UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA "ENZO FERRARI"

Corso di Laurea Triennale in Ingegneria Informatica

Sviluppo di un'applicazione per l'analisi della produzione aziendale

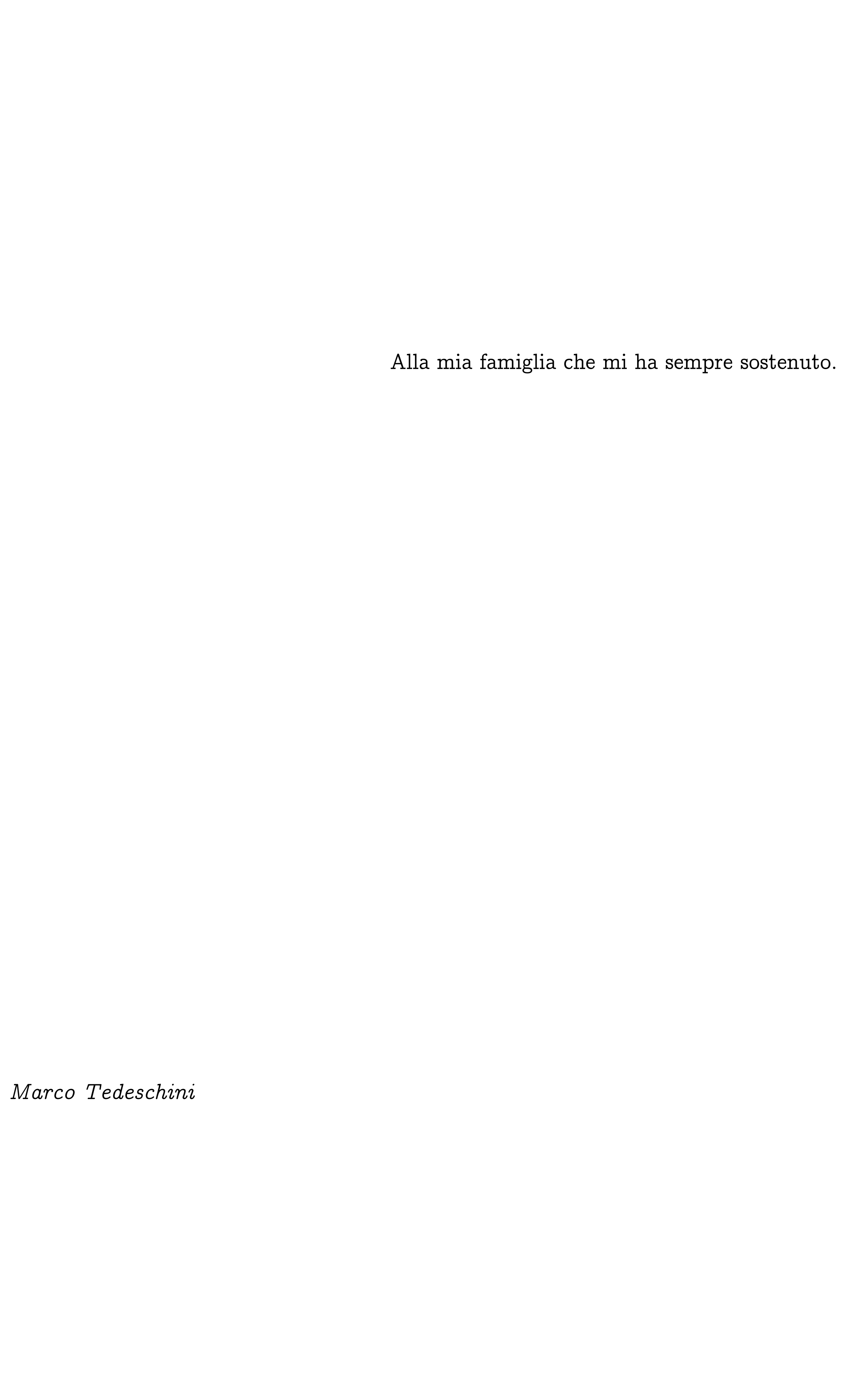
Relatrice

Chiar.ma Prof.ssa Sonia Bergamaschi

Laureando

Marco Tedeschini

Anno Accademico 2021/2022



Alla mia famiglia che mi ha sempre sostenuto.

Marco Tedeschini

Indice

Introduzione 1
1.1 Scopo della tesi 1
1.2 Struttura della tesi 1
Contesto 3
2.1 MultiData S.r.l. 3
2.2 Progetto del tirocinio 3
2.3 Risultati ottenuti 4
Metodologia 8
3.1 Linguaggi di programmazione e framework utilizzati 8
3.2 Python Django 9
3.3 Microsoft SQL Server per Django 10
3.4 ReactJS 11
3.5 Redux 12
Backend 15
4.1 Object-relational-mapping 15
4.1.1 Modelli 18
4.1.2 Operazioni CRUD 21
4.2 Django CLI 23
4.3 Ciclo richiesta/risposta 24
4.4 Django SQL Server per ARM 26
Frontend 30
5.1 Interfaccia con React 30
5.1.1 Component 30
5.1.2 Bundle 32
5.2 Logica degli stati con Redux 35
Esperimento finale 39
6.1 Descrizione dell'esperimento 39
6.2 Risultati 40
6.3 Conclusioni e lavoro futuro 43
Conclusioni 44
Bibliografia 46

Capitolo 1

Introduzione

Il seguente elaborato fornisce la descrizione del progetto svolto durante l'attività di tirocinio curricolare presso l'Azienda Multidata S.r.l..

L'esperienza si è rivelata un'ottima opportunità per approfondire le mie conoscenze e migliorare le mie competenze.

1.1 Scopo della tesi

L'obiettivo di questa tesi è descrivere le tecniche di progettazione, lo sviluppo e il funzionamento dell'applicazione gestionale in ambito industriale prodotta per l'Azienda MultiData S.r.l., chiarendo il processo decisionale che ha portato a preferire alcune tecnologie a discapito di altre e le basi delle scelte implementative impiegate.

Mi sono concentrato molto sui dettagli di queste scelte, perché, a mio avviso, possono essere prese come riferimento per la progettazione e la creazione di applicazioni solide e ben strutturate.

Ho preferito tralasciare la semplice e monotona spiegazione del codice in sé, adottando invece un metodo rivolto per lo più a una tesi di analisi e ricerca. Difatti, tratterò i risultati ottenuti in un solo paragrafo, il prossimo, per concentrarmi il più possibile sulle tecniche di sviluppo.

Valori e termini specifici presenti nel documento saranno volutamente modificati o omessi in virtù dell'obbligo di riservatezza aziendale.

1.2 Struttura della tesi

La struttura della tesi rispecchia il mio percorso di tirocinio svoltosi in azienda.

Nel capitolo 2 viene illustrato il contesto nel quale ho operato e nel quale ho avuto modo di svolgere il mio tirocinio, analizzando rispettivamente l'azienda MultiData S.r.l (2.1) e il progetto di tirocinio (2.2) con i conseguenti risultati ottenuti (2.3).

Proseguendo, nel capitolo 3, vengono analizzati tutti gli strumenti e le metodologie impiegati, utili ai fini dello sviluppo dell'applicazione. In particolare, mi focalizzo sui linguaggi di programmazione e i relativi framework (3.1), per poi scendere nel dettaglio

analizzando: Python Django (3.2), Django MSSQL (3.3), ReactJS (3.4), e infine Redux (3.5).

Successivamente troviamo la contrapposizione di due sezioni: Backend (capitolo 4) e Frontend (capitolo 5).

Il Backend si articola in Object-relational-mapping (4.1) - a sua volta scomposto nei Modelli (4.1.1) per i quali è caratterizzato e le relative Operazioni CRUD (4.1.2) - per poi proseguire con Django CLI (4.2), il Ciclo richiesta/risposta (4.3.) e Django SQL server per ARM (4.4).

Il Frontend, invece, si ramifica in due componenti, una delle quali avente delle sottocategorie: in merito troviamo l'interfaccia con React (5.1) - che vede la scomposizione in Component (5.1.1) e Bundle (5.1.2) - e parallelamente la logica degli stati con Redux (5.2).

Infine, nel capitolo 6, viene esplicato l'esperimento finale mediante la sua descrizione (6.1), i risultati (6.2) e le immediate conclusioni, relative anche al lavoro futuro (6.3).

Capitolo 2
Contesto

2.1 MultiData S.r.l.

Fig. 2.1 Logo dell'Azienda [1]

MultiData S.r.l. è un'Azienda leader nel settore dell'automazione e vede la sua specializzazione in impianti di mescolazione e dosaggio di gomma e plastica.

Nasce nel 1994 in provincia di Modena, quando l'automazione industriale era ancora agli albori. All'inizio l'Azienda si occupava della trasformazione dei quadri elettromeccanici in quadri con PC e PLC nelle poche ditte specializzate del settore.

Nel 2009, accogliendo le richieste di numerosi clienti, lancia sul mercato il suo progetto di maggior successo: *DosareX*, un unico programma di supervisione che racchiude tutti i software indipendenti di controllo utilizzati su una linea di produzione.

Adottando questa soluzione, MultiData riuscì a garantire un maggiore controllo e una migliore gestione dei flussi dei dati.

Ad oggi offre software completi al fine di garantire il migliore controllo produttivo degli impianti, con lo scopo di migliorare la qualità ed essere sempre più competitiva.

2.2 Progetto del tirocinio

Durante il tirocinio svolto presso MultiData S.r.l. mi è stato assegnato il compito riguardante la realizzazione di un progetto software per l'azienda.

In particolare, mi è stato chiesto di creare *ex novo* un'applicazione per gestire e analizzare i dati raccolti dai processi produttivi degli impianti dei suoi clienti.

Prima della progettazione dell'applicazione ho svolto ulteriori attività utili alla comprensione dei lavori svolti in azienda, tra le quali:

- analisi dei database e successiva comprensione dei dati;
- affiancamento ai tecnici MultiData nello sviluppo dei loro software per la verifica delle logiche di scambio dati con i sistemi gestionali e i database;
- studio delle *stored procedure* dell'azienda, utili a semplificare l'interazione con il database;
- dialogo informativo con gli sviluppatori per indagare quali fossero le migliori soluzioni per la progettazione del programma richiesto.

2.3 Risultati ottenuti

La progettazione segue le linee di un'accurata e scrupolosa analisi di tutti i dettagli. Il risultato che ne deriva è da ritenersi comunque in via di sviluppo; infatti, sarà possibile integrare nuove funzionalità alla solida struttura che ho sviluppato.

Nell'ambiente dell'automazione industriale l'analisi dei dati è un aspetto fondamentale e da non sottovalutare.

Ogni linea di produzione genera una considerevole quantità di informazioni utili, che lette e rilevate da sensori, misuratori e analizzatori, vengono salvate nei database.

L'analisi di questi dati "grezzi" è un processo complicato, per il quale accorrono in aiuto operazioni di *data mining*[1].

Attraverso le fasi di preparazione, trasformazione e filtraggio – come il *data cleaning*[2] – si possono estrarre, mediante l'utilizzo di tecniche analitiche, informazioni implicite che possono essere molto significative per un'azienda [2].

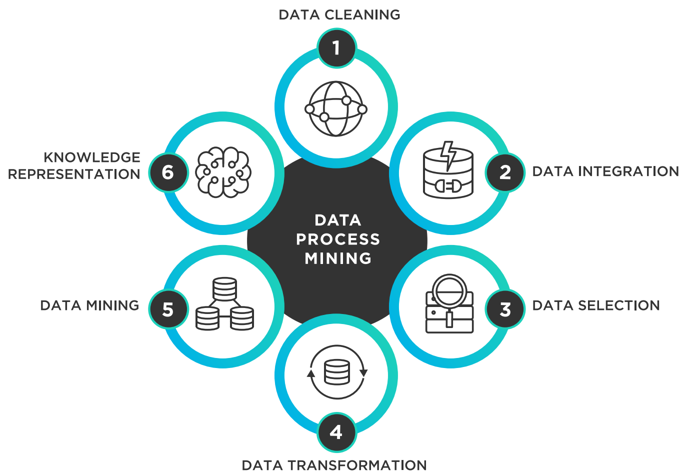


Fig. 2.2 Processi del *data mining.* [3]

---

[1] Pratica di analisi di grandi banche di dati al fine di generare nuove informazioni.

[2] È il processo di rilevamento e correzione (o rimozione) di record imprecisi, con una certa soglia di affidabilità, da un set di record. È capace di garantire la correttezza di una grande quantità di dati.

Scendendo nei dettagli della MultiData, le entità base che definiscono la struttura di un sito produttivo sono le seguenti:

- Reparti
- Linea di produzione
- Macchina
- Tipologia della macchina
- Stato della macchina
- Turno di lavoro
- Manutenzione programmata

Il sito produttivo è suddiviso in reparti a loro volta scomposti in linee di produzione. Ognuna di esse si avvale della composizione di più macchine.

Ciascuna macchina appartiene ad una specifica tipologia per mezzo della quale sono definiti degli stati macchina.

Proseguendo, troviamo i turni di lavoro correlati al sito produttivo, al reparto o alla singola linea di produzione. Le manutenzioni programmate sono associate alla macchina. L'elenco degli stati macchina per una certa tipologia, così come i turni di lavoro, hanno una validità temporale definita da una coppia di date; quindi, è possibile avere molteplici configurazioni di stati macchina per diversi periodi.

