POLITECNICO DI TORINO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Piattaforma web assicurativa in modalità White Label

Relatore:
prof. Paolo Garza

Candidato:
Burlacu Alexandru Iustin

Aprile 2021

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di tesi dal laureando Burlacu Alexandru Iustin presso l'azienda Axieme.
L'obiettivo della tesi era la creazione di una piattaforma web scalabile nell'ambito delle assicurazioni finalizzata alla digitalizzazione di prodotti assicurativi più o meno complessi.

Ringraziamenti

Desidero esprimere la mia gratitudine al Prof. Paolo Garza, relatore della mia tesi, per il concreto aiuto datomi durante la stesura del lavoro.

Desidero altresì ringraziare la mia famiglia e i miei amici per il prezioso sostegno, l'incoraggiante appoggio e la costante vicinanza negli anni di studio.

Esprimo il mio grazie anche ai miei compagni di corso per il tempo vissuto assieme durante le lezioni e le numerose esperienze condivise.

Infine, un grazie davvero speciale a tutto il team di Axieme che mi hanno dato la possibiltà di mettermi in gioco e di sviluppare la piattaforma.

Torino, Aprile 2021 *Burlacu Alexandru Iustin*

Indice

Capitolo 1

Introduzione

Tale capitolo delinea l'idea sottostante la nascita della società Axieme e descrive brevemente il progetto a cui ho preso parte.

1.1 Principi generali

Il sistema assicurativo, se confrontato con altri modelli di business, rimane ancora oggi un settore poco dinamico ed arretrato per quanto riguarda la digitalizzazione dei processi, dei prodotti assicurativi e dello sviluppo IT.
Scarsa flessibilità, processi di emissione molto lenti, asimmetria informativa (ne è un esempio l'impossibilità da parte degli utenti di accedere ad un portale per controllare lo stato della loro polizza) sono solo alcuni dei problemi che questo settore presenta.
Inoltre, la reputazione e l'affidabiltà dell'utente, nonché contraente della polizza assicurativa, che dovrebbero essere uno dei fattori fondamentali nel momento in cui si preventiva un premio di polizza e si firma un contratto, non vengono minimamente considerati (salvo il ramo RCA). Tutto questo spinge la psiche umana a considerare l'assicurazione più come la solita tassa da pagare invece che un'opportunità da sfruttare per tutelare sé stessi e le persone (oltre che le cose) care.
È proprio questa staticità del settore che spinge le persone a non tutelarsi se non si è obbligati dalla legge e a rendere quello assicurativo uno tra i servizi più odiati dai consumatori. In questo contesto nasce quindi l'idea di Axieme.

1.2 Axieme

Figura 1.1 Axieme

Axieme è una Start-up Insurtech (termine nato dall'unione di “insurance” e “technology”, è il neologismo usato per descrivere la trasformazione tecnologica e digitale in corso nel settore assicurativo) fondata nel 2016, con sedi a Torino e Cagliari.
Axieme è nata con una visione ben definita, ovvero portare il processo di digitalizzazione nel settore assicurativo, andando così a dare quel forte impulso di cui questa realtà ha bisogno. Quindi si è dovuta ingegnare per proporre una serie di innovazioni e modelli semplici in grado di adattarsi perfettamente a tutte queste nuove esigenze del mercato ed essere pronta a rispondere alle nuove sfide future.

1.2.1 Giveback©, cerchie e rating

Una prima idea che Axieme propone è quello di premiare gli utenti più virtuosi, ovvero quelle persone che nel corso della copertura assicurativa non hanno aperto dei sinistri. Axieme ha introdotto nelle polizze assicurative, il meccanismo del “Giveback©”, ovvero il rimborso di una parte del premio alla scadenza della polizza secondo la logica “meno sinistri = maggiore rimborso”.
Il Giveback© diventa in questo modo l’elemento premiante per tutti quegli assicurati virtuosi e può diventare il punto di svolta in un settore ancora così arretrato e che non riesce a sfruttare al meglio tutte le tecnologie attuali. Quindi questo semplice nuovo modello inizia a tener conto anche dell’affidabilità e della reputazione del singolo individuo nel momento in cui viene stipulato un contratto.
La piattaforma riunisce tutti gli assicurati che stipulano una polizza simile in un gruppo di persone (cerchia legata al prodotto con una relazione 1 a 1) e tiene traccia dei sinistri avvenuti in questo gruppo. In relazione diretta con i numeri di sinistri troviamo un altro elemento che Axieme propone, il “rating” (varia da un minimo di 0 ad un massimo di 5).
Inizialmente, al momento della creazione di un nuovo utente, ad esso viene assegnato un rating (rappresentato come “stelline”) che nel corso degli anni può scendere o salire in base al suo comportamento. Per esempio se l’utente apre un sinistro il rating scende abbassando così anche il Giveback©, al contrario se l’utente si “*comporta bene non aprendo sinistri*”, il rating aumenta, facendo salire anche il valore del Giveback©. Allo scadere della polizza sarà il comportamento della cerchia, il numero di persone all’interno di essa e il rating

dell’assicurato a determinare, secondo un algoritmo proprietario quale sarà il Giveback© totale, con la logica **minor numero di sinistri => maggiore Giveback©**.

Figura 1.2 Giveback

Questo nuovo modello, oltre a portare un vantaggio economico per l’assicurato, migliora anche il comportamento delle persone, riducendo al contempo le frodi assicurative.

1.2.2 Esempio pratico del modello

Ipotizziamo un valore di Giveback© massimo all’interno della cerchia del 20%.
Se una persona apre un sinistro, il Giveback© massimo della cerchia si abbassa per esempio al 15%, mentre quello dell’assicurato con il sinistro in corso diventa istantaneamente 0%.
Successivamente, se una persona si aggiunge alla stessa cerchia, questo fa sì che la percentuale del Giveback© della cerchia stessa, risalga ad esempio al 18% e così via fino a poter giungere di nuovo al valore massimo (20%), mentre quello del cliente con il sinistro continua a rimanere 0%.
Questo comportamento viene definito “**effetto fisarmonica**”, ovvero il valore del Giveback© sale o si abbassa in base a quello che accade all’interno della cerchia, dove l’apertura di un sinistro lo abbassa e inserimento di una nuova persona al gruppo lo fa di nuovo salire.

1.3 Piattaforma

La piattaforma è composta da 5 aree:

- un'area personale del cliente (**Wallet**) dove esso può controllare lo stato delle proprie polizze e vedere in tempo reale il Giveback©. Sempre da questo portale può pagare e/o rinnovare una polizza, vedere lo storico delle transazioni (pagamenti e richieste Giveback©), richiedere l'apertura di un sinistro e molto altro;
- un'area administrator privata, necessaria a gestire le polizze (**Backend**), interagire e cambiare lo stato delle altre parti della piattaforma;
- un'area pubblica (**Assicurazione**) alla quale un utente può accedere per scegliere un prodotto assicurativo e configurarlo seguendo un percorso guidato di quotazione che passa attraverso una serie di passi. Questa parte della piattaforma viene esposta in 2 modalità:
  - personalizzata con il marchio, i colori e i prodotti di Axieme;
  - personalizzata con il marchio e i colori di un partner (assicurativo, come ad esempio un broker, o non-assicurativo, come una banca). Quest'area di business nasce sotto il brand "**Axidigital**" - (vedi punti seguenti);
- un'area pubblica (**Axidigital**) alla quale i partner possono accedere per acquistare il modello di Axieme, esposto in diverse modalità:
  - acquistare la piattaforma come partner di Axieme e quindi vendere i suoi prodotti;
  - acquistare il software direttamente, avendo la possibilità di creare i propri prodotti e personalizzarli a piacere;
- una seconda area administrator (**Badmin**) privata dei partner (coloro che acquistano la piattaforma da **Axidigital**), dove in base ai privilegi ottenuti (funzionalità disponibili) in fase di acquisto, si possono svolgere una serie di attività.

In particolare la parte web (**Assicurazione)** è composta da 6 pagine:

1. *assicurazione*: punto di ingresso, dove si sceglie il prodotto da un catalogo proposto;
2. *landing page*: pagina sulla quale vengono descritte le caratteristiche principali del prodotto e solitamente le FAQ;
3. *chatbot*: area preliminare nella quale si raccolgono alcune informazioni necessarie in fase di quotazione;
4. *quotatore*: pagina in cui viene calcolato il preventivo;
5. registrazione: form che raccoglie tutti i dati dell'utente. Se compilato, questi vengono raccolti e inviati al **Backend/Badmin** per notificare un nuovo ordine;
6. *thankyou page*: pagina finale del journey. Per i clienti più premurosi esiste anche la possibilità di acquistare subito la polizza collegandosi ad un gateway di pagamento esterno (Nexi), oppure richiedere un finanziamento grazie alla piattaforma di Soisy. La stessa funzionalità viene offerta anche dall'area personale, alla quale si accede con le credenziali inviate al termine della registrazione.

L’oggetto della mia tesi è stato sviluppare una parte di questa piattaforma. La sfida principale è stata quella di creare una piattaforma del tutto dinamica e indipendente dal prodotto assicurativo o dal partner. Lo sviluppo si è concentrato particolarmente sul trovare una soluzione che crei un legame diretto tra **Backend/Badmin** e **Assicurazione**, ovvero permettere ad un operatore di collegarsi, fissare una serie di parametri con la conseguenza di avere un impatto su **Assicurazione**.
Per esempio, il **Backend** e il **Badmin** offrono un modo semplice di andare a creare una landing page di un nuovo prodotto andando ad impostare le caratteristiche principali di questo, ovvero i titoli delle varie sezioni, le descrizioni, le immagini etc, e **Assicurazione** sulla base di queste informazioni e un template pre stabilito è in grado di renderizzare la pagina di conseguenza. Lo stesso approccio si è cercato di utilizzare anche nelle successive pagine, trovando molteplici difficoltà nella sezione del quotatore, in quanto alcuni prodotti assicurativi solo molto complessi e necessitano di scrittura di codice ad-hoc.
Un’altra funzionalità implementata sulla piattaforma è la possibilità di fornirla facilmente ad alcuni partner e offrire loro la possibilità di crearsi il proprio sito, renderizzato con dei template pre-definiti e con prodotti già presenti a catalogo (messi a disposizione da Axieme) oppure configurarbili in autonomia. Anche questa funzionalità l’obiettivo era quello di gestirla con alcuni click dal **Backend** o dal **Badmin**.

1.4 Capitoli successivi

Nei prossimi capitoli verranno approfondite le soluzioni adottate, ponendo particolare enfasi sulle difficoltà riscontrate durante tutto il ciclo di sviluppo e come questo sono state superate.
Nel capitolo 2 ci sarà una breve spiegazione delle tecnologie, dei Framework e delle librerie utilizzate.
Il capitolo 3 è dedicato alla descrizione dello sviluppo della piattaforma, spiegando progetto per progetto come è stato sviluppato. Sempre in questo capitolo ci saranno dei riferimenti alle soluzioni simili dalle quali ci si è ispirati.
Il capitolo 4 evidenzierà le prestazioni della piattaforma confrontando l’applicazione sviluppata inizialmente come monolita (tutte le funzionalità nello stesso progetto) rispetto ai micro-servizi introdotti successivamente (logiche divise in più progetti), in ambiente Azure e Aruba.
Il capitolo 5 è lasciato alle conclusioni e alle considerazioni generali.

Capitolo 2

Tecnologie utilizzate

Tale capitolo descrive le tecnologie utilizzate (framework, database e strumenti per la messa in produzione del software) e metodologie di sviluppo adottate (AGILE).

2.1 Framework - CodeIgniter

Figura 1.3 CodeIgniter

La parte del backend e frontend dell'applicazione è sviluppata interamente utilizzando il framework ***CodeIgniter*** versione 3. Questo è un framework open source ed è basato sul paradigma di programmazione MVC (Model - View - Controller). I **controller** sono la parte fondamentale e sono scritti come classi PHP che estendono *CI_Controller* (classe padre da cui ereditano alcune importanti proprietà). Il loro compito principale è quello di fare da tramite tra l'elaborazione dei dati che riceve dal **model** e la loro visualizzazione nella **view**. Il **model** è sempre scritto come classe PHP che estende *CI_Model* e si occupa dei dati, fornendo il punto di contatto con lo strato di persistenza (il database). Inoltre esso deve essere caricato dal controller nel suo costruttore per poter eseguire i suoi metodi e quindi interfacciarsi al database. In una fase successiva si è passati alla versione 4 di CodeIgniter, per sviluppare alcune aree della piattaforma.

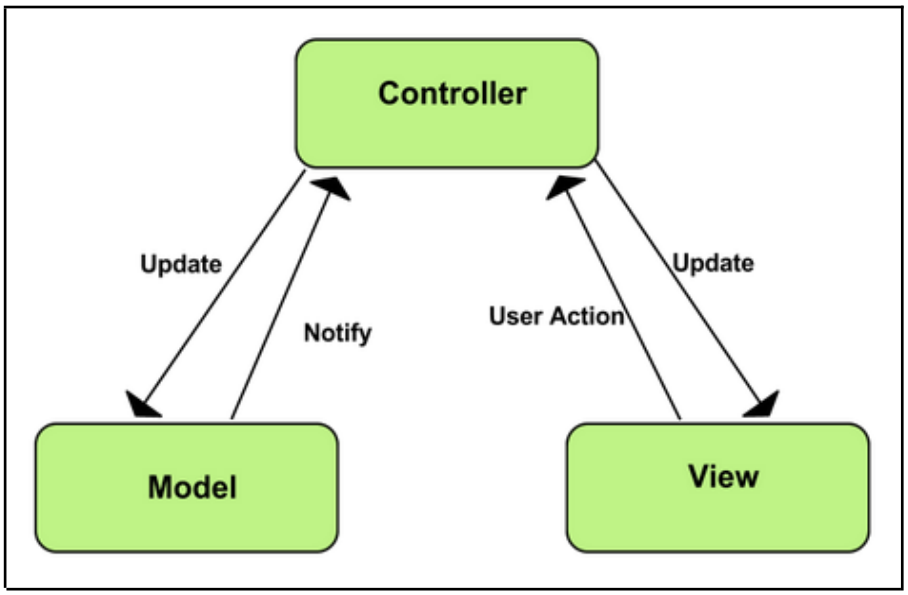


Figura 1.4 MVC

2.1.1 Controller

```
class MyController extends CI_Controller {

        public __construct( ) {
                loadModel( “MyModel” );
        }

        public function index( ) {

                $data = getDataFromModel( );

                // elaborazione e preparazione dei dati ricevuti dal Model

                echo view( “index”, $data );
        }
}
```

Nel controller, dopo aver interagito con il model per recuperare i dati dal database, questi vengono in qualche modo elaborati e trasformati, se necessario, e poi passati come parametri alla vista, la quale gli visualizzerà nel template. Nel controller si può utilizzare anche la sessione per evitare di fare troppe richieste al model e quindi al database, guadagnando così in prestazioni.

2.1.2 Model

```
class MyModel extends CI_Model {

        public __construct( ) {
                loadDatabase( );
        }

        public function getDataFromModel( ) {
                $data = loadDataFromDb( );
                return $data;
        }

        public function insertData( $data ) { ... }
}
```

Per poter collegarsi al database è necessario prima configurare in un file di configurazione (*database.php*) alcune impostazioni per potervi accedere. I parametri principali sono *hostname*, *username* e *password* e il *database* che si intende usare.

Il framework permette di interagire con il database in diversi modi:

1. scrivere le query direttamente nel codice utilizzando il linguaggio SQL e poi utilizzare i metodi ereditati (*CRUD*) dal Model per eseguirle; ma è una tecnica non consigliata in quanto vulnerabile dal punto di vista della sicurezza e soggetta ad attacchi, quali SQL Injection;
2. utillizzare il pattern "*ACTIVE RECORD*" che permette tramite una serie di metodi del framework di interfacciarsi al database (*CRUD*). Risolve in parte i problemi di sicurezza.
3. scrivere le query che devono essere eseguite nel database utilizzando il meccanismo delle **STORED PROCEDURE**, prevenendo così tecniche di attacco quali SQL Injection. Inoltre in questo modo si mantengono distaccate le logiche del database dalle logiche del codice, fornendo nel model soltanto l'interfaccia per interagire con lo strato di persistenza senza fornire i dettagli di come le query vengano realmente implementate, limitandosi a fare solo le chiamate.

2.1.3 View

```
<html>
        <head>

        </head>
        <body>
                <?php
                        echo view( “index.php”, $data );
                ?>

        </body>
</html>
```

Il template delle viste viene costruito utilizzando i tag HTML. All’interno di esse, utilizzando un tag particolare **<?php ?>**, è possibile caricare altre viste e passare a queste i dati ricevuti dal controller.
Ruolo delle viste è quello di occuparsi di renderizzare le pagine dell’applicazione e di fornire al client un modo per interagire con l’applicativo. Gli eventi dell’utente vengono raccolti nella vista e comunicati al controller passandoli eventuali parametri. Questo elabora la richiesta, utilizza il model per interagire con il database e restituisce la risposata alla vista, la quale la visualizzerà a schermo.

2.1.4 Route

Un altro elemento fondamentale del framework sono le rotte, le quali permettono facilmente di intercettare e gestire le richieste di un client. Per configurare tali rotte si lavora principalmente nel file “*routes.php*” definendole come segue:

$route[‘default_controller’] = “MyController/index”

E’ necessario che ci sia questa riga in questo file, in quanto essa costituisce la rotta di default e il punto di ingresso nella nostra applicazione. Senza di questa il framework non trovando la default route non avrebbe un punto di ingresso e avremo un errore 404 NOT FOUND.

Esempio URL di richiesta:

https://virtual-host/assicurazione/<controller>/<metodo>/<parametro>

Si può notare come nella richiesta ci sono tutti gli elementi necessari per poter gestire una richiesta del client, ovvero il controller, il metodo da invocare ed i relativi parametri

(quest'ultimi possono anche essere opzionali). Questa richiesta viene definita nel file di configurazione come segue:

$route['assicurazione/(:any)'] = "MyController/index/$1";

ovvero si indica che sotto la url "*assicurazione/(:any)*" si va ad invocare il metodo *index()* nella classe *MyController* passandoli in ingresso il parametro specificato nella URL.

2.2 MariaDB

Figura 1.5 MariaDB

Per sviluppare lo strato di persistenza si sono utilizzati sia i database relazionali (MariaDB) che i database non relazionali (MongoDB). In particolare MariaDB contiene tutta l'infrastruttura delle tabelle e le stored procedures utilizzate dai model per eseguire le query. Di seguito, due esempi di stored più utilizzate (select e insert):

```
CREATE DEFINER='localchost'@'*' PROCEDURE 'get_data' (IN _id INT(10))
    BEGIN
        SELECT
        FROM
        JOIN
        WHERE
    END

CREATE DEFINER='localchost'@'*' PROCEDURE 'set_data' (IN _nome
                        VARCHAR(255), _cognome VARCHAR(255), ...)
    BEGIN
        INSERT INTO "table"(  )
        VALUES(  );
    END
```

Le stored procedures sono una tecnica che permette di scrivere le query direttamente sul database e costituiscono una prima difesa contro attacchi come SQL INJECTION.

Nel model per poterle richiamare si utilizza la parola chiave “CALL” seguito dal nome della query che si intende invocare:

```
$stored = “CALL get_data()”;
$this→db→query( $stored );
```

Per rispettare le normative GDPR i dati sensibili legati ad un utente vengono cifrati utilizzando un algoritmo di cifratura simmetrico (AES-128 in modalità ECB). La cifratura dei dati viene eseguita nei model prima di invocare le query, utilizzando una libreria che CodeIgniter mette a disposizione.

2.3 MongoDB

Figura 1.6 MongoDB

MongoDB lo si è utilizzato principalmente per andare a salvare dati di grosse dimensioni quali pdf di polizze, questionari degli utenti e immagini renderizzate sulle diverse pagine della piattaforma. La comunicazione con questo tipo di database non è nativa del framework, ma avviene mediante il supporto di una libreria di terze parti, che mette a disposizione un driver per agevolare l’interazione.
Inoltre la libreria fornisce anche una serie di metodi base per andare a fare le interrogazioni e recuperare tali documenti. Oltre alla libreria è necessario anche caricare nel progetto anche un file di configurazione per impostare i parametri di connessione (localhost, username, password etc).

2.4 CodeIgniter 4

Nel momento in cui siamo diventati anche software house, ci si è presto accorti che il metodo di sviluppo offerto dalla versione 3 non era più tanto adatto, in quanto molte operazioni base il framework, o non lo permetteva oppure era limitante (banalmente l’interazione con il database).
Questo fatto impattava direttamente sulla messa in produzione di un progetto, in quanto ogni volta che si modificava una stored procedure nell’ambiente dev, bisognava segnarsi la modifica e ricordarsi poi di riportarla anche in ambiente di produzione. Un’azione poco scalabile e soggetta ad errori dovuti principalmente a dimenticanze.

Allora dopo una breve analisi, valutando anche l'opzione di cambiare framework (Laravel oppure Spring per la parte backend) si è deciso di utilizzare la versione 4 del framework CodeIgniter, in quanto non avevamo il tempo necessario per migrare tutto il codice nei nuovi framework. Inoltre, se avessimo deciso di utilizzare Spring, non solo si doveva migrare il codice, ma si doveva anche convertire il codice PHP in codice Java e adattarsi alla metodologia di sviluppo che il framework obbliga di seguire (controllers, data transfer objects, entities e repositories). Un'operazione lunga che in quel momento non potevamo permetterci, data la mole di lavoro.

2.4.1 Novità introdotte dal framework

La versione 4 di CodeIgniter ha introdotto una serie di migliorie che si pongono come obiettivo quello di rendere lo sviluppo più semplice e scalabile:

- un ORM (object-relational mapping) più avanzato, ovvero uno strato intermedio tra il model e il database che permetta facilmente di scrivere le query e di interfacciarsi con lo strato di persistenza senza dover passare più attraverso le stored. In questo modo, essendo che le query sono scritte direttamente nel codice, la messa in produzione è diventato molto più scalabile. Lo svantaggio di questa soluzione è stato quello che abbiamo riscritto tutte le stored e tradotte in codice.
- La possiblità di avere un model per ogni tabella del database che si porta dietro la configurazione di ciascuna tabella (regole di validazione, colonne della tabella e altri parametri).
- Introduzione delle entities, ovvero quelle classi che rappresentano le tabelle in un database relazionale.
- Una serie di file di configurazione con la principale funzione di preparare facilmente l'ambiente locale, sviluppo (pre-produzione) e produzione.
- Un modo diverso di definire le rotte e di raggrupparle in gruppi. Inoltre si definisce direttamente nel file di configurazione se si tratta di una post, get oppure put.
- I filtri che possono essere invocati prima o dopo uno specifico metodo. Per esempio un filtro che controlla se l'utente è loggato oppure no.

2.5 Tecnica di sviluppo: AGILE

Dal punto di vista dell'ingegneria del software, durante tutte le fasi del progetto il metodo di sviluppo adottato è stato quello *AGILE*. Questa scelta ci ha permesso una flessibilità massima dal punto di vista di sviluppo del codice. Inoltre si fonde perfettamente con lo "stile di vita di una start-up" nel quale la flessibilità, la dinamicità e una mentalità aperta a nuovi cambiamenti sono requisiti molto importanti e necessari. Il metodo *AGILE* ci ha permesso in determinati momenti di ritornare indietro sui nostri passi e andare a ridefinire i requisiti e la loro implementazione. Inoltre essendo un progetto ex-novo, senza avere tante fonti di ispirazione, è stato spesso necessario rivedere il codice sviluppato fino a quel

momento per adattarlo alle nuove esigenze che il mercato crea e anche alle esigenze e nuove idee sviluppate dal team.

