Università degli Studi Roma Tre
Dipartimento di Ingegneria
Corso di Laurea Triennale in Ingegneria Informatica

Un marketplace per il commercio di prossimità: l'esperienza di Daje!

Relatore:
Prof. Paolo Merialdo

Candidato:
Fabio Barbieri
matricola 526229

Anno Accademico 20/21

Sommario

In questo testo si descrive la progettazione, realizzazione e caratterizzazione di un sistema di applicativi web-oriented per il commercio elettronico di prossimità. In particolare, si farà riferimento all'esperienza di Daje! - una startup romana nata agli inizi della pandemia da Coronavirus. Dopo aver introdotto le peculiarità del paradigma di business, si descriveranno i requisiti minimi che un simile sistema deve soddisfare per implementarlo. A partire da questi, nella parte centrale del testo verrà illustrata l'architettura del sistema e il design dei suoi componenti, con particolare attenzione alle metodologie e tecnologie utilizzate. In conclusione, verranno esposti i risultati di business raggiunti grazie all'implementazione di questi applicativi.

Alla mia famiglia

Indice

Capitolo 1

Il commercio elettronico di prossimità

In questo capitolo viene introdotto il contesto del progetto di tesi. Dopo aver introdotto le caratteristiche essenziali di un E-commerce, si discuterà di come le componente di prossimità abbia sfidato Daje! sul quesito che segue: come si può organizzare un simile sistema per permettere ai consumatori di acquistare commodities online da piccoli negozi di quartiere, che non hanno mai venduto online? La risposta a questa domanda corrisponde alla formulazione di un nuovo paradigma - il commercio elettronico di prossimità. Si introdurrà quindi questo paradigma.

1.1 Alcuni dati sul commercio elettronico in Italia

La percentuale di acquisti online nel periodo 2020 - 2021 è cresciuta esponenzialmente rispetto agli anni precedenti [8]. Iniziamo dando qualche dato sull'impressionante crescita del settore, ponendo l'attenzione sul segmento di maggior interesse nel contesto di questo progetto. Ciò che il periodo in questione ha messo in evidenza infatti, è la confidenza acquisita dai consumatori nell'acquisto online di quella che comunemente chiamiamo "spesa". In precedenza, i grandi numeri del commercio elettronico venivano per la maggiore dall'acquisto nei settori dell'elettronica e della moda. Nel periodo in questione invece, l'acquisto online di generi alimentari - e più in generale del settore Food & Grocery - ha presentato una maggiore e continuativa espansione. Secondo una ricerca sul commercio elettronico in ambito Food dell'Osservatorio Netcomm - Politecnico di Milano [9] - nel solo 2021 si è registrato un aumento del 38% rispetto all'anno precedente. Nel 2020 lo stesso settore aveva già marcato una crescita più che doppia sul mercato online rispetto alla media precedente e nello stesso anno, con oltre 39 milioni di ordini evasi, la spesa online nel settore aveva registrato una movimentazione di 2,7 miliardi di euro. Una crescita dell'70% sul 2019 e pari a circa 1 miliardo di euro in termini assoluti. In particolare il segmento relativo alla consegna dei soli generi alimentari - il Food Delivery - si è confermato quello con la crescita maggiore (fino al 56%), superando la cifra di 1,4 miliardi di euro. Stesso valore, ma con un incremento del 39%, per il Grocery Alimentare - i

prodotti da supermercato - seguito dal segmento dell'Enogastronomia: +17% per circa 750 milioni di euro in termini assoluti. Questa e altre ricerche, ci danno evidenza dell'impatto che l'ampliamento dell'offerta dei servizi di e-commerce ha avuto e continuerà ad avere nel prossimo futuro.

Un altro elemento da evidenziare per il mercato del Food online riguarda la crescita degli acquisti effettuati tramite dispositivi mobili, i quali hanno registrato un incremento del 68% attestandosi come il principale canale d'acquisto con un peso complessivo del 59% sull'intero comparto e movimentazioni sino a 1,5 miliardi di euro.

In totale, la spesa complessiva generata dalla "spesa online" ha superato i quattro miliardi di euro nel suddetto periodo. Come sottolineato dalla ricerca stessa, e com'era facile intuire, l'avvento dell'emergenza sanitaria è la principale causa dell'incremento di domanda online e, conseguentemente, dell'accelerazione nella trasformazione dei processi di vendita dei principali operatori di settore, che non hanno perso tempo per rimodulare le loro strategie e portarle in rete.

Dobbiamo tuttavia porre l'accento sul fatto che questi numeri escludono le vendite delle attività locali (o "di quartiere"). Queste infatti hanno registrato numeri sostanzialmente nella media, non avendo avuto modo di rimodulare i loro servizi online. Sebbene questa sia la situazione dalla parte delle attività locali, secondo una recente indagine condotta da Shopify [12], sei consumatori su dieci, soprattutto giovani, hanno espresso una preferenza per l'acquisto da realtà indipendenti; mentre dall'inizio della pandemia ad oggi, tre consumatori su dieci dichiarano di optare per piccole attività [7]. Indipendentemente dall'età poi, il 65% del totale dei consumatori dichiara di sostenere le piccole attività mentre secondo altre stime, il 76% dei consumatori si affida a brand territoriali per i propri acquisti. La ricerca condotta da Shopify ha poi mostrato come negli ultimi due anni il 61% dei consumatori in Italia ha espresso un cambiamento nelle proprie abitudini di acquisto, il 39% preferisce non recarsi fisicamente in negozio mentre il 63% spende di più per lo shopping online rispetto a quello tradizionale. Notabile, in tal senso, che l'Italia si aggiudichi il secondo posto in Europa dopo la Gran Bretagna.

Un ultimo dato, ancora secondo Netcomm, ed è forse il più rilevante in questo contesto, il 51% degli italiani che ha sperimentato gli acquisti online ritiene che anche i negozi tradizionali debbano cambiare il proprio modello di business in maniera permanente. Risulta evidente dunque che portare i propri prodotti online non è più una scelta, ma una necessità per il prossimo futuro.

1.2 Caratteristiche essenziali di un E-Commerce

Nel seguito di questa sezione, tenteremo di riassumere le caratteristiche essenziali di una piattaforma di e-commerce, col fine di giustificare alcune scelte di business

nell'ambito del progetto.

Contrariamente a quanto intuitivamente si può immaginare, un e-commerce va ben oltre l'implementazione di servizi web-oriented per consentire l'acquisto online. Questi hanno evidentemente il loro ruolo, ma altre importanti responsabilità devono essere prese in considerazione. Dobbiamo pensare ad un e-commerce come un sistema composto da (almeno) sei elementi, riassunti dalla figura 1.1.

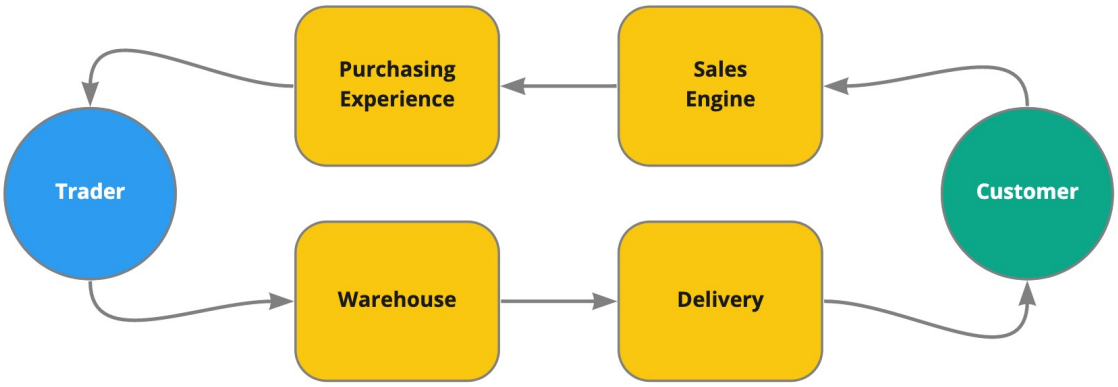

Figura 1.1: Componenti di un E-commerce [5]

Partiremo necessariamente da un mercato: un insieme di utenti-clienti che hanno caratteristiche simili e hanno bisogno di acquistare uno o più prodotti in uno specifico settore. Risulta evidente che, allo stesso modo di un negozio in una strada poco trafficata, un e-commerce poco visitato non sarà in grado di generare le vendite desiderate. Sarà dunque necessario un sistema organizzato (un *sales engine*) per raggiungere gli utenti, guidarli all'acquisto e dar luogo cosi ad una *conversione*. Oggi gran parte di questa attività si svolge online mediante processi di content marketing, email marketing e social-media marketing, sui quali sorvoliamo. L'obiettivo di un sales engine è in definitiva quello di mantenere il flusso di visitatori almeno costante, e sperabilmente, far raggiungere al maggior numero di utenti il luogo virtuale in cui può avvenire l'acquisto. Qui si deve poi fornire un'esperienza che guidi a scegliere e ad acquistare il prodotto giusto, in modo semplice ed immediato. Sarà necessario quindi uno studio sulla user-experience, sull'identità degli utenti stessi e sul "branding" dei rivenditori. La merce deve poi essere gestita, questa viene normalmente conservata in un magazzino e preparata per la spedizione nei più svariati modi. Può trattarsi di materiali non deperibili, oppure di alimenti che devono essere tenuti nella catena del freddo o, ancora, di oggetti fragili che devono essere trattati con cura. Una volta ricevuto l'ordine, occorre preparare i prodotti per la spedizione e affidarsi ad una infrastruttura di delivery. Anche in questo caso, la consegna ha caratteristiche molto diverse in funzione del mercato: dai corrieri di Amazon che consegnano scatole, ai rider in bicicletta che consegnano entro un raggio di due chilometri. Tutto il sistema deve essere governato da un venditore che, oltre a conoscere molto bene il prodotto che vende, deve avere queste competenze per gestire uno store online, organizzare campagne di marketing, fornire assistenza post-vendita e via di seguito.

Tutti gli e-commerce possono essere ricondotti a questo modello, dal piccolo negozio online costruito con piattaforme low-code alla Shopify a colossi come Amazon o realtà come Deliveroo, che abbiamo imparato a conoscere negli ultimi anni.

Il modello che stiamo per descrivere è un'alternativa alle classiche implementazioni di e-commerce dei principali operatori del settore. Elenchiamo brevemente alcune di queste, nei loro tratti salienti, per mostrare la peculiarità (e le necessità) dell'idea di Daje!

1. Piattaforma e-commerce proprietaria: Alcuni operatori del settore, principalmente catene di supermercati, hanno sviluppato la loro piattaforma proprietaria per servizi di consegna a domicilio. In Italia, notabili le realtà di Esselunga e Coop, i cui consumatori abituali hanno già a disposizione servizi online con una decente esperienza utente. Queste, si sono preoccupate di implementare precisamente tutte le componenti elencate, avendo a disposizione le giuste risorse in termini economici e umani. Che abbiano sviluppato internamente il servizio, o che abbiano dato in out-sourcing la responsabilità, hanno ora a disposizione (proprietaria) l'intero meccanismo.

2. Servizi di consegna di terze parti: In questo modello, viene delegata la componente di delivery a società diverse da quella del rivenditore, realtà indipendenti che impiegano i propri addetti alle consegne (*shoppers*). In Italia il più importante di questi è Everli (già Supermercato24).

3. Dark-Stores.
   Questa categoria comprende tutte le società di e-commerce che non hanno un negozio fisico e operano solo online. In concorrenza diretta con la grande distribuzione, utilizzano una rete di magazzini locali per raggiungere l'obiettivo di consegnare la spesa ai propri clienti in un tempo molto breve. È un modello molto simile a quello dei servizi di consegna per i ristoranti. In questa categoria troviamo aziende già affermate come l'olandese PicNic, insieme a startup come la tedesca Gorillas e la inglese Dija.

Questi modelli consentono alle persone di acquistare la spesa online, ma nessuno di essi consente ai clienti di acquistare dai piccoli negozi nei loro quartieri.

1.3 Il modello di Daje!

Agli inizi della pandemia di Coronavirus, dall'esigenza delle attività locali di continuare ad operare i propri commerci durante i periodi di lockdown è nato il progetto di Daje! Una startup operante sul territorio romano, di cui riportiamo la mission con le parole ufficiali:

> *Utilizziamo Internet e la tecnologia per aiutare le famiglie a fare acquisti nei piccoli negozi vicino a loro, sostenendo così l'economia e il benessere del quartiere in cui vivono. Lavoriamo per offrire ai clienti un'esperienza di acquisto eccellente, fornire ai rivenditori strumenti semplici per gestire cataloghi e ordini e coordinare un servizio di consegna a domicilio efficiente ed economico.* [5]

In queste parole è assente ogni riferimento alle impossibilità imposte dal periodo pandemico. Queste infatti hanno agito esclusivamente da driver di un'idea di business che va oltre l'emergenza e che recepisce la situazione che abbiamo cercato di riassumere nella sezione sui dati. Quando in piena emergenza attività locali come panetterie, drogherie, macellerie fino ai banchi dei mercati rionali hanno iniziato a muoversi online; organizzando ordini e consegne sui canali che conoscevano - principalmente social networks come Facebook e Instagram - Daje! ha osservato i loro tentativi per sintetizzare l'idea di un progetto che potesse dotarle di strumenti utili a garantirgli la competitività necessaria nei nuovi scenari del commercio online; sostituendosi al singolo negoziante nell'implementazione di tutte le responsabilità illustrate nella sezione precedente. Il problema è dunque il seguente: come si potrebbe organizzare un e-commerce per permettere a un consumatore di comprare online la spesa dai negozi di quartiere, che non hanno mai venduto online? Rispondere a questa domanda corrisponde a formulare un nuovo paradigma, che abbiamo chiamato e-commerce di prossimità. Per la realizzazione del nostro modello, abbiamo escluso di proporre ai singoli negozianti di aprire il proprio e-commerce, per semplici e quasi ovvi motivi di alfabetizzazione digitale. Le abilità tecnologiche degli esercenti interessati si fermano spesso all'uso di qualche app sul cellulare. Da escludere quindi soluzioni proprietarie alla Shopify, in quanto le possibilità che un fornaio, un macellaio o un agricoltore possano investire tempo e denaro per imparare a vendere online con successo sono molto rare. Questo processo necessità dell'implementazione delle componenti analizzate, le quali già nell'industria sono sempre più spesso responsabilità di figure professionali verticalizzate (e molto varie). Chi si occupa di cibo conosce molto bene i prodotti che vende, spesso li coltiva o li produce, li sa disporre sul bancone in modo accattivante ed è bravo a raccontarli ai clienti quando entrano in un negozio. Questo è il lavoro che Daje! ha tentato di accompagnare online. Allo stesso tempo poi, se ogni negoziante avesse un proprio negozio online, sarebbe estremamente difficile per un cliente fare la spesa. Come tenteremo di giustificare più avanti, risulta molto più comodo affidarsi ad un aggregatore che consenta la creazione di un unico carrello, emulando il processo reale della spesa. L'e-commerce di prossimità è dunque, necessariamente, un marketplace che aggrega i negozianti

di un quartiere in un unica app.

