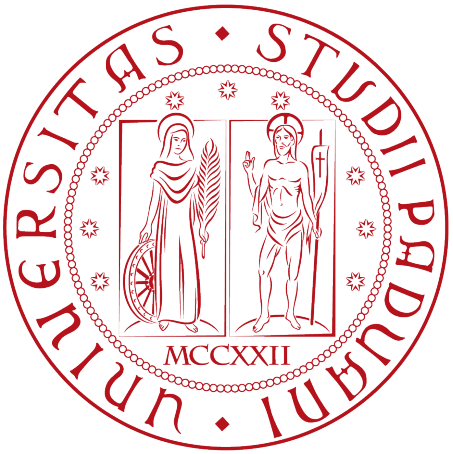
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Analisi, progettazione e sviluppo di un order management system

Tesi di laurea

Relatore

Prof. Paolo Baldan

Laureando

Francesco Protopapa

Anno Accademico 2021-2022

Sommario

Il presente documento descrive il lavoro svolto da Francesco Protopapa durante lo stage svolto presso l'azienda Moku S.r.l. della durata di circa trecentoventi ore. Durante questo periodo, lo stagista si è unito ad un team di sviluppo composto da professionisti e ha collaborato alla progettazione e all'implementazione di parte del back-end$_G$ di un sistema di gestione degli ordini.
Il back-end del sistema è stato realizzato utilizzando tecnologie come Ruby on Rails e GraphQL con il fine di ottenere un prodotto in grado di processare ordini provenienti da diversi siti di vendita e di gestirne fatturazione e spedizione. All'interno del documento, a seguito di un'introduzione al contesto applicativo, vengono ripercorse le fasi di analisi, progettazione, sviluppo e testing, le quali hanno portato alla realizzazione del prodotto. Infine, vengono fatte alcune considerazioni conclusive sull'esito dello stage, sullo stato del prodotto realizzato e sul raggiungimento degli obiettivi prefissati.

Ringraziamenti

Ringrazio i miei genitori per tutti gli sforzi e i sacrifici che hanno fatto per me in tutti questi anni, senza il vostro supporto non avrei mai raggiunto questo traguardo.

Ringrazio mia sorella Chiara per essermi stata sempre vicina, nei bei momenti ed in quelli più difficili.

Ringrazio tutti i miei altri familiari, soprattutto nonna Etta e nonna Maria per avermi sempre sostenuto e i nonni Francesco e Nicola perché anche se non hanno potuto assistere al raggiungimento di questo mio traguardo, sono sicuro che sarebbero stati orgogliosi di me.

Ringrazio tutti i miei amici per i bei momenti passati insieme in questi anni.

Ringrazio il prof. Paolo Baldan per i preziosi consigli ed il supporto fornitomi durante la stesura della presente tesi.

Infine, desidero ringraziare tutto il team di Moku per la bellissima esperienza di stage.

Padova, Luglio 2022 Francesco Protopapa

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Capitolo 1

Introduzione

1.1 L'azienda

Figura 1.1: Logo Moku S.r.l.

Moku S.r.l. è una software house di circa 20 dipendenti con sede a Treviso. L'azienda lavora a stretto contatto con i clienti utilizzando metodologie di lavoro Agile$_G$, per ogni progetto viene formato un team composto da un project manager, uno o più sviluppatori back-end, uno o più sviluppatori front-end$_G$ ed un designer. In questa azienda gli stagisti vengono inseriti all'interno del team di un progetto commissionato da un cliente in modo da avere un'esperienza di lavoro reale, con la possibilità di potersi confrontare con colleghi più esperti.

Metodo di lavoro

Moku S.r.l. lavora seguendo il framework Scrum, un modello di sviluppo Agile basato sul Lean Thinking che utilizza pratiche consolidate e ben documentate. Il lavoro è suddiviso in sprint della durata di due settimane. Al termine di ciascuno sprint si effettua una sprint review con il cliente in cui vengono mostrati i risultati ottenuti durante le due settimane, si risolvono eventuali dubbi emersi e si effettua una pianificazione dello sprint successivo. A metà dello sprint, se necessario, si effettua una mid-sprint review con il cliente durante la quale si discute dei dubbi emersi nella prima settimana dello sprint in corso. All'inizio di ciascuna giornata lavorativa si svolge uno stand-up meeting, un incontro di circa 15 minuti fra tutti i componenti del team in cui ciascuno descrive lo stato del proprio lavoro, eventuali problemi che si sono presentati e gli obiettivi per la giornata.

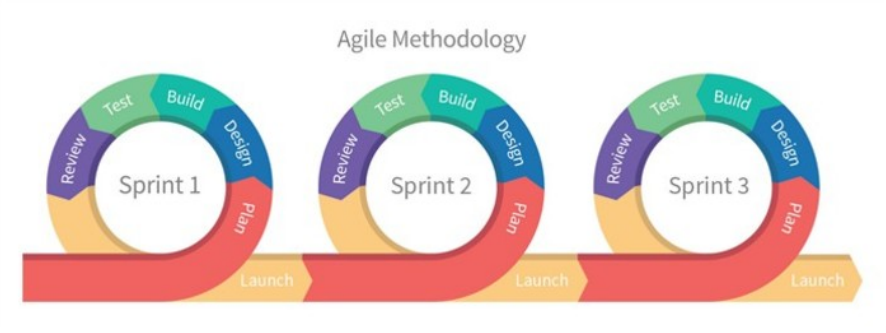


Figura 1.2: Metodologia Agile

1.2 Il progetto

Abstract

Il lavoro svolto durante lo stage presso l'azienda Moku S.r.l., riguarda lo sviluppo per conto di una rilevante azienda nel mondo degli e-commerce management, di un sistema informativo per gestire i flussi di acquisto online in modo scalabile, modulare ed estensibile. Il cliente gestisce migliaia di ordini e-commerce ogni giorno, per conto di decine di clienti, su un ampio numero di canali (siti e-commerce custom, Shopify, Amazon, eBay ecc.), 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 clienti anche la logistica (con propri magazzini), l'assistenza cliente ed il marketing. Il progetto prevede la realizzazione di un sistema informativo, che per semplicità verrà definito Order Management System (OMS)$_G$, il quale si occupi 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 sopra menzionati (logistica, assistenza clienti, amministrazione, marketing). Tale applicativo dovrà poi interfacciarsi ad altri software verticali per ottemperare a funzionalità specifiche come fatturazione e spedizione.

Obiettivi

Lo stage aveva come obiettivo quello di analizzare la situazione attuale del sistema utilizzato dal cliente al fine di progettare ed implementare una soluzione che rispettasse i requisiti forniti. Oltre a questo, era previsto che lo stagista imparasse a lavorare secondo il framework Scrum coordinandosi in modo diretto con il cliente finale. Anche se non obbligatorio, era desiderabile che durante lo stage venisse realizzata una suite di testing del software prodotto ed una documentazione completa.

Lavoro svolto

Nel momento in cui ho iniziato lo stage, il progetto era già stato avviato da circa un mese, durante il quale è stata svolta gran parte dell'analisi dei requisiti richiesti dal cliente. Nei due mesi successivi mi sono unito al team di sviluppo e ho collaborato alla realizzazione di una parte del back-end del prodotto.

1.3 Principali problematiche

Attualmente il cliente di Moku S.r.l. gestisce migliaia di ordini provenienti diversi canali di vendita per conto di decine di clienti. In particolare, gli ordini provengono da siti diversi quali Kmaori, Prestashop, Woocommerce, Magento e BindCommerce. Per la gestione della logistica, il cliente utilizza:

* Agilis$_G$, ovvero un Warehouse Management System (WMS) che gestisce la parte fiscale;
* Gsped$_G$, una piattaforma SaaS per la gestione delle spedizioni.

Questi software attualmente sono integrati agli e-commerce del cliente attraverso dei plugin, la comunicazione tra i siti e i software di logistica avviene tramite File Transfer Protocol (FTP)$_G$ e gli ordini vengono inviati in formato eXtensible Markup Language (XML)$_G$, mentre la documentazione viene generata in formato PDF.

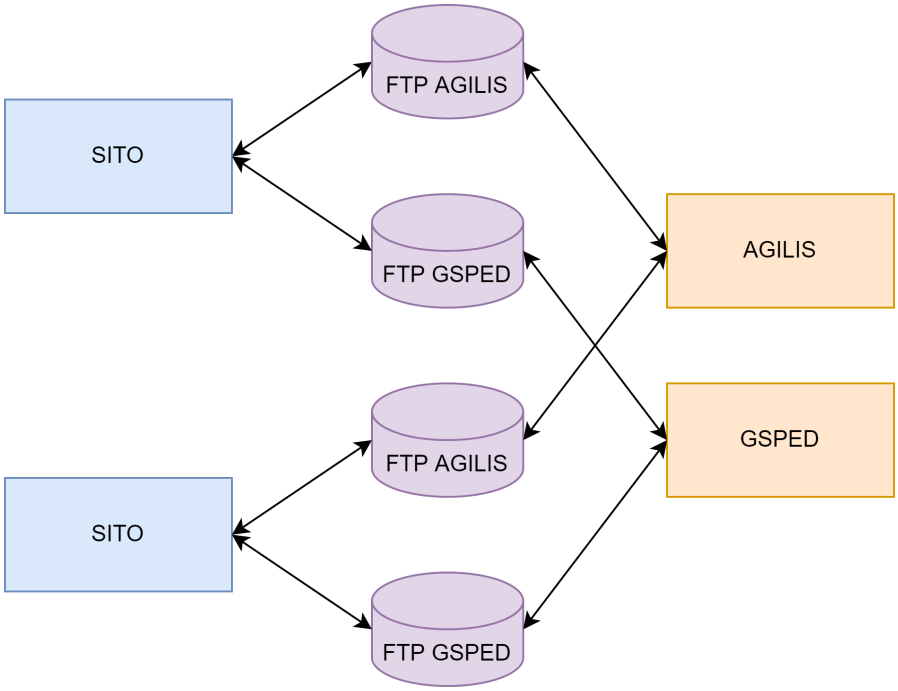


Figura 1.3: Architettura pre-OMS

Il cliente desidera estendere le funzionalità del sistema attualmente in uso in modo tale da:

* raccogliere gli ordini provenienti dai vari canali di vendita per renderli consultabili da backoffice a seconda dei permessi dei vari utenti;
* intercettare gli ordini anomali prima che arrivino ai software di logistica per poterli modificare e correggere. Questa richiesta è dovuta al fatto che gli ordini provenienti dai siti possono avere informazioni mancanti, contenere degli errori o essere sospetti di frode;

* migliorare la gestione delle disponibilità dei prodotti all'interno dei siti di vendita. Attualmente un prodotto può essere venduto su più canali di vendita, i quali non sanno se quel prodotto è stato già venduto su un altro sito e vengono a conoscenza della disponibilità reale solamente una sola volta al giorno tramite un file XML inviato da Agilis;
* calcolare automaticamente le tasse che il cliente di Moku S.r.l. applica ai propri clienti, questo compito attualmente è svolto manualmente da un operatore.