2.5.1 Axidigital

Dopo aver raggiunto uno stato abbastanza stabile della piattaforma, le nuove opportunità di business presentatosi (**Axidigital** con il progetto **AxiSales**) hanno fatto sì che fosse necessaria una completa rivoluzione della piattaforma, Questo si è tradotto dal punto di vista IT in una completa revisione del codice e della sua struttura rivedendo anche la struttura del database. Inoltre si è iniziato ad adottare il concetto dei microservizi andando a creare una serie di endpoint che raccolgono a fattore comune tutte le funzionalità base della piattaforma.
Si è sviluppato così un certo numero di servizi fornitori di funzionalità, accessibile tramite API solo internamente e una serie di servizi utilizzatori di tale funzionalità. In questo modo si è creata una separazione dal punto di vista logico tra la parte che si occupa di processare i dati (fornitori) e quelli che devono utilizzare tali dati (utilizzatori). Inoltre questo ha portato anche il vantaggio di non avere lo stesso codice ripetuto più volte essendo così consoni al paradigma di programmazione **DRY** (*don't repeat yourself*), portando il beneficio di avere un'unica manutenzione del codice centralizzata. Un altro vantaggio di questa soluzione è che ha permesso di introdurre una serie di ottimizzazioni e migliorie con lo scopo di migliorare la leggibilità, incrementare le performance.

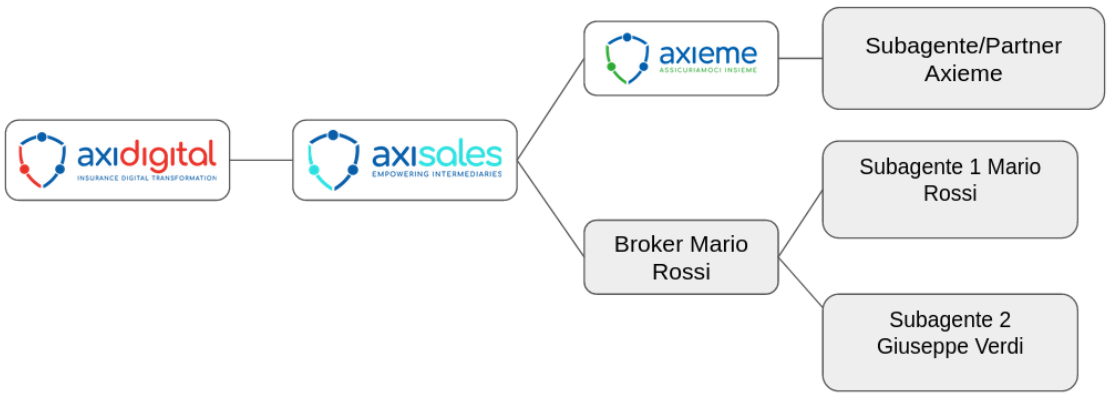


Figura 1.7 Axidigital con il progetto AxiSales

Questo schema è un esempio illustrativo di come il concetto di servizi viene usato, dove abbiamo **Axidigital** con il progetto **AxiSales** che offre un servizio, ovvero la possibilità di sfruttare le funzionalità della piattaforma. Axieme e altri Broker assciurativi sono utilizzatori di tale piattaforma in base ai privilegi che hanno ottenuto in fase di acquisto.

2.6 Jenkins

Figura 1.8 Jenkins

Jenkins è uno strumento open source che permette l'esecuzione automatica di una serie di task volti ad effettuare continuous integration, delivery e deployment (CI/CD).
È un software scritto in Java e per poter essere eseguito necessita di un server web (accessibile dal browser tramite la porta 8080) che supporti i servlet, per esempio *Apache Tomcat*.
Il successo di Jenkins è dovuto al fatto che è fortemente modulare e integra al suo interno vari sistemi di versionamento del codice (es. git, svn) e permette l'integrazione con qualsiasi tecnologia/linguaggio di programmazione grazie alla sua espandibilità data dal supporto ai plugin.

2.6.1 Continous integration, delivery, deployment

Per continous integration si intende quel processo che ha come obiettivo finale la costruzione di un software applicativo. E' suddiviso principalmente in tre fasi:

1. fase durante la quale si ottiene il codice sorgente dell'applicazione;
2. fase durante la quale viene fatto il "*build*" dell'applicativo;
3. fase durante la quale il software viene impacchettato in un modo tale da poter essere ridistribuito.

La continous delivery (CD) si occupa di rilasciare il pacchetto fornito dalla continous integration (CI) in un ambiente di pre produzione, quindi il suo compito è quello di scaricare il sorgente da qualche repository, farne il build, creare un pacchetto e distribuirlo in questo ambiente di pre produzione rendendolo così accessibile a chi si occupa di fare i test sul software, prima della messa in produzione.

Il continous deployment è il processo più esteso di tutti quanti in quanto si occupa di rilasciare l'applicativo direttamente in un ambiente di produzione, prevedendo quindi un'integrazione totale del codice a partire dal suo sviluppo fino al suo utilizzo.

Tutto questo processo viene svolto dal Jenkins in modo automatico mediante degli script, quindi non richiede interventi manuali da parte del progettista e permette inoltre di vedere l'applicativo come fosse sempre operativo.
Le altre funzionalità offerte dal Jenkins sono il monitoring delle risorse e la notifica di eventi in seguito alla messa in produzione.

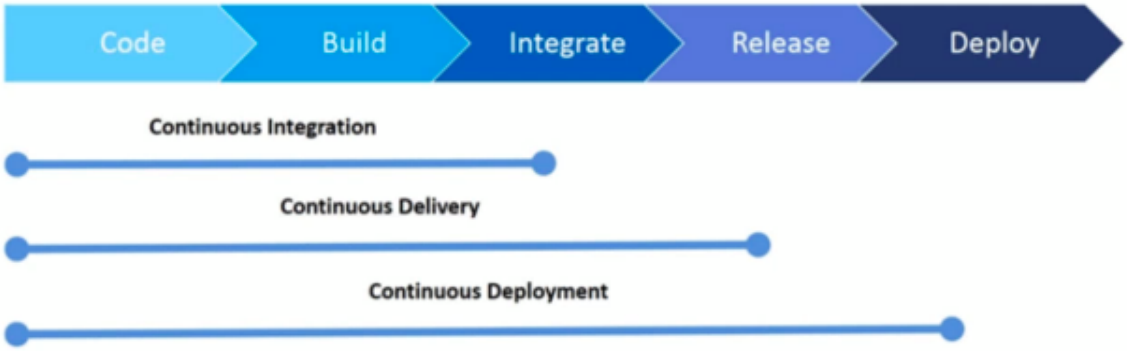


Figura 1.9 CI/CD

2.7 Docker

Figura 1.10 Docker

Docker è una tecnologia open source che fornisce uno standard per la costruzione, lo spostamento ed il rilascio di applicazioni basate sui container. Il container si basa sul concetto di virtualizzazione del software, condividendo così il kernel con la macchina host. A differenza di una macchina virtuale, i container richiedono molte meno risorse non dovendo emulare anche l'hardware ed eseguire un intero sistema operativo. Dal punto di vista del Sistema Operativo i container sono costituiti da namespace distinti e isolati tra loro (introdotti in Linux a partire dalla versione 2.4.19) dove il namespace è un insieme di processi che accedono a risorse kernel virtualizzate.

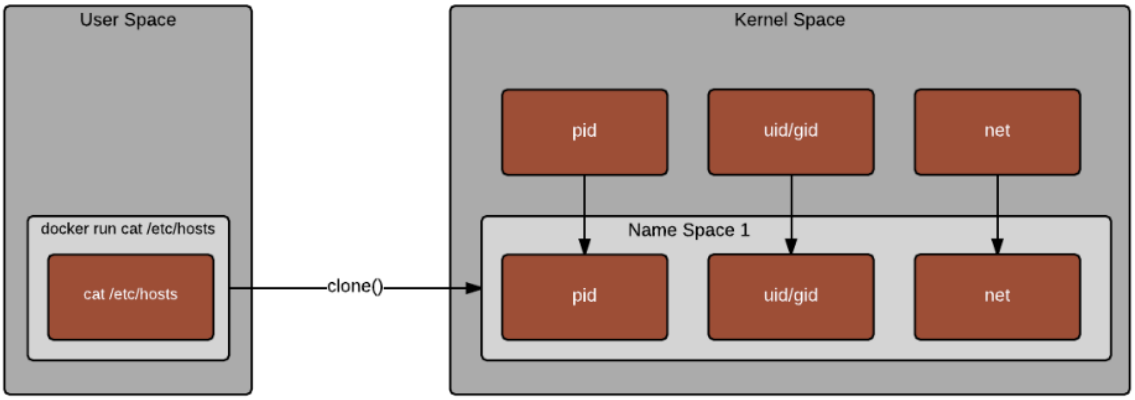


Figura 1.11 Namespace

Per contenere lo spazio di memorizzazione e facilitare la compresenza di molti container nelllo stesso host, Docker adotta un approccio del file system stratificato, così che esso appaia come uno stack di strati accessibili in sola lettura. Ciascun strato descrive del differenze rispetto allo strato sottostante. Per permettere ai processi di salvare i propri dati, il contenitore mette a disposizione un ulteriore strato accessibile sia in lettura che scrittura. Grazie a questo strato, tutte le operazioni di scrittura finiscono al suo interno e tutte le

operazioni successive su quei dati si fermano in tale strato. Si può dedurre che i file che stanno nei livelli più bassi sono quelli che potenzialmente possono essere condivisibili fra più processi, in quanto tendono a cambiare raramente, mentre quelli che stanno più in alto sono più riservati e utilizzati da meno container, avendo lo strato più esterno dedicato al solo container che lo ha modificato.

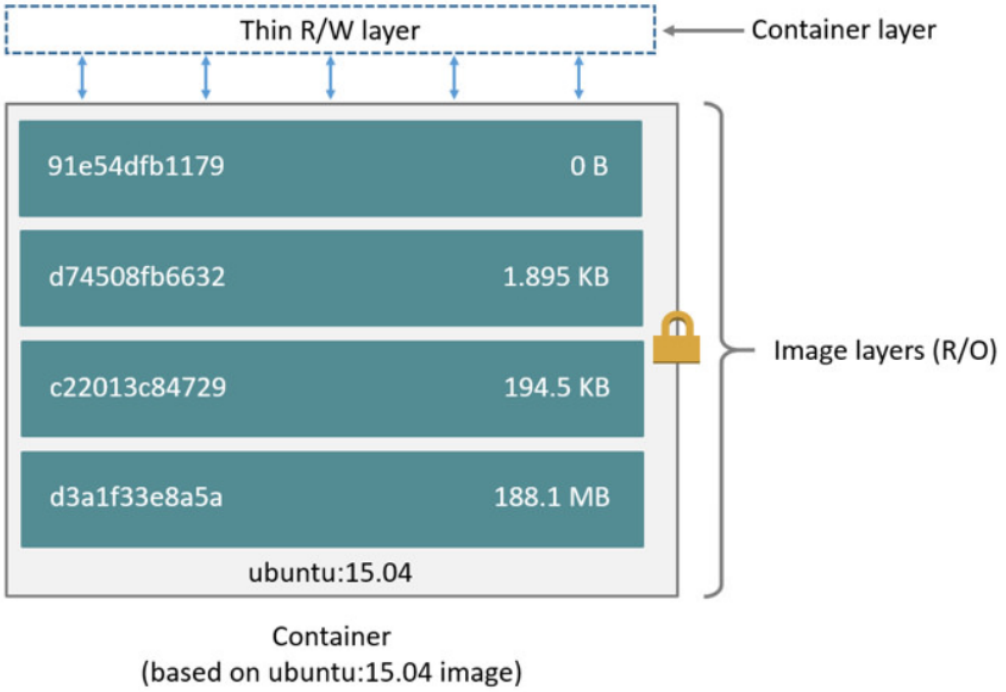


Figura 1.12 File system stratificato

I punti di forza dei container sono:

1. *incapsulamento*: un container racchiude al suo interno l'applicativo, file di configurazione, dipendenze etc;
2. l*eggerezza*: rispetto ad una VM richiede molto meno risorse in termini di RAM, CPU, disco e puo essere avviato in pochi secondi;
3. *scalabilità*: una macchina può facilmente ospitare decine, centinaia di container;
4. *portabilità*: grazie ad un file di configurazione (DocherFile), un container può essere spostato facilmente dal ambiente di sviluppo all'ambiente di produzione;
5. orientato ai DEV-OPS.

2.7.1 Componenti

I 4 componenti principali del Docker sono:

1. *Docker Engine:* è il core della piattaforma ed è il processo demone eseguito in background sulla macchina che deve ospitare i container, fornendo tutte le funzionalità e i servizi del Docker.
2. *Docker Client:* è l'interfaccia API che si occupa di raccogliere i comandi del client ed inviarli al docker engine (ogni comando inizia con la parola chiave "docker").
3. Docker Image: è un'insieme di file e parametri necessari a configurare un'applicazione a runtime.
4. *Docker Container:* il container è un'istanza in esecuzione di un'immagine alla quale è stata aggiunta uno strato R/W sovrapposto agli strati accessibili solo in lettura (immutabili).

2.7.2 Utilizzo in Axieme

I container sono utilizzati per la messa in produzione di una serie di progetti che devono essere eseguiti sul cloud AZURE di Microsoft. In particolare, grazie all'introduzione dei micro-servizi, sono stati sviluppati una serie di progetti che contengono solo le funzionalità core della piattaforma e poi altri che sono solo utilizzatori di tali servizi. A partire da questi progetti si sono create poi delle immagini Docker pronte per essere eseguite su Azure.
Questo approccio è ancora in fase di sviluppo e testing, ma ci permette di scalare facilmente e di ridurre i tempi di sviluppo per progetti futuri.

Capitolo 3

Sviluppo piattaforma white label

Tale capitolo descrive lo sviluppo delle varie parti che costituiscono la piattaforma e la sua implementazione durante tutto il periodo della tesi.

3.1 Introduzione

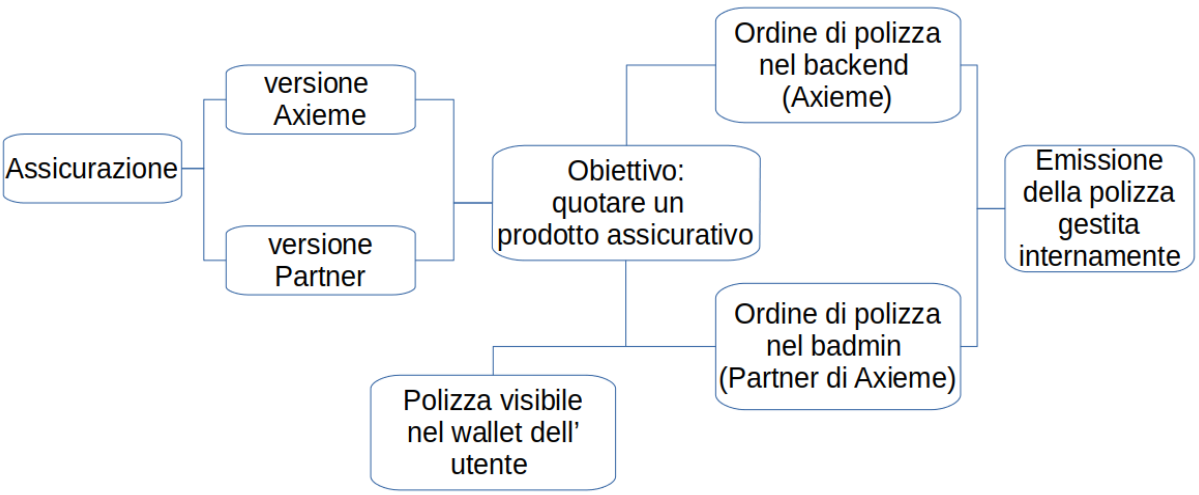


Figura 1.13 Schema riassuntivo piattaforma

Il progetto prevedeva lo sviluppo di una piattaforma in modalità **WHITE LABEL**, ovvero sviluppare una serie di applicativi, inizialmente pensati per essere usati solo internamente, poi successivamente “brandizzati” con parametri di altri partner in modo da farli apparire come se lo strumento fosse loro. Per fare questo è stato necessario astrarsi il più possibile in una prima fase di definizione dei requisiti e cercare si creare un modello che permettesse in seguito di mantenere e modificare il codice facilmente. Nonostante questo approccio, è stato spesso necessario rivedere il codice e ridefinire i requisiti. Il pattern MVC del framework, i metodi di sviluppo *AGILE* e le competenze degli altri compagni del team sono stati i punti di forza per questo progetto.

Il mio compito è stato principalmente quello di creare un'applicazione che permettesse la digitalizzazione di un prodotto assicurativo, con l'obiettivo finale di permettere successivamente ad un operatore (anche esterno ad Axieme) di collegarsi ad una dashboard e di configurare il prodotto autonomamente.
Per ottenere questo risultato ci si è ispirati ad altri Framework, più moderni del CodeIgniter, quali Spring o Angular, per andare a definire una serie di moduli dinamici (simili ai component di Angular), i quali contengono già tutte le informazioni logiche e funzionali necessarie al loro utilizzo.

3.2 Sviluppo

Lo sviluppo della piattaforma è stato guidato principalmente dalle esigenze del mercato e dalle opportunità di business presentatesi durante tutto il periodo della tesi. Inizialmente si doveva sviluppare una piattaforma, la quale sarebbe stata utilizzata solo internamente ad Axieme. Questo fece sì che tutta la concentrazione fosse rivolta verso una struttura del codice che permettesse in poco tempo la digitalizzazione di un prodotto assicurativo, senza preoccuparsi del fatto che la piattaforma dovesse essere poi venduta e utilizzata anche da terzi. Questa fase iniziale, non avendo ancora le idee molto chiare e nessuna fonte da cui prendere spunto, ha portato ad avere una struttura del database e del codice in generale molto disordinata e con una serie di entropie al suo interno che non sono state risolte facilmente.
Successivamente acquisendo più esperienza e avendo dei requisiti sempre più chiari, è stato possibile, in più sedute di analisi, riorganizzare inizialmente tutto lo strato di persistenza e poi anche il codice basandosi sulle nuove logiche e sulla nuova struttura di questo database.
Le nuove opportunità di business sviluppate dagli altri colleghi hanno avuto anch'esse un impatto sulla piattaforma, soprattutto dal punto di vista dell'ampliamento di quest'ultima. Se fino ad ora la piattaforma doveva essere utilizzata solo da Axieme, con il progetto **Axidigital**, questa si è trovata a dover essere venduta anche a partner con diversi livelli di utilizzo. Il risultato di questo nuova esigenza è stato quello di rivedere gran parte del codice di tutti i progetti introducendo anche il concetto dei microservizi e container.
Analizzando i vari progetti, si è raccolto a fattor comune tutte le funzionalità core e successivamente sono state spostate su una serie di endpoint, creando così dei servizi fornitori. Tutti gli altri progetti sono stati trasformati in utilizzatori di tali servizi.
A tutto questo si è aggiunta l'idea di utilizzare anche i container, introducendo cosi ulteriore flessibilità e scalabilità, non dipendendo neanche più dalle macchine fisiche che devono eseguire l'applicativo.

3.3 Assicurazione

Questa parte della piattaforma è quella dedicata alle quotazioni di un prodotto assicurativo, cioè il processo che ha come obiettivo il calcolo di un premio.
Lo sviluppo di questa parte è passato attraverso diverse fasi di revisione durante le quali il codice è stato ottimizzato sempre di più fino ad arrivare al punto in cui si è diviso in due progetti separati. Uno dedicato solo ad Axieme e in relazione diretta con il **Backend,** e l'altro dedicato al progetto **Axidigital**, quindi venduto ai broker assicurativi e legato direttamente al **Badmin.**
Inizialmente erano molto simili, poi successivamente, secondo alcune logiche di business interne, la parte dedicata ad **Axidigital** è stata rivista al punto che si è cercato di minimizzare oppure eliminare le dipendenze legate al concetto di Giveback©, cerchia e rating, ottenendo così una piattaforma utilizzata solo per la digitalizzazione dei prodotti.
Tutto questo processo di digitalizzazione (dei prodotti più semplici, principalmente "a pacchetti predefiniti") può essere fatto autonomamente, seguendo una serie di passi guidati da **Badmin.**
Nelle prossime sezioni spiegherò tutte le fasi dello sviluppo e le parti di cui è composta **Assicurazione**, definendo di più i dettagli tecnici, i problemi riscontrati e le soluzioni adottate. Inoltre, utilizzerò come esempio il primo prodotto sviluppato: **CASA**.

3.3.1 Landing Page

La landing è quella pagina che descrive le caratteristiche principali di un prodotto assicurativo. In più presenta una sezione finale nella pagina nella quale vengono definite le FAQ (frequently asked question).
La prima versione del MVC - Landing si basava su una sola tabella del database contente informazioni quali titoli delle sezioni, sottotitoli, descrizioni, immagini etc. In più sempre questa tabella conteneva le domande e le risposte delle FAQ (si è dimostrato velocemente non il posto più adatto). Il model si occupava di interrogare, con l'ausilio delle stored procedures tale tabella, dato in ingresso un parametro (nome prodotto recuperato dal URL). Il controller poi riceveva tali dati e se necessario gli trasformava adattandoli ai parametri della vista, la quale con l'utilizzo del tag php <?= ?> gli renderizzava.

Esempio:

- controller: effettua il mapping (se necessario) tra i parametri arrivati dal model e quelli inviati alla vista.
- View: rappresentazione grafica dei dati. Di seguito un esempio di vista:

```
<div id="introduzione">
        <h1><?= $titolo ?></h1>
        …..
        <p class="descrizione"><?= $descrizione ?></p>
</div>
```

Questa soluzione con una sola tabella si è dimostrata essere poco scalabile ed inefficiente, in quanto i diversi prodotti digitalizzati presentavano non tutti le stesse caratteristiche e quindi sezioni diverse. Questo portava ad avere righe della tabella per metà vuote, sprecando così spazio e risorse. Allora si sono definite altre 2 tabelle ausiliarie (landing_sezioni e legame_landing_sezione) con lo scopo principale di ottimizzare l'uso della tabella landing, introducendo un piccolo degrado in prestazioni perché è stato necessario introdurre le join tra le diverse tabelle.

- **Landing**: tabella contenente solo più quei parametri comuni fra tutte le landing (es. informazioni sul seo, parametri utilizzati nel codice etc.).

id_landing	id_prodotto	seo_description	...
1	7	...	…

- **Sezioni landing**: tabella contente le informazioni delle varie sezioni che compongono la pagina (titolo, descrizione, identificativi di mongo per recuperare le immagini etc). Si sono definiti dei template HTML base per ogni sezione:

  - sezione introduttiva;
  - sezione dedicata alla descrizione del prodotto;
  - sezione descrittiva dedicata ad Axieme;
  - sezione dedicata alla descrizione delle coperture, massimali etc del prodotto (opzionale);
  - sezione per le FAQ.

id_sezione	titolo	copy
1	Assicurazione Casa	…..
2	Descrizione Prodotto	…..