L'applicazione si occupa di integrare, in un unico ecosistema, esigenze relative al monitoraggio produttivo delle macchine dei suoi clienti. Il suo compito è quello di fornire all'utente una panoramica chiara, ma al tempo stesso dettagliata, dell'andamento della produzione aziendale.

Il prodotto finale, quindi, combina performance adeguate a visualizzare i dati in tempo reale, con un'interfaccia grafica comoda e reattiva.

La UX[3] è incentrata sulla semplicità: appena un utente si connette, è immediatamente indirizzato alla home. Ciò è pensato per rendere tutti i dati immediatamente fruibili al personale, in quanto l'utilizzo del programma è incentrato principalmente, come detto, sulla loro visualizzazione e non sulla loro modifica. Sarà dunque ottimale, per l'Azienda, che i suoi dipendenti possano avere un'idea chiara e concisa dell'andamento della produzione in un certo lasso di tempo, solamente con un click del mouse o un accesso dai loro *device*.

[3] User Experience, esperienza d'uso dell'utente finale.

La UI[4] della Home è semplificata al massimo, pur attenendosi ai principi del design minimale, per cui risulta comunque gradevole alla vista. In alto troviamo tre riguardi (*card*) che chiarificano immediatamente il volume e l'andamento della produzione, anche e soprattutto grazie alle icone colorate opportunamente con le tinte semaforiche.

Il primo di essi mostra (e permette di modificare) la data relativa ai dati da visualizzare. Immediatamente una icona rende l'idea dell'andamento della produzione specifica (ossia quella relativa a un limitato e certo periodo di campionamento, nel nostro caso un'ora), in tale data, rispetto a quella del giorno immediatamente precedente. Esso implementa uno scorrimento laterale dei giorni grazie a delle frecce poste al lato del periodo e un calendario (che comprende anche l'input da tastiera) per una consultazione rapida di una data ben precisa.

Al centro abbiamo il volume totale della produzione giornaliera e la relativa icona che ne rappresenta la variazione – positiva, neutra o negativa – rispetto alle 24 ore precedenti.

Il riquadro di destra, infine, mostra la differenza in percentuale tra il tempo materialmente impiegato per portare a termine la produzione dell'ultimo ciclo e quello impiegato per il ciclo immediatamente precedente.

Subito sotto le *card*, l'attenzione dell'utente è attirata dal grafico che visualizza i dati delle ultime 48 ore lavorative: due aree, rispettivamente una blu e una verde, mostrano all'utente le varie pesate effettuate. Un'apposita libreria si occupa di tracciarne la curva di interpolazione, affinché all'operatore sia evidente l'andamento in base al tempo.

Due pratici bottoni permettono di selezionare il giorno lavorativo corrente e quello precedente, in modo che sia possibile dimensionare il grafico per avere una panoramica più precisa o più generale dei dati, in base all'occorrenza.

Scorrendo ancora, si arriva alle tabelle che forniscono in modo chiaro e conciso i dati principali che riguardano le produzioni e le rispettive pesate.

Mentre la prima tabella raffigura le produzioni della giornata, la seconda al caricamento risulta vuota, ma si popolerà opportunamente alla selezione di una produzione da parte dell'utente, mostrando i dati di tutte le relative pesate per una rapida consultazione.

L'interazione con le tabelle è semplice: è possibile e immediato anche per l'utente meno esperto, grazie alle apposite funzioni, cercare tra i dati, ordinare questi ultimi, decidere il numero di righe da visualizzare e scorrere tra le pagine della tabella.

L'utente più esperto, inoltre, ha accesso a un'altra sezione: quella delle impostazioni, da cui è possibile configurare il servizio di notifica preferito (tra e-mail e SMS) e alcuni settaggi sulla visualizzazione dei dati, in particolare quelli relativi alle tolleranze.

---

[4] User Interface, interfaccia di gestione da parte dell'utente. In questo caso, l'interfaccia grafica.

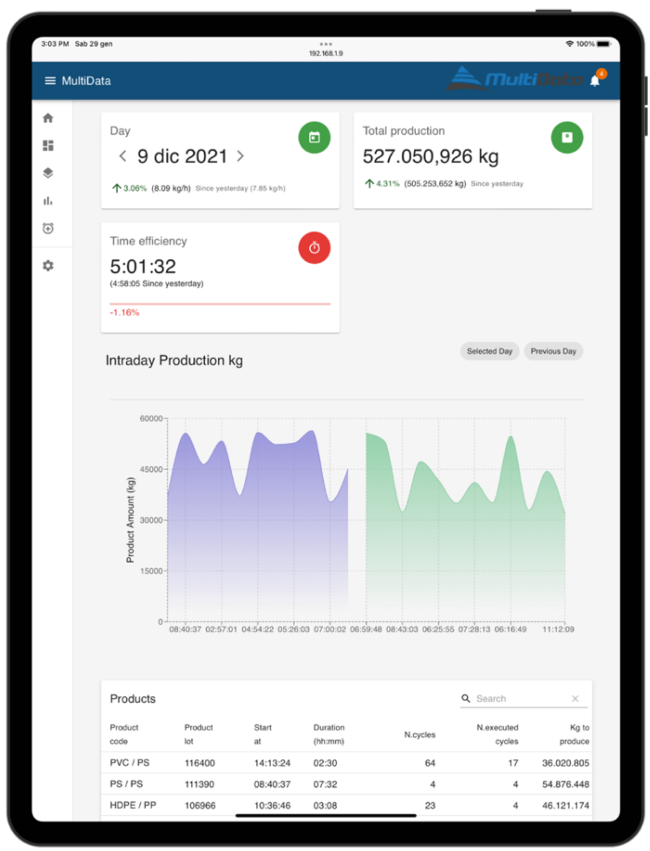


Fig. 2.3 Landing page su *iPad Pro* 5th generation

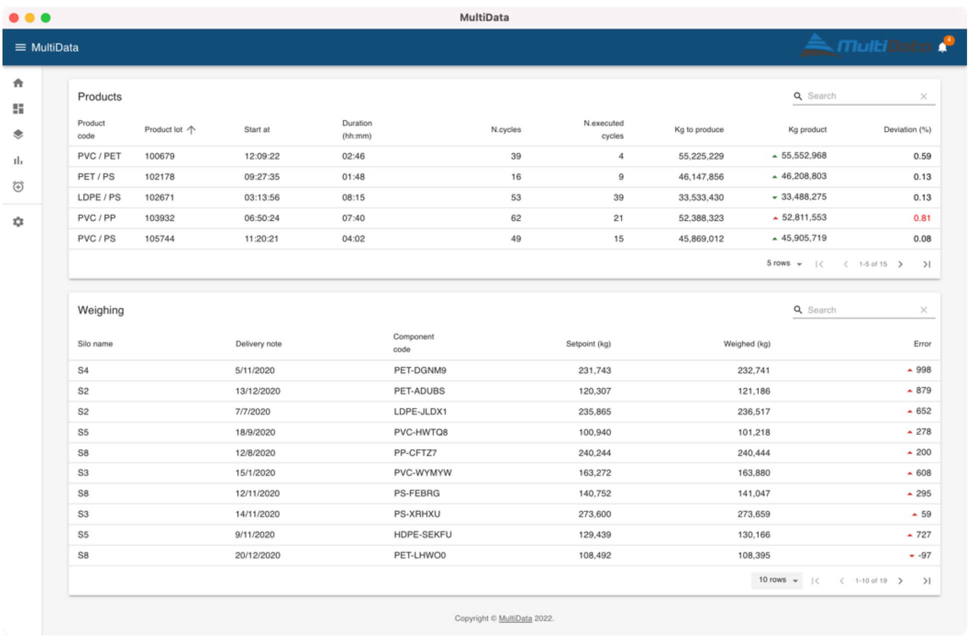


Fig. 2.4 Vista delle tabelle su Applicazione desktop (cross-platform)

Il prodotto ottenuto è da ritenersi in via di sviluppo; infatti, sarà possibile per i tecnici dell'Azienda creare nuove feature da aggiungere alla solida struttura che ho creato.

Capitolo 3

Metodologia

3.1 Linguaggi di programmazione e framework utilizzati

Prima di iniziare, ho redatto – insieme al responsabile in ufficio del tirocinio – un documento con la specifica dei requisiti software (SRS), per il quale abbiamo svolto un attento e scrupoloso studio, avvalendoci della selezione dei linguaggi di programmazione più utili per la sua realizzazione.

Per il frontend dell'applicazione ho optato per il web framework *ReactJS*[5], mentre per il backend la mia scelta è ricaduta su *Django REST Framework (DRF)*[6]. Entrambi sono framework largamente utilizzati nello sviluppo di software.

I due sono collegati tramite chiamate API utilizzando richieste *axios*[7] nel frontend.

Questo metodo consente di disaccoppiare completamente frontend e backend con un risvolto vantaggioso in termini di modularità.

Dal momento che è definito uno standard di *endpoint* e dati restituiti con cui lavorare, il frontend e il backend possono essere sviluppati in parallelo.

La netta separazione fra frontend e backend, poi, migliora anche la scalabilità del prodotto: grazie a essa, infatti, si potranno in futuro affiancare nuove interfacce anche per dispositivi diversi.

È già stata testata la compilazione di una soluzione desktop sfruttando la libreria *Electron*[8].

Un possibile ampliamento potrebbe essere la realizzazione di un'applicazione per dispositivi *mobile* tramite *React Native*[9], il quale potrebbe permettere una maggior fruibilità dei dati.

---

[5] https://reactjs.org

[6] https://www.django-rest-framework.org

[7] *Axios* è una libreria JavaScript utilizzata per effettuare richieste HTTP da node.js o con XMLHttpRequests dal browser e supporta l'API Promise nativa di JS ES6.

[8] https://www.electronjs.org

[9] React Native è un framework JavaScript per la scrittura di applicazioni mobili con rendering nativo per iOS e Android. Si basa su React, per la creazione di interfacce utente, ma invece di indirizzare il browser, si rivolge alle piattaforme mobili (https://reactnative.dev).

3.2 Python Django

Python è un linguaggio di programmazione ad alto livello, facile da imparare e da usare, e con molte funzionalità.

Uno dei vantaggi di Python è indubbiamente la ricchissima libreria standard, corredata da un'ampia offerta di librerie e framework. Questi ultimi spesso permettono integrazioni efficaci ed efficienti, nonché uno sviluppo rapido e un testing agevole e immediato.

Uno dei framework più conosciuti è proprio *Django*[10]: un ottimo progetto gratuito e open source creato da sviluppatori esperti e che gode di una vasta comunità in grado di mantenerlo costantemente aggiornato.

Django ha molto da offrire in termini di sviluppo delle web app e supporta tutte le più recenti tecnologie per la crescita di esse.

La scelta di Django è stata determinata dalla sua filosofia completa, ma al tempo stesso pulita e semplice, in chiaro stile Python: Django, infatti, si pone come obiettivo quello di offrire un supporto per uno sviluppo rapido, vale a dire un design pulito e pragmatico.

Il codice viene scritto in modo immediato, grazie alle API e alla chiarezza della documentazione. Lo sviluppatore può quindi dedicarsi interamente alla cura del suo software, delegando alla tecnologia di Django la gestione del backend con una semplice configurazione.

Per la verità ho valutato anche altre soluzioni: principalmente, ho preso in considerazione il web framework *Flask*. Si tratta di un framework molto più semplice di Django: è scorrevole e consente di sviluppare con una certa facilità un'API RESTful[11].

Flask ha, inoltre, un'estensione chiamata *Flask API* che replica esattamente ciò che *DRF* fa per Django.

La documentazione di Flask, tuttavia, sconsiglia lo sviluppo di software basato sullo stesso framework per la fase di produzione, in quanto la libreria è ancora in fase di sviluppo e non è ritenuta sufficientemente sicura per il rilascio di software esposto alla rete.

---

[10] https://www.djangoproject.com

[11] È un'interfaccia di programmazione delle applicazioni (API o API web) conforme ai vincoli dello stile architetturale REST (REpresentational State Transfer).

3.3 Microsoft SQL Server per Django

L'Azienda MultiData si avvale del più famoso RDBMS prodotto da *Microsoft*, *Microsoft SQL Server (MSSQL)*, per registrare i suoi dati di produzione.

Fig. 3.1 Logo di Microsoft SQL Server

La prima versione di SQL Server risale al 1989 ed era in grado di lavorare con basi di dati di medio-piccole dimensioni, ma, a cominciare dalla release dell'anno 2000, Microsoft ha cominciato a impiegarlo anche per operare con grandi database.

La versione attuale (2019, nome in codice Seattle) comprende cinque differenti versioni:

- *Express* è gratuita ed è limitata nell'utilizzo e nelle performance;
- *Web*, simile alla precedente, distribuita unicamente ai service provider, concede minori condizionamenti;
- *Standard* è concessa tramite licenza solo alle aziende e rimuove i maggiori limiti;
- *Enterprise* rimuove i limiti hardware e software: ad oggi è in grado di lavorare con database da 524 PB, riesce a indirizzare 12 TB di RAM e supporta fino a 640 core logici;
- *Developer* offre le stesse funzionalità della versione Enterprise, limitandone però l'uso ai soli ambienti di sviluppo e non alla produzione.

Microsoft come dialetto usa *Transact-SQL*, variante dello standard *SQL-92* e sfrutta come protocollo, a livello applicazione, *Tabular Data Stream (TDS)*, pur supportando anche il più diffuso *ODBC (Open Database Connectivity)*.