1.4 Soluzione scelta

La soluzione scelta è quella di realizzare un sistema di gestione degli ordini (OMS) posto nel mezzo tra i canali di vendita e i software di logistica. L'OMS ha il compito di raccogliere, elaborare e persistere:

* gli ordini provenienti dai vari canali di vendita;
* la documentazione prodotta da Agilis e Gsped;
* le informazioni relative alle disponibilità dei prodotti prodotte da Agilis;
* le informazioni relative allo stato delle spedizioni prodotte da Gsped.

Nella prima fase del progetto, l'OMS si integra al sistema già esistente in modo completamente retrocompatibile, per fare questo è necessario duplicare i server FTP esistenti in modo che i canali di vendita vadano ad interagire con un server FTP diverso rispetto a quello con cui si interfacciano i software di logistica. È quindi compito dell'OMS inviare i file XML degli ordini ai server FTP osservati dai software di logistica e caricare sui server FTP dei canali di vendita tutti i file prodotti da Agilis e Gsped. In questo modo è possibile rendere disponibili le informazioni sugli ordini a backoffice e permetterne la modifica, inviare ai canali di vendita informazioni sulle disponibilità dei prodotti con maggiore frequenza e calcolare automaticamente le tasse applicate ai clienti. Per fare questo, è solamente necessario cambiare i server FTP sui quali si interfacciano i canali di vendita e i software di logistica senza modificare in nessun modo le logiche già esistenti.

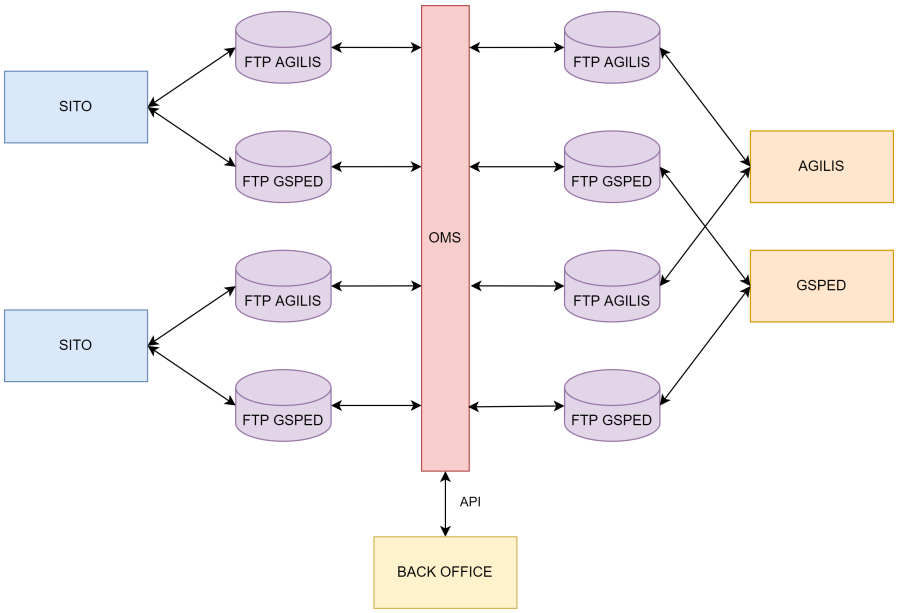


Figura 1.4: Architettura iniziale OMS

Nella fase successiva, l'idea è quella di fare comunicare l'OMS con i siti di vendita e i software di logistica esclusivamente tramite Application Program Interface (API)$_G$, tuttavia questa fase si svolgerà in un periodo successivo al mio stage e quindi non verrà trattata ulteriormente.

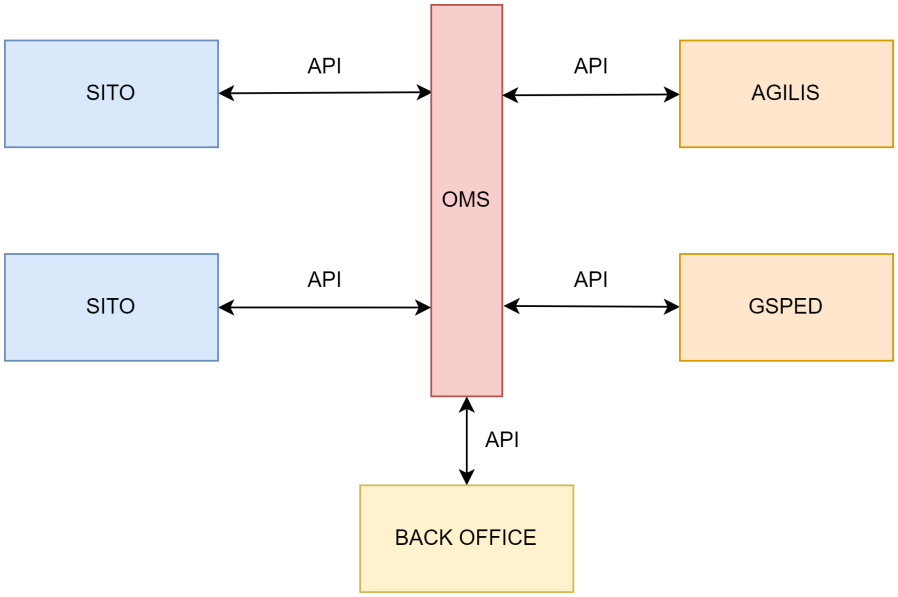


Figura 1.5: Architettura finale OMS

1.5 Strumenti e tecnologie utilizzate

Git con il paradigma Git Flow

Git è un sistema di controllo di versione open source progettato per avere a che fare con ogni tipo di progetto in modo veloce ed efficiente. Nel corso del progetto è stato utilizzato il paradigma Git Flow andando a creare un feature branch per ogni task risolto.

GraphQL

GraphQL è un linguaggio di query per API sviluppato da Facebook come alternativa a REST. Il punto di forza di GraphQL è quello di poter effettuare query che richiedono esattamente quello di cui sia ha bisogno, rendendo le applicazioni che lo utilizzano più veloci e stabili.

Ruby on Rails

Ruby on Rails è un framework open source per lo sviluppo web scritto in Ruby. Il framework è stato creato da David Heinemeier Hansson e la sua architettura è basata sul pattern Model-View-Controller. Il principio cardine sul quale è fondato il framework è "convention over configuration", questo significa che seguendo determinate convenzioni non ci sarà bisogno di effettuare alcuna configurazione dato che il framework si occuperà di farlo.

Bitbucket

Bitbucket è un servizio hosting di repository per progetti che usano i sistemi di controllo versione Mercurial o Git. È stato preferito a GitHub perché essendo un software di proprietà di Atlassian si integra più facilmente con altri strumenti utilizzati dall'azienda come ad esempio Jira.

Jira

Jira è un software proprietario prodotto e sviluppato da Atlassian che permette di gestire e monitorare i progetti di software. Tramite le Scrum Board, Jira facilita la gestione di progetti che utilizzano metodologie di lavoro Agile, permettendo di suddividere task complessi in sotto-task che possono essere assegnati ai membri del team e svolti in un determinato sprint.

1.6 Descrizione del prodotto ottenuto

Date le grandi dimensioni del progetto, il prodotto ottenuto nei due mesi del mio stage non è ancora pronto per il rilascio, questo era stato preventivato ancora prima dell'inizio dello stage e non è dovuto ad alcuna problematica riscontrata. Il progetto infatti prevede di andare avanti ancora per diversi mesi prima di arrivare ad un rilascio effettivo e successivamente è prevista una fase di miglioria e manutenzione del prodotto che andrà avanti per diversi anni. Il lavoro svolto durante il mio stage costituisce quindi una solida base per il back-end del progetto. Attualmente il back-end dell'OMS è in grado di porsi nel mezzo della comunicazione tra i siti di venditi e i software di

logistica come descritto in precedenza. Sono state implementate e testate le seguenti funzionalità:

* lettura, salvataggio e inoltro ai software di logistica degli ordini in arrivo;
* query e mutation GraphQL per la visualizzazione e modifica dei dati degli ordini;
* gestione delle disponibilità dei prodotti con invio di un file XML ai canali di vendita sia quando si riceve un file di disponibilità da Agilis che quando si riceve un ordine;
* lettura, salvataggio e inoltro ai canali di vendita della documentazione prodotta dai software di logistica;
* calcolo delle tasse per i clienti a partire dai documenti di vendita.

1.7 Struttura del resto della relazione

Il secondo capitolo descrive l'analisi dei requisiti.

Il terzo capitolo approfondisce la fase di progettazione attraverso l'uso di diagrammi ER.

Il quarto capitolo descrive la fase di codifica e testing, evidenziando i punti di forza del framework utilizzato.

Il quinto capitolo presenta le considerazioni finali sul progetto.

Convenzioni tipografiche

Tutte le abbreviazioni, gli acronimi e i termini di uso non comune vengono definiti all'interno del glossario situato alla fine del documento, la prima occorrenza di un termine presente nel glossario utilizza la seguente nomenclatura: parola_{G} .

Capitolo 2

Analisi dei requisiti

L'analisi dei requisiti è stata effettuata nel mese antecedente all'inizio del mio stage. Nel momento in cui mi sono unito al progetto, era già stata verificata la fattibilità dei requisiti forniti dal cliente i quali sono stati trasformati in user stories.

2.1 Analisi dei requisiti secondo la metodologia Agile

Secondo la metodologia di lavoro Agile, i requisiti sono espressi ad alto livello con frasi brevi che contengono solo le informazioni più importanti, senza dover produrre documentazione complessa e formale. Questo permette di facilitare la comunicazione con il cliente che potrebbe non essere in grado di comprendere i dettagli tecnici. I requisiti vengono delineati in buona parte all'inizio del progetto, tuttavia, è possibile raccoglierne di ulteriori nel corso dei vari sprint. I requisiti vengono espressi tramite user stories le quali contengono una breve descrizione dell'interazione di un utente o sistema esterno col sistema. A differenza dei casi d'uso, le user stories sono scritte in linguaggio naturale e sono molto più semplici, meno esaustive e ad alto livello. Ogni user story viene associata a molteplici task che una volta svolti ne determinano il completamento, ogni task è pensato per essere svolto all'incirca in una giornata lavorativa, suddividere una user story in più task ne facilita dunque la stima del costo.

2.2 User stories

La tabella 2.1 illustra le user stories create nel mese prima dell'inizio del mio stage e quelle che sono state aggiunte negli sprint successivi. I codici utilizzati per identificare le user stories sono gli stessi che vengono utilizzati per identificare i task di ciascuna user story, per questo i numeri dei codici non risultano essere consecutivi.

2.3 Task

La tabella 2.2 illustra i task associati alle user stories svolti da me durante lo stage. I codici utilizzati per identificare i task seguono lo stesso modello dei codici utilizzati per identificare le user stories, ad ogni task è associata una user story.