Ogni sezione, viene rappresentata dal punto di vista del codice da un modulo HTML elementare, il quale tramite PHP viene popolato (utilizzo dei tag php <?= ?>) con i dati ricevuti dal Controller.

Di seguito un esempio di modulo HTML di una sezione introduttiva e come questa viene renderizzata dal browser:

```
<section id=”introduzione”>
    <h1><?= $titolo ?></h1>
    <h3><?= $sottotitolo ?></h3>
    <div class=”card”>
        <div style=”url( <?= base_url (“image/”) . $img ?> )” ></div>
          ...
    </div>
</section>
```

Figura 1.14 Rappresentazione grafica di una sezione

Nel div che contiene l’immagine, questa viene recuperata da MongoDB chiamando il metodo mappato sulla url “/*image*”, al quale si passa come parametro l’identificativo univoco che il database non relazionale restituisce in fase di inserimento. Successivamente l’immagine viene renderizzata dall’browser correttamente grazie ad alcuni meta-dati settati nell’header di risposta (in particolare il formato del file binario – in questo caso .png oppure .jpeg).

- **Legame landing sezione**: tabella BRIDGE che associa a ciascuna landing le sezioni che la compongono.

id_landing	id_sezione
1	1
1	2

Utilizzando questo metodo si va ad ottimizzare sia codice, che risorse utilizzate sul database. In più ha facilitato il compito lato **Backend/Badmin,** permettendoci di creare delle interfacce grafiche molto semplici che permettessero ad un operatore di inizializzare facilmente tali landing. Le immagini vengono salvate sul database non relazionale e l'identificativo restituito da MongoDB viene memorizzato sul database relazionale nella tabella delle sezioni.

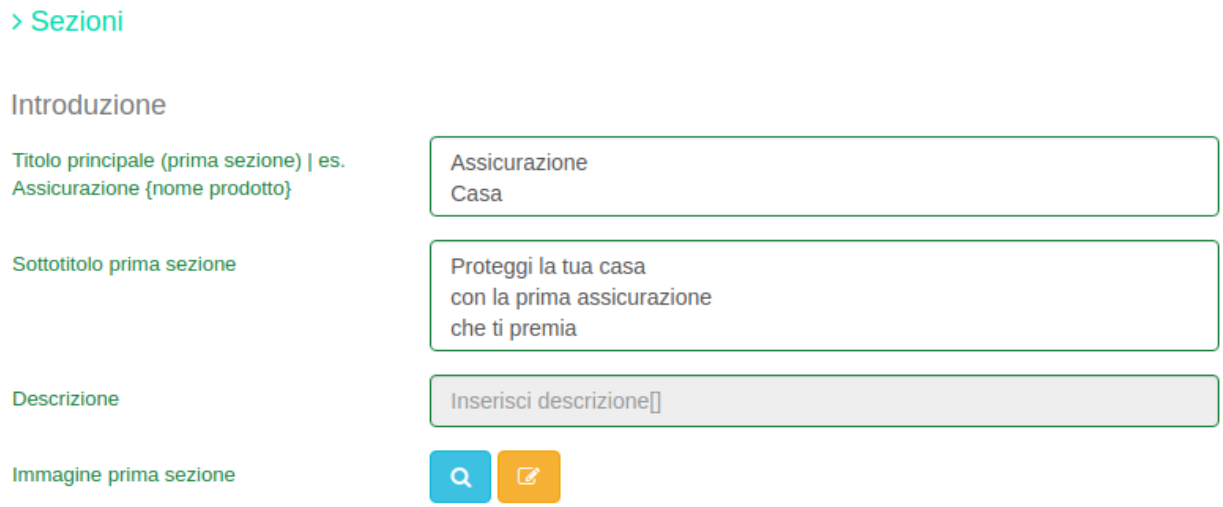


Figura 1.15 Backend: creazione landing – sezione introduttiva

L'ultima sezione della landing è riservata solitamente alle FAQ. Come detto precedentemente, le domande e le risposte venivano direttamente salvate sulla tabella landing. Si è dimostrato fin da subito il posto più inadatto in quanto non si potevano definire un numero arbitrario di domande e risposte, ma dipendeva da quante colonne avevamo dedicato sul database. Soluzione poco scalabile perché ogni volta che si doveva aggiungere una nuova FAQ alla pagina, significava dover aggiungere altre 2 nuove colonne alla tabella, con l'impatto di finale di ottenere di nuovo una tabella per metà vuota per quei prodotti che di FAQ ne hanno poche.
Per rimediare a questo problema, la soluzione adottata è simile alla precedente dove si sono introdotte 2 nuove tabelle. La prima contenente i dati delle domande e risposte e la seconda comportandosi da BRIDGE associando a ciascuna landing le proprie FAQ. In questo modo si è eliminato sia il problema della tabella per metà vuota che del numero arbitrario di domande; lo svantaggio è che ha introdotto di nuovo le join degradando le prestazioni.

- **Tabella landing faq**: ciascuna riga contiene una domanda e la relativa risposta.

id_faq	domanda	risposta
1	…..	…..

- **Tabella legame landing faq**: associa a ciascuna landing le sue FAQ

id_landing	id_faq
1	1

L'inserimento delle FAQ dal **Backend** si è dimostrato un po' più complicato, in quanto si deve fare attenzione a non creare doppioni (2 identiche), rischiando così di avere la tabella landing_faq che esploda facilmente come dimensione; in più ogni volta che si deve modificare una FAQ per una determinata landing, è necessario controllare che essa non venga visualizzata anche in landing di altri prodotti e in caso affermativo, la si deve prima clonare quella riga della tabella (inserimento), e poi fare la modifica di questa nuova riga.

3.3.2 Chatbot

Il chatbot è quella parte di **Assicurazione** che ha lo scopo di raccogliere alcuni parametri (es. fatturato annuale, età etc...) utilizzati in fase di quotazione per applicare dei filtri e calcolare il premio.
Anche qui l'approccio iniziale è stato quello di avere una sola tabella chatbot contenente per ogni step del chatbot i suoi parametri:

id	step1	domanda	risposta	step2	domanda	...	...	...
1	1							

Anche in questo caso la soluzione adottata, presentava più svantaggi che vantaggi.
Primo fra tutti riguardava il fatto che si aveva una tabella che cresceva di nuovo orizzontalmente, introducendo nuove colonne in tutti quei casi che si doveva aggingere una nuova tipologia di chatbot, ottenendo di nuovo una distribuzione dei dati in tabella non ottimale.
L'approccio successivo è stato quello di definire una serie di blocchi elementari che rappresentassero tutte le tipologie di chatbot necessari per la raccolta di informazioni (es. domande con risposta binaria, slider etc...). Per sviluppare questi moduli ci si è ispirati ad Angular e ai suoi componenti, andando a definire per ogni modulo oltre al template HTML anche la parte di scripting (Jquery in questo caso), necessaria a rendere quel modulo interattivo. In più, tramite PHP si popolano questi moduli con i dati che arrivano dal controller e dal database. I moduli definiti sono:

1. modulo composto da una domanda e da una risposta binaria;
2. modulo di tipo slider utile per qui casi in cui si devono raccogliere dati quali fatturato, età;
3. modulo di auto-completamento utilizzato per raccogliere dati come nome delle province oppure nomi di attività lavorative;
4. modulo dove l'input è di tipo numerico.

Di seguito un esempio HTML di modulo elementare (domanda con risposta binaria) contenente anche la parte di script (i dati tra i tag php <?= ?> sono quelli che arrivano dal controller e recuperati tramite il model dal database):

```
<div id="<?= $step ?>">
        ….
        <p> <?= $domanda ?></p>
        <div class="..">
                <button id="risposta1" class="..."><?= $risposta1 ?></button>
                <button id="risposta2" class="..."><?= $risposta2 ?></button>
        </div>
        <input type="hidden" id="risposta_binaria" />
</div>
<script>
        …
        $("#risposta1").click( function() {
                $("#risposta_binaria").val(1);
        });
        $("#risposta2").click( function() {
                $("#risposta_binaria").val(2);
        });
</script>
```

In questo esempio il modulo è costituito da una domanda, la quale viene recuperata ogni volta dinamicamente dal database. I due bottoni rappresentano le risposte a tale domanda. L'input di tipo hidden (nascosto) viene utilizzato come contenitore della risposta, 1 o 2 a seconda del pulsante cliccato. Il contenuto viene salvato grazie allo script che reagisce al click su uno dei 2 bottoni.
Rappresentazione grafica del modulo HTML:

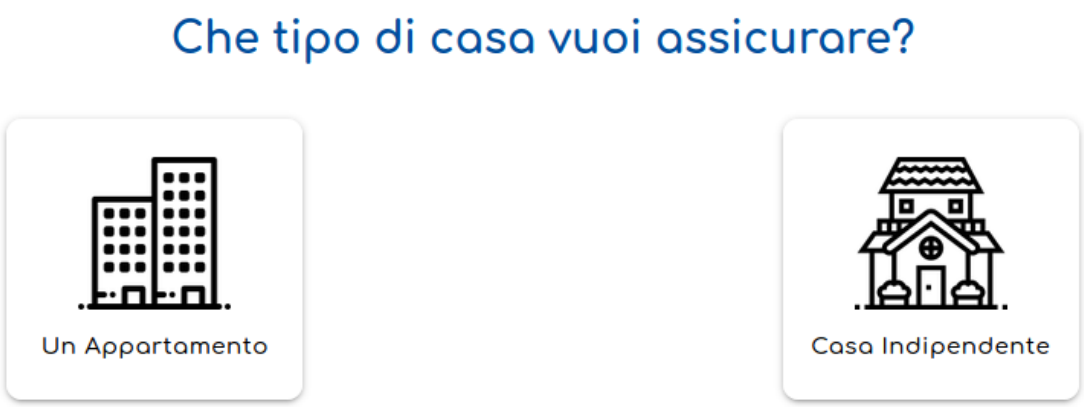


Figura 1.16 Rappresentazione grafica modulo con risposta binaria

Per quanto riguarda lo strato di persistenza, ad ogni modulo viene associata una tabella, nella quale vengono salvati i parametri che la vista deve renderizzare. Viene utilizzata anche in questo caso una tabella BRIDGE che associa ad ogni prodotto i suoi moduli; in più nella tabella BRIDGE è possibile definire anche l'ordine di come i vari moduli devono essere visualizzati nel chatbot.

- **Tabella chatbot binario**:

id	domanda	risposta_1	img_risposta_1	risposta_2	img_risposta_2
1	Che tipo di casa vuoi assicurare?	Un appartamento	….	Casa indipendente	….

- **Tabella legame chatbot prodotto**:

id_prodotto	id_chatbot	ordine_step
7	1	2

Lato **Backend/Badmin** abbiamo sviluppato anche per questa sezione un'interfaccia grafica che permettesse la configurazione di un chatbot, inserendo tutti i dati necessari per la configurazione di ciascun modulo.

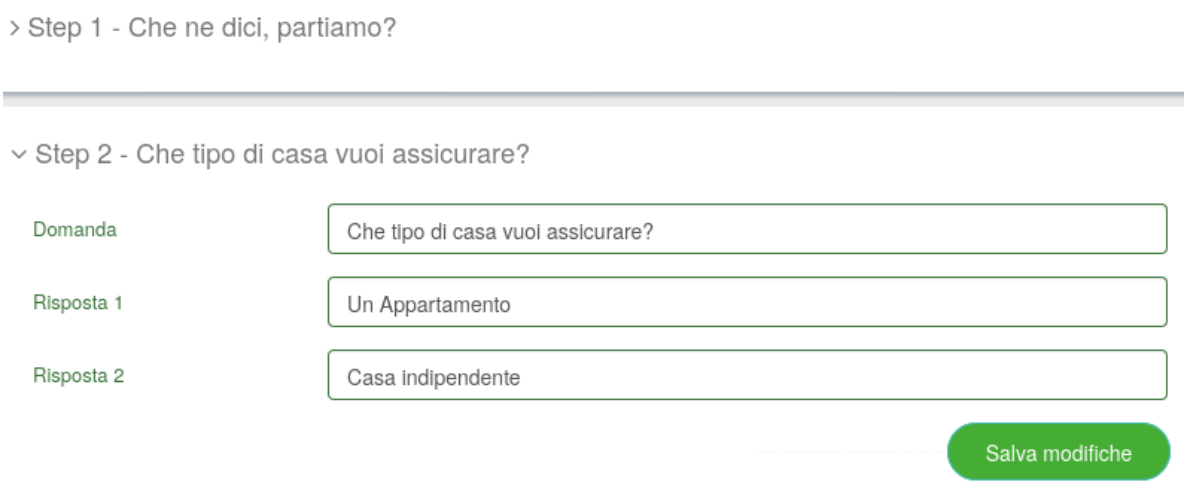


Figura 1.17 Backend creazione chatbot

3.3.3 Quotatore

Il quotatore è la parte della piattaforma che si occupa della quotazione di un prodotto assicurativo.
Per quotazione si intende quel processo tale per cui presi in ingresso una serie di parametri e una specifica configurazione, gli elabora e produce come risultato un premio e un valore di Giveback©, se il prodotto lo prevede.
Internamente, i prodotti assicurativi sono stati classificati in due macro categorie:

1. **prodotti assicurativi a pacchetto**: tipologia semplice in quanto il calcolo del premio non è soggetto a molti vincoli e spesso viene ricavato da alcune tabelle fornite dalla compagnia assicurativa. Il lavoro di digitalizzazione consiste:

   - lato **Backend**: creare delle interfacce che permettano la raccolta dei premi, insieme ad altri parametri e li inserisca sul database;
   - lato **Assicurazione**: permettere ad un utente di interagire con i pacchetti e visualizzare premio e Giveback© relativo;

2. **prodotti assicurativi NON a pacchetto**: più complessi in quanto le varie garanzie all'interno sono spesso correlate e vincolate tra loro e il calcolo del premio richiede l'aggiunta di ulteriore codice per gestire tali vincoli e correlazioni.
   Per questa tipologia di prodotti, spesso il punto di partenza per il calcolo del premio del premio lordo sono le seguenti formule:

   $$premio\ lordo = [\ (\ massimale \cdot tasso / 1000\ ) + una\_tantum\ ] \cdot accessori \cdot imposta \cdot sconto$$

   oppure

   $$premio\_lordo = premio\_netto \cdot imposta \cdot sconto$$

- *massimale*: valore massimo che può essere risarcito per i danni provocati;
- *tasso*: fattore moltiplicatore (usualmente pro-mille), dipendente dal tipo di prodotto e altri parametri quali classi di territorialità delle province;
- *una tantum*: valore fisso o percentuale che interviene una tantum nel calcolo di un premio;
- *accessori*: oneri extra applicati al calcolo del premio (dipendenti dal prodotto);
- *imposta*: tributo attribuito allo stato italiano. Ad ogni garanzia assicurativa è stabilita una specifica aliquota;
- *sconto*: percentuale di sconto stabilita internamente e anche questa dipendente dal prodotto.

Per il calcolo del Giveback© è necessario calcolare il premio imponibile (premio lordo al netto delle imposte) e moltiplicare questo per la percentuale di Giveback© massimo (stabilita secondo alcune politiche interne):

premio imponibile = [(massimale · tasso / 1000) + una_tantum] · accessori · sconto

Giveback© = premio imponibile · % Giveback© massimo

3.3.3.1 Prodotti assicurativi NON a pacchetto

Il primo prodotto assicurativo digitalizzato appartenente a questa categoria è stato "**CASA**". Poichè si approcciava per la prima volta una digitalizzazione, si può dire che questo è stato il prodotto più complesso da digitalizzare a causa dell'alto numero di vincoli e regole che legano fra loro le diverse garanzie che esso contiene.
La digitalizzazione è il processo di studio e analisi del prodotto in base al set informativo ricevuto dalla **Compagnia Assicurativa**. Questo documento è normalmente composto da 3 documenti:

1. *condizioni generali assicurative* (CGA): file pubblico, molto corposo che descrive il prodotto assicurativo;
2. *documento informativo precontrattuale* (DIP): anche questo è un file pubblico che riassume le CGA in poche pagine, evidenziando le caratteristiche più importanti del prodotto stesso;
3. *norme tariffarie*: documento privato che contiene tutti i parametri che concorrono al calcolo del premio lordo. Questo file è quello più utilizzato durante la fase di calcolo in quanto contiene i tassi, le imposte, gli accessori e tutti i vincoli tra le garanzie.

Inizialmente il lavoro di studio dei documenti veniva svolto dagli altri membri del team più competenti a estrapolare i dati da tali documenti. Il mio compito si limitava a prendere il risultato del loro lavoro (spesso dei fogli excel con tassi, premi, imposte, accessori) e tradurlo in codice PHP che facesse la quotazione. Successivamente, essendo aumentato il carico di lavoro per tutti i membri, questo lavoro di analisi ho iniziato a svolgerlo autonomamente essendo così più produttivi in quanto gli altri membri non dovevano più fare quel lavoro di digitalizzazione sui fogli di calcolo.

3.3.3.2 Sviluppo

Dal punto di vista dello sviluppo, vista l'esperienza precedente accumulata grazie al chatbot e landing, fin da subito si è approcciato un metodo che utilizzava più di una tabella. Si è definita una prima tabella che contiene l'id del prodotto e l'insieme delle macro-garanzie di cui esso è composto. Insieme a questa tabella se ne definita un'altra che contiene per ciascuna macro-garanzia le proprie sotto-garanzie e tutti quei parametri necessari per configurare un modulo elementare HTML del quotatore.

Per rappresentare quindi dinamicamente tale modulo, in modo che esso non dipenda dalla garanzia in sé, si sono definite queste due tabelle primarie:

- **Tabella garanzie quotatore**: contiene per ciascun prodotto le sezioni di cui è composto.

id	id_prodotto	nome_macro_garanzia
1	7	INCENDIO E DANNI
2	7	FURTO
3	7	RESPONSABILITA CIVILE
4	7	FURTO
...	...	...

- **Tabella moduli quotatore**: contiene i riferimenti alle garanzie e i relativi parametri (nomi, imposte accessori etc).

id	garanzia_id	nome_garanzia	parametro	imposta	accessori
1	1	RISCHIO LOCATIVO	rischio_locativo	1.2225	1.1
2	1	INCENDIO CONTENUTO	incendio_contenuto	1.2225	1.1
3	1	RICORSO TERZI	ricorso_terzi	...	...
...	...	...	...	...	...

Oltre a queste 2 tabelle principali sono state definite altre tabelle di appoggio, le quali contengono altre informazioni usate per il calcolo (es. tabelle con tassi e premi, tabelle per le province, comuni etc).

Sulla base delle informazioni contentute in queste 2 tabelle, ciascun modulo HTML è formato da una checkbox, una garanzia (es. Furto), un massimale (opzionale) e un premio.

```
<div id=”modulo>
   <label><?= $nome_garanzia ?></label>
   <input type=”checkbox” id=”id<?= $parametro ?>” name=”name<?= $parametro ?>”>
   <select id=”select<?=  $parametro ?>” name=”name<?= $parametro ?>”>
       <options>...</options>
    </select>
    <span id=”id_premio_<?= $parametro  ?>”></span>
</div>
```

Dal punto di vista grafico un modulo si presenta come segue:

Figura 1.18 Modulo quotatore

Per quanto riguarda il rendering della pagina, il controller intercetta la richiesta del browser (per es. /quotatore/casa), poi in base ad un parametro contenuto nell'url (casa) fa una chiamata al model passandoglielo. Il model con l'ausilio di una stored interroga il database (le tabelle citate sopra) e ritorna il risultato al controller. Questo manipola in qualche modo la risposta e carica la vista del quotatore passandogli in ingresso tali dati.
Nella vista, il lavoro consiste nell'andare a replicare il modulo elementare in base a quante garanzie comprende il prodotto (il numero di garanzie vengono definite in fase di digitalizzazione del prodotto).

Il processo di quotazione viene svolto principalmente lato server. In questo caso non avendo ancora le conoscenze dei componenti di Angular, non si sono creati gli script del tutto dinamici e legati direttamente al modulo HTML, ma gli si è definiti ad-hoc per ciascun prodotto. Facendo in questo modo, la digitalizzazione di un nuovo prodotto, richiedeva sempre la scrittura di un nuovo script, avendo in molti casi del codice ripetuto e molto difficile da mantenere.

Lato client mediante un form vengono raccolti tutti i dati (stato della checkbox (attiva/non attiva), massimale per ciascuna garanzia, alcuni input di tipo hidden) e poi tramite una chiamata AJAX tutto il form viene serializzato creando un oggetto JSON e mandato in POST al server.
Il server ricalcola per tutte le garanzie il premio applicando le formule sopra citate e poi ritorna sempre in un oggetto JSON tutti i dati al client, dove mediante uno script si aggiorna la vista con i premi e il Giveback© ricalcolati.

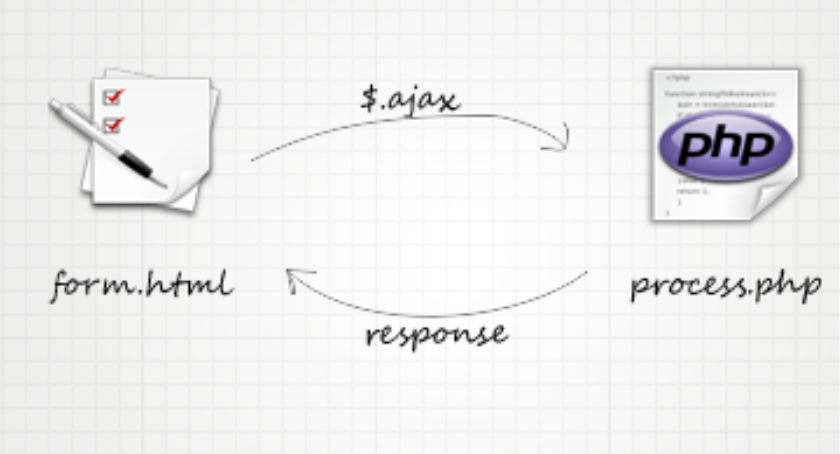


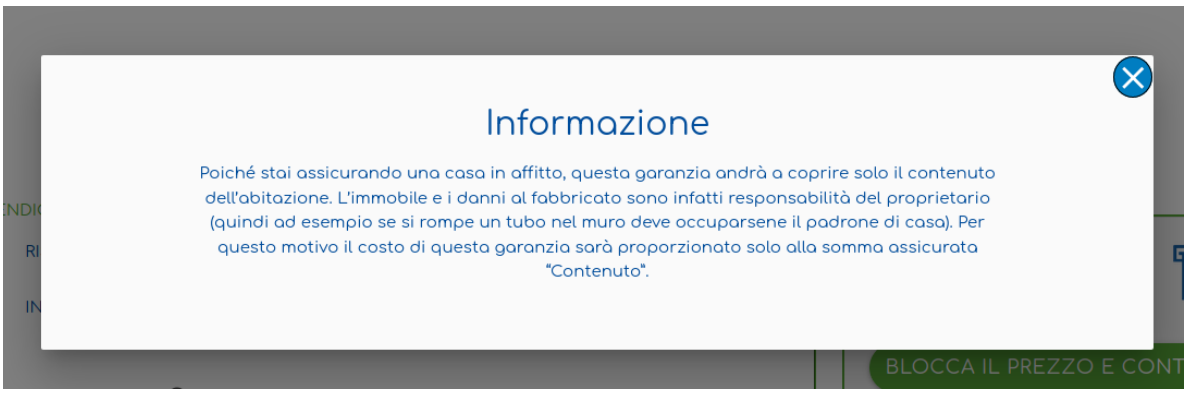
Figura 1.19 Chiamata ajax per il calcolo del premio

Sempre utilizzando degli script si vanno ad implementare i vincoli esistenti tra le diverse garanzie. Per vincolo si intende quel rapporto di "*esistenza*" tra 2 garanzie (base e facoltativa), ovvero, non si può selezionare una garanzia "facoltativa", senza avere anche la garanzia "base", quindi il comportamento dello script è quello di selezionare oltre alla garanzia "facoltativa", anche la "base", notificando all'utente questa informazione.

