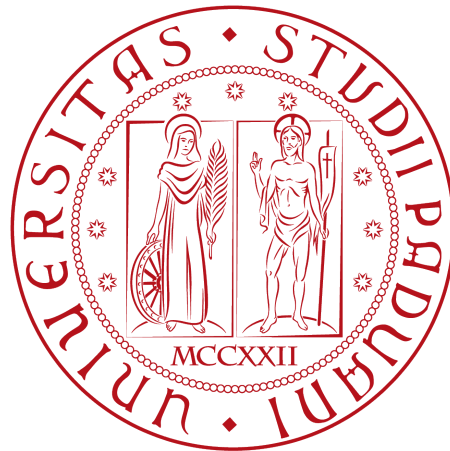
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Analisi, progettazione e sviluppo di un sistema di reportistica di un order management system

Tesi di laurea triennale

Relatore

Prof. Francesco Ranzato

Laureando

Marco Volpato

Anno Accademico 2022-2023

Marco Volpato: *Analisi, progettazione e sviluppo di un sistema di reportistica di un order management system,* Tesi di laurea triennale, © Settembre 2023.

Computer science is no more about computers than astronomy is about telescopes.

— Edsger W. Dijkstra

Sommario

Il seguente documento descrive il lavoro svolto al termine del percorso di studi della laurea triennale in Informatica, presso l'azienda Moku s.r.l. con sede a Treviso. Durante il periodo di stage, è stata svolta l'analisi, progettazione e realizzazione di un sistema di reportistica integrato in un OMS già esistente ed in continua evoluzione. Le funzionalità principali comprendono il filtraggio dei dati su base temporale e l'esportazione dei report generati in formato compatibile con Excel.

Ringraziamenti

Ringrazio i miei genitori, per tutti gli sforzi e sacrifici sostenuti per permettermi di arrivare a questo obiettivo.

Ringrazio i miei amici, tutti i miei familiari, anche chi non c'è più, per avermi sostenuto ed incoraggiato.

Ringrazio il prof. Francesco Ranzato per il supporto fornitomi durante la stesura del presente elaborato.

Infine, desidero ringraziare i miei colleghi di Moku per l'esperienza vissuta.

Padova, Settembre 2023 Marco Volpato

Indice

1 Introduzione **1**
1.1 Azienda . . . 1
1.2 Scopo . . . 1
1.3 Preventivo iniziale . . . 2

2 Progetto **3**
2.1 Piattaforma preesistente . . . 3
2.2 Obiettivi del progetto . . . 3
2.3 Metodo di lavoro . . . 4
2.4 Strumenti a supporto . . . 5
2.5 Tecnologie di sviluppo . . . 7

3 Analisi dei requisiti **9**
3.1 Architettura preesistente . . . 9
3.2 Utilizzatori . . . 10
3.3 Funzionalità . . . 10
3.4 Struttura di un report . . . 10
3.5 Tipologie di report . . . 11
3.6 Documenti di vendita . . . 11

4 Progettazione **12**
4.1 Gestione dei documenti di vendita . . . 12
4.1.1 Acquisizione . . . 12
4.1.2 Persistenza . . . 12
4.2 Definizione dei report . . . 13
4.3 Generazione dei report . . . 13
4.3.1 Performance e pre-aggregazione . . . 13
4.3.2 Scelta della libreria a supporto . . . 13
4.4 Interfacciamento con il frontend . . . 14

5 Realizzazione **15**
5.1 Modelli . . . 15
5.1.1 Documenti di vendita . . . 15
5.1.2 Report . . . 20
5.2 Definizioni delle tipologie di report . . . 21
5.2.1 DSL specifico . . . 21
5.2.2 Definition . . . 21
5.2.3 Registro . . . 22
5.3 Servizi . . . 23

5.3.1 FastExcelBackend 23
5.3.2 Generator 24
5.4 Controller 26
5.4.1 SaleDocumentsController 26
5.4.2 GeneratedExcelsController 26
5.5 GraphQL 28
5.5.1 Query 28
5.5.2 Mutation 28

6 Conclusioni **30**
6.1 Consuntivo finale 30
6.2 Raggiungimento degli obiettivi 31
6.3 Conoscenze acquisite 31
6.4 Valutazione personale 31

A Gestione paperless delle spedizioni **32**
A.1 Introduzione 32
A.2 Progettazione e realizzazione 32
A.2.1 Configurazione 32
A.2.2 Classi implementate 33
A.2.3 Scheduling 35

B Ulteriori listati di codice **36**
B.1 Generazione report 36
B.2 GraphQL 39

Acronimi e abbreviazioni **41**

Glossario **42**

Bibliografia **47**

Elenco delle figure

1.1 Logo dell'azienda 1

2.1 Logo di Bitbucket 5
2.2 Logo di Jira 5
2.3 Logo di Slack 5
2.4 Logo di Insomnia 6
2.5 Logo di RubyMine 6
2.6 Logo di GraphQL 7
2.7 Logo di Ruby 7
2.8 Logo di Ruby on Rails 8

3.1 Schema ER della gerarchia progetto, brand e canale 9
3.2 Esempio di configurazione a progetto, brand e canali 10

4.1 Benchmark della libreria *FastExcel* 14

5.1 Schema ER documenti di vendita 15
5.2 Schema delle classi implementate e le loro relazioni 23
5.3 Esempio di *mutation* `reportGenerate` e relativa risposta 29

Elenco delle tabelle

1.1 Preventivo orario iniziale 2

5.1 Campi notevoli del modello `Main` 16
5.2 Campi notevoli del modello `DocumentLine` 18
5.3 Campi notevoli del modello `VatRate` 18
5.4 Campi notevoli del modello `Billing` 19
5.5 Campi notevoli del modello `Shipping` 19
5.6 Campi notevoli del modello `SavedReport` 20

6.1 Consuntivo orario finale 30

Elenco dei listati di codice

1 Estratto dal codice del modello `Main` . . . 16
2 Esempio di *query* per l'aggiornamento della colonna `kind` di `Main` . . . 18
3 Esempio di definizione di un report . . . 21
4 Registro delle definitioni delle possibili tipologie di report . . . 22
5 Codice del controller `SaleDocumentsController` . . . 26
6 Codice del controller `GeneratedExcelsController` . . . 27
7 Codice della classe `UpsSender` . . . 33
8 Codice della classe `UpsGenerator` . . . 33
9 Codice della classe `Processor` . . . 34
10 Codice del *Rake task* che inizializza i `Processor` . . . 35
11 Estratto della classe `Definition` . . . 36
12 Codice della classe `FastExcelBackend` . . . 37
13 Codice della classe `Generator` . . . 38
14 Codice delle query `savedReports` e `reportDefinitions` . . . 39
15 Estratto della mutation `reportGenerate` . . . 39
16 Estratto della mutation `reportSave` . . . 40

Capitolo 1

Introduzione

1.1 Azienda

Moku S.r.l è una software house di circa 20 dipendendi con sede a Treviso. Fondata nel 2013, nata come startup all'interno dell'incubatore H-Farm ed ora indipendente, è una realtà giovane con un team variegato che spazia dallo sviluppo web al mobile. L'approccio dell'azienda allo sviluppo software è agile e mette al centro il cliente, rendendolo costantemente partecipe nella realizzazione dei prodotti commissionati e seguendolo fin dall'inizio per indirizzarlo verso la soluzione più adatta.

Figura 1.1: Logo dell'azienda

L'azienda lavora prevalentemente nell'ambito della trasformazione digitale, realizzando software su misura. Attraverso l'evento Stage-it, Moku e l'Università di Padova hanno nel tempo consolidato il loro rapporto di collaborazione.

1.2 Scopo

Lo scopo del progetto di stage è l'integrazione di un sistema di reportistica all'interno di un Order Management System (OMS). Il software, già esistente ed in continua evoluzione, permette di aggregare e manipolare ordini provenienti da vari canali e-commerce, all'interno di un'unica piattaforma. Tale applicativo si interfaccia inoltre con altri software che gestiscono specifiche funzionalità come spedizioni e fatturazione. Il sistema di reportistica si occuperà di aggregare i dati in persistenza nel database e di generare report specifici, permettendo il filtraggio dei dati su base temporale o su altre caratteristiche. Il motore di generazione da realizzare, dovrà quindi essere personalizzabile dall'utente, permettendogli di creare resoconti mensili relativi alle vendite, i quali saranno poi inviati ai propri clienti.

È quindi richiesto che venga svolta l'analisi dei requisiti e la conseguente progettazione e realizzazione del sistema di reportistica, tenendo conto delle particolarità

architetturali della piattaforma con cui si dovrà integrare. Saranno inoltre utilizzati gli strumenti, le tecnologie e le pratiche già impiegati all'interno dell'azienda ospitante.

1.3 Preventivo iniziale

Viene di seguito riassunto il preventivo orario inizialmente preventivato dall'azienda per la riuscita del progetto.

Tabella 1.1: Preventivo orario iniziale

Descrizione attività	**Indice settimana**	**Ore impiegate**
Comprensione sistema e obiettivi	1	20
Analisi dei requisiti	1	20
Progettazione	2-3	60
Studio e setup ambiente di sviluppo	3	20
Implementazione	4-7	150
Test e validazione	7-8	30
Documentazione	8	20
Totale		**320**

Nella sezione 6.1 del capitolo conclusivo seguirà il consuntivo orario finale.

Capitolo 2

Progetto

2.1 Piattaforma preesistente

L'azienda sta sviluppando per conto di una rilevante società nel mondo degli e-commerce, un sistema informativo per gestire i flussi di acquisto online in modo scalabile, modulare ed estensibile. Il cliente gestisce migliaia di ordini ogni giorno, per conto di decine di clienti, su un ampio numero di canali (siti e-commerce *custom*, *Shopify*, *Amazon*, *eBay*...), fornendo supporto ai propri utenti attraverso ottimizzazione della customer experience, analisi predittive e digitalizzando tutti i processi di vendita. Tale società gestisce per conto dei propri cliente anche la logistica, l'assistenza cliente ed il marketing. La piattaforma software, per semplicità definita OMS, si occupa di raccogliere gli ordini provenienti da varie fonti, smistarli nel canale di competenza e renderli poi consultabili da backoffice a seconda dei permessi dei vari utenti.

2.2 Obiettivi del progetto

È richiesta la realizzazione di un motore di generazione di report personalizzabile dall'utente che permetta di creare dei resoconti mensili relativi alle vendite da inviare ai propri clienti. I report da produrre sono di vario tipo e l'utente avrà la facoltà di scegliere la tipologia, i canali di vendita a cui si riferisce e il range temporale dei dati sorgente. Tutti questi parametri confluiranno all'interno di procedure che dovranno:

- validare le richieste dell'utente;
- ottenere dal database della piattaforma i dati necessari in maniera efficiente;
- eventualmente processare i dati ottenuti, sempre tenendo conto di velocità ed efficienza;
- trasformare i dati ottenuti per renderli utlizzabili da altre procedure e dal Frontend;
- eventualmente generare il report in formato XLSX.

