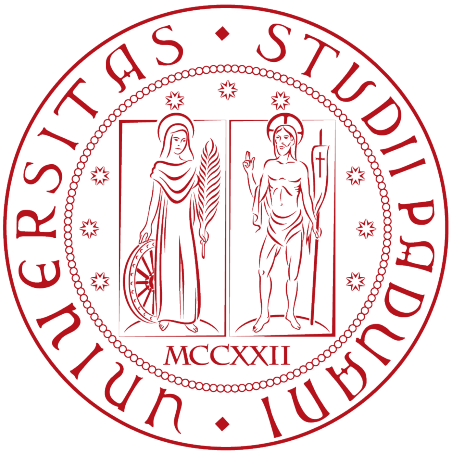
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



moviEXPENSE 2: studio e aggiornamento di un'applicazione mobile per la registrazione di note spese

Tesi di laurea

Relatore
Prof. Francesco Ranzato

Laureando
Alessio Banzato
Matricola 2042381

Anno Accademico 2023-2024

Next time is next time. Now is now.

— Hirayama, Perfect days (2023)

Sommario

Il presente documento descrive il lavoro svolto durante lo *stage* curriculare svolto presso VISIONEIMPRESA s.r.l. Società Benefit di Pernumia (PD).
L'obiettivo principale dello *stage* era l'aggiornamento dell'applicazione *mobile* moviEXPENSE, precedentemente sviluppata da VISIONEIMPRESA per facilitare e velocizzare la registrazione delle note spese, in modo da ridurre la quantità di documenti cartacei prodotti e i tempi di attesa per la ricezione dei dati da parte del personale amministrativo dell'azienda utilizzatrice.
L'aggiornamento riguardava in particolare la sicurezza dell'applicazione, in quando le stringhe di connessione ai database venivano trasmesse in chiaro, la correzione di *bug*, l'implementazione di nuove funzionalità e il miglioramento di quelle già presenti, e, infine, il *restyling* dell'interfaccia grafica.

Ringraziamenti

Ringrazio innanzitutto il professor Francesco Ranzato, relatore della tesi, per la disponibilità e la rapidità nella comunicazione.

Desidero ringraziare Francesco Turra e tutta VISIONEIMPRESA per avermi dato l'opportunità di svolgere lo *stage* in tempi rapidi e compatibili con la presente sessione di laurea.

Ringrazio anche la mia famiglia e in particolare i miei genitori per il supporto, anche economico, datomi durante questi tre anni.

Un ringraziamento va anche ai compagni di corso che ho conosciuto, in particolare al gruppo *Error_418*, ottimi compagni di lavoro durante il progetto di ingegneria del *software*.

Ringrazio ovviamente i miei amici Marta e Gianmarco, che mi hanno sempre supportato, spingendomi ad andare sempre avanti in qualsiasi situazione, soprattutto grazie ai *sushi* post sessione.

Infine vorrei ringraziare anche tutti gli amici trovati e ritrovati in questi tre anni, che hanno contribuito ad alleggerire il percorso che si sta ora concludendo.

Padova, Settembre 2024 Alessio Banzato

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 L'azienda

Figura 1.1: Logo di VISIONEIMPRESA s.r.l. Società Benefit

VISIONEIMPRESA è una *software house* nata negli anni Ottanta a Pernumia, in provincia di Padova. Dal 2016 fa parte di *Office Group*, un gruppo nato dall'unione di *Office Information Technologies* di Montegrotto Terme con altre aziende di Veneto e Lombardia.
L'obiettivo di VISIONEIMPRESA è lo sviluppo e la vendita di *software* gestionali per agevolare il lavoro delle piccole e medie aziende e contribuire alla loro digitalizzazione. I *software* sviluppati si dividono principalmente in due gruppi:

- ***software* Vision**: il *software* primario è VisionEnterprise, ideato per migliorare la gestione dell'organizzazione aziendale nel suo complesso. In questo gruppo di *software* rientrano le **soluzioni verticali**, gestionali per specifiche tipologie di aziende, ad esempio VisionENERGY per il commercio di prodotti petroliferi o VisionFRESH per l'ingrosso di prodotti ortofrutticoli;
- ***software* movi**: applicazioni *mobile* specializzate in determinate azioni, ad esempio l'analisi dati (moviCHECK), l'invio di ordini (moviORDER) o l'invio di *ticket* per l'assistenza post-vendita (moviTICKET).

1.1.1 Società Benefit

Nel 2023 VISIONEIMPRESA s.r.l. è diventata anche **società benefit**. Una società benefit non è altro che una tipologia di azienda che aggiunge a fianco degli obiettivi di profitto finanziario quello di avere un effetto positivo su ambiente e società. L'Italia è stato il primo Paese al di fuori degli USA ad introdurre questa tipologia di azienda.

VISIONEIMPRESA, quindi, si impegna a perseguire obiettivi tra i quali rientrano il mantenimento di un buon equilibrio tra vita personale e lavorativa dei dipendenti, la collaborazione con istituti di formazione del territorio (scuole superiori e università) e la riduzione dell'impatto ambientale dell'azienda.

1.2 Descrizione dello *stage*

1.2.1 Introduzione al progetto

Figura 1.2: Logo di moviEXPENSE

Il progetto di *stage* si basava sullo studio e l'aggiornamento di moviEXPENSE, l'applicazione di VISIONEIMPRESA dedicata alla registrazione di note spese. L'app nasce con l'obiettivo di ottimizzare il processo di registrazione delle spese, cercando di ridurre a zero i rischi ad esso correlati.
La normale registrazione di spese avviene tramite accumulo di scontrini e fatture e compilazione di moduli. Tutti questi elementi vengono poi consegnati all'amministrazione dell'azienda che deciderà come gestirli. Questo processo, però, rischia di utilizzare diverse quantità di carta per la modulistica, e non è sicuro dal punto di vista della raccolta di fatture e scontrini, in quanto questi potrebbero essere persi per distrazione o potrebbero danneggiarsi in svariati modi.
Con moviEXPENSE, invece, non è più necessaria la modulistica cartacea poiché tutta trasposta nell'applicazione, e non c'è il rischio di perdita di scontrini e fatture in quanto possono essere salvati insieme alla spesa tramite foto.
Un altro problema che moviEXPENSE risolve è quello della comunicazione con l'amministrazione aziendale: dal momento in cui una spesa viene salvata nell'applicazione, è subito visibile dall'amministrazione, in quanto i dati sono tutti salvati su *cloud*, e di conseguenza è immediatamente gestibile, andando così a ridurre drasticamente i tempi morti del processo.

1.2.2 Obiettivi

Obbligatori

- Studio degli strumenti e delle tecnologie da utilizzare, esposte nel dettaglio nel capitolo successivo;
- Analisi e revisione del sistema di autenticazione dell'app, in particolare:
  - comprensione dei meccanismi di autenticazione presenti;
  - correzione delle API$_G$ di autenticazione;
  - correzione delle API$_G$ di collegamento ai *database*.
- Implementazione di ulteriori controlli per rendere più sicura e veloce la registrazione delle spese;

- Correzione degli errori presenti nell'applicazione, segnalati nell'analisi condivisa a inizio *stage*;
- Implementazione delle novità introdotte nell'analisi condivisa a inizio *stage*;
- Rivisitazione dell'interfaccia grafica su *smartphone*;
- *Testing* dell'applicazione.

Desiderabili

- Creazione della documentazione con i seguenti livelli di priorità decrescente:
  1. Documentazione tecnica delle modifiche effettuate e delle novità introdotte;
  2. Manuale utente delle modifiche effettuate e delle novità introdotte;
  3. Documentazione tecnica generale dell'applicazione;
  4. Manuale utente generale dell'applicazione.

Opzionali

- Rivisitazione dell'interfaccia grafica per uso su *tablet*.

1.2.3 Pianificazione e svolgimento del lavoro

Pianificazione

Le attività da svolgere durante il periodo di *stage* sono state distribuite nell'arco di otto settimane, per un totale di 300 ore di lavoro. Di seguito è riportata nel dettaglio la suddivisione delle attività per periodi:

1. **Studio**: settimana dedicata allo studio dell'applicazione e delle tecnologie da utilizzare;
2. **Autenticazione**: miglioramento del precedente sistema di autenticazione, da effettuare basandosi sul lavoro svolto su altre app, in particolare moviORDER e moviCHECK;
3. ***Bug fixing***: correzione degli errori presenti nell'applicazione;
4. **Novità**: implementazione di nuove funzionalità e ampliamento di quelle già esistenti;
5. **Interfacce**: rinnovamento dell'interfaccia grafica ispirato dai *mockup*$_G$ di moviORDER;
6. **Test e documentazione**: *testing* dell'applicazione e redazione della documentazione richiesta.

Nella seguente tabella sono riportati i periodi sopra citati e la loro suddivisione settimanale e oraria.