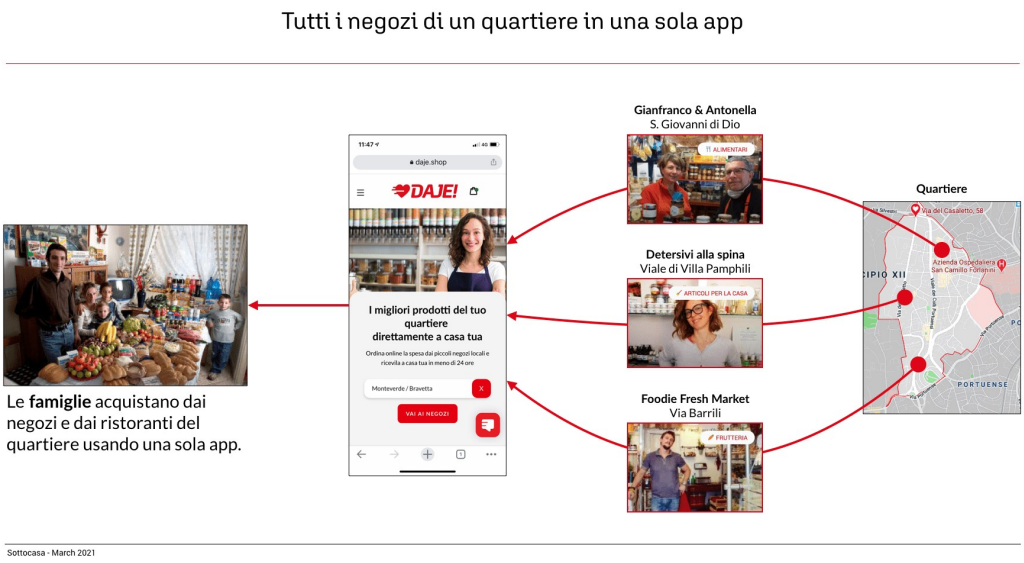

Figura 1.2: Overview del modello

Questa scelta consente di costruire un brand riconoscibile e facilmente memorizzabile. Allo stesso tempo, permette di ottimizzare il costo di acquisizione dei clienti e organizzare un sales engine efficiente, che i singoli negozianti da soli non sarebbero mai in grado di mettere in piedi. Una volta conquistata l'attenzione dei consumatori, l'esperienza di acquisto è uno degli elementi più delicati di un e-commerce e ancor più di un marketplace di prossimità. Bisogna infatti considerare alcune questioni chiave. Innanzitutto, non dobbiamo dimenticare che compriamo con gli occhi: l'esperienza di entrare in un negozio di alimentari, ricolmo di opportunità d'acquisto, è difficile da replicare online. In secondo luogo, la maggior parte dei prodotti che si possono trovare nei negozi di alimentari, frutta e verdura, macellerie o pescherie sono venduti a peso, non confezionati. Pertanto, un e-commerce di prossimità deve prevedere un processo che tenga conto delle peculiarità dell'esperienza del negozio di alimentari locale, dove i prezzi di molti prodotti cambiano quasi quotidianamente e il costo finale dipende dal peso del prodotto sulla bilancia. Il magazzino è probabilmente la parte più semplice del sistema. I punti vendita sono già dei sistemi organizzati per la gestione dei processi che interessano il ciclo delle merci. Naturalmente il settore del fresco e ancor più del deperibile, è caratterizzato da una rotazione molto veloce dei prodotti, non sempre disponibili. Molti negozi di alimentari spesso cambiano intenzionalmente l'assortimento di alcuni dei loro prodotti per invogliare i clienti a tornare per nuove offerte. Pertanto, è necessario mettere nelle mani dei commercianti uno strumento che permetta loro di gestire i propri cataloghi. Infine, il sistema di consegna deve essere efficiente ed economicamente sostenibile. Aziende come Deliveroo e Glovo hanno adottato un modello pensato per prendere un pasto caldo da un ristorante e consegnarlo al cliente in meno di 20 minuti. Ogni consegna

viene effettuata singolarmente ed è quindi molto costosa (i ristoratori pagano fino a 1/3 del valore dello scontrino, mentre il cliente può pagare fino a 10 euro per il servizio di consegna). Questo modello non è applicabile alla spesa, in cui il margine sui prodotti è già contenuto e soprattutto, non vi è un reale bisogno di rapidità nella consegna. In un e-commerce di prossimità, il cliente fa la spesa riponendo in un unico carrello la merce proveniente da più negozi. Il giorno successivo, un fattorino raccoglie tutti gli articoli dei clienti dai negozianti, li organizza ed effettua le consegne. Poiché negozi e clienti si trovano nello stesso quartiere, un fattorino può ritirare e consegnare dozzine di in una sola mattina, contenendo il costo del servizio.

Nel proseguimento di questo testo, sorvoleremo sugli ulteriori aspetti legati al business, di cui ci siamo limitati brevemente ad esporre le caratteristiche e il contesto, per concentrarci sull'analisi dei requisiti per gli asset tecnologici che ne hanno consentito l'implementazione.

Capitolo 2

Le sfide da affrontare e i problemi da risolvere

In questo capitolo approfondiremo i requisiti di maggiore interesse nell'ambito del progetto. Dopo averli descritti informalmente, individueremo i casi d'uso e svolgeremo un'analisi tecnica servendoci del modello di dominio, dei diagrammi di sequenza di sistema e dei contratti per i casi d'uso implementati dai servizi in esame.

2.1 Consentire l'acquisto online

Il requisito fondamentale del sistema che implementa il marketplace è ovviamente consentire l'acquisto online. Poichè tutto il modello di business è incentrato sulla prossimità, si palesa immediatamente una restrizione sulla scelta dei prodotti. Il cliente deve poter acquistare soltanto prodotti di esercenti nel suo quartiere o in quelli limitrofi. Questo vincolo serve essenzialmente due finalità. La prima riguarda la fattibilità dell'intero modello, se un utente potesse ordinare da esercenti molto distanti da lui, sarebbe impossibile organizzare in modo efficiente la struttura di delivery quando il numero di ordini aumenta. La seconda è invece legata all'esperienza dell'utente sulla piattaforma: si evita di far "spendere" interazioni per azioni che comunque non potrebbero portare all'acquisto. L'utente dovrà pertanto fornire dapprima il suo CAP, in modo che gli sia permesso l'acquisto dai soli esercenti che operano in quelle zone.
Inserito il CAP, il sistema mostra una galleria di prodotti, filtrati in modo tale che il venditore risieda in quel CAP. Questa è un'altra sottile scelta di UX. Ci si è più volte chiesto se fosse il caso di privilegiare l'esercente o il prodotto, mostrandolo da subito all'utente. Numerosi A/B test hanno alla fine mostrato come la prima fosse la scelta migliore dal punto di vista dell'utente. Si è scelto dunque di non mostrare la lista di esercenti nella zona, ma da subito la galleria dei loro prodotti. D'altronde, quando entriamo in un supermercato ci concentriamo dapprima sui prodotti e poi sulle marche. In questo modo, risulta più facile per il cliente comporre un carrello. Si è cercato dunque di minimizzare quanto possibile le interazioni finalizzate alla ricerca dei prodotti, a questo scopo, l'applicazione permette di filtrare i prodotti (e

i relativi esercenti) per categoria. Un'altra difficile sfida è quindi l'ideazione di una tassonomia adeguata dei prodotti (su cui sorvoleremo). Sarà poi possibile navigare la vetrina di un esercente specifico, raggiungendola a partire dai prodotti o tramite una barra di ricerca. Con i prodotti a vista, l'utente è in grado di comporre un carrello, aggiungendo o rimuovendo elementi (righe d'ordine). Riempito il carrello, sarà possibile proseguire con il *checkout*. In questa fase, il cliente può modificare ancora le righe d'ordine prima di inserire l'indirizzo di consegna e il metodo di pagamento. A questo punto, l'utente potrà registrarsi (qual'ora non fosse ancora registrato) per salvare le sue preferenze per l'indirizzo di consegna e il metodo di pagamento. In questo momento, potrà inserire delle note per il fattorino, insieme ad altri appunti per i singoli negozianti (un esempio reale: "l'orata pulita e sfilettata"). Completato l'ordine, l'applicazione informa l'utente sui tempi previsti di consegna (il giorno seguente) e invia una mail con il riepilogo dell'ordine e informazioni per l'assistenza post-acquisto. Il tempo ha poi mostrato come gli utenti che tornano spesso all'app acquistino fondamentalmente sempre gli stessi prodotti, per questo un'altra funzionalità a a disposizione dell'utente registrato è l'accesso ai prodotti e negozi preferiti, cioè quelli da cui ha acquistato più frequentemente e ancora, l'accesso agli ordini già effettuati e la possibilità di sottometterli di nuovo, con lo stesso carrello.

Abbiamo descritto questi requisiti in maniera informale, poichè come dettaglieremo nel capitolo successivo, si è scelto di adottare una soluzione low-code, Shopify, la quale ha consentito l'implementazione di questi direttamente, senza necessità di un progetto tecnico specifico. D'altronde si tratta pur sempre di commercio elettronico, per cui ripeterne l'analisi e il progetto dell'implementazione dei requisiti fondamentali sarebbe poco interessante. Ci concentreremo invece sui requisiti più peculiari per il dominio d'esame, di cui invece svolgeremo l'analisi tecnica.

2.2 Gestire Il ciclo di vita degli ordini

Come abbiamo accennato in precedenza, frutta, verdura, carne e tutti i generi alimentari che troviamo in banchi del mercato o piccoli negozi sono generalmente venduti al peso. Il prezzo finale è quindi determinato al momento della vendita, pertanto non è noto a priori il totale dello scontrino. L'applicazione tiene conto di questo e, come vedremo, ha digitalizzato con successo il ciclo di vita degli ordini senza impattare il workflow abituale degli esercenti. Quando un cliente sottomette un ordine autorizza un budget di spesa per il suo carrello, entro il quale deve essere mantenuto l'importo scontrinato. Sottomesso un ordine (un carrello), l'applicazione dovrà estrarre dall'ordine principale gli ordini per i singoli esercenti, in modo da notificarli per tempo e permetterne il processamento. Non solo il prezzo finale non è noto al momento della sottomissione dell'ordine, ma gli stessi prodotti in questione potrebbero non essere disponibili al momento della consegna, in quanto la loro disponibilità è altamente variabile. Sebbene un negoziante abbia modo, di gestire il suo catalogo; non potrà prevedere con sicurezza la disponibilità per ciascuna riga di un ordine sottomesso giorni prima. Pertanto, una volta notificato dell'ordine,

il negoziante dovrà comunicare queste disponibilità - diremo che dovrà confermare l'ordine. Un esercente ha la facoltà di specificare il suo calendario d'apertura, ma potrebbe ancora non essere in grado di confermare un ordine per uno specifico giorno di consegna (il negozio deve rimanere chiuso quel giorno). Per tenere in conto di questa evenienza, l'esercente deve poter declinare l'ordine completamente. In un secondo momento, se l'ordine è confermato, l'esercente potrà effettivamente preparare l'ordine, effettuerà cioè le pesate ed applicherà eventuali sconti o altre modifiche ai prezzi finali. A seguito di questo processo, egli comunicherà l'importo finale tramite l'applicativo. Il sistema dovrà quindi aggregare i singoli totali per esercente, comunicare il prezzo finale al cliente e addebitare lo stesso importo sul metodo di pagamento inserito in fase di checkout. Il cliente sarà quindi notificato sulla presa in consegna dell'ordine e su eventuali variazioni sul carrello che aveva composto, in base a quanto specificato dall'esercente sulla disponibilità delle righe d'ordine. Un'ulteriore sfida si presenta quindi per gli amministratori. Chi è addetto all'operatività del servizio deve poter monitorare l'intero processo, per questo l'applicazione deve permettere loro di avere una vista sugli ordini e sul loro stato. Un amministratore potrà quindi visualizzare una lista di ordini ed eventualmente filtrarli per data, quartiere o singolo esercente. Nella fase iniziale del progetto, gli amministratori contattano direttamente (human-tasks) il cliente qualora la composizione del carrello sia variata significativamente, cioè in caso di mancata disponibilità di molti dei prodotti che aveva inserito nel carrello. La gestione di questi casi limite, è stata delegata ad attività umana, con l'intenzione di automatizzarla nella fase successiva del progetto.

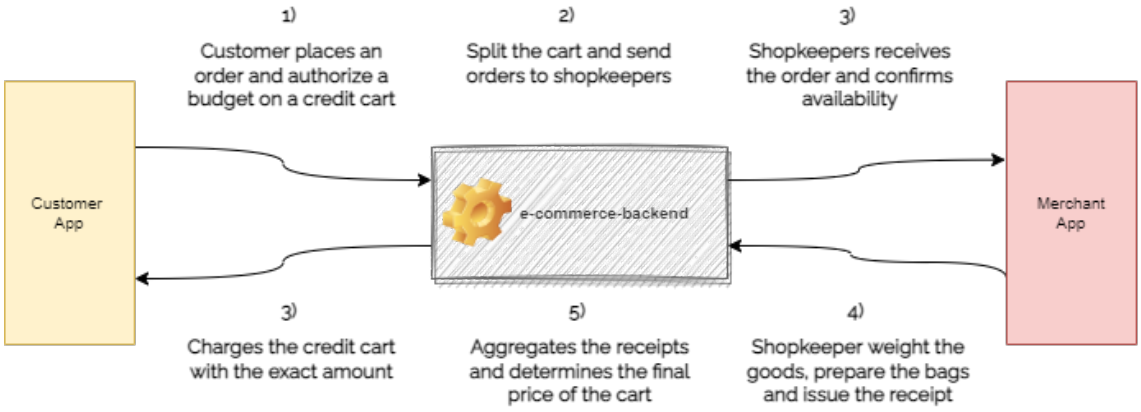

Figura 2.1: Flusso di gestione degli ordini

Dobbiamo quindi individuare almeno tre attori che interagiscono con il sistema: il cliente, l'esercente e l'amministratore. Questi interagiscono ovviamente con finalità diverse: il cliente vuole acquistare un carrello da una moltitudine di negozi nel suo quartiere e ricevere la spesa a casa; l'esercente vuole prendere visione degli ordini relativi al suo negozio e gestirli; l'amministratore vuole monitorare e registrare il processo, al fine di gestire i casi limite e notificare i clienti.
A partire da questa descrizione, abbiamo individuato precisamente i casi d'uso per

il flusso di gestione degli ordini, per i quali ci serviremo del modello di dominio mostrato in figura.

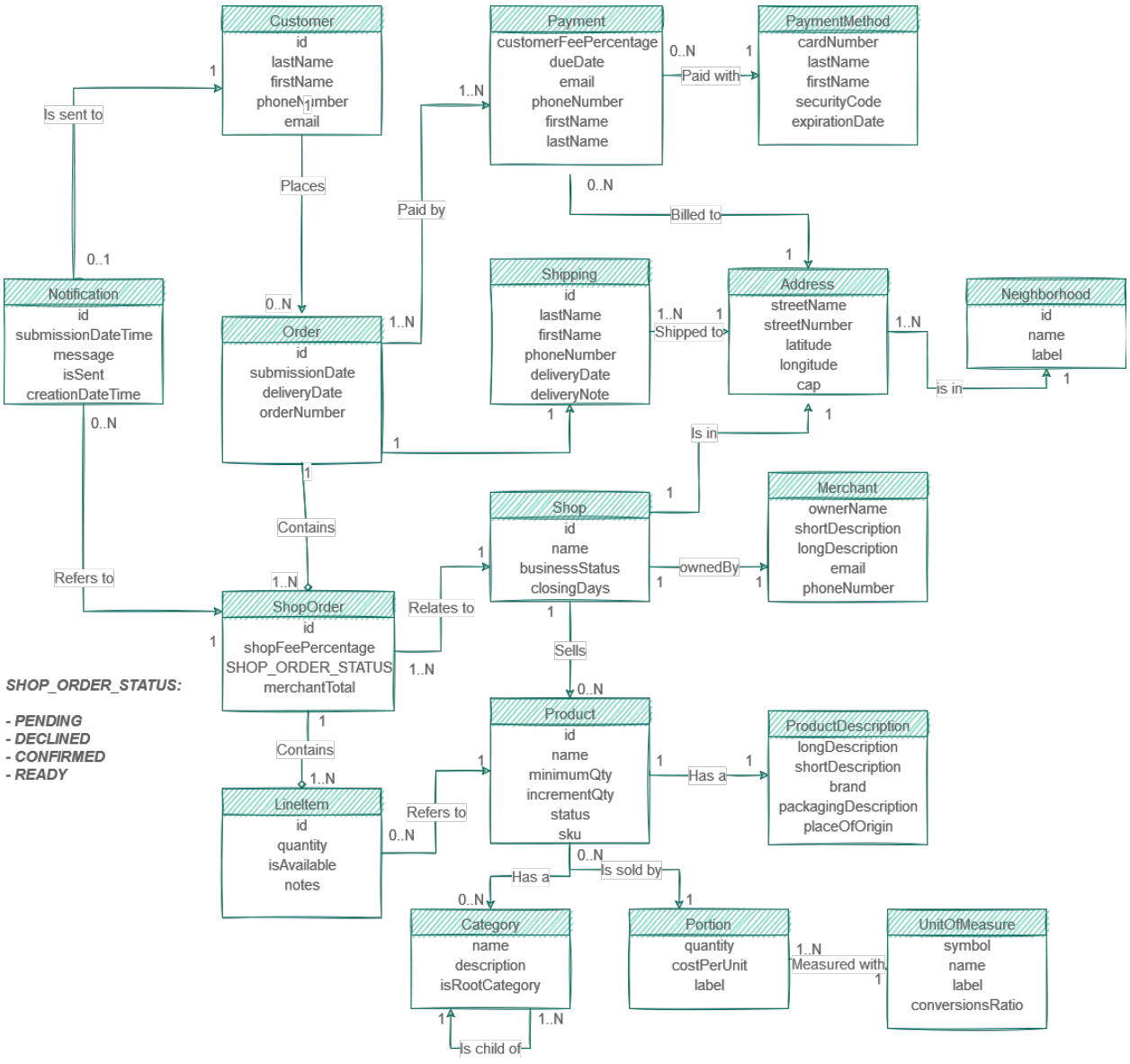

Figura 2.2: Modello di domino

Conferma dell'ordine
Attore primario: Esercente
Interessi: l'esercente vuole confermare la disponibilità delle righe d'ordine per il giorno di consegna.

1. Un esercente richiede la lista degli ordini da confermare.
2. L'esercente sceglie un ordine da confermare.
3. L'esercente indica la disponibilità di ciascuna riga d'ordine presente.
4. L'esercente conferma l'ordine.

Requisiti speciali: Con "Ordine" intendiamo qui referenziare un ordine-per-negoziante (ShopOrder).