Dovrà essere inoltre predisposto il necessario per acquisire i documenti di vendita degli ordine, in quanto sono questi la sorgente principale dei dati nei report.

Attraverso il *frontend*, il report sarà visibile in formato tabellare e sarà in questo punto in cui l'utente potrà inoltre decidere se scaricare il report nel formato *XLSX*.

L'effettiva implementazione della parte di visualizzazione nel *frontend* esula però dagli obiettivi del progetto di stage e verrà realizzata successivamente o parallelamente da terzi, per cui non viene trattata.

2.3 Metodo di lavoro

L'azienda lavora seguendo, e promuovendo attivamente, il framework *Scrum*. È un modello di sviluppo Agile ampiamente utilizzato nel contesto del project management e dello sviluppo software. Si basa su un approccio iterativo e incrementale, in cui il lavoro viene organizzato in cicli chiamati sprint. Durante ogni sprint, il team si impegna a consegnare un incremento di prodotto funzionante.

Nel caso di Moku, gli sprint durano 2 settimane ed al loro termine i risultati ottenuti vengono mostrati al cliente, attraverso una *sprint review*. Durante queste occasioni, si risolvono eventuali dubbi emersi e si effettua una pianificazione dello sprint successivo. Inoltre, se necessario, si effettua anche una *mid-sprint review* a metà dello sprint attuale.

All'inizio di ogni giornata lavorativa si svolge uno *stand-up meeting*, un incontro di circa 15 minuti fra tutti i componenti del gruppo di lavoro, nel quale si discute dello stato generale del progetto e soprattutto lo stato del proprio lavoro individuale. Questa pratica in particolare, consente di agire subito su eventuali problemi riscontrati nella giornata precedente, e di risolverli pianificando il lavoro da svolgere nella giornata corrente.

2.4 Strumenti a supporto

Bitbucket

Servizio di hosting per repository Git, prodotto da Atlassian e basato su cloud. Offre strumenti per la collaborazione, consentendo ai team di lavorare insieme in modo più efficiente. Ad esempio, gli sviluppatori possono creare pull request per revisioni del codice da parte di altri membri del team. Bitbucket è integrato con altri strumenti di sviluppo come Jira, consentendo una maggiore sincronizzazione tra le varie attività dello sviluppo.

Figura 2.1: Logo di Bitbucket

Jira

È una piattaforma di gestione dei progetti sviluppata da Atlassian. È ampiamente utilizzata per la pianificazione, il monitoraggio e la gestione delle attività di sviluppo del software. Possono essere create e gestite le attività all'interno di progetti. Le attività possono essere assegnate a membri del team, suddivise in categorie, priorizzate e seguite nel tempo. Jira offre una visione chiara dello stato delle attività e permette di tenere traccia dei progressi e delle scadenze. Si integra con Bitbucket, favorendo una migliore collaborazione tra i diversi strumenti utilizzati nel processo di sviluppo del software.

Figura 2.2: Logo di Jira

Slack

Software di messaggistica orientato alla collaborazione nell'ambito dello sviluppo software. È possibile creare canali di comunicazione dedicati per i diversi progetti, team o argomenti. Questi canali sono come stanze virtuali in cui i membri del team possono, oltre che inviare messaggi, condividere file e documenti. Slack offre anche integrazioni con altre applicazioni e servizi, come Google Drive, Trello, GitHub e Jira. Ciò consente di collegare facilmente i flussi di lavoro e di condividere informazioni da fonti diverse all'interno della stessa piattaforma.

Figura 2.3: Logo di Slack

Insomnia

Client HTTP che permette di testare e interagire con API attraverso richieste HTTP. Con funzionalità come la visualizzazione delle richieste e delle risposte, l'organizzazione delle richieste in cartelle o ambienti, è uno strumento che semplifica il processo di sviluppo e debug delle API. L'interfaccia di Insomnia fornisce una comoda visualizzazione delle richieste e delle risposte, mostrando informazioni come il metodo HTTP utilizzato, i parametri, gli header e il corpo della richiesta. Supporta inoltre diversi tipi di autenticazione per consentire l'accesso a API protette.

Figura 2.4: Logo di Insomnia

RubyMine

È un IDE progettato specificamente per il linguaggio di programmazione Ruby e il framework Ruby on Rails. Offre un'ampia gamma di strumenti e funzionalità per semplificare il processo di sviluppo di applicazioni Ruby, includendo funzionalità come il completamento automatico del codice, la navigazione intelligente tra i file, il debugging avanzato, e una suite completa di strumenti per il testing e l'analisi del codice.

Figura 2.5: Logo di RubyMine

2.5 Tecnologie di sviluppo

GraphQL

È un linguaggio di interrogazione per Application Program Interface (API), nato in Facebook e successivamente reso pubblico e open source. A differenza degli approcci tradizionali basati su REST, GraphQL consente ai client di specificare esattamente quali dati desiderano ricevere dal server. Invece di ottenere un'intera risorsa con tutte le sue proprietà, un client può richiedere solo i campi specifici di interesse, riducendo così la quantità di dati trasferiti e migliorando le prestazioni dell'applicazione. Possono anche essere combinate più richieste insieme, consentendo di ottenere più risorse simultaneamente, riducendo così la latenza delle richieste.

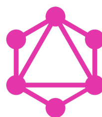


Figura 2.6: Logo di GraphQL

GraphQL fornisce anche un sistema di tipi che consente agli sviluppatori di definire la struttura dei dati in modo chiaro e preciso. Ciò aiuta a garantire che le *query* siano corrette a livello sintattico e semantico e fornisce una documentazione automatica delle risorse esposte. Oltre che le *query* per ottenere dati, sono supportate le *mutation* che consentono ai client di eseguire operazioni di modifica dei dati come l'inserimento, l'aggiornamento o l'eliminazione.

Ruby

È un linguaggio di programmazione dinamico, orientato agli oggetti e di alto livello. Con ormai 30 anni di storia, ha sempre avuto l'obiettivo di essere semplice, flessibile ed espressivo.

Figura 2.7: Logo di Ruby

Ruby tende ad avere una particolare enfasi sulla sintassi pulita e leggibile, con un linguaggio che si avvicina molto al linguaggio naturale. Tecniche come quella del *"monkey patching"*, consentono agli sviluppatori di adattare il linguaggio stesso, e le eventuali librerie terze, alle proprie esigenze specifiche, senza dover modificare il codice sottostante. A rafforzare questa sua capacità di adattarsi ad esigenze molto specifiche, è il suo potente sistema di metaprogrammazione. Questa caratteristica permette di scrivere codice che può manipolare e modificare se stesso durante l'esecuzione. Gli sviluppatori possono definire nuove classi, creare metodi dinamicamente e persino cambiare il comportamento delle classi esistenti in fase di esecuzione.

Ruby beneficia una vasta comunità di sviluppatori e di una ricca raccolta di gemme (librerie di terze parti). Queste gemme estendono ulteriormente le funzionalità del linguaggio e coprono una vasta gamma di esigenze.

Ruby on Rails

Chiamato anche soltanto Rails, è un framework di sviluppo web open source basato sul linguaggio Ruby. È diventato molto popolare grazie alla sua filosofia di sviluppo rapido di applicazioni web e alla sua capacità di semplificare molti aspetti dei processi di sviluppo.

Figura 2.8: Logo di Ruby on Rails

Rails è basato sul pattern architetturale Model-View-Controller (MVC). Questo pattern suddivide l'applicazione in tre componenti principali:

- *Model*: gestisce i dati dell'applicazione e contiene la logica di business, regolando gli accessi in lettura e scrittura;
- *View*: si occupa di gestire la visualizzazione dei dati richiesti, forniti dal *Model*;
- *Controller*: agisce come intermediario fra il *Model* e la *View*, aggiornando i dati in base all'input dell'utente.

Tale struttura promuove la separazione dei compiti e rende più facile organizzare e mantenere il codice.

Rails include anche un potente Object-Relational Mapping (ORM) chiamato *ActiveRecord*, che permette di mappare i dati del database direttamente agli oggetti Ruby, consentendo agli sviluppatori di manipolare i dati utilizzando la sintassi di Ruby senza doversi preoccupare di interrogare manualmente il database tramite SQL.

Un'altra caratteristica di Rails è l'approccio alla programmazione *"Convention over Configuration"* (Convenzione anziché configurazione). Questo significa che Rails fornisce una serie di convenzioni standardizzate che consentono di risparmiare tempo e sforzi nella configurazione manuale. Ad esempio, i nomi dei file, delle classi e delle tabelle del database seguono convenzioni predefinite, semplificando l'interazione tra le varie parti dell'applicazione.

Tutte queste peculiarità consentono di sviluppare applicazioni web complesse in modo efficiente.

Capitolo 3

Analisi dei requisiti

3.1 Architettura preesistente

L'architettura della piattaforma OMS preesistente ruota attorno ad alcuni concetti chiave, i quali saranno rilevanti anche durante la progettazione e realizzazione del sistema di reportistica.

Gli ordini che confluiscono all'interno dello piattaforma, vengono acquisiti e trattati con modalità diverse a seconda della sorgente, d'ora in poi definita *canale*. I canali possono essere di varia natura, a titolo esemplificativo si possono citare *Ebay* e piattaforme *custom* basate su Drupal. I canali appartengono ad uno o più *brand*, i quali a loro volta appartengono e sono quindi accomunati da un solo *progetto*. Questa struttura gerarchica permette, in generale, una grande flessibilità nella gestione degli ordini all'interno della piattaforma.

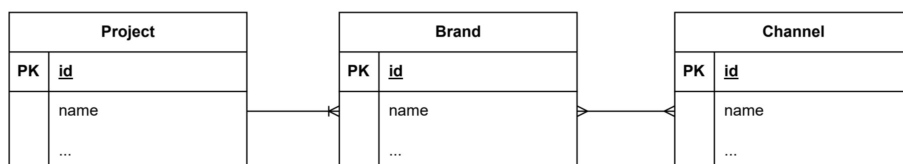


Figura 3.1: Schema ER della gerarchia progetto, brand e canale

Come illustrato dalla figura 3.1, progetti, brand e canali sono modellati nel database nelle loro tabelle dedicate, mappate a loro volta nei modelli:

- `Project`
- `Brand`
- `Channel`

Riassumendo la gerarchia appena descritta:

- ogni progetto può avere più brand associati;
- un brand può appartenere soltanto ad un progetto;
- un brand ha uno o più canali;
- i canali possono essere condivisi fra vari brand.