La porta di default usata dal DBMS è la 1433.

Django di default non fornisce un collegamento a MSSQL, però Microsoft ha sviluppato sulla base dell'*ORM* di Django un suo backend in grado di offrire una connettività a SQL Server e al nuovo *DB Azure SQL*, chiamando questa integrazione di terze parti *mssql-django*[12].

[12] https://github.com/microsoft/mssql-django

3.4 ReactJS

React (noto anche come *React.js* o *ReactJS*) è una libreria *JavaScript* che ho scelto per lo sviluppo del frontend. Si tratta di software open source ed è utilizzata per la creazione di interfacce utente basate su componenti.

È sviluppata da *Meta* (conosciuta come *Facebook*) e da una florida comunità di sviluppatori e aziende.

React può essere utilizzata come base per lo sviluppo di *SPA* (*single page application*).

La programmazione dell'interfaccia utente avviene attraverso i *components*: classi o funzioni JavaScript che possono accettare dati in input, ad esempio proprietà (*props*), e restituiscono elementi di React atti a contenere le istruzioni che descrivono come una sezione dell'interfaccia utente (UI) debba apparire sullo schermo del client.

Oltre alla ricezione di dati, un componente può mantenere uno stato interno e questa peculiarità fa sì che, a ogni cambio di stato, React aggiornerà tutte e sole le parti della UI che dipendono unicamente da tali dati [4].

In effetti lo stesso nome React, attribuito alla libreria, vede le sue origini in questa caratteristica: si fa riferimento alla reattività che lega le risposte dell'interfaccia ai cambiamenti di stato e notevole è la fluidità con cui esse avvengono.

Avendo sfruttato tale libreria per l'applicazione si potrà anche decidere, in futuro, di conservare il backend per effettuare il rendering, a lato server, e compilare il frontend per applicazioni mobili, attraverso l'uso della tecnologia di React Native.

React negli ultimi anni ha via via acquisito sempre più popolarità.

Le ragioni dell'utilizzo di React sono riconducibili alle prestazioni, all'ampia offerta di librerie aggiuntive e, soprattutto, alla semplicità di apprendimento: punto di estrema forza, grazie all'ottima documentazione ricca di esempi.

Come mostrato nel seguente grafico – rispetto ai competitors *Angular.js*, *Vue.js* e *jQuery* – React ha visto una crescita incessante e quasi lineare, con un numero di download che è praticamente triplicato negli ultimi due anni.

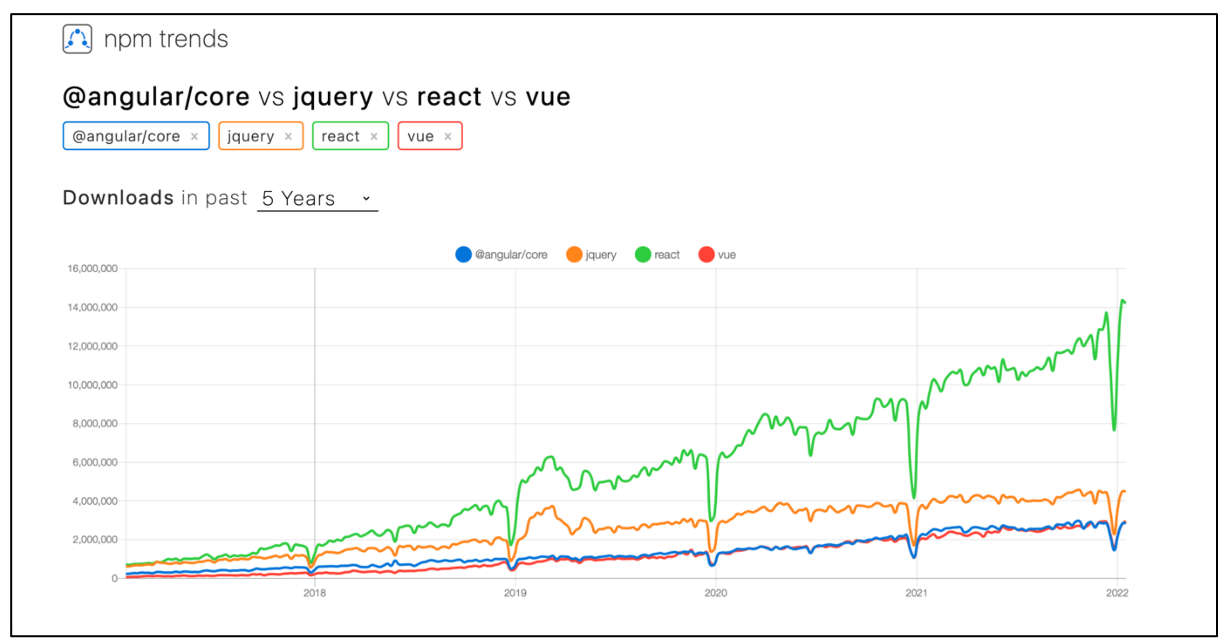


Fig. 3.2 npm downloads trends [5] al 16 Gennaio 2022

3.5 Redux

Redux è una libreria che fornisce un modo semplice di separare lo stato e la logica di una applicazione web dalla presentazione.

Essa mantiene lo stato di un'intera applicazione in un albero di stati immutabile (definito in un oggetto), che non può essere modificato direttamente. Quando qualcosa cambia, viene creato un nuovo oggetto (usando i *reducer*).

La capacità di poter gestire gli stati dell'applicazione permette di muoversi molto facilmente nella cronologia di essi.

È uno strumento fondamentale per lo sviluppo di un'applicazione; infatti, permette il *debug* dei singoli stati.

Utilizzando Redux insieme a React è possibile creare un software in cui si ha una netta separazione delle responsabilità tra la presentazione e la logica insieme allo stato dell'applicazione (*decoupling*).

Nello specifico, in una applicazione di questo tipo, ReactJS si occuperà, con l'utilizzo di componenti *stateless* (senza stato), di gestire la presentazione grafica, mentre Redux è incaricato di gestire la logica e l'intero stato della applicazione.

Queste due tecnologie combinate implementano il pattern a flussi di dati unidirezionale (chiamato più comunemente *one-way dataflow* o *unidirectional dataflow*). Questo schema permette di avere un'applicazione a stati predicibili.

L'insieme del pattern a flussi unidirezionali e l'immutabilità degli stati rendono il prodotto finale estremamente robusto e performante.

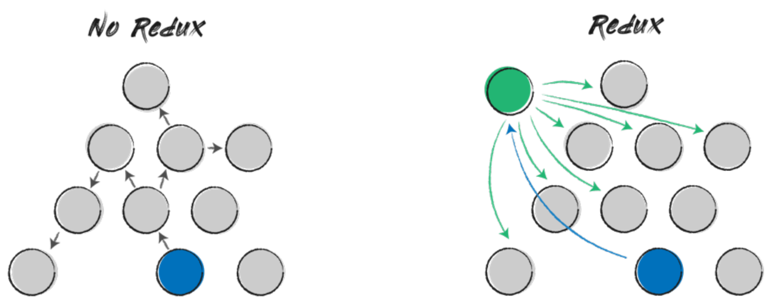


Fig. 3.3 Gestione stati senza e con l'uso di Redux [6]

Inoltre, tale ecosistema fornisce interessanti librerie di terze parti per integrare operazioni asincrone in una applicazione.

In un'architettura di questo tipo potremmo dire che i componenti ReactJS riflettono lo stato dell'applicazione senza conoscerne la logica dalle azioni di Redux.

Questo permette ad un'applicazione di essere estremamente flessibile: per esempio, potremmo completamente rivedere la presentazione con ReactJS senza necessariamente modificare logica e stati gestiti da Redux.

Parlando di Redux è doveroso descrivere nel dettaglio cosa sono *store*, *action*, *reducer* e *dispatcher*.

Le "azioni" sono semplici oggetti che attraverso il *dispatcher* inviano informazioni allo *store*. Grazie ai middleware (tipo *Redux-Thunk*[13] o *Redux-Saga*[14]) possiamo creare *action* asincrone per eseguire chiamate *AJAX* per la comunicazione con il server.

I *reducer* o "riduttori" sono semplici funzioni che consentono di aggiornare sezioni dello *store* in base al contenuto delle *action*.

Lo *store* è un archivio che contiene lo stato dell'applicazione. Attraverso le "azioni" e i "riduttori" è possibile aggiornare i suoi contenuti. Le sole modifiche allo *store* vengono

---

[13] Consente di scrivere "azioni" che restituiscono una funzione invece di una action. Può essere utilizzato per ritardarne l'invio o per determinare che si verifichi una certa condizione. La funzione riceve come parametri i metodi dello store dispatch e *getState*.

[14] È una libreria che mira a migliorare e gestire nel mondo più semplice gli effetti collaterali che si possono creare nell'applicazione, ad esempio: azioni asincrone come il recupero dei dati o l'accesso alla cache del browser.

notificate ai componenti ReactJS che rifletteranno sul browser le modifiche prodotte da questi cambiamenti di stato.

Come detto in precedenza, lo stato dell'applicazione è immagazzinato in una struttura ad albero all'interno di uno *store*. L'unico modo per cambiarlo è di creare un'azione, un oggetto che descrive ciò che sta accadendo, e di inviarlo allo *store*.

È fondamentale che la modifica degli stati avvenga seguendo questa sequenza di eventi, senza violare l'immutabilità della struttura di oggetti.

Per specificare in che modo lo stato venga aggiornato in risposta a un'azione si scrivono funzioni (riduttori) che calcolano un nuovo stato in base al vecchio stato.

Per poter capire a fondo questa tecnologia bisogna sempre tenere a mente questi punti chiave e le relazioni che intercorrono tra loro.

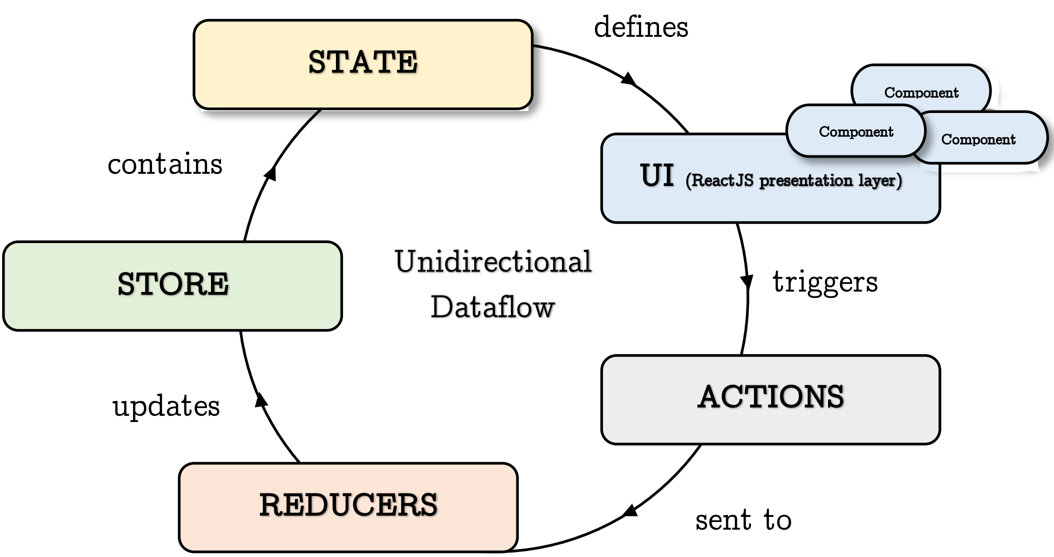


Fig. 3.4 Unidirectional Dataflow

Capitolo 4

Backend

4.1 Object-relational-mapping

Una delle caratteristiche di *Django* che mi hanno portato a scegliere questa tecnologia a discapito delle altre opzioni riguarda l'integrazione di un livello di *Object-relational mapping* (*ORM*[15]) che può essere utilizzato per interagire con vari database.

Questo mi permette di poter accedere ai dati dal codice Python senza dover scrivere "*raw*" *Structured Query Language*. Dunque, non ci sono problemi con le query *SQL* che, sotto la scocca, saranno usate da Django per interrogare il database per poi ricevere i dati necessari.

Gli sviluppatori Django hanno un modo speciale di manipolare l'oggetto modello Python corrispondente a differenza di molti altri framework Python che gestiscono direttamente il database tramite SQL. Django, per impostazione predefinita, funziona immediatamente con i sistemi di gestione di database relazionali come *PostgreSQL*, *MySQL*, *MariaDB*, *SQLite* e *Oracle*.