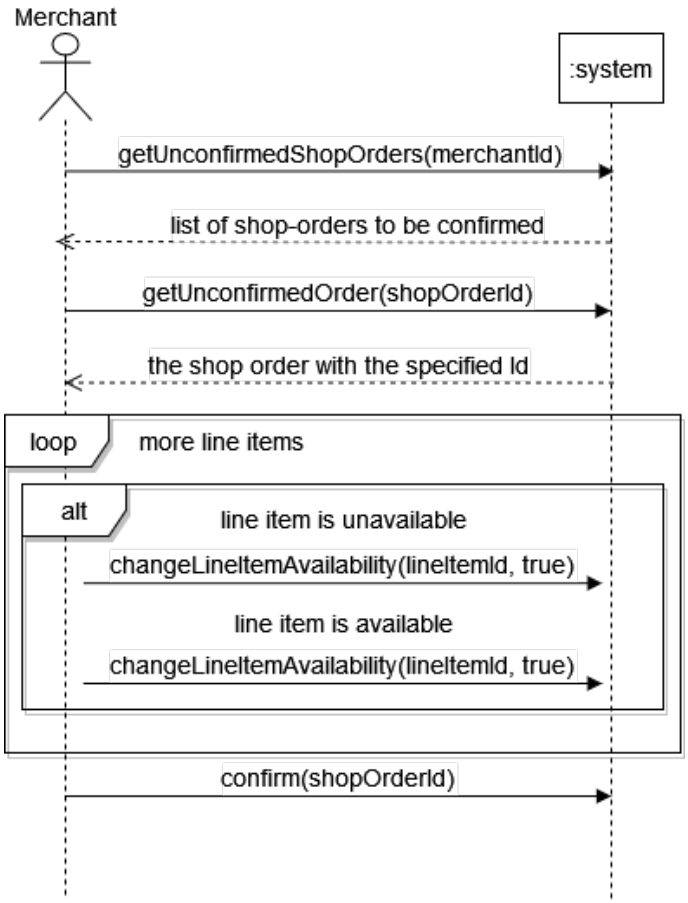

Figura 2.3: UC1: Diagramma di sequenza di sistema

Le due operazioni di sistema importanti per questo caso d'uso sono certamente *changeLineItemAvailability* e *confirm*. Le cui finalità sono rispettivamente: confermare la disponibilità di un prodotto per lo specifico ordine e portare l'ordine nello stato *confermato*, in modo da poterlo preparare in seguito.

Contratto CO1: changeLineItemAvailability(lineItemId, availability):

Pre-condizioni:

1. Un ordine per il negozio con questo *merchantId* è stato creato e completamente inizializzato.

Post-condizioni:

1. Aggiornato il valore dell'attributo *isAvailable* con il valore di *availability* dell'istanza di *LineItem* con questo *lineItemId*.

Contratto CO2: confirm(shopOrderId):

Pre-condizioni:

1. Vedi le pre-condizioni del contratto CO1.

Post-condizioni:

1. Aggiornato il valore dell'attributo *shopOrderStatus* con il valore *CONFIRMED*.
2. Creata un'istanza di *Notification* (**NT**).
3. Valorizzato il campo *creationDateTime* con un valore che rappresenta il momento attuale.
4. Valorizzato il campo *message* di (**NT**) con un messaggio di conferma.
5. Formata l'associazione tra **NT** e l'istanza di *shopOrder* con id *shopOrderId* (ora **SO**).
6. Formata l'associazione tra **NT** e l'istanza di *Customer* relativa ad *Order*, di cui **SO** fa parte.

Declino dell'ordine
Attore primario: Esercente
Interessi: l'esercente vuole declinare un ordine poichè non è in grado di garantire la consegna per il giorno previsto

1. Un esercente richiede la lista degli ordini da confermare.
2. L'esercente sceglie un ordine da confermare.
3. L'esercente declina l'ordine.

Requisiti speciali: nessuno.

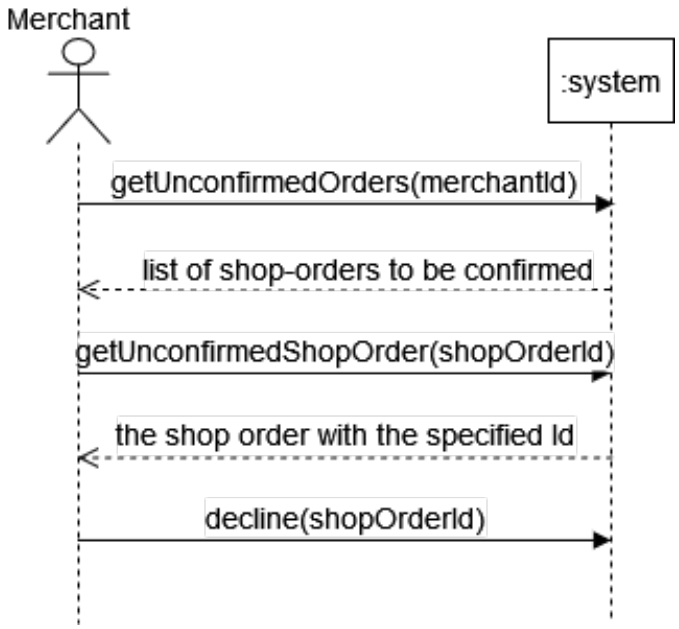

Figura 2.4: UC1.a: Diagramma di sequenza di sistema

Contratto CO3: decline(shopOrderId):

Pre-condizioni:

1. Vedi le pre-condizioni del contratto CO1.

Post-condizioni:

1. Aggiornato il valore dell'attributo *shopOrderStatus* con il valore *DECLINED*.
2. Creata un'istanza di *Notification* (**NT**).
3. Valorizzato il campo *creationDateTime* con un valore che rappresenta il momento attuale.

4. Valorizzato il campo *message* di (**NT**) con un messaggio che informa sull'impossibilità di consegnare l'ordine.

5. Formata l'associazione tra **NT** e l'istanza di *shopOrder* con id *shopOrderId* (ora **SO**).

6. Formata l'associazione tra **NT** e l'istanza di *Customer* relativa ad *Order*, di cui **SO** fa parte.

Preparazione di un ordine
Attore primario: Esercente
Interessi: l'esercente vuole preparare un ordine, cioè inserire il totale scontrinato a seguito delle eventuali pesate.

1. Un esercente richiede la lista degli ordini da preparare.

2. L'esercente sceglie un ordine da preparare.

3. L'esercente indica l'importo finale dell'ordine.

4. L'esercente conferma l'avvenuta prepazione dell'ordine.

Requisiti speciali: La lista degli ordini da preparare contiene solo ordini confermati, con consegna in due giorni e con importo finale che non può superare quello autorizzato dal cliente in fase di checkout.
Nota: con "Ordine" intendiamo qui referenziare un ordine-per-negoziante (ShopOrder).

Le due operazioni di sistema importanti per questo caso d'uso sono certamente *insertShopOrderTotal* e *prepare*. Le cui finalità sono rispettivamente: inserire il prezzo totale scontrinato del carrello e portare l'ordine nello stato *READY*, cioè pronto per la consegna.

Contratto CO4: insertShopOrderTotal(shopOrderId, total):

Pre-condizioni:

1. L'istanza di *ShopOrder* con id *shopOrderId* (ora **SO**) ha il campo *shopOrderStatus* valorizzato a *CONFIRMED*.

2. L'ordine è in consegna tra due giorni, cioè l'istanza di *Order* a cui è associato **SO** ha un valore di *deliveryDate* che rispetta questo vincolo.

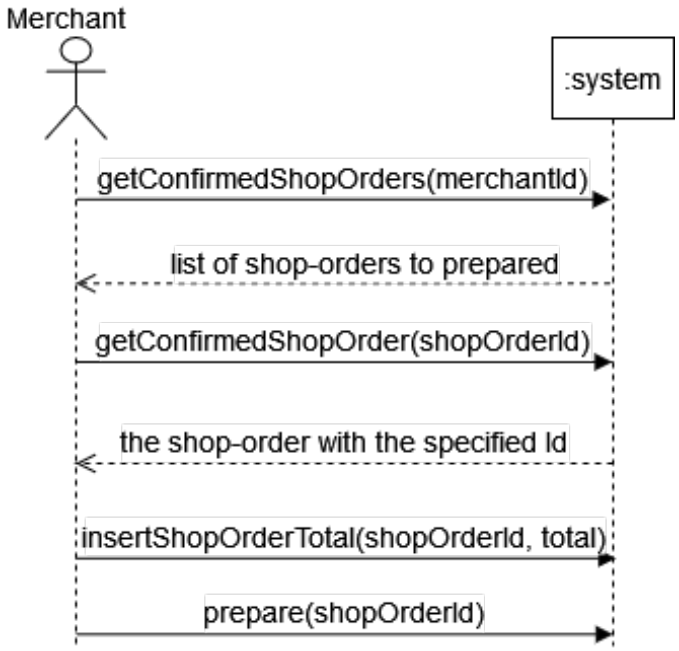

Figura 2.5: UC2: Diagramma di sequenza di sistema

Post-condizioni:

1. Aggiornato il valore dell'attributo *merchantTotal* con il valore del totale scontrinato.

Contratto CO5: prepare(shopOrderId):

Pre-condizioni:

1. L'istanza di *ShopOrder* con id *shopOrderId* (ora **SO**) ha il campo *shopOrderStatus* valorizzato a *CONFIRMED*.
2. L'istanza **SO** ha il campo *merchantTotal* valorizzato, non nullo e diverso da 0.

Post-condizioni:

1. Aggiornato il valore dell'attributo *shopOrderStatus* con il valore *READY*.

Notifica del cliente sullo stato di un ordine
Attore primario: Amministratore
Interessi: L'amministratore vuole notificare il cliente sullo stato di un ordine che ha sottomesso.:

1. Un amministratore richiede la lista degli ordini effettuati in un certo lasso temporale.
2. L'amministratore sceglie un ordine per negoziante.
3. L'amministratore invia una notifica al cliente con i dettagli dell'ordine per il negozio.

Requisiti speciali: L'invio della notifica non deve togliere troppo tempo ad un amministratore.

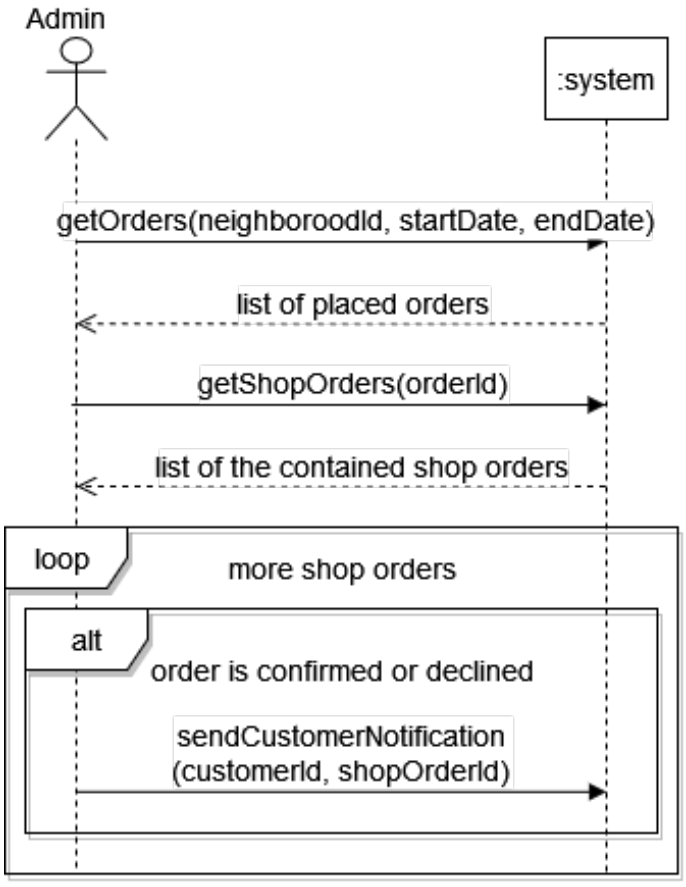

Figura 2.6: UC3: Diagramma di sequenza di sistema

L'operazione di sistema importante per questo caso d'uso è *insertShopOrderTotal*. Con questa operazione, si vuole inviare al cliente un messaggio di notifica sullo stato dell'ordine.

Contratto CO6: sendCustomerNotification(shopOrderId):

Pre-condizioni:

1. Un'istanza di *Notification* è stata creata ed associata ad un *ShopOrder* con id *shopOrderId*.

Post-condizioni:

1. Aggiornato il valore dell'attributo *isSent* con il valore *true*.
2. Aggiornato il valore dell'attributo *submissionDateTime* con un valore che rappresenta il momento attuale.

Capitolo 3

Overview del progetto

In questo capitolo verrà introdotta l'architettura di massima della soluzione, discutendo le sfide e le scelte intraprese dal punto di vista tecnico, metodologico e di business. Si discuterà poi delle tecnologie scelte per l'implementazione dei singoli servizi.

3.1 Scelte dettate dal Time-to-Market

Come tutte le start-ups, anche Daje! ha operato con la necessità di minimizzare il più possibile il Time-to-Market, tenendo bassi i costi senza ledere la qualità del servizio. Una scelta obbligata per un progetto la cui ambizione di scalabilità è legata alla previa validazione dell'idea di business. Questo obiettivo, di validare l'idea di business su quanti più quartieri possibili, ha quindi significativamente influenzato le scelte tecnologiche per l'implementazione dell'intera soluzione. Si è scelto quindi, per lo sviluppo di questa, di adottare quanto più possibile servizi di terze parti che consentissero di raggiungere l'operatività con il minimo sforzo di sviluppo. Bisognava dunque capire cosa valeva la pena implementare da subito come tecnologia proprietaria e cosa si sarebbe potuto ottenere con servizi terzi. In relazione ai requisiti discussi in precedenza, tratteremo ora due di questi servizi e della loro applicazione al dominio in esame.

3.1.1 Shopify: Una soluzione low-code per consentire l'acquisto online

Shopify è una tra le molte piattaforme low-code/no-code (SaaS) che permettono la creazione di piattaforme di e-commerce in cloud. In particolare, esso è tra i più completi se consideriamo la definizione di e-commerce che abbiamo dato nel primo capitolo. Per un elenco completo degli strumenti a disposizione nella piattaforma si rimanda alla sua documentazione ufficiale [13], mentre qui vogliamo evidenziare i motivi per cui la sua adozione ha avuto senso in Daje! e descrivere come questi hanno consentito l'implementazione del caso d'uso relativo all'acquisto. Dal punto

di vista tecnico, per implementare qualsiasi applicazione web, oltre alla logica di business e alla sua modellazione, dobbiamo pensare alle seguenti responsabilità:

1. Implementazione e gestione dell'infrastruttura web:
2. Pubblicazione degli applicativi
3. Sicurezza degli applicativi
4. Integrazioni con altri servizi: strumenti di marketing e gateway di pagamento.

L'impiego di Shopify in Daje! - e in generale per startups di commercio elettronico - ha consentito di non curarsi - momentaneamente - di queste responsabilità critiche e dispendiose, per concentrarsi esclusivamente sullo sviluppo del servizio e dell'esperienza utente. Per quanto riguarda il design e lo sviluppo frontend infatti, Shopify consente di implementare *temi* personalizzati mediante l'utilizzo di consolidate tecnologie web (HTML, CSS e Javascript). Più in particolare, Shopify modella il dominio delle vendite con strutture dati fisse - comuni a tutti gli account - con le quali mediante *Liquid*, si possono costruire esperienze personalizzate complesse a piacere. Liquid è il *template engine* sviluppato da Shopify e scritto in Ruby e rappresenta la spina dorsale di tutti i suoi temi [14]. Un template language permette di riutilizzare gli elementi statici che definiscono il layout di una pagina web, popolando dinamicamente i suoi elementi con le informazioni memorizzate nelle strutture dati di un negozio Shopify. In un tema, gli elementi statici sono scritti in HTML e quelli dinamici in Liquid; il quale fornisce allo scopo strutture sintattiche per il controllo di flusso (ad es. if statements) e per l'iterazione sugli elementi di una struttura (ad es. for statement). Dal punto di vista dei dati, una buona panoramica delle parti comuni del dominio delle vendite modellate in Shopify è data nel diagramma ERD in [2]. Inevitabilmente, queste strutture non aderiscono perfettamente al modello di dominio che abbiamo mostrato con la figura 2.2. Pur tuttavia, è stato possibile sopperire a questa limitazione e adattare le informazioni al dominio in esame per rendere cosi possibile l'implementazione del caso d'uso descritto nella sezione 2.1. Per adattare le strutture native, Shopify mette a disposizione la struttura dei *metafields* [15].