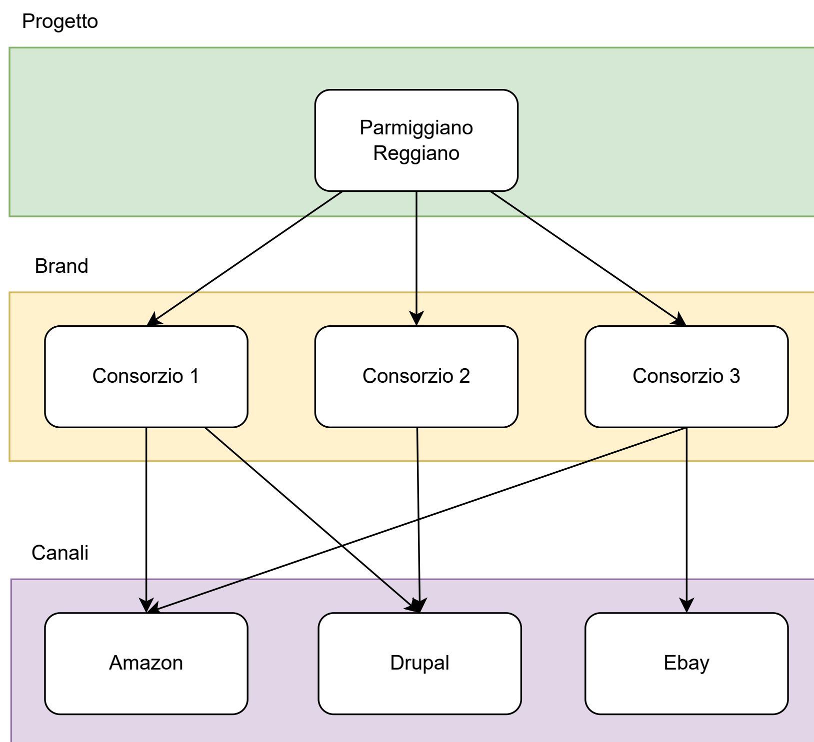


Figura 3.2: Esempio di configurazione a progetto, brand e canali

3.2 Utilizzatori

Gli utilizzatori del sistema di reportistica saranno tutti gli utenti che hanno già accesso alla piattaforma OMS.

3.3 Funzionalità

Dovrà essere prevista la possibilità di filtrare i dati in base a:

- canale di provenienza
- intervallo temporale

Questi filtri dovranno poi essere poi facilmente riutilizzabili in futuro, associandoli ad una tipologia di report ed un nome.
Il cliente vorrebbe inoltre che il nuovo sistema di reportistica sia più veloce ed efficiente rispetto alla soluzione *legacy* già in loro possesso.

3.4 Struttura di un report

I report attualmente generati dal cliente, attraverso software *legacy*, sono in formato compatibile con Excel, quindi dei fogli di calcolo in forma tabellare. In particolare, all'interno di essi, si possono individuare i seguenti elementi:

- Intestazione: la parte superiore del report, solitamente la prima riga del foglio;
- Colonne: ogni colonna rappresenta una determinata categoria di dati (importi, nomi, codici...);
- Righe: ogni riga contiene le informazioni di un ordine, proveniente dai documenti di vendita (fatture, note di credito...);
- Formule e calcoli: operazioni matematiche o analisi sui dati, per esempio la media o la somma di un'intera colonna;
- Formattazione: alcune colonne possono prevedere un certo stile di carattere o colore.

3.5 Tipologie di report

Come menzionato nella precedente sezione, il cliente dispone già di un sistema di reportistica interno. Dovranno essere quindi riprodotte le stesse tipologie di report già in uso, e predisporre la soluzione per future estensioni. In particolare, è richiesta l'implementazioni delle seguenti tipologie di report, detti anche *cruscotti*:

- *Report 5102*: lista degli ordini esteri con le rispettive imposte;
- *Report 5112*: lista degli ordini italiani con le rispettive imposte disaggregate per aliquata IVA;
- *Report 5113*: lista degli articoli di tutti gli ordini italiani con le relative spese disaggregate per tipologia.

3.6 Documenti di vendita

Almeno inizialmente, i documenti di vendita saranno l'unica sorgente dei dati dei report implementati. Tali documenti, dovranno essere oppurtunamente acquisiti e memorizzati.

Capitolo 4

Progettazione

4.1 Gestione dei documenti di vendita

4.1.1 Acquisizione

L'acquisizione dei documenti di vendita avviene in maniera asincrona, quindi dopo la ricezione dell'ordine stesso all'interno della piattaforma OMS. Il cliente ha la possibilità di inviare tali documenti attraverso chiamate HTTP ad un endpoint specifico, esposto dalla piattaforma. I dati dei documenti saranno codificati in formato JSON.

Dovrà quindi essere implementato un *controller* specifico, che dovrà:

- gestire la chiamata in arrivo;
- estrarre i dati dalla chiamata;
- validare i dati;
- memorizzare appropriatamente il documento di vendita acquisito.

Sarà anche gestita l'autenticazione tramite un *token* inserito negli *header HTTP* della chiamata.

4.1.2 Persistenza

I documenti di vendita saranno persistiti in maniera strutturata nel database. Sarà prevista inoltre la possibilità di riprocessare i dati grezzi in formato JSON, in caso fosse in futuro necessario. Questa funzionalità è ritenuta utile in quanto è possibile che in futuro possano cambiare le logiche di gestione dei documenti di vendita, rendendo necessario adeguare i dati già esistenti.

4.2 Definizione dei report

Definire nuove tipologie di report, e modificare quelli già in essere, dovrà essere reso il più possibile agevole ed efficiente. Tale scopo sarà raggiunto sfruttando le caratteristiche del linguaggio Ruby, in particolare il suo potente motore di metaprogrammazione. In concreto, verrà realizzato un Domain Specific Language (DSL) specifico per definire la struttura ad alto livello di un report ed i dati che lo compongono.

In particolare, il *DSL* fornirà delle istruzioni specifiche per definire:

- il nome e la descrizione del report;
- la formattazione delle colonne, per esempio colonne di tipo valuta o percentuale;
- l'intestazione delle colonne, anche detto *header*.

4.3 Generazione dei report

4.3.1 Performance e pre-aggregazione

Al fine di rendere il motore di reportistica veloce ed efficiente, dovranno essere presi degli accorgimenti tali da rendere la generazione dei report conforme alle richieste del cliente. Analizzando la lista di azioni da intraprenderà per fornire all'utente il report desiderato, è emerso che la parte più onerosa dal punto di vista computazionale è il calcolo di valori aggregati.

Esempi di alcuni valori aggregati da calcolare sono:

- totale dei soli prodotti che ricadono in una certa aliquota IVA;
- totale delle sole spese di spedizione;
- codici degli articoli che rispondono a determinate caratteristiche.

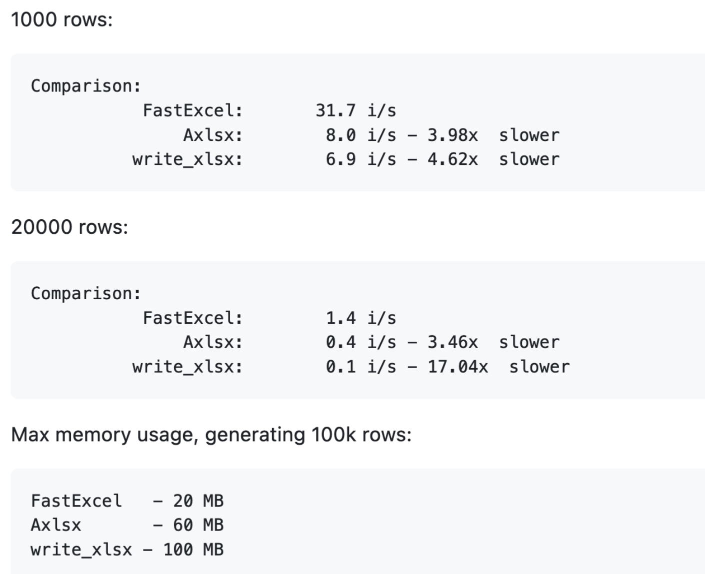
Da qui, l'intuizione di effettuare quella che d'ora in poi verrà definita *pre-aggregazione* dei dati. Ciò consisterà nel calcolare i valori aggregati, che serviranno per generare i report, all'acquisizione dei documenti di vendita, e non durante la generazione dei report stessi. Ciò consentirà di avere un importante, e soprattutto tangibile, impatto sulle performance.

4.3.2 Scelta della libreria a supporto

Per generare i report in formato XLSX, sempre tenendo conto delle esigenze di performance, è stata prestata particolare attenzione alla scelta della libreria a cui delegare questa parte. Come precedentemente menzionato, l'ecosistema di librerie terze nell'ambito del linguaggio Ruby è particolarmente florido, e ciò ha permesso di trovare una *gemma* che risponde a tutte le caratteristiche richieste, di seguito riassunte:

- facile utilizzo;
- documentazione chiara e completa;
- stato attivo di manutenzione e sviluppo;
- basso utilizzo di risorse, sia computazionali che di memoria.

Confrontando criticamente varie possibilità, la scelta è ricaduta su *FastExcel*.



1000 rows:

```
Comparison:
           FastExcel:       31.7 i/s
               Axlsx:        8.0 i/s - 3.98x  slower
          write_xlsx:        6.9 i/s - 4.62x  slower
```

20000 rows:

```
Comparison:
           FastExcel:        1.4 i/s
               Axlsx:        0.4 i/s - 3.46x  slower
          write_xlsx:        0.1 i/s - 17.04x  slower
```

Max memory usage, generating 100k rows:

```
FastExcel   - 20 MB
Axlsx       - 60 MB
write_xlsx - 100 MB
```

Figura 4.1: Benchmark della libreria *FastExcel*

La figura 4.1 mostra un estratto dalla documentazione della libreria, dove vengono riportati i risultati di alcuni benchmark che mettono a confronto le performance di altre librerie che offrono funzionalità simili. In particolare, confrontando *FastExcel* rispetto alla libreria più lenta fra quelle prese in esame, si può notare che:

- generando 20000 righe, è circa 17 volte più veloce;
- generando 100000 righe, occupa 5 volte meno memoria.

4.4 Interfacciamento con il frontend

L'interfacciamento con la parte di Frontend avverrà attraverso *query* e *mutation* GraphQL. In particolare, sarà:

- realizzata la logica per ottenere, validare, e all'occorrenza trasformare, i dati in input dall'utente
- definito i tipi di dati interscambiati fra client e server.