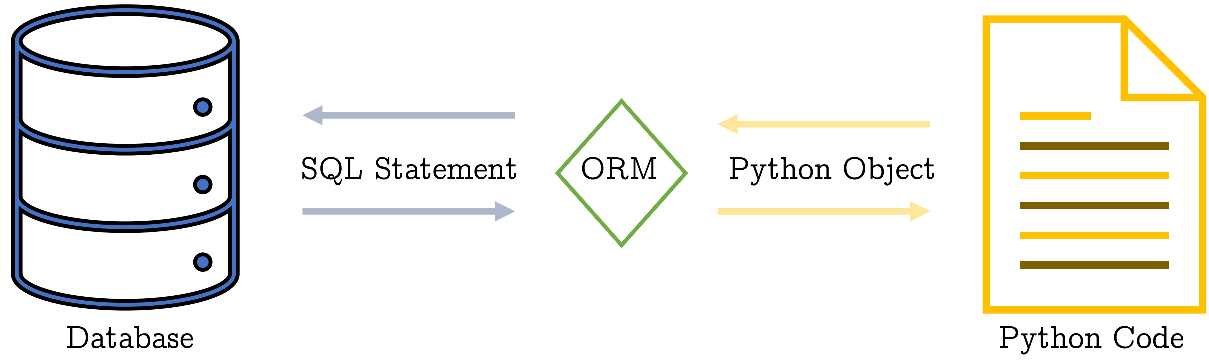


Fig. 4.1 ORM: Conversione SQL Statement in oggetti Python

La tecnica dell'*Object-relational mapping* offre numerosi vantaggi in termini di sviluppo: in primo luogo essa si occupa, al posto dello sviluppatore, della costruzione

---

[15] In informatica è una tecnica di programmazione che consente di convertire dati tra sistemi di tipo incompatibili utilizzando linguaggi di programmazione orientati agli oggetti.

dell'istruzione finale e della sua traduzione nel dialetto proprio del DBMS configurato. Il codice sarà dunque più semplice da aggiornare, mantenere e riutilizzare. Inoltre, se gli statement sono scritti correttamente, sarà ORM a occuparsi del miglioramento e dell'ottimizzazione delle query.

Inoltre, ORM – come detto – svincola lo sviluppo dal DBMS, per cui saranno molto rari i casi in cui lo sviluppatore dovrà interagire direttamente con l'SQL. Questo passaggio, noto come *Database Abstraction*, facilita la transizione da un database ad un altro senza complicazioni. ORM, quindi, evita al programmatore la replica di query simili che differiscano per le implementazioni nel dialetto semplice [7].

Per esempio, un'operazione semplice come la proiezione delle prime 100 tuple di una selezione è ottenuta aggiungendo "*limit 0,100* alla fine dell'istruzione *select* in *MySQL limit*, mentre la stessa operazione è il frutto di "*select top 100 * from table*" in MSSQL. Nel nostro caso, sarà proprio ORM a occuparsi di completare opportunamente la query prima di inviarla al driver.

```
1  SELECT * FROM Table LIMIT 0,100;         -- MySQL / PostgreSQL
2  SELECT TOP 100 * FROM Table;             -- MSSQL
3  SELECT * FROM Table WHERE ROWNUM = 100;  -- Oracle
4
```

Dal momento che i dati vengono trattati come oggetti, in caso di relazioni, il livello ORM si occuperà di estrarre automaticamente le informazioni con le loro relative dipendenze.

La maggior parte dei livelli ORM tratta l'aggiunta di nuovi dati (inserimento SQL) e l'aggiornamento dei dati (aggiornamento SQL) allo stesso modo, rendendo la scrittura e la manutenzione del codice un'operazione semplicissima.

L'uso di SQL espone al rischio di subire un attacco di tipo *SQL injection (SQLi)* che può portare un utente malintenzionato a eseguire codice SQL arbitrario su un database, con conseguente fuga di dati sensibili o eliminazione di record o perdita di dati.

Per prevenire questo genere di attacchi è necessario filtrare molto accuratamente i dati presi in input ed è doveroso mantenere aggiornati e controllati questi sistemi di sicurezza. ORM protegge le applicazioni da possibili attacchi di SQL injection, grazie all'abilità del framework di filtrare i dati.

Il grafico a barre sottostante è stato redatto dalla nota compagnia *Akamai Technologies*[16] che si occupa della distribuzione di contenuti via internet e della

[16] https://www.akamai.com

sicurezza sul web. In questo caso è ben chiaro come la maggior parte degli attacchi informatici analizzati appartengono alla tipologia SQLi.

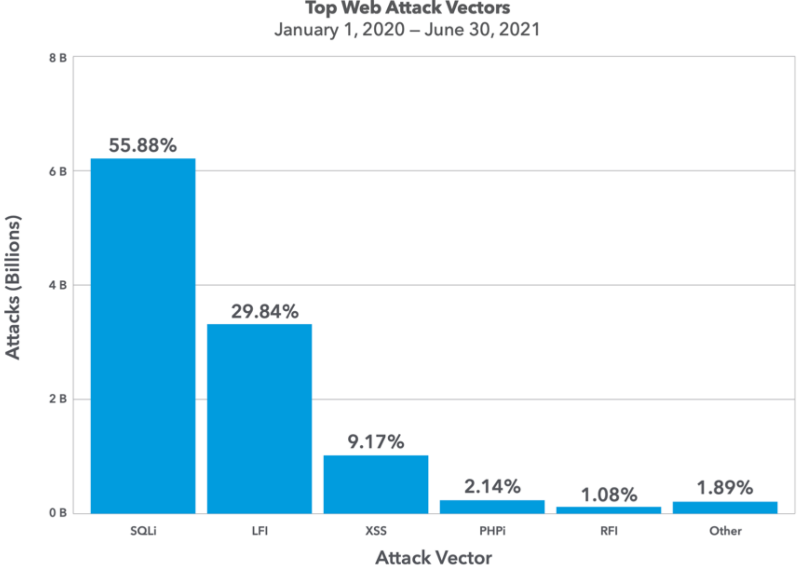


Fig. 4.2 Grafico che mostra come gli attacchi SQLi siano i più frequenti [8]

Bisogna infine aggiungere che, grazie all'uso di un efficace sistema di *cache management*, le entità gestite da ORM vengono memorizzate in memoria, riducendo così il carico sul database.

Ciononostante, ORM presenta anche alcuni svantaggi. Innanzitutto, la tecnica utilizzata è piuttosto complessa e rende le operazioni di *debugging* molto difficoltose, infatti spesso non è immediato risalire a eventuali errori nelle query.

Svincolandosi poi dal DBMS, ORM toglie al programmatore la facoltà di ottimizzare le query a piacimento in base alle sue conoscenze di questo o quell'altro dialetto SQL. Se è vero che, a lungo termine, il vantaggio offerto dalla compatibilità con più DBMS risulta alla base del successo di questa tecnica, per applicazioni pratiche ciò potrebbe essere un limite e non un vantaggio.

Non va poi tralasciato il fatto che ci sono alcune situazioni (poche, per la verità) in cui lo sviluppatore sarà comunque costretto a scrivere da sé le query SQL. In questi casi è facilmente intuibile che il programmatore dovrà occuparsi di ottimizzare le query e, soprattutto, di proteggerle adeguatamente dal rischio di *injection*.

Infine, qualora si lavori su applicazioni di *Big Data Analysis*, può essere utile minimizzare il numero delle richieste al DBMS: per fare questo sarà inevitabile un lungo lavoro a livello ORM che richiede spiccate competenze e, soprattutto, conoscenze – frutto di anni di studio ed esperienza – con notevoli costi sui tempi di sviluppo.

4.1.1 Modelli

In effetti, l'ORM di Django è solo un modo "pythonic" per creare le query SQL, per interrogare e manipolare il database e, infine, deserializzare i risultati.

Il codice Python si integrerà con il database attraverso la creazione di un model per ogni tabella.

Il *Model*, di cui vedremo presto un esempio, è volto alla definizione della struttura della tabella cui fa riferimento; i suoi campi corrispondono alle colonne e devono essere inizializzati chiamando l'opportuno costruttore del tipo a cui appartiene il contenuto della cella.

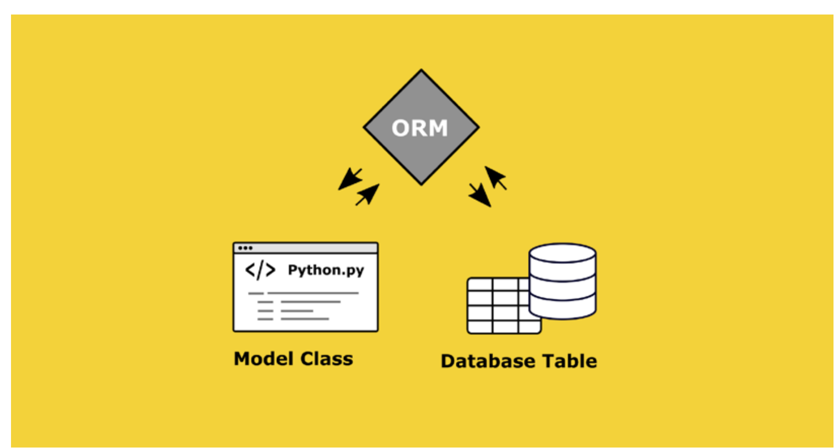


Fig. 4.3 Conversione delle *model class* in tabelle nel database [7]

All'interno del pacchetto *models* sono infatti racchiuse numerose classi volte a rappresentare proprio i diversi tipi di dati adottati nei vari DBMS, la cui ottimizzazione è senz'altro il fulcro dell'efficacia di ORM: esse possono non solo mappare i dati – serializzandoli e deserializzandoli sui tipi di Python – ma possono validare gli stessi dati, semplificandone di conseguenza i controlli. Per svolgere al meglio quest'ultimo compito, si utilizzano i *Validator*.

I Validator sono dei callable che ricevono un valore e sollevano un'eccezione di classe *ValidationError* se tale valore non rispetta alcuni criteri. Il loro punto di forza consiste nella capacità di riutilizzare una logica di validazione tra più tipi o tra più campi.

All'interno dei *Field* i Validator sono fondamentali per accertarsi che i dati inseriti siano accettabili dalla colonna della tabella in questione. Per ogni Field l'attributo (di tipo lista) validators include tutti i Validator che devono essere impiegati per assicurarsi della correttezza del dato da manipolare.

Esempi fondamentali sono il *RegexValidator* e una sua sottoclasse, l'*URLValidator*, che sfrutta le regular expression per validare una stringa che dovrebbe rappresentare un URL.

L'URLValidator accetta nel parametro "schemes" una lista di schemi URL/URI che devono essere approvati. Quella di default è ["http", "https", "ftp", "ftps"].

Qualora si debba verificare la compatibilità di un indirizzo e-mail con lo standard *RFC-2822*, si può usare la classe *EmailValidator*.

Analizzando più approfonditamente la struttura delle classi dei Field, si nota innanzitutto che è stata curata la semantica dei nomi attraverso l'adozione di uno standard semplice ma efficace: ognuno di essi, infatti, è composto dal tipo, seguito da Field, secondo la sintassi del camel case.

Sono state valorizzate anche le proprietà comuni – a partire da una serie di parametri presenti nella superclasse – che possono essere istanziate opportunamente dai costruttori delle sottoclassi.

Tra queste proprietà le più evidenti sono i valori booleani *primary key*, *unique* ed *editable* che, qualora settate a vere, codificano rispettivamente l'istanza della classe come chiave primaria, come univoca nel valore e ne abilitano la modifica. Quest'ultima, per esempio, potrà essere inibita automaticamente dai costruttori di alcune sottoclassi i cui valori debbano essere generati automaticamente.

Vi è poi un altro parametro interessante: il *default*, che può accettare una funzione lambda atta a istanziare la classe con un valore calcolato opportunamente a runtime.

Andiamo, dunque, a estrapolare qualche esempio per mostrare alcune scelte implementative interessanti, che rendano idea dell'ottima integrazione con i vari DBMS di cui l'ORM dispone.

I valori numerici interi (corrispondenti cioè al tipo *int* in Python) sono mappati nella classe *IntegerField*, estesa, tra le altre, dalla classe *AutoField* che si occupa di gestire un valore automatico incrementale.

BooleanField, poi, rappresenta i valori booleani, ma ha il difetto che possa solo rappresentare i valori "Vero" e "Falso", non potendo dunque essere nullo.

Per ovviare a questo limite è a disposizione la sua estensione, la classe *NullBooleanField*.

Per rappresentare il testo, soprattutto se breve, è usata la classe *CharField*, il cui costruttore ha un parametro molto importante per alcune sottoclassi, le quali lo sfrutteranno in particolare per la validazione dei dati: il riferimento è a *max_length* che, come suggerisce il nome, misura la lunghezza massima in caratteri del testo.

Esempi sono EmailField, estensione di CharField con max_length = 254, che integra un EmailValidator per controllare la consistenza dei dati inseriti, e URLField, sottoclasse di CharField con URLValidator e istanziata con max_length limitato, per scelte implementative, a 200 caratteri.

Anche la rappresentazione delle date e dei tempi è piuttosto precisa e in linea con i tipi di dati dei DBMS più diffusi. Per la sola data (e, dunque, per relazionarsi con la

classe date del pacchetto datetime), si usa il *DateField*, il quale ha alcuni parametri che ne rendono particolarmente comodo l'utilizzo. In particolare, grazie ad auto_now, il field è aggiornato con la data corrente a ogni salvataggio (ossia ogni qualvolta venga chiamato il metodo *save()*), implementazione particolarmente usata per salvare i timestamp per le ultime modifiche.

Vi è poi un parametro molto simile, *auto_now_add*, che differisce da quanto appena visto perché valido solo al momento della creazione del dato (senza cioè alcuna interazione con il metodo *save()*).

Se uno di questi parametri è settato a "Vero", il costruttore si preoccupa di settare a Falso l'option editable, onde evitare modifiche da parte dell'utente.

Per poter rappresentare un orario si usa la classe *TimeField*, mappata su *datetime.time*, mentre, qualora si volesse salvare sia la data che l'ora, sarà necessario l'utilizzo di *DateTimeField*, mappato a sua volta su *datetime.datetime*.