I *Metafields* sono una soluzione flessibile per aggiungere informazioni addizionali ad una risorsa nativa. Memorizzano un valore insieme al suo tipo di dato, l'attributo *type*. Per identificare un metafield sono poi disponibili gli attributi *owner-resource* - la risorsa a cui un'istanza si riferisce, un *namespace* - per evitare conflitti nei nomi di diversi metafields e una *key* - che identifica univocamente un valore nell'ambito del namespace. Il *value* di un metafield può essere una stringa in formato JSON, per cui è possibile memorizzare gli attributi di una nuova definizione concettuale come attributi di un oggetto standard. L'attributo *key* di un metafield rappresenta di fatto il nome di un'istanza di questa classe (ad. es *"Category"*), che va ad aggiungersi a quelle native in Shopify. Una delle limitazioni più rilevanti del modello dei dati nativo, è l'impossibilità di definire più *Shop* in un account. In Shopify, esiste logicamente un solo negozio, mentre per il progetto è necessario, come abbiamo visto, rappresentarne una moltitudine. Per rappresentare un esercente locale, si è

scelto di usare la struttura nativa delle *Collections*. Una *Collection* è una collezione di prodotti che può essere creata manualmente, oppure definendo regole che automaticamente determinano se un prodotto sarà incluso. Creando una collezione di prodotti, è stato possibile definire su di essa opportuni (*metafields*), con cui modellare le informazioni sugli esercenti che li vendono e le categorie merceologiche a cui appartengono. Liquid mette poi a disposizione dei filtri con cui operare sugli attributi di una risorsa - e quindi anche sui metafields definiti per essa. Mediante questi, è stato ad esempio possibile filtrare collezioni di prodotti in base al valore di un metafield con *key* "merchant", ottenendo cosi i prodotti di un singolo negoziante al fine di inserirli nel template delle varie pagine, insieme alle informazioni del negozio stesso memorizzate nel campo *value*. La stessa strategia è stata utilizzata quindi per le categorie (*category*) e i quartieri. La figura 3.1 mostra le relazioni tra le risorse native e i metafields definiti nel caso dei negozianti.

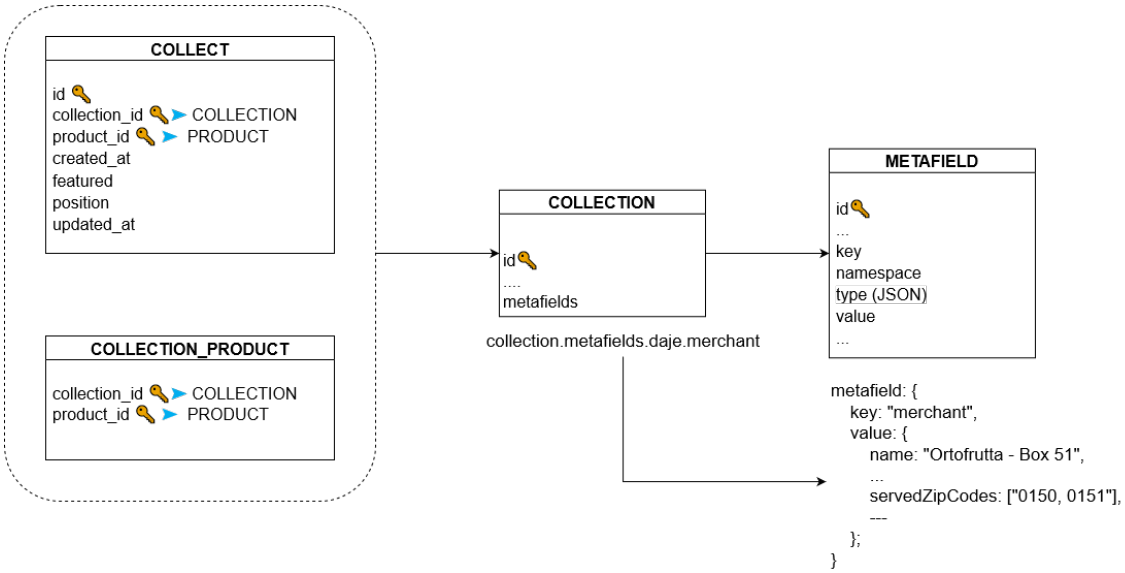

Figura 3.1: Relazione tra Collection e Metafield per la gestione di più negozi

Per soddisfare il requisito sull'area servita per il cliente, cioè filtrare i prodotti da mostrare per CAP in cui risiedono i negozianti che li vendono, è stato necessario integrare una API di geolocalizzazione esposta da Google (Geocoding API [11]). Ottenuto il CAP dell'indirizzo del cliente, è stato sufficiente filtrare con Liquid le collezioni di prodotti per metafields con *key: Merchant* che contenessero il valore del CAP inserito nell'array di *servedZipCodes*. Il resto delle operazioni descritte nel caso d'uso relativo all'acquisto sono state implementate con funzionalità integrate di Shopify.

Come approfondiremo nel seguito, l'adozione di questa piattaforma ha avuto i suoi svantaggi dal punto di vista tecnico. I metafields hanno permesso di plasmare il database interno secondo le necessità, ma si tratta di un *workaround*. Mentre da un lato questa soluzione permette l'implementazione delle interfacce utente deside-

rate, dall'altro rende difficoltosa la manutenzione e l'evoluzione del modello dei dati per la gestione delle informazioni, la gestione dei cataloghi e l'onboarding di nuovi negozianti. Nel capitolo sul design dei servizi di backend, approfondiremo come sia stato possibile integrare i dati degli ordini in Shopify con gli altri applicativi, di fatto rendendo questa implementazione il frontend per i clienti di Daje!

3.1.2 Adalo: Una soluzione NoCode per consentire la gestione degli ordini agli esercenti

Adalo è una giovane soluzione no-code per la creazione di applicazioni mobile native multipiattaforma e applicazioni web. Per gli stessi motivi che hanno portato all'adozione di Shopify, Adalo ha rappresentato un valido compromesso tra rapidità di sviluppo e aderenza della soluzione ai casi d'uso in esame. Con Adalo, è stato possibile realizzare un'applicazione mobile multipiattaforma, in dotazione ai negozianti, per per la gestione degli ordini come descritta nel capitolo 2.

Diamo una panoramica dei concetti principali per sviluppare in Adalo.

1. Screens: Sono le "viste" o "pagine" dell'applicazione.
2. Components: i componenti essenziali con cui costruire l'applicativo. Una lista completa di quelli a disposizione può trovarsi nella documentazione ufficiale [1]. Certamente i più significativi per il caso d'uso in esame sono le *Custom List* - liste di elementi UI composti da gruppi di altri elementi.
3. Actions: Usate per specificare azioni al click di un *Component*. Mediante esse, è possibile definire logiche condizionali per i cambiamenti di stato nell'interfaccia. Come vedremo nel seguito, con la funzionalità delle *Custom Actions* è possibile implementare funzioni non-native, come ad esempio l'uso di servizi esterni mediante REST APIs. Con le *Actions* è poi possibile implementare operazioni CRUD selle collezioni del Database.
4. Database Collections: In Adalo è presente un database interno nel quale è possibile definire classi e relazioni con il modello relazionale. Le classi definite nel database vengono connese ai componenti UI, popolando il loro stato con le informazioni contenute nelle loro istanze. La collezione nativa più importante nell'applicazione per i negozianti è certamente "Users" con cui è stato possibile delegare il flusso di autenticazione agli automatismi di Adalo.

Il vantaggio principale di questa soluzione è la possibilità di ottenere build iOS e Android a partire dalla stessa implementazione, riducendo cosi drasticamente l'effort per lo sviluppo dell'app grazie all'utilizzo di un editor drag-and-drop per la creazione delle viste, degli elementi di UI e del loro stato.

Come abbiamo visto, i clienti acquistano in Shopify e i dati sugli ordini vivono nel suo backend, pertanto la sfida principale nell'implementazione dell'app con Adalo è l'integrazione dei dati sugli ordini-per-negoziante all'interno della piattaforma.

Le funzionalità che hanno permesso questa integrazione sono le *Custom Actions* - come anticipato - e, soprattutto, le *External Collections*. Le *External Collections* sono una potente funzionalità che consente di connettere un'app con altre app e servizi mediante REST API. Tutto ciò che è necessario, per usarle come collezioni standard e dunque popolare liste, definire azioni su di esse e form per operazioni CRUD, è la loro configurazione. Per ottenere allora una collezione da un servizio esterno, occorre dapprima configurare il Base URL di un'API e aggiungere le informazioni per l'autorizzazione mediante un Query Paramater o un Header Parameter (comunemente un header "Authorization": "Bearer some-very-secure-key"). Ogni collezione esterna ha poi cinque endpoint configurabili, che in sostanza permettono di definire operazioni CRUD mediante chiamate REST all'API definita nel Base URL. Gli endpoints configurabili sono quindi:

1. Get All Records
2. Get One Record
3. Create a Record
4. Update a Record.
5. Delete a Record.

Ogni endpoint ha un metodo e un URL specifico. Il metodo è impostato automaticamente da Adalo, mentre l'URL definisce l'endpoint di una specifica API sul server, a cui possono essere aggiunti dei parametri di filtraggio (query parameters). Adalo consente infine di specificare un nome per la *Result Key* di un'API che restituisce un array. Ad esempio "orders" per un'API che ritorna una Top Level Key "orders". Definiti gli endpoints, si testa l'integrazione con l'API e, in caso di successo, viene mostrata la risposta di esempio ricevuta da questa. Le proprietà della collezione (della nuova classe) vengono automaticamente recuperate e aggiunte, in base alla risposta di esempio dell'API integrata. A questo punto, è possibile utilizzare questa collezione come fosse una collezione definita direttamente sul database interno di Adalo. In questo modo, come vedremo nel capitolo sul design, si sono potuti integrare in app i dati sugli ordini creati in Shopify, permettendo ai negozianti di cambiare il loro stato e gestirne il ciclo di vita.

3.2 Architettura

In questo paragrafo mostriamo lo schema architetturale del progetto, descrivendo descrivendo l'origine di alcune scelte metodologiche e le tecnologie utilizzate per ogni servizio e componente.

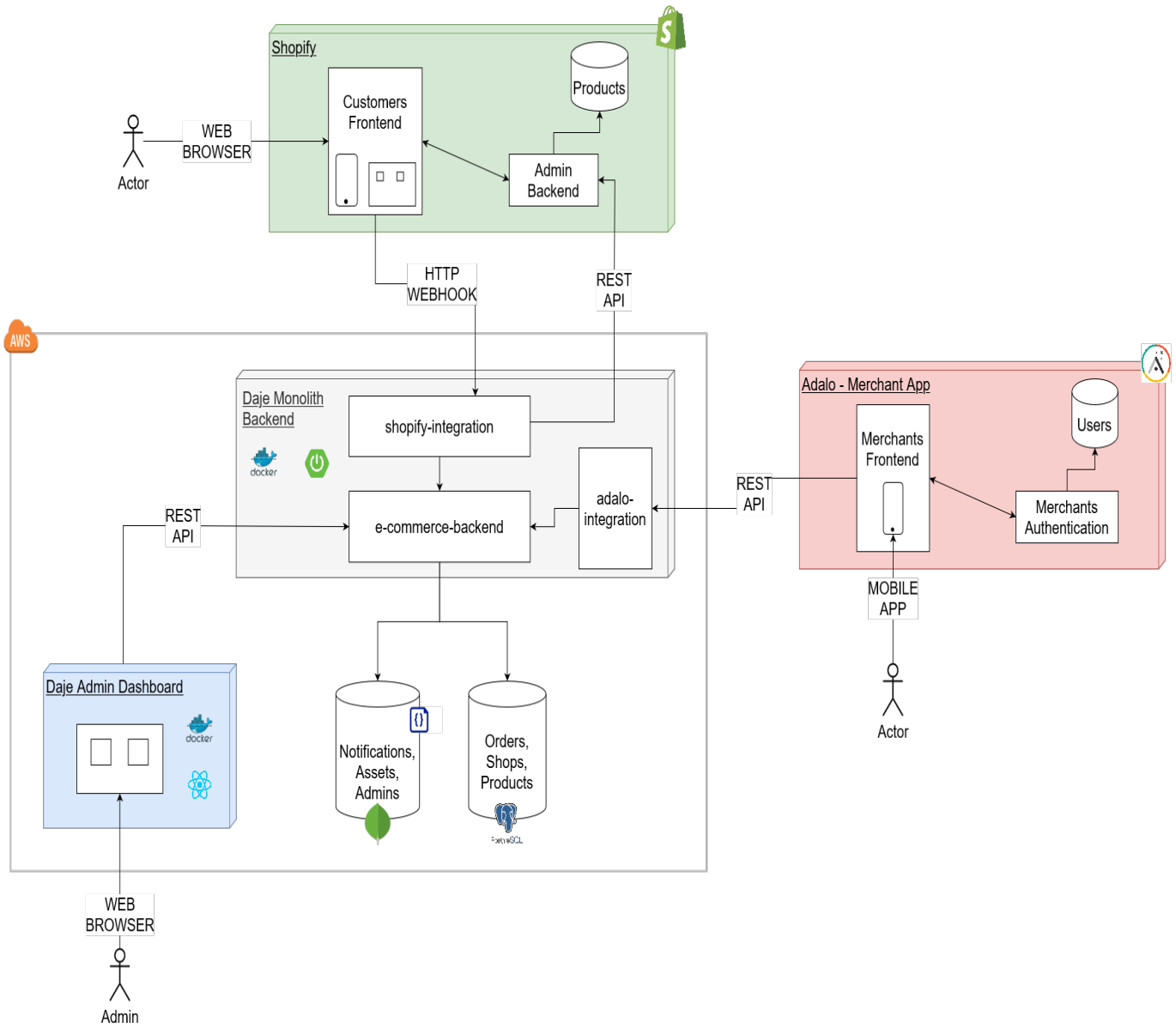

Figura 3.2: Architettura e tecnologie

Nei paragrafi precedenti abbiamo visto come si è potuto realizzare il frontend per i clienti e per i negozianti. Per integrare questi due servizi, è necessario connettere le *External Collections* in Adalo con l'Admin API di Shopify. Ancora una volta però, le strutture dati presenti in Shopify si dimostrano inadatte allo scopo. Sebbene sia possibile connettere direttamente una collezione esterna con la *Order API* di

Shopify, una simile integrazione non consentirebbe l'implementazione dei nostri casi d'uso. In particolare, le collezioni esterne in Adalo hanno bisogno di essere definite in modo quanto più aderente possibile al modello di dominio, in quanto non è possibile operare su di esse complesse manipolazioni, ma solo utilizzarle come oggetti per l'accesso ai dati. Chiariamo meglio questo punto, mostrando la relazione che c'è in Shopify tra un *Order* e le sue *lineItems* con la figura 3.3.

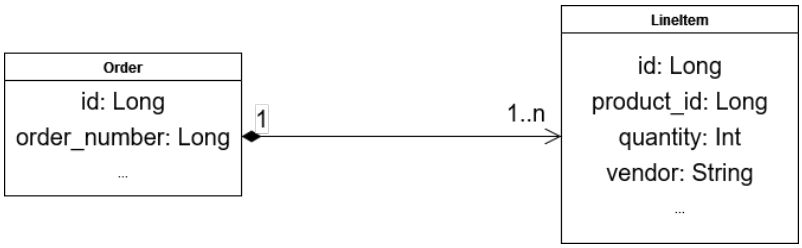

Figura 3.3: Relazione tra Order e LineItem in Shopify

Poichè in Shopify esiste logicamente un solo negozio, e il cliente riempie un carrello con i prodotti in questo, non è possibile distinguere nativamente uno *ShopOrder* da queste strutture, al fine di integrarlo in Adalo come collezione esterna. Occore cosi un servizio, che si frapponga alle due API, in grado di estrarre gli ordini per negorziante da un ordine Shopify. Questo è il servizio principale di backend sviluppato nell'ambito del progetto. Al fine di soddisfare anche i requisiti per gli amministratori - di cui in questo testo ne abbiamo dettagliato uno solo - si è scelto di implementare una dashboard web che permettesse di accedere ai dati sugli ordini in maniera maggiormente aderente all'operatività del business, rispetto a quella già fornita in Shopify. Per questo motivo, il servizio di backend non si limita ad operare il *mapping* tra i dati in Shopify e quelli del nostro dominio, ma persiste questi ultimi in una base di dati relazionale, interrogata allo scopo da questa dashboard. Dobbiamo poi considerare l'invio delle notifiche. Per questo caso d'uso, si sarebbe potuto implementare un servizio di backend diverso da quello dedicato agli ordini e in generale alle funzioni dell'e-commerce. Nello stile architetturale dei microservizi, avremmo potuto orchestrare questi due in modo che cooperassero alla realizzazione di tutti i requisiti discussi. Nell'ambito di questo progetto però, si è scelto di non implementare subito la soluzione "a microservizi" considerando maggiormente vantaggiosa una riduzione dello sforzo di sviluppo infrastrutturale e accettando il trade-off sull'accoppiamento delle funzionalità. Si è scelto dunque di implementare un monolite, facendo attenzione a che questo potesse essere progettato con il massimo della coesione, in modo che fosse possibile in un secondo momento estrarre da esso N microservizi. Mostreremo il design dell'applicazione di backend nel capitolo successivo. La figura 3.1 riassume l'interazione dei servizi implementati per la realizzazione del marketplace e le tecnologie utilizzate. Si è scelto di utilizzare un database documentale (NoSQL) per la persistenza delle informazioni inerenti le notifiche. Questa scelta è motivata da un lato dalla netta separazione delle due aree, backoffice e e-commerce, dall'altra, dalla natura *schemaless* di queste informazioni.