Esempio:

- garanzia facoltativa (dipendente da): rottura tubi
- garanzia base: incendio e danni contenuto

Selezionando "rottura tubi", il quotatore mostra il seguente messaggio e automaticamente



verrà selezionata anche la garanzia "incendio e danni contenuto".
Vale anche il discorso inverso, ovvero deselezionando la garanzia "base", verrà rimossa dalle garanzie selezionate anche la garanzia "facoltativa".

3.3.3.3 Prodotti a pacchetto

Questa tipologia di prodotto è caratterizzata dal fatto che non presenta una grande complessità nel processo di digitalizzazione, in quanto non ci sono calcoli, formule da applicare e vincoli tra garanzie, ma i premi e le garanzie contenute nel pacchetto sono spesso ricavati da una serie di tabelle che la compagnia assicurativa fornisce. Quindi il lavoro consiste principalmente nel tradurre queste informazioni in tabelle da salvare sul nostro database e poi interrogarle in fase di quotazione.
L'altra grande differenza riguarda la vista HTML elementare che è diversa in quanto l'utente non ha più la libertà di configurare il preventivo a piacere interagendo con le garanzie e massimali, ma può solo più interagire con i pacchetti preconfigurati con una serie di garanzie fisse e massimali, spesso non più a scelta. Inoltre per ogni pacchetto la compagnia fornisce i premi netti e le imposte da applicare per ricavare il lordo, quindi il processo di calcolo, una volta individuato il pacchetto che l'utente ha scelto e quindi il premio netto e l'imposta associata, consiste nell'applicare la seguente formula:

premio lordo = premio_netto + imposta - sconto

Nel database si sono definite 3 tabelle primarie più una serie di tabelle ausiliarie dipendenti dal prodotto che contengono i premi dei pacchetti principalmente.

- **Pacchetti quotatore**: contiene i pacchetti e l'id del prodotto a cui sono legati.

id	id_prodotto	nome_pacchetto
1	1	TEMPO LIBERO EASY
2	1	TEMPO LIBERO TOP
3	1	...
...	...	...

- **Garanzie pacchetto**: contiene tutte le garanzie contenute nei vari pacchetti.

id	nome_garanzia
1	INDENNIZZO MASSIMO INFORTUNI
2	ASSISTENZA E RIABILITAZIONE
...	...

- **Legame pacchetto garanzie**: tabella che associa a ciascun pacchetto le garanzie che esso contiene.

id_pacchetto	id_garanzia
1	1
1	2

- Altre tabelle ausiliarie che presentano la seguente struttura e che per ciascun pacchetto contengono il premio netto associato:

id	id_pacchetto	premio_netto
1	1	99
2	2	119

Per quanto riguarda lo sviluppo, la complessità maggiore si è presentata nel momento in cui si doveva individuare il pacchetto che l'utente ha cliccato nella vista. Allora nel HTML si sono definiti una serie di hidden che associasse al pacchetto all'id che questo ha nel database e il premio netto associato.

```
<div id="pacchetto>
    <input type="hidden" id="pacchetto_<?= $parametro ?>"
            name="pacchetto_<?= $parametro ?>">
    <input type="hidden" id="premio_<?= $parametro ?>"
            name="premio_<?= $parametro ?>">
    <span><?= $nome_garanzia ?>
</div>
```

Con l'ausilio di JQUERY al click sul pacchetto, interviene uno script che lo intercetta e tramite AJAX raccoglie i dati contenuti nel form e li invia al server. Il server prende il premio netto, applica la formula per ricavare il premio lordo e Giveback© e invia la risposta al client che visualizza il premio.

Dal punto di vista dell'utente il pacchetto si presenta in questo modo:

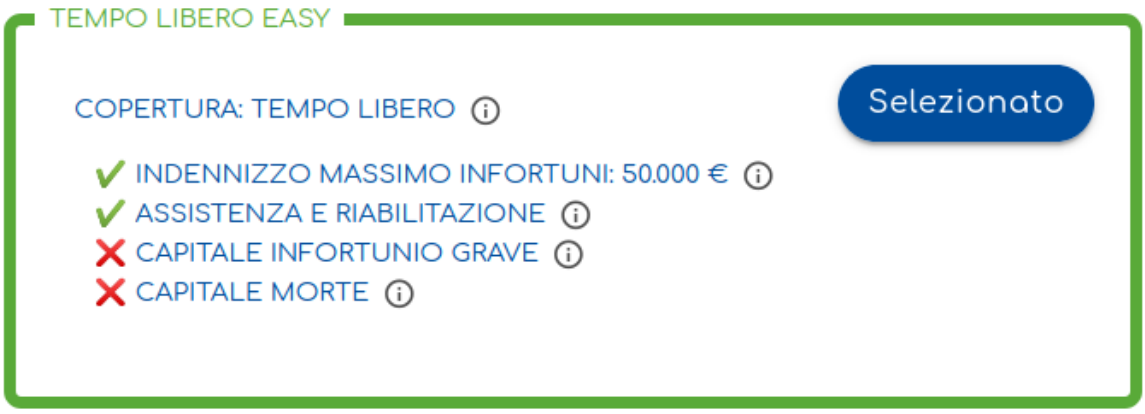


Figura 1.20 Pacchetto

Esso è composto da:

1. un titolo;
2. una serie di garanzie che riassumono brevemente cosa quel pacchetto contiene;
3. una "i" informativa vicina a ciascuna garanzia, la quale al click apre un dialog che la descrive meglio.

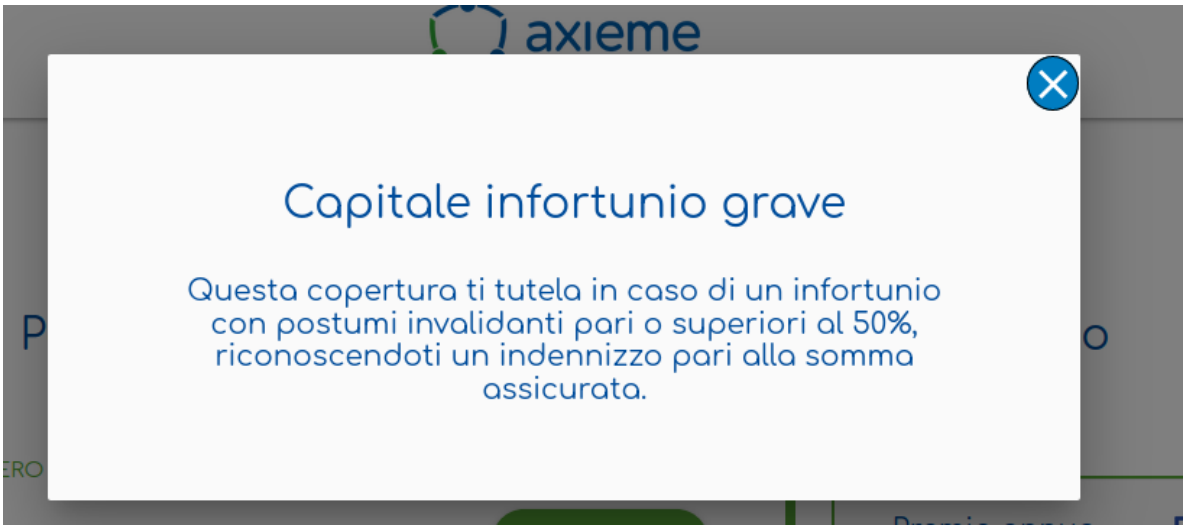


Figura 1.21 Modal garanzia

3.3.4 Registrazione

La registrazione è quel processo che si presenta alla fine del journey e ha come scopo quello di raccogliere i dati dell'utente (nome, cognome, email etc..), creare un ordine andando a salvare sul database tutti i dati, notificare all'utente il riepilogo della quotazione appena fatta e all'operatore il fatto che un nuovo ordine è stato creato.
La registrazione in tutto il percorso è stato uno dei punti più critici in quanto il problema principale da risolvere è stato quello di cercare di avere un form, di per sé complesso, in quanto le informazioni necessarie per emettere la polizza sono tante, il più semplice possibile e con una UX (user experience) per l'utente molto performante.
Il form si presenta in 2 forme. La prima dedicata alle solo persone fisiche (identificate da un codice fiscale) e la seconda per le persone giuridiche (identificate da una partita iva). Nella vista questa distinzione viene realizzata nel seguente modo:

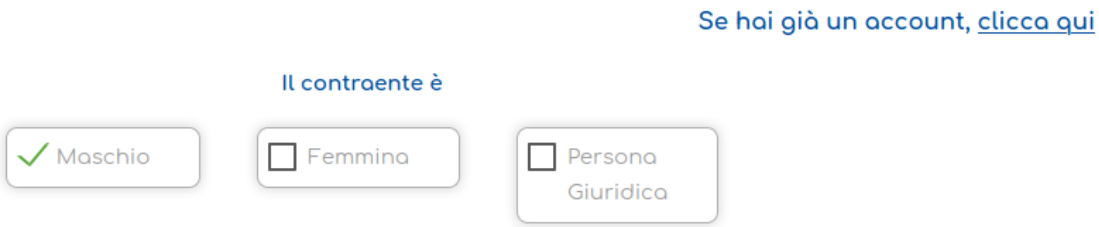


Figura 1.22 Personalità fisica o giuridica

Lo sviluppo è passato attraverso varie fasi, dove all'inizio si aveva un form molto complesso che richiedeva molte informazioni e presto ci si è accorti che la UX non era ottimale in quanto spesso i clienti riscontravano molte problematiche nel compilarlo.
Dopo varie analisi, non riuscendo a ridurre il numero di di informazioni da richiedere all'utente, si è adottata una soluzione che utilizzi degli input HTML auto-complete per quanto riguarda l'inserimento della provincia e del comune di residenza. Anche questa soluzione ha creato non poche problematiche, sia lato utente che sviluppo in quanto si era introdotto un vincolo molto forte, ovvero il cliente doveva cliccare sul suggerimento fornito, in quanto questo click scatenava dietro le quinte 2 eventi importanti:

1. riempire un input di tipo hidden con l'id della provincia appena selezionata usato poi in fase di validazione lato client per verificare la correttezza del dato (ovvero che l'utente abbia iserito una provincia valida, bloccando così prematuramente quegli utenti malintenzionati);
2. modificare nell'input successivo (sempre di tipo auto-complete) i comuni associati a quella provincia, evitando così di avere nella pagina tutti comuni italiani. Anche qua c'era il vincolo del click sul comune suggerito che riempiva un hidden utile nella validazione.

Figura 1.23 Auto-complete provincia

Essendo questo un vincolo molto forte, ci si è accorti ben presto che la maggioranza dei clienti non cliccava sui suggerimenti, quindi gli script non venivano attivati correttamente e quindi c'era un alto numero di clienti che venivano perse nel form di registrazione.
La soluzione successiva è stata quella di integrare all'interno della piattaforma le API di Google Maps con lo scopo di ridurre al minimo il numero di informazioni che l'utente dovesse compilare ed eliminare questa dipendenza dal click sul suggerimento. Con questa soluzione, solo dopo che l'utente ha compilato i 4 input (indirizzo, comune, numero civico e cap), viene scatenata una chiamata alle API di Google Maps, con il compito di validare tali dati. Se la risposta delle API è positiva, allora l'utente può proseguire a compilare i campi successivi, altrimenti i campi vengono svuotati e viene notificato all'utente l'errore.

Un'altra miglioria apportata al form di registrazione è stata la possibilità da parte dell'utente di fare una login, passando attraverso la sua area personale (**Wallet**). Per riuscire a fare questo, si è utilizzato un cookie a livello di sessione, che facesse in modo che due progetti differenti (**Wallet** e **Assicurazione**) condividessero la stessa sessione. Questo è stato facilitato anche dal fatto che i due progetti girassero sullo stesso server e quindi configurando una serie di parametri nei due progetti CodeIgniter hanno reso possibile tale condivisione.
Nel **Wallet**, una volta individuato l'utente e verificato che questo sia uno valido per la piattaforma, invece di accedere alla sua area personale, questo viene re-diretto al form di registrazione di **Assicurazione** con tutti i campi compilati (presi dalla sessione). L'utente così deve solo verificare la correttezza dei dati e procedere con la creazione dell'ordine.

3.3.4.1 Sviluppo

La vista del form viene costruita nel controller andando a definire 3 vettori contenenti ciascuno:

1. le informazioni relative alla persona fisica:
   - nome
   - cognome
   - codice fiscale
   - luogo e data nascita
   - residenza

2. le informazioni relative alla persona giuridica:
   - partita iva
   - ragione sociale
   - loco sede legale
3. le informazioni comuni a entrambe le personalità
   - email
   - telefono

Per ciascun input si definiscono una serie di proprietà, quali classi, name, id, icona etc.

```
$personalita_fisica = array(
                ‘nome’ => array(
                        ‘name’ => ‘nome’,
                        ‘classe’ => ‘fisica’,
                        ‘icon’ => ‘account_circle’
                        ‘type’ => ‘text’,
                        ‘label’ => ‘Nome’
                        ),
                ‘cognome’ => array(),
                        ...
                )
```

Allo stesso modo si sono definiti gli altri 2 vettori contenenti le informazioni della personalità giuridica e le informazioni generali (email e telefono).

Per creare la vista nel form di registrazione, sulla base dei 3 vettori che il controller li fornisce, si replica n volte il seguente modulo HTML:

```
<div class=’<?= $classe ?>’
    <i class=”materialize”><?= $icon ?></i>
    <input type=”<?= $type ?>” name =”<?= $name ?>”
    <label><?= $label ?></label
</div>
```

Ottenendo la seguente rappresentazione grafica:

Figura 1.24 Campi del form di registrazione

Il passaggio di dati tra quotatore e registrazione viene fatto in POST e nel controller della registrazione, per evitare packet sniffing e quindi modifiche del premio e del Giveback© durante questo passaggio viene fatta una verifica del premio. Essendo che la chiamata al controller della registrazione viene fatta in POST, si hanno tutti i parametri per poter ricalcolare il premio. Quindi, sulla base di questi parametri viene chiamata la calcola premio di ciascun prodotto per ricalcolarlo e poi si fa un confronto tra il premio ricevuto in POST e quello ricavato dal calcolo. Se questi non coincidono, l'utente viene rediretto alla pagina del quotatore dove può nuovamente rifare il preventivo, altrimenti accede al form.
La registrazione fa largo uso della sessione, in quanto dopo aver passato questa validazione sul premio, vengono richiamati una serie di metodi che hanno il compito di salvare in sessione tutti i dati della quotazione. Questo è necessario in quanto il submit del form viene effettuato sempre con una POST e i dati passati al server sono quelli del form di registrazione(informazioni sull'utente). Senza la sessione per salvare i parametri della quotazione, si perderebbero tutte le informazioni sul premio e le garanzie selezionate.
Lo scopo principale della registrazione è quello di creare un ordine, ma questo implica anche la creazione di un'utenza, di un contratto e di un titolo, solo se tutti i dati sono stati validati correttamente.

3.3.4.2 Validazione form client/server

Prima di effettuare la submit del form al server, questo deve essere validato lato client. Per effettuare tale validazione si utilizza uno script della libreria JQUERY, il quale definisce le regole che il form deve rispettare e i relativi messaggi di errore, utilizzando le seguente sintassi:

- rules : {'nome' : {required: true}}
- messages : {'nome' : {required: "Campo obbligatorio"}}

Successivamente utilizzando lo script *validate()*, applicato al form, viene invocata la validazione di tale regole e se una di queste non fa "match", viene mostrato il messaggio di errore associato ad essa.

- Esempio:

```
$("form").validate( {
        rules: rules,
        messages: messages,
        errorPlacement: function( ) { … },
        invalidHandler: function( ) { … },
        submitHandler: funciont( ) { … }
});
```

Questo script presenta anche una serie di parametri e callback:

1. rules: oggetto JSON che contiene le regole da validare;
2. messagges: oggetto JSON che contiene i messaggi di errore associate alle regole;
3. errorPlacement: invocata se il forn non è valido, ha il compito di mostrare i messaggi di errore vicino all'input HTML che l'ha generato;
4. invalidHandler: callback chiamata se il form non è valido e mostra un errore generale;
5. submitHandlet: callback chiamata se il form è valido. Raccoglie tutti i dati e poi con una chiamata ajax esegue la submit verso il server.

Le stesse regole di validazione vengono utilizzate anche lato server, scritte in php e presentando una sintassi molto simile, dove si definiscono le regole per ogni input, i messaggi di errore e infine si invoca un metodo di validazione con l'obiettivo di effettuare tale controllo.

3.3.4.3 Utenza

Successivamente alla validazione, il controller della registrazione prende tutti i dati contenuti nella POST e sulla base della email inserita controlla se si tratta di un'utenza nuova sulla piattaforma, oppure una già esistente. A seconda del risultato di questa verifica si aprono 2 strade possibili:

1. utenza nuova: questo implica la creazione di una nuova entry nel database sulla tabella utente (la piattaforma genera randomicamente e univoco anche un codice che può essere utilizzato in seguito per il **PROGRAMMA REFERRAL**) alla quale successivamente lega un'anagrafica:

    - fisica: se l'utente ha compilato il form di persona fisica (codice fiscale);
    - giuridica altrimenti (partita iva).

2. utenza esistente a cui bisogna solo legare l'ordine appena fatto.

Prima di creare l'utenza viene validato il codice fiscale (questo per evitare di avere dati "sporchi" sul database), utilizzando una libreria di terze parti a cui si passano le seguenti informazioni:

- nome, cognome, data di nascita, sesso, luogo di nascita
- codice fiscale inserito dall'utente utilizzato per essere confrontato con quello che la libreria calcola. In seguito al confronto tra i 2 CF, se essi non sono uguali, allora la registrazione viene interrotta e si notifica all'utente che ci sono stati dei problemi con il CF inserito.

Un fattore importante durante la creazione di un nuovo utente è il **PROGRAMMA REFERRAL**, ovvero se l'utente arriva sulla piattaforma sulla base di un invito da parte di qualcun altro. Questo comporta l'inserimento del "Codice Amico" nel form di registrazione.

Figura 1.25 Codice amico

Se l'utente inserisce un codice valido, allora colui che si è appena registrato riceve un Giveback© bonus di 5 € (subito disponibile), mentre l'invitante riceve 10 € solo il contratto dell'invitato viene attivato.
Questo viene realizzato sul database utilizzando 2 tabelle (giveback_disponibile e giveback_bonus) che contengono ciascuna informazioni sul valore del Giveback© e le utenze che hanno partecipato al PROGRAMMA REFERRAL.

3.3.4.4 Contratto

Una volta creata oppure recuperata l'utenza e ottenuto l'identificativo univoco dell'utente (primary key della tabella utente) il prossimo passo e la creazione di un contratto.
Su questa tabella vengono salvati i dati, quali data dell'ordine, tipologia di polizza assicurativa, vari premi etc.
Il contratto viene creato nello stato iniziale "NON ATTIVO" e con una data che riporta il momento in cui il preventivo è stato creato. Successivamente da backend questo viene attivato e vengono inseriti tutti i dati necessari per la gestione successiva della polizza, quali data emissione, data scadenza etc.

3.3.4.5 Titolo

Dopo il contratto viene inserita una entry sulla tabella dei titoli. Questa tabella è quella che tiene traccia dei pagamenti effettuati dal cliente.
Secondo politiche interne, un titolo deve passare attraverso una serie di stati. Lo stato iniziale, alla creazione del titolo, è "*insoluto*". Una volta che l'utente ha effettuato l'acquisto, l'operatore deve attivare il contratto e procedere con la "*distinta*" del titolo (lo stato passa da insoluto a distintato) e vengono aggiornate anche la data di pagamento e la modalità (carta o bonifico).

3.3.4.6 Garanzie

Uno degli ultimi step nella registrazione comporta il salvataggio delle risposte del chatbot e delle garanzie selezionate da parte dell'utente.
Una prima implementazione si basava sulla la creazione di una tabella nuova per ogni nuovo prodotto digitalizzato. Questa soluzione si è dimostrata molto scomoda (sia per salvare i date che recuperarli) perché poco scalabile. Ad ogni nuovo prodotto digitalizzatto si dovevano creare la tabella, la stored per fare l'insermiento e quella per recuperare questi dati. Ciascuna tabella contiene una foreign key verso la tabella "contratto" ed essendo tabelle dedicate, la struttura delle altre colonne varia a seconda del prodotto assicurativo trattato.

3.3.4.7 Notifiche

L'ultimo step nella registrazione sono l'invio delle mail verso l'utente per notificarli il riepilogo delle garanzie appena selezionate e verso Axieme per notificare all'operatore la creazione dell'ordine. Nella mail di riepilogo verso l'utente vengono allegati anche una serie di documenti creati e/o recuperati ogni volta dinamicamente e dipendenti dal prodotto.
In particolare, alla fine di ogni quotazione viene creato un questionario il quale viene compilato automaticamente dall'applicazione utilizzando tutte le informazioni che l'utente fornisce durante il suo percorso nel journey (risposte date nel chatbot, premio lordo risultante dalla quotazione, dati anagrafici). Definito il template del questionario (un file in formato word), questo è stato parametrizzato in maniera tale da poter essere compilato direttamente nel codice utilizzando una libreria di terze parti (PhpWord).
In seguito un esempio che mostra una parte del questionario parametrizzato con i dati anagrafici dell'utente:

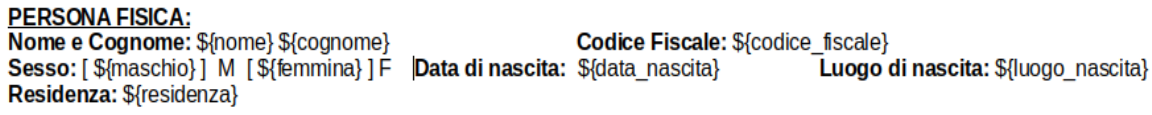


Figura 1.26 Questionario parametrizzato

Utilizzando questa sintassi - ${parametro} – nel codice viene creato un array associativo contenente come chiave il nome del parametro e come valore quello che deve gli deve essere sostituito, poi con l'ausilio della libreria PhpWord si esegue questa sostituzione generando in output un file word che successivamente verrà convertito in pdf.
Questo file pdf, essendo necessaria la sua visualizzazione anche nel **Backend/Badmin** da parte dell'operatore, viene salvato su MongoDB. Il metodo della libreria di Mongo che si occupa di inserire nel database non relazionale, restituisce un identificativo univoco, che viene poi salvato sulla tabella del contratto. Il percorso inverso (fatto principalmente nel Backend), consiste nell'utilizzare questo identificativo per recuperare il documento da Mongo e poi settandoli alcuni meta-dati (tipo di documento) visualizzare corettamente il

file. Questo approccio viene utilizzato ogni qual volta è necessario allegare alle mail dei documenti oppure visualizzarli.
Inoltre all'utente viene inviata anche un'email in cui trova le credenziali per poter accedere alla sua area personale (**Wallet**), nella quale si troverà anche l'ordine di polizza appena effettuato e la possibilità di effettuare il pagamento.