Tabella 2.1: Tabella delle user stories

Codice	User story
OMS-15	L'utente desidera visualizzare i progetti, brand e canali
OMS-21	L'utente vuole visualizzare i documenti associati all'ordine
OMS-25	L'utente desidera vedere gli ordini arrivati al canale via XML
OMS-30	L'utente vuole visualizzare i dati strutturati della fattura
OMS-33	L'ecommerce manager e l'amministrazione del cliente vogliono avere i dati aggiornati mensilmente sulle fee di progetto
OMS-42	L'order manager vuole modificare gli ordini da attenzionare
OMS-48	L'amministratore vuole visualizzare il log delle modifiche all'ordine (automatici o fatti da un utente)
OMS-50	Il canale vuole avere la disponibilità prodotti aggiornata
OMS-61	L'amministrazione vuole avere l'archivio prodotti minimale nell'OMS per gestire made_in, customs_code, filtraggio sku e altri campi non presenti nello shop
OMS-76	L'OMS vuole ricevere lo stato dell'ordine aggiornato

Tabella 2.2: Tabella dei task

Codice	Task	User Story
OMS-26	Modelli ordine e riga d'ordine	OMS-25
OMS-27	Graphql ordine	OMS-25
OMS-67	modello FTP e relativo Active Admin	OMS-25
OMS-70	Ftp-worker	OMS-25
OMS-111	Modellazione generica di una fee e calcolo	OMS-33
OMS-154	Modello documento di vendita (fattura, corrispettivo, nota di credito)	OMS-30
OMS-155	Creazione revenue mensile storicizzata e agganciarla al calcolo fee	OMS-30
OMS-88	Lettura dei documenti da FTP ingresso e caricamento su FTP uscita	OMS-21
OMS-85	Mutation per modificare i campi di ordine, riga d'ordine, inclusi billing, shipping, metadati.	OMS-42
OMS-91	Gestire le politiche di dependence in caso di cancellazioni Ordini/Canli/Brand	OMS-42
OMS-138	Mutation per rivalidare l'ordine dopo la modifica	OMS-42
OMS-143	Verificare la validità del codice fiscale	OMS-42
OMS-145	Mutation per ignorare gli errori e mandare avanti l'ordine	OMS-42
OMS-94	Generazione XML ed invio verso gli shop quando viene ricevuto un nuovo ordine (e viene attenzionato)	OMS-50
OMS-95	Aggiunta disponibilità nel modello Product	OMS-50
OMS-96	Aggiornamento disponibilità da Agilis	OMS-50
OMS-105	Worker per lettura degli XML con gli stati	OMS-76
OMS-73	Importazione CSV prodotti (incrementale e con cancellazione)	OMS-61
OMS-131	Aggiungere sku alla lista prodotti se non presente per progetti mono-brand	OMS-61

2.4 Esito dell'analisi

Nel periodo che ha preceduto il primo sprint a cui ho preso parte, assieme al cliente è stata chiarita quale fosse la corretta relazione tra progetti, brand e canali. A seguito di questa discussione, si è compreso come un progetto sia un utente (cliente) del cliente di Moku S.r.l. e che ad ogni progetto possano essere associati diversi brand, mentre ogni ordine passa attraverso un solo canale e può essere relativo a diversi brand. Oltre a questo, il cliente ha specificato quali dati attualmente presenti nei file inviati dai siti di vendita ritiene superflui in modo da non considerarli in fase di progettazione.
Successivamente, il cliente ha spiegato le corrette configurazioni dei server FTP, specificando che ne esistono di due tipi. I server utilizzati da Agilis hanno una cartella nella quale i siti di vendita inviano gli ordini, una cartella in cui Agilis carica la documentazione e una cartella in cui Agilis carica le disponibilità dei prodotti. I server utilizzati da Gsped invece hanno una cartella nella quale i siti di vendita inviano gli ordini, una cartella in cui Gsped carica la documentazione e una cartella in cui Gsped carica gli stati delle spedizioni.
In seguito, il cliente ha chiarito come attualmente effettua il calcolo delle tasse imposte ai suoi clienti. Le tasse sono imposte mensilmente sulla base del fatturato con o senza iva a seconda del cliente. Oltre a questo, ciascun cliente può avere un modo diverso con cui queste tasse vengono calcolate, queste diverse tipologie di calcolo verranno discusse nel dettaglio nel capitolo successivo. Infine, il cliente ha spiegato che il fatturato viene calcolato sommando i valori presenti nei documenti di vendita di ogni mese, i quali vengono prodotti da Agilis, specificando quali valori presenti in questi documenti sono necessari al calcolo corretto del fatturato mensile.

2.5 Strumenti a supporto dell'analisi

Google Meet

Google Meet è un'applicazione utilizzata per effettuare videochiamate sviluppata da Google, tale applicazione in aggiunta alla condivisione di audio e video permette agli utenti di condividere il proprio schermo e ha il vantaggio di funzionare direttamente sul browser non richiedendo alcuna installazione. Durante il mio stage questa applicazione è stata utilizzata per effettuare le sprint review con il cliente e gli stand-up meeting con i colleghi in smart-working.

Slak

Slak è un software di collaborazione aziendale utilizzato per inviare messaggi in modo istantaneo ai membri del team. Il principale vantaggio nell'utilizzo di Slak è quello di poter organizzare la comunicazione del team attraverso canali specifici accessibili solo a determinate persone, oltre a questo con Slak è possibile inviare messaggi privati ad uno specifico membro del team.

Capitolo 3

Progettazione

Seguendo la metodologia di lavoro Agile, la progettazione è stata svolta nella prima parte di ogni sprint per essere poi seguita da codifica e testing. I paragrafi successivi descrivono dunque come si è evoluta l'architettura del back-end del prodotto nel corso dei quattro sprint svolti durante il mio stage.

3.1 L'influenza di Rails sull'architettura

L'architettura del back-end è fortemente influenzata dall'utilizzo di Ruby on Rails come framework per la codifica. Dato che in Rails ogni classe è associata ad una tabella del database, si è scelto di progettare l'architettura del back-end mediante l'uso di un diagramma ER. Rails inoltre mette a disposizione metodi getter e setter che verranno quindi omessi, inoltre il framework identifica ogni classe con un attributo ID che viene generato automaticamente e permette di non dover individuare chiavi primarie nella tabella.

3.2 Notazione utilizzata

3.2.1 Classi

Per rappresentare una classe viene utilizzata la notazione Unified Modeling Language (UML)$_G$ in cui una classe è raffigurata da un rettangolo suddiviso in tre parti delle quali:

* quella più in alto contiene il nome della classe;
* quella nel mezzo contiene gli attributi;
* quella in basso contiene i metodi.

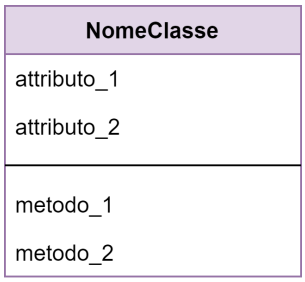


Figura 3.1: Esempio diagramma di una classe

3.2.2 Relazioni uno a uno

Le relazioni uno a uno sono rappresentate da una linea che collega due classi, ad esempio se ogni istanza della classe A è associata ad una sola istanza della classe B e viceversa allora verrà rappresentata nel seguente modo:

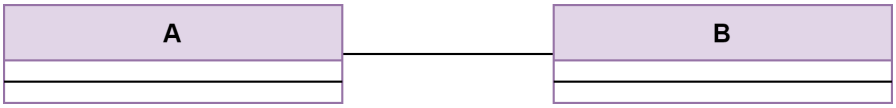


Figura 3.2: Esempio diagramma di una relazione uno a uno

3.2.3 Relazioni uno a molti

Le relazioni uno a molti sono rappresentate da una linea con una freccia che collega le due classi, ad esempio se ogni istanza di A è associata a più istanze di B allora verrà rappresentata nel seguente modo:

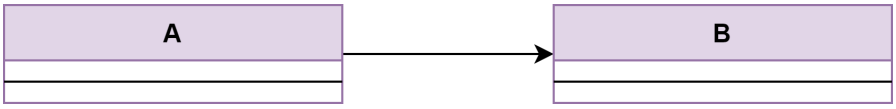


Figura 3.3: Esempio diagramma di una relazione uno a molti

3.2.4 Relazioni molti a molti

Le relazioni molti a molti sono rappresentate da una linea con due frecce che collega le due classi, ad esempio se ogni istanza di A è associata a più istanze di B e viceversa allora verrà rappresentata nel seguente modo:

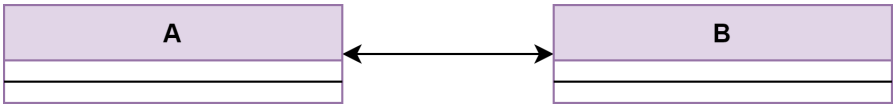


Figura 3.4: Esempio diagramma di una relazione molti a molti

3.2.5 Nomenclatura

La Nomenclatura utilizzata è la seguente:

* i nomi delle classi sono scritti in CamelCase;
* i nomi degli attributi sono scritti in snake_case;
* i nomi dei metodi sono scritti in snake_case.

3.3 Sprint 1

Nel primo sprint è stata definita l'architettura per modellare un ordine anche se in modo non del tutto completo. Il cliente di Moku S.r.l. offre i suoi servizi a decine di clienti, ciascun cliente viene identificato come progetto e ad ogni progetto possono corrispondere uno o più brand. Un canale invece rappresenta quello che è un canale di vendita come ad esempio Prestashop o Woocommerce, dato che ad ogni canale possono corrispondere uno o più brand e ad ogni brand possono corrispondere uno o più canali, si è scelto di creare una classe ChannelInstance per rappresentare tale relazione. Ogni ordine corrisponde ad un file XML inviato su un canale di vendita e contiene le informazioni relative a fatturazione e spedizione e possiede diverse righe d'ordine, ciascun ordine può contenere prodotti di uno o più brand.

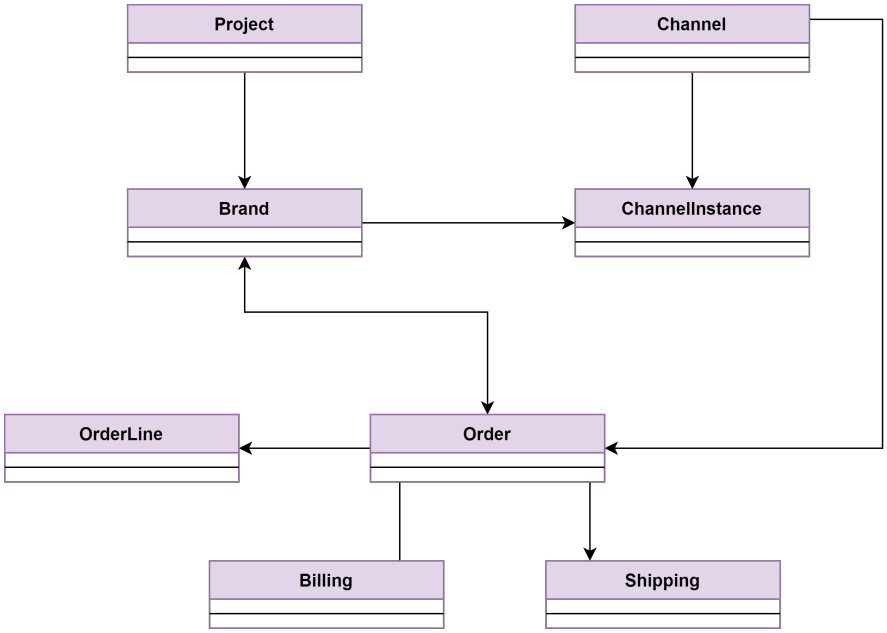


Figura 3.5: Diagramma delle classi sprint 1

3.3.1 Project

La classe Project rappresenta un progetto, ovvero un cliente. È caratterizzata da un nome, un codice e dalla data in cui è stato aperto il progetto. Ogni progetto deve soddisfare i seguenti vincoli:

* *name* deve essere presente;
* *code* deve essere presente e univoco.