Settimana	Attività	Ore
1	Studio	40
2	Autenticazione	40
3	*Bug fixing*	40
4-5	Novità	80
6	Interfacce	40
7-8	Test e documentazione	60
Totale		300

Tabella 1.1: Pianificazione del lavoro per settimane e ore

Svolgimento

Lo *stage* si è svolto completamente in presenza a Pernumia, presso la sede di VISIONEIMPRESA. All'inizio dei lavori sono stato accolto dall'amministratore, il quale mi ha presentato l'azienda, i principali *software* da loro sviluppati e ha fatto una panoramica sul mio progetto.
Successivamente mi ha condiviso un documento di analisi redatto da lui e da un utilizzatore di moviEXPENSE presso *Office Group*. In questo documento erano presenti tutti gli aspetti critici dell'applicazione e tutte le novità che dovevano essere apportate per migliorare l'app e il suo utilizzo.
Tutto il lavoro è stato svolto utilizzando un *computer* aziendale con sistema operativo Windows 10, e uno *smartphone* aziendale nel quale eseguire l'applicazione.
Durante lo *stage* ho lavorato in modo indipendente, avendo come riferimenti principali:

- **Francesco Turra**: amministratore dell'azienda, consulente, analista, con il quale sono stato in contatto per esporre l'andamento del lavoro e validare le modifiche e implementazioni;
- **Paolo Stefani** (tutor aziendale) e altri programmatori: per qualsiasi problema o difficoltà con il codice o il *setup* dell'ambiente di sviluppo. Con Paolo in particolare anche per la validazione finale del lavoro svolto.

1.3 Struttura del documento

Il secondo capitolo presenta gli strumenti e le tecnologie utilizzate durante lo *stage*.

Il terzo capitolo descrive l'architettura dell'applicazione.

Il quarto capitolo espone le principali modifiche effettuate e novità apportate, concludendo con una breve descrizione delle attività di verifica e validazione.

Il quinto capitolo espone le conclusioni tratte dall'esperienza di *stage*.

1.4 Convenzioni tipografiche

1.4.1 Acronimi, abbreviazioni, glossario

Tutti gli acronimi, le abbreviazioni e i termini che necessitano di una definizione, poiché ambigui o non di uso comune, sono evidenziati in blu. La loro definizione è disponibile nelle apposite sezioni "Acronimi e abbreviazioni" e "Glossario" presenti al termine del documento. Se il documento è in consultazione digitalmente, è possibile cliccare sul termine desiderato per essere portati alla sua definizione.
I termini appartenenti al glossario si differenziano dagli altri dalla presenza di una "G" al pedice.

1.4.2 Elenchi

Per migliorare la leggibilità, l'organizzazione e l'efficacia del testo, ho utilizzato degli elenchi puntati o numerati. L'uso di elenchi numerati è limitato ai casi in cui:

- è presente una sequenza di azioni ordinate;
- è presente una gerarchia di priorità;
- alcuni punti dell'elenco devono essere citati in seguito.

1.4.3 Stili di testo

Grassetto

Utilizzato per evidenziare parole chiave all'interno del testo o evidenziare termini iniziali degli elenchi.

Corsivo

Utilizzato per tutti i termini in lingua straniera o appartenenti al gergo tecnico.

`Monospaziato`

Utilizzato per i nomi di classi, metodi, tabelle dei *database*, attributi, variabili, file.

Capitolo 2

Tecnologie e strumenti

In questo capitolo sono presentate le tecnologie e gli strumenti utilizzati per lavorare al progetto di stage. Per ogni tecnologia e strumento verrà riportata la versione utilizzata, il suo scopo all'interno del progetto e una breve descrizione.

2.1 Tecnologie

2.1.1 *Frontend*

Xamarin

Figura 2.1: Logo di Xamarin

- **Versione**: 5.0.0.2578;
- **Utilizzo**: sviluppo dell'interfaccia grafica.

Xamarin è un *framework* per lo sviluppo di applicazioni ideato da Microsoft. È un progetto *open-source* basato sull'utilizzo del linguaggio **XAML** per la creazione diretta delle interfacce, e **C#** per il *code-behind*$_G$. Questo *framework* è stato sviluppato per poter realizzare applicazioni Android, iOS e Windows partendo dalla stessa *codebase*$_G$. A partire da maggio 2024 Xamarin non è più supportato, a favore di .NET MAUI. La migrazione da Xamarin a .NET MAUI è stata considerata, però, un lavoro troppo impegnativo per il progetto di *stage*, che si è svolto dunque utilizzando il codice Xamarin già presente nel *repository*.

2.1.2 *Backend*

C#

Figura 2.2: Logo di C#

- **Versione**: 12.0;
- **Utilizzo**: implementazione delle API$_G$ e scrittura del *code-behind*$_G$.

C# (dove il simbolo "#" è pronunciato *sharp*) è un linguaggio di programmazione sviluppato da Microsoft all'inizio degli anni 2000 per essere utilizzato con .NET Framework. È orientato agli oggetti ma supporta diversi paradigmi, tra cui funzionale e concorrente. Gode di tipizzazione forte, *garbage collector*$_G$, una sintassi molto simile ai suoi predecessori e ispiratori Java e C++ e utilizza l'indentazione Allman: le parentesi graffe che delimitano un blocco di codice sono allineate all'inizio della parola chiave che definisce il blocco di codice, ad esempio

```
while (condition())
{
    do_something();
}
```

.NET

Figura 2.3: Logo di .NET

- **Versione**: 8.0.302;
- **Utilizzo**: implementazione delle API$_G$ e *code-behind*$_G$.

.NET (pronunciato *dotnet*) è un *framework open-source* di Microsoft utilizzato per lo sviluppo di applicazioni. È dotato di un ecosistema che comprende:

- linguaggi, tra cui C#, F#, Visual Basic;
- *Common Language Runtime*, ovvero una macchina virtuale che esegue il codice intermedio ottenuto dalla compilazione dei linguaggi utilizzati;
- una serie di librerie$_G$ che implementano diverse funzioni, utilizzabili includendo nel codice i corretti *namespace*.

2.2 Strumenti

2.2.1 Strumenti di sviluppo

Microsoft Visual Studio

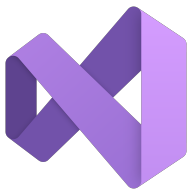


Figura 2.4: Logo di Visual Studio

- **Versione**: 17.10.3;
- **Utilizzo**: codifica e verifica.

Visual Studio è un IDE$_G$ sviluppato da Microsoft utilizzato principalmente per lo sviluppo con linguaggi quali C#, C++, F#, Visual Basic e i *framework* .NET e Xamarin.
Può essere configurato per lo sviluppo *mobile* offrendo di conseguenza un ampio insieme di funzionalità specifiche, ad esempio l'esecuzione e il *debug* su un simulatore, il *deploy* e il *debug* su dispositivo fisico, l'*hot reload*$_G$ e la configurazione di diversi dispositivi su cui poter eseguire l'applicazione. Oltre a ciò, Visual Studio ha una funzione di correzione della sintassi e di suggerimento automatico.

Microsoft SQL Server Management Studio

Figura 2.5: Logo di Microsoft SQL Server Management Studio

- **Versione**: 20.1.10.0;
- **Utilizzo**: gestione e interrogazione di *database*.

Microsoft SQL Server Management Studio (SSMS) è un'applicazione Microsoft concepita per l'amministrazione di *server* SQL Microsoft. Offre la possibilità di connettersi a *server* SQL e visualizzarne i contenuti grazie alla funzionalità di esplorazione delle risorse. In questo modo è possibile visualizzare i *database* presenti in un *server* e tutte le relative tabelle, eseguire *query* per visualizzare determinati dati oppure modificare la struttura o i contenuti delle tabelle.

Swagger-UI

Figura 2.6: Logo di Swagger-UI

- **Versione**: 3.1.0;
- **Utilizzo**: testing delle API_G.

Swagger-UI è uno strumento che offre un'interfaccia grafica per utilizzare le API_G all'interno di un progetto, senza necessariamente avere il codice dell'applicazione che le utilizza. In questo strumento, le API_G che richiedono parametri in *input* forniscono già il *template* dell'informazione in formato JSON, e, sempre nello stesso formato, restituiscono le risposte, che possono essere poi copiate o scaricate in un *file* dedicato. All'interno del progetto di *stage* questo strumento è stato particolarmente utile durante la configurazione del progetto per l'esecuzione in locale e successivamente per verificare il corretto funzionamento delle nuove API_G sviluppate.

2.2.2 Strumenti di analisi di rete

Wireshark

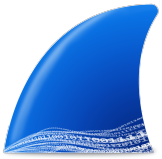


Figura 2.7: Logo di Wireshark

- **Versione**: 4.2.6;
- **Utilizzo**: analisi del traffico di rete in *localhost*.

Wireshark è un *software open-source* e *cross-platform* utilizzato per l'analisi della comunicazione all'interno di una rete e lo sviluppo di protocolli di comunicazione. Nella pagina principale permette la selezione della rete di cui si vuole analizzare il traffico, e, iniziata l'analisi, viene mostrata la lista di pacchetti di rete trasmessi, ognuno colorato in modo diverso secondo delle regole definite in automatico. Cliccando su un pacchetto è possibile poi vederne il contenuto.
All'interno del progetto, Wireshark è stato particolarmente utile nell'analizzare il traffico di rete in *localhost* per verificare la sicurezza della trasmissione delle credenziali di autenticazione dell'app.