Si è rivelato molto utile non avere uno schema fisso per questi dati, in ogni momento è stato possibile aggiungere o togliere informazioni dalla definizione della struttura che definisce le notifiche, senza minare l'integrità della base di dati o rallentare lo sviluppo. La base di dati documentale poi, contiene informazioni utili ad altri casi d'uso, qui non trattati, inerenti all'integrazione dei cataloghi dei prodotti in Shopify (FrontstoreAssets). Inoltre, eventuali evolutive verso funzionalità real-time sono di più facile realizzazione con questo tipo di tecnologia [6].

Questo progetto è essenzialmente un progetto di integrazione di servizi diversi. Lo stesso backend sarà certamente frammentato in microservizi operanti in cloud e per questo motivo, lo sviluppo delle componenti proprietarie ha seguito un'approccio API First [17]. Questo approccio consiste essenzialmente nello sviluppare l'API (in particolare una REST API) relativa ad uno scenario di caso d'uso prima della sua effettiva implementazione. Per la definizione delle API è stato fatto uso della specifica OpenApi (3.0.0) e dello Swagger Editor [16]. Questo approccio ha permesso di accelerare notevolmente lo sviluppo, garantendo la possibilità - mediante il plugin Gradle OpenAPI Generator - di generare automaticamente il codice infrastrutturale di ogni singola API [10].

Infine, lo stack tecnologico del backend è composto dalle seguenti tecnologie:

1. Linguaggio: Java (11)
2. Frameworks: Spring Boot
3. Build tools: Gradle
4. Build plugins: OpenAPI-Generator, Sonar
5. Virtualization tools: Docker
6. ORMs: Spring Data Mongo, Spring Data JPA
7. Libraries: Mapstruct, Lombok
8. DBMS: MongoDB, PostgreSQL
9. Deploy (Cloud): Amazon Web Services (EC2, RDS)

Mentre la dashboard di backoffice è stata implementata con l'uso della libreria React.

Capitolo 4

Design delle soluzioni

4.1 L'E-commerce: Marketplace e Backoffice

Prima di addentrarci nella discussione sul progetto dei singoli casi d'uso, mostriamo con la figura 4.1 una panoramica delle componenti d'interesse dell'architettura logica del backend, in cui si evidenzia la separazione delle responsabilità nell'ottica di una migrazione del progetto a microservizi. Al fine di disaccoppiare la logica delle API di business da quella relativa all'integrazione con le soluzioni no-code scelte, abbiamo implementato queste ultime in modo che comunicassero mediante i DTOs (*Data Transfer Object*) che le altre consumano in input, seguendo il pattern Facade. In questo modo, le API e la relativa logica di business possono evolvere in autonomia ed essere riutilizzate facilmente da client diversi da Shopify e da Adalo (ne è un esempio la dashboard di backoffice per gli admin).

Con la figura 4.1 vogliamo riassumere l'approccio generale, mostrando la relazione fondamentale che intercorre tra le classi dei package integration ed e-commerce-backend: i servizi di integrazione consumano le API di business.

Le classi Shopify-Entity-Controller e Adalo-Entity-Controller hanno la responsabilità di definire gli endpoint per la comunicazione con i servizi terzi. In particolare ci saranno N singleton di questo tipo, parametrici con Entity (ad es AdaloOrderController, ShopifyOrderController). Queste delegano alla classi Entity-IntegrationService la logica d'integrazione, cioè lo scambio di messaggi con le API di business e l'effettivo mapping delle entità. Quest'ultima responsabilità è delegata dagli IntegrationService agli Entity-Mapper, che sanno come trasformare un DTO dal servizio al corrispondente DTO di dominio. La presenza degli IntegrationService si rende necessaria per aumentare la coesione. Questi infatti si occupano della comunicazione con le api del dominio, una responsabilità già molto ampia, che si è deciso di separare da quella del mapping uno a uno con i DTO di dominio, delegata appunto agli Entity-Mappers.

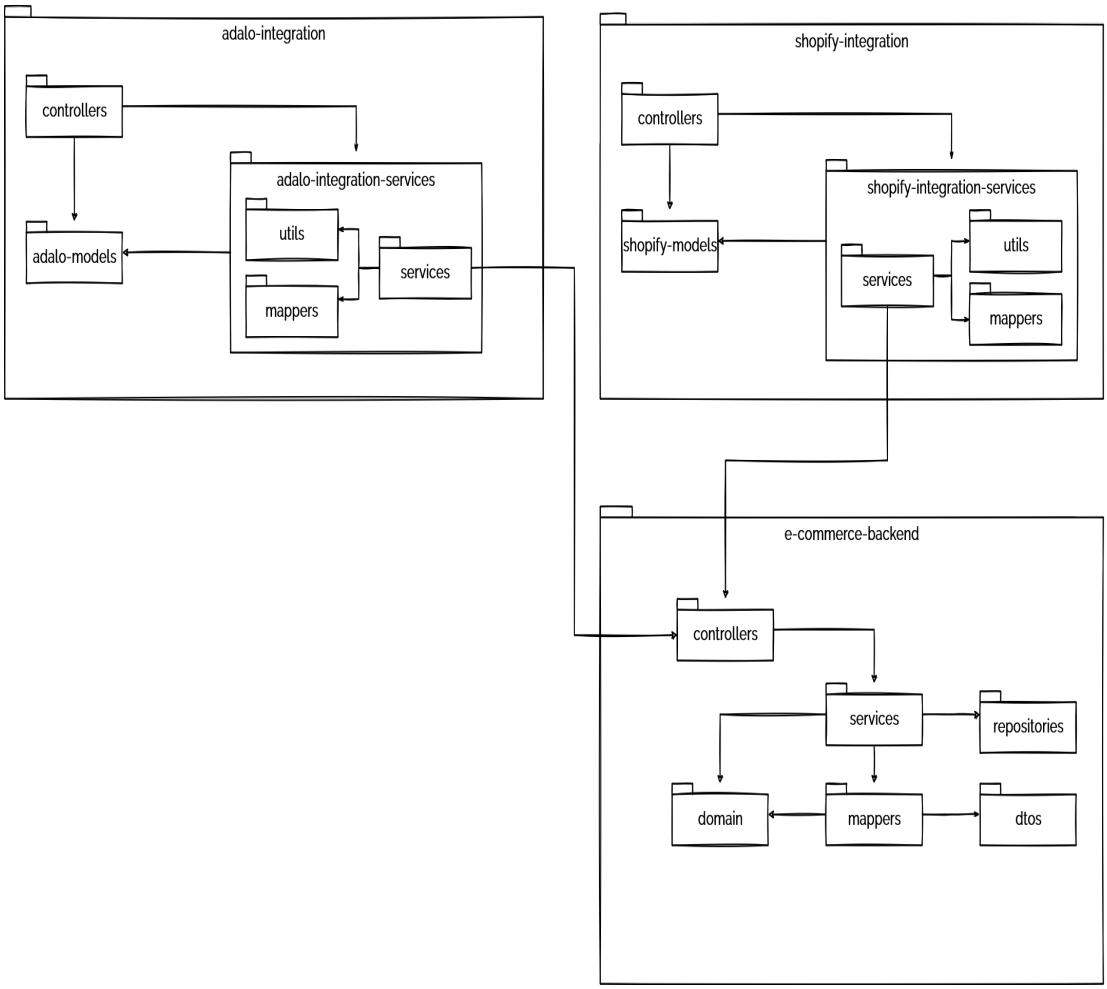

Figura 4.1: Packages

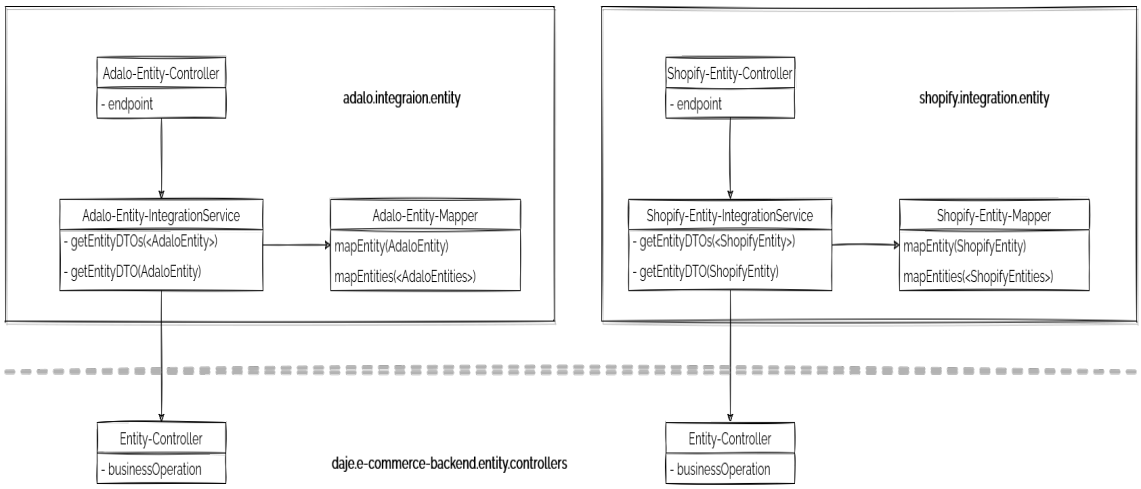

Figura 4.2: Approccio per l'integrazione

4.1.1 Il servizio di backend per la persistenza degli ordini

Lo scopo di questo servizio è quello di portare fuori da Shopify i dati sugli ordini e persisterli in un forma aderente al modello di dominio, al fine di servirli ad altre componenti per la realizzazione di tutti i casi d'uso discussi. A questo scopo, come mostra la figura 3.3, è stato possibile definire un *webhook* in Shopify grazie al quale ad ogni ordine sottomesso, viene istanziata una chiamata HTTP ad un endpoint della classe *ShopifyOrderController* nel package *shopify-integration.controllers*. Poichè questo endpoint è esclusivamente per il webhook di Shopify, non avrebbe senso definire un'API mediante una specifica Open Api.

Le figure 4.2, 4.3, 4.5 e 4.6 mostrano invece il diagramma di sequenza di questa operazione, diviso nelle parti fondamentali per semplicità. Per semplificare ulteriormente gli schemi, abbiamo evitato di mostrare chiamate ai metodi privati, getters e setters, laddove questi non aggiungessero valore. La prima, mostra il mapping dei dati relativi agli *ShopOrder*, mentre le altre due rappresentano il flusso di estrazione dei dati di un *Order*: in particolare del *Customer*, del *Payment* e dello *Shipping*.

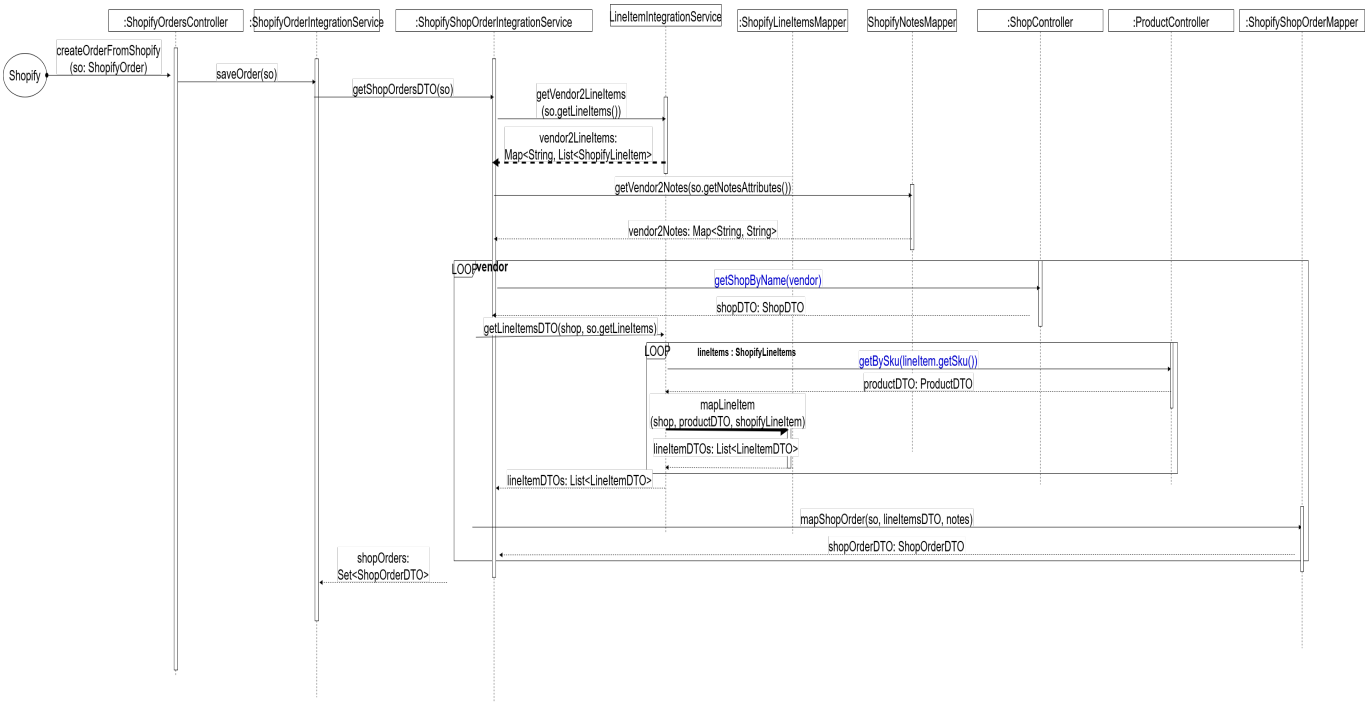

Figura 4.3: Diagramma di sequenza per la persistenza di un ordine da Shopify - Ordini

Nella figura 4.3, vediamo in azione la collaborazione tra le classi descritte in precedenza. In particolare, notiamo come la logica di estrazione dei dati sia delegata agli IntegrationService. I metodi *getVendor2LineItems* e *getVendor2Notes* si rendono necessari per estrarre rispettivamente, una mappa String -> List<LineItem> e una

String -> Notes, dove la key è il name di uno *Shop*. Questa necessità è dovuta alla struttura dell'oggetto *ShopifyOrder* che l'operazione prende in ingresso. Come abbiamo visto infatti, non è possibile da questo estrarre direttamente gli *ShopOrder*. Allo scopo, dopo aver recuperato queste mappe, si cicla sui valori nell'entryset (sui "vendor") per recuperare gli oggetti *ShopDTO*. All'interno di questo ciclo poi, è necessario costruire tutte le *LineItemDTO* a partire dallo Shop e dai dati ricevuti da Shopify (di tipo *ShopifyLineItems*). La costruzione di questi oggetti viene poi delegata ai vari Mappers.
Il metodo *getShopByName(vendor)* non è mostrato nella sua interezza. Questo metodo definito in *ShopController* rappresenta un endpoint di business, un'API cioè di cui esiste la specifica OpenApi e che segue il pattern consolidato Controller-Service-Repository. Il controller, delega cosi allo *ShopService* la responsabilità di recuperare uno *Shop* che abbia nome "vendor" e di gestire eventuali errori. Lo stesso vale per gli altri metodi in blu, in questa e nelle prossime figure.

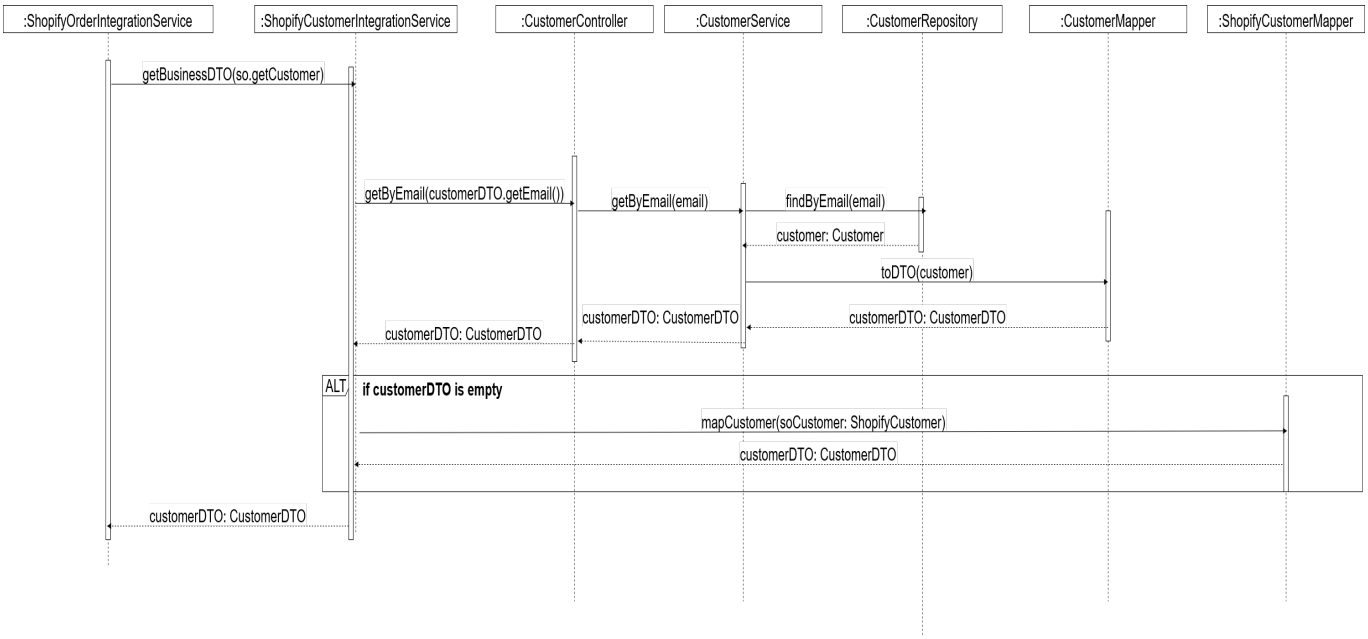

Figura 4.4: Diagramma di sequenza per la persistenza di un ordine da Shopify - Customer

La figura 4.4 mostra il proseguimento dell'operazione. Per costruire un *OrderDTO* infatti, abbiamo bisogno anche degli oggetti *ShippingDTO*, *CustomerDTO*. In questo caso c'è bisogno di controllare che il Customer non esista già nel DB. Questo controllo è rappresentato dal flusso che parte con il metodo *getByEmail(shopifyCustomer.getEmai())*. Se il cliente ha già acquistato in passato, allora il corrispettivo DTO verrà restituito ed associato al nuovo Order, altrimenti, attraverso il *ShopifyCustomerMapper*, un nuovo DTO viene creato (senza id) e restituito allo scopo.