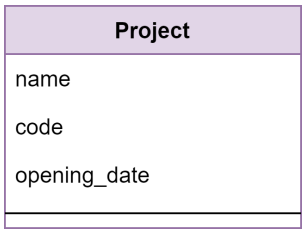


Figura 3.6: Diagramma della classe Project

3.3.2 Brand

La classe Brand rappresenta un marchio associato ad un progetto ed è caratterizzata da un nome e da un codice. Le istanze di Brand devono rispettare i seguenti vincoli:

* *name* deve essere presente;
* *code* deve essere presente e univoco.

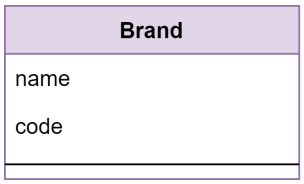


Figura 3.7: Diagramma della classe Brand

3.3.3 Channel

La classe Channel rappresenta un canale di vendita con relativo nome e url. La classe presenta inoltre una piattaforma come ad esempio Woocommerce, Prestashop o Amazon e una data che serve a stabilire quali ordini processare. Channel dispone di un metodo per ottenere il nome completo e presenta il seguente vincolo:

* *name* deve essere presente e univoco.

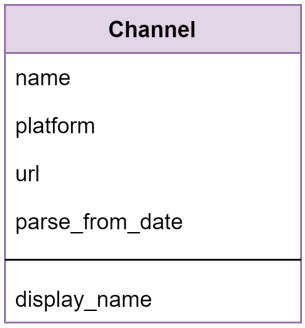


Figura 3.8: Diagramma della classe Channel

3.3.4 Order

La classe Order costituisce il nucleo dell'intero progetto e possiede diversi attributi, la maggior parte dei quali corrispondenti ai dati presenti nei file XML che arrivano dai canali di vendita come il metodo di pagamento, le note o i vari importi. Particolare attenzione va posta sull'attributo raw_data che contiene il file XML originale dal quale è stato generato l'ordine, questo attributo potrà risultare utile in futuro per una gestione diversa dei file XML o per risolvere particolari errori. È presente, inoltre, un attributo metadata che permetterà di salvare qualsiasi coppia chiave-valore non prevista. La classe Order non presenta particolari vincoli dato che si è deciso di accettare qualsiasi dato e segnalare al backoffice eventuali anomalie, l'unico vincolo presente è il seguente:

* per ogni canale di vendita, non ci possono essere due ordini con lo stesso *external_code*.

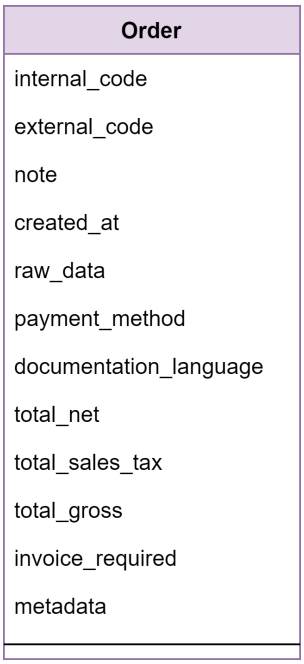


Figura 3.9: Diagramma della classe Order

3.3.5 Billing

La classe Billing rappresenta i dati di fatturazione di un ordine, gli attributi della classe corrispondenti ai dati presenti nel file XML dell'ordine corrispondente. Per le stesse ragioni spiegate in precedenza, la classe Billing non presenta particolari vincoli.

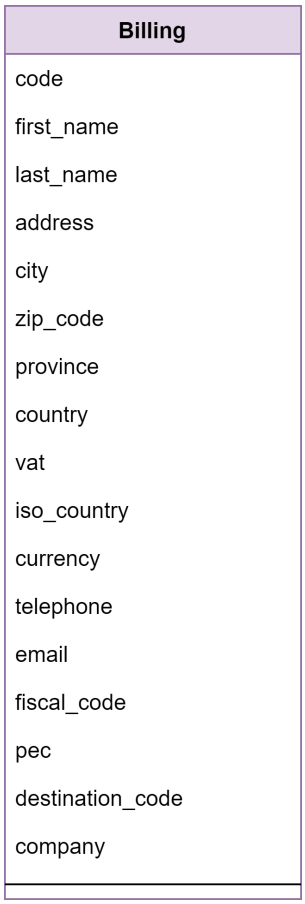


Figura 3.10: Diagramma della classe Billing

3.3.6 Shipping

La classe Shipping è molto simile alla classe Billing, rappresentando però i dati di spedizione di un ordine. Shipping presenta inoltre un metodo *shipping_status_history* per ottenere una lista con gli stati di spedizione precedenti dell'ordine.

Figura 3.11: Diagramma della classe Shipping

3.3.7 OrderLine

La classe OrderLine rappresenta una riga d'ordine che può corrispondere ad un prodotto acquistato, ad un coupon o ad una spedizione a seconda del valore dell'attributo kind. Di rilevante importanza è anche l'attributo sku che identifica un prodotto all'interno di un progetto. Come per le altre classi relative a un ordine, la classe OrderLine non presenta particolari vincoli.

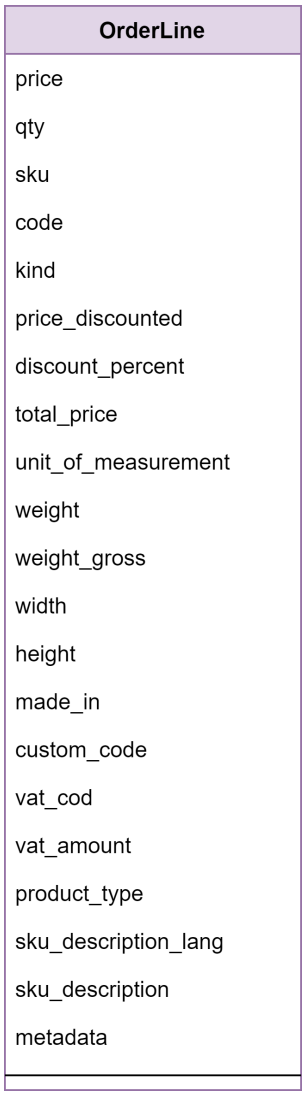


Figura 3.12: Diagramma della classe OrderLine

3.4 Sprint 2

Nel secondo sprint del progetto, si è passato a modellare le classi che permettono di realizzare i vari flussi sui server FTP. Ad ogni canale di vendita sono associati più server FTP, sui quali a seconda del tipo verranno caricati ordini o file prodotti da Agilis e Gsped. All'ordine che è già stato progettato e implementato nello sprint precedente, vengono associati dei documenti allegati che rappresentano un qualsiasi tipo di documento che ha a che fare con l'ordine. È stato inoltre necessario creare una classe per i prodotti che appartengono in modo diretto ad un brand ed in modo indiretto ad un progetto.

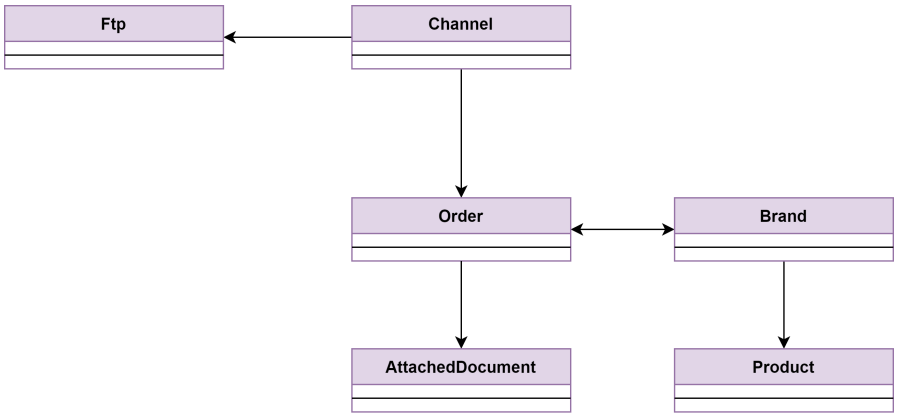


Figura 3.13: Diagramma delle classi sprint 2

3.4.1 Ftp

La classe Ftp rappresenta un server FTP con relativo url, porta, username e password. È inoltre presente un attributo path che indica la cartella di partenza del server FTP. Esistono quattro tipi di server FTP:

* i server su cui i siti di vendita caricano gli ordini per Agilis che devono possedere le cartelle *deliveries*, *documentation*, *availabilities*;
* i server su cui i siti di vendita caricano gli ordini per Gsped che devono possedere le cartelle *deliveries*, *documentation*, *statuses*;
* i server su cui Agilis carica i file prodotti che devono possedere le cartelle *deliveries*, *documentation*, *availabilities*;
* i server su cui Gsped carica i file prodotti che devono possedere le cartelle *deliveries*, *documentation*, *statuses*.

Le istanze di questa classe mettono a disposizione i seguenti metodi:

* *connect* che restituisce una connessione al server FTP gestendo eventuali errori;
* *connect!* che restituisce una connessione al server FTP senza gestire eventuali errori;
* *process* che processa il server FTP, ad esempio se sul server vengono caricati ordini, gli ordini verranno letti, salvati, inoltrati e cancellati dal server FTP.

Ftp presenta i seguenti vincoli:

* *url*, *kind*, *username* e *password* devono essere presenti;
* *kind* deve assumere uno dei seguenti valori: *agilis_shop*, *gsped_shop*, *gsped*, *agilis*.

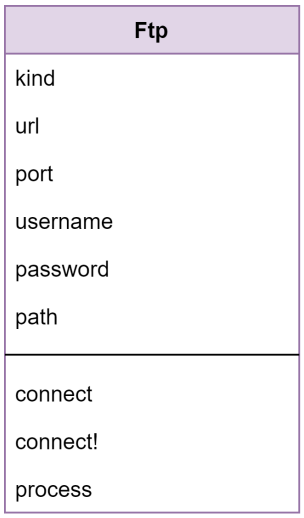


Figura 3.14: Diagramma della classe Ftp

3.4.2 AttachedDocument

La classe AttachedDocument rappresenta un documento allegato ad un ordine, il documento viene salvato via cloud utilizzando Amazon $S3_G$. Ogni documento è caratterizzato da un nome e un tipo relativo allo scopo del documento. AttachedDocument mette a disposizione i seguenti metodi:

* *document_ url* che ritorna un url al documento salvato in remoto su S3;
* *mime_ type* che restituisce il tipo MIME del documento;
* *attach_ document(file, file_ name)* che effettua l'associazione tra un'istanza di AttachedDocument e un file.

Le istanze di questa classe devono soddisfare i seguenti vincoli:

* il documento non deve superare i 100MB ed il tipo MIME può essere solo: *text/xml*, *application/xml* o *application/pdf*;
* l'attributo *kind* può assumere solo i seguenti valori: *xml_ original*, *xml_ modified*, *billing_ document* o *shipping_ document*.