2.2.3 Strumenti di collaborazione e gestione di progetto

Strumenti di Microsoft Office

Figura 2.8: Logo di Microsoft Outlook 2013

Figura 2.9: Logo di Microsoft Word 2013

Due *software* Microsoft molto utilizzati da VISIONEIMPRESA sono **Outlook** e **Word**. Il primo viene utilizzato per comunicare all'interno dell'azienda. Per comunicazioni più rapide e urgenti utilizzano l'applicazione **3CX**, ma per comunicazioni generali o per comunicare con gli stagisti viene utilizzato Outlook, all'interno del quale è stato creato un gruppo per facilitare l'invio di comunicazioni utili a tutti i dipendenti.
I documenti vengono invece scritti con Microsoft Word, e, se necessario, condivisi all'interno di un cloud aziendale chiamato Travaso, dove ogni dipendente ha una propria cartella personale. Tramite Word sono stati condivisi il piano di lavoro, il piano di test, l'analisi di moviEXPENSE e ho redatto la documentazione riguardante il lavoro svolto.

Git

Figura 2.10: Logo di Git

Git è un *software* per il controllo di versione (VCS) ideato da Linus Torvalds. È un VCS distribuito, di conseguenza non esiste un *database* centrale per la raccolta delle versioni, ma questo è distribuito ad ogni sviluppatore che sta lavorando al progetto.
Questa tipologia di *software* permette di organizzare i flussi di lavoro in *branch* (rami), facilitando quindi la collaborazione, e consente di raggruppare le modifiche effettuate al codice sorgente in *commit* a cui viene associato un messaggio che descrive l'obiettivo di tali modifiche e un codice identificativo creato da Git; in questo modo è più semplice identificare le modifiche effettuate e, in caso sia necessario, riportare il *software* ad uno stato ben preciso.
I VCS distribuiti sono spesso utilizzati in combinazione con un servizio di *hosting* per il codice, ad esempio GitHub, GitLab o Bitbucket. Quest'ultimo è stato utilizzato durante lo *stage* ed è descritto di seguito.

Suite Atlassian

Oltre agli strumenti di Microsoft Office citati in precedenza, l'azienda fa un forte utilizzo della *suite* **Atlassian**, in particolare di **Jira**, **Bitbucket** e **Confluence**. Durante lo *stage* sono stati utilizzati questi ultimi due.

Figura 2.11: Logo di Bitbucket

Bitbucket Servizio di *hosting* di *repository* basato su Git. Permette di caricare i progetti a cui si sta lavorando contestualmente alla cronologia dei *commit* di Git, condividendo così il lavoro svolto con tutti gli altri sviluppatori. Al suo interno è possibile creare *branch* e visualizzare le modifiche effettuate in un preciso *commit*.

Figura 2.12: Logo di Confluence

Confluence *Software* per la collaborazione *online* riguardo la stesura di documentazione. Viene utilizzato per scrivere la documentazione dei progetti *software* a cui si sta lavorando, offrendo un *editor* di testo integrato oppure permettendo di caricare documenti presenti in locale.

Capitolo 3

Design

In questo capitolo viene descritta la struttura dell'applicazione seguendo un approccio top-down, partendo quindi dal frontend per poi passare al backend descrivendo API$_G$ e database.

3.1 Premessa

Nonostante lo *stage* si basasse sul lavorare ad un'applicazione già esistente, è stato comunque necessario realizzare uno studio riguardo la struttura di tale app, cercando di individuarne architettura e *pattern* utilizzati. L'oggetto di questo capitolo, pertanto, non sarà la descrizione di un'architettura realizzata durante lo *stage*, bensì la descrizione dell'architettura già presente nell'applicazione, alla quale mi sono dovuto adattare per effettuare le modifiche e aggiunte necessarie al progetto.
Ho ritenuto quindi l'approccio *top-down* il più ragionevole da adottare, in quanto rispecchia la metodologia di studio da me seguita.

3.2 Panoramica

L'applicazione ha un'architettura a **microservizi**, come illustrato nella Figura 3.1. Il suo funzionamento si basa su tre *database* (CommonDb, SysDb e il *database* aziendale contenente i dati da utilizzare) e due tipologie di API$_G$ (*gateway* e *data provider*).
Per utilizzare l'applicazione è necessario effettuare una *login*. Questa utilizza le API$_G$ *gateway* che interrogano il CommonDb, in particolare la tabella `Utente`, che contiene *username* e *password* di tutti gli utenti dell'applicazione.
Dopo aver effettuato la *login* è possibile utilizzare l'app, e ogni volta che dei dati vengono caricati dal o nel *database* aziendale viene messa in atto la seguente procedura:

1. l'applicazione fa una chiamata alle API$_G$ *gateway*;
2. queste ricavano dal CommonDb l'url delle API$_G$ *data provider* dell'azienda associata all'utente registrato;
3. viene effettuata una chiamata alle API$_G$ dell'azienda (*data provider*), le quali interrogano il SysDb, per ricavare i parametri di connessione al *database* aziendale.

I dettagli dell'architettura di ogni parte dell'applicazione verranno illustrati nelle sezioni seguenti.

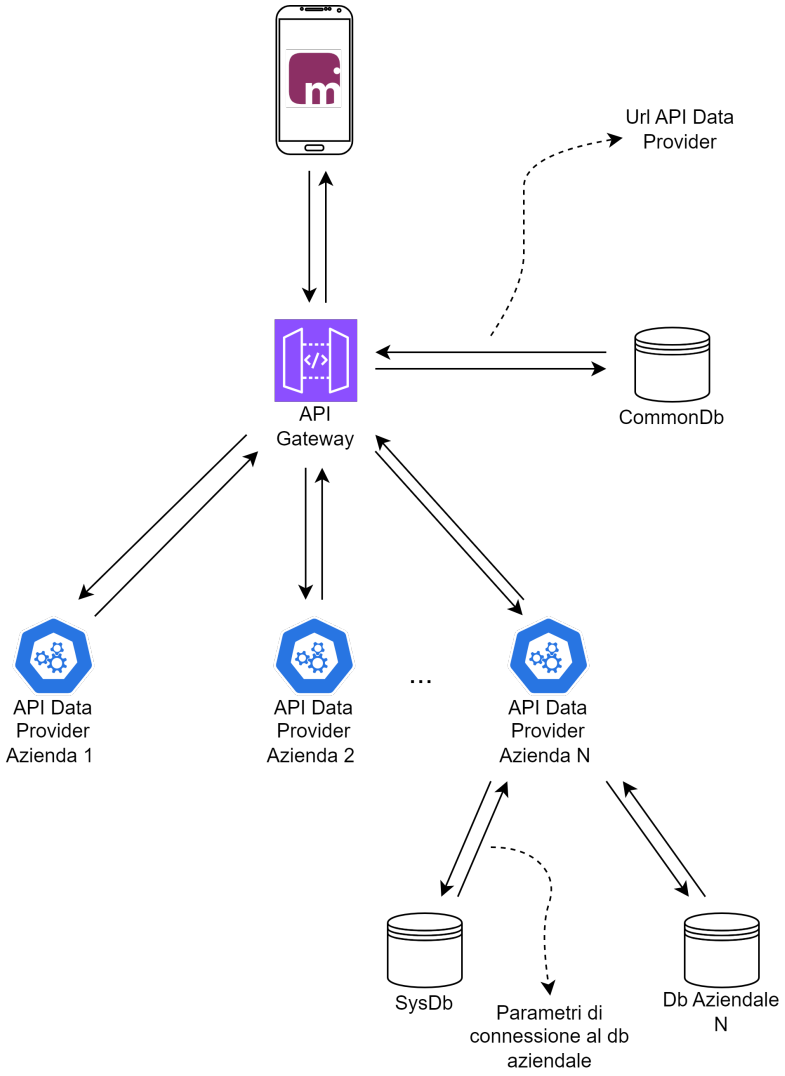


Figura 3.1: Architettura di moviEXPENSE

3.3 *Frontend*

Il *frontend* di moviEXPENSE è stato realizzato con il framework Xamarin, il quale impone l'utilizzo del *pattern* architetturale ***Model-View-ViewModel*** (**MVVM**). Questo *pattern* è molto utile per separare la logica dell'applicazione dall'interfaccia grafica, rendendo l'applicazione più semplice da modificare e testare. MVVM, infatti, separa il *frontend* in tre parti:

- **Model**: contiene tutte le classi dei dati di modello, ovvero tutti i dati strutturati che appartengono al dominio dell'applicazione;
- **View**: si occupa della rappresentazione grafica delle informazioni. Rappresenta dunque l'interfaccia utente, occupandosi della sua struttura e della sua presentazione;

- **ViewModel**: implementa le proprietà e i comandi con cui la *View* può effettuare dei *data binding*. Presenta anche un sistema di notifiche verso la *View* che permette di segnalare ogni cambiamento che avviene ai dati durante l'esecuzione del programma.