Per raffigurare i numeri decimali sono presenti due classi: FloatField e DecimalField.

FloatField è mappato su float, (a 64 bit, in Python), mentre *DecimalField* sfrutta la classe Python *Decimal* del pacchetto decimal. Quest'ultimo dispone di due opzioni, *max_digits*, che serve per limitare il numero di cifre (intere e decimali) del numero, e *decimal_places*, che ne indica la precisione.

Infine, è doveroso un accenno anche alla classe *UUIDField*, mappato su UUID di Python, fondamentale per la gestione degli identificativi univoci universali. Per il suo utilizzo occorre inserire un metodo di default, altrimenti l'identificativo non può essere generato in automatico.

È rilevante considerare come solo PostgreSQL supporti nativamente il datatype *uuid*, che sarà proprio quello in cui il dato sarà convertito qualora si usasse questo DBMS; parallelamente, in altri sistemi, la colonna del database sarà popolata da array di *char* di lunghezza 32 caratteri.

Sono inoltre presenti alcuni Field per la gestione delle relazioni, in quanto Django ORM esprime la sua massima potenzialità con i RDBMS.

Per rappresentare una chiave esterna, è usata la classe *ForeignKey*, il cui costruttore richiede il riferimento al modello (qualora non fosse ancora stato creato come istanza, è possibile indicarne il nome come stringa) e all'azione da compiere in caso di eliminazione (*on_delete*). Per delineare quale azione svolgere a seguito di una cancellazione, si usano delle variabili del modulo *django.db.models*, che corrispondono alle equivalenti nei vari dialetti SQL: *CASCADE*, che avvia una cancellazione a cascata, *SET NULL* e *SET_DEFAULT*, che, a cascata, modificano i dati al loro

valore nullo o di default, *ON_PROTECT*, *RESTRICT* e, infine, *DO_NOTHING* (che ignora semplicemente l'eliminazione).

Per definire le *metaoptions*, si usa una classe *Meta* interna alla classe del Model stessa, che avrà salvati nei suoi campi, tra gli altri, il nome della tabella e i nomi mnemonici singolari e plurali. È interessante notare che, ai fini dell'internalizzazione dell'output, troviamo la pratica, ormai cristallizzata, di importare la funzione *gettext()* della libreria django.utils.translations rinominandola con un semplice underscore, per poter usare le linee, molto pythonic, *verbose_name = _("")* e *verbose_name_plural = _("")* che renderanno semplicissima, nonché molto elegante, la traduzione automatica del testo.

Di seguito è mostrata una porzione di codice riguardante la creazione di un modello (tabella).

```
1  from django.db import models
2  from django.utils.translation import gettext as _
3  
4  class Weighing(models.Model):
5      _id = models.AutoField(primary_key=True, db_column='id')
6      silo_number = models.IntegerField(db_column='silo_number')
7      silo_name = models.CharField(max_length=10, db_column='silo_name')
8      delivery_note = models.CharField(max_length=20, null=True, db_column='delivery_note')
9      component_code = models.CharField(max_length=30, db_column='component_code')
10     set_point = models.DecimalField(max_digits=18, decimal_places=0, db_column='set_point')
11     weighed = models.DecimalField(max_digits=18, decimal_places=0, db_column='weighed')
12     error = models.DecimalField(max_digits=18, decimal_places=0, db_column='error')
13     weighing_time = models.DateTimeField(null=False, db_column='weighing_time')
14 
15     ...
16 
17     class Meta:
18         db_table = "Weighing"
19         verbose_name = _("Weighing")
20         verbose_name_plural = _("Weighings")
21 
22     def __str__(self):
23         return f"{self._id} - {self.weighed} ({self.weighing_time})"
24 
```

4.1.2 Operazioni CRUD

Con la terminologia *CRUD* (*Create*, *Read*, *Update* e *Delete*) ci si riferisce alle quattro operazioni fondamentali per la gestione di un database. Django ha una *shell* interattiva, che verrà spiegata nel dettaglio nel paragrafo successivo, e che ci permette di poter accedere direttamente al DBMS con grande facilità.

```
(venv) marcotedeschini@marco MultiData % python manage.py shell
Python 3.9.7 (default, Oct 22 2021, 13:24:00)
[Clang 13.0.0 (clang-1300.0.29.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
>>> from apps.reports.models import Weighing
>>> import datetime
>>>
```

La prima delle quattro operazioni basilari è la *CREATE*: semplicemente si occupa della creazione o della aggiunta di nuovi record nel nostro set di dati. È importante notare come nell'esempio sottostante alla tredicesima riga l'*id* abbia il valore *None*: questo accade perché come abbiamo detto in precedenza il campo AutoField, genera l'*id* univoco solamente dopo che viene chiamata la funzione di *save()*. Infatti, nelle righe successive (15, 17) vediamo rispettivamente il salvataggio e la successiva assegnazione di tale valore.

```
>>> weigh = Weighing(
...         silo_number=21,
...         silo_name="S21",
...         delivery_note="01/01/2022",
...         component_code="PVC",
...         set_point=130000,
...         weighed=90000,
...         error=1,
...         weighing_time=datetime.datetime.now()
...     )
>>>
>>> weigh
<Weighing: None - 90000 (2022-01-24 11:40:50.595198)>
>>>
>>> weigh.save()
>>> weigh
<Weighing: 824 - 90000 (2022-01-24 11:40:50.595198)>
>>>
```

La *READ* è la seconda operazione. È importante non solo per la lettura dei dati, ma anche per la ricerca in essi, attraverso possibili filtri o per individuare informazioni preesistenti.

```
>>> Weighing.objects.all()
<QuerySet [<Weighing: 423 - 13720 (2020-07-17 12:25:03)>, <Weighing: 424 - 24423 (2020-09-04
11:30:01)>, <Weighing: 425 - 25734 (2020-01-25 16:05:00)>, <Weighing: 426 - 15201 (2020-04-07
15:21:06)>, '...(remaining elements truncated)...']>
>>>
```

Modificare i dati è molto facile, ed è *UPDATE* ad occuparsene. Su Django una volta modificati i valori basta richiamare la funzione *save()* ed il gioco è fatto.

```
>>> weigh.set_point = 125000
>>> weigh.save()
>>>
```

Per ultimo ho tenuto come esempio $DELETE$, che si può utilizzare con la funzione omonima e serve per l'eliminazione dei record.

```
>>> weigh.delete()
(1, {'reports.Weighing': 1})
>>>
```

4.2 Django CLI

La root del progetto, che prende il nome dall'Azienda, contiene lo script *manage.py*: una command-line utility che permette l'amministrazione di Django da shell, implementando vari comandi.

Analizziamo, nel dettaglio, i più impiegati.

Troviamo innanzitutto *dbshell*, che permette di accedere direttamente all'interfaccia command-line (CLI) del motore del database in uso; più precisamente si ha accesso diretto al comando psql, qualora si usasse PostgreSQL, mysql per MySQL e per MariaDB, sqlite3 per SQLite e sqlplus per Oracle.

Secondariamente rinveniamo a *startapp*, utile per la realizzazione di un nuovo pacchetto Django e che consente di inizializzarlo nella maniera corretta, in modo da essere produttivi nei tempi più rapidi e immediati possibili.

Dopodiché consideriamo *runserver*, il quale avvia un leggerissimo web server per default sulla porta 8000 e sull'indirizzo IP 127.0.0.1, sebbene sia possibile impostare entrambi in modo differente. Chiaramente, è assolutamente sconsigliato l'utilizzo in fase di produzione perché pensato per essere un semplice strumento di debug, privo di qualsiasi sistema di protezione o di ottimizzazione.

In seguito, esaminiamo *makemigrations*, in grado di creare nuove migrazioni sulla base dei cambiamenti avvenuti nei modelli. Genera, dunque, i comandi SQL per le app installate, idonei ad apportare al database le stesse modifiche applicate in precedenza ai modelli e non ancora parte di una migrazione.

Troviamo, poi, *sqlmigrate* che richiede come parametri il label di un pacchetto, i cui dati devono essere migrati, e il nome del file di migrazione; esso stampa a video la query SQL, nel dialetto corretto, per poter svolgere l'intera operazione.

Analizziamo successivamente *migrate*, il quale sincronizza lo stato del database con il set di modelli e migrazioni correnti, applicando al database le migrazioni prodotte dal comando *makemigrations*.

```
(venv) marcotedeschini@marco MultiData % python manage.py makemigrations
Migrations for 'reports':
  apps/reports/migrations/0001_initial.py
    - Create model Weighing
(venv) marcotedeschini@marco MultiData %
(venv) marcotedeschini@marco MultiData % python manage.py sqlmigrate reports 0001_initial
--
-- Create model Weighing
--
CREATE TABLE [Weighing] ([id] int IDENTITY (1, 1) NOT NULL PRIMARY KEY, [silo_number] int NOT
NULL, [silo_name] nvarchar(10) NOT NULL, [delivery_note] nvarchar(20) NULL, [component_code]
nvarchar(30) NOT NULL, [set_point] decimal(18, 0) NOT NULL, [weighed] decimal(18, 0) NOT
NULL, [error] decimal(18, 0) NOT NULL, [weighing_time] datetime2 NOT NULL);
(venv) marcotedeschini@marco MultiData %
(venv) marcotedeschini@marco MultiData % python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, reports, contenttypes, sessions
Running migrations:
  No migrations to apply.
(venv) marcotedeschini@marco MultiData %

```

Da ultimo troviamo *test*, efficace per compiere test automatizzati perfettamente integrati da un'apposita suite in Django.

4.3 Ciclo richiesta/risposta

All'interno della root del progetto è presente una cartella omonima contenente alcuni moduli per le impostazioni e una descrizione delle apps.

Più precisamente, nel modulo *settings.py*, troviamo una serie di impostazioni generali, relative alla connessione con i database; parallelamente sono presenti anche le impostazioni collegate ai vari plugin.

Nel caso posto in analisi in questo documento, all'interno di *settings.py*, vi sono numerose impostazioni relative alle classi del REST framework.

Nel dettaglio, a titolo esemplificativo, è stato inserito un limite al quantitativo di dati scambiati, grazie all'uso della pagination class. Al contrario, sono stati impostati opportunamente i filtri backend, per l'ottimizzazione delle operazioni di ricerca.

La mappatura delle app - la quale consente che esse siano raggiungibili dall'esterno attraverso un preciso path - è gestita interamente dal modulo url.py, che collega a un preciso URI ciascuna app.

Django supporta, inoltre, due diverse interfacce del web service con il codice.

La più comune è *WSGI* (*Web Server Gateway Interface*), che presenta un livello orientato al server o al gateway, frequentemente correlato all'esecuzione di un software come *Apache* o *Nginx*, e uno dal lato dell'applicazione, le cui API possono essere chiamate direttamente da Python. Più recentemente, si è iniziata a diffondere quello che, a tutti gli effetti, si configura come il successore naturale di WSGI: *ASGI*

(*Asyncronous Server Gateway Interface*), che aggiunge il supporto alle chiamate asincrone. Le impostazioni per queste API risiedono, rispettivamente, nei file *wsgi.py* e *asgi.py*.

Nella cartella apps è presente una sottocartella per ciascuna di esse, al cui interno troviamo i file che effettivamente costituiscono le logiche di serializzazione e deserializzazione dei dati, nonché dell'amministrazione dell'intero backend.

In verità, l'uso del Django Rest Framework (DRF), di cui si tratterà, richiede esplicitamente questa struttura per le cartelle dell'applicazione.

Come si evince dallo schema del funzionamento del ciclo di richiesta-risposta in un'applicazione per Django, il middleware richiesto dal DRF consiste di un paio di componenti: view e serializers.

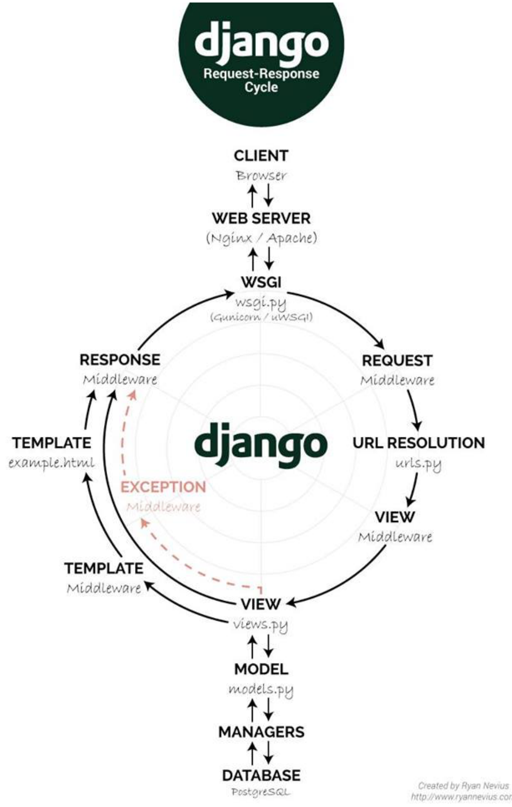


Fig. 4.4 Django ciclo di richiesta e risposta [9]

Le view, che troviamo nel modulo omonimo, sono funzioni che accettano una richiesta e offrono una risposta *HTTP*; è proprio compito loro gestire i reindirizzamenti e tutta quella serie di controlli sui permessi (in base per esempio agli header della richiesta) per ammettere o meno le query sui database.