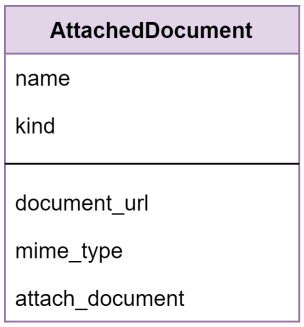


Figura 3.15: Diagramma della classe AttachedDocument

3.4.3 Product

La classe Product rappresenta un prodotto associato ad un Brand tenendone traccia della disponibilità e di altre informazioni rilevanti. Un prodotto è indirettamente associato ad un progetto all'interno del quale è identificato tramite un codice sku. Le istanze di Product devono soddisfare i seguenti vincoli:

* *sku* deve essere presente;
* *sku* deve essere univoco all'interno del brand;
* *sku* deve essere univoco all'interno del progetto.

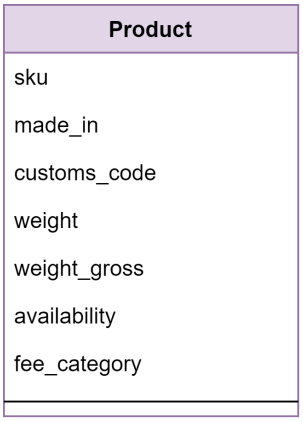


Figura 3.16: Diagramma della classe Product

3.5 Sprint 3

Il calcolo delle tasse imposte ai clienti risulta essere particolarmente complesso e con numerose casistiche, le quali hanno richiesto una soluzione in grado di astrarre i concetti il più possibile. Le tasse a seconda del cliente, sono calcolate mensilmente sul fatturato senza iva oppure sul totale incassato, possono essere calcolate a livello di progetto, brand o canale e seguono una delle sette regole descritte nella tabella della figura 3.17, da notare che nella tabella i casi 3 e 4 sono differenti dato che il 3 calcola la tassa per ogni scaglione andando a sottrarre ogni volta, mentre il caso 4 calcola tutto l'importo nello scaglione corrispondente. A tutto questo si aggiunge la possibilità di avere tasse separate a seconda della modalità di incasso come Adyen o Paypal. Per modellare nel modo più astratto possibile il calcolo delle tasse è stata creata una classe Fee associata ad un progetto, brand o canale che rappresenta la tassa di un singolo mese, ad essa sono poi associate le classi FeeCategory e FeePaymentGateway che rappresentano rispettivamente i ricavi per ciascuna categoria di prodotto e i ricavi per ciascuna modalità di pagamento. Al fine di poter calcolare la tassa, ad un progetto, brand o canale può essere associata una FeeConfiguration la quale assieme alle classi FeeConfigurationCategory, FeeRange e PaymentGateway permette di avere a disposizione tutte le informazioni necessarie al corretto calcolo delle tasse.

Numero	CASI	ESEMPIO
1	Percentuale fissa	18% con o senza minimo mensile (es. € 750)
2	A scaglioni con prima soglia ad importo fisso, e a percentuale le restanti	▪ fino a € 10.000 => € 1.200 ▪ da € 10.001 a 50.000 => 12% ▪ da € 50.001 a 100.000 => 10% ▪ oltre € 100.001 => 8%
3	A scaglioni con percentuale	▪ fino a € 10.000 => 14% ▪ da € 10.001 a 50.000 => 12% ▪ da € 50.001 a 100.000 => 10% ▪ oltre € 100.001 => 8% con o senza minimo mensile (es. € 750)
4	In base al singolo scaglione	▪ fino a € 10.000 => 14% ▪ da € 10.001 a 50.000 => 12% ▪ da € 50.001 a 100.000 => 10% ▪ oltre € 100.001 => 8% con o senza minimo mensile (es. € 750)
5	Importo fisso più percentuale	€ 500 mese + ▪ fino a € 10.000 => 14% ▪ da € 10.001 a 50.000 => 12% ▪ da € 50.001 a 100.000 => 10% ▪ oltre € 100.001 => 8%
6	Percentuale diversa in base alla tipologia di articoli venduti	▪ 13,5% prodotti categorie A/B/C ▪ 7% prodotti categorie D/E/F con o senza minimo mensile (es. € 750)
7	Percentuale diversa in base all'anno di apertura del progetto	1° anno => 10% 2° anno => 12% dal 3° anno in poi => 14% con o senza minimo mensile (es. € 750)

Figura 3.17: Tabella riassuntiva per il calcolo delle tasse

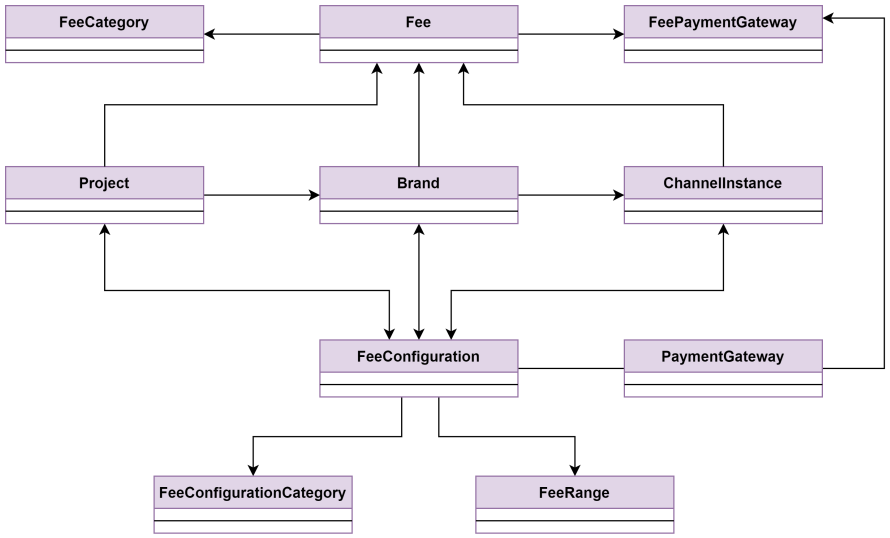


Figura 3.18: Diagramma delle classi sprint 3

3.5.1 Fee

La classe Fee rappresenta la tassa di un determinato mese imposta ad un cliente, è caratterizzata dal fatturato e l'iva su cui si andrà a calcolare l'importo tramite il metodo *calculate_fee*. Oltre a questo, la classe mette a disposizione il metodo *fee_configuration* che ritorna tutte le configurazioni associate alla tassa e il metodo *monthly_minimum* che ritorna il minimo mensile per la tassa. Ogni istanza di Fee deve rispettare i seguenti vincoli:

* mese e anno della tassa devono essere presenti e validi;
* per ogni progetto ci può essere una sola Fee per ogni mese;
* per ogni brand ci può essere una sola Fee per ogni mese;
* per ogni canale ci può essere una sola Fee per ogni mese;
* ogni Fee si riferisce ad uno e uno solo tra progetto, brand e canale.

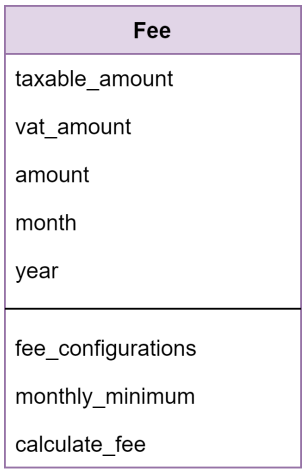


Figura 3.19: Diagramma della classe Fee

3.5.2 FeeCategory

La classe FeeCategory rappresenta il fatturato e l'iva di una singola categoria di prodotto per una Fee. La somma di tutti i *taxable_ amount* e *vat_ amount* delle FeeCategory di una Fee corrisponderà ai valori presenti nell'istanza Fee associata. FeeCategory presenta i seguenti vincoli:

* il nome della categoria deve essere presente;
* una Fee non può avere più di una FeeCategory con lo stesso nome.

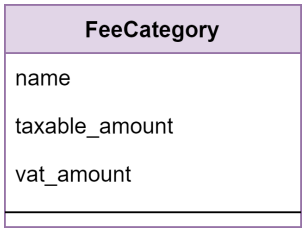


Figura 3.20: Diagramma della classe FeeCategory

3.5.3 FeePaymentGateway

Analogamente a FeeCategory, la classe FeePaymentGateway rappresenta il fatturato e l'iva di una singola modalità di pagamento per una Fee, con la sola differenza che FeePaymentGateway non presenta un nome perché è già associato alla classe PaymentGateway. Le istanze di FeePaymentGateway presentano i seguenti vincoli:

* il gateway di pagamento associato deve essere presente;
* una Fee non può avere più di una FeePaymentGateway con lo stesso gateway di pagamento.

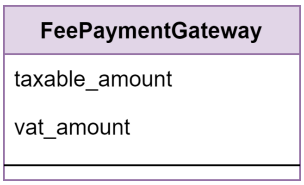


Figura 3.21: Diagramma della classe FeePaymentGateway

3.5.4 FeeConfiguration

La classe FeeConfiguration ha lo scopo di generalizzare tutti i concetti espressi nella tabella in figura 3.17 cercando di astrarli il più possibile al fine di evitare di avere una classe per ogni configurazione possibile ed essere mantenibile in caso di nuove configurazioni. In particolare, ogni FeeConfiguration può avere un minimo mensile, un importo fisso da aggiungere ed un booleano che indica se la tassa va calcolata sul fatturato considerando o meno l'Iva. Per calcolare l'importo della Fee nei casi 1, 2, 3, 4, 5 e 7 si utilizza la classe associata FeeRange e l'attributo *range_kind* indica il tipo degli scaglioni. Per calcolare l'importo della Fee nel caso 6 invece si utilizza la classe associata FeeConfigurationCategory. Infine, ogni FeeConfiguration è associata ad uno o più gateway di pagamento per permettere il calcolo nel caso in cui si vogliano avere Fee diverse per i gateway di pagamento. Le istanze di FeeConfiguration presentano i seguenti vincoli:

* *range_kind* deve assumere uno dei seguenti valori: *tiered_pricing*, *volume_pricing* o *year*;
* se l'attributo *range_kind* è presente, allora la configurazione deve avere almeno un FeeRange associato.

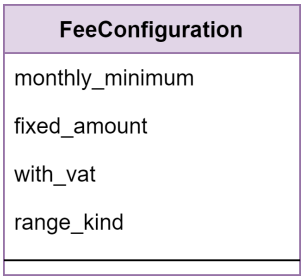


Figura 3.22: Diagramma della classe FeeConfiguration

3.5.5 FeeConfigurationCategory

La classe FeeConfigurationCategory rappresenta la percentuale di tassa da applicare a una categoria di prodotto per una configurazione di Fee. Le istanze di FeeConfigurationCategory presentano i seguenti vincoli:

* il nome deve essere presente;
* una FeeConfiguration non può avere più di una FeeConfigurationCategory con lo stesso nome.