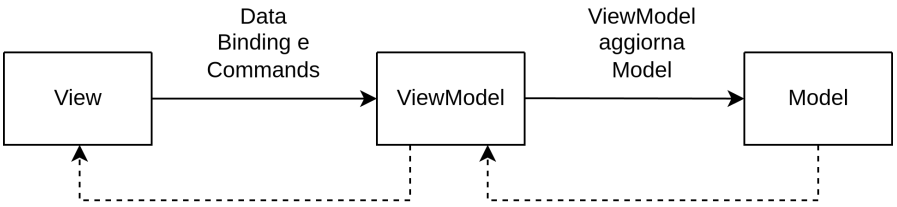


Figura 3.2: Struttura del *Model-View-ViewModel*

I tre *namespace* principali del *frontend* sono quindi:

1. `MoviExpense.Models`;
2. `MoviExpense.Views`;
3. `MoviExpense.ViewModels`.

3.3.1 MoviExpense.Models

Questo *namespace* contiene le classi che rappresentano gli oggetti di dominio dell'applicazione. Le quattro classi principali sono `User`, `Utente`, `Spesa` e `NotaSpese`.

User: è la classe utilizzata per l'autenticazione nel sistema. Contiene *username* e *password* presi dal form di *login*, un booleano che conferma o meno il salvataggio delle credenziali e il *token* JWT$_G$ di autenticazione, inizialmente lasciato vuoto e assegnato quando l'identità viene confermata.

Utente: rappresenta l'utente autenticato. È dotata di id, *username*, codice utente, email, descrizione e il valore `TariKm`, che indica la tariffa di rimborso chilometrico per spese di carburante.

Spesa: rappresenta l'oggetto principale dell'applicazione, ovvero la spesa che viene registrata. Contiene una serie di id oltre al proprio, che vanno ad identificare:

- la nota in cui è inserita;
- l'utente che registra la spesa;
- la causale;
- il tipo di pagamento;
- il fornitore presso cui è stata effettuata la spesa;
- la valuta con cui è stata effettuata.

Oltre a questi dati sono presenti le informazioni proprie della spesa, come l'importo, la data, il numero e la data della fattura, se emessa. Sono presenti poi alcuni *flag* booleani che indicano se è presente una fattura, se la spesa è

stata selezionata per essere inserita in una nota, se è rimborsabile e se è stata effettuata entro il comune di sede dell'azienda. Infine è presente un attributo che indica lo stato della spesa, il quale verrà spiegato nel dettaglio all'interno del prossimo capitolo.

NotaSpese: modella una nota spese, ovvero un insieme di spese da inviare all'amministrazione dell'azienda. Ogni nota è identificata da un id, ed è dotata di:

- descrizione (ad esempio "Spese rimborsabili luglio 2024");
- data in cui viene effettuata;
- un *check* che segnali se è una nota rimborsabile o meno;
- id dell'utente che registra la nota.

Sono presenti anche un *flag* `IsEditable` che indica se la nota è modificabile e un intero che rappresenta lo stato della nota. Come per le spese, anche lo stato delle note sarà illustrato nel dettaglio nel prossimo capitolo.

All'interno del *namespace* si trovano altre classi minori, che rappresentano attributi per le spese, come la causale, il tipo di pagamento o l'allegato, oppure altre entità dell'applicazione, ad esempio i totali delle spese divisi per determinate categorie.

3.3.2 MoviExpense.Views

Ogni classe di questo *namespace* rappresenta una pagina dell'applicazione. Le pagine sono scritte utilizzando XAML, linguaggio prediletto di Xamarin, e per ogni *file* `.xaml` che definisce la struttura e l'aspetto della pagina, è presente un *file* `.xaml.cs` che inizializza tutte le componenti, carica le informazioni statiche nella pagina (ad esempio stringhe) e definisce i *binding* con la *ViewModel*. Il nome di ogni *file* di questo *namespace*, e di conseguenza ogni classe, termina con la parola `"Page"`.

LoginPage: presenta il logo dell'applicazione e due *entry* per inserire *username* e *password*.

HomePage: contiene un breve testo di benvenuto e due pulsanti, uno per accedere alla pagina delle spese, l'altro per accedere alla pagina delle note. In basso è presente una *label* che mostra il numero di versione dell'app.

SpesePage: mostra la lista delle spese associate all'utente presenti nel *database*. Queste sono filtrabili in base allo stato della spesa oppure possono essere mostrate tutte insieme. Da qui è possibile eliminare una spesa che sia ancora modificabile, visualizzarne i dettagli, creare una nuova spesa oppure accedere alla pagina di visualizzazione dei totali delle spese (`TotaliSpesePage`).

ViewEditSpesaPage: questa pagina presenta una *form* per la creazione, modifica o solamente visualizzazione di una spesa. Da questa pagina è possibile raggiungere le pagine di selezione (`SelCauPage`, `SelValutaPage`, ...), nelle quali si possono selezionare alcuni campi della spesa, come la causale, la valuta, il tipo di pagamento e il fornitore.

NotePage: offre le stesse funzionalità di `SpesePage` ma è basata sulle note spese associate all'utente e presenti nel *database*. Non è possibile, però, visualizzare i totali.

ViewEditNotaPage: permette di compilare una *form* per la creazione di una nota spesa. Da qui si può accedere alla pagina `AddSpesePage`, utile ad inserire le spese desiderate nella nota, ed è possibile accedere alla visualizzazione dei totali di tali spese.

3.3.3 MoviExpense.ViewModels

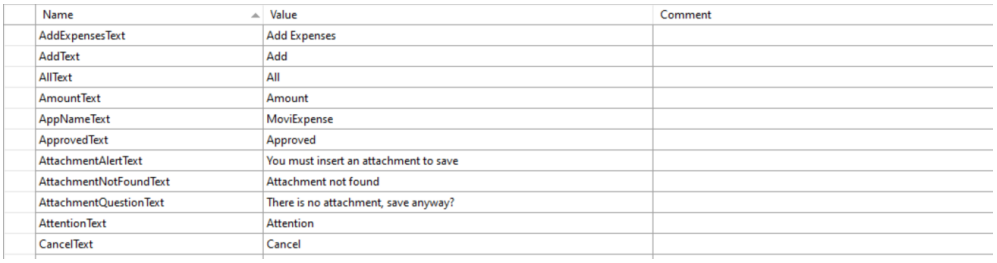
Per ogni classe della *View* e del *Model* è presente una classe di *ViewModel*, in particolare queste sono distinte dal nome: le classi associate alla *View* hanno lo stesso nome che hanno nella *View* ma sostituiscono la parola `"Page"` finale con `"Manager"` (ad esempio a `ViewEditSpesaPage` corrisponde `ViewEditSpesaManager`), mentre quelle associate al *Model* hanno lo stesso nome che hanno nel *Model* ma preceduto da `"Vm"` (alla classe `Spesa` corrisponde la classe `VmSpesa`).
Le classi `"Vm"` sono utilizzate dalla *ViewModel* per comunicare con il *Model*. Hanno come attributo un oggetto del *Model* di cui effettuano il *property wrap*, ovvero ne gestiscono i meccanismi di modifica e accesso. Questo permette di avere una migliore sincronizzazione dei dati tra *Model* e *View*, anche grazie al meccanismo di notifica presente in queste classi. Tutte le classi della *ViewModel* implementano l'interfaccia `INotifyPropertyChanged`, a cui appartiene il metodo `OnPropertyChanged` che permette di notificare alla *View* i cambiamenti nei dati del *Model*, così che possano essere effettuate le modifiche necessarie all'interfaccia grafica. Di seguito è riportato un esempio di *property wrapping*:

```
1     public int IdSpesa
2     {
3         get => Spesa.IdSpesa;
4         set
5         {
6             Spesa.IdSpesa = value;
7             OnPropertyChanged(nameof(IdSpesa));
8         }
9     }
```

Le classi `"Manager"`, invece, effettuano il *binding* con la *View* e si occupano di:

- comunicare con le API$_G$;
- gestire i dati non statici, ovvero di tutti i dati presi da *database*, e che possono quindi subire modifiche durante l'utilizzo dell'app;
- gestire i comandi inviati dalla *View*. Ad esempio quando si clicca l'icona per eliminare una spesa, viene chiamato un metodo di una classe `"Manager"` della *ViewModel*.

Oltre ai tre *namespace* appena descritti, ne sono presenti altri di supporto. Tra questi i principali sono: `MoviExpense.Converters` che contiene dei convertitori utili ad esempio ad assegnare proprietà booleane legate a valori non booleani, e `MoviExpense.Resources` che contiene tutte le stringhe dell'applicazione. L'immagine seguente riporta come viene visualizzato il *file* `Resources.resx` su Visual Studio.



Name	Value	Comment
AddExpensesText	Add Expenses	
AddText	Add	
AllText	All	
AmountText	Amount	
AppNameText	MoviExpense	
ApprovedText	Approved	
AttachmentAlertText	You must insert an attachment to save	
AttachmentNotFoundText	Attachment not found	
AttachmentQuestionText	There is no attachment, save anyway?	
AttentionText	Attention	
CancelText	Cancel	

Figura 3.3: *File* `Resources.resx` aperto in Visual Studio

3.4 *Backend*

3.4.1 API

Entrambe le API$_G$ si basano su un'architettura *layered* a tre livelli:

1. ***Presentation layer***: livello di accesso a cui arrivano le chiamate;
2. ***Business layer***: si occupa di elaborare le chiamate ed indirizzarle al *database*;
3. ***Data Access layer***: livello che modella i dati del *database* in oggetti di dominio.