Capitolo 5

Realizzazione

5.1 Modelli

In seguito, verranno elencati e descritti soltanto gli elementi salienti delle implementazioni, omettendo quindi dettagli minori o non inerenti, in modo da garantire una fruizione più scorrevole dei contenuti.

5.1.1 Documenti di vendita

I modelli di seguito descritti, sono raggruppati nel modulo `SaleDocuments`. Ciò si riflette anche nel nome delle tabelle del database associate ai modelli. Il framework Rails segue infatti la convenzione di aggiungere come suffisso il nome del modulo a cui appartiene il modello al nome della tabella ad esso associato. In questo caso, il suffisso sarà `sale_documents_`.

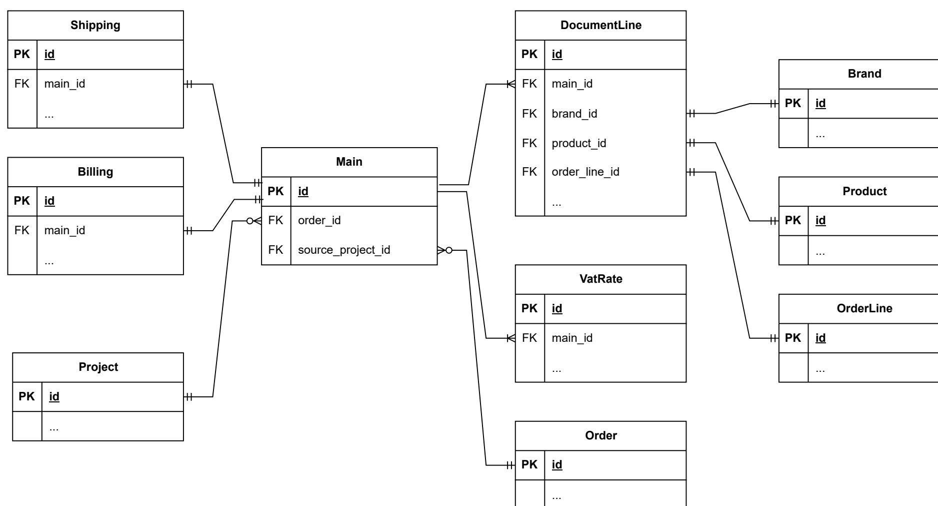


Figura 5.1: Schema ER documenti di vendita

La figura 5.1 rappresenta lo schema Entity Relationship (ER) dei modelli, e quindi delle relative tabelle, implementati nel database.

I seguenti modelli non sono stati implementati perchè già presenti nell'architettura attuale della piattaforma:

- `Project`
- `Brand`
- `Product`
- `Order`
- `OrderLine`

Main

Si tratta del modello più importante di tutta la gerachia riguardante i documenti di vendita, in quanto raggruppa e fornisce punti di accesso a tutti gli elementi che compongono un singolo documento.

Tabella 5.1: Campi notevoli del modello `Main`

Nome	Tipo	Descrizione
`number`	Stringa	Numero progressivo del documento
`kind`	Intero	Tipologia di documento
`currency`	Stringa	Valuta a cui fanno riferimento tutti gli importi espressi nel documento
`exchange_rate`	Decimale	Tasso di cambio al momento della creazione fra la valuta del documento ed Euro
`raw_data`	JSON	Dati grezzi, non processati, del documento. Serve per poter riprocessare il documento in futuro, in caso sia necessario
`report_aggregate_data`	JSON	Dati e valori pre-aggregati
`total_net`	Decimale	Totale al netto dell'IVA
`total_vat`	Decimale	Totale delle sole imposte

Essendo PostgreSQL il DBMS in uso, è possibile utilizzare come tipo delle colonne il formato JSON. Ciò si rende utile per salvare dati che non seguono uno schema preciso all'interno delle colonne della tabella. Nella tabella 5.1 sono contrassegnati quali campi fanno uso del tipo *JSON*.

Listato 1: Estratto dal codice del modello `Main`

```
1 module SaleDocuments
2   class Main < ApplicationRecord
3     belongs_to :order
4     belongs_to :source_project, class_name: 'Project', inverse_of:
      ↪  :sale_documents
```

```
5       has_one :shipping,
6               class_name: 'SaleDocument::Shipping',
7               inverse_of: :main,
8               dependent:  :destroy
9       has_one :billing,
10              class_name: 'SaleDocument::Billing',
11              inverse_of: :main,
12              dependent:  :destroy
13      has_many :document_lines,
14               class_name:  'SaleDocument::DocumentLine',
15               foreign_key: :sale_document_main_id,
16               inverse_of:  :main,
17               dependent:   :destroy
18      has_many :vat_rates,
19               class_name:  'SaleDocument::VatRate',
20               foreign_key: :sale_document_main_id,
21               inverse_of:  :main,
22               dependent:   :destroy
23
24      enum kind: {
25        sale:   0,
26        credit: 1,
27
28        # ...
29
30      }
31
32      # ...
33
34    end
35  end
```

Come si può notare dal listato 1, nel modello vengono dichiarate le associazioni con altri modelli, rispettando quanto descritto nello schema 5.1. In particolare, `Main` ha associazioni di tipo *uno a uno* con:

- `Order`
- `Project`
- `Shipping`
- `Billing`

Ha invece associazioni di tipo *uno a molti* con:

- `DocumentLine`
- `VatRate`

Degna di nota è anche la dichiarazione come *enumeratore* delle possibili tipologie di documento, dove viene per esempio assegnato alla tipologia `credit` (nota di credito) il numero intero `1`. Ciò rende più chiare ed agevole le operazioni di lettura e scrittura, in

quanto ci si potrà riferire al contenuto della colonna `kind` non attraverso un valore numerico intero ma attraverso la sua etichetta.

Listato 2: Esempio di *query* per l'aggiornamento della colonna `kind` di `Main`

```
1 # find document with id 123 and update kind to sale
2 SaleDocuments::Main.find(123).update! kind: :sale
```

DocumentLine

Rappresenta una riga del documento di vendita. Ogni riga fa riferimento ad un articolo acquistato, con tutte le informazioni utili ai fini della fatturazione.

Tabella 5.2: Campi notevoli del modello `DocumentLine`

Nome	Tipo	Descrizione
`serial_number`	Stringa	Numero di serie del prodotto
`barcode`	Stringa	Codice a barre del prodotto
`price`	Decimale	Prezzo unitario
`total_price`	Decimale	Prezzo totale
`coupon_code`	Stringa	Codice del coupon, se presente
`price_discounted`	Decimale	Prezzo unitario se il coupon è presente
`quantity`	Decimale	Quantità di prodotti

VatRate

Rappresenta un'aliquota IVA. Ogni documento di vendita può averne una o più.

Tabella 5.3: Campi notevoli del modello `VatRate`

Nome	Tipo	Descrizione
`percentage`	Decimale	Percentuale aliquota IVA
`net_amount`	Decimale	Ammontare netto della merce all'aliquota specifica
`total`	Decimale	Ammontare lordo della merce all'aliquota specifica

Billing

Rappresenta i dati di fatturazione, compresi quelli del cliente a cui fa riferimento il documento.

Tabella 5.4: Campi notevoli del modello `Billing`

Nome	Tipo	Descrizione
`number`	Stringa	Numero progressivo della fattura
`first_name`	Stringa	Nome del cliente
`last_name`	Stringa	Cognome del cliente
`vat_number`	Stringa	Partita IVA del cliente, se presente
`fiscal_code`	Stringa	Codice fiscale del cliente, se presente
`country`	Stringa	Paese di fatturazione

Shipping

Rappresenta i dati relativi alla spedizione.

Tabella 5.5: Campi notevoli del modello `Shipping`

Nome	Tipo	Descrizione
`first_name`	Stringa	Nome del cliente
`last_name`	Stringa	Cognome del cliente
`address1`	Stringa	Prima riga dell'indirizzo di destinazione
`address2`	Stringa	Seconda riga dell'indirizzo di destinazione
`country`	Stringa	Paese di destinazione
`province`	Stringa	Provincia o regione di destinazione
`zip_code`	Stringa	Codice di avviamento postale della destinazione

5.1.2 Report

SavedReport

Questo modello consente di consente di salvare con un nome i filtri associati ad una tipologia di report. Ciò permette all'utente in futuro di generare un report con gli stessi filtri già specificati in passato.

Tabella 5.6: Campi notevoli del modello `SavedReport`

Nome	Tipo	Descrizione
`name`	Stringa	Nome del report impostato dall'utente
`kind`	Intero	Tipologia del report salvato
`start_date`	Data	Data di inizio validità dei dati del report
`end_date`	Data	Data di fine validità dei dati del report

5.2 Definizioni delle tipologie di report

5.2.1 DSL specifico

Il DSL implementato per definire un report, comprende le seguenti istruzioni:

- `named`: permette di dichiarare il nome del report;
- `description`: permette di dichiarare la descrizione del report;
- `header`: permette di dichiarare l'*header*, quindi la lista delle intestazioni delle colonne;
- `col`: consente di specificare il formato di un determinata colonna;
- `cols`: consente di specificare il formato di un intervallo contiguo di colonne;
- `pre_aggregate`: consente di dichiarare quali dati pre-aggregati sono necessari e come calcolarli.

5.2.2 Definition

La classe `Definition` si occupa di implementare il DSL precedentemente descritto. Essa espone alcuni metodi statici, i quali perciò potranno essere invocati senza necessità creare un'istanza della classe. Ognuno di quei metodi implementa un'istruzione del *DSL*, i cui parametri saranno temporaneamente memorizzati attraverso la Metaclasse di `Definition`.
Il metodo `build` si occupa infine di istanziare la definizione con i parametri corretti. Una istanza di questa classe quindi, racchiude tutte le informazioni necessarie per definire e conseguentemente generare un report.

Listato 3: Esempio di definizione di un report

```
class Report5102 < Definition
  named 'Cruscotto 5102'
  description 'IVA paese destinazione'

  header [
    'Progetto',
    'Brand',
    'Canale',
    # more header titles...
    'IVA merce',
    'Totale doc.'
  ]

  col 5, :date
  cols 11, 16, :eur, sum: true

  pre_aggregate do |document|
    data = document.document_lines_for_report

    # pre-aggregration instructions...
```

```
21
22      data.transform_values!(&:to_f)
23      data
24    end
25  end
```

Il listato 3, mostra come sia relativamente semplice dichiarare la struttura di un report. In questo caso il report denominato *"Cruscotto 5102"*, avrà:

- l'intestazione come definita dall'*array* di stringhe passato all'istruzione `header`;
- la colonna di indice 5 formattata come data;
- le colonne comprese fra indici 11 e 16 formattate come valuta Euro, inoltre la somma di ognuna verrà mostrata nell'ultima riga del report (come specificato dal *flag* `sum: true`)

L'istruzione `pre_aggregate`, permette di definire di quali dati pre-aggregati necessita il report e come estrarli da un documento di vendita.
Il risultato del blocco, quindi i valori pre-aggregati, verrà opportunamente persistito nel campo `report_aggregate_data` del modello `Main`.
Ciò avverrà subito dopo l'acquisizione del documento di vendita, in modo da avere tali dati già disponibili quando verrà richiesta la generazione di un report, senza quindi la necessità di calcolarli al momento.