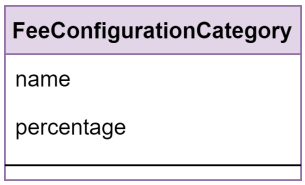


Figura 3.23: Diagramma della classe FeeConfigurationCategory

3.5.6 FeeRange

La classe FeeRange rappresenta un singolo scaglione appartenente ad una configurazione di Fee. La classe generalizza sia il concetto di scaglione in base al fatturato che quello di calcolo della tassa in base all'anno di creazione del progetto. Dato che una FeeConfiguration può avere più di un FeeRange, per rappresentare lo scaglione basterà avere un attributo che indica il limite inferiore, mentre il limite superiore verrà stabilito in base allo scaglione successivo. Considerati i diversi modi di calcolare le tasse, ad ogni scaglione può avere una percentuale o un importo fisso. Le istanze di FeeRange presentano i seguenti vincoli:

* deve essere presente uno ed un solo attributo tra *percentage* e *fixed_ amount*;
* la percentuale se presente deve essere compresa tra 0 e 100;
* l'importo fisso se presente deve essere maggiore di 0;
* il limite inferiore deve essere presente;
* una FeeConfiguration non può avere più di un FeeRange con lo stesso limite inferiore;
* se una FeeConfiguration ha almeno un FeeRange allora deve averne uno con limite inferiore uguale a 0.

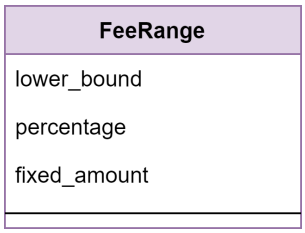


Figura 3.24: Diagramma della classe FeeRange

3.5.7 PaymentGateway

La classe PaymentGateway rappresenta un gateway di pagamento come, ad esempio, Paypal o Stripe ed è caratterizzata da un codice tramite il quale il gateway di pagamento viene identificato a back-end ed un nome che viene utilizzato per la visualizzazione dei gateway di pagamento a front-end. Le istanze di PaymentGateway presentano i seguenti vincoli:

* il codice deve essere presente;

* il codice deve essere univoco.

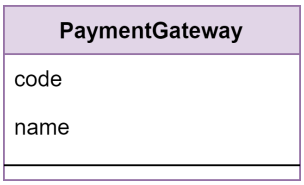


Figura 3.25: Diagramma della classe PaymentGateway

3.6 Sprint 4

Durante il quarto sprint ci si è concentrati sui documenti di vendita, essi vengono inviati da Agilis in formato XML e possono essere associati a un ordine, una spedizione o una fatturazione. Ogni documento di vendita presenta diverse linee di dettaglio contenenti informazioni più specifiche sui prodotti e sulle relative quantità, inoltre ciascun documento di vendita è associato ad un gateway di pagamento. Le informazioni contenute all'interno dei vari documenti di vendita sono fondamentali per poter generare le istanze di Fee e permetterne il calcolo della tassa.

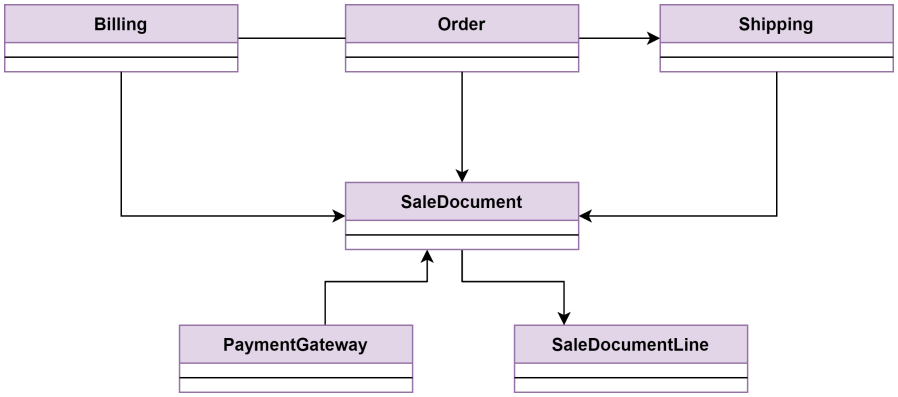


Figura 3.26: Diagramma delle classi sprint 4

3.6.1 SaleDocument

La classe SaleDocument rappresenta un documento di vendita ricevuto da Agilis ed i suoi attributi rispecchiano quelli presenti nel file XML. Di particolare importanza sono gli attributi *taxable_ amount* e *vat_ amount* perché utili alla generazione delle istanze di Fee. SaleDocument presenta il seguente vincolo:

* *kind* deve assumere uno dei seguenti valori: *invoice*, *credit_ note*, *receipt*, *receipt_ note* o *uk_ document*.

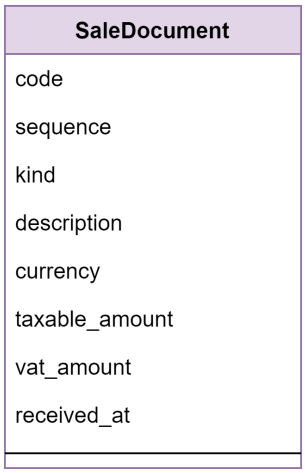


Figura 3.27: Diagramma della classe SaleDocument

3.6.2 SaleDocumentLine

La classe SaleDocumentLine rappresenta una linea di dettaglio di un documento di vendita ricevuto da Agilis ed anche i suoi attributi rispecchiano quelli presenti nel file XML. Essenziali per la generazione delle istanze di Fee sono gli attributi *total_price*, *qty*, *vat_amount*, *product_sku* e *product_fee_category*. Le istanze di SaleDocumentLine presentano il seguente vincolo:

- *kind* deve assumere uno dei seguenti valori: *item*, *coupon*, *shipping_charges* o *payment_charges*.

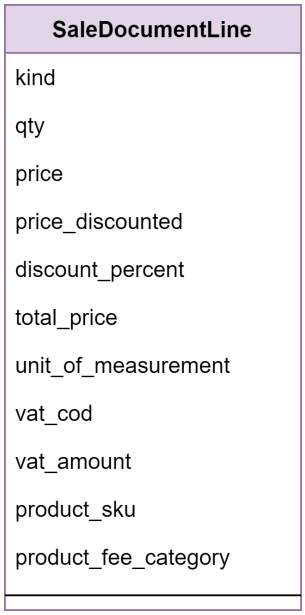


Figura 3.28: Diagramma della classe SaleDocumentLine

3.7 Diagramma ER complessivo

Il seguente diagramma ER rappresenta l'intera architettura del sistema ottenuta al termine del mio stage.

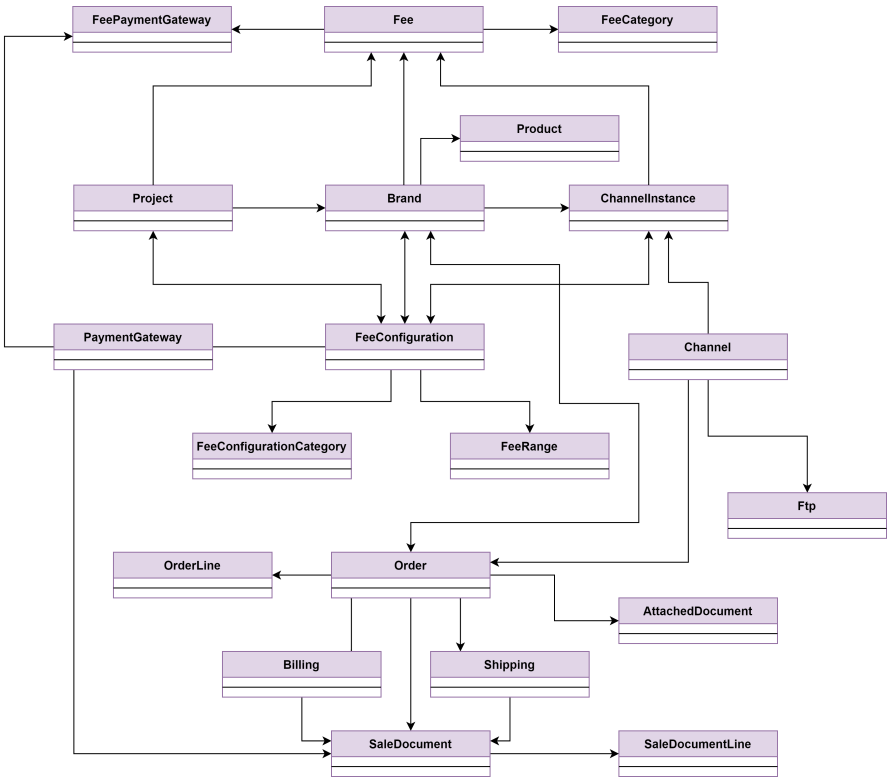


Figura 3.29: Diagramma ER completo

Capitolo 4

Realizzazione e testing

4.1 Codifica

La codifica segue quella che è la filosofia di Rails, ad ogni classe viene associata una tabella nel database il quale viene costruito in modo incrementale tramite delle migrazioni. In questo modo è sempre possibile ottenere lo schema aggiornato del database anche collaborando con altri sviluppatori.

4.1.1 Generazione dei modelli

Un modello di Rails non è altro che una classe di Ruby in grado di interagire con il database tramite l'estensione della classe ActiveRecord. Per illustrare il procedimento di generazione, verrà preso come esempio il modello Project. Come prima cosa è necessario utilizzare il comando `rails generate model project`, il quale genera la classe Project nel file *app/models/project.rb* e un file di migrazione nella cartella *db/migrate/<timestamp>_create_projects.rb* il quale servirà per creare la tabella *projects* nel database. Inizialmente il contenuto del file di migrazione risulta essere il seguente:

```
class CreateProjects < ActiveRecord::Migration[7.0]
  def change
    create_table :projects do |t|

      t.timestamps
    end
  end
end
```

Listato 4.1: Prima migrazione del modello Project

Per aggiungere gli attributi del modello basta modificare il file di migrazione nel seguente modo:

```
class CreateProjects < ActiveRecord::Migration[7.0]
  def change
    create_table :projects do |t|
      t.string :name
      t.string :code
```

```
      t.timestamps
    end
  end
end
```

Listato 4.2: Prima migrazione del modello Project aggiornata

A questo punto basta lanciare il comando `rails db:migrate` per creare la tabella *projects* nel database e aggiungere gli attributi alla classe Project. Una volta che il file contenente la migrazione è stato caricato sul branch di sviluppo e quindi potenzialmente utilizzato anche da altri sviluppatori, è fortemente consigliato non modificarlo più al fine di non creare incongruenze tra i database degli sviluppatori. In caso fosse necessario aggiungere, modificare o rimuovere attributi, è possibile farlo creando una nuova migrazione tramite il comando `rails generate migration nome_della_migration`. Durante il mio stage, ad esempio, è stato necessario aggiungere l'attributo *opening_ date* al modello Project. Per farlo è bastato utilizzare il comando `rails generate migration add_opening_date_to_projects` e modificare il file di migrazione nel seguente modo, lanciando in seguito il comando `rails db:migrate`

```
class AddOpeningDateToProject < ActiveRecord::Migration[7.0]
  def change
    add_column :projects, :opening_date, :date
  end
end
```

Listato 4.3: Seconda migrazione del modello Project

Con la generazione dei modelli effettuata nel modo indicato, Rails crea automaticamente anche gli attributi *ID, created_ at e updated_ at* oltre a dei metodi per effettuare le operazioni di creazione, lettura, modifica e cancellazione. Al termine di queste operazioni, il contenuto del file `app/models/project.rb` risulta essere il seguente:

```
class Project < ApplicationRecord

end
```