La figura 4.5 mostra il diagramma di sequenza per l'estrazione degli oggetti Ship-

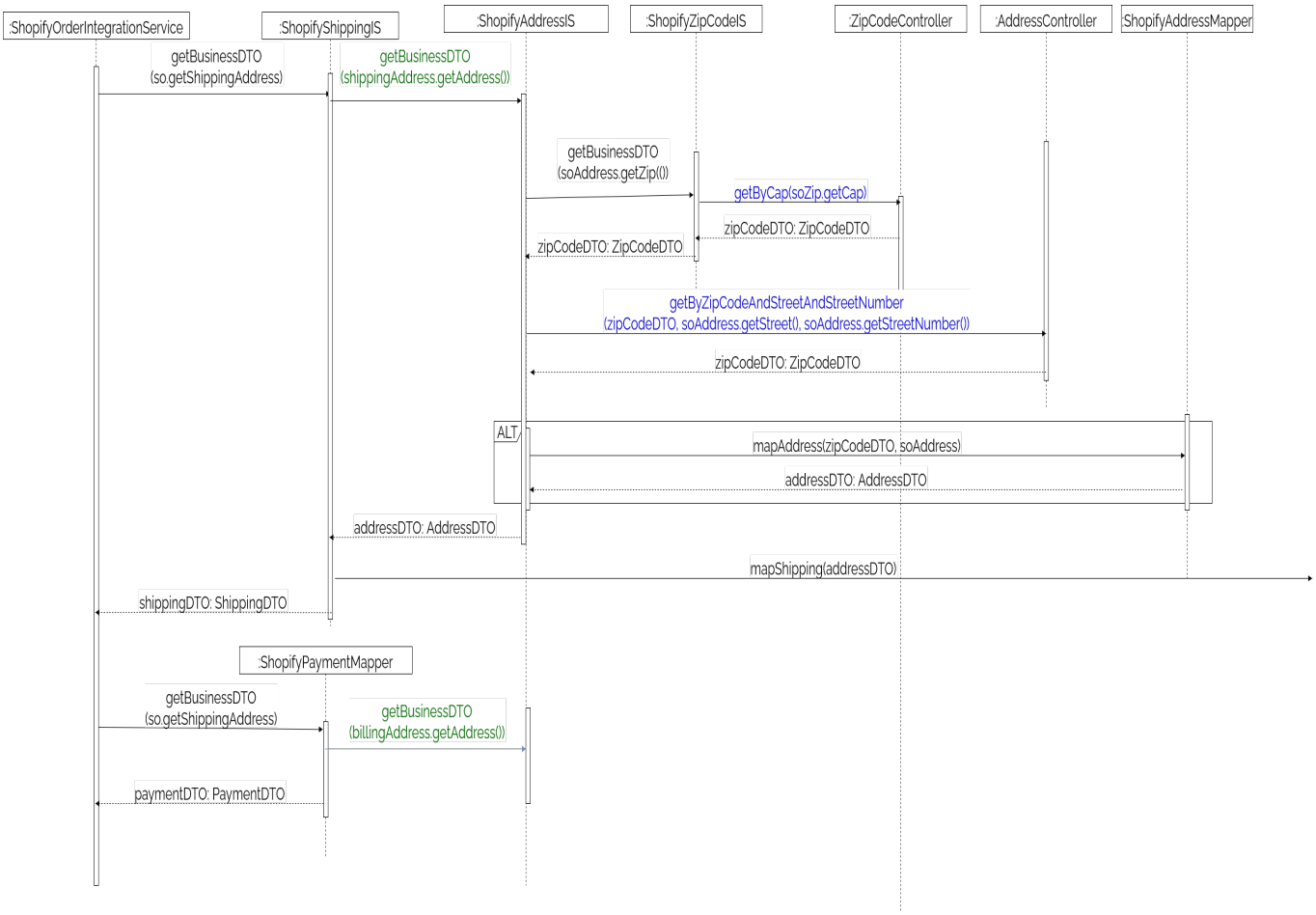

Figura 4.5: Diagramma di sequenza per la persistenza di un ordine da Shopify - Shipping e Payment

pingDTO e PaymentDTO, necessari per completare le informazioni di un OrderDTO. Questi due oggetti hanno entrambi bisogno di un *AddressDTO* per essere mappati. L'oggetto che contiene le informazioni sull'indirizzo (rispettivamente di spedizione e di fatturazione) è recuperato mediante l'interazione di *ShopifyAddressIntegrationService* con *AddressController*. Il metodo in verde *getBusinessDTO(sa : ShopifyAddress)* è mostrato nella sua interezza soltanto per il flusso di estrazione dello ShippingDTO, ma si ripete in maniera perfettamente analoga anche per il PaymentDTO. La figura 4.6 mostra gli oggetti ritornati all'IntegrationService e il mapping dell'ordine in un OrderDTO. A questo punto, sarà sufficiente passare il DTO al metodo *createOrder(orderDTO: OrderDTO)* di *OrderController* per persistere le informazioni relative al nuovo ordine. Questa operazione avrà dunque la responsabilità di persistere i dati, ora aderenti al modello di dominio e alle strutture definite nel package e-commerce-backend. Mediante opportune policy di CASCADE, si è fatto in modo di persistere i dati in un'unica transazione, al momento di persistere un'istanza di Order, vengono persistite anche tutte le relazioni in esso contenute (ShopOrders, LineItems, Customer, etc.).

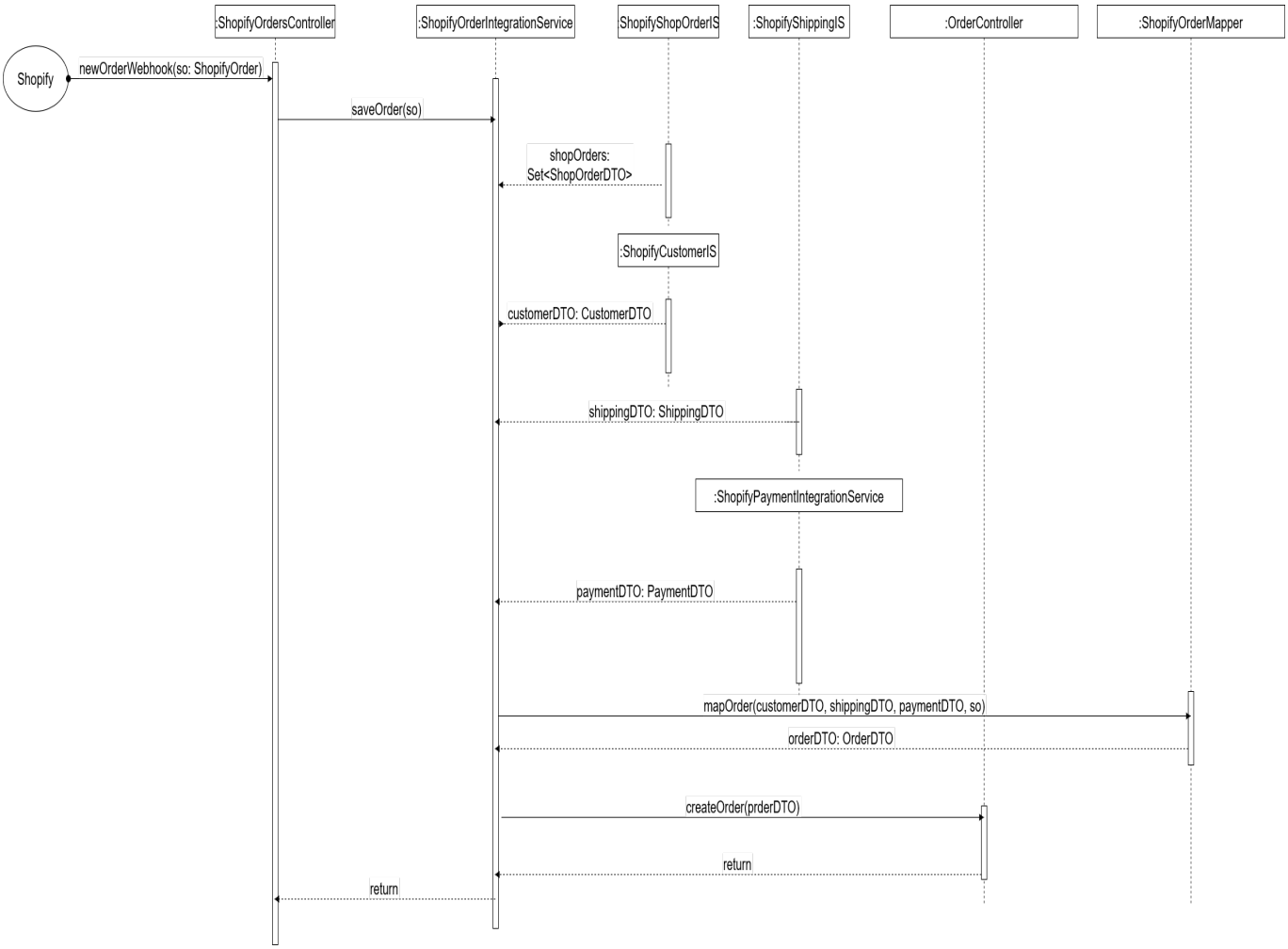

Figura 4.6: Diagramma di sequenza per la persistenza di un ordine da Shopify - oggetti restituiti

4.1.2 Il servizio di notifica per i clienti

Lo scopo di questo servizio è essenzialmente servire gli amministratori con le notifiche da inviare ai clienti. Come abbiamo visto, un amministratore deve poter accedere ad una dashboard e, una volta scelto un ordine da notificare, può aprire una chat Whatsapp in cui è pre-compilato il messaggio di notifica e decidere di inviarlo. Si è scelto di non inviare le notifiche automaticamente per non dover sopperire ad ulteriori costi di registrazione a servizi terzi, con l'intenzione di aggiungere questa funzionalità in un secondo momento. Un esempio di servizi di questo tipo è dato da Twilio (SaaS), che espone un'API con cui è possibile inviare messaggi Whatsapp a pagamento. Mostriamo la definizione dell'API con cui è possibile recuperare una notifica.

```
1 get:
2     tags:
3     - notifications
```

```yaml
summary: returns the notification for the specified shopOrderId
operationId: getNotification
parameters:
    - in: path
    name: shopOrderId
    description: A ShopOrder unique identifier
    required: true
    schema:
        type: string
responses:
    '200':
        description: notification retrieved.
        content:
            application/json:
                schema:
                \$ref: '#/components/schemas/NotificationDTO'
    '400':
        description: Notification could not be found.
    '404':
        description: Bad request.
    '500':
        description: Something wen wrong retrieving the notification.
```

Come vedremo nella sezione successiva, esiste un'unica notifica per ordine, che viene creata ogni volta che l'ordine per negoziante cambia il suo stato in confermato o declinato. La figura 4.7 mostra invece lo schema dell'oggetto centrale del caso d'uso, presente nel package *backoffice.customers.notifications* del progetto di backend. Poi-

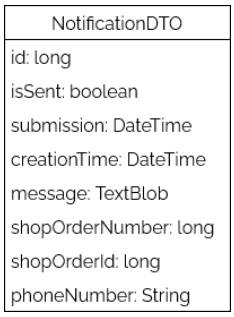

Figura 4.7: Oggetto NotificationDTO

chè l'invio effettivo del messaggio avviene su Whatsapp, è stato necessario definire un evento nella UI della dashboard e la rispettiva API per la modifica degli attributi *isSent* e *submissionDateTime*. In questo modo, è stato possibile mostrare nella dashboard quali ordini sono già stati notificati. La definizione OpenApi di questa operazione è riportata nel frame di pagina seguente.

```yaml
put:
    tags:
    - notifications
summary: update the notification status and submission
operationId: updateNotificationStatus
parameters:
    - in: path
    name: notificationId
    description: A Notification unique identifier
    required: true
    schema:
        type: string
    - in: path
    name: isSent
    description: boolean that tells if a notification has been sent
    required: true
    schema:
        type: string
responses:
    '200':
        description: notification updated.
        content:
            application/json:
                schema:
                \$ref: '#/components/schemas/NotificationDTO'
    '400':
        description: Notification could not be found.
    '404':
        description: Bad request.
    '500':
        description: Something went wrong updating the notification.
```

La figura 4.8 mostra invece il diagramma di sequenza per questa operazione, con cui viene aggiornato lo stato dell'istanza di *Notification* mediante il pattern Controller-Service-Repository. Notiamo come oltre al valore di *isSent*, viene aggiornato contemporaneamente il valore di *submissionDate* con un valore che rappresenta l'istante attuale.

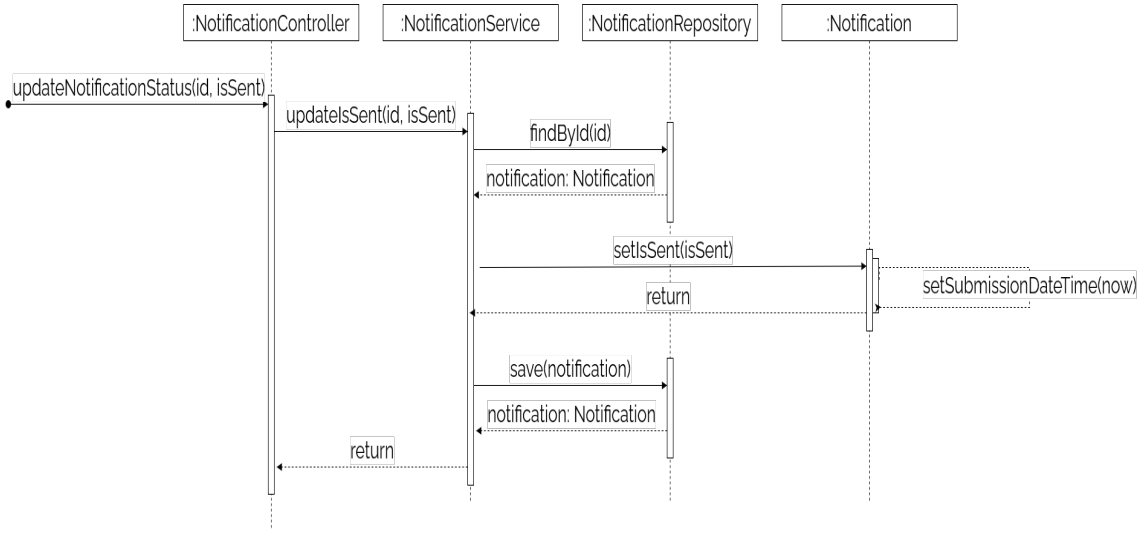

Figura 4.8: Diagramma di sequenza per l'operazione updateNotificationStatus

4.2 L'App Merchant

La figura 4.9 mostra le viste principali implementate in Adalo per l'applicazione in dotazione ai negozianti.

