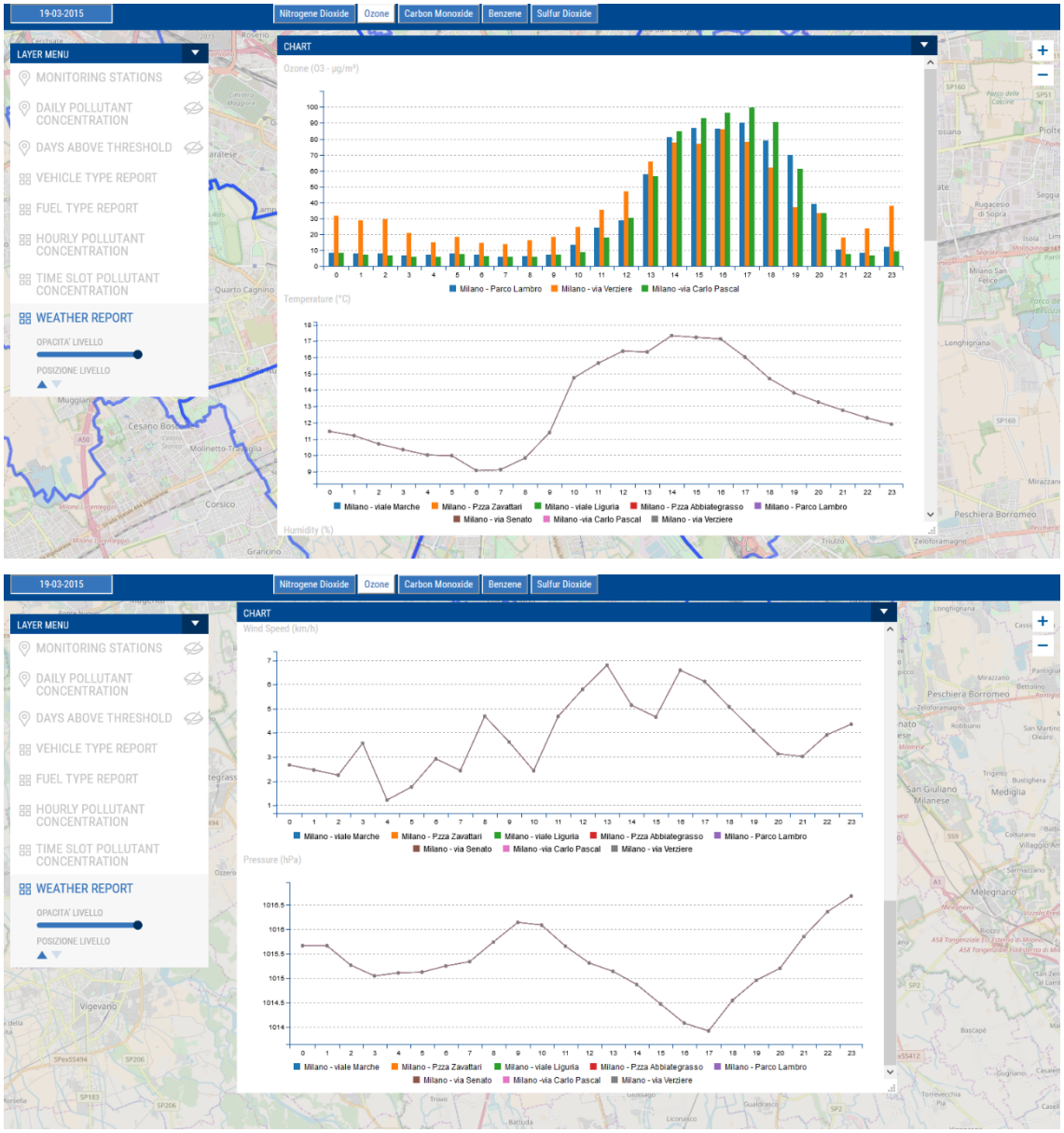

Figura 4.7.8 Scenario di analisi "Weather Report"

4.3.2 Server

Quando il client si connette la prima volta al server, questo gli fornisce tutto il codice javascript e HTML necessario per la SPA. In contemporanea, trasferisce al client un file in formato JSON contenente alcune impostazioni di configurazione: in particolare viene indicato l'elenco degli scenari da rendere disponibili all'utente e, per ciascuno di essi, alcuni dettagli sulle modalità di visualizzazione e le funzioni da richiamare per ricevere dal server stesso i dati opportuni.

A questo punto il server resta in attesa di richieste da parte del client. Ogniqualvolta quest'ultimo necessita di informazioni da visualizzare, effettua una richiesta http al server, il quale implementa alcune procedure atte a rispondere a tali domande.