3.3.5 Thankyou page

Questa è la pagina finale del journey e ha l'obiettivo principale di far convertire l'ordine appena creato in un contratto, quindi contiene tutti i riferimenti verso le piattaforme in cui poter effettuare il pagamento.
La piattaforma con la quale la nostra comunica per effettuare i pagamenti è quella di Nexi, alla quale vengono inviati in POST una serie di parametri obbligatori (premio, url di ritorno in caso di errore/successo etc) e altri facoltativi (dati sull'utente come l'email).
Sempre da questa pagina l'utente può raggiungere la sua area riservata, nella quale si trova l'ordine appena creato.

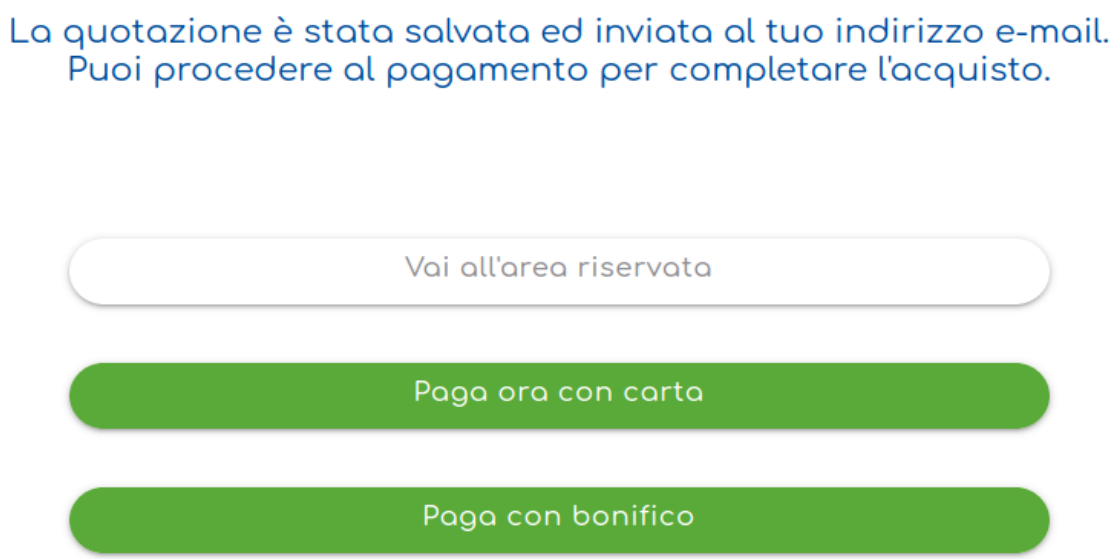


Figura 1.27 Thankyou page

3.3.6 Google Analytics e GTM

Figura 1.28 Google Analytics

La piattafroma utilizza gli script di Google Tag Manager per monitorare eventi che gli utenti generano durante il loro percorso all'interno del journey assicurativo. Principalmente vengono monitorati i click su alcuni pulsanti che sono considerati punti chiave per fornire dati statistici. Questi vengono poi mandati a Google Analytics, dove saranno poi oggetto di analisi per gli altri membri del team. Queste analisi successivamente hanno come obiettivo quello di andare a migliorare sempre di più la piattaforma, cercando di ottenere un "conversion rate" di utenti maggiore per quanto riguarda l'acquisto di una polizza assicurativa.
Dal punto di vista dello sviluppo, è stato necessario scrivere una serie di script e collegare gli eventi (click su pulsanti) della nostra piattaforma agli eventi che GTM e Google Analytics suggeriscono o che noi abbiamo creato. Questi script raccolgono i dati della pagina, li inviano a Google Analytics, fornendo così i dati statistici utili dal punto di vista strategico sulla piattaforma.

3.4 Assicurazione nel progetto Axidigital

Figura 1.29 Axidigital

Il lock-down provocato dalla pandemia COVID-19 ha fatto sì che molti broker e compagnie assicurative ancora poco "digitali", iniziassero a capire sempre di più l'importanza del web e quanto fosse più comodo utilizzare la rete internet per fornire dei servizi in tempo reale al cliente.
In questo contesto sono nate in Axieme nuove opportunità di business che hanno portato ad una revisione completa dello sviluppo e ad una trasformazione radicale dell'organizzazione del codice e della piattaforma. Non si doveva sviluppare solo più software per essere utilizzato internamente, ma con il progetto Axidigital, Axieme è diventata anche software-house, quindi il nostro software doveva diventare adatto per essere venduto anche a terzi. Quindi, sulla base del codice e del modello sviluppato fin a quel punto, si è analizzata la piattaforma per trovare il modo più veloce e scalabile per ovviare a questa trasformazione.
Il punto di partenza di questa analisi, è stato quello di andare a studiare i prodotti già digitalizzati fino a quel momento e cercare di classificarli con l'obiettivo di creare delle interfacce che permettessero al terzo la digitalizzazione di un prodotto (semplice) senza che ci sia necessità di scrivere codice da parte mia. Studiando tutti i prodotti, gli si è classificati in una serie di tipologie con difficoltà di digitalizzazione via via crescente. Dopo questa analisi, si è deciso che la piattaforma deve permettere autonomamente al terzo, la creazione di un prodotto (a pacchetto) di tre tipologie. Tutto quello che sta al di fuori di queste tipologie sono dei prodotti complessi che richiedono una digitalizzazione pressoché ad-hoc e probabilmente del codice aggiuntivo.
Successivamente per essere rapidi anche a digitalizzare i prodotti più complessi è stata sviluppata una nuova infrastruttura (database e codice) che lo permettesse, riducendo notevolmente i tempi di creazione di un prodotto e di un journey assicurativo. Tutto questo lavoro di digitalizzazione, inizialmente veniva fatto manualmente lavorando tanto sul database inserendo i dati nelle opportune tabelle, poi successivamente sono state create le interfacce con l'obiettivo di facilitare in primis il lavoro di inserimento e poi migliorarle per fornirle all'operatore e peremettere diretamente a lui la creazione di un prodotto.

Il risultato finale di tutte queste analisi è stato quello di dividere la piattaforma in più parti introducendo i servizi:

1. una serie servizi raggiungibili tramite API, i quali sono fornitori della struttura dati per la digitalizzazione e quotazione di un prodotto. Inoltre forniscono anche tutti i parametri per la creazione di tutte le pagine del journey (landing, chatbot, quotatore, registrazione);
2. una serie di utilizzatori di questi servizi: il loro compito è solo quello di fare il rendering delle varie pagine HTML e di interagire con l'utente finale;
3. altri servizi per la gestione delle anagrafiche e dell'emissione di polizza.

Il parametro principale da tenere a mente durante tutti i team meeting e le analisi è stato quello della "**scalabilità**", ovvero cercare di scrivere tutti questi servizi in modo da poter essere utilizzati da più progetti in parallelo e se l'indomani arrivasse un nuovo progetto, questo potesse essere gestito rapidamente in termini temporali, aggiungendo solo le nuove funzionalità.

3.4.1 Classificazione prodotti

Per quanto riguarda i prodotti a pacchetto, essendo che sono più semplici da digitalizzare, si è deciso di fornire ad un'operatore una serie di interfacce (il più semplici possibili) che gli permettessero autonomamente di inserire i pacchetti, le garanzie, i premi, i vincoli relativi e successivamente legarli tra di loro.
Questi prodotti sono stati classificati in 3 categorie con difficoltà crescente:

1. *pacchetto base di tipo 1*: caratterizzato da una complessità minima in quanto la digitalizzazione consiste nell'attribuire a ciascun pacchetto il premio, l'imposta da applicare e le garanzie contenute. Non presenta quindi nessun vincolo dovuto al chatbot.

Figura 1.30 Pacchetto tipo 1

2. *tipo 2*: introduce un massimale da selezionare nel pacchetto, quindi oltre ai parametri introdotti sopra, aggiunge uno o più massimali, quindi il premio viene ricavato incrociando il pacchetto e il massimale.

Figura 1.31 Pacchetto tipo 2

3. *tipo 3*: inizia a introdurre i vincoli del chatbot (per esempio il fatturato dell'azienda oppure l'età dell'assicurato). Quindi il calcolo del premio è l'incrocio del vincolo del chatbot, del massimale e del pacchetto scelto.

Figura 1.32 Fatturato indicato nel chatbot

Di conseguenza, nel quotatore il premio verrà filtrato sulla base del fatturato indicato, oltre al massimale.

Figura 1.33 Pacchetto tipo 3

Nel database, queste 3 tipologie di pacchetti trovano ciascuno una o più tabelle associate. Principalmente c'è una tabella che definisce i premi, una che tiene traccia di un elenco di massimali possibili (per i pacchetti tipo 2 e 3) e un'altra che definisce le fasce di fatturato,

età etc, all'interno delle quali ci sono determinati premi pre-stabiliti (pacchetto tipo 3 - le fasce dipendono dal prodotto).

- **Tabella premio**:

id	premio
1	50
2	75
..	...
10	150

- **Massimali**: elenca un certo numero di valori di massimale incontrati durante la digitalizzazione dei vari prodotti:

id	massimale
1	500
2	750
..	...
30	750.000
31	1.000.000

- **Range**: contiene per ciascun prodotto i valori minimi e massimi che identificano una fascia all'interno della quale ci sono certi premi associati:

id	minimo	massimo	prodotto_id
1	0	15.000	...
2	15.000	50.000	...
3	50.000	100.000	...
4	100.000	150.000	...
5	…	...	...

Oltre a queste 3 tabelle, ci sono altre tabelle ausiliarie per definire il pacchetto (nome, percentuale di imposta etc), le garanzie contenutevi e una tabella bridge (many to many) che associa a ciascun pacchetto le sue garanzie.

- **Pacchetti**: definisce per ciascun prodotto i pacchetti.

id	prodotto_id	nome_pacchetto	imposta_id	...
1	10	PACCHETTO TOP	1	...
2	10	PACCHETTO PREMIUM	1	...
..	...	...	...	...
10	15	**FIRST**	...	...

Per completare un pacchetto è stato necessario creare altre 3 tabelle bridge (una per tipo) con lo scopo di determinare le combinazioni pacchetto – premio – possibile massimale – ed eventualmente anche le fasce di fatturato (per esempio).

- **Legame tipo 1**: tabella molto semplice che lega il pacchetto di tipo 1 al suo premio.

id_paccheto	id_premio
1	1
2	2
..	...

- **Legame tipo 2**: ogni riga di questa tabella crea una possibile combinazione pacchetto – massimale – premio. Per esempio: il pacchetto con id =10 (**FIRST**), per il massimale con id = 30 (equivalente a 750.000 €) è associato il premio con id = 10 (150 €), invece per il massimale con id = 31 (=1.000.000 €) il premio è quello con id = 11 e così via.

id_paccheto	id_premio	id_massimale
10	10	30
10	11	31
..	...	...

- **Legame tipo 3**: come la precedente, ma la combinazione per ricavare il premio è data dalla tripletta massimale – pacchetto – fascia all'interno del quale il fatturato (per esempio) si pone.

id_paccheto	id_premio	id_massimale	id_range
15	15	30	1
15	16	31	2

I prodotti che non rientrano in queste 3 categorie richiedono spesso dello sviluppo in quanto introducono dei vincoli che gli altri non presentano, in particolari quelli con le garanzie selezionabili.
Inoltre, il problema principale per questi prodotti era che la struttura dati precedente era ancora inadatta, poco flessibile e poco scalabile per certi aspetti (troppo codice ripetuto e tempi di digitalizzazione ancora troppo alti che portavano solo ulteriore entropia nel codice). Allora si è implementato una soluzione del tutto nuova che riorganizzava sia la struttura dei moduli HTML che la quotazione in sé, introducendo per ogni garanzia tutti i parametri che la caratterizzano durante la quotazione (premio netto, massimale, imposta, accessori, modalità di calcolo e vincoli).

3.4.2 Sviluppo

Sulla base del codice sviluppato in precedenza e a seguito di diversi meeting lato IT, si è deciso di iniziare ad approcciare i micro-servizi e di riscrivere gran parte della piattaforma, utilizzando la versione 4 del framework CodeIgniter.
Sono stati sviluppati dei core fornitori di servizi (**Core Axidigital** per quanto riguarda la parte di gestione del journey e **Core Axieme e Core Builder** per la parte dedicata all'emissione di polizza e gestione utenza (entrambi in continuo sviluppo)).
Scopo di questi core è quello di raccogliere a fattor comune le funzionalità base che la piattaforma deve offrire (es. digitalizzazione di un prodotto), evitando così di coppiare ed incollare il codice. In questo modo la logica veniva spostata in un unico punto comune a tutti quelli che hanno bisogno di queste funzionalità.
Gli altri progetti (**Assicurazione, Backend, Badmin, Wallet**) non devono più occuparsi loro di gestire le funzionalità, replicando il codice ogni volta, ma si trasformano in utilizzatori dei servizi che i core offrono. Idealmente questi devono offrire solo le interfacce con cui l'utente deve interagire.

3.4.3 Infrastruttura di Axidigital

Il punto di partenza del progetto **Axidigital** è stato quello di rivedere la struttura del codice e riorganizzare il database sulla base dell'esperienza acquisita fino a quel momento. Avendo chiaro, più o meno, quali servizi la piattaforma deve offrire si sono rimodellate sia le tabelle del database che la struttura dati e i moduli HTML dei progetti. Inoltre, si è rivista anche la parte di quotazione, in quanto non potevamo più permetterci di scrivere ogni volta una funzione nuova la quale quotasse ciascun prodotto in parte, in quanto, il calcolo è spesso simile e le garanzie presentano più o meno tutte, le stesse caratteristiche (massimali, tassi, premi netti etc).
Analizzando i modi con cui si calcolavano i premi nei "*vecchi quotatori*", si sono definite diverse modalità di calcolo che sono state aggiunte come una delle proprietà che caratterizzano la garanzia, oltre al nome, al tasso, al massimale etc.

Altra caratteristica di cui si è tenuto conto è stato il fatto che la piattaforma dovesse essere esposta in modo tale da permetterci rapidamente di venderla ai terzi e si adatti facilmente alle esigenze e ai prodotti assicurativi che essi vogliono digitalizzare.
Per semplificare il modo in cui dovevamo fornire i servizi, internamente si è fatta una previsione su quali potrebbero essere i diversi acquirenti/utilizzatori della piattaforma, riuscendo in un certo senso a classificarli in diverse entità:

1. *compagnia assicurativa*: all'interno di questa catena si pone a livello più alto essendo la fornitrice dei prodotti e le percentuali di provvigioni, le quali le entità sotto si devono in qualche modo dividere;
2. *concessionario*: entità che per conto della compagnia è in grado di emettere le polizze e fornire i suoi prodotti ai distributori. Spesso sono individuati anche con il nome di gerenza della compagnia.
3. *distributori*: coloro che sono iscritti al RUI - Registro unico degli intermediari (ovvero quell'entità che tiene i dati dei soggetti che svolgono l'attività di distribuzione assicurativa e riassicurativa sul territorio italiano) e sono abilitati a digitalizzare i prodotti e venderli per conto della compagnia assicurativa o del concessionario;
4. *partner* dei distributori: i distributori possono stringere accordi commerciali con i partner, iscritti anch'essi al RUI oppure no, con l'obiettivo di publicizzare e vendere una polizza;
5. *agenti* dei partner o dei distributori: sono coloro che hanno il compito di vendere al cliente la polizza;
6. *cliente*: consumatore finale.

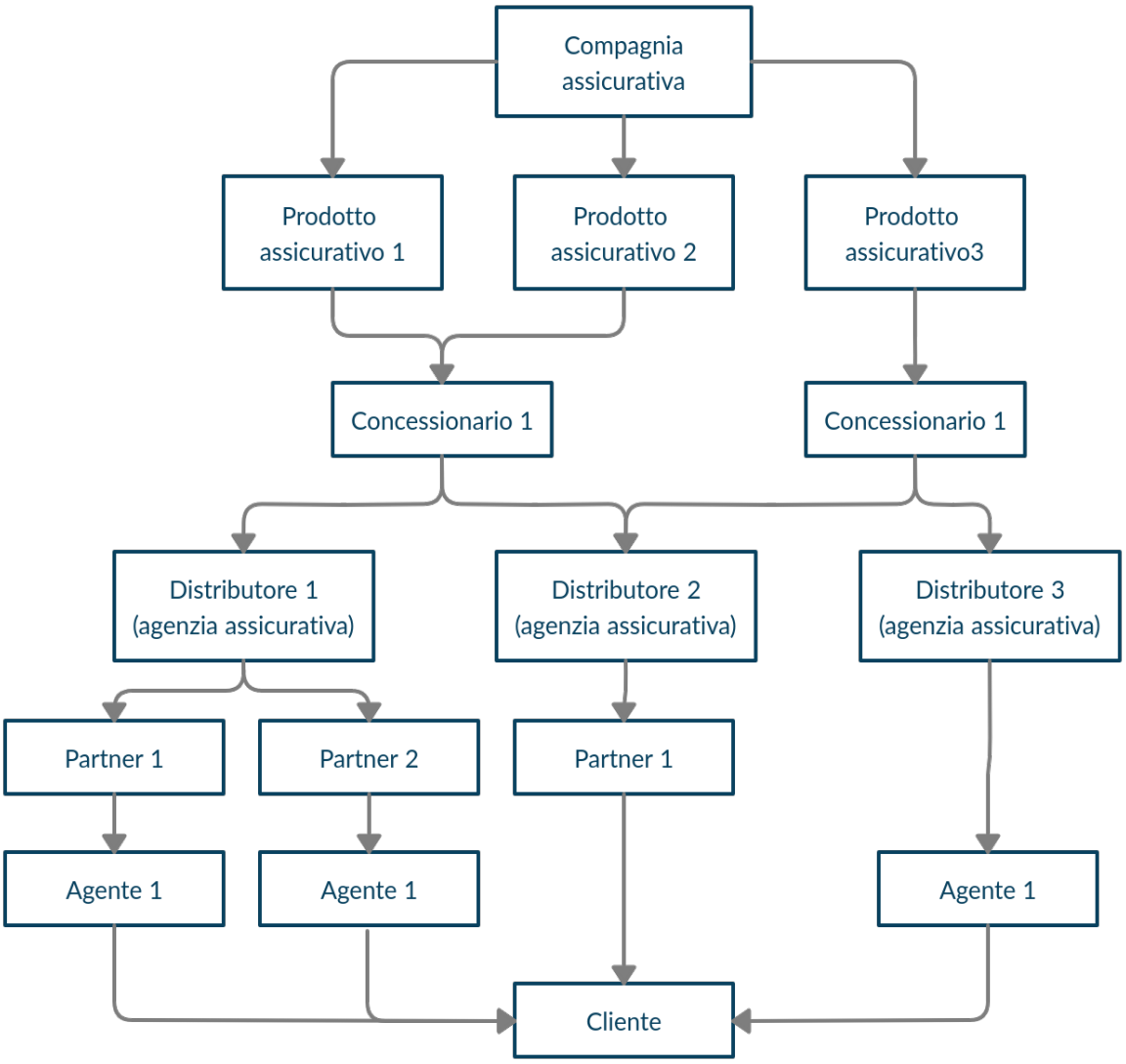


Figura 1.34 Catena di vendita Axidigital

3.4.3.1 Database e struttura dati della catena di vendita

Per supportare l'infrastruttura che **Axidigital** richiede, è stato necessario rimodellare sia la struttura dati che il database, aggiungendo nuove tabelle e modificare la logica di quelle già esistenti.
Logicamente il database lo si può vedere come se fosse diviso in 2 parti.

1. Una dedicata alla gestione del journey e dei prodotti e interagisce con il **Core Axidigital**. Questo costituisce un database condivisibile fra tutti coloro che acquistano l'uso della piattaforma. Nei prossimi esempi chiamerò questo database "**db_axidigital**".
2. Un database ("**db_axieme**") dedicato alla gestione dell'emissione della polizza e dell'utenza e interagisce con il **Core Axieme** per quanto riguarda il partner Axieme.

Ciascuna delle entità descritte nella catena di vendita ha una sua tabella nel database "db_axidigtal", nella quale vengono riportate le loro caratteristiche principali e le funzionalità di cui possono usufruire.

Distributore

Questa tabella contiene tutte le informazioni che descrivono un distributore oppure un'agenzia assicurativa, tra cui anche quelle che dimostrano l'abilitazione per poter vendere un prodotto asscicurativo (numero di iscrizione e data iscirizione al RUI - registro unico degli intermediari).

id	ragione_sociale	partita_iva	numero_rui	nome_rappresentante	...
1	Axieme	…	…	…	...

Partner

Questa è una delle tabelle più importanti in quanto definisce i parametri per fare il rendering delle pagine HTML del journey, quali colori, url, logo etc. Questa tabella è affiancata da altre 2 ausiliarie che definiscono altri parametri della pagina HOME del partner e di un'eventuale pagina CHI SIAMO, quali titolo, frasi, immagini.
Per default ogni distributore ha come partner sè stesso, definendo così sulla tabella distributore tutte quelle caratteristiche "burocratiche e amministrative" e sulla tabella del partner i parametri per creare il template del journey assicurativo (colori, logo, descrizioni etc). Il distributore può avere un numero di partner che dipende dai suoi accordi commerciali.

id	url	hex1	hex2	logo	...
1	/axieme	…	…	…	..

Legame tra Distributore e Partner

Questa tabella è quella che lega i distributori ai suoi partner, ai profotti che condividono tra di loro, alle provvigioni che possono ricevere ed eventualmente al Giveback©.

id_prodotto	id_distributore	id_partner	id_provvigione	id_giveback	...
1	1	1	1	1	…

Questa struttura viene replicata per tutte le altre entità (concessionario, compagnia e agenti) andando così a definire sia l'entità che il rapporto che essa ha con il prodotto e con chi sta sotto/sopra di lei nella catena.

Giveback© e Provvigioni

Le altre 2 tabelle molto importanti in questa catena sono quelle che definiscono le percentuali di Giveback© e le provvigioni che spettano a ciascuno.
Nella tabella delle provvigioni queste possono essere un valore fisso in euro (has_prov_fisse = 1) oppure un valore in percentuale che si applicano rispettivamente al premio netto e premio accessori.

id	has_prov_fisse	prov_fisse	prov_perc_netto	prov_perc_accessori
1	0	0	20	20

Dalla tabella del Giveback© si può ricavare la percentuale massima da applicare al premio imponibile per ricavare il Giveback© massimo.

id	perc_giveback_max	...	...	...
1	15	...	...	...