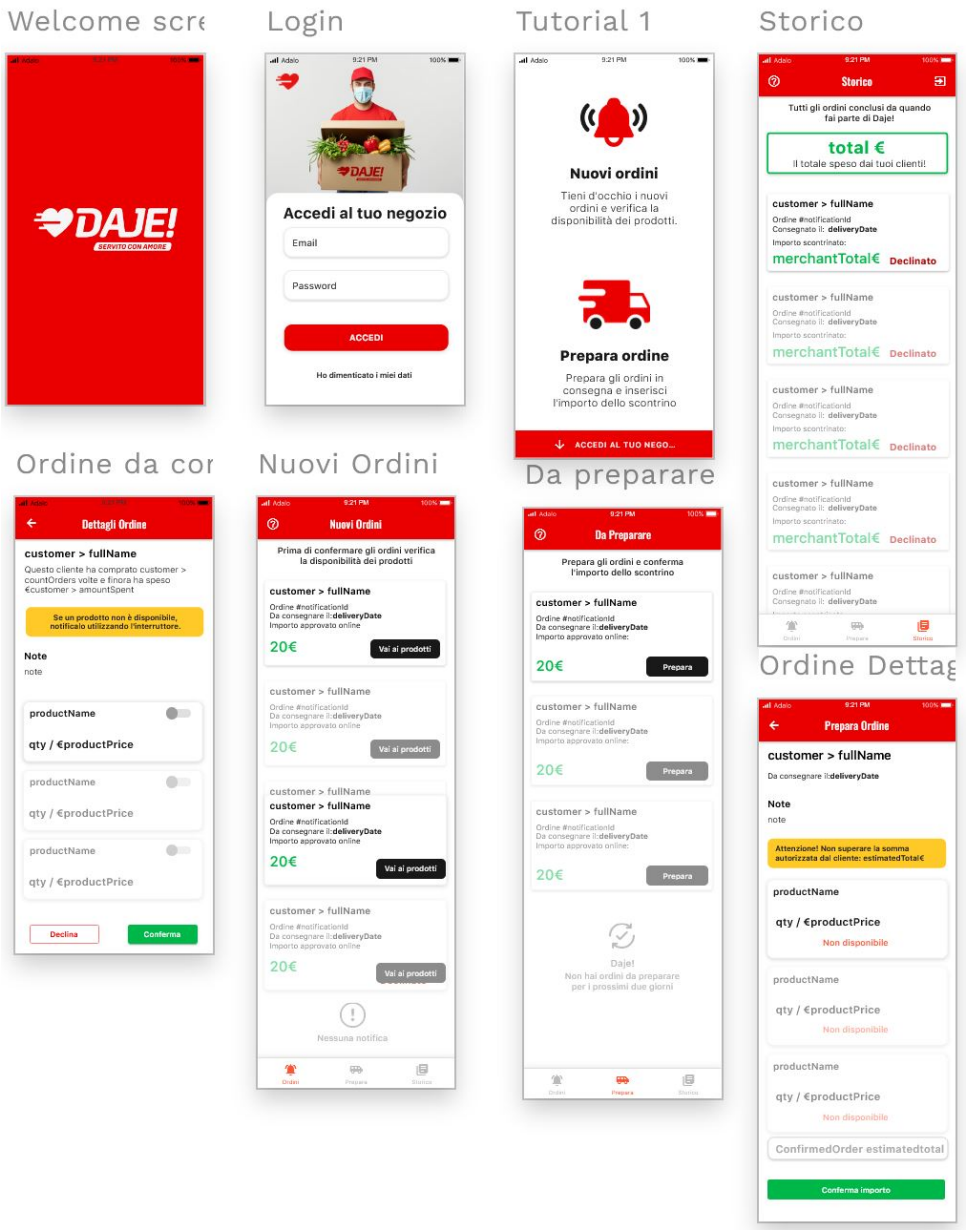

Figura 4.9: AppMerchant: Pagine

Per implementare le viste sugli ordini (Nuovi ordini, Ordine da confermare, Da preparare, Ordine dettaglio e Storico), è stato necessario definire le seguenti collezioni esterne, e connetterle con le relative API del backend in grado di fornirne le istanze valorizzate con i dati degli ordini in Shopify.

1. Order: Da recuperare in base al suo stato e alla data di consegna prevista, per aggiornarne la disponibilità e il prezzo finale.

2. LineItem: Poichè in Adalo una collezione può avere solo attributi di tipo primitivo (Number, Text, Date, Boolean), è necessario recuperare le righe d'ordine via API e dunque mediante una collezione esterna diversa.

3. CustomerShopHistoryDetails: Per mostrare all'esercente quanto ha speso il cliente presso il suo negozio fino a quel momento.

4. ShopStats: Per mostrare all'esercente il totale speso presso di lui in Daje!

Le "statistiche" relative al negozio *CustomerShopHistoryDetail* e *ShopStats* hanno rappresentato un elemento di forte gradimento per gli esercenti favorendo una rapida adozione dell'applicativo. La figura 4.10 mostra il diagramma delle classi nel package *adalo-integration.models*. Queste rappresentano nel backend le collezioni esterne definite in Adalo, che avranno dunque gli stessi attributi.

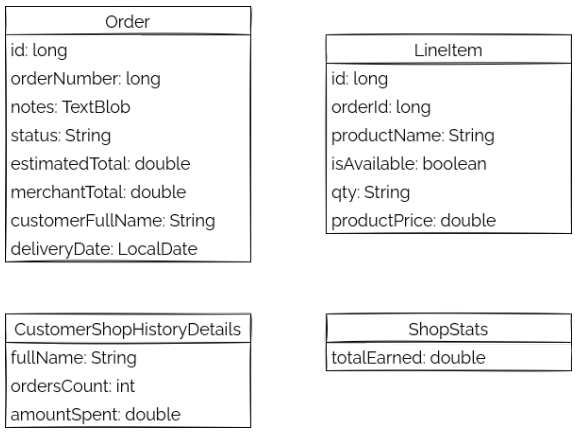

Figura 4.10: AppMerchant: Collezioni esterne

Queste classi sono prive di relazioni fra esse e di metodi, sono essenzialmente DTOs e i loro attributi vengono calcolati opportuamente come vedremo a breve. Si è scelto di non persistere questi dati, ma di adottare opportune strategie di caching per evitare che venissero ricalcolati ogni volta senza necessità. Dobbiamo evidenziare infatti che i dati di una collezione esterna vengono ogni volta recuperati da chiamate agli endpoint, per cui senza una cache il carico sul backend sarebbe notevole. Effettuato il login (che abbiamo delegato completamente ad Adalo) l'esercente naviga sulla pagina degli ordini da confermare. Qui troverà la lista, ordinata per data di consegna, degli ordini di cui deve confermare la disponibilità dei prodotti. Adalo

effettuerà allora una chiamata al backend per recuperare tali ordini con oppurtuni query parameters per popolare la *Custom List* con cui si sono implementate le cards degli ordini. Prima di mostrare il diagramma di sequenza per questa integrazione (figura 4.11) facciamo alcune considerazioni sulla configurazione delle collezioni esterne in Adalo. Come abbiamo visto, per configurare una collezione esterna occorre "connettersi" con un test all'endpoint definito. Tuttavia, in fase di configurazione non è possibile inserire query parameters per la chiamata HTTP (saranno *null* sull'endpoint del Controller). Nel caso degli Order, ad esempio, l'endpoint recupera gli ordini per *email*, cioè in definitiva per negoziante. Si riesce comunque a distinguere quando una chiamata senza questo parametro è di configurazione e quando invece è malformata, il parametro *email* è infatti mandatorio per un'istanza di User nel DB interno di Adalo. Pertanto, in nessun caso l'assenza della email può essere inputato ad una richiesta malformata. Al fine di gestire le chiamate di configurazione, sebbene non lo mostriamo nei diagrammi che seguono, le classi di tipo AdaloController collaborano con classi di utils che hanno il compito di restituire un oggetto di default in caso di chiamata di configurazione.

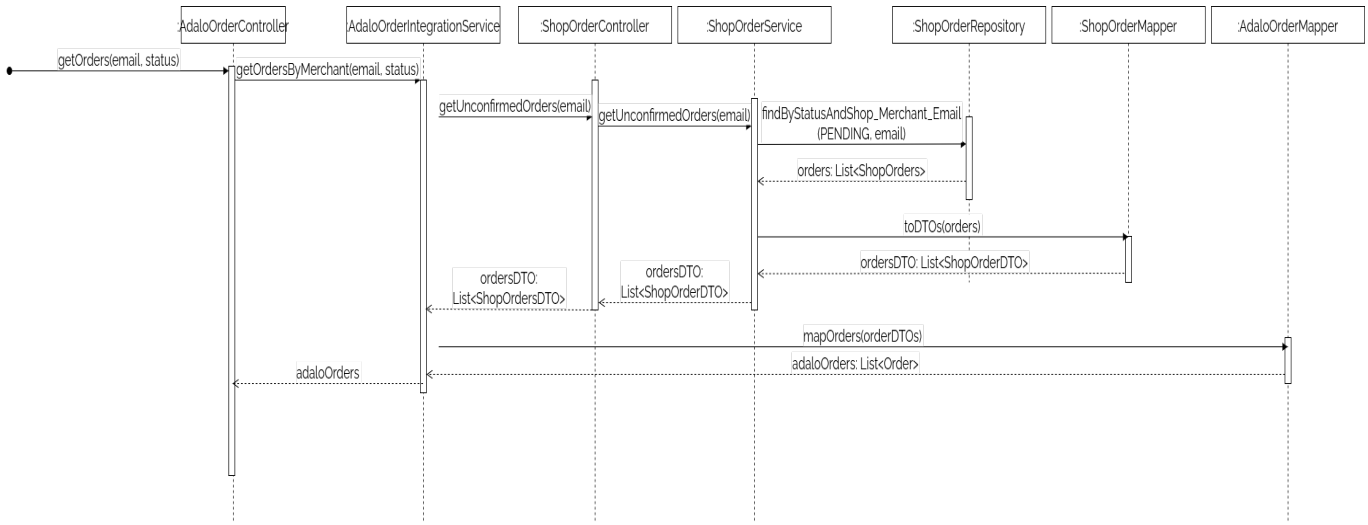

Figura 4.11: AppMerchant: Recupero ordini da confermare

Recuperati gli ordini, l'esercente procederà scegliendo un ordine da confermare, navigando nella pagina relativa. Qui, Adalo dovrà recuperarne le righe d'ordine mediante l'endpoint, e mostrarle ancora tramite una Custom List. La figura 4.12 mostra il diagramma di sequenza di questa integrazione.

Poichè in Adalo è possibile passare le istanze di una sola collezione ad una vista successiva, e il requisito era qui di mostrare una modale di conferma all'esercente, prima di modificare lo stato dell'ordine attivando la relativa action (e istanziando quindi una HTTP PUT per confermare l'ordine), è stato necessario fare una chiamata HTTP PUT per ogni riga d'ordine per la quale lo stato dell'attributo *isAvailable*

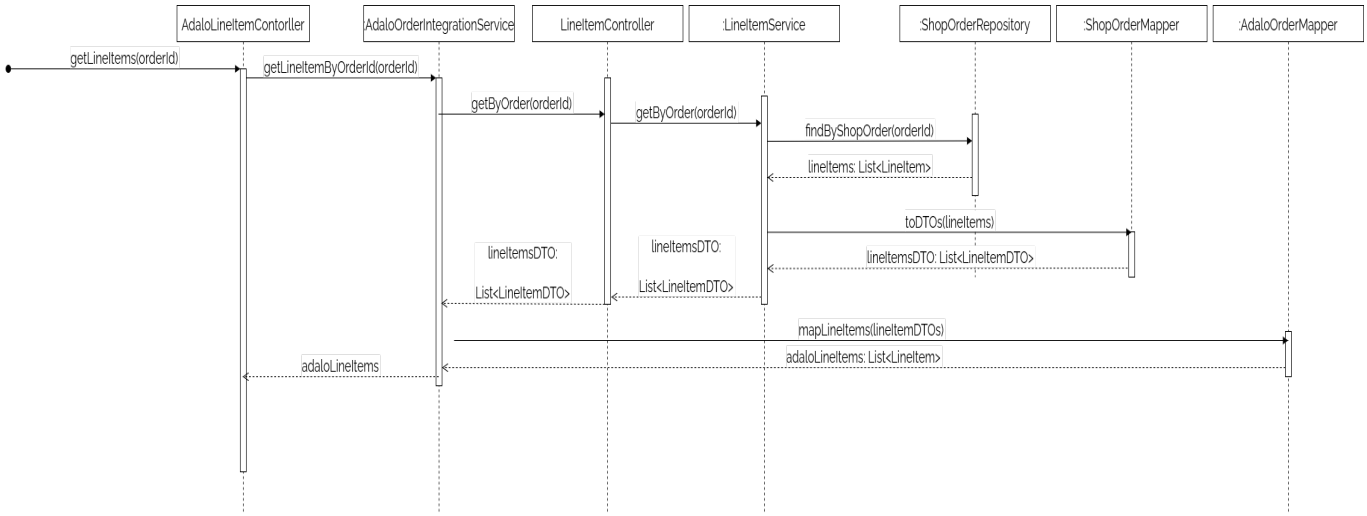

Figura 4.12: AppMerchant: Recupero righe d'ordine

è cambiato mediante lo switch in figura 4.11 (Ordine da confermare). Non è stato possibile dunque aggiornare mediante una sola chiamata HTTP lo stato dell'ordine e quello delle rispettive righe tutto in una volta, in quanto come detto, questi dati risiedono in collezioni diverse. La figura 4.13 mostra l'operazione per l'aggiornamento della disponibilità di una riga d'ordine, mentre la figura 4.14 quella per la sua conferma, cioè del suo cambiamento di stato.

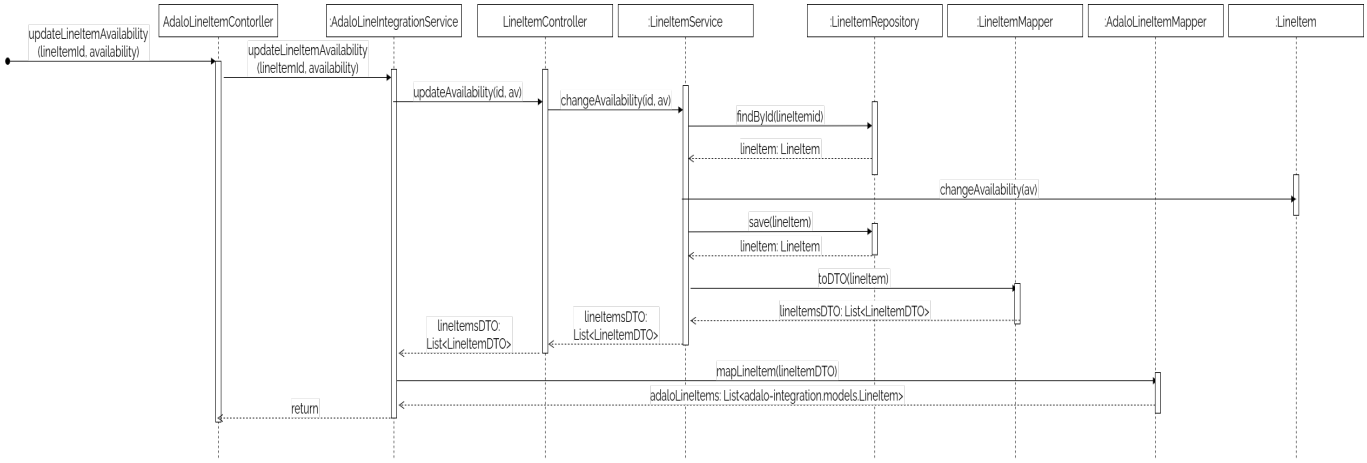

Figura 4.13: AppMerchant: Modifica disponibilità riga d'ordine

L'operazione è la stessa anche per il caso d'uso "declina", basterà infatti cambiare il parametro della query inerente lo stato in cui si vuole portare l'ordine. Il metodo in verde *createNotification(order : ShopOrder)* ha la responsabilità di creare e persistere una nuova istanza di *Notification* a partire dall'ordine e dal suo stato.

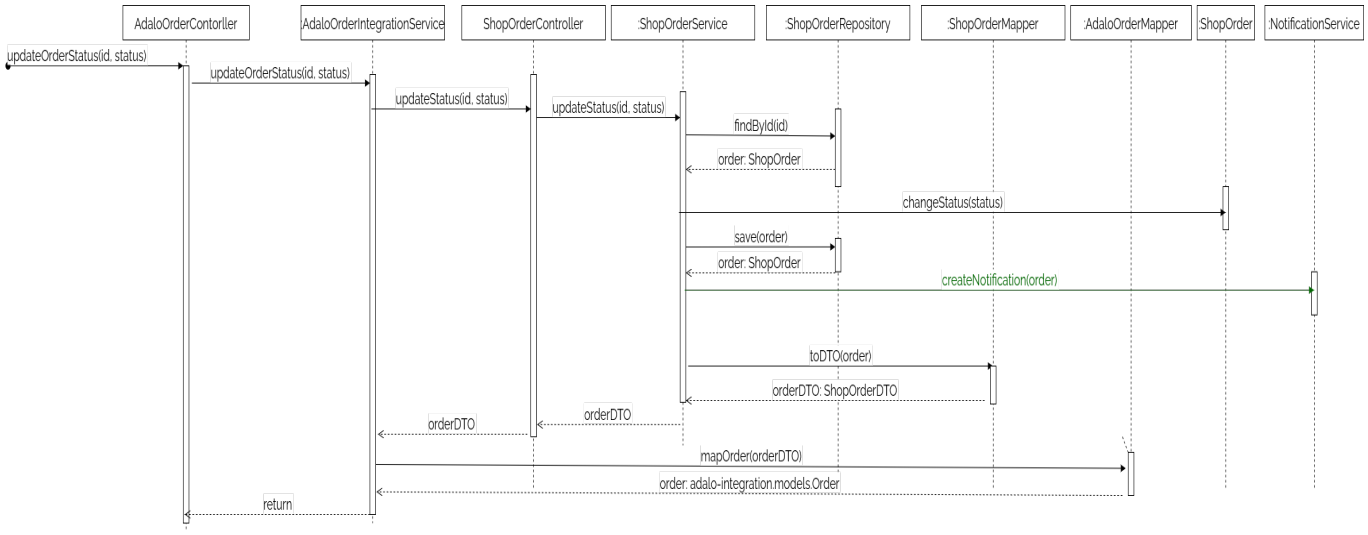

Figura 4.14: AppMerchant: Modifica dello stato dell'ordine

Nella vista degli ordini da preparare - cioè degli ordini confermati con data di consegna a due giorni - l'esercente, dopo aver scelto un ordine, può inserire il totale scontrinato e cambiare ulteriormente lo stato dell'ordine in *READY* cioè pronto per la consegna. In questo caso, poichè il campo *merchantTotal* fa parte di *Order* ed è di tipo double, è necessaria una sola chiamata HTTP PUT.