Listato 4.4: Modello Project

4.1.2 Associazioni

Rails permette di creare diversi tipi di associazione tra ActiveRecord le quali permettono di semplificare alcune operazioni comuni tra classi associate. Prima di essere creata a livello di ActiveRecord, un'associazione deve essere presente a livello di database, per fare questo è necessario specificarlo all'interno di una migrazione. Per esempio, ad ogni istanza della classe Brand, è associata un'istanza della classe Project, il file di migrazione per la creazione di Brand ha quindi il seguente contenuto:

```
class CreateBrands < ActiveRecord::Migration[7.0]
  def change
    create_table :brands do |t|
      t.string :name
      t.string :code
      t.references :project, null: false, foreign_key: true
```

```
      t.timestamps
    end
  end
end
```

Listato 4.5: Prima migrazione del modello Brand

A livello di ActiveRecord esistono diversi tipi di associazione, verranno descritte solo quelle utilizzate all'interno del progetto. Prendendo come esempio la classe Order, è possibile trovare quasi tutti i tipi di associazione utilizzati nel progetto.

```
class Order < ApplicationRecord
  has_many :order_lines, dependent: :destroy, after_add: :
      denormalize
  has_one :billing, dependent: :destroy
  has_many :shippings, dependent: :destroy
  has_many :attached_documents, dependent: :destroy
  has_many :sale_documents, dependent: :restrict_with_exception
  belongs_to :source_channel, class_name: 'Channel'
  has_and_belongs_to_many :denormalized_brands, class_name: '
      Brand'

  # more code here ...

end
```

Listato 4.6: Associazioni della classe Order

- *belongs_to* crea una connessione con un altro modello in modo tale che ogni istanza del modello che dichiara l'associazione appartiene ad una istanza del modello dichiarato come parametro. Nell'esempio ogni istanza di Order appartiene ad un'istanza di Channel;
- *has_one* indica che un altro modello ha un riferimento a questo modello a livello di database. Nell'esempio, dunque, ogni istanza di Billing ha un riferimento a Order a livello di database e un'associazione *belongs_to* a livello di ActiveRecord. Inoltre, due istanze diverse di Billing non potranno avere un riferimento allo stesso Order;
- *has_many* è un tipo di associazione molto simile ad *has_one*, ma in questo caso viene rappresentata un'associazione di tipo uno a molti, dall'altro lato della relazione è quindi possibile avere un'associazione di tipo *belongs_to*. Nel codice di esempio è possibile notare come ad ogni ordine siano associate diverse istanze di OrderLine;
- *has_and_belongs_to_many* rappresenta una relazione di tipo molti a molti. Per creare questo tipo di associazione è necessario che sia presente una tabella all'interno del database contenente gli indici delle due tabelle coinvolte.

Come è possibile notare dal codice nel listato 4.6, Rails permette di stabilire cosa accade alle classi associate in caso di cancellazione dell'istanza di una classe inserendo un ulteriore parametro alla dichiarazione dell'associazione:

- `dependent: :destroy` tutte le istanze associate alla classe vengono distrutte;

* `dependent: :restrict_with_exception` viene sollevata un'eccezione se sono presenti istanze associate alla classe al momento della distruzione;
* `dependent: :nullify` la chiave esterna delle istanze associate alla classe assume il valore *nil*;
* `dependent: :delete_all` le istanze associate alla classe vengono eliminate direttamente dal database.

4.1.3 Validazioni

Per implementare tutti i vincoli individuati in fase di progettazione, sono state utilizzate le validazioni messe a disposizione da Rails. Le validazioni sono usate per assicurare che vengano salvati nel database solo dati corretti, esse vengono eseguite a livello di modello prima che viene creata o modificata l'istanza di una classe.

```
class FeeRange < ApplicationRecord
  belongs_to :fee_configuration
  validates :lower_bound, uniqueness: {scope: :
      fee_configuration_id}, presence: true
  validates :percentage, numericality: {greater_than_or_equal_to:
       0.0, less_than_or_equal_to: 100.0}, allow_blank: true
  validates :fixed_amount, numericality: {
      greater_than_or_equal_to: 0.0}, allow_blank: true

  validate :percentage_xor_fixed_amount

  def percentage_xor_fixed_amount
    return if percentage.present? ^ fixed_amount.present?

    errors.add :base, 'Only one of percentage or fixed amount
        must be present'
  end
end
```

Listato 4.7: Validazioni del modello FeeRange

Come è possibile notare dal listato 4.7, Rails mette a disposizione due modi diversi per effettuare le validazioni. Il primo avviene tramite la funzione `validates` la quale permette di effettuare le validazioni più comuni in modo semplice, nell'esempio viene utilizzata per assicurarsi che la percentuale sia compresa tra 0 e 100, l'importo fisso sia maggiore di 0 e che ogni istanza di FeeConfiguration non abbia due istanze di FeeRange associate con lo stesso limite inferiore. Nel caso sia necessario effettuare una validazione più complessa, è possibile utilizzare la funzione `validate` che permette di effettuare validazioni personalizzate, nell'esempio è stata utilizzata per assicurarsi che uno e uno solo tra la percentuale e l'importo fisso sia presente.

4.1.4 Servizi

Durante la fase di codifica, si è cercato di lasciare il corpo dei modelli il più semplice possibile. Per fare questo, tutte le logiche di business più complesse sono state implementate all'interno della cartella *app/services* e poste all'interno di moduli. In particolare, i servizi implementano le seguenti logiche di business:

* il processamento dei server FTP;
* il parsing dei file relativi agli ordini;
* la gestione delle disponibilità dei prodotti;
* la generazione ed il calcolo delle tasse.

```
module XmlLegacy
  module Availabilities
    module XmlGenerator
      def self.generate_xml products
        require 'builder'
        xml = Builder::XmlMarkup.new indent: 2
        xml.instruct! :xml, version: '1.0', encoding: 'UTF-8'
        xml.availabilities do |availabilities|
          products.each do |product|
            availabilities.availability do |availability|
              availability.sku product.sku
              availability.availability product.availability
            end
          end
        end
      end
    end
  end
end
```

Listato 4.8: Esempio di servizio

Nel listato 4.8, viene riportato l'esempio di un servizio che permette di generare un file XML contenente le disponibilità di una lista di prodotti.

4.1.5 Gestione asincrona

Per evitare che operazioni come la scrittura di un file su un server FTP rendano l'esecuzione del programma troppo lenta, Rails consente di gestire queste operazioni in modo asincrono. Per fare questo, le funzioni gestite in modo asincrono invece di essere eseguite immediatamente quando chiamate, vengono inviate ad una coda detta *delayed_jobs_queue* la quale viene processata parallelamente al resto del programma.

```
  module XmlLegacy
  module Ftp
    module Writer

      # more code here ...

      class << self
        def push_all_availabilities channels
          channels.each do |channel|
            products = Product.where brand_id: channel.brands.
                pluck(:id)
            next if products.none?
```

```
            xml = XmlLegacy::Availabilities::XmlGenerator.
                generate_xml products
            channel.ftps.where(kind: 'agilis_shop').each do |ftp|
              connection = ftp.connect!
              connection.chdir 'availabilities'
              temp_file = Tempfile.new 'availabilities.xml'
              temp_file.write xml
              temp_file.close
              connection.putbinaryfile temp_file, 'availabilities
                  .xml'
              connection.close
            end
          end
        end
        handle_asynchronously :push_all_availabilities
      end

      # more code here ...

    end
  end
end
```

Listato 4.9: Esempio di gestione asincrona

4.1.6 Scheduler

Il calcolo delle tasse richiede che una volta al mese vengano generate tutte le istanze di Fee a partire dai documenti di vendita presenti nel sistema. Per la gestione di compiti come questo è stata utilizzata la libreria *whenever* la quale permette di programmare l'esecuzione di un compito in modo semplice. Nel listato che segue viene mostrato come il calcolo delle tasse viene programmato per essere eseguito il primo giorno di ogni mese alle ore 6:00.

```
require File.expand_path(File.dirname(__FILE__) + '/environment')

every 1.month, at: '6:00 am' do
  runner 'FeeCalculation::Generator.generate_fees'
end
```

Listato 4.10: Esempio di scheduler

4.1.7 Graphql

Graphql è stato utilizzato per fornire delle API in grado di visualizzare e modificare i dati di determinati modelli. Seguendo il principio di Rails *Convention over Configuration*, la configurazione dei tipi di Graphql risulta essere molto semplice in quanto il framework è in grado di associare automaticamente i tipi di Graphql ai modelli se presentano lo stesso nome.

```
module Types
  class ProjectType < Types::Base::BaseObject
    field :id, ID, null: false
```

```
    field :name, String, null: true
    field :code, String, null: true
    field :brands, [BrandType], null: false
  end
end
```

Listato 4.11: Dichiarazione ProjectType Graphql

```
module Types
  class QueryType < Types::Base::BaseObject

    # more code here ...

    field :project_by_id, ProjectType, null: true do
      description 'Gets a project by id'
      argument :project_id, ID
    end

    def project_by_id project_id:
      Project.find project_id
    end

    # more code here ...

  end
end
```

Listato 4.12: Query project_by_id

Figura 4.1: Schermata della query GraphQL project_by_id

4.1.8 Active Admin

Active Admin è una libreria che ha permesso di realizzare in modo semplice un'interfaccia web per la gestione dei dati presenti nel sistema da parte di un amministratore. Anche in questo caso la semplicità nella realizzazione dell'interfaccia è data dall'utilizzo di determinate convenzioni che permettono di associare gli elementi dell'interfaccia ai modelli.

```
  ActiveAdmin.register Project do
```

```
  menu priority: 1
  permit_params :name, :code, user_ids: []

  index do
    selectable_column
    id_column
    column :name
    column :code
    actions
  end

  show do
    attributes_table :id, :name, :code do
      panel t('activerecord.models.brand.other') do
        table_for project.brands do
          column :name
          column :code
        end
      end
      panel t('activerecord.models.user.other') do
        table_for project.users do
          column :name do |user|
            link_to user.name, admin_user_path(user)
          end
          column :email
        end
      end
    end
  end

  form do |f|
    f.input :name
    f.input :code
    f.input :users, as: :searchable_select, multiple: true
    f.actions
  end
end
```

Listato 4.13: Project Active Admin

Si può notare come il solo codice illustrato nel listato 4.13 permetta di ottenere le interfacce mostrate nelle figure 4.2 e 4.3 le quali permettono di gestire i dati dei vari progetti.