3.4.4 Assicurazione

Assicurazione anche nel progetto **Axidigital**, continua a rimanere quel progetto dedicato all'e-commerce dei prodotti assicurativi. Rispetto al progetto descritto nella sezione 3.3, le modifiche maggiori riguardano una diversa struttura del codice molto più pulita e più semplice anche da mantenere. E' cambiato anche il modo in cui vengono recuperati e salvati i dati (interagendo con i core tramite API) e di conseguenza come questi vengono renderizzati nelle pagine HTML.
Le pagine con cui l'utente interagisce continuano a rimanere le stesse (landing, chatbot, quotatore, registrazione, thankyou page) e ad eccezione della pagina del quotatore, queste vengono caricate con la stessa logica e struttura delle tabelle nel database descritte nella sezione 3.3.
Nelle prossime pagine descriverò in modo più approfondito come la struttura del quotatore è stata modificata, evidenziando i benefici che questa ha portato e utilizzerò come esempio di prodotto non a pacchetto, ma garanzie selezionabili, "*CASA*".

3.4.4.1 Quotatore – struttura database prodotti complessi

Lo sviluppo del quotatore ha avuto come requisito primario quello di avere una struttura scalabile che permettesse con poche righe di codice di gestire la quotazione di un prodotto a garanzie complesso. Per ottenere questo risultato è stato necessario rivedere la parte di database destinata alla gestione del prodotto, spostando così gran parte del lavoro nello strato di persistenza. In particolare analizzando i prodotti, si sono estrapolate le sue caratteristiche principali e si è cercato di tradurle in tabelle, arrivando alle seguenti conclusioni:

- ogni prodotto è caratterizzato da una serie di garanzie;
- ogni garanzia appartiene ad una sezione (incendio, furto, rc etc) che le raccoglie sulla base di caratteristiche comuni;
- ciascuna garanzia è caratterizzata spesso da un massimale, un tasso oppure un premio netto per quanto riguarda il calcolo - questo ha fatto sì che si potesse definire una serie di modalità di calcolo comuni a più garanzie e riutilizzabile per tutte quelle che presentano certe caratteristiche;
- una garanzia può dipendere da altre garanzie sia dal punta di vista del calcolo del premio che esistenza - non posso selezionare una garanzia facoltativa senza aver selezionato anche la garanzia base a cui è legata.
- una garanzia facoltativa può essere inclusa già in un'altra base, quindi selezionando la base si deve avere anche la facoltativa.

Lo sforzo più grande che si è fatto durante lo sviluppo è stato quello di cercare di tradurre in codice e tabelle tutte queste informazioni ottenendo così una struttra dati adatta alla digitalizzazione di un prodotto complesso a garanzie.

Partendo dai documenti che la compagnia fornisce, in particolare il file che contiene le norme tariffarie, si individuano dapprima tutte le sezioni, le garanzie che queste contengono, le aliquote e le percentuali accessorie da applicare nel calcolo.
Queste vengono salvate sul database "**db_axidigital**" in una serie di tabelle e legate tra di loro con delle tabelle "many to many".

- **Sezioni**: contiene le informazioni sulle sezioni e i valori di imposta e accessori da utilizzare per ricavare il premio lordo di ciascuna garanzia.

id	nome	imposta	accessori
1	INCENDIO E DANNI	1.2225	1.1
2	ELETTRONICA	1.2125	1.1
3	FURTO	…	...
4	…	…	...

- **Legame sezione prodotto**: questa tabella lega il prodotto alle sue sezioni e al distributore, definendo anche l'ordine di apparizione nel quotatore.

id_distributore	id_prodotto	id_sezione	ordine_visualizzazione
1	1	1	1
1	1	2	2
1	...	...	...

Per definire meglio il prodotto è necessario ora descrivere come vengono inserite le garanzie all'interno delle sezioni e quali sono le loro caratteristiche ai fini di permettere la quotazione.

- **Garanzie**: raccoglie un elenco di tutte le possibili garanzie incontrate durante la digitalizzazione di diversi prodotti.
  Le informazioni da sottolineare in questa tabella sono quelle che ne descrivono la tipologia, ovvero se si tratta di una garanzia che presenta un massimale selezionabile (select_massimale) oppure no (semplice), un'eventuale imposta da applicare che potrebbe essere diversa da quella delle sezione. Importante anche la colonna **model**, ovvero quel parametro che definisce in fase di creazione del quotatore, per le garanzie con dei massimali selezionabili quale tabella utilizzare per recuperare il massimale e il premio netto associato.

id	nome	imposta	parametro	tipologia	model
1	INCENDIO E DANNI FABBRICATO	NULL	incendio_fabbricato	select_massimale	NULL
2	INCENDIO E DANNI CONTENUTO	NULL	incendio_contenuto	select_massimale	NULL
3	FENOMENO ELETTRICO	NULL	fenomeno_elettrico	semplice	NULL
4	…	…	...	massimale fisso	...
5	...	...	tutela_legale	select_massimale_vincolata	TutelaLegaleModel

Di seguito, un esempio di utilizzo della colonna **“model”**:
Nel documento tariffario che la compagnia fornisce, i premi di una certa garanzia possono essere definiti in forma tabellare, associando a ciascun massimale il relativo premio netto.
Per quella garanzia, nel database viene definita una tabella ad-hoc, utilizzando uno standard pre-stabilito per nominare le colonne dei premi, ovvero ciò che segue la parola “*premio_netto_*”, deve essere uguale a quello che viene scritto nella colonna *parametro* sulla tabella delle **“Garanzie”**, in questo caso “*tutela_legale”*.
Così facendo, nel momento in cui vengono recuperate tutte le garanzie chiamando un end point che **Core Axidigital** fornisce, nel caso della “*select_massimale_vincolata*”, si utilizza il model indicato (TutelaLegaleModel in questo caso) per recuperare l’elenco dei massimali e premi netti relativi e li si aggancia direttamente alla garanzia in questione (Tutela legale).

id	massimale	premio_netto_**tutela_legale**	default_massimale
1	10.000	40	1
2	15.000	50	0
...	...	...	...

Dal punto di vista del modulo HTML nella pagina del quotatore, questo viene tradotto come una select che visualizza tali massimali.

Figura 1.35 Modulo HTML

- **Legame sezione garanzia:** questa tabella attribuisce a ciascun distributore un prodotto e per ogni sezione definisce le garanzie di cui è composta.

id_distributore	id_prodotto	id_sezione	id_garanzia	vincolato_a	incluso_in
1	1	1	1	NULL	NILL
1	1	1	2	NULL	NULL
1	1	1	3	1	NULL
1	1	1	4	NULL	2
...	...	...	...	...	...

Le colonne *"vincolato_a"* e *"incluso_in"* esprimono il concetto di vincolo tra garanzie e inclusione. Prendendo l'esempio in questione, per quanto riguarda la garanzia **3**, è vincolata alla garanzia 1, quindi nel quotatore nel momento in cui viene selezionata la garanzia facolativa 3, viene obbligatoriamente inclusa anche la garanzia 1. Di conseguenza, vale anche il concetto inverso, cioè quando viene deselzionata lagaranzia base 1, viene rimossa anche la garanzia facoltativa 3.
Lo stesso ragionamento viene fatto anche per il vincolo di inclusione. In questo caso, la garanzia 4 viene selezionata sempre quandola garanzia 2 viene inclusa, e a differenza del vincolo di prima, la garanzia facoltativa non può mai essere deselzionata.

Questa tabella definisce solo alcune delle caratteristiche del prodotto e delle sue garanzie. Successivamente, ci si è accorti che non bastava solo lei, ma che fosse necessario introdurre per ogni prodotto, una nuova tabella parallela che definisca in modo più approfondito una garanzia. Quindi per ogni prodotto complesso viene creata una nuova tabella parallela con l'obiettivo di raccogliere tutte le informazioni e i parametri necessari durante la fase di quotazione.

- **Legame sezione garanzia del prodotto in questione:** questa tabella definisce tutti gli altri parametri (tassi, premi etc) utilizzati in fase di quotazione e in particolare la modalità di calcolo da utilizzare per ricavare il premio lordo. La colonna “select_id” fa riferimento ad una tabella ausiliaria (select_massimali) che tiene traccia dei valori dei massimali da mostrarre nella pagina del quotatore per la select di quella garanzia.

id_sezione	id_garanzia	modalita_calcolo	tasso	premio_netto	select_id	...
1	1	1	0.10	NULL	1	...
1	2	2	0.15	NULL	2	...
1	3	2	NULL	15	NULL	...
1	4	3	...	...	...	...

- **Select massimali**: tabella che definisce per ogni garanzia, il valore iniziale del massimale, il suo valore finale, il passo e il valore di default.

id	valore_iniziale	valore_finale	passo	valore_default
1	50.000	500.000	10.000	100.000
2	10.000	100.000	50.000	50.000
3	...	...	...	...

Le tabelle elencate fino a questo punto sono quelle più importanti per il processo di digitalizzazione. Oltre a queste, spesso vengono aggiunte altre ausiliarie che dipendono dal prodotto in sé e dalla sua complessità. Quelle più frequenti sono quelle relative a contenere i premi e gli eventuali tassi ad-hoc che le garanzie presentano. La creazione di una nuova tabella segue comunque delle regole pre-stabilite, in modo da poterle utilizzare facilmente con quello che è stato sviluppato (es. standard da seguire nelle tabelle riservate ai premi: la colonna **premio_netto_”*garanzia*”**, dove ***garanzia*** fa riferimento al nome del parametro scritto nella tabella delle **Garanzie**).

3.4.4.2 Quotatore – pagina della quotazione

Definita gran parte della struttura del database, il punto successivo è stato quello di tradurla in una struttura dati efficiente da poter essere utilizzata nella quotazione. Il punto di partenza è stato quello di rivedere i moduli HTML, cercando di non avere più tanti moduli per ciascuna tipologia, ma uno solo che li racchiuda tutti.
Per agevolare la quotazione e renderla indipendente dal prodotto e dalla garanzia in sé la struttura dati viene costruita già nel HTML, utilizzando la sintassi dei tag <input> che pemette di creare degli array associativi utilizzando gli attributi “name” e “value”.

Esempio modulo HTML:

```
<div>
        <input type=”checkbox” id=”checkbox_<?= parametro?>” name=”garanzie[
        <?= $sezione ?>][<?= $parametro ?>][checkbox] value=”1” />
        <?php
                // eventuale stampa dei massimali utilizzando un ciclo for per stampare
                // le options della select
        ?>
        <div id=”premio_<?= $parametro ?>”>
                // div riservato alla stampa del premio
        </div>

        <input type=”hidden”  name=”garanzie[ <?= $sezione ?>][<?= $parametro ?>]
                [premio_netto] value=”<?= $premio_netto ?>” />
        <input type=”hidden”  name=”garanzie[ <?= $sezione ?>][<?= $parametro ?>]
                [tasso] value=”<?= $tasso ?>” />
        <input type=”hidden”  name=”garanzie[ <?= $sezione ?>][<?= $parametro ?>]
                [modalita_calcolo] value=”<?= $modalita_calcolo ?>” />

        // altri input di tipo hidden contenenti altri parametri necessari al calcolo
</div>
```

Il modulo è sempre composto da una checkbox, un’eventuale select per il massimale e una serie di input hidden che tengono traccia dei tassi, premi netti da utilizzare nel calcolo. Dal punto di vista di cosa l’utente vede, non cambia niente rispetto alla sezione 3.3, ma quello che effettivamente cambia è la logica di calcolo e la struttura dati utilizzata.

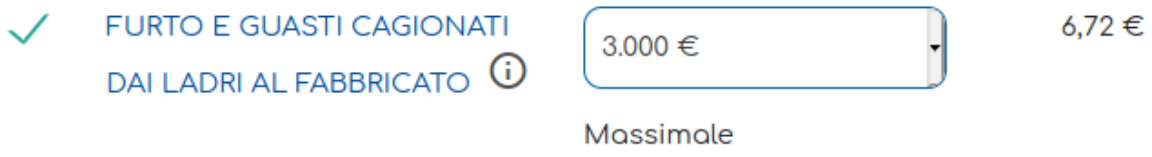


Figura 1.36 Modulo HTML

3.4.4.3 Quotatore – struttura dati

La struttura dati che si viene creare già nel HTML è la seguente:

```
garanzie
   [‘incendio_danni’]                // nome della sezione
      [‘incendio_contenuto’]         // nome della garanzia
         [‘checkbox’] = 1            // valore checkbox → 1=selezionata, 0=non selezionata
         [‘massimale’] = 100000      // massimale della garanzia
         [‘modalita_calcolo’] = 1    // modalità di calcolo
         [‘tasso’] = 0.10            // tasso da utilizzare nella formula
         [‘parametro’] = ”incendio_contenuto”
         [‘imposta’] = …
         [‘accessori’] = ...
      [‘impianti_fotovoltaici’]
         [‘checkbox’] = 0
         [‘massimale’] = 15000
         [‘modalita_calcolo’] = 2
         [‘premio_netto’] = 15
         [‘parametro’] = ”impianti_fotovoltaici”
         [‘imposta’] = …
         [‘accessori’] = …
 [‘furto’]                          //  nome della sezione
    [‘furto_e_guasti’]              //  nome della garanzia
         [‘checkbox’] = 1
         [‘massimale’] = 30000
         [‘modalita_calcolo’] = 2
         [‘premio_netto’] = 20
         [‘parametro’] = ”furto_e_guasti”
         [‘imposta’] = …
         [‘accessori’] = …
      …
 [‘responsabilita_civile’]      //  nome della sezione
    …
 [‘tutela_legale’]              //  nome della sezione
    …
 [‘assistenza’]                 //  nome della sezione
    …
```

Si tratta di un array associativo che elenca per ogni sezione, le garanzie di cui è composta e per ogni garanzia indica quali sono i parametri che la caratterizzano (premio netto, tasso etc) indicando per ciascuna una modalità di calcolo.

3.4.4.4 Quotatore – modalità di calcolo

Utilizzando questa struttura dati, si è definita una libreria composta da un solo metodo pubblico *"quotaGaranzie()"* invocato sempre tramite una chiamata AJAX ogni volta che l'utente interagisce con le garanzie. Questo metodo è molto semlice e non fa altro che fare un doppio ciclo for per scorrere la struttura dati citata sopra; quello più esterno per scorrere le sezioni del pordotto e quello più interno per scorrere le garanzie di ciascuna sezione.
All'interno di esso troviamo un switch-case sulle diverse modalità di calcolo per invocare un metodo privato adatto al calcolo del premio della garanzia interessata.

```
foreach ( $garanzie as $sezione => $elenco_sezioni ) {
    foreach ( $elenco_sezioni as $garanzia => $parametri_garanzia ) {
        if ( $parametri_garanzia['checkbox'] == 1 ){
            switch ( $parametri_garanzia['modalita_calcolo'] ) {
                case 1:
                    $premio[$garanzia] = self::calcolaPremio(…);
                    break;
                case 2:
                    $premio[$garanzia] = self::calcolaPremioDatoIlPremioNetto(…);
                    break;
                …

                default:
                    break;
            }
        }
    }
}
```

Le modalità di calcolo principali rimangono sempre quelle descritte nelle sezioni precedenti alle quali vengono aggiunte ulteriori casistiche dipendenti dalle diverse garanzie:

- modalità di calcolo 1:

  $premio\ lordo = [\ (\ massimale \cdot tasso / 1000\ ) + una\_tantum\ ] \cdot accessori \cdot imposta \cdot sconto$

- modalità di calcolo 2:

  $premio\ lordo = premio\_netto \cdot accessori \cdot imposta \cdot sconto$

- altre modalità: dipendono dalla garanzia in sé e spesso riguardano le garanzie facoltative in quanto devono utilizzare i massimali delle garanzie base acui sono legare oppure ricavare il tasso applicando una determinata percentuale (tutte queste informazioni vengono estrappolate dal documento tariffario che la compagnia fornisce).

3.4.4.5 Registrazione

Questa pagina si presenta alla fine del journey. Per quanto riguarda la costruzione del form e la validazione dei campi lato client, logicamente non è cambiato niente rispetto a quello descritto nella sezione 3.3.4, quello che è cambiato invece è la logica di salvare i dati.
Essa non viene più fatta direttamente da questo progetto, ma è demandata al **Core Axieme** e **Core Builder**, due servizi sottostanti che hanno accesso allo strato di persistenza. Il controller espone due metodi pubblici:

1. il primo invocato nel passaggio dalla pagina dal quotatore alla pagina di registrazione che deve validare il premio e salvare in sessione i dati della quotazione. Per fare questo, nel caso dei prodotto a garanzie, invoca un metodo privato che cicla su di esse e salva per ognuna tutte le sue caratteristiche (nome, massimale, premio e id) ottenendo la seguente struttra dati:

```
{
  “incendio_fabbricato” :
    {
        “premio_lordo” : 10,
        “id_garanzia” : 1,
        “massimale”: 100000,
        “nome_garanzia” : “INCENDIO E DANNI FABBRICATO”
    },
  “incendio_contenuto” :
    {
        “premio_lordo” : 15,
        “id_garanzia” : 2,
        “massimale”: 150000,
        “nome_garanzia” : “INCENDIO E DANNI CONTENUTO”
    },
  ...
}
```

   Per i prodotti a pacchetto, in sessione viene salvato solo l’id del pacchetto selezionato durante la quotazione e l’eventuale massimale.

2. Il secondo raccoglie tutti i dati dalla POST della pagina di registrazione e dalla sessione, costruisce un oggetto JSON e chiama il **Core Axieme** per costruire un nuovo ordine.

3.4.5 Core Axidigital

Nel momento in cui i progetti sono diventati numerosi ci si è presto accorti che spesso ci ritrovavamo nella situazione in cui dovevamo copiare ed incollare lo stesso codice sui vari progetti. Allora in questo contesto è nata l'idea di avere un punto centralizzato che gestisca queste logiche comuni creando il **Core Axidigital**.
Il core axidigital è quel servizio che si occupa di fornire tutti gli end point necessari per quanto riguarda la gestione del journey assicurativo (es. tutti i parametri per creare una landing) e digitalizzazione di un prodotto (recuperare le garanzie con tutte le sue caratteristiche). Esso si pone come uno strato intermedio tra il database "**db_axidigital**" e tutti i progetti che hanno bisogno di questi servizi.
E' stato sviluppato utilizzando CodeIgniter 4 ed espone come API REST una serie di end point, i quali internamente vengono mappati su metodi all'interno dei controller.

3.4.5.1 Esempio di end point

Essendo fornito come API REST, all'esterno viene fornita una rotta del tipo /prodotto/{id} che ha come scopo, quello di ritornare un prodotto con le sue caratteristiche dato il suo id (chiave primaria della tabella).
Questa rotta viene mappata sul metodo find() del controller Prodotto. Solitamente nella verisone 4 del framework la prima riga di ogni controller individua il model con cui quel controller deve interagire di più (in questo caso ProdottoModel) per recuperare i dati dal database.

All'interno del file di configurazione Route.php vengono inserite le rotte che devono essere accessibili dall'esterno:

```
$routes→get("prodotto/(:num)", "Prodotto::find/$1");
```

Il significato di questa riga indica che sulla url prodotto/1 invocata nel browser come GET ritorna al chiamante le informazioni sul prodotto con id = 1 se esiste, altrimenti ritorna il codice *404 NOT FOUND* e un messaggio di errore.
Per ottenere questo risultato il controller interessato è *Prodotto* e il metodo da invocare è *find()* passandoli come parametro l'id ricavato dall'url.

- Esempio di **Controller**:

```
class Prodotto extends Resource {

  protected $modelName = “App\Models\ProdottoModel”;   // definizione del model

  public function find($id) {
        $result = $this→model→select(...)
                        →join(...)
                        →where(“id”, $id);

        if ($result == null) {
                return $this→respond(
                                [“code” => 404,
                                 “message” => “...”
                                ],
                                Response::HTTP_NOT_FOUND
                                );
        }

        return $this→respond(
                        [“code” => 200,
                         “result” => $result
                        ],
                        Response::HTTP_OK
                        );
        }
}
```

- Esempio di **Model**:

```
class ProdottoModel extends Model {

        protected $table = “prodotto”;              // tabella a cui fare riferimento sul db

        protected $primaryKey = “id”;                   // chiave primaria della tabella

        protected $allowedFields = [“id”, ……];      // colonne presenti nella tabella

        protected $validationRules =
                [“nome_prodotto” => “required”];  // regole di validazione sui campi

}
```

L’end point più importante e complesso di questo servizio è quello che ritorna tutte le garanzie dato un prodotto e un distributore. Questa risorsa viene esposta sia in GET che POST (per quei casi in cui è necessario passare alcuni filtri da applicare alle caratteristiche delle garanzie).

```
$routes→get(“quotatore/garanzie/(:num)/(:num)”, “Prodotto::findGaranzie/$1/$2”);
$routes→post(“quotatore/garanzie/(:num)/(:num)”, “Prodotto::findGaranzie/$1/$2”);
```

Il risultato ritornato da questo servizio è un oggetto JSON contenente per ogni sezione le garanzie di cui è composto e per ciascuna ritorna tutte le caratteristiche necessarie al calcolo del premio (massimali, tassi, premi etc).

```
{
    0 : {
        “nome_sezione” : “INCENDIO E DANNI”,
        “imposta” : … ,
        “accessori” : … ,
        “garanzie” : {
              0 : {
                 “nome_garanzia” : “INCENDIO E DANNI FABBRICATO”,
                  …
                 “caratteristiche_modulo” : {
                       “tasso” : 0.16,
                        “premio_netto” : null,
                         ...
                    }
               },
              1 : {
                 “nome_garanzia” : “INCENDIO E DANNI CONTENUTO”,
                  …
                 “caratteristiche_modulo” : {
                       “tasso” : 1.46,
                        “premio_netto” : null,
                         ...
                    }
               }
          }
     },
     1 : {
        “nome_sezione” : “CRISTALLI”,
         ...
    }
}
```

3.4.6 Core Axieme

Questo servizio è uno strato intermedio tra diversi progetti (**Assicurazione** e **Backend**) e il database la cui funzione primaria è quella di normalizzare i dati prima di inserirli nelle tabelle (per esempio, **Assicurazione** che richiede la creazione di un ordine) e fornire tali dati quando qualcuno ne ha bisogno, per esempio **Backend** o **Wallet**.
È stato sviluppato anch'esso utilizzando la versione 4 del framework CodeIgniter e logicamente espone una serie di end point raggiungibili solo all'interno della stessa macchina.

3.4.6.1 Autenticazione ed integrità

Essendo che l'end point di creazione di un ordine uno pubblico, per evitare problemi di sicurezza si è introdotto il calcolo del HMAC per avere autenticazione ed integrità dei messaggi. Quindi il progetto **Assicurazione** prima di inoltrare la chiamata calcola e inserisce nella richiesta anche l'HMAC applicato ad alcuni campi dell'oggetto JSON.
La richiesta viene intercettata da un filtro del Framework, il quale si occupa si calcolare l'HMAC sui medesimi campi usati **Assicurazione** e se il risultato è lo stesso inoltra la chiamata al Controller, altrimenti notifica l'errore al chiamante.