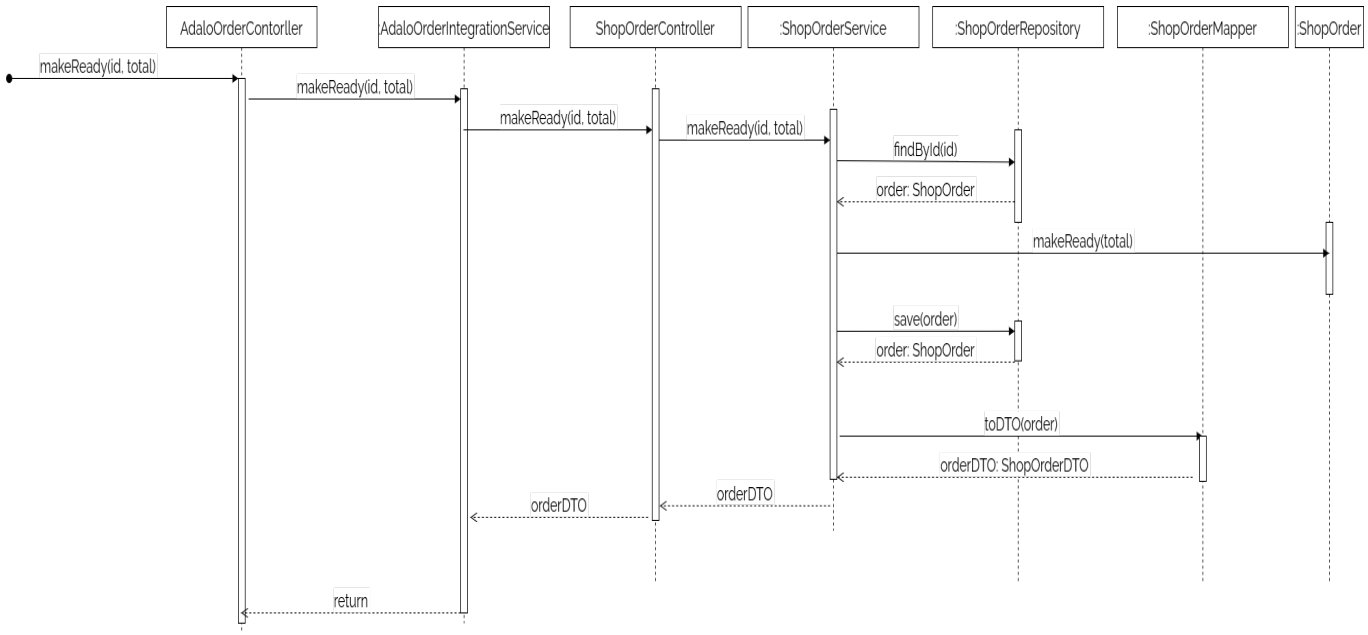

Figura 4.15: AppMerchant: Preparazione di un ordine

Capitolo 5

Risultati

In questo capitolo conclusivo, vogliamo illustrare i risultati raggiunti mediante questo progetto. Elenchiamo in maniera sintetica quali essi siano.

1. Minimizzazione delle ore lavoro per la gestione del ciclo di vita degli ordini.
2. Minimizzazione degli errori nella gestione del ciclo di vita degli ordini.
3. Miglioramento dell'esperienza dei partners e della loro percezione del brand.
4. Implementazione di assets fondamentali.

Per ciò che concerne il primo risultato, l'applicazione in dotazione ai negozianti ha consentito di risparmiare moltissime risorse in termini di attività umane nella gestione dell'operatività del servizio. L'implementazione della gestione del ciclo di vita degli ordini, senza tecnologia proprietaria, è stato a lungo il collo di bottiglia per le attività legate all'espansione del business. Con più di cento esercenti all'attivo sulla piattaforma, la gestione degli ordini richiedeva un eccessivo sforzo umano a poco valore aggiunto. L'insieme degli asset tecnologici con cui Daje! ha raggiunto l'operatività agli inizi è risultato insufficiente a soddisfare le richieste dopo l'apertura massiva sui diversi quartieri di Roma. Per dare un'idea delle risorse necessarie per questa attività, dobbiamo prima mostrare la tecnologia e i processi che permettevano al servizio di essere operativo. Diamo una panoramica della situazione precedente a questi sviluppi, mostrando la limitatezza del contesto tecnologico pre-esistente in termini di scalabilità e sviluppi futuri. La parte alta della figura 5.1 riassume il processo di integrazione di servizi terzi e l'azione umana coinvolta nell'intero processo. In questa fase, quando un cliente sottomette un ordine, Shopify si occupa di registrarlo nel suo database interno. Attraverso *Integromat*, una piattaforma per l'integrazione di servizi [3], l'ordine viene dapprima estratto dal database di Shopify e poi persistito in un semplice Google Sheet. Attraverso un altro *job* Integromat l'ordine viene spacchettato nei diversi ordini-per-negoziante e persistito su un ulteriore sheet. Da qui, con i dati dell'ordine, un amministratore può comporre e inviare una notifica ai vari esercenti. Questi, con il processo descritto nei capitoli precedenti confermano l'ordine, inviando un'ulteriore notifica verso gli admin, che a loro volta dovranno comporre un nuovo messaggio di conferma da inoltrare ai clienti. Quando

l'ordine è pronto per la consegna, l'esercente notifica nuovamente l'amministratore, il quale può inserirlo nel giro di consegne previste. Tutte le notifiche avvengono su Whatsapp - una scelta basata sullo studio dei clienti tipo.

Per sostenere questo processo, almeno una persona nelle operations doveva occuparsi di gestire le notifiche per un periodo che comprendesse gli orari d'apertura degli esercenti, in genere circa otto ore. Con l'introduzione degli asset di cui abbiamo mostrato il progetto, non solo questo tempo si è drasticamente ridotto a circa un'ora in uno slot prefissato (abbiamo stimato una riduzione dell'impiego umano di circa il 90%), ma si sono dotati gli amministratori di uno strumento di facile utilizzo, in grado di minimizzare gli errori che inevitabilmente si sono presentati nell'attività, come mancate comunicazioni, comunicazioni errate e/o destinatari errati, sempre più comuni con l'aumentare degli esercenti attivi. Con l'utilizzo della dashboard con cui scegliere gli ordini e notificare i clienti, la percentuale per questo tipo di errori si è essenzialmente azzerata.

Dal punto di vista degli esercenti poi, abbiamo condotto una semplice verifica della bontà dell'esperienza fornita interrogando direttamente gli utilizzatori. La quasi totalità di essi ha dichiarato di preferire l'uso dell'applicazione rispetto all'uso della messaggistica istantanea. Sebbene Whatsapp fosse inizialmente preferito - per ovvi motivi di abitudine e familiarità - dopo circa un mese di utilizzo hanno riscontrato sostanziali miglioramenti nel processare gli ordini di Daje!. I principali vantaggi che hanno individuato sono infatti: una maggiore aderenza al proprio workflow offline, con conseguente diminuzione degli impedimenti durante l'orario di servizio e la possibilità di prendere visione delle statistiche del negozio sulla piattaforma, senza contatto diretto con l'amministrazione. In luogo di ricevere notifiche *push*, l'applicazione ha consentito loro di organizzare in autonomia la conferma e la preparazione degli ordini. Questo ha poi migliorato la percezione del brand presso i partners, consolidandone l'immagine di un'opportunità concreta di crescita, senza stravolgimenti per il lavoro che fanno da sempre.

Dal punto di vista del business, i servizi di backend implementati rappresentano il punto di partenza di qualsiasi scenario evolutivo. Senza, non solo non si sarebbero potuti implementare gli altri componenti che abbiamo trattato, ma l'attività sarebbe rimasta legata a Shopify e agli altri servizi terzi indefinitamente, dovendo sopportare costi di gestione non ammortizzabili e l'impossibilità di scalare ed evolvere mantenendo la soluzione manutenibile.

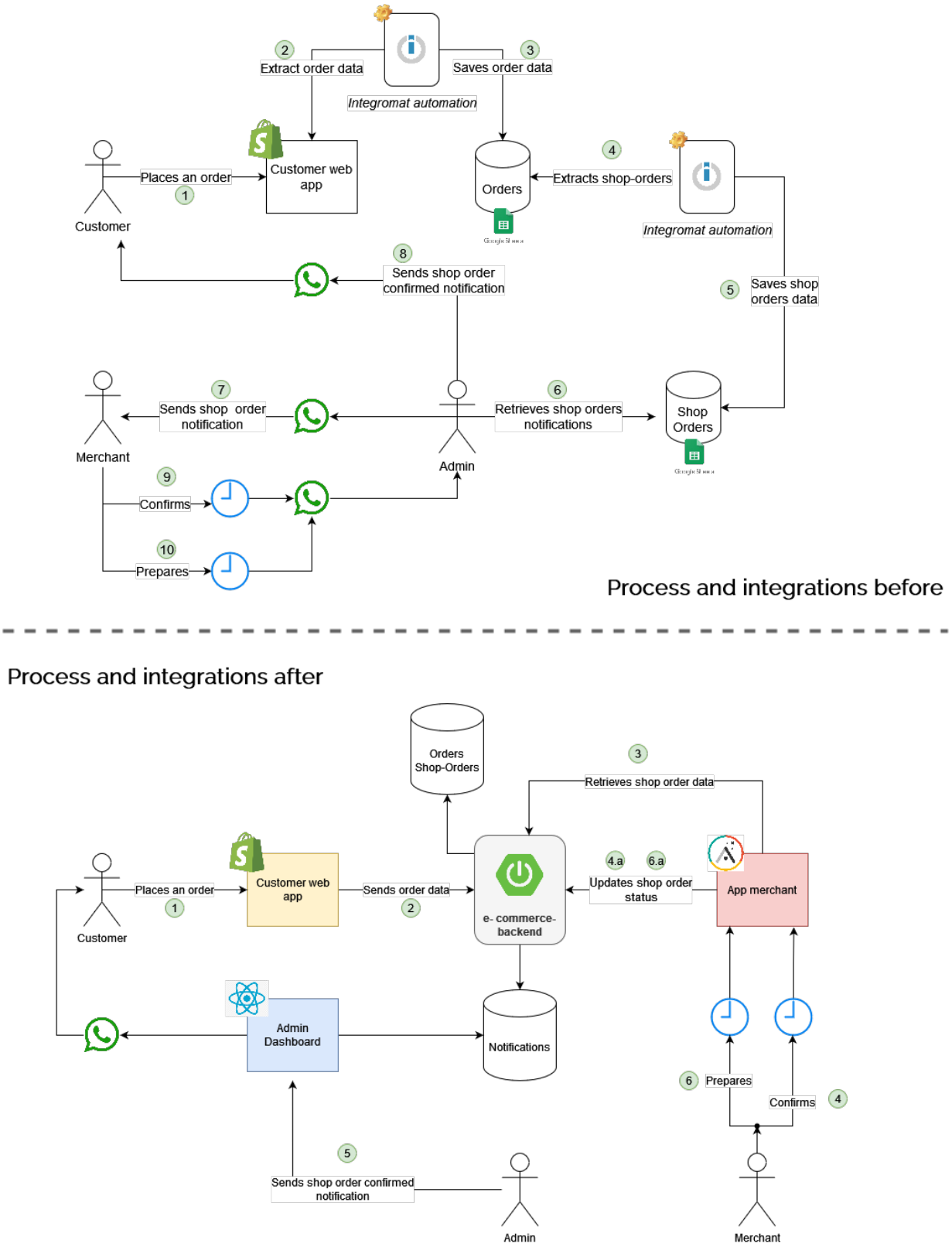

Figura 5.1: Integrazioni per la gestione degli ordini con e senza tecnologia proprietaria

CONCLUSIONI

In questo progetto di tesi si è sviluppato e descritto un insieme di applicativi web-oriented per l'implementazione di un marketplace per il commercio elettronico di prossimità. In particolare, abbiamo fatto riferimento ad un caso reale, quello di Daje! - startup romana nata durante il primo periodo pandemico - la cui mission è stata riportata nei primi capitoli.

Nel primo capitolo abbiamo fornito il contesto in cui il business è nato, fornendo i dati sull'espansione del settore nel periodo in esame, giustificando cosi la missione del progetto. Abbiamo poi mostrato le peculiarità del modello di Daje! in relazione ai modelli più classici del commercio elettronico. Nel secondo capitolo abbiamo descritto le principali sfide da affrontare e i problemi da risolvere per l'implementazione di un simile sistema, nella forma dell'analisi dei requisiti dello stesso. In particolare, ci siamo concentrati sulla gestione del ciclo di vita degli ordini, mostrando come l'intero processo rappresentasse la principale sfida, nonchè la differenza principale con le altre più semplici implementazioni di un e-commerce. Nel terzo capitolo, abbiamo tentato di giustificare alcune scelte sull'uso di soluzioni no-code per la minimizzazione del time-to-market del progetto, dandone un overview dal punto di vista degli asset tecnologici coinvolti. Al fine di introdurre la trattazione sulla progettazione di un'applicazione mobile in dotazione ai negozianti per la gestione del ciclo di vita degli ordini, abbiamo poi mostrato l'intera architettura della soluzione descrivendo le metodologie e le tecnologie con cui ogni componente è stato sviluppato. Nel quarto capitolo si è descritto il progetto tecnico di questi componenti, mostrando la progettazione della base di dati e dei servizi in grado di interrogarla, nonchè il progetto delle API per l'integrare dei servizi coinvolti. Abbiamo concluso la trattazione del progetto mostrando i risultati che si sono potuti raggiungere con l'utilizzo delle soluzioni sviluppate, ponendo l'accento sull'importanza che queste hanno avuto per il business nella sua fase iniziale.

Bibliografia

[1] Adalo. *Documentation*. URL: https://help.adalo.com/.

[2] Fivetran. *Shopify Data Model*. URL: https://fivetran.com/docs/applications/shopify.

[3] Make (ex Integromat). *Integromat Platform (Make)*. URL: https://www.make.com/en/product.

[4] Craig Larman. *Applicare UML e i Pattern - Analisi e progettazione orientata agli oggetti*. 2016.

[5] Nicola Mattina. *The story of Daje and how we invented proximity e-commerce*. 2021. URL: https://nicolamattina.substack.com/p/the-story-of-daje-and-how-we-invented?.

[6] MongoDB. *NoSQL vs Relational Databases*. URL: https://www.mongodb.com/scale/nosql-vs-relational-databases.

[7] Matteo Moroni. *5 abitudini di acquisto legate alla pandemia destinate a durare*. URL: https://www.shopify.com/it/blog/abitudini-di-acquisto-pandemia.

[8] Netcomm. *E-Commerce: Da canale secondario a ruolo determinante di vendita e interazione*. URL: https://www.osservatori.net/it/ricerche/comunicati-stampa/ecommerce-da-canale-secondario-a-ruolo-determinante-di-vendita-e-interazione.

[9] Netcomm. *Il nuovo scenario del Food & Grocery E-Commerce*. URL: https://www.consorzionetcomm.it/download/il-nuovo-scenario-del-foodgrocery-e-commerce/.

[10] OpenAPITools. *Generator Repository*. URL: https://github.com/OpenAPITools/openapi-generator/tree/master/modules/openapi-generator-gradle-plugin.

[11] Google Maps Platform. *Geocoding API*. URL: https://developers.google.com/maps/documentation/geocoding/.

[12] Shopify Reports. *Il futuro del commercio*. URL: https://cdn.shopify.com/s/files/1/1354/3229/files/Report_-_Il_futuro_del_commercio_-_Shopify.pdf?v=1607350240.

[13] Shopify. *Documentation*. URL: https://shopify.dev/.

[14] Shopify. *Liquid*. URL: https://shopify.dev/api/liquid.

[15] Shopify. *Metafields*. URL: https://shopify.dev/api/admin-rest/2022-04/resources/metafield.

[16] Swagger.io. *OpenAPI Specification*. URL: https://swagger.io/specification/.

[17] Swagger.io. *Understanding the API-First Approach to Building Products*. URL: https://swagger.io/resources/articles/adopting-an-api-first-approach/.