In particolare, è possibile configurare in questa sede anche gli handler per le risposte in caso di errore: sono, infatti, presenti, funzioni di default come *page_not_found()*, *server_error()*, *permission_denied()* e *bad_request()*, che è possibile sovrascrivere per gestire nel modo desiderato, rispettivamente, i codici di stato 404 (not found,

risorsa non trovata), 500 (internal server error, errore del server), 403 (forbidden, richiesta che il server non deve soddisfare) e 400 (bad request, richiesta malformata).

I serializzatori, disponibili in serializers.py, sono oggetti che estendono la classe Serializer dentro il pacchetto *rest_framework.serializers* e che si occupano della conversione dei dati, per la comunicazione con il database, rendendone possibile il salvataggio in memoria e il raggruppamento in liste.

Sempre nel file serializers.py sono presenti le classi che implementano la visualizzazione dei risultati delle query come liste, estendendo la classe *rest_framework.generics.ListAPIView* e all'interno di cui sono definiti il set di oggetti su cui lavorare (tramite il parametro query_set, collegato a una query effettuata dal modulo model), i filtri di ricerca (*filter_fields*) e, per l'appunto, la classe del serializzatore (definita, per la verità, nello stesso file e assegnata al parametro *serializer_class*).

4.4 Django SQL Server per ARM

La ragione implementativa dietro all'utilizzo di Microsoft SQL Server è da ricercarsi nell'ampissima popolarità che ha tale DBMS tra le aziende del territorio.

La scelta di Django dipende dal fatto che, qualora l'Azienda volesse scegliere, in un più o meno prossimo futuro, una soluzione diversa per la base di dati, la migrazione sarà possibile a un minimo prezzo in termini di tempo e difficoltà.

Il driver sviluppato da Microsoft di MSSQL per Python è *pyodbc*[17], disponibile all'installazione tramite pip.

Il suo uso sarebbe stato, sì, quasi una ovvia scelta dal punto di vista implementativo, dato il supporto diretto da parte della stessa Azienda, Microsoft, che cura il DBMS, ma avrebbe, al tempo stesso, limitato oltremodo la portabilità del prodotto finale.

Scendendo nei dettagli, il driver di Microsoft si appoggia a ODBC (Open Database Connectivity), un'API standard per la connessione del client al DBMS, che però non è in alcun modo supportata sulle piattaforme *ARM*.

Credo, però, che precludere il funzionamento di una soluzione software al mondo ARM, in questo momento storico, non possa essere una scelta lungimirante.

ARM, infatti, negli ultimi anni sta vivendo un momento di fasti, sia per quanto riguarda il settore mobile, sia - soprattutto - per i dispositivi desktop e notebook prodotti da *Apple Inc.*. Sugli ultimi dispositivi, Apple ha montato nuovi processori di cui ha curato la progettazione e ne ha affidato la produzione alla *TMSC ltd.*, a

---

[17] https://github.com/mkleehammer/pyodbc

cominciare dall'*Apple Silicon APL1102*, commercializzato con il nome di *Apple M1*, un processore che ha saputo coniugare ottime prestazioni con consumi energetici minimi.

Inoltre, nella libreria di mssql-django mancano alcuni tipi di dato e strutture che vengono utilizzati nei database della ditta, ad esempio "numeric" oppure il "decimal identity (1,1)".

Per risolvere questi problemi, ho applicato alcune modifiche al progetto di Microsoft.

Al fine di permettere un utilizzo anche per dispositivi con architettura *AArch64* ho scelto di utilizzare un'interfaccia scritta in *Cython* sulle basi del driver *FreeTDS*[18]: *pymssql*[19]. Mentre, per l'aggiunta dei "field" mancanti, ho applicato modifiche direttamente sull'*ORM*.

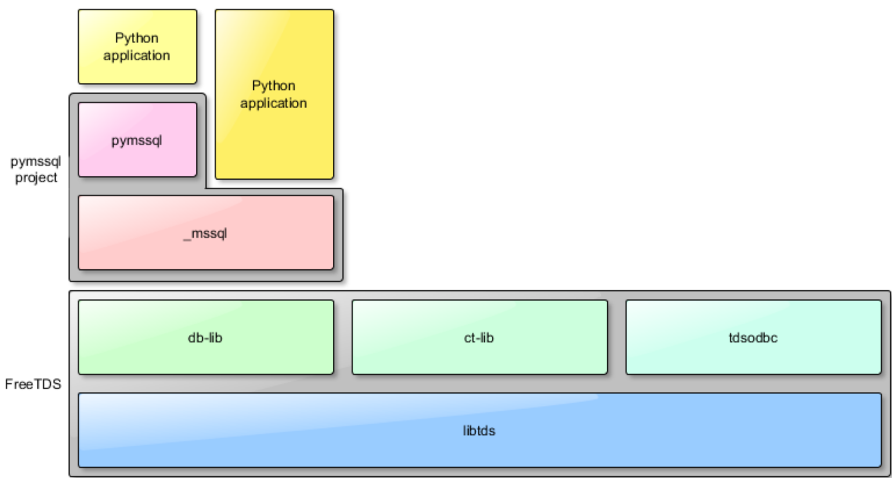


Fig. 4.5 Architettura *pymssql*: composto dai moduli: *pymssql* e *_mssql* [10]

In accordo con l'Azienda, ho effettuato una fork del pacchetto di Django (*django-mssql-arm*[20]) che si occupa dell'integrazione con MSSQL, sostituendo a pyodbc il driver pymssql, che, sotto la scocca, non usa ODBC, ma FreeTDS, che effettivamente supporta a pieno l'architettura ARM.

La libreria è attualmente disponibile su GitHub e ha le stesse compatibilità di pymssql supportando i sistemi: *Linux*, *macOS* e *Windows*. Oltre ciò è stata testata con

[18] https://www.freetds.org

[19] https://pymssql.readthedocs.io

[20] https://github.com/tede12/django-mssql-arm

successo sulle piattaforme Windows con architettura x86_64 e macOS sia eseguito su arm64 sia su x86_64.

Per aggiungerla a un nuovo progetto Django, è necessario soltanto, dopo averla "clonata", apporre una modifica al file di impostazioni, come segue nell'esempio.

settings.py

```
DATABASES = {
    "default": {
        "ENGINE": "mssql_arm",   #
        "HOST": "localhost",
        "NAME": "database",
        "USER": "sa",
        "PASSWORD": "MyPass@word",
        "PORT": 1433,
    },
}

```

Django-mssql-arm trae le sue origini dallo stesso pacchetto di base da cui i backend di default degli altri DBMS supportati nativamente da Django ereditano le proprietà ed estendono le funzioni.

Il modulo *base.py* contiene la classe *DatabaseWrapper*, che agisce come un wrapper per le API della connessione al database.

In questo modulo sono creati sia la nuova connessione (andando a prendere i parametri giusti dalle impostazioni globali), sia il nuovo cursore; sono definiti, inoltre, i data type secondo le modalità già trattate nel paragrafo a riguardo dei Fields supportati da Django.

Fra i moduli troviamo *client.py*, che tratta l'integrazione con la shell (nel dettaglio, il comando per avviare la suite di amministrazione di un DB MSSQL è sqlcmd, disponibile per più sistemi operativi).

Individuiamo inoltre *consts.py*, che contiene unicamente delle costanti, utili per adattare il codice alle impostazioni proprie del DBMS.

A seguire vediamo come *creation.py* sia il modulo che si occupa della creazione dei database; nel mio caso è stato utile per automatizzare la creazione e, successivamente, la distruzione dei database provvisori ai fini di testing all'interno della suite di Django.

features.py, invece, è un modulo che racchiude un elenco di funzionalità generalmente previste dai DBMS. Flaggando vera o falsa l'una o l'altra tra queste, si comunicano a ORM le features disponibili o meno nel DBMS su cui sta, nell'immediato, operando.

Procedendo, troviamo il modulo *fields.py*, non presente nel pacchetto di base, ma indispensabile per l'integrazione di alcuni fields che sono supportati da MSSQL e che erano necessari per il progetto, quale *nchar* (la rappresentazione dei caratteri *Unicode*), ma che non erano già presenti tra i field "standard".

functions.py, un altro modulo, implementa una serie di funzioni utili a garantire la compatibilità tra i vari DBMS, fungendo da wrapper per le varie funzioni SQL. Troveremo, la funzione logaritmo naturale che effettivamente ritornerà un collegamento all'istruzione SQL che calcola la medesima operazione algebrica, mentre, per la funzione modulo, il template ritornato consisterà nel calcolo del modulo come resto della divisione del primo valore per il secondo, mancando un modo per esprimere il medesimo concetto con un unico operatore in MSSQL.

Sempre tra i file evidenziamo *introspection.py*, il quale si occupa della mappatura dei codici dei tipi ai field specificati in Django.

Infine, *schema.py* è un modulo relativo alla semplificazione della costruzione degli scheletri delle query, dalle più semplici alle più complesse. Esso è indispensabile per la costruzione degli schemi su cui poi saranno assemblate le varie query di creazione, selezione, aggiornamento ed eliminazione di dati. Per ciascuna di queste, però, occorre che lo schema sia sufficientemente elastico da coprire tutti i possibili casi d'uso (per esempio, deve essere in grado di creare query capaci tanto di mutare il contenuto di una colonna, quanto il tipo dei dati al suo interno).

Capitolo 5

Frontend

5.1 Interfaccia con React

5.1.1 Component

Vediamo ora come avviene la creazione di una applicazione React.

Innanzitutto, si parte dai componenti, che si possono vedere come dei blocchi di costruzione.

La parte interessante di tutto ciò è che ogni "blocco" si può comporre con altri e creare parti di UI indipendenti e riutilizzabili.

Concettualmente i componenti sono implementati come funzioni JavaScript [4].

Nel seguente modo ho definito una funzione JavaScript che verrà chiamata, per l'appunto, "functional component".

```
1  function Greeting(props) {
2      return <h1>Hello {props.name}</h1>
3  };
4
```

Lo stesso componente può essere scritto con una classe *ES6*[21] ("class component").

```
1  import React from "react";
2
3  class Greeting extends React.Component {
4      render() {
5          return <h1>Hello {props.name}</h1>
6      }
7  };
8
```

Un altro modo per dichiarare il componente *Greeting*, usando un'*arrow function*, è il seguente:

```
1  // written using ES6's arrow function
2  const Greeting = (props) => <h1>Hello {props.name}</h1>;
3
```

[21] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

React mette a disposizione dei link *CDN*[22] per poter lavorare direttamente su un "template" *html*. Non tutte le librerie aggiuntive dispongono di link CDN.

Con React l'uso dei CDN è possibile, ma sono veramente pochi i casi in cui questo sia effettivamente conveniente o indispensabile. I CDN, infatti, aggiungono notevole complessità al progetto e – per quanto questo sia solo raramente un problema – rendono imprescindibile l'accesso a internet per recuperare le risorse esterne.

Inoltre, i file recuperati da remoto non sono ottimizzati nella chiave del programma, né sono garantiti sistemi di *pre-caching*.

Nondimeno, sfruttando risorse non già presenti nel progetto, si aggiunge a quest'ultimo un ulteriore *point of failure*: nonostante spesso si possa fare affidamento su queste librerie, i CDN non sono infallibili. In caso di malfunzionamenti resta ben poco da fare se non attendere che il problema venga risolto.

Occorre notare poi, che in tutti i casi in cui la sicurezza è una priorità, non è consigliabile l'uso di risorse esterne. I CDN in questa circostanza, dal momento che sarebbero in grado di raccogliere dati sull'utente o sul sistema, si rivelerebbero facilmente un'arma a doppio taglio, esponendo i dati sensibili a servizi esterni.