3.4.6.2 Creazione di un ordine

L'end point più complesso del **Core Axieme** riguarda la creazione di un ordine. In ingresso esso riceve un oggetto JSON contenente i dati della quotazione e le informazioni anagrafiche dell'utente che ha compilato il form di registrazione a fine journey.
Il suo compito consiste nel normalizzare questo oggetto JSON applicando una serie di filtri. Il risultato di tali filtri sono una serie di altri oggetti JSON mappati ciascuno una tabella di interesse nel database. Per la creazione di un ordine le tabelle rimangono sempre quelle descritte nella sezione 3.3.4 (utente, anagrafica, polizza, contratto etc). Quindi il risultato dei filtri sono tanti oggetti quante sono le tabelle interessate.
Oltre a normalizzare i dati, questo metodo si occupa anche di definire la configurazione del database nel quale inserirli. La chiamata arrivando dal progetto **Assicurazione** di Axieme, il core andrà a settare come database quello di **Axieme** (**db_axieme**). Se invece essa arrivasse da un progetto **Assicurazione** di un partner, il database settato sarebbe quello del partner in questione.
Di seguito, un esempio di oggetto JSON che l'end point può ricevere a fine journey. Esso porta le informzioni della persona che ha compilato il form (nome, cognome, email), più quelle per quanto riguarda le risposte date nel chatbot e infine tutti i dati della quotazione (premio lordo, garanzie selezionate etc).

```
{
   “nome” : “Mario”,
   “cognome” : “Rossi”,
   “email” : “mario.rossi@email.com”,
   “premio_lordo_totale” : 100,
   …
   “risposte_chatbot” :
       {
         1 : …          // la chiave dell’object indica lo step del chatbot
         2 : …          // il value indica la risposta data
         ...
       },
  “garanzie” :
       {
         “incendio_fabbricato” :
             {
                 “premio_lordo” : 10,
                 “id_garanzia” : 1,
                 “massimale”: 100000,
                 “nome_garanzia” : “INCENDIO E DANNI FABBRICATO”
             },
         “incendio_contenuto” :
             {
                 “premio_lordo” : 15,
                 “id_garanzia” : 2,
                 “massimale”: 100000,
                 “nome_garanzia” : “INCENDIO E DANNI CONTENUTO”
             },
         ...
       }
}
```

Su questo oggetto vengono applicati i filtri. Ognuno, non è altro che un’array le cui chiavi fanno riferimento alle chiavi che devono estrarre dall’oggetto sopra citato per costruirne uno nuovo con solo i dati della tabella interessata (per esempio, per la tabella utente verrano estratti solo i dati dell’utente (email, telefono, etc)).

- filter_utente = array(‘email’, ‘telefono’, …)
- filter_anagrafica_fisica = array(‘nome’, ‘cognome’, ‘codice_fiscale’, ...)
- filter_contratto = array(‘premio_lordo’, ‘premio_netto’, ..)

Una volta applicati i filtri e settate le dovute configurazioni, **Core Axieme** può far la richiesta al **Core Builder** passandoli l’oggetto che i filtri hanno creato, più la

configurazione del database di interesse e le tabelle da utilizzare. Inoltre per ciascuna tabella indica se ci sono oppure no dei campi da cifrare prima di effettuare l'inserimento.

```
{
   "database" : "db_axieme",
      …
   "utente" :
      {
         "table": "utente",
         "params_to_crypt": ["telefono"],
         "data" :
            {
                "email" : "mario.rossi@email.com",
                "telefono" : …
                   …
            }
      },
  "anagrafica" :
      {
         "table": "anagrafica_fisica",
         "params_to_crypt": ["nome", "cognome", ...],
         "data" :
            {
                "nome" : "Mario",
                "cognome" : "Rossi",
                   …
            }
      },
   "contratto" :
      {
         "table": "contratto",
          ...
      },
    "garanzie" :
      {
        ...
      },
    "risposte_chatbot" : {}
}
```

Ricevuta la risposta dal **Core Builder**, inizia la seconda fase durante la quale si deve notificare al utente l'avvenuto ordine, allegandoli nella mail anche i dovuti documenti (dipendenti dal prodotto). Quindi il **Core Axieme** mette in pratica la stessa procedura di creazione dei documenti e loro inserimento su MongoDB descritta nella sezione 3.3.4.7.

3.4.7 Core Builder

Questo servizio è lo strato che comunica con il database. L'idea generale che sta dietro a questo core è quella di avere un punto centralizzato che svolge le operazioni base sul database (insert, select, update) indipendentemente dal progetto utilizzatore che lo sta chiamando. Per ottenere questo risultato è necessario comunque configurarlo con una serie di parametri che sempre l'utilizzatore deve fornirgli in ingresso. Quindi, oltre ai dati da inserire/estrarre dalle tabelle, riceve sempre nello stesso oggetto JSON anche la configurazione del database e quali tabelle utilizzare nelle transazioni.
Il vantaggio di aver sviluppato anche questo strato è stato principalmente quello di aver evitato la riscrittura dello stesso codice sui differenti progetti. Spesso ci si trovava nella situazione in cui dovevamo copiare e incollare il codice per la creazione di un ordine. Questo servizio, andando a centralizzare quest'operazione, ha portato il beneficio di avere un solo punto in cui poterlo farlo. Per contro, invece ha introdotto del lavoro aggiuntivo nei core normalizzatori (**Core Axieme** per esempio), che deve applicare i filtri e fornire al **Core Builder** i dati organizzati in una determinta struttura dati, più anche la configurazione del database.
Per la creazione dell'ordine, il lavoro che il builder svolge è quello di scorrere la struttura dati che riceve e invocare i metodi per validare i campi e se la validazione va a buon fine chiamare le insert nelle varie tabelle. Inoltre, in alcuni casi deve anche completare la struttura delle colonne che deve inserire nel database, per esempio convertire il nome delle province e dei comuni nei rispettivi id (primary key). Per fare questo fa una chiamata ad un altro servizio (**Core Axidigital**), il quale ricevendo il nome della provincia o il nome del comune, ritorna gli id associati.

Oltre alle insert, il **Core Builder** ha altre classi utilizzate per fare l'update dei dati sul database e le relative select (più o meno complesse a seconda del numero di join che si devono eseguire). Anche in questi casi l'approccio adottato è stato quello di rendere trasparente del tutto il fatto che al builder non interessi chi è l'entità che gli sta richiedendo il servizio. Quindi deve ricevere sempre dal chiamante i parametri di configurazione del database e in alcuni casi le tabelle interessate dalle query.

3.4.8 Architettura a Microservizi

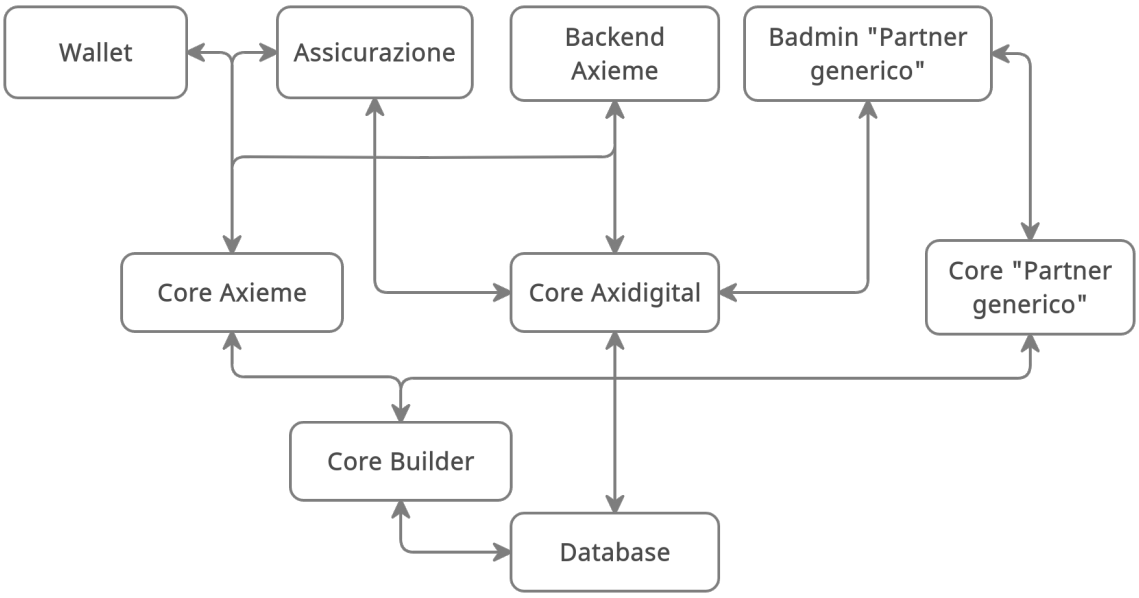


Figura 1.37 Architettura a Microservizi

Questo semplice schema mostra come i diversi progetti nell'architettura a microservizi comunicano tra di loro. Partendo dal basso abbiamo:

- Database: esso comunica con il Core Builder per quanto riguarda la gestione di utenze, polizze etc, e con il Core Axidigital per la digitalizzazione di prodotti assicurativi e gestione dei partner.
- Core Builder: interfaccia verso il database per tutti i progetti utilizzatori (per esempio: Core Axieme che richiede qualche operazione CRUD).
- Core Axieme: mette a disposizione tutti gli end point per gestire polizze, utenze etc. Il più importante è quello che riguarda la creazione di un ordine di polizza utilizzato dal progetto Assicurazione.
- Core Axidigital: contiene tutti gli end point per gestire prodotti e partner e comunica direttamente con il database.
- Wallet, Assicurazione, Backend: progetti utilizatori.
- Per concludere, ci sono i progetti degli altri partner che seguono questo schema, ovvero Assicurazione, Wallet e Badmin sono progetti utilizzatori di servizi messi a disposizione dai vari core.

3.5 Badmin

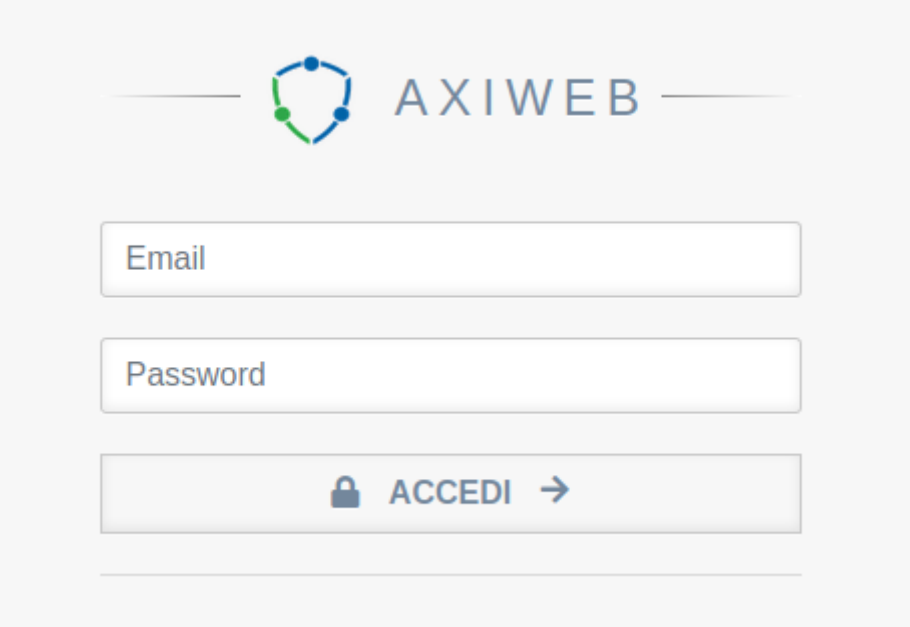


Figura 1.38 Login Badmin

Questa parte della piattaforma è stata sviluppato per essere venduta ai broker assicurativi con l'obiettivo primario di permettere a questi di configurarsi il sito, digitalizzare autonomamente un prodotto e poi successivamente gestire gli ordini che arrivano dal journey.
Lo sviluppo di questa parte della piattaforma è stato semplificato molto dall'esperienza acquisita precedentemente e dal fatto che si era giunti a questo progetto con delle idee molto chiare. Inoltre l'introduzione dei vari servizi fornitori (i core) ha fatto sì che la logica funzionale non venga fatta qua, ma direttamente su di essi. Quindi l'unico compito del Badmin era quello di raccogliere è dati, inviarli al core e poi con le risposte ricevute popolare le viste principali.

3.6 Wallet

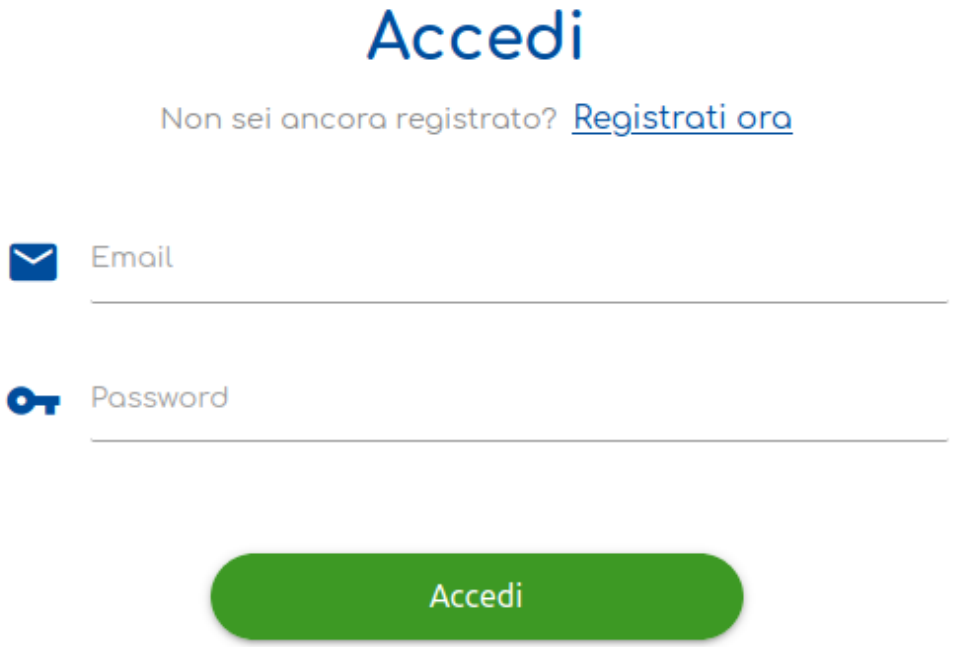


Figura 1.39 Login Wallet

Quest'area è quella riservata all'utente, utilizzatore della piattaforma nella quale può accedere con le credenziali ricevute in seguito alla creazione di un ordine, per controllare principalmente lo stato delle sue polizze attive.
In questo portale esso trova anche altre funzionalità, quali:

- possibiltà di pagare/rinnovare una polizza, utilizzando la piattaforma Nexi;
- comunicare con Axieme per modifiche sulla piattaforma;
- richiedere l'apertura di un sinistro;
- scaricare le quietanze del pagamento (in formato pdf);
- visualizzare, sempre in formato pdf, il contratto che la compagnia assicurativa fornisce in seguito all'emissione di polizza;
- richiedere di riscuotere il Giveback alla scadenza della polizza;
- possibilità di donare il Giveback ad alcune ONLUS con cui Axieme ha stipulato dei rapporti.

Capitolo 4

Infrastruttura e performance della piattaforma

Tale capitolo è dedicato a misurare le performance che la piattaforma offre facendo anche alcune considerazioni sui risultati ottenuti utilizzando i servizi di Aruba e quelli di Azure.

4.1 Infrastruttura

La raggiungibilità della piattaforma su internet da parte degli utenti è realizzata grazie a due servizi terzi:

- *Aruba*: per quanto rigurada il dominio pubblico di Axieme e di quei partner che non hanno bisogno di uno proprio, ma utilizzano quello che la società mette a disposizione.
- *Azure*: per tutti quei partner che hanno richiesto esplicitamente l'utilizzo di questa piattaforma come servizio e vogliono avere il proprio dominio.

L'obiettivo di questo capitolo è quello di descrivere brevemente le architetture di questi due servizi e metterli a confronto misurando i tempi di risposta che offrono sulle varie pagine della nostra piattaforma, confrontando anche l'applicativo inizialmente sviluppato come monolita ed eseguito su Aruba piuttosto che micro-servizi ed eseguito sia su Aruba che Azure.

4.2 Aruba

Figura 1.40 Aruba

Aruba è una società italiana che offre servizi di data centre, web hosting, e-mail, registrazione di nome di dominio e molti altri.
Le risorse che abbiamo a disposizione su questa piattaforma per eseguire i diversi applicativi sono 2 CPU e 8 GB di RAM.
Tutte le richieste che arrivano dall'esterno vengono intercettate tutte da un firewall che le filtra e per quelle che lo attraversano, prima fa un NAT dell'IP pubblico su un IP privato e poi le inoltra verso un server Ngnix. Esso in questo contesto svolge il ruolo di reverse proxy, il quale grazie a diversi virtual host pre-configurati inoltra tutte queste richieste ai diversi applicativi (**Assicurazione**, **Wallet**, **Backend**, **Badmin**).

4.3 Azure

Figura 1.41 Azure

Azure è la piattaforma cloud di Microsoft che offre servizi di cloud computing, web-hosting e anche in questo caso, molti altri.
Le risorse a disposizione su questa piattaforma, a differenza di quelle utilizzate su Aruba sono molto inferiori, dovuto al fatto che sono molto più costose. I vari applicativi vengono eseguiti con una sola CPU e 2 GB di RAM, ma le presatazioni comunque non ne risentono molto.
Le richieste dall'esterno vengono intercettate dall'Application Gateway, un componente di sicurezza che le filtra e anche in questo caso effettua un NAT dell'IP pubblico in uno privato. Poi, tramite una serie di listener e regole pre-configurate, inoltra tutte le richieste ad un server Apache, il quale tramite i virtual host le ribalta ai vari applicativi.

4.4 Piattaforme a confronto

L'obiettivo di questo paragrafo è di misurare e confrontare i tempi di risposta che ciascun componente MVC offre nelle 3 diverse metodologie di sviluppo (monolita su Aruba, micro-serivizi sempre su Aruba e micro-servizi su Azure), cercando di fornire anche una descrizione più dettagliata di come le richieste vengono effettuate e come successivamente queste vengono gestite nei controller.
Importante anche sottolineare il modo in cui sono state modificate le logiche di richiesta nell'applicativo monolita (basato sulle STORED PREOCEDURE eseguite direttamente sul database) rispetto ai micro-servizi (richieste ai core e utilizzo degli ORM per eseguire le query direttamente nei model).
In generale sono state eseguite 20 misurazioni per ciascuna piattaforma.

4.4.1 Landing page (applicativo monolita)

Questa è la pagina che fornisce all'utente informazioni sul prodotto ed è composta da 4 sezioni (introduzione, descrizione delle caratteristiche del prodotto, descrizione delle coperture, faq). Nel controller per renderizzare questa vista vengono fatte 5 chiamate ai model per eseguire 5 stored differenti:

1. una prima chiamata per ottenere alcune informazioni di alto livello per le landing (seo description, seo keywords, immagini). Nella risposta ci sono anche i parametri per inizializzare la sessione.
2. Una seconda chiamata per ricavare i dati delle prime 3 sezioni (introduzione, descrizione del prodotto, descrizione delle coperture). La risposta contiene anche gli id delle immagini salvate su MongoDB. Quindi nelle viste c'è una chiamata verso un metodo della libreria di Mongo che si occupa di interpretare il codice binario associato a questo id e renderizzare l'immagine.
3. Una terza chiamata per recuperare le faq e renderizzare anche l'ultima sezione della pagina.
4. La quarta chiamata ha come obiettivo quello di ricavare i dati del partner (colori, logo, informazioni anagarfiche). Anche in questo caso il logo viene salvato su Mongo e quindi è necessaria l'invocazione del metodo di libreria per interpretare il binario come immagine.
5. L'ultima chiamata è un controllo di sicurezza. Essendo che è una pagina pubblica si deve effettuare un check che il partner in questione sia valido e si controlla anche che abbia effettivamente accesso al prodotto in causa, stabilito negli accordi commericali stabiliti inizialmente.

Prendendo i tempi di risposta di questo controller su Aruba, mediamente variano tra i 2 ms e i 10 ms (per quelle pagine dove ci sono più sezioni).

4.4.2 Landing page (micro-servizi)

Nel architettura a micro-servizi vengono effettuate le stesse richieste dell'applicativo monolita, ma queste non vengono più risolte con l'ausilio delle STORED, ma con delle chiamate verso alcuni end-point interni messi a disposizione da **Core Axidigital**. I metodi di questi end-point eseguono le query utilizzando i model e i relativi ORM.
Prendendo i tempi di risposta della pagina, si nota che questi sono aumentati su entrambe le piattaforme. La differenza è dovuta principalmente al numero di risosrse che si hanno a disposizione su queste due.

- Azure: variano tra i 90 ms e i 120 ms (influenzati dal numero di sezioni di cui la pagina è composta).
- Aruba: variano tra i 70 ms e i 100 ms, in quanto le risorse sono maggiori che su Azure.

Questo degrado di prestazioni è dovuto principalmente al fatto che le richieste vengono risolte su diversi layer di servizio (il front-end di **Assicurazione** fa la richiesta – **Core Axidigital** utilizzando gli ORM, interagisce con lo strato di persistenza per risolverla), ma ha il vantaggio che non si debba più riscrivere lo stesso codice nei differenti progetti (si deve riscrivere solo il front-end che fa le chiamate), in quanto tutta la logica è centralizzata sul core. Quindi questa soluzione è molto più scalabile.

4.4.3 Chatbot (applicativo monolita)

Il rendering della pagina del chatbot dipende dal prodotto assicurativo e dalla sua complessità. Per alcuni prodotti il numero di passi non è noto a priori (default è 2), ma dipende dal numero di filtri che si devono applicare per ricavare il premio in fase di quotazione. Di default ci sono sempre almeno 2 passi, un entry point introduttivo per marcare l'inizio del chatbot e un exit point (solitamente la richeista della provincia) per individuare la fine del percorso. Questi due step sono utilizzati anche per Google Analytics per scopi statistici e commerciali .
Nel controller di questa pagina viene effettuata una prima richiesta per recuperare i diversi moduli che la compongono (risposta binaria, slider, autocomplete); la seconda richiesta serve per ottenere le province italiane e alcune informazioni extra connesse ad esse (es. classe territoriale che le compagnie attribuiscono a ciascuna di esse). In più per alcuni prodotti potrebbe essereci una richiesta ulteriore per recuperare informazioni sulle attività che gli utenti intendono asscicurare. Tutte queste richieste venogono risolte sempre con le STORED PROCEDURE nel database, essendo questa query una molto complessa, in quanto deve ritornare tutti i moduli (con i rispettivi parametri di configurazione) di cui è composto il chatbot in una sola chiamata.
I tempi di risposta, a seconda della complessità del prodotto e del numero di steps, oscillano tra 2 ms (prodotto più semplice con 2 passi) e 5 ms per i prodotti più complessi (5 passi). Questi tempo sono influenzate anche dal grado di complessità della query stessa e dal numero di join che devono essere effettuate.

4.4.4 Chatbot (micro-servizi)

In questo caso la logica per recuperare i moduli di cui il chatbot è composto, è leggermente diversa rispetto all'applicazione monolita, ma resta comunque il fatto che la complessità dipende dal prodotto.
Dapprima viene effettuata una richiesta sempre verso il **Core Axidigital** per recuperare il numero di passi di cui il chatbot è composto; poi vengono effettuate tante richieste quanti sono questi sono. L'end point è sempre lo stesso e a seconda del parametro in ingresso, dipendente dalla tipologia di modulo (binario, slider, autocomplete), esso ritorna le informazioni di quel modulo (copy, domanda e risposta). Anche qui il vantaggio principale è legato al fatto che non si è piu dovuto riscrivere più volte la logica del chatbot sui diversi progetti, ma essendo centralizzata sul core, gli utilizzatori devono solo fare le richieste passandoli correttamente i parametri.
I tempi di risposta di questa pagina dipendono sempre dal numero di passi di cui è composto il chatbot (influenzata dalla complessità del prodotto):

- Azure: variano tra i 120 ms (chatbot con 2 steps) e i 280 ms (6 steps – essendo questo uno dei prodotti più complessi digitalizzati).
- Aruba: variano tra i 100 ms e i 250 ms (più risorse).