Il listato 11 in appendice B contiene l'implementazione sottostante di `Definition`.

5.2.3 Registro

Al fine di poter rendere sempre disponibili le definizioni delle possibili tipologie di report, è stato implementato un registro che le mantiene in memoria.

Listato 4: Registro delle definitioni delle possibili tipologie di report

```
1  module Reports
2    module Definitions
3      REGISTRY = {
4        report5102: Report5102.build,
5        report5112: Report5112.build,
6        report5113: Report5113.build
7      }.freeze
8
9      def self.for_kind kind
10       REGISTRY.fetch kind
11     end
12   end
13 end
```

Come mostrato nel listato 4, il registro è la costante `REGISTRY`. Viene quindi dichiarata la costante di tipo `Hash`, una mappa chiave-valore nel linguaggio Ruby, dove le chiavi sono i simboli che identificano la tipologia di report e i valori oggetti di tipo `Definition`.

Il metodo statico di utilità `for_kind`, invocato passando la tipologia di report desiderata, fornisce un accesso al registro semanticamente più corretto.

5.3 Servizi

Verranno di seguito descritte nel dettaglio le classi implementate aventi il compito di adeguatamente generare un report a partire dalle richieste dell'utente.

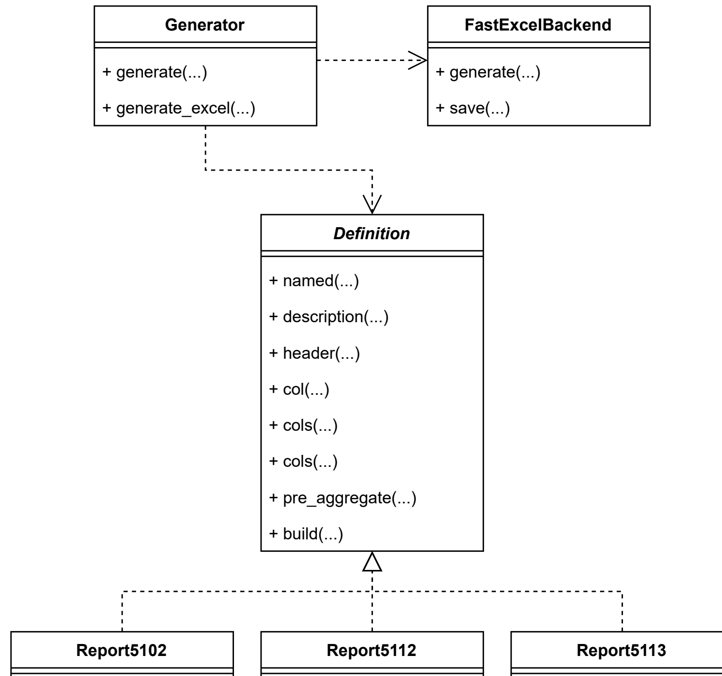


Figura 5.2: Schema delle classi implementate e le loro relazioni

5.3.1 FastExcelBackend

Questa classe fornisce tutte le funzionalità necessarie al fine di generare un file XLSX valido, contenente i dati del report appena generato. In particolare assolve ai seguenti compiti:

- interfacciarsi con la libreria *FastExcel*;
- allocare in memoria un buffer dove poter memorizzare il file mentre viene costruito iterativamente;
- trasformare ogni riga del report in una riga di celle;
- definire come trattare celle particolari, quali quelle formattate come data o valuta;
- persistere in memoria di massa il file, rendendolo disponibile per essere scaricato dall'utente.

Dopo l'inizializzazione, un oggetto di tipo `FastExcelBackend` espone i metodi:

- `generate`: genera in memoria il file. Viene invocato specificando le righe del report, l'*header* (i titoli di ogni colonna), i formati che devono avere le varie colonne e il nome del foglio.

- `save`: salva in memoria persistente il file generato, con il nome specificato.

L'interfacciamento con la libreria *FastExcel* consente di utilizzare alcune operazioni di base per la generazione di un file *XLSX* valido, in particolare:

- Aggiungere un nuovo foglio;
- Aggiungere un numero arbitrario di righe;
- Impostare il contenuto delle celle;
- Impostare il formato delle celle, ad esempio data o valuta;
- Aggiungere formule alle celle.

Tutte le operazioni precedentemente elencate quindi, verranno utilizzate in combinazione l'una con le altre nella maniera il più possibile efficiente per ottenere il risultato voluto.
Ad esempio, l'ordine delle operazioni che in generale vengono eseguite per generare un report prevede:

- Aggiunta di un nuovo foglio;
- Nel foglio appena creato, aggiunta di una riga alla volta;
- Dopo aver creato una riga, vengono aggiunte tutte le relative celle con il loro contenuto. Se il contenuto è una formula, viene aggiunta appropriatamente;
- Se alcune celle della riga prevedono una formattazione specifica, viene impostata.

In appendice B, il listato 12 contiene un'estratto del codice di questa classe.

5.3.2 Generator

Questa classe ha l'importante compito di generare il report sia in formato grezzo, sia come file XLSX attraverso `FastExcelBackend`.

Una istanza di `Generator` viene creata fornendo l'istanza della definizione del report che si vuole generare (Dependency injection di `Definition`) e gli eventuali filtri sui dati sorgente. Viene inoltre istanziato internamente un oggetto di tipo `FastExcelBackend` che si occuperà di generare il report in formato XLSX se richiesto dall'utente.

La classe espone i seguenti metodi:

- `generate`: si occupa di generare le righe del report, invocando `generate_reports_rows` di `Definition`.
- `generate_excel`: partendo dalle righe già generate (se non lo sono viene invocato `generate`), crea il file *XLSX* invocando `generate` sull'istanza di `FastExcelBackend`. Il nome del file, se non specificato, viene generato in maniera randomica in modo da evitare conflitti di nomi nel momento in cui il file verrà persistito in memoria di massa.

Viene inoltre esposto il metodo statico di utilità `for_report_kind`, il quale si occupa di istanziare automaticamente il `Generator` semplicemente specificando il nome della tipologia di report e gli eventuali filtri sui dati, senza perciò la necessita di dover manualmente fornire un oggetto di tipo `Definition`.

I metodi di generazione possono essere invocati più volte, con la garanzia di ottenere sempre lo stesso risultato. Ciò avviene perchè i dati grezzi del report vengono calcolati una sola volta e successivamente memorizzati appropriatamente all'interno dei campi del `Generator` per essere riutilizzati.

In appendice B, il listato 13 contiene un'estratto del codice di questa classe.

5.4 Controller

5.4.1 SaleDocumentsController

Questo controller implementa le logiche di acquisizione dei documenti di vendita.

Listato 5: Codice del controller `SaleDocumentsController`

```
module Orders
  class SaleDocumentsController < ApplicationController
    def create
      channel = Channel.find_by! agilis_code:
      ↪ params[:channel_channel_code]
      order   = channel&.orders&.find_by! external_code:
      ↪ params[:order_order_code]
      project = order&.denormalized_projects&.first

      data = params.require(:sale_document).permit!.to_h
      SaleDocuments::Utils.validate_document data

      # create billing...
      # create shipping...
      # create VAT rates...

      # create complete document into transaction
    end

    # errors handling...
  end
end
```

Ottenere il *canale* sorgente dell'ordine a cui fa riferimento il documento è la prima operazione svolta dal controller. Dal *canale*, vengono allora volta ricavati l'*ordine* ed infine il *progetto*. Non appena il resto dei dati in arrivo viene validato, vengono svolte le seguenti azioni:

- vengono estratti i dati di fatturazione, e creata un'istanza del modello `Billing`;
- vengono estratti i dati di spedizione, e creata un'istanza del modello `Shipping`;
- vengono estratte le varie aliquote IVA che compongono il totale del documento, e create tante istanze di `VatRate` quante sono le aliquote;

Infine, tutti i modelli precedentemente istanziati vengono agganciati ad una nuova istanza di `Main`. Il controller quindi termina segnalando al *client* che l'operazione è andata a buon fine.

5.4.2 GeneratedExcelsController

I report in formato XLSX, archiviati in memoria di massa come file, possono essere scaricati dall'utente grazie a questo controller. Le richieste di tipo *HTTP GET* in arrivo, vengono validate e ne viene estratto il *token* generato da `Generator`. Questo viene a

sua volta validato, in modo da accertare l'autenticità della richiesta e l'autorizzazione dell'utente che l'ha iniziata.

Listato 6: Codice del controller `GeneratedExcelsController`

```
module Reports
  class GeneratedExcelsController < ApplicationController
    def show
      verifier  = Rails.application.message_verifier
      ↪  Settings::VERIFIER_NAME
      filenames = verifier.verify params[:token]

      report_path = Settings::REPORTS_PATH.join
      ↪  filenames[:real_filename]
      report_data = File.read report_path
      File.delay.delete report_path

      send_data report_data, filename: filenames[:pretty_filename]
    end

    # errors handling...
end
```

Se tutto va a buone file, dal *token* vengono estratti il percorso in cui è memorizzato il file ed il nome con il quale l'utente desidera riceverlo.

Al fine di evitare eccessivo consumo di memoria di massa, prima di inviare il file e soddisfare la richiesta, viene istanziato un Delayed job che si occuperà di cancellare il file in maniera asincrona, non appena questo avrà cessato la sua utilità.

5.5 GraphQL

Vengono di seguito descritte le *query* e *mutation* implementate per l'interfacciamento con il Frontend della piattaforma.

5.5.1 Query

In particolare, le *query* espongono le seguenti tipologie di informazioni:

- `savedReports`: espone tutte le istanze di `SavedReport` create dall'utente che le richiede;
- `reportDefinitions`: grazie al registro di definizioni precedentemente descritto, espone la lista di tutte le tipologie di report disponibili.

Il listato 14 in appendice B, mostra l'implementazione delle *query* implementate.

5.5.2 Mutation

Le operazioni di tipo *mutation*, che quindi possono potenzialmente mutare i dati persistiti nella piattaforma, rappresentano una parte più critica e complessa rispetto alle *query* di sola lettura.

reportSave

Questa *mutation* consente di salvare con un nome i filtri associati ad una tipologia di report.