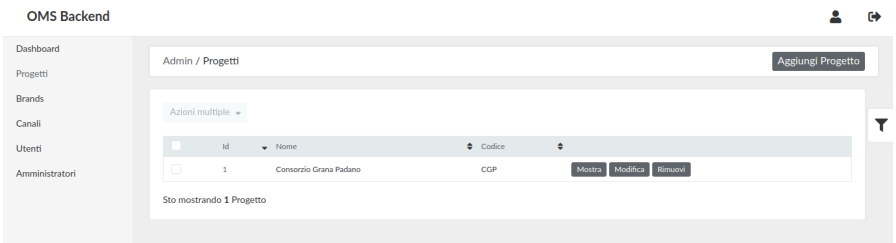


Figura 4.2: Schermata Active Admin per la gestione dei progetti

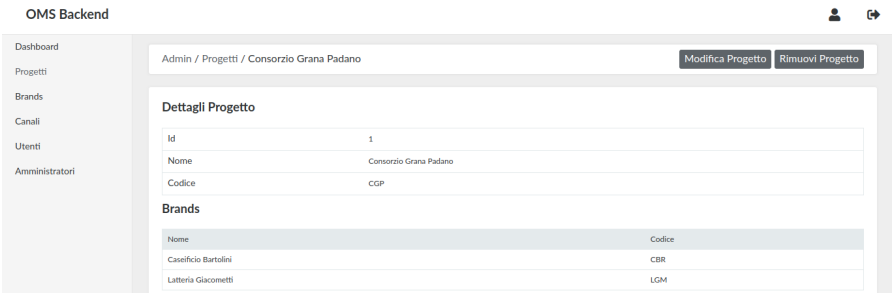


Figura 4.3: Schermata Active Admin per la visualizzazione di un singolo progetto

Oltre all'interfaccia per la gestione dei progetti, sono state realizzate anche le seguenti interfacce:

* interfaccia per la gestione dei canali;
* interfaccia per la gestione dei brand;
* interfaccia per l'importazione e per l'esportazione dei file CSV relativi ai prodotti di un brand.

Active Admin rimane comunque una libreria per la gestione dei dati lato back-end, tutte le interfacce per la gestione degli ordini da parte del backoffice vengono realizzate dagli sviluppatori front-end del team.

4.2 Test

Durante il corso dello stage sono stati scritti dei test automatici col fine di verificare la correttezza dei modelli relativi al calcolo delle tasse, tuttavia, non c'è stato abbastanza tempo per implementare dei test automatici anche per gli altri modelli. Le librerie che hanno permesso la realizzazione dei test automatici sono *Rspec* e *FactoryBot*.

4.2.1 FactoryBot

La libreria *FactoryBot* permette di automatizzare la creazione di istanze dei modelli per l'esecuzione dei test.

```
FactoryBot.define do
  factory :fee do
    taxable_amount { 0.0 }
    vat_amount { 0.0 }
    month { 1 }
    year { 2020 }
    trait :brand_fee do
      brand
      channel_instance { nil }
      project { nil }
    end
    trait :channel_instance_fee do
      channel_instance
      brand { nil }
      project { nil }
```

```
    end
    trait :project_fee do
      project
      brand { nil }
      channel_instance { nil }
    end
  end
end
```

Listato 4.14: Esempio di Factory

Il listato 4.14 illustra il codice per l'automatizzazione della creazione delle istanze di Fee, tramite le definizioni dei `trait`, *FactoryBot* permette di definire delle caratteristiche da dare alle istanze di Fee che vengono create. In questo esempio vengono definiti dei valori di default per gli attributi delle istanze di Fee che tuttavia possono essere modificati.

4.2.2 Rspec

Rspec è invece una libreria che facilita la scrittura e l'esecuzione dei test, prima di definire i test è necessario impostare l'ambiente su cui essi verranno eseguiti, per fare questo vengono create le istanze dei modelli necessari tramite i metodi di creazione forniti da *FactoryBot*. Ogni gruppo di test deve essere preceduto da una descrizione che spiega cosa si sta andando a testare mentre ogni singolo test deve essere preceduto da una descrizione che indica l'esito atteso del test.

```
require 'rails_helper'

RSpec.describe Fee, type: :model do
  describe 'range with difference, fixed amount first range,
      monthly minimum, with vat' do
    before do
      @brand = create :brand
      @fee_configuration = create :fee_configuration,
                                  :tiered_pricing,
                                  :payment_gateways,
                                  monthly_minimum: 700,
                                  with_vat: true,
                                  brands: [@brand]

      # other configurations here ...

    end

    it 'should calculate the fee correctly' do
      @fee.taxable_amount = 10_000
      @fee.vat_amount = 0
      @fee.fee_payment_gateways.first.update taxable_amount: 10
          _000
      @fee.fee_payment_gateways.first.update vat_amount: 0
      @fee.calculate_fee
      expect(@fee.amount).to eq(700)
    end
```

```
    # other tests here ...
  end
  # other test groups here ...
end
```

Listato 4.15: Esempio di test

Capitolo 5

Conclusioni

5.1 Analisi del prodotto ottenuto

Il prodotto realizzato durante il mio stage, anche se non ancora pronto per il rilascio, verrà utilizzato dal cliente per molti anni a venire. Attualmente, l'OMS è in grado di porsi nel mezzo tra i siti di vendita e i software di gestione della logistica riuscendo a gestire gli ordini e permettendone la modifica. Oltre a questo, l'OMS permette di gestire le disponibilità dei prodotti, gli stati delle spedizioni ed il calcolo delle tasse per i clienti. Tutte queste funzionalità sono state testate con un dataset reale composto da oltre 5000 ordini, 900 file di disponibilità e 14000 stati di spedizione. Prima che il prodotto possa essere effettivamente utilizzato dal cliente, è necessario implementare le ultime funzionalità richieste e risolvere alcune problematiche riscontrate.

Criticità

Il prodotto presenta principalmente due criticità. La prima riguarda la libreria utilizzata per eseguire le operazioni su un server FTP da Rails, attualmente questa libreria non è in grado di caricare file che superano una dimensione di circa 100kB mentre viene utilizzata una connessione resa sicura tramite crittografia. Dato che a seguito di svariate prove non è stato possibile risolvere il problema, attualmente i test vengono effettuati con una connessione non sicura, ma dato che ciò non potrà accadere in un ambiente di produzione, in futuro si rimpiazzerà questa libreria con dei comandi di shell lanciati da codice. La seconda criticità riguarda la gestione delle disponibilità, dato che attualmente i siti di vendita ricevono il file contenente le disponibilità una volta al giorno e si aspettano che tale file sia nominato *availabilities.xml*, c'è il rischio che inviando un file ogni volta che si riceve un ordine, il file inviato vada a sovrascrivere quello precedente senza che il sito di vendita lo legga. In regime di concorrenza, inoltre, c'è il rischio che si verifichi il seguente scenario:

* il sito di vendita riceve un file con le disponibilità e lo legge senza cancellarlo subito;
* arriva un nuovo file che sovrascrive quello appena letto;
* il sito pensando che il file presente sul server sia quello appena letto, lo cancella.

In questo scenario, il sito non avrà le disponibilità dei prodotti aggiornate. Per risolvere questo problema, in produzione si potrà pensare di inviare i file ad intervalli di tempo regolari anziché ogni volta che si riceve un ordine.

Sviluppi futuri

A seguito del primo rilascio, verranno estese le funzionalità del prodotto al fine di permettere la comunicazione dell'OMS con i siti di vendita e i servizi di logistica tramite API considerate più all'avanguardia rispetto all'attuale FTP. In futuro, inoltre, si potrà pensare di generalizzare l'architettura attuale per permettere all'OMS di interagire con un qualsiasi software ERP anziché solamente con Agilis. Analogamente verrà valutata la possibilità di estendere il prodotto attuale al fine di poter interagire con qualsiasi software per la gestione delle spedizioni evitando così di essere vincolati all'utilizzo di Gsped.

5.2 Raggiungimento degli obiettivi

Il raggiungimento degli obiettivi fissati all'inizio dello stage è riassunto dal seguente elenco:

* Soddisfatti
  - gestione e pianificazione del progetto attraverso kanban board condivisa;
  - analisi e progettazione dei modelli back-end con diagramma ER, a partire dai requisiti raccolti;
  - analisi ed implementazione API GraphQL;
  - analisi ed integrazione con software esterni via XML;
  - coordinamento con il cliente finale.
* Parzialmente soddisfatti
  - suite di testing del software prodotto;
  - documentazione completa.

5.3 Valutazione personale

Valuto molto positivamente la mia esperienza di stage perché mi ha dato la possibilità di entrare a far parte di un team di sviluppo che lavora su un progetto reale. Durante questa esperienza ho imparato ad utilizzare tecnologie come Ruby on Rails e GraphQL che non avevo mai avuto modo di conoscere in ambito universitario, inoltre ho avuto la possibilità di comprendere a pieno la metodologia di lavoro Agile. Lo stage mi ha fatto crescere molto dal punto di vista professionale perché mi ha permesso di interagire in modo diretto con il cliente oltre ad avere avuto la possibilità di confrontarmi con altri colleghi più esperti di me molto frequentemente.

Acronimi e abbreviazioni

API Application Program Interface. 5, 6, 38, 46

FTP File Transfer Protocol. 3, 4, 10, 20, 21, 37, 45, 46

OMS Order Management System. 2, 4–6, 10, 45, 46

UML Unified Modeling Language. 13

XML eXtensible Markup Language. 3, 4, 7, 10, 15, 17, 29, 30, 37, 46

Glossario

Agile Insieme di metodi di sviluppo del software fondati su un insieme di principi comuni che si contrappongono ad altre metodologie di sviluppo proponendo un approccio meno strutturato caratterizzato dal coinvolgimento diretto e continuo del cliente nel processo di sviluppo. 1, 6, 9, 13, 46

Agilis Software gestionale che integra tutti i processi di business rilevanti di un'azienda tra cui la gestione della fatturazione e del magazzino. 3, 4, 7, 10, 11, 20, 21, 29, 30, 46

API In informatica con il termine *Application Programming Interface API* si indica un insieme di procedure atte a risolvere uno specifico problema di comunicazione tra diversi computer, tra diversi software o tra diversi componenti di software . 47

back-end In informatica, il termine *back-end* denota la parte di programma non visibile all'utente che interagisce con il front-end per il corretto funzionamento del programma. iii, 1, 2, 6, 13, 28, 41, 46

front-end In informatica, il termine *front-end* denota la parte visibile all'utente di un programma e con cui egli può interagire. 1, 28, 41

FTP *File Transfer Protocol FTP* è un protocollo di livello applicativo per la trasmissione di dati tra host basato su TCP e con architettura di tipo client-server, il protocollo è spesso reso sicuro utilizzando un sottostrato SSL/TLS prendendo il nome di FTPS. 47

Gsped Sistema *Software as a service* che permette di gestire le spedizioni di un'azienda integrandosi con i canali di vendita ed il magazzino. 3, 4, 11, 20, 21, 46

OMS *Order Management System OMS* è un software di gestione degli ordini prodotto da Moku S.r.l. 47

S3 Servizio web di memorizzazione offerto da Amazon Web Services, il servizio permette di memorizzare qualsiasi tipo di dato, permettendo utilizzi quali backup, disaster recovery o archiviazione. 22

UML In ingegneria del software *UML, Unified Modeling Language* è un linguaggio di modellazione e specifica basato sul paradigma object-oriented. 47

XML *eXtensible Markup Language XML* in informatica è un metalinguaggio per la definizione di linguaggi di markup, ovvero un linguaggio basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o in un testo. 47

Bibliografia

Siti web consultati

Documentazione di Active Admin. URL: https://activeadmin.info/documentation.html.

Documentazione di Ruby. URL: https://ruby-doc.org/.

Guida per Ruby on Rails. URL: https://guides.rubyonrails.org/.

How to GraphQL. URL: https://www.howtographql.com/.

Stack Overflow. URL: https://stackoverflow.com/.

Tutorial Ruby on Rails. URL: https://www.railstutorial.org/book.

Wikipedia. URL: https://it.wikipedia.org/.