Il funzionamento generale delle API$_G$ è il seguente:

1. viene effettuata una chiamata http dall'esterno (dal *frontend* o da un'altra API$_G$);
2. la chiamata viene intercettata da un oggetto `Controller` specifico che utilizza un oggetto `Service` per chiamare il metodo corrispondente alla chiamata http ricevuta;
3. il `Service` "inoltra" la chiamata a una classe specifica del *business layer*;
4. nel *business layer* si utilizza un oggetto `ExpContext` per accedere al *database*;
5. i dati prelevati o inseriti nel *database* sono modellati con un oggetto del *data access layer*.

Di seguito è riportato il funzionamento delle API$_G$ in modo *top-down* attraverso i *namespace* in cui avvengono le chiamate. Dato che entrambe le API$_G$ funzionano in modo analogo, nei nomi dei *namespace* sarà presente `[NomeAPI]`, che potrà essere quindi `"Gateway"` o `"DataProvider"`.

MoviExpense.Api.[NomeApi].Controllers

È presente una classe `Controller` per ogni classe del modello (`SpesaController`, `CausaleController`, ...). Questo *namespace* rappresenta il livello di accesso alle

API_G, ed è incaricato di accogliere le chiamate http emesse dall'esterno.
Ogni classe `Controller` ha un attributo della classe ad esso associata nel *namespace* seguente e un metodo per ogni chiamata che possono ricevere. Questi metodi sono identificati tramite attributi di *routing*, ad esempio la `Get` generale nella classe `SpesaController` ha il seguente attributo di *routing*

```
1     [HttpGet] //Attributo di routing
2     public IActionResult Get(int filterType, bool rimborsabile) {...}
```

Anche la classe stessa è identificata da diversi attributi, in particolare:

- `[Authorize]`: indica che è necessario essere autenticati per accedere alle funzionalità dell'API_G;
- `[Route("api/[controller]")]`: specifica parte dell'url dell'API_G;
- `[ApiController]`: consente di abilitare gli attributi di *routing*, le risposte automatiche per i codici di errore 400 e il *binding* con altri attributi.

All'interno di ogni metodo costruiscono un oggetto di tipo `Param`, costituito da *username* dell'utente e codice dell'azienda associata, e chiamano il metodo del `Service` corrispondente alla chiamata ricevuta. Nel caso della `Get` inserita prima viene chiamato `_service.GetAll(...)`, dove `_service` è di tipo `ISpesaService`.

MoviExpense.Api.[NomeApi].Services

È presente un'interfaccia e una classe concreta per ogni `Controller`, ad esempio sono presenti `ISpesaService`, `SpesaService`, `ICausaleService`, `CausaleService`, ...
Ogni classe concreta ha come attributo un oggetto del *business layer* tramite cui chiama il metodo corrispondente alla chiamata http ricevuta.
Continuando l'esempio iniziato precedentemente, ci sarà un oggetto `SpesaBL bl` che in un metodo `Get` chiamato dal `Controller` effettuerà la chiamata `bl.GetAll(...)`.

MoviExpense.Api.[NomeApi].BL e MoviExpense.Api.[NomeApi].DAL

`DAL` (*data access layer*) ha il compito di modellare i dati del *database* in oggetti di dominio, di conseguenza avrà una classe per ogni tipo di oggetto (`Spesa`, `Causale`, ...). Questi oggetti sono utilizzati nelle classi `BL` (*business layer*).
Nelle classi `BL` viene ricavata la stringa di connessione al *database* aziendale tramite il *database* Sys, e utilizzati degli oggetti `ExpContext` per effettuare la connessione ed eseguire le *query* necessarie a soddisfare la chiamata http ricevuta.

Gateway

Il funzionamento delle API_G *gateway* corrisponde a quello delle *data provider* solamente per effettuare `Get` sui dati dell'utente autenticato e per effettuare l'autenticazione (in questo caso c'è solamente una chiamata ulteriore). In tutti gli altri casi, il loro funzionamento è uguale solamente fino alla sezione riguardante le classi `Service`. Quando la chiamata arriva a una classe del *namespace* `MoviExpense.Api.Gateway.Services`, infatti, non viene inoltrata al *business layer* ma viene ricavato dal CommonDb l'url

delle API$_G$ *data provider* corrispondenti all'azienda dell'utente e la chiamata viene inoltrata a queste.

Autenticazione Come detto in precedenza, in caso di autenticazione le API$_G$ *gateway* funzionano allo stesso modo delle *data provider*, con una sola aggiunta: quando si arriva al *business layer*, dopo aver effettuato la *query* sul CommonDb per verificare che l'utente esista e le credenziali siano corrette (utilizzando una classe del *data access layer*), viene chiamato un metodo del *namespace* `MoviExpense.Api.Gateway.Authentication`, che contiene la classe `Engine`. Questa classe permette la gestione dell'autenticazione tramite JWT$_G$.

3.4.2 *Database*

Come si può vedere nella Figura 3.1, nel sistema dell'applicazione sono presenti tre *database*: CommonDb, SysDb e un *database* aziendale.

CommonDb

Viene utilizzato per:

- autenticazione (tabella `Utente`);
- recuperare delle informazioni dell'utente (tabella `Utente`);
- recuperare il numero di versione dell'applicazione (tabella `AppInfo`);
- recuperare l'url delle API$_G$ *data provider* (tabelle `UtenteAzienda` e `Azienda`).

SysDb

Contiene una sola tabella chiamata `Azienda`, che viene utilizzata per ricavare i parametri utili a costruire la stringa di connessione per il *database* aziendale.

Database aziendale

La maggior parte delle tabelle presenti in questo *database* sono autoesplicative e sono comunque già state spiegate in precedenza parlando del *Model*. L'unica informazione in più è la presenza delle tabelle `*GrpUtente`. Gli utenti dell'applicazione sono organizzati per gruppi. Ogni gruppo può avere accesso a determinate causali, tipi di pagamento e fornitori, di conseguenza sono presenti tutte le tabelle utili a collegare queste alla tabella `GrpUtente`, ad esempio `CausaleGrpUtente` definisce a quali causali hanno accesso i diversi gruppi utente.

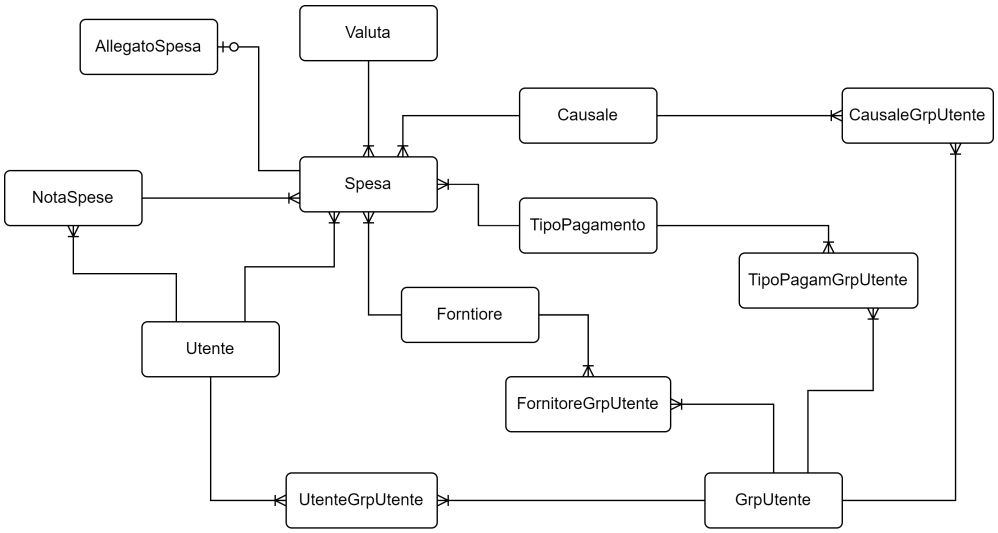


Figura 3.4: Schema entità-relazione *database* aziendale

Capitolo 4

Codifica e test

In questo capitolo verranno esposte in modo approfondito le principali attività di codifica effettuate nel progetto di stage. Per ogni argomento verrà esposta la richiesta effettuata dall'azienda, e successivamente l'attività svolta. Al termine del capitolo è presente una breve sezione riguardante i test.

4.1 Autenticazione

> Sono state segnalate dagli utilizzatori dell'app dei problemi di sicurezza riguardo il passaggio in chiaro di credenziali e stringhe di connessione.

Tramite Wireshark ho effettuato l'analisi del traffico di rete in *localhost* per verificare se c'erano realmente problemi di sicurezza, e in caso positivo, che dati riguardassero. La segnalazione degli utilizzatori dell'app si è rivelata essere un problema reale, in quanto sia le credenziali di *login* che stringhe di connessione ai *database* venivano passate in chiaro. Non era stato, infatti, implementato alcun sistema di cifratura delle *password* e la trasmissione delle informazioni avveniva sempre tramite protocollo http.
Per quanto riguarda la *login*, come consigliato, ho cercato di analizzare il sistema implementato in altre due app, moviORDER e moviCHECK, il quale è risultato più robusto. Ho implementato così un sistema di cifratura delle *password*, e per l'utente "demo" ho inserito la *password* cifrata all'interno del *database*.
Dopo aver effettuato un colloquio con l'amministratore dell'azienda, si è concluso che non fosse troppo ragionevole proseguire fino in fondo con il lavoro riguardante la sicurezza dell'applicazione, poiché risultava essere un'attività troppo grande e che richiedesse una certa attenzione anche da parte del *team* di sviluppo. L'attività riguardante la sicurezza è stata comunque considerata soddisfatta grazie alle analisi effettuate e alle proposte di implementazione presentate.