Le operazioni che svolge sono, nell'ordine:

- estrazione dei parametri in arrivo specificati dall'utente;
- validazione dei parametri;
- creazioni di un modello di tipo `SavedReport` con i parametri specificati;
- persistenza nel database del report salvato rappresentato dal modello.

Il listato 16 in appendice B, contiene un'estratto del codice della mutation.

reportGenerate

Questa *mutation* rappresenta per il Frontend la porta di accesso al sistema di reportistica vero e proprio. Si occupa infatti di istanziare tutte le classi rilevanti precentemente descritte, al fine di generare un report secondo i requisiti specificati dall'utente.

In particolare, le operazioni svolte comprendono:

- estrazione dei parametri in arrivo specificati dall'utente;
- validazione dei parametri;
- determinare se i filtri sui dati provengono da un'istanza di `SavedReport` precedentemente creata o oppure se vengono specificati al momento;
- istanziare un oggetto di tipo `Generator` per il tipo di report ed i filtri ad esso associati;

- generare il report secondo le disposizioni dell'utente attraverso il `Generator` appena istanziato.

La *mutation* prevede in risposta i dati del report in formato grezzo, ed anche l'indirizzo univoco dal quale può venire scaricato in formato Excel. Tale campo può venire richiesto o meno, in questo modo è possibile determinare se è necessaria la generazione in tale formato.
La generazione in formato Excel avviene quindi solo se espressamente richiesto dall'utente, in modo da non impiegare inutilmente risorse computazionali.

Viene di seguito riportato un esempio reale di come viene strutturata un *mutation* e la relativa risposta. Vengono richiesti soltanto i dati grezzi del report e non l'indirizzo per scaricarlo nel formato Excel.

```
1  mutation report {
2    reportGenerate(
3      input: {
4        startDate: "2019-01-01"
5        endDate: "2023-02-01"
6        kind: report5112
7      }
8    ) {
9        reportData {
10         header
11         rows
12         formatting {
13           format
14           from
15           to
16           sum
17         }
18       }
19   }
20 }
```


```
1  {
2    "data": {
3      "reportGenerate": {
4        "reportData": {
5          "header": [
6            "Progetto",
7            "Brand",
8            "Canale",
9            "Tipo documento",
10           "N. documento",
11           "Serie",
12           "Data documento",
13           "N. ordine",
14           "Data ordine",
15           "Servizi_22 IMP",
16           "S_IVA_22",
17           "Totale documento",
18           "Codice pagamento",
19           "Descrizione pagamento"
20         ],
21         "rows": [
22           [
23             "Consorzio Parmigiano Reggiano",
24             "Latteria Giacometti",
25             "Delizie non italiche",
26             "Note Accredito emessa",
27             "941",
28             "5",
29             "2023-01-31T08:44:37.040Z",
30             "2",
31             "2023-01-17T08:40:17.291Z",
32             0.0,
33             0.0,
34             "91.8",
35             "407",
36             "adyen"
37           ]
38         ],
```


Figura 5.3: Esempio di *mutation* `reportGenerate` e relativa risposta

Il listato 15 in appendice B, mostra le parti principali dell'implementazione della mutation.

Capitolo 6

Conclusioni

6.1 Consuntivo finale

Viene di seguito riportato il consuntivo orario finale. Gli indici di settimana corrispondono alle settimane reali di calendario.

Tabella 6.1: Consuntivo orario finale

Descrizione attività	**Indice settimana**	**Ore impiegate**
Formazione su strumenti e tecnologie	1-2	24
Comprensione degli obiettivi e piattaforma preesistente	2	8
Analisi dei requisiti	2	8
Progettazione	2-3	32
Sviluppo	3-8	176
Analisi spedizioni paperless	8	8
Progettazione gestione paperless	8	8
Sviluppo gestione paperless	8-9	56
Totale		**320**

Rispetto al preventivo iniziale, riassunto nella tabella 1.1, si evidenziano in particolare le seguenti differenze:

- Il tempo inizialmente preventivato per lo studio e setup dell'ambiente di sviluppo si è rivelato sovrastimato. Tale attività ricade quindi nel periodo di formazione iniziale;
- L'attività di sviluppo comprende quella inizialmente preventivata di test e validazione. Questo perchè il testing delle funzionalità man mano implementato è stato parte integrante dello sviluppo e non una attività a sè stante.

6.2 Raggiungimento degli obiettivi

Nel complesso, tutti gli obiettivi sono stati raggiunti, sia quelli obbligatori sia quelli desiderabili. Il lavoro svolto ha fornito un valore aggiunto al prodotto già esistente, arricchendone le funzionalità e ampliandone le prospettive. Inoltre, grazie alla buona pianificazione e gestione dei tempi, è stato possibile sfruttare il tempo rimanente per perseguire ulteriori obiettivi che non erano stati inizialmente previsti (appendice **??**).

6.3 Conoscenze acquisite

Nei due mesi di stage in Moku è stato possibile acquisire importanti conoscenze tecniche, tra le quali:

- il linguaggio di programmazione Ruby;
- il framework Ruby on Rails;
- implementare e progettare API basate su GraphQL;

Oltre alle competenze tecniche, non trascurabili anche quelle trasversali come:

- autonomia nel ricercare e valutare soluzioni complesse;
- gestione dei tempi;
- lavoro in team.

6.4 Valutazione personale

Durante lo stage in Moku è stato possibile mettere in pratica conoscenze pregresse ma soprattutto acquisirne di nuove. L'ambiente all'interno dell'azienda favorisce la condivisione di conoscenze e stimola il relazionarsi con tutti i membri del team. Nel complesso, quindi, l'esperienza è stata estremamente positiva, grazie anche al supporto del tutor aziendale e dei colleghi.

Appendice A

Gestione paperless delle spedizioni

A.1 Introduzione

Come accennato precedentemente, grazie al tempo rimanente durante il periodo di stage, è stato possibile ampliare ulteriormente le funzionalità del prodotto, in modi non inizialmente previsti ma che si sono resi necessari su indicazioni ed esigenze del cliente.

In particolare è stato implementato un sistema di gestione *paperless* delle spedizioni, ovvero l'invio automatizzato della lista degli ordini da spedire ai corrieri che si occupano della logistica.

A.2 Progettazione e realizzazione

L'invio della lista degli ordini agli spedizionieri, dovrà avvenire ogni giorno ad un orario preciso, il quale dovrà poter essere configurabile con facilità. Inoltre, in base alla destinazione e alla tipologia dell'ordine, è necessario allegare i documenti per l'esportazione. Ogni spedizioniere avrà le proprie modalità di ricezione.

Gli spedizionieri inizialmente trattati saranno:

- *UPS*: ricezione dei dati attraverso il caricamento in una cartella SFTP dedicata;
- *DHL*: ricezione dei dati attraverso e-mail ad un indirizzo predefinito.

A.2.1 Configurazione

I parametri che dovranno essere configurabili per ogni spedizioniere sono:

- orario di invio di ordini ed eventuali documenti;
- canali di vendita da cui provengono gli ordini da trattare.

Il cliente necessità di modificare spesso i canali di vendita associati agli spedizionieri. Al contrario invece, l'orario di invio non è previsto che cambi con molta frequenza.

Alla luce di queste considerazioni, si è deciso di persistere i canali di vendita associati nel database, e l'orario di invio in un file di configurazione YAML.

A.2.2 Classi implementate

Tutte le classi implementate, concernenti la spedizione *paperless*, sono state raggruppate all'interno del modulo `PaperlessShipping`.

Sender

La classe `Sender` implementa le logiche di invio dei dati specifiche per ogni spedizioniere. A tale scopo, le classi specializzate che ereditano da essa sono:

- `UpsSender`: gestisce le logiche di invio per *UPS*;
- `DhlSender`: gestisce le logiche di invio per *DHL*;

Listato 7: Codice della classe `UpsSender`

```
1  module PaperlessShipping
2    module Senders
3      class UpsSender < Sender
4        def send_files orders, list
5          # get destination emali address from configuration...
6
7          files = orders.map(&:paperless_shipping_documents).flatten!
8          email = UpsPaperlessMailer.with(files: files, list: list,
   ↪   address: address).shipping_documents_email
9          email.deliver_later
10       end
11     end
12   end
13 end
```

La classe espone il metodo `send_files`, il quale può venire invocato con una lista di parametri variabile a seconda del tipo specializzato di `Sender`.

Generator

La classe `Generator` si occupa di generare i dati che verranno trasmessi agli spedizionieri, nel formato a loro atteso. A tale scopo, le classi specializzate che ereditano da essa sono:

- `UpsGenerator`: genera i dati per *UPS*;
- `DhlGenerator`: genera i dati per *DHL*;

Listato 8: Codice della classe `UpsGenerator`

```
1  module PaperlessShipping
2    module Generators
3      class UpsGenerator < Generator
4        def generate orders
5          orders.map do |order|
6            # generate required data for each order...
```

```
7          end
8        end
9      end
10   end
11 end
```

La classe espone il metodo `generate`, che invocato insieme alla lista degli ordini, servirà ad ottenere i dati generati.

Processor

La classe `Processor` ha lo scopo di raggruppare le funzionalità fornite dalle altre classi, nascondendo all'utilizzatore le complessità sottostanti ed esponendo un'interfaccia semplificata. È stato quindi applicato il *design pattern* Facade.

Listato 9: Codice della classe `Processor`

```
1  module PaperlessShipping
2    class Processor
3      def initialize carrier_kind
4        @carrier_kind = carrier_kind
5        @carrier = Carrier.find_by kind: @carrier_kind
6        @sender = Senders.for_carrier @carrier_kind
7        @generator = Generators.for_carrier @carrier_kind
8      end
9
10     def process
11       return unless @carrier.docs_send_active &&
         ↪  orders_to_send.length.positive?
12
13       Rails.logger.info "#{@carrier.name} - Processing paperless
         ↪  shipping..."
14       list = @generator.generate orders_to_send
15       if @sender.send_files orders_to_send, list
16         @carrier.update last_docs_send: DateTime.now
17         update_orders
18         Rails.logger.info "#{@carrier.name} - Successfully processed
           ↪  #{orders_to_send.length} orders."
19       else
20         Rails.logger.info "#{@carrier.name} - Failed processing
           ↪  orders"
21       end
22     end
23
24   private
25
26     def orders_to_send
27       # get orders to send from database...
28     end
29
```

```
30      def update_orders
31          # mark orders as processed...
32      end
33    end
34  end
```

Dopo essere stato inizializzata solamente attraverso il parametro `carrier_kind`, che indica lo spedizionire da trattare, l'oggetto di tipo `Processor` espone il metodo `process` che svolge il compito di:

- ottenere gli ordini, insieme ad eventuali documenti allegati, da inviare al corriere;
- terminare l'esecuzione se non ci sono ordini di inviare al corriere;
- generare i dati nel formato previsto attraverso l'oggetto di tipo `Generator` dedicata;
- inviare i dati attraverso l'oggetto di tipo `Sender` secondo le modalità previste;
- marcare gli ordini appena trattati come già processati.