Nel prossimo esempio[23] è mostrato, a scopo didascalico, l'uso dei CDN per l'inserimento dello stesso componente visto in precedenza all'interno di un documento *HTML*.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <title>Get Started with React</title>
6  </head>
7
8  <body>
9  <div id="root">
10 </div>
11 <!-- React CDN Links (https://reactjs.org/docs/cdn-links.html) -->
12 <script src="https://unpkg.com/@babel/standalone/babel.js"></script>
13 <script crossorigin src="https://unpkg.com/react/umd/react.development.js"></script>
14 <script crossorigin src="https://unpkg.com/react-dom/umd/react-dom.development.js"></script>
15
16 <script type="text/babel">
17     // My component
18     const Greeting = (props) => <h1>Hello {props.name}</h1>;
19     ReactDOM.render(
20         <Greeting name="World!"/>,
21         document.getElementById('root')
22     );
23 </script>
24 </body>
25 </html>
26
```

[22] Content Delivery Network consente di distribuire le risorse a un gruppo di server che si trovano in diverse aree geografiche e che collaborano per la trasmissione dei contenuti.

[23] https://codepen.io/marco-tedeschini/pen/jOaOWGv

Output:

Hello World!

5.1.2 Bundle

È consigliato utilizzare *Webpack* e *Babel* per la fase di "compilazione" (transpile).

Webpack si occupa principalmente di ottimizzare le prestazioni; nel dettaglio:

- Può creare un singolo bundle o più blocchi (detti *chunk*) che vengono caricati in modo asincrono a *runtime* (per ridurre il tempo di caricamento iniziale).
- Le dipendenze vengono risolte durante la traduzione, riducendo le dimensioni finali del codice.
- I *loaders* possono pre-elaborare i file durante la compilazione, ad es. *TypeScript* in JavaScript, stringhe *Handlebars* in funzioni compilate, immagini in *Base64*, ecc.

L'obiettivo iniziale di Babel era quello di convertire il codice *ECMAScript 2015+* (*ES6+*) in una versione di JavaScript, compatibile con quelle precedenti, che potesse essere eseguita da motori JavaScript meno recenti.

Al giorno d'oggi Babel è utilizzato per il transpile delle sintassi, derivate da JS, ma non retrocompatibili, come *JSX* e *TypeScript*. Ad esempio, le *arrow function* – specificate in *ES6* – sono convertite in normali dichiarazioni di funzione.

Inoltre, Babel fornisce *polyfill* per le funzionalità che mancano completamente dagli ambienti JavaScript, come i metodi statici quali *Array.from()* e alcuni *built-in* (ad esempio *Promise*), disponibili solo in ES6+.

Per proseguire, bisogna spiegare a cosa ci si riferisca quando si parla di *DOM* (*Document Object Model*): consiste in un'API per documenti HTML e XML che ne definisce la struttura logica secondo un modello orientato agli oggetti. Sempre all'interno del *DOM* sono definiti il modo in cui si accede ai nodi e come si manipola un documento.

In React un insieme di valori immutabili è passato al render dei componenti come proprietà nei tag HTML. I *component* non possono modificare direttamente alcuna proprietà, tuttavia, riescono a passare una funzione di *callback* tramite la quale possiamo effettuare alcune modifiche. Questo intero processo è noto con la locuzione "properties flow down; actions flow up": le proprietà, infatti, scendono di livello di component in component, mentre gli eventi seguono il flusso inverso [11].

ReactDOM è un pacchetto che fornisce metodi che possono essere utilizzati per interagire con il DOM. È necessario per inserire o aggiornare elementi React [12].

Fornisce molte funzioni di supporto, le cui principali sono:

- *render(element, container[, callback])*

  Questa funzione viene utilizzata per renderizzare gli elementi React all'interno del DOM. Viene fornita, per l'appunto, la possibilità di usare una *callback* per eseguire codice dopo che è avvenuta la renderizzazione.

- *hydrate(element, container[, callback])*

  È molto similare a render; ma, mentre vediamo come quest'ultimo cambi il nodo soltanto qualora vi fosse una differenza tra il DOM iniziale e quello corrente, *hydrate()* – chiamata su nodi che hanno un markup *server-rendered* – prova a collegare i listener degli eventi al nuovo nodo.

React crea una cache della struttura dati in memoria, la quale individua le modifiche apportate e, solo allora, aggiorna il browser. Ciò consente al programmatore di pensare il codice come se l'intera pagina fosse renderizzata a ogni modifica, ma in verità la libreria renderizza solo e unicamente i componenti che effettivamente cambiano [11].

Questo è ottenuto grazie alla creazione in memoria di un *VirtualDOM*, che è mappato sul DOM reale ma da cui quest'ultimo non riceve eventi se non le effettive modifiche.

Nel prossimo esempio è mostrato la renderizzazione di un componente attraverso il DOM di React[24].

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3
4  const Greeting = (props) => <h1>Hello {props.name}</h1>;
5
6  ReactDOM.render(
7      <Greeting name="World!"/>,
8      document.getElementById('root')
9  );
10
```

Output:

Hello World!

React offre un ottimo sistema di composizione che permette una certa ereditarietà tra i componenti, aumentando le possibilità del riuso del codice. Spesso occorre restituire

[24] https://codepen.io/marco-tedeschini/pen/yLPLeKZ

molteplici elementi per un singolo componente: per fare ciò esiste il component *Fragment*[25], talmente usato che lo stesso standard di React ne prevede una sintassi breve, caratterizzato da un tag privo del nome. Questa scelta è molto comoda nello sviluppo anche se non supporta né chiavi né attributi e, al contrario, l'uso di *Fragment* ammette "key" come unico attributo.

```
function App() {
    return (
        <>
            <Greeting name="World!"/>
            <Greeting name="Foo!"/>
            <Greeting name="Bar!"/>
        </>
    );
}

ReactDOM.render(
    <App/>,
    document.getElementById('root')
);

```

Output:

Hello World!

Hello Foo!

Hello Bar!

Nell'esempio sottostante invece si vuole mostrare come avviene la scrittura del codice relativo all'innestamento di più component. *Greeting* è utilizzato in due contesti diversi, nel primo caso innestato al component *Container*, che semplicemente aggiunge un bordo tratteggiato intorno al testo degli elementi figli; nel secondo, è usato come da esempi precedenti[26].

```
const Container = (props) => <div style={{border: "1px solid black"}}>
    {props.children}
</div>

ReactDOM.render(
    <>
        <Container>
            <Greeting name="World!"/>
        </Container>
        <Greeting name="World!"/>
    </>,
    document.getElementById('root')
);

```

Output:

[25] https://codepen.io/marco-tedeschini/pen/yLPLOeB

[26] https://codepen.io/marco-tedeschini/pen/qBVBNYg

Hello World!

Hello World!

È importante sottolineare che per la realizzazione di un nuovo progetto React è indispensabile utilizzare il pacchetto *CRA* (*create-react-app*[27]). Si occupa di fornire molteplici template per la scrittura di un'applicazione React e di configurare autonomamente i tool Webpack, Babel ed ESLint[28] da utilizzare per la fase di transpile.

Allo stesso tempo grazie alla tecnologia di *webpack-dev-server*[29] questo strumento fornisce un *development server* per testare comodamente il progetto su un browser nella rete locale (*localhost*).

5.2 Logica degli stati con Redux

Ora analizzeremo nel dettaglio come si lavora con Redux.

Prima di tutto è bene definire quale sia lo standard da seguire per la struttura dei file (*file structure*). È possibile anche in questo caso fare uso del pacchetto CRA che fornisce un pratico template ("--template redux") di partenza per un progetto che vede impiegati ReactJS e Redux.

Per rendere più chiara la separazione tra il *presentation layer* (React) e il *logic state layer* (Redux), ho creato una sottocartella, in *src*, chiamata "backend". Questa si riferisce al backend del client e conterrà, per l'appunto, il codice di Redux.

All'interno di questa *directory* sono presenti cinque file che rappresentano i punti chiave sopra citati. Di seguito sono mostrati, a titolo esemplificativo, degli esempi di codice.

ActionTypes.js contiene la mappatura di tutte le azioni che possono essere eseguite sul reducer che all'occorrenza in uno switch-case sceglierà dove dispacciarle.

---

[27] https://create-react-app.dev

[28] Strumento che si occupa dell'analisi di codice statico JavaScript per la rilevazione di errori critici.

[29] https://webpack.js.org/configuration/dev-server

src/backend/ActionTypes.js

```
export const WEIGHING_FETCHING                              = 'WEIGHING/FETCHING'
export const WEIGHING_FETCH_SUCCEEDED                       = 'WEIGHING/FETCH_SUCCEEDED'
export const WEIGHING_FETCH_FAILED                          = 'WEIGHING/FETCH_FAILED'

```

RootReducer.js: l'approccio più comune nella scrittura della logica di un riduttore è di avere più funzioni "slice reducer" ognuna responsabile dei suoi aggiornamenti della propria specifica porzione di stato.

I riduttori possono rispondere alla stessa azione, aggiornare in modo indipendente la propria sezione secondo le loro necessità e, una volta aggiornate, le *slice* vengono combinate nel nuovo oggetto di stato.

Tale modello è così comune, che Redux fornisce il tool *combineReducers* per implementare questo comportamento, esso accetta come parametro un oggetto di "slice reducer" e restituisce una nuova funzione di "riduzione".

src/backend/RootReducer.js

```
import {combineReducers} from "redux";
import {connectRouter} from 'connected-react-router'
import WeighingReducer from "./WeighingReducer";

// root of all reducers
const createRootReducer = (history) => combineReducers({
    router: connectRouter(history),
    //... can add more reducers
    WeighingReducer
})
export default createRootReducer;

```

WeighingReducer.js è invece un esempio di "slice reducer".

src/backend/WeighingReducer.js

```
import * as t from './ActionTypes'
let initialState = {
    loading: false,
    weighing: []
};

const reducer = (state = initialState, action = {}) => {
    switch (action.type) {
        case t.WEIGHING_FETCH_SUCCEEDED:
            return {...state, loading: false, weighing: action.payload.records}
        case t.WEIGHING_FETCHING:
            return {...state, loading: true, weighing: []}
        case t.WEIGHING_FETCH_FAILED:
        default:
            return state; // do nothing
    }
};

export default reducer;

```

Store.js configura lo store e può caricare delle estensioni (Thunk e Saga), che in genere vengono scelte o create in base al middleware che si utilizza.

src/backend/Store.js

```
import {createBrowserHistory} from 'history'
import {applyMiddleware, compose, createStore} from 'redux'
import {routerMiddleware} from 'connected-react-router'
import thunk from 'redux-thunk'
import createRootReducer from './RootReducer'
export const history = createBrowserHistory()

// Connect our store to the reducers
export default function configureStore(preloadedState) {

    const store = createStore(
        createRootReducer(history),
        preloadedState,
        compose(
            applyMiddleware(
                routerMiddleware(history), // for dispatching history actions
                thunk,
            ),
        ),
    )
    // initSaga();
    return store
}

```

WeighingActions.js è l'ultimo file che troviamo nella cartella. Qui il codice:

src/backend/WeighingActions.js

```
import * as t from './ActionTypes'
import axios from 'axios';

export function getWeighing(PROD_ID) {
    return (dispatch, getState) => {
        dispatch({type: t.WEIGHING_FETCHING});

        // used for development
        axios.get(`http:/localhost/api/weighing/?product_id=${PROD_ID}`)
            .then(response => doThingsWithData(response))
            .then(response => {
                dispatch({type: t.WEIGHING_FETCH_SUCCEEDED, payload: response})
            })
            .catch(error => {
                dispatch({type: t.WEIGHING_FETCH_FAILED})
            });
    }
}

```

Per ultimo, mostro come avviene il collegamento tra il backend del nostro client (basato sulla tecnologia Redux) e la presentazione in React. È chiaro come le azioni, una volta "dispacciate", producano nuovi stati che vengono mappati in proprietà del componente.

src/App.react.js

```
import React from "react";

/* backend client */
import {connect} from "react-redux";
import {getWeighing} from "../backend/WeighingActions"

export default class App extends React.Component {
    componentDidMount() {
        this.props.getWeighing()
    }
    render() {
        if (this.props.loading)
            return (<span>Loading in progress</span>)
        return (<span>Number of weighs: {this.props.weighing.length}</span>)
    }
}

const mapStateToProps = state => {
    return {
        loading: state.WeighingReducer.loading,
        weighing: state.WeighingReducer.weighing,
    }
}
const mapDispatchToProps = {
    getWeighing
}

export default connect(mapStateToProps, mapDispatchToProps)(App);
```

Se il progetto aumenta di dimensioni, è convenzione suddividere in sottocartelle i file della cartella "backend" per una vista migliore del *backend-client*.

Capitolo 6

Esperimento finale

6.1 Descrizione dell'esperimento

Una volta terminato il progetto per il tirocinio ho voluto approfondire ulteriormente, le informazioni che avevo imparato e studiato. Ho per questo deciso di completare il mio percorso con dei test riguardanti le modifiche apportate al software di Microsoft per l'integrazione di SQL Server con Django.

Trattandosi propriamente di un Object Relational Mapping, ho voluto confrontare i due backend analizzando le performance delle operazioni fondamentali CRUD.

I risultati ottenuti li ho effettuati su un computer con sistema operativo macOS con architettura *Intel* a 64 bit. Per l'esperimento ho caricato il DBMS su Docker con l'immagine di Microsoft SQL Server 2019[30] ed ho creato un comando custom su Django[31] utile per la distribuzione.

[30] https://hub.docker.com/_/microsoft-mssql-server

[31] https://docs.djangoproject.com/en/4.0/howto/custom-management-commands/

6.2 Risultati

Ecco i risultati del benchmark che ho effettuato e di cui i valori ho presentato nella tabella sottostante. I test sono stati effettuati, rispettivamente, per 500, 1000, 1500, 5000, 10000 e 20000 righe.