4.2 *Bug fixing* e nuove funzionalità

4.2.1 ChkGiustificativo

> È necessario dare diversi livelli di obbligatorietà all'allegato presente nella spesa (foto dello scontrino o della ricevuta di pagamento).

All'interno della tabella `Causale` nel *database* aziendale ho inserito una colonna chiamata `ChkGiustificativo`. Questa colonna ha come tipo `char(1)` e può assumere i seguenti valori:

- **A**: allegato opzionale con avviso se mancante. L'utente viene solamente avvisato in caso di mancanza dell'allegato, ma è possibile salvare comunque;
- **O**: allegato obbligatorio. Non è possibile salvare se l'allegato non è presente;
- **N**: nessun allegato. Non vengono effettuati controlli di presenza/assenza e si salva in ogni caso.

La modifica del *database* ha generato una serie di modifiche a cascata a partire dalle classi dei *data access layer* delle API_G fino a raggiungere le classi del *Model* nel *frontend*. Tutti i controlli vengono effettuati nel *ViewModel*, in particolare nella classe `ViewEditSpesaManager`, ovvero la classe *ViewModel* relativa alla pagina di modifica/creazione delle spese, ora è presente uno `switch` nella funzione di salvataggio della spesa appena creata o modificata.

```
1     public async Task<bool> SaveSpesaAsync()
2     {
3         ...
4         switch (ChkGiustificativo)
5         {
6             case 'A':
7                 checkResp = await CheckCaseA(uri, content);
8                 IsSaving = false;
9                 return checkResp;
10            case 'O':
11                checkResp = await CheckCaseO(uri, content);
12                IsSaving = false;
13                return checkResp;
14            case 'N':
15                var resp = await Client.PostAsync(uri, content);
16                IsSaving = false;
17                return resp.IsSuccessStatusCode;
18            default:
19                IsSaving = false;
20                ErrorChkGiustificativo();
21                return false;
22        }
23        ...
24    }
```

I metodi `CheckCaseA` e `CheckCaseO` effettuano dei controlli riguardo la presenza dell'allegato. Se questi controlli hanno esito positivo, allora procedono al salvataggio della spesa tramite chiamata API_G utilizzando i parametri passati (`uri` e `content`).

4.2.2 Data nota spese

> La data mostrata di *default* quando si crea una nuova nota spese dovrà essere parametrizzata per rispettare le specifiche esigenze aziendali.

Nella versione dell'app a cui ho lavorato, la data mostrata di *default* alla creazione di una nota spese era la data del giorno in cui la nota veniva creata. È possibile, però, che alcune aziende abbiano un protocollo per cui necessitano di avere le note spese dei dipendenti solamente a fine mese oppure secondo altri parametri temporali. Per poter rispondere a questa necessità, si è deciso di aggiungere una nuova tabella al *database* aziendale chiamata `Param`, la cui struttura è presentata in Figura 4.1.

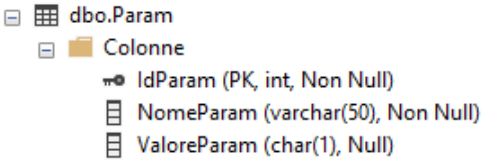


Figura 4.1: Struttura tabella Param da SSMS

Nella colonna `NomeParam` è stato inserito il parametro `DataNotaSpese`, che può assumere i seguenti valori:

- **S**: la data di *default* corrisponde alla domenica della settimana precedente;
- **M**: la data di *default* corrisponde all'ultimo giorno del mese precedente;
- **O**: la data di *default* è la data del giorno in cui viene creata la nota.

Il valore di questo parametro sarà poi impostato a seconda dei protocolli seguiti dall'azienda utilizzatrice.
Successivamente alla creazione della tabella ho creato tutte le API$_G$ necessarie e creato le funzioni che calcolano la data corretta in base al parametro all'interno del *frontend*, in particolare nella *ViewModel*.

4.2.3 Stati spese e note

Come anticipato nella Sezione 3.3.1, spese e note spese hanno un attributo chiamato `Stato`. Questo attributo indica la posizione in cui si trova una spesa/nota all'interno del suo ciclo di vita, il quale è riassunto negli schemi seguenti

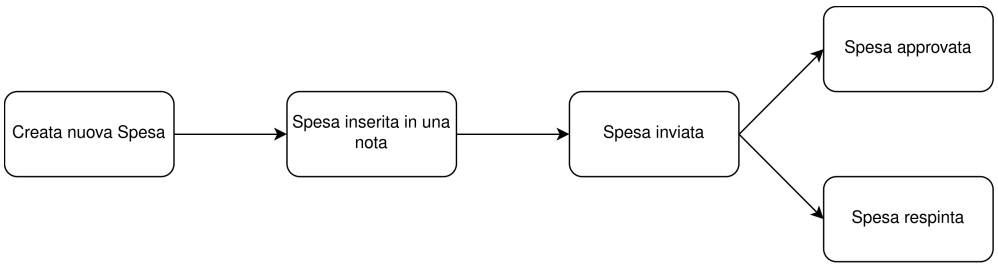


Figura 4.2: Ciclo di vita di una spesa

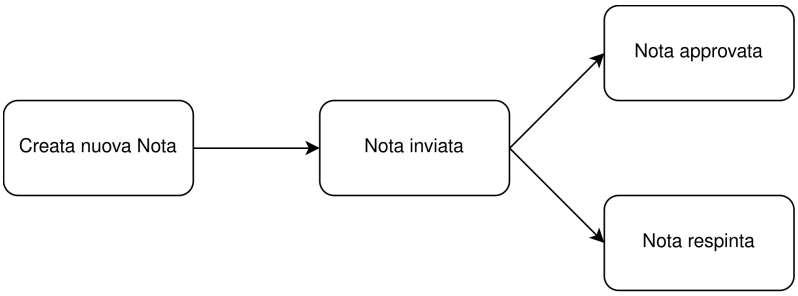


Figura 4.3: Ciclo di vita di una nota

Il sistema di stati già implementato nell'applicazione, però, non copriva tutto il ciclo di vita di spese e note spese, in quanto le spese potevano essere registrate, inserite in una nota o inviate, mentre le note spese inserite, inviate e ricevute. Di conseguenza è stato richiesto di rivedere il sistema di stati aggiungendo i mancanti e modificando a dovere quelli già presenti.
Il nuovo sistema di stati si presenta come segue.

Stati spese

- **0**: non in nota;
- **1**: in nota;
- **2**: inviata (inserita in una nota inviata);
- **3**: approvata;
- **4**: respinta.

I valori fino al 2 compreso si aggiornano automaticamente in base a ciò che avviene alla nota in cui è stata inserita la spesa, mentre il 3 e il 4 vengono aggiornati dall'amministrazione dell'azienda, che, presa una nota, può approvare o respingere le singole spese al suo interno.

Stati note

- **0**: registrata
- **1**: inviata;
- **2**: approvata;
- **3**: respinta.

Come nelle spese, gli stati corrispondenti ad "approvata" e "respinta" vengono aggiornati dall'esterno dell'applicazione.
Un cambiamento di stato in una nota causa cambiamenti di stato nelle spese che contiene, nello specifico:

- **0→1**: quando una nota viene inviata, tutte le spese al suo interno passano dallo stato 1 (in nota) allo stato 2 (inviata);

- **1→2**: quando una nota viene approvata, tutte le spese al suo interno, ad eccezione di quelle respinte, vengono approvate, quindi passano dallo stato 2 (inviata) allo stato 3 (approvata);
- **1→3**: quando una nota viene respinta, tutte le spese al suo interno vengono riportate allo stato 0 (non in nota) e la nota viene dunque "svuotata". La nota respinta rimane visibile all'interno dell'applicazione ma non è più modificabile. Le spese che conteneva, invece, possono essere modificate e inserite in un'altra nota.