Solitamente le richieste contengono alcuni parametri, che possono essere:

- *idPollutant*: una stringa contenente l'identificativo dell'inquinante del quale si necessitano i dati.
- *timestamp*: possono esserne presenti uno o due. In un caso vengono richiesti dati inerenti al giorno indicato nel timestamp, nell'altro tutti quelli relativi al periodo temporale compreso tra il primo e il secondo timestamp.
- *threshold*: rappresenta il valore inserito dall'utente che determina se la concentrazione media di inquinante nel giorno è critica o meno.

Per poter fornire i dati chiesti, il server si connette al database ed effettua la query specifica per le informazioni volute. Una volta ottenute tutte le informazioni dal DB, le invia al client all'interno di un documento in formato JSON.

Le richieste a cui il server è in grado di rispondere sono le seguenti:

1. *getZones*: non richiede alcun parametro, restituisce un file in formato KML con i dati necessari per disegnare sulla mappa i poligoni che delimitano le zone della città.

2. *getSensors*: non richiede nessun parametro, restituisce l'elenco dei sensori della rete di monitoraggio con la loro disposizione all'interno delle stazioni. Una stazione può contenere più sensori, mentre un sensore appartiene ad una unica stazione.
3. *getPollutant*: richiede come parametri l'identificativo dell'inquinante e due timestamp, restituisce le misure di quell'inquinante relative ad ogni sensore e per un periodo di giorni compresi tra i due timestamp, con granularità giornaliera.
4. *getVehicles*: richiede come parametri due timestamp, restituisce le misure del traffico diviso per tipologia di utilizzo per un periodo di giorni compresi tra i due timestamp, con granularità giornaliera.
5. *getFuelTypes*: richiede come parametri due timestamp, restituisce le misure del traffico diviso per tipologia di carburante per un periodo di giorni compresi tra i due timestamp, con granularità giornaliera.
6. *getAvgPollutant*: richiede come parametri l'identificativo dell'inquinante e due timestamp, restituisce la media delle misure di quell'inquinante tra i vari sensori per un periodo di giorni compresi tra i due timestamp, con granularità giornaliera.
7. *getHourlyPollutant*: richiede come parametri l'identificativo dell'inquinante e un timestamp, restituisce le misure di quell'inquinante relative ad ogni sensore e per il giorno indicato dal timestamp, con granularità oraria.
8. *getHourlyWeather*: richiede come parametro un timestamp, restituisce le misure relative al meteo per il giorno indicato dal timestamp, con granularità oraria.
9. *getDaysAboveThreshold*: richiede come parametri un inquinante, due timestamp e un valore numerico, la threshold. Restituisce, per ciascun sensore, la percentuale di giorni, nel periodo temporale identificato dai due timestamp, con una media giornaliera di inquinante maggiore della soglia.

4.3.3 Database

Le informazioni, per poter essere utilizzate e mostrate sul cruscotto, sono state memorizzate all'interno del database MongoDB in modo che la struttura dei dati sia ottimizzata per rispondere in maniera efficiente alle richieste effettuate dal cruscotto. I dati sono stati inseriti all'interno di alcune collezioni, ciascuna delle quali ha lo scopo di rispondere ad una famiglia di domande:

1. In una prima collezione sono state inserite le informazioni sui sensori presenti nella rete di monitoraggio utilizzata: per ciascun sensore viene indicato l'inquinante rilevato e la stazione nella quale vi è il sensore. Vengono inoltre memorizzate la latitudine e la longitudine della stazione, il nome e la sigla dell'inquinante. Ogni stazione può avere più sensori, ma un sensore può appartenere ad una sola stazione. La funzione *"getSensors"* utilizza le informazioni presenti in questa collezione per rendere lo scenario *"Monitoring Station"*, che ha lo scopo di mostrare nel suo insieme la rete di sensori per il monitoraggio della qualità dell'aria.
2. Due collezioni contengono i dati relativi all'andamento medio degli inquinanti e del meteo. La prima con granularità giornaliera, la seconda oraria. Per ogni sensore, per ogni giorno dell'anno per quanto riguarda la prima collezione e per ogni ora nel caso della seconda collezione, vengono memorizzate:
    a. Le informazioni sull'inquinante rilevato dal sensore: il valore medio delle misurazioni, l'unita di misura, nome e sigla.
    b. Le informazioni sul meteo: il valore medio di temperatura, in gradi celsius, di umidità, in percentuale, della velocità del vento, in km/h e di pressione, in hPa.
    c. Altre informazioni relative al sensore: la stazione che lo ospita e le sue coordinate geografiche.

    Le informazioni memorizzate in queste collezioni vengono prelevate rispettivamente dalle funzioni *getPollutant* e *getHourlyPollutant* per rendere gli

scenari *"Daily Pollutant Concentration"* e *"Hourly Pollutant Concentration"* fornendo i dati relativi all'andamento degli inquinanti per ciascun sensore presente sulla rete. Le informazioni presenti nella seconda collezione vengono anche utilizzate dalla funzione *getHourlyWeather* per lo scenario *"Weather Report"* sia per i dati inerenti gli inquinanti sia per quelli riguardanti i fattori meteorologici.