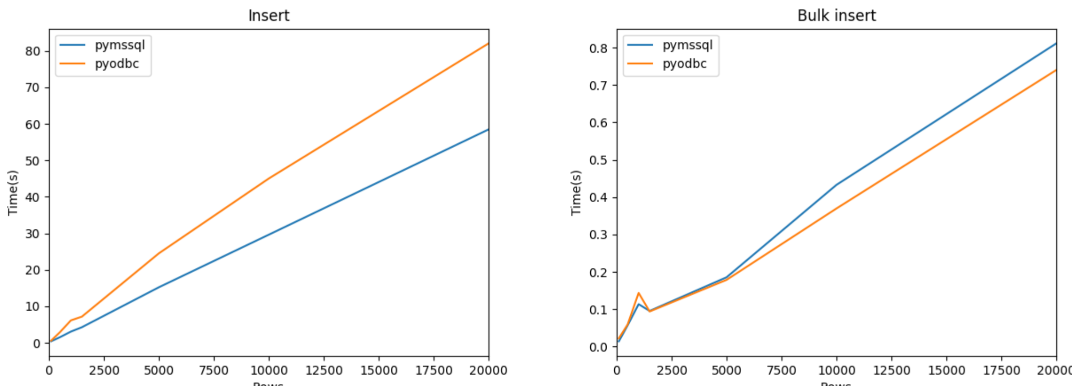


Fig. 6.1 Insert e Bulk Insert

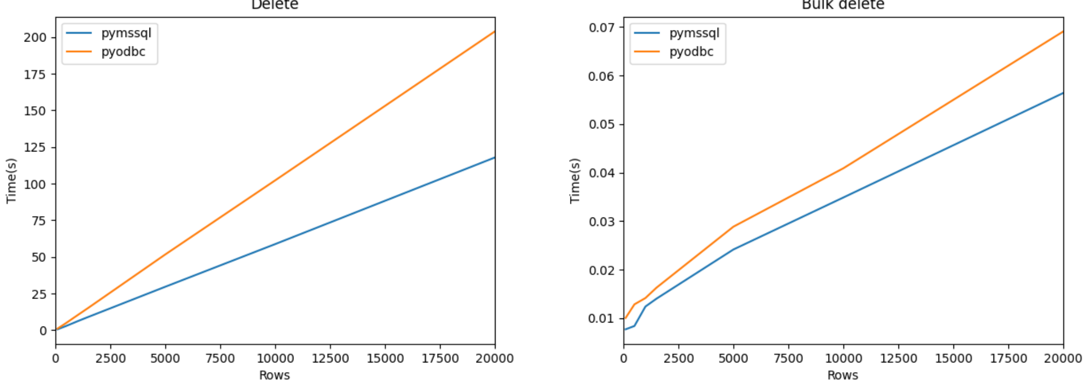


Fig. 6.2 Delete e Bulk Delete

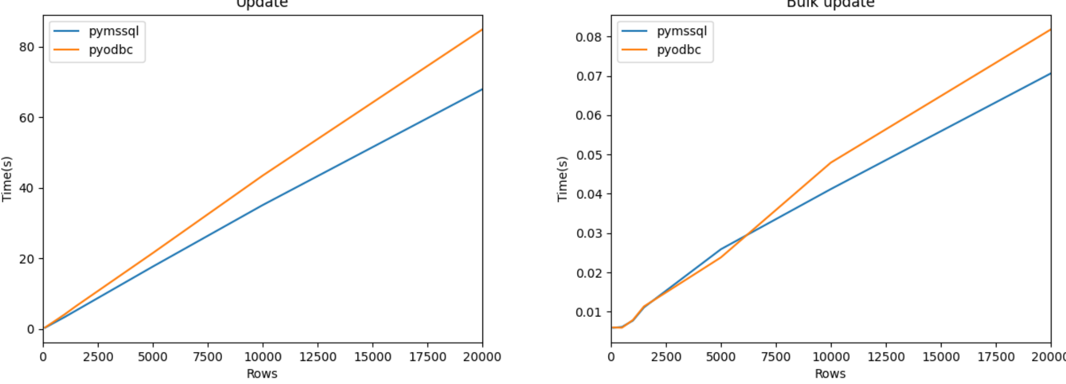


Fig. 6.3 Update e Bulk Update

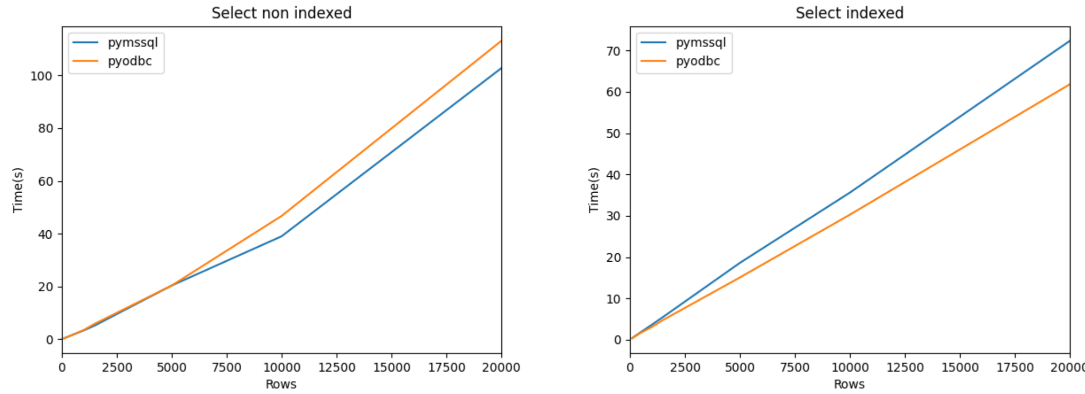


Fig. 6.4 Select e Select Indexed

Analizzando i dati, emerge che complessivamente pymssql riporta risultati migliori in particolare nelle operazioni semplici. In queste ultime, infatti, è evidente che la mancanza di un layer (ODBC) renda il codice più rapido. Quanto più la stessa operazione viene ripetuta, tanto più questo tempo inficerà la durata dell'esecuzione.

I risultati più eclatanti sono riscontrabili, più precisamente, nella ripetizione di singole *delete*, dove il tempo di esecuzione di pymssql è più veloce, in ogni test effettuato, di circa il 70%.

Risultati simili si riscontrano anche nelle *insert*, dove, però, il grafico è molto meno lineare: la differenza in percentuale aumenta fino alle 1000 query di inserimento, dove pymssql dimezza i tempi di esecuzione di pyodbc, per poi vedere la forbice assottigliarsi nei test successivi.

Pyodbc offre però una ottima integrazione con i thread del DBMS, infatti nelle operazioni *bulk* (dove, cioè, le query sono processate in parallelo) riporta tempi leggermente migliori sui volumi di dati maggiori, quando l'onerosa creazione dei thread è compensata dalla loro efficienza.

Per quanto riguarda le selezioni indicizzate, pyodbc riporta tempi più rapidi, probabilmente dovuti a una migliore implementazione del supporto a tale tipologia di query.

Nel dettaglio, i dati raccolti sono stati i seguenti:

Legenda	
scarto percentuale > **50%**	
25% < scarto percentuale ≤ **50%**	
0% ≤ scarto percentuale ≤ **25%**	

PYMSSQL	100	501	1000	1500	5000	1000	20000
INSERT	0,3992	1,5147	3,0597	4,2575	15,2018	29,6158	58,4723
INSERT BULK	0,0143	0,0570	0,1137	0,0956	0,1855	0,4328	0,8112
DELETE	0,6213	2,9383	6,0667	9,0643	29,6221	58,7147	117,8011
DELETE BULK	0,0077	0,0084	0,0124	0,0140	0,0242	0,0349	0,0564
UPDATE	0,3499	1,7149	3,4022	5,1923	17,6674	35,1007	67,9372
UPDATE BULK	0,0059	0,0061	0,0077	0,0111	0,0259	0,0412	0,0706
SELECT	0,3425	1,7089	3,3362	5,2002	20,3558	39,0009	102,7149
SELECT INDEXED	0,3644	1,8320	3,6269	5,4623	18,5850	35,5954	72,3342

Fig. 6.5 Tabella raffigurante il tempo (s) delle operazioni svolte con *pymssql*

PYODBC	100	500	1000	1500	5000	10000	20000
INSERT	0,5799	2,8832	6,1336	7,1567	24,5000	45,0105	82,0485
INSERT BULK	0,0224	0,0593	0,1438	0,0941	0,1787	0,3696	0,7400
DELETE	1,1073	5,0211	10,1602	15,2595	51,6549	102,1771	203,7903
DELETE BULK	0,0101	0,0129	0,0142	0,0163	0,0289	0,0409	0,0691
UPDATE	0,4232	2,0703	4,1775	6,4361	21,5054	43,4564	84,8792
UPDATE BULK	0,0060	0,0059	0,0079	0,0113	0,0238	0,0479	0,0818
SELECT	0,3136	1,6872	3,5043	5,8821	20,3421	46,7149	113,0616
SELECT INDEXED	0,3047	1,6795	3,0656	4,7322	15,0539	30,2514	61,8203

Fig. 6.6 Tabella raffigurante il tempo (s) delle operazioni svolte con *pyodbc*

6.3 Conclusioni e lavoro futuro

Ho trovato appassionante l'idea di effettuare una fork per garantire un futuro nel mondo ARM a un progetto interessante come può essere pymssql.
Penso che la libreria possa ambire a un futuro utilizzo in altri lavori, anche se alcune fasi dello sviluppo non possono dirsi concluse.
Occorrerà un'attenta analisi dei dati riportati, per migliorare sia le operazioni concorrenti, sia l'implementazione delle selezioni indicizzate.
Per gestire, comunque, database con volumi di dati non enormi e i cui accessi non siano molto frequenti e, soprattutto, difficilmente occorrano inserimenti di grandi volumi di dati, le prestazioni non solo sono sufficienti, ma possono essere considerate un punto di forza.

Capitolo 7
Conclusioni

Grazie a questo tirocinio ho avuto la possibilità di assistere per la prima volta ad un ambiente lavorativo e di misurarmi con esso, lasciandomi travolgere, imparando e scoprendo tutte le mie risorse e i miei punti di forza.

Ho saputo giocare le mie carte migliori, buttandomi in un'esperienza lavorativa per me nuova, con l'intento di volerla vivere appieno, in modo da arricchire il mio bagaglio personale, ricavando le più disparate e migliori conoscenze e, soprattutto, competenze.

Ho riscontrato un ambiente dinamico, aperto al dialogo, pronto ad accogliermi, e colleghi disponibili al confronto, in grado di colmare ogni mio dubbio o difficoltà.

Ho apprezzato la disponibilità offertami dall'Azienda MultiData, motore fondamentale di questo tirocinio, di poter svolgere un lavoro di ricerca e di comparazione, ma anche la possibilità di decidere in piena libertà - sotto l'indispensabile supporto e sostegno dei tecnici - le tecnologie da utilizzare. Questo ha fatto sì che potessi approcciarmi a nuovi linguaggi e a nuove librerie, in modo da arricchire le mie conoscenze sostanziosamente.

Vorrei spendere un encomio anche per i colleghi, che hanno saputo accogliermi e darmi preziosissimi consigli, mostrandosi sempre disponibili e pazienti dinanzi le mie richieste.

Il prodotto finale è risultato in linea con le aspettative dell'Azienda e mie; in particolare, ho apprezzato l'aver potuto investire questo tempo di studio sull'approfondimento di Django - che ritengo tutt'ora una scelta vincente, per la praticità d'uso e le buone prestazioni - e sullo studio delle librerie JavaScript React e Redux.

Queste ultime, infatti, in questo nostro mondo sempre in continua e incessante evoluzione, stanno conquistando una fetta di mercato sempre più ampia; il loro utilizzo, nel prossimo futuro, coinvolgerà sia lo sviluppo web - da cui compiono i primi passi e per cui vedono il loro esordio - sia quello desktop, per la creazione di interfacce grafiche semplici ma responsive, grazie a librerie come Electron.

Anche la creazione di una fork in ambito di produzione è stata un'esperienza certo rischiosa ma gratificante: confrontarmi a basso livello con le dinamiche di un driver, mi ha permesso di arrivare a conoscere e approfondire meglio i sistemi di interconnessione tra il software e il database.

Grazie a ciò, ora, sono pienamente supportate le architetture ARM e, oltre al frontend, si potrà includere anche il backend su un dispositivo con la medesima architettura.

Si tratta di un prodotto - dunque di un software - robusto, affidabile e leggero, ma al contempo pratico, semplice nell'utilizzo e, tutto sommato, facile da aggiornare.

Bibliografia

[1] «MultiData,» [Online]. Available: https://www.multidata.it.

[2] S. ACM, «Data Mining Curriculum,» 2006.

[3] Tibco, «What is Data Mining?,» [Online]. Available: https://www.tibco.com/reference-center/what-is-data-mining.

[4] «ReactJS,» [Online]. Available: https://reactjs.org.

[5] J. Potter, «npm trends,» [Online]. Available: https://www.npmtrends.com/@angular/core-vs-react-vs-vue-vs-jquery.

[6] B. Cameron, «A beginner's guide to Redux with React,» 19 06 2019. [Online]. Available: https://medium.com/@bretcameron/a-beginners-guide-to-redux-with-react-50309ae09a14.

[7] D. Polzer, «Django for Data Scientists,» 7 02 2021. [Online]. Available: https://towardsdatascience.com/a-beginners-guide-to-using-djangos-impressive-data-management-abilities-9e94efe3bd6e.

[8] M. McKeay, «API: The Attack Surface That Connects Us All,» 10 2021. [Online]. Available: https://www.akamai.com/content/dam/site/en/documents/state-of-the-internet/soti-security-api-the-attack-surface-that-connects-us-all.pdf.

[9] R. Nevius, «Django request/response cycle,» [Online]. Available: https://www.ryannevius.com.

[10] p. developers, «pymssql,» [Online]. Available: https://pymssql.readthedocs.io/en/latest/intro.html.

[11] H. Patel, «React JS: What and Why?,» 26 09 2020. [Online]. Available: https://harsh-patel.medium.com/react-js-what-and-why-e6cad2dfb4c3.

[12] K. Pirzada, «A Closer Look at ReactDOM.render,» 22 01 2020. [Online]. Available: https://www.newline.co/@KumailP/a-closer-look-at-reactdomrender-the-need-to-know-and-more--891fed64.

[13] F. Falciani, «object-relational-mapping,» [Online]. Available: https://psicografici.com/object-relational-mapping/.

[14] K. McLaughlin, «An introduction to the Django ORM,» 24 11 2017. [Online]. Available: https://opensource.com/article/17/11/django-orm.