Il seguente schema riassume il nuovo sistema di stati per le spese e le note:

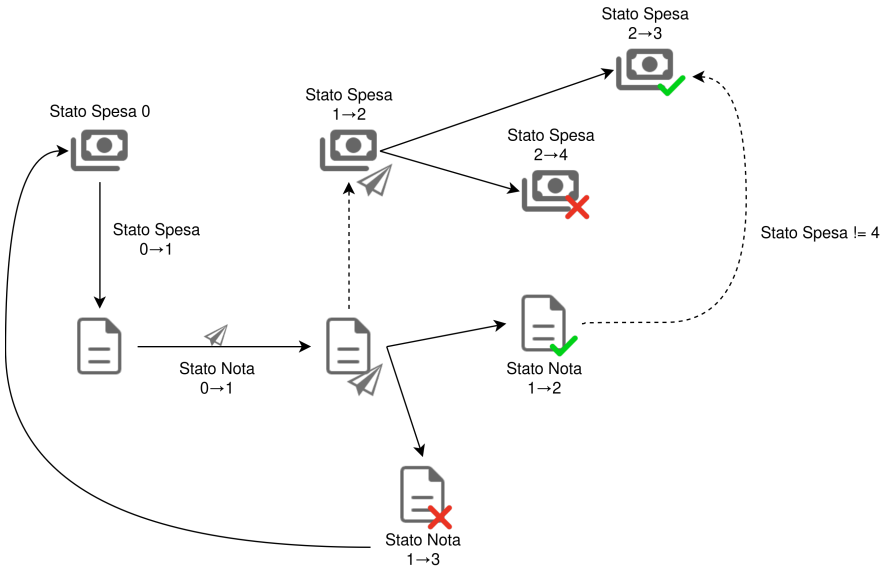


Figura 4.4: Funzionamento del nuovo sistema di stati per spese e note

Sistema di icone

Il nuovo sistema di stati implementato doveva avere anche un riscontro grafico per essere presentato all'utente, pertanto si è deciso di implementare un sistema di icone che lo rappresentasse. Questo sistema è parzialmente introdotto nella Figura 4.4. Nello specifico le icone utilizzate sono le seguenti:

(a) Cestino per stato spese/-note 0 **(b)** Nota per stato spese 1 **(c)** Aeroplano di carta per stato 1 note e 2 spese **(d)** Spunta verde per stato 2 note e 3 spese **(e)** Croce rossa per stato 3 note e 4 spese

Figura 4.5: Icone utilizzate per gli stati di note/spese

4.3 *Restyling*

4.3.1 Introduzione

La motivazione principale che ha spinto alla scelta di effettuare un *restyling* dell'applicazione è stata che essa si trovava ancora in uno stato primitivo sotto l'aspetto estetico. L'interfaccia grafica era poco curata e conteneva diversi errori nella sua presentazione. Dopo aver utilizzato l'app durante le prime settimane di *stage*, sono riuscito ad individuare le seguenti caratteristiche e problematiche:

- l'interfaccia era ricca di angoli e l'aspetto generale era molto spigoloso;
- alcuni elementi non risultavano ben allineati, e ciò portava ad una UI disordinata;
- alcuni elementi non venivano visualizzati correttamente poiché non gli veniva affidata una sufficiente quantità di spazio.

Dato che non erano presenti *mockup*$_G$ specifici per moviEXPENSE, sotto indicazione dell'amministratore di VISIONEIMPRESA mi sono stati condivisi i *mockup*$_G$ dell'applicazione moviORDER, così da avere un punto di riferimento per lo stile da seguire. Le due applicazioni, però, sono molto diverse tra loro e hanno un numero di funzionalità poco compatibile l'una con l'altra. I *mockup*$_G$ sono dunque serviti come ispirazione generale, da cui ho tratto delle linee guida da utilizzare per il mio lavoro.
Gli obiettivi perseguiti durante questa attività sono stati dunque:

- la riduzione degli angoli a favore di un *layout* più rotondo. A questo proposito è stato aggiunto all'interno del progetto il pacchetto `Xamarin.Forms.PancakeView`[1];
- sostituzione delle icone utilizzate con icone *Material Symbols* di Google Fonts in stile *rounded*[2];
- riordinamento di alcuni elementi per migliorarne la visualizzazione;
- aggiunta di *label* in alcune pagine, così da guidare l'utente durante l'utilizzo dell'app.

Un altro obiettivo del *restyling*, non legato però all'aspetto grafico dell'applicazione, è stato il raggruppare in classi gli elementi grafici che notavo si ripetevano molto spesso in modo uguale all'interno di più pagine, così da semplificare la loro modifica, evitando di dover ritoccare lo stesso codice su diverse parti dell'applicazione.

4.3.2 Implementazione

Per presentare i risultati del *restyling* ho deciso di utilizzare cinque pagine dell'applicazione, in quanto la struttura delle restanti è molto simile a quelle selezionate. Le pagine scelte sono:

- pagina di *login*;
- *home*;
- lista delle note spese;
- pagina di visualizzazione e modifica di una nota;
- pagina di aggiunta delle spese alla nota.

[1] https://github.com/sthewissen/Xamarin.Forms.PancakeView
[2] https://fonts.google.com/icons?icon.style=Rounded

Pagina di *login*

Nella pagina di *login* ho aggiunto qualche dettaglio come le *label* che indicano che informazioni inserire e le icone associate.

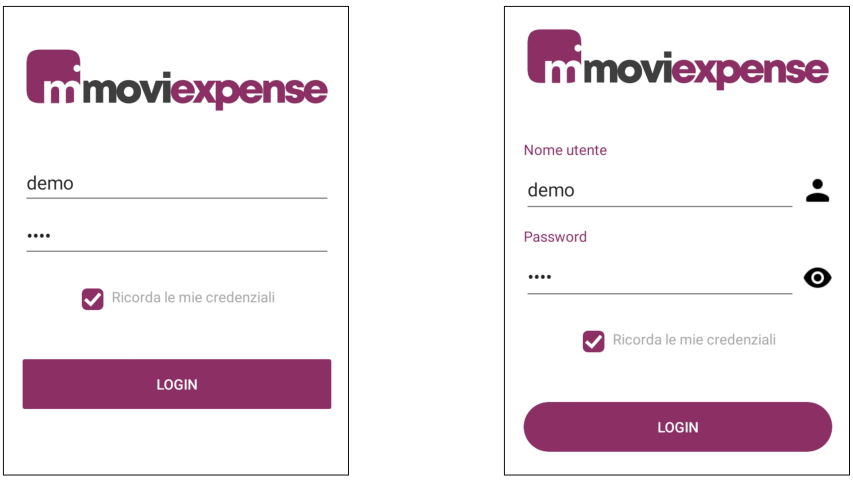


Figura 4.6: Pagina di *login* prima (sinistra) e dopo (destra) il *restyling*

L'aggiunta dell'icona associata alla *password* (icona dell'occhio) ha reso possibile anche l'implementazione di un pulsante che permette la visualizzazione della *password* in chiaro, con cambio di icona associato.

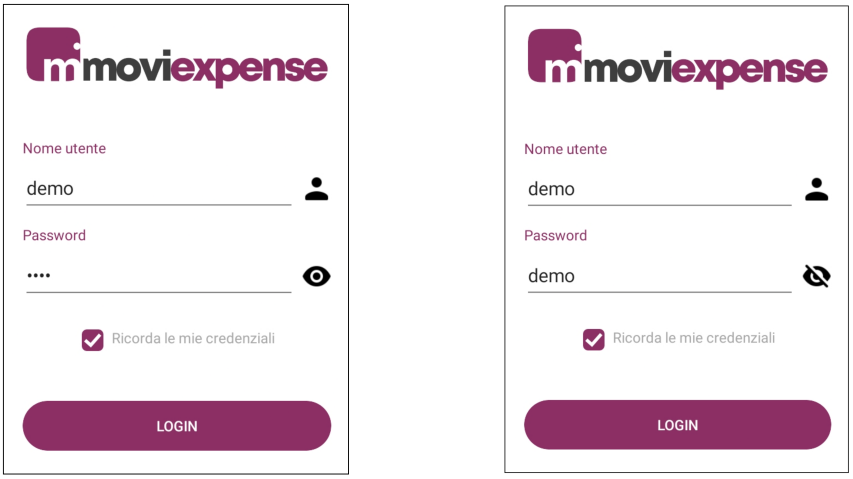


Figura 4.7: Visualizzazione in chiaro della *password* nella pagina di *login*

Home

Tra le implementazioni richieste durante lo *stage* c'era l'inserimento di un modo per visualizzare il numero di versione dell'app nella *home*. Ho scelto di implementarlo tramite una *label* in basso alla pagina, e questo ha causato a cascata una serie di modifiche: in primis ho aggiunto dei margini ai pulsanti "Spese" e "Note", per poi

arrotondarne gli angoli; successivamente ho inserito una `PancakeView` superiore, di cui ho arrotondato gli angoli inferiori, e ho cambiato il colore del testo di benvenuto in bianco.