3. Altre due collezioni memorizzano le informazioni sul traffico medio giornaliero. Per ciascuna data vengono memorizzati, oltre al valore complessivo del traffico, nella prima le quantità di veicoli per ciascuna tipologia di utilizzo, nella seconda quelle per tipologia di carburante. L'accesso ai dati di queste due collezioni avviene rispettivamente tramite le funzioni *getVehicles* e *getFuelTypes* per rendere gli scenari *"Vehicle Type Report"* e *"Fuel Type Report"*.
4. Un'ultima collezione riporta i dati sull'andamento medio giornaliero degli inquinanti, calcolato come media tra i vari sensori: per ciascun giorno dell'anno, per ciascun inquinante è riportato il valore medio di tale inquinante, calcolato considerando ogni sensore in grado di rilevarlo. Anche in questo caso i dati vengono utilizzati per gli scenari *"Vehicle Type Report"* e *"Fuel Type Report"*, in particolare per la resa del grafico relativo agli inquinanti. La funzione che si occupa di accedere alle informazioni di questa collezione è *getAvgPollutant.*

Capitolo 5

Conclusioni

Questa tesi si è focalizzata sullo studio di un processo di analisi di dati con l'obiettivo di monitorare l'inquinamento nell'ambito urbano attraverso tecniche di business intelligence e sullo sviluppo di una applicazione che sia in grado di rendere graficamente, in maniera efficace, i risultati di tale analisi. Partendo dai dati inerenti all'inquinamento urbano e ponendoli in relazione alle informazioni sul traffico e sul meteo, sono stati selezionati alcuni KPI che, inseriti all'interno di scenari costruiti ad hoc, permettano all'utente una corretta valutazione della qualità dell'aria in un contesto urbano e gli consentano di attuare strategie con il fine di minimizzare l'impatto umano sull'ambiente. Tali scenari sono stati realizzati grazie ad un cruscotto che, sfruttando i frameworks AngularJS e Node.js, è stato sviluppato con le modalità di una Single Page Application.

Questo sistema può ancora essere modificato ed ampliato: da un lato integrando set di dati relativi ad una città diversa, da un altro inserendo nuove tecniche di analisi, da un altro ancora sviluppando nuovi strumenti da unire all'interno del cruscotto per variare le modalità di visualizzazione delle analisi.

Bibliografia e sitografia

[1] Elena Baralis, Tania Cerquitelli, Silvia Chiusano, Paolo Garza, and Mohammad Reza Kavoosifar, *Analyzing air pollution on the urban environment.* Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy.

[2] W. H. Inmon, *Building the data warehouse*, John Wiley & Sons, Inc. New York, NY, USA, 1992

[3] https://en.wikipedia.org/wiki/Data_warehouse, (ultimo accesso novembre 2017)

[4] Matteo Golfarelli, Stefano Rizzi, *Data warehouse, teoria e pratica della progettazione*, McGraw Hill, 2006

[5] http://www.arpalombardia.it/, (ultimo accesso novembre 2017)

[6] http://dati.comune.milano.it/, (ultimo accesso novembre 2017)

[7] https://www.wunderground.com/, (ultimo accesso novembre 2017)

[8] http://singlepageappbook.com/, (ultimo accesso novembre 2017)

[9] Atzeni, Ceri, Paraboschi, Torlone, "*Database systems*", McGraw Hill, 1999

[10] Gray, Jim. "*The Transaction Concept: Virtues and Limitations*" September 1981

[11] Armando Fox and Eric Brewer, "*Harvest, Yield and Scalable Tolerant Systems*", *Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99)*, IEEE CS, 1999, pg. 174–178.

[12] Seth Gilbert and Nancy Lynch, "*Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*", *ACM SIGACT News*, Volume 33 Issue 2 (2002), pg. 51–59.

[13] Eric Brewer, "*Towards Robust Distributed Systems*", Portland, Oregon, USA - July 16 - 19, 2000.

[14] https://docs.angularjs.org/guide, (ultimo accesso novembre 2017)

[15] https://docs.angularjs.org/guide/databinding, (ultimo accesso novembre 2017)

[16] http://leafletjs.com/, (ultimo accesso novembre 2017)

[17] http://c3js.org/reference.html, (ultimo accesso novembre 2017)

[18] http://getbootstrap.com/docs/4.0/getting-started/introduction/, (ultimo accesso novembre 2017)

[19] https://nodejs.org/en/docs/guides/, (ultimo accesso novembre 2017)

[20] https://www.mongodb.com/what-is-mongodb, (ultimo accesso novembre 2017)

[21] http://bsonspec.org/, (ultimo accesso novembre 2017)

[22] https://docs.mongodb.com/manual/core/data-modeling-introduction/, (ultimo accesso novembre 2017)

[23] https://docs.mongodb.com/manual/, (ultimo accesso novembre 2017)

[24] https://docs.mongodb.com/manual/sharding/, (ultimo accesso novembre 2017)