A.2.3 Scheduling

Lo *scheduling* di esecuzione, cioè la programmazione temporale dell'esecuzione del flusso *paperless*, avviene nei seguenti passaggi:

- un Cron job avvia alla mezzanotte di ogni giorno un Rake task;
- il *task* ottiene, per ogni spedizioniere, l'orario di invio dal file di configurazione;
- viene creato un Delayed job che avvierà i `Processor`, uno per ogni corriere, all'orario desiderato.

Listato 10: Codice del *Rake task* che inizializza i `Processor`

```
1   namespace :paperless_shipping do
2     desc 'Schedule paperless shipping send documents to carriers'
3     task schedule: :environment do
4       PaperlessShipping::Carrier.all.each do |carrier|
5         # get send time from configuration...
6
7         # initialize the processor for the current carrier
8         processor = PaperlessShipping::Processor.for_carrier
          ↪  carrier.kind
9
10        # this will run as a delayed job
11        processor.delay(run_at:
          ↪  docs_send_time.strftime('%H:%M')).process
12      end
13    end
14  end
```

Appendice B

Ulteriori listati di codice

B.1 Generazione report

Listato 11: Estratto della classe `Definition`

```
module Reports
  module Definitions
    class Definition

      # ...

      class << self
        def named name
          arguments[:name] = name
        end

        def description description
          arguments[:description] = description
        end

        def header header
          arguments[:header] = header
        end

        def cols from, to, with_formatting, sum: false
          formatting << {from: from, to: to, format: with_formatting,
          ↪  sum: sum}
        end

        def col number, with_formatting, sum: false
          cols number, number, with_formatting, sum: sum
        end

        def sheet_name name
          arguments[:sheet_name] = name
        end
```

```
31
32        def build
33          arguments[:formatting] = formatting
34          new(**@arguments)
35        end
36
37      private
38
39        # ...
40      end
41    end
42  end
43 end
```

Listato 12: Codice della classe `FastExcelBackend`

```
1  module Reports
2    module Excel
3      class FastExcelBackend
4        def initialize
5          @workbook = FastExcel.open constant_memory: true
6          @workbook.default_format.set(
7            font_family: 'Calibri'
8          )
9          @formats = {
10           date: @workbook.number_format('dd/mm/yyyy'),
11           eur:  @workbook.number_format('0.00"€"'),
12           perc: @workbook.number_format('0.00"%"')
13         }
14       end
15
16       # Generates Excel in memory
17       def generate rows, header, formatting, sheet_name
18         add_sheet sheet_name
19         add_header header
20         add_rows rows, formatting
21         add_sums formatting
22       end
23
24       # Saves Excel to file
25       def save filename
26         content = @workbook.read_string
27         File.binwrite filename, content
28       end
29
30     private
31
32       # private methods omited
33
```

```
34       end
35     end
36   end
```

Listato 13: Codice della classe `Generator`

```ruby
module Reports
  class Generator
    def initialize definition, **filters
      @definition  = definition
      @filters     = filters
      @backend     = Excel::FastExcelBackend.new
      @report_data = nil
    end

    # Generates report data
    def generate force=false
      rows         = @definition.generate_report_rows @filters, force
      @report_data = {
        rows:       rows,
        header:     @definition.report_header,
        formatting: @definition.report_formatting
      }
    end

    # Generates report in Excel format from generated data
    def generate_excel *generate_args, filename: nil
      generate(*generate_args) if @report_data.nil?
      filename ||= "#{SecureRandom.uuid}.xlsx"
      @backend.generate @report_data[:rows],
                        @report_data[:header],
                        @definition.report_formatting,
                        @definition.report_sheet_name
      Settings::REPORTS_PATH.mkdir unless
      ↪  Settings::REPORTS_PATH.directory?
      @backend.save Settings::REPORTS_PATH.join filename
      filename
    end

    def self.for_report_kind kind, filters
      definition = Definitions.for_kind kind
      Generator.new definition, **filters
    end
  end
end
```

B.2 GraphQL

Listato 14: Codice delle query `savedReports` e `reportDefinitions`

```
field :saved_reports, [SavedReportType], resolver:
↪  Resolvers::GenericResolver[SavedReport], null: true do
  description 'Gets all saved reports'
end

field :report_definitions, [ReportDefinitionType], null: false do
  description 'Get all report types'
end

def report_definitions
  Reports::Definitions::REGISTRY.map do |kind, definition|
    {
      kind:        kind.to_s,
      name:        definition.report_name,
      description: definition.report_description,
      header:      definition.report_header
    }
  end
end
```

Listato 15: Estratto della mutation `reportGenerate`

```
module Mutations
  module ReportMutations
    class ReportGenerate < Mutations::Base::BaseMutation
      type Types::GeneratedReportType

      argument :input, Types::Inputs::ReportInput, required: false
      argument :saved_report_id, GraphQL::Types::ID, required: false
      validates required: {one_of: [:input, :saved_report_id]}

      extras [:lookahead]

      def resolve input: nil, saved_report_id: nil, lookahead: nil
        generate_excel = lookahead.selects? :excel_url

        if input.nil?
          # initialize filters from saved report...
        else
          # get filters from user input...
        end

        filters = {start_date: start_date, end_date: end_date,
        ↪  channels: channels}

```

```
23          generator   = Reports::Generator.for_report_kind kind,
            ↪  filters
24          report_data = generator.generate
25
26          if generate_excel
27            # generate excel report and save to a temporary file...
28          else
29            # no report url needed, returning null...
30            report_url: nil
31          end
32
33          {
34            excel_url:   report_url,
35            report_data: report_data
36          }
37        end
38      end
39    end
40  end
```

Listato 16: Estratto della mutation `reportSave`

```
1   module Mutations
2     module ReportMutations
3       class ReportSave < Mutations::Base::BaseMutation
4         class ReportSaveInput < Types::Inputs::ReportInput
5           argument :name, GraphQL::Types::String, required: true
6           # other parameters...
7         end
8
9         type Types::SavedReportType
10        argument :input, ReportSaveInput, required: true
11
12        def resolve input: nil
13          # extract and validate filters from user input...
14
15          SavedReport.create! name: name,
16                              start_date: start_date,
17                              end_date: end_date,
18                              kind: kind,
19                              user: current_user,
20                              **source_filters
21        end
22      end
23    end
24  end
```

Acronimi e abbreviazioni

API Application Program Interface. 7, 42

DSL Domain Specific Language. 13, 21, 43

ER Entity Relationship. 15

IDE Integrated Development Environment. 44

OMS Order Management System. 1, 3, 44

ORM Object-Relational Mapping. 8

REST Representational State Transfer. 45

SFTP Secure File Transfer Protocol. 45

YAML YAML Ain't Markup Language. 46

Glossario

Agile è un approccio metodologico nel campo dello sviluppo del software che promuove la flessibilità, la collaborazione e il coinvolgimento attivo del cliente nel processo di sviluppo. L'agile si basa su principi come la consegna incrementale e la rapida risposta ai cambiamenti. Le metodologie agili, come Scrum e Kanban, si concentrano sulla creazione di software funzionante in modo iterativo e sulla gestione delle priorità in base alle esigenze emergenti, consentendo un'adattabilità maggiore rispetto ai modelli tradizionali di sviluppo del software. L'approccio agile favorisce la collaborazione, la comunicazione e l'auto-organizzazione dei team, consentendo un miglioramento continuo del prodotto e una maggiore soddisfazione del cliente . 4, 42

API *(Application Programming Interface)* è un insieme di regole, protocolli e strumenti che permettono la comunicazione tra diverse applicazioni software. Una *API* definisce le modalità con cui le diverse componenti di un sistema possono interagire tra loro, consentendo ad una applicazione di utilizzare i servizi offerti da un'altra senza la necessità di conoscere i dettagli interni di quest'ultima. Le *API* fungono da intermediari, consentendo lo scambio di dati e funzionalità tra le diverse applicazioni, semplificando lo sviluppo di software complesso e favorendo l'integrazione tra sistemi eterogenei. . 31, 41

benchmark è un processo o un set di misurazioni standardizzato, utilizzato per valutare e confrontare le prestazioni di un sistema, un'applicazione o in generale un componente hardware o software. Attraverso l'esecuzione di test specifici, un *benchmark* fornisce un punto di riferimento obiettivo per valutare le capacità e le prestazioni di un sistema o componente, consentendo agli sviluppatori e agli utenti di confrontare diverse soluzioni e prendere decisioni informate sulla base dei risultati ottenuti . 14, 42

Cron job Un *cron job* è un processo automatico che viene eseguito in modo programmato e periodico su un sistema operativo Unix-like. Consiste in un comando o uno script che viene pianificato tramite il *cron scheduler* per essere eseguito a intervalli specifici, come ogni minuto, ogni ora o ogni giorno, in base alla configurazione desiderata. I cron job sono spesso utilizzati per automatizzare compiti ripetitivi o pianificare l'esecuzione di script, programmi o altre attività nell'ambito dell'amministrazione di sistema e dello sviluppo software . 35, 42

DBMS *Database Management System* è software progettato per consentire la creazione, l'organizzazione, la gestione e l'interrogazione di grandi quantità di dati strutturati. Un *DBMS* offre un'interfaccia che permette agli utenti di accedere e

manipolare il contenuto del database in modo sicuro, consentendo loro di svolgere tutte le operazione in maniera sicura, veloce ed efficiente . 16, 42, 45

Delayed job è una libreria per il framework Ruby on Rails che consente di eseguire operazioni in background in modo asincrono. Spesso utilizzato per elaborare compiti di lunga durata o intensivi dal punto di vista delle risorse, permette di spostare le operazioni che richiedono molto tempo al di fuori del flusso principale dell'applicazione. Attraverso *Delayed Job*, è possibile creare e programmare una coda di task, in cui ognuno rappresenta un'operazione che deve essere eseguita in un secondo momento. La coda dei task può includere attività come l'invio di e-mail, l'elaborazione di grandi quantità di dati o in generale l'esecuzione di compiti complessi . 27, 35, 43