Come per le landing, le presazioni sono peggiorate anche su questa pagina, ma il vantaggio guadagnato è la scalabilità.

4.4.5 Quotatore – rendering pagina (applicativo monolita)

La visualizzazione della pagina della quotazione dipende dalla tipologia di prodotto in quanto a catalogo ci sono sia quelli a pacchetto che usano un determinato template HTML che quelli a garanzie e ne utilizzano un altro diverso.
In questo caso, i tempi di risposta della pagina dipendono dalla complessità del modulo HTML elementare oltre che dalla complessità del prodotto in sè. Le prime istruzioni nel controller si occupano di recuperare i dati ricevuti dal chatbot. Ricevuti questi dati, per i prodotti a pacchetto si eseguono in catena le seguenti richieste al database:

1. una prima query per recuperare tutti i pacchetti e le garanzie di cui ciascuno di essi è composto;
2. un'eventuale query per recuperare parametri aggiuntivi (massimale e premio associato), se previsto dal pacchetto.

Ottenuti questi dati dal database, nel controller vengono normalizzati e poi viene caricata la vista. I tempi per caricare questa pagina variano dai 2 ms ai 5 ms in base al numero delle interazioni che si devono eseguire con il database e dalla complessità delle query intererssate.

I prodotti NON a pacchetto invece, eseguono una prima query per recuperare tutte le loro garanzie e poi, anche in questo caso, spesso è necessario fare una seconda e una terza richiesta al database per ricevere altri parametri utili in fase di calcolo del premio (tassi, premi, coefficienti).
I tempi di caricamento della pagina dipendono sempre dalla complessità del prodotto e variano tra i 2 ms e 5 ms.

4.4.6 Quotatore - rendering pagina (micro-servizi)

Nella nuova versione del progetto si è semplificato notevolmente il modo in cui vengono caricati i moduli HTML elementari sia per i prodotti a pacchetto che a garanzie, spostando sul **Core Axidigital** la complessità di costruire la struttura dati adatta a popolare tali moduli.
Nel controller viene fatta una prima richiesta verso il core per recuperare le informazioni sulla tipologia di prodotto. Anche in questo caso è necessaria una distinzione tra i prodotti a pacchetti e quelli a garanzie in quanto la logica di costruzione del prodotto in sé è diversa.
Per i prodotti a pacchetto vengono prima richiesti i pacchetti, poi per ciascun pacchetto viene fatta una richiesta per recuperare le sue garanzie e un'altra per i premi. Per quei pacchetti che prevedono anche dei massimali, la richiesta per recuperare i premi restituisce anche tali massimali. I tempi per caricare questa vista variano a seconda della complessità dei pacchetti e dal numero di garanzie che sono composti (il tempo è proporzionale al numero di garanzie) e se hanno oppure no il massimale.

- Azure: variano tra i 50 ms (prodotti molto semplici – senza massimali) e i 70 ms (prodotti con massimali).
- Aruba: simili, in quanto la logica di caricamento è esattamente la stessa.

Le prestazioni per i prodotti a garanzie sono peggiorate notevolmente rispetto all'applicativo monolita in quanto l'end point di **Core Axidigital** ha il compito di costruire la struttura dati molto complessa, citata nel capitolo 3 - sezione 3.4.4.3 (ad ogni garanzia associare tutte le caratteristiche che permettano il calcolo del premio – imposta, massimale e/o tasso e/o premio netto).
In questo caso i tempi dipendono principalmente dal numero di garanzie di cui è composto il prodotto in questione e dalla loro complessità (se hanno dei massimali è necessario inserire nella struttura dati i valori di massimale e l'eventuale premio associato).

- Azure: variano tra i 150 ms (poche garanzie) e i 300 ms (prodotto con numerose garanzie).
- Aruba: anche in questo caso molto simili a quelli di Azure per lo stesso motivo dei prodotti a pacchetto (stessa logica, ma diversa piattaforma e risorse).

4.4.7 Quotatore – calcolo premio (applicativo monolita)

Come nel rendering delle pagine anche per il calcolo del premio è necessaria la distinzione logica tra prodotti a pacchetto e quelli a garanzie, in quanto la modalità di calcolo è diversa. Per i prodotti a pacchetto il premio netto, al quale si devono applicare le imposte e l'eventuale percentuale di sconto, arriva spesso dalla POST inviata dal browser in AJAX a ogni interazione con il pacchetto (per esempio scelta di un massimale - ciascuno ha un suo premio netto associato - in questo caso al massimale di 250.000 € corrisponde un premio netto di 267 €, a quello di 500.000 €, un premio di 270 € e così via).

```
<select id="id_select" name="name_select">
  <option value="" disabled="">…</option>
  <option value="267">250.000 €</option>
  <option value="270">500.000 €</option>
  <option value="320" selected="selected">1.000.000 €</option>
  <option value="880">1.500.000 €</option>
  <option value="961">2.000.000 €</option>
</select>
```

Figura 1.42 Associazione premio netto – massimale

In quei pacchetti dove non è prevista la scelta del massimale, si utilizza invece l'id del pacchetto (salvato in un input di tipo hidden e passato sempre nella POST) per fare una richiesta al database e ricavare il premio netto. Ottenuto il premio netto, si applica la metodologia di calcolo descritte nelle sezioni precedenti per ricavare il lordo. I tempi di risposta per il calcolo variano a seconda del tipo di pacchetto:

- pacchetti senza scelta di massimale (si può scegliere solo il pacchetto, se più di uno): tra 680 ns e 900 ns;
- pacchetti con scelta di massimale: tra i 750 ns e i 200 ms. Questa grande differenza è dovuta anche al fatto che per alcuni prodotti, che presentano un premio molto alto, vengono fatte delle richieste verso servizi esterni per mensilizzare il premio lordo.

Figura 1.43 Premio mensile

Nei prodotti a garanzie, la metodologia di calcolo applicata è leggermente diversa, in quanto si devono applicare le formule sulla singola garanzia selezionata, quindi i tempi di risposta sono proporzionali al numero di garanzie selezionate (maggiore è il numero di garanzie selezionate nel quotatore e maggiore sarà il tempo impiegato da questo per il calcolo del premio lordo).
In questo caso, non si fanno richieste al database per ricavare i premi, ma si utilizzano i parametri che arrivano nella POST e la sessione che spesso contiene alcuni coefficienti.
I tempi per il calcolo variano tra i 2 ms e i 200 ms - tempi dipendenti dal numero di garanzie selezionate e anche in questo caso, dalle richieste verso end point esterni (calcolo del premio mensile).

4.4.8 Quotatore - calcolo premio (micro-servizi)

Nel architettura a micro-servizi per i prodotti a pacchetto, l'obiettivo principale è stato quello di rimuovere il più possibile l'interazione con il database per ricavare il premio lordo e quindi di migliorare leggermente le prestazioni. Quindi tutti i dati necessari al calcolo, arrivano principalmente dalla POST effettuata in AJAX dal browser ad ogni interrazione dell'utente con la pagina oppure dalla sessione, lato server.
La metodologia di calcolo (formule) è rimasta quella utilizzata nell'applicativo monolita. I tempi di risposta sono:

- Azure: tra i 10 ns (prodotti senza massimale) e i 200 ms (prodotti con massimale e/o mensilizzazione premio).
- Aruba: simili a quelli su Azure.

Per i prodotti a garanzie è cambiata la struttura dati utilizzata per il calcolo, con la conseguenza che sono stati introdotti dei cicli *for* per scorrerla. Per ricavare il premio, in base alla modalità di calcolo descritta nella sezione 3.4.4.3 vengono invocati i rispettivi metodi di calcolo. Anche qui i tempi di risposta sono proporzionali al numero di garanzie selezionate nel quotatore:

- Azure: tra i 100 ns (poche garanzie selezionate) e i 200 ms (tante garanzie selezionate e/o mensilizzazione del premio).
- Aruba: simili a quelli su Azure.

Le prestazioni sono migliorate leggermente in quanto è stata migliorata la metodologia di calcolo in sé (per tutte quelle garanzie non selezionate, il premio veniva messo di default a zero senza dover invocare il metodo per il calcolo). Un altro vantaggio è quello di aver reso molto più rapida la digitalizzazione del prodotto, evitando così quel lavoro di copia incolla del codice che si faceva nell'applicativo monlita ogni volta che si creava un nuovo prodotto. Inoltre ha anche permesso di iniziare a implementare delle interfacce per creare le garanzie, associarle ai prodotti, ai premi, ai massimali etc.

4.4.9 Registrazione – caricamento form

Il modo in cui viene caricato il form di registrazione nelle 2 piattaforme (applicativo monolita e micro-servizi) è rimasto invariato. Il controller prima di caricare la vista effetua anche il ricalcolo del premio e per confrontarlo con quello che arriva nella POST (evitando così attacchi di tipo MITM attivi – modifica del premio). I tempi per caricare questa pagina sono i seguenti:

- Aruba (applicativo monolita): tra i 240 ns e i 400 ns (per tutti quei prodotti che richiedono ulteriori input oltre a quelli per raccogliere i dati anagrafici).
- Aruba/Azure (micro-servizi): tra i 200 ns (prodotti a pacchetto) e i 50 ms (prodotti a garanzie). Questo peggioramento è dovuto al fatto che è stato introdotto, per i prodotti a garanzie un metodo (esegue un doppio ciclo for - complessità $O(n^2)$) per salvare in sessione le garanzie selezionate in fase di quotazione.

4.4.10 Registrazione – submit form (applicativo monolita)

L'obiettivo della registrazione è quello di raccogliere i dati dell'utente e salvarli, quindi sono necessarie le transazioni per interagire più di una volta con il database e le sue tabelle. La complessità è in parte nel controller, in parte sul database nelle stored. Le operazioni eseguite sono le seguenti:

1. verifica esistenza utente sulla piattaforma;
2. se l'utente è uno nuovo, lo si deve creare e agganciarli un'anagrafica (fisica o giuridica);
3. creata l'utenza, oppure recupercato l'id di quella già presente nel database, si creano il contrattto, il titolo, le garanzie (o il pacchetto) selezionate nel quotatore e infine si notifica tutto con l'invio delle mail.

Essendo eseguite tutte queste operazioni, anche complesse in alcuni casi, le prestazioni ne risentono abbastanza e sono influenzate (in peggio) anche dall'invio delle mail di notifica dell'avvenuto ordine (sia ad Axieme, che al potenziale cliente).
Un altro fattore che abbassa le presatazioni sono quelle legate all'interazione con il database non relazionale (MongoDB) e la creazione dei file pdf da allegare alle mail verso l'utente.
Per quanto riguarda i tempi di risposta, essi dipendono dal numero di tabelle e inserimenti che si devono effettuare e siamo già all'ordine del secondo, ponendo così la registrazione come la parte peggiore del journey assicurativo. Essi variano dai 5 secondi (caso in cui non tutte le tabelle sono interessate dalla transazione) ai 6 secondi (nel caso peggiore in quanto si deve inserire in tutte le tabelle) .

4.4.11 Registrazione - submit form (micro-servizi)

In questo progetto, **Assicurazione** si comporta solo da front-end in quanto deve raccogliere tutti i dati della quotazione e inviarli all'end-point, spostando così tutta la complessità di inserimento nelle tabelle sul **Core Axieme,** che deve applicare correttamente i filtri descritti nella sezione 3.4.6.2, e sul **Core Builder** che si occupa di aprire la transazione, fare le opportune verifiche di validazione e inserire nelle tabelle nel caso in cui tutti i filtri vengono passati correttamente.
Infine, una volta inseriti tutti i dati, il **Core Axieme** con i parametri ottenuti dal builder deve generare (solo per alcuni partner – uno tra questi, Axieme) i file in formato pdf (questionario di adeguatezza descritto nella sezione 3.3.4.7), interagire con MongoDB per salvarlo e inviare tutte le mail di notifica verso l'utente e verso Axieme.
Per quanta riguarda le prestazioni, esse sono:

- Azure: tra i 900 ms e i 2 secondi. Il miglioramento è dovuto al fatto che per alcuni progetti-partner non è necerssario generare i file pdf che devono essere salvati successivamente su MongoDB.
- Aruba: essendo questa piattaforma utilizzata per Axieme, si devono eseguire le stesse operazioni che si facevano nell'applicativo monolita. I tempi di risposta variano tra i 4 secondi e i 5 secondi.

4.4.12 Tabella riassuntiva delle prestazioni

-	Aruba(app. monolita)	Aruba (microservizi)	Azure (micro-servizi)
Landing page	2 ms - 10 ms	70 ms - 100 ms	90 ms - 120 ms
Chatbot	2 ms - 5 ms	100 ms - 250 ms	120 ms - 280 ms
Quotatore rendering (prodotti a pacchetto)	2 ms - 5 ms	50 ms - 70 ms	simili a quelli di Aruba
Quotatore rendering (prodotti a garanzie)	2 ms - 5 ms	150 ms - 300 ms	simili a quelli di Aruba
Quotatore calcolo premio (prodotti a pacchetto)	680 ns - 900 ns (prodotti senza massimale) 750 ns – 200 ms (prodotti con massimale)	10 ns (prodotti senza massimale) - 200 ms (prodotti con massimale)	simili a quelli di Aruba
Quotatore calcolo premio (prodotti a garanzie)	2 ms - 200 ms	100 ns - 200 ms	simili a quelli di Aruba
Registrazione caricamento form	240 ns - 400 ns	200 ns - 50 ms	simili a quelli di Aruba
Registrazione submit form	5 s - 6 s	4 s - 5 s	900 ms - 2 s

In generale, analizzando questi tempi di risposta, si può osservare che nell’architettura a micro servizi è stato introdotto un calo drastico delle prestazioni, favorito però dal guadagno ottenuto in scalabilità del codice e dalla rapidità con cui si è potuto affrontare nuovi progetti, essendo che la parte core della piattaforma non doveva essere più riscritta.

4.5 Centreon

Figura 1.44 Centreon

Centreon è una piattaforma open source utilizzata per il monitoraggio delle risorse lato server. In Axieme essa viene utilizzata per monitorare il carico di lavoro sul database (sia MariaDB che MongoDB) e le relative percentuali di utilizzo di RAM e CPU usate dai diversi applicativi.
E' importante utilizzare strumenti come questo in quanto possono essere una prima linea di difesa contro gli attacchi informatici (standard OWASP TOP 10). Offrendo informazioni in tempo reale per quanto riguarda l'utilizzo delle risorse, è possibile individuare fin da subito tutte quelle situazioni anomale, peremttendo un pronto intervento in una fase dove non è stato creato alcun danno maggiore.

Di seguito un esempio di grafico che monitora costantemente il database relazionale con i picchi che si riferiscono al momento in qui vengono eseguite le query. Il report si estende di solito all'ultima settiamana.

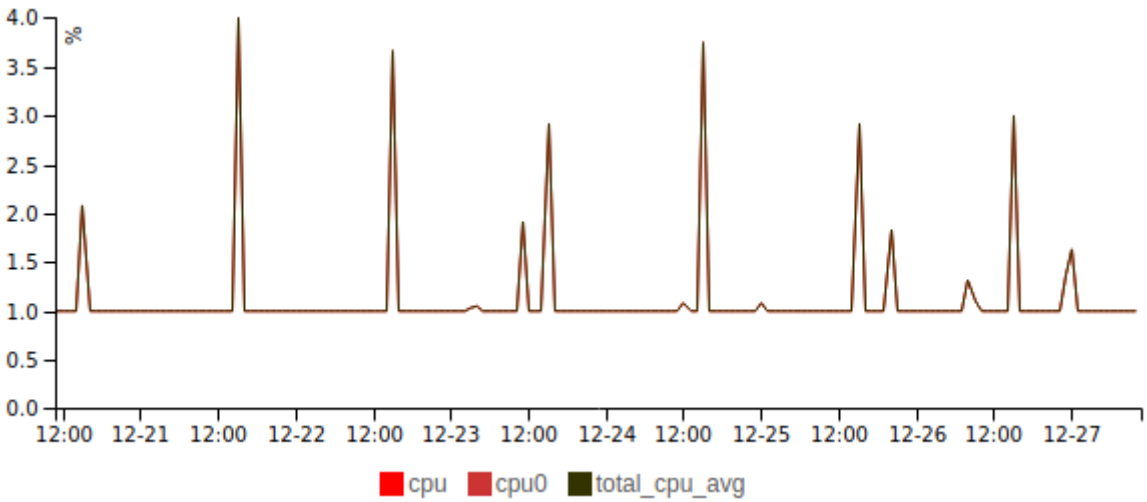


Figura 1.45 Monitoring database

Questo grafico invece mostra la CPU utilizzata per eseguire gli applicativi, con i picchi che si riferisono alle operazioni più complesse.

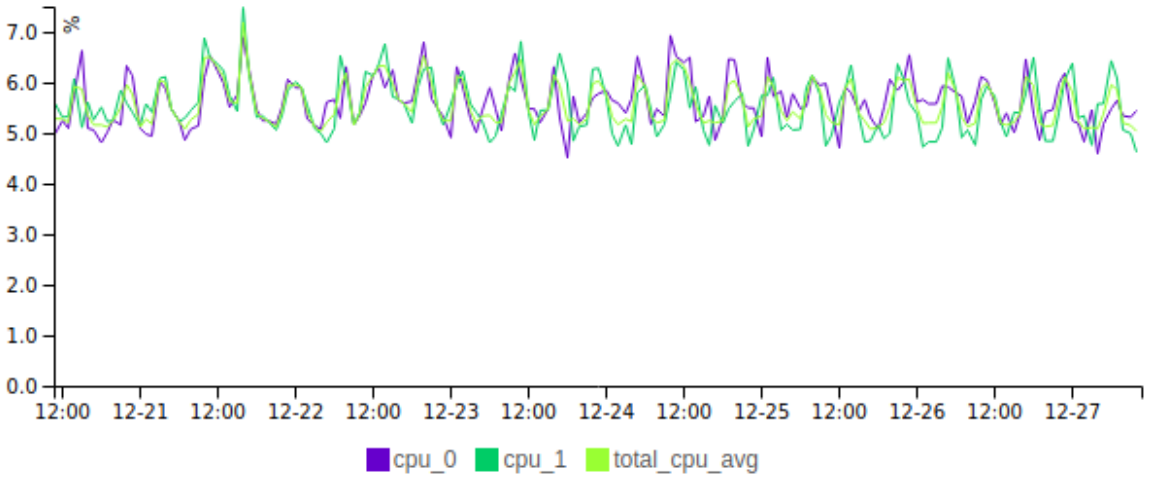


Figura 1.46 Monitoring CPU server

Altri grafici simili, vengono utilizzati per monitorare le risorse utilizzate da MongoDB, dal server Ngnix, dal server Apache etc.

Capitolo 5
Conclusioni

Tale capitolo è dedicato alle considerazioni generali sulla tesi riassumendo i punti chiavi dello sviluppo.

5.1 Considerazioni

Il progetto iniziale prevedeva lo sviluppo di una piattaforma web nell'ambito assicurativo da utilizzare solo internamente per la digitalizzazione e la vendita di prodotti assicurativi. Successivamente, grazie alle opportunità di business sviluppate dagli altri colleghi, questa è stata revisionata e trasformata, diventando adatta anche per essere messa a disposizione di altri partner (assicurativi e non), personalizzandola come se fosse loro (modalità white label).
Lo svilippo è stato quindi guidato da queste nuove esigenze di business ed è passato attraverso diversi stati, partendo da un applicativo monolita (scalabilità molto bassa) basato su un certo schema delle tabelle nel database e una certa struttura dati, arrivando ad un architettura a micro-servizi e una struttura dati molto più ottimizzato (si è perso però dal punto di vista prestazionale essendo che le richieste devono essere risolte da diversi strati, ma si è guadagnato in scalabiltà e manutenzione del codice).

5.1.1 Sviluppo iniziale come applicativo monolita

La prima versione del progetto, sviluppata utilizzando la versione 3 del Framework CodeIgniter, era un applicativo monolita, ovvero tutte le operazioni venivano eseguite all'interno di questo stesso progetto (dall'interazione con il database (con l'ausilio dei model e delle STORED PROCEDURES), alla manipulazione dei dati nei controller, fino al rendering delle viste).
Un approccio che inizialmente andava bene, ma dal momento in cui i volumi interni sono aumentati e i requisiti sono stati modificati in corso d'opera (tecnica di sviluppo AGILE), questo modo di sviluppare, si è dimostrato inadatto per gestire e soddisfare tutte le nuove richieste.

Il problema principale era legato al fatto che in molti casi dovevamo copiare ed incollare lo stesso codice in diversi progetti, dando origine a molte problematiche, quali difficoltà di mantenere il codice, correzione di bugs e messa in produzione molto onerosa etc.
Inoltre la struttura dati sviluppata per la digitalizzazione dei prodotti (essendo io alle prime armi con i prodotti assicurativi) era inadatta per soddisfare in breve tempo le esigenze. Quindi è stata necesssaria una revisione totale della piattaforma, guidata dall'esperienza acquisita dalla digitalizzazione di diversi prodotti, per poter soddisfare al meglio tutte le nuove richieste presentatosi.

5.1.2 Revisione della piattaforma e architettura a micro-servizi

La revisione della piattaforma è stata spesso guidata dalle nuove opportunità create dagli altri colleghi. Essendo che questa doveva gestire quotazioni e digitalizzazione di nuovi prodotti assicurativi, in poco tempo, è stato necessario ricreare la struttura dati utilizzata per i calcoli sulla base anche dell'esperienza accumulata grazie al progetto monolita.
Inoltre, essendo che questa doveva essere esposta in modalità white label, si è dovuto rivedere anche la struttura dati per gestire tutte le entità che potevano acquisirne l'utilizzo d (sia broker assicurativi che partner non assicurativi).
Per agevolare lo sviluppo si è passato alla versione 4 del framework, che offre diverse migliorie per gestire meglio il codice e si è iniziato anche ad adottare i micro-servizi, introducendo diversi core, con la funzionalità di raccogliere a fattor comune tutte quelle operazioni che nel progetto monolita si dovevano copiare e incollare:

1. **core Axidigital** per gestire i prodotti e le diverse entità che possono utilizzare la piattaforma;
2. **core Builder**: per raccogliere in un unico punto tutte le funzionalità core della piattaforma (gestione delle polizze, delle utenze etc);
3. **core Axieme**: uno strato intermedio che si occupa di ricevere le richieste dai vari applicativi (Assicurazione, Backend, Wallet), le normalizza in qualche maniera e le inoltra verso il **core Builder**;
4. **core Partner**: strato intermedio (verso il **core Builder**) per i progetti partner.

Questa nuova modalità di sviluppo a microservizi e la revisione della struttura dati (modificato anche lo schema delle tabelle nel database) ha permesso una rapida digitalizzazione di tutti quei prodotti a pacchetti che rientrano nella classificazione spiegata nel capitolo 3 sezione 3.4.1, ma anche dei prodotti a garanzie, i quali però possono richiedere del codice ad-hoc per gestire alcune casistiche (dipendenti dal prodotto in sè) di calcolo non previste in fase di revisione.