Dependency injection è un design pattern utilizzato per gestire le dipendenze tra le classi. Consiste nell'iniettare le dipendenze di una classe da parte di un'entità esterna, anziché farle istanziare internamente. In pratica, le dipendenze vengono passate come parametri al momento della creazione o attraverso metodi setter. Questo approccio favorisce la modularità, la separazione delle responsabilità e la facilità di testabilità delle classi, in quanto rende le dipendenze esplicite e sostituibili . 24, 43

Drupal è un sistema di gestione dei contenuti open source e altamente personalizzabile, utilizzato per la creazione e la gestione di siti web. Fornisce un'ampia gamma di funzionalità e moduli predefiniti che consentono agli utenti di creare e organizzare contenuti ed inoltre implementare funzionalità di e-commerce . 9, 43

DSL *(Domain Specific Language)* è un tipo di linguaggio di programmazione progettato per essere utilizzato in domini specifici. Viene creato per fornire un linguaggio che sia ottimizzato per il dominio di interesse, offrendo un'interfaccia più intuitiva e semplificata per descrivere concetti, strutture ed operazioni rilevanti . 21, 41

ER è un modello concettuale utilizzato nell'ambito dei database per descrivere le relazioni tra le entità all'interno di un sistema. L'*ER* si basa su due concetti principali: le entità, che rappresentano oggetti o concetti del mondo reale, e le relazioni, che indicano come le entità sono connesse tra loro. Attraverso esso, è possibile identificare e definire gli attributi delle entità, nonché le cardinalità e i vincoli che regolano le interazioni tra di esse . 41, 43

Excel *Excel* è un software di foglio di calcolo sviluppato da Microsoft, ampiamente utilizzato per organizzare, analizzare e manipolare dati numerici e testuali. Questo strumento è ampiamente utilizzato in settori come la contabilità, la finanza, le risorse umane, la ricerca scientifica e molti altri, offrendo agli utenti la possibilità di automatizzare i calcoli, semplificare la gestione dei dati e generare report efficaci . iii, 10, 43

Facade è un design pattern che fornisce un'interfaccia semplificata per interagire con un sottosistema complesso o una libreria di codice. L'obiettivo principale di facade è quello di semplificare l'uso di un sistema complesso, nascondendone la complessità sottostante e fornendo un punto di accesso unificato. Questo pattern migliora la manutenibilità del codice, promuovendo la separazione delle responsabilità ed in generale riducendo le dipendenze delle varie componenti di un sistema da quelle più interne e critiche . 34, 43

Frontend è la componente di un'applicazione responsabile della presentazione e dell'interazione con gli utenti. È quindi il lato visibile e interattivo del software, attraverso il quale gli utenti possono interagire con le varie funzionalità previste dall'applicazione. . 3, 14, 28, 44

Git nell'ambito dello sviluppo software, *Git* è un sistema di controllo versione distribuito che consente agli sviluppatori di tenere traccia delle modifiche apportate ai loro file nel corso del tempo, facilitando il lavoro collaborativo e la gestione del codice sorgente. Permette di creare un *repository*, un archivio in cui vengono conservate tutte le versioni di un progetto. Ogni modifica apportata ai file viene registrata come un *commit* all'interno del repository, consentendo agli sviluppatori di tornare a qualsiasi punto nel tempo e di confrontare le diverse versioni del codice . 5, 44

HTTP il protocollo *HTTP (Hypertext Transfer Protocol)* è un protocollo di comunicazione utilizzato per consentire il trasferimento di informazioni tra client e server. È il protocollo su cui si basa la comunicazione tra il browser di un utente e i server web che ospitano le risorse richieste. HTTP permette la trasmissione di documenti ipertestuali, come pagine web, attraverso le richieste del client e le risposte fornite dal server. Le richieste contengono informazioni sulle azioni desiderate, come recuperare una pagina o inviare dati, mentre le risposte contengono informazioni sullo stato dell'operazione richiesta e i dati richiesti. È un protocollo *stateless*, quindi senza stato, il che significa che ogni richiesta e risposta sono indipendenti l'una dall'altra . 12, 44

IDE nell'ambito dello sviluppo software, un *IDE, Integrated Development Environment* è un software che fornisce un ambiente completo per lo sviluppo. Esso combina un *editor* di testo con funzionalità di autocompletamento del codice, debug, compilazione e testing, offrendo agli sviluppatori uno strumento centralizzato per la scrittura, la modifica e la gestione del codice. Gli IDE spesso includono anche funzionalità per navigare facilmente nel codice e l'integrazione con strumenti esterni come compilatori e librerie . 6, 41

JSON *(JavaScript Object Notation)* è un formato di testo leggibile dalle macchine e dalle persone, utilizzato per la rappresentazione di dati strutturati. È ampiamente utilizzato dove è necessario lo scambio di informazioni tra applicazioni e sistemi eterogenei. I dati in formato JSON sono organizzati in coppie chiave-valore e possono includere oggetti annidati, *array* e tipi di dati primitivi come stringhe, numeri, booleani e *null*. Grazie alla sua semplicità e leggibilità, JSON è diventato uno standard de facto per l'interoperabilità tra diverse piattaforme e linguaggi di programmazione . 12, 16, 44

Metaclasse è una classe speciale in programmazione orientata agli oggetti che definisce il comportamento delle classi stesse. In altre parole, una metaclasse è una classe di livello superiore che controlla la creazione, la struttura e il comportamento delle altre classi all'interno di un programma. . 21, 44

OMS *(Order Management System)* è un software utilizzato per gestire in modo efficiente il ciclo di vita degli ordini, dal momento della loro ricezione fino alla

loro evasione e consegna. Un *OMS* consente di centralizzare e automatizzare diverse attività legate agli ordini, tra cui la registrazione delle richieste dei clienti, l'evasione degli ordini e la gestione dei resi . iii, 9, 10, 12, 41

ORM *(Object-Relational Mapping)* è una tecnica di programmazione per facilitare la gestione e l'interazione tra un database relazionale e un'applicazione orientata agli oggetti. Un *ORM* agisce come uno strato di astrazione che mappa le entità del database, come tabelle e relazioni, in oggetti del linguaggio di programmazione utilizzato, consentendo agli sviluppatori di manipolare i dati come se fossero oggetti comuni. Questo approccio semplifica il processo di interrogazione e manipolazione dei dati, riducendo la complessità dello sviluppo e migliorando la produttività. L'ORM offre anche funzionalità aggiuntive come la gestione delle transazioni e la generazione automatica del codice *SQL* sottostante, contribuendo a ridurre l'onere di lavoro e fornendo un'interfaccia più intuitiva per l'interazione con il database. . 41, 45

PostgreSQL è un DBMS relazionale ad alte prestazioni e open source. È progettato per gestire grandi volumi di dati e supporta numerose estensioni al linguaggio SQL standard. *PostgreSQL* offre una vasta gamma di funzionalità, tra cui il controllo dell'integrità sui dati, transazioni, gestione avanzata degli indici e supporto per le estensioni personalizzate. È noto per la sua affidabilità, scalabilità e capacità di adattarsi a una varietà di casi d'uso . 16, 45

Rake task è un'azione automatizzata che può essere eseguita tramite il sistema di build e task management Rake. Rake è uno strumento ampiamente utilizzato nel framework Ruby on Rails e in altri progetti Ruby per definire e gestire compiti di routine. Questi compiti possono includere operazioni come la creazione di database, l'importazione ed esportazione di file, la generazione di report e l'esecuzione di test . 35, 45

Report un *report* è un documento formale che presenta in modo organizzato e dettagliato le informazioni, le analisi e le conclusioni relative a uno specifico argomento o attività. Nell'ambito delle vendite online è un documento che fornisce una panoramica dettagliata delle attività di vendita e delle performance di un'azienda . 1, 3, 45

REST è un'architettura software utilizzata per progettare servizi web scalabili, modulari e interoperabili che sfruttano il protocollo HTTP come base di comunicazione. Si basa su principi chiave come l'uso di risorse identificabili univocamente tramite *URI (Uniform Resource Identifier)* e l'adozione di operazioni standard del protocollo HTTP, come GET, POST, PUT e DELETE, per manipolare le risorse. Inoltre, REST promuove l'uso dei concetti di stato e rappresentazione, consentendo ai client di accedere e manipolare lo stato delle risorse tramite rappresentazioni dei dati in formati comuni come JSON o XML . 7, 41

SFTP il protocollo *SFTP (Secure File Transfer Protocol)* è un protocollo sicuro per il trasferimento di file su una rete. Sostanzialmente, fornisce le stesse caratteristiche e funzionalità di FTP, garantendo però la cifratura delle informazioni in transito. È ampiamente utilizzato nelle applicazioni e negli ambienti in cui la sicurezza e l'integrità dei dati sono prioritari, come nel trasferimento di file sensibili, nei backup remoti e nella gestione dei server . 32, 41

SQL *(Structured Query Language)* è un linguaggio di interrogazione e programmazione utilizzato per la gestione e la manipolazione dei dati in un database relazionale. Fornisce un insieme di comandi che consentono di creare, modificare e interrogare le tabelle di un database, eseguire operazioni di inserimento, aggiornamento ed eliminazione dei dati, nonché definire vincoli e relazioni tra le tabelle. . 8, 45, 46

H-Farm è un'azienda italiana che si occupa di innovazione digitale e imprenditorialità nel settore dell'informatica. Fondata nel 2005, ha sede nella campagna veneta, vicino a Venezia. Essa opera come un vero e proprio ecosistema digitale, offrendo una combinazione unica di servizi che spaziano dalla formazione all'accelerazione di startup, dalla consulenza all'investimento in progetti innovativi . 1, 46

XLSX il formato di file *XLSX* è utilizzato per rappresentare dati tabulari all'interno di un foglio di calcolo, ed è comunemente associato al software. La sua flessibilità e interoperabilità lo rendono uno dei formati più diffusi per la gestione e l'analisi dei dati nel contesto informatico . 3, 13, 23, 24, 26, 46

YAML *(YAML Ain't Markup Language)* è un formato di serializzazione dei dati leggibile dalle persone e facilmente interpretabile dalle macchine. È spesso utilizzato per rappresentare dati strutturati, come configurazioni di software, in modo conciso e comprensibile. YAML utilizza una sintassi semplice basata su indentazione e caratteri speciali . 32, 41

Bibliografia

Siti web consultati

Documentazione di Ruby: https://docs.ruby-lang.org

Guide di Ruby on Rails: https://guides.rubyonrails.org/

Documentazione di Ruby on Rails: https://api.rubyonrails.org

Documentazione della libreria graphql-ruby: https://graphql-ruby.org/guides

Documentazione della libreria fastexcel: https://github.com/Paxa/fast_excel

RubyGems: https://rubygems.org/

Wikipedia: https://www.wikipedia.org/