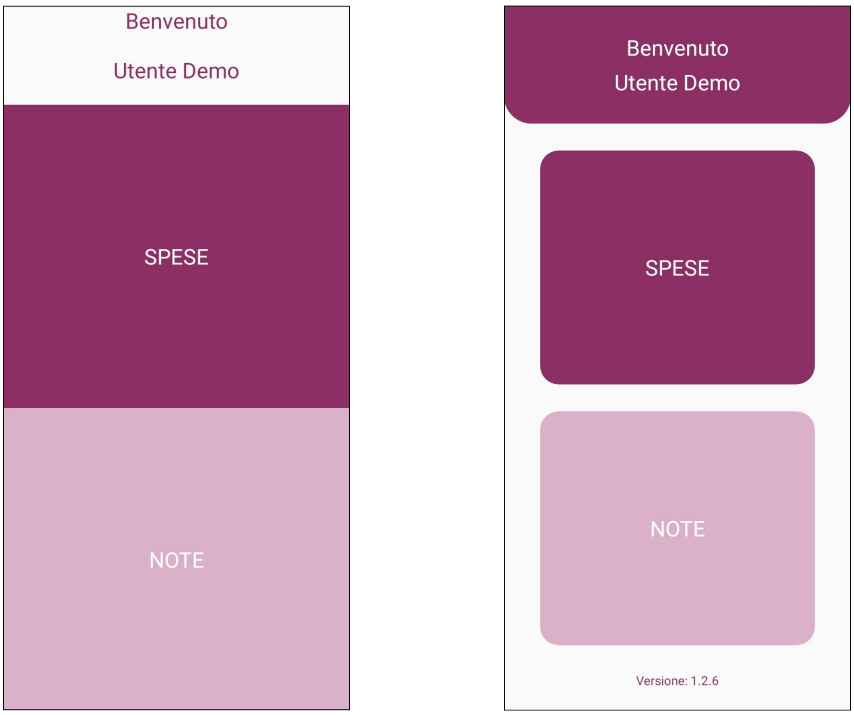


Figura 4.8: Pagina *home* prima (sinistra) e dopo (destra) il *restyling*

Lista delle note spese

Il *layout* generale della pagina delle note e di quella delle spese è stato modificato in modo analogo: ho aggiunto una `PancakeView` per la lista degli elementi e ho reso più grande la barra superiore, per favorire un migliore effetto visivo in combinazione con la nuova *view*. Ho rivisitato il *template* del singolo elemento della lista e aggiunto dei margini per migliorarne la visualizzazione. La dimensione del font nel nome del singolo elemento è stata ingrandita e le icone sono state sostituite dalle *Material Symbols*, in particolare ora per ogni nota è presente l'icona associata al suo stato (Sezione 4.2.3) e non più solo il cestino. In conseguenza all'aggiunta delle icone per la visualizzazione degli stati ho pensato fosse necessario evidenziare che l'icona del cestino fosse un pulsante, e quindi l'ho caratterizzata impostando un colore al `Button` presente in corrispondenza dell'icona.

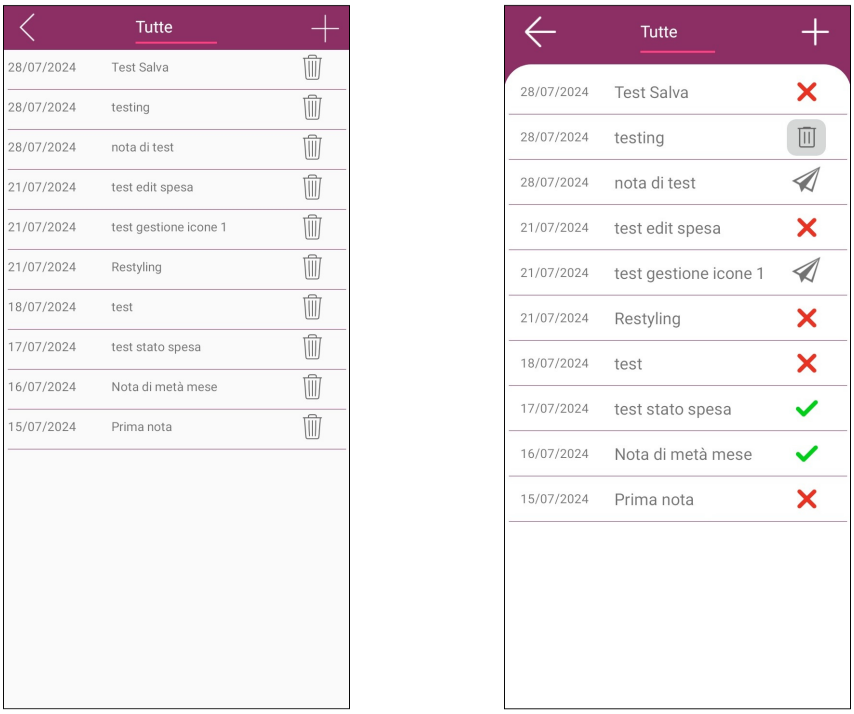


Figura 4.9: Pagina note prima (sinistra) e dopo (destra) il *restyling*

Pagina di visualizzazione e modifica di una nota

Anche qui, come in tutte le altre pagine, ho aggiunto una `PancakeView` e ingrandito la barra superiore. In questo caso ho stondato gli angoli del `Frame` interno con il *form* che presenta i dati della nota. La stondatura degli angoli ha seguito una basilare regola di UI *design*:

$$\text{innerRadius} = \text{outerRadius} - \text{gap}$$

dove `innerRadius` e `outerRadius` indicano rispettivamente la curvatura dell'angolo interno ed esterno, mentre `gap` è lo spazio tra l'elemento interno e quello esterno.
Ho riordinato ed allineato i dettagli delle singole spese presenti nella lista spese della nota, e infine ho spostato in basso il pulsante "Salva" che prima si trovata nella barra superiore, posizionandolo all'interno di una barra inferiore dedicata ed evidenziandolo di un colore diverso, in modo tale che risultasse in risalto. Questa scelta di posizionamento è stata fatta in quanto è il pulsante più importante della schermata, e in questo modo risulta più facilmente raggiungibile. Inoltre, dato che è un elemento presente in più pagine, ho ritenuto ragionevole raggruppare il codice che lo definisce all'interno di una classe chiamata `SaveButton`, nella quale sono definiti: colore del pulsante, colore del testo, margini e `CornerRadius`.

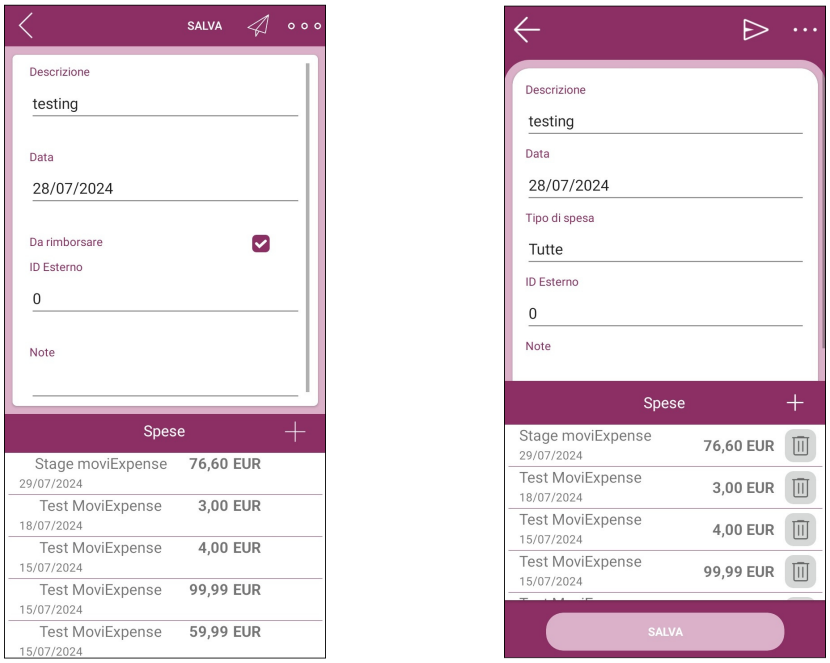


Figura 4.10: Pagina di visualizzazione e modifica nota prima (sinistra) e dopo (destra) il *restyling*

Quando la nota si trova in uno stato non più modificabile, quindi quando è stata inviata, approvata o respinta, tutti i campi modificabili e tutti gli elementi cliccabili correlati a modifiche vengono disabilitati (Figura 4.11). Inoltre, viene visualizzata nella barra superiore una *label* dove è scritto lo stato in cui si trova la nota, in modo tale che l'utente possa aver presente lo stato anche quando visualizza i dettagli della nota oltre che nella lista delle note.

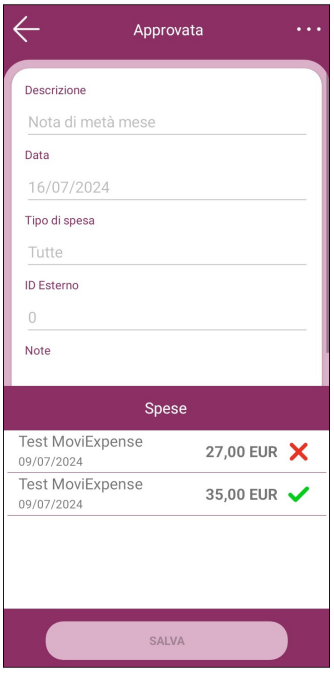


Figura 4.11: Visualizzazione nota non modificabile

Pagina di aggiunta delle spese alla nota

Per quanto riguarda la pagina di aggiunta delle spese a una nota, ho apportato le seguenti modifiche:

1. inserita `PancakeView` per il contenuto della pagina;
2. aggiunto dei margini laterali alle singole spese;
3. raggruppato importo e valuta in un unico elemento;
4. cambiato le icone;
5. spostato il pulsante "Salva" in basso e sostituito con un oggetto `SaveButton`, come nella pagina mostrata precedentemente, con la differenza che il testo del pulsante recita "Aggiungi" anziché "Salva";
6. aggiunto una *label* che indica l'oggetto della pagina.

Le modifiche n. 1, 4, 6 sono state applicate alle altre pagine simili a quella di aggiunta spese, ovvero le pagine di selezione: selezione della causale, selezione della valuta, selezione del tipo di pagamento e selezione del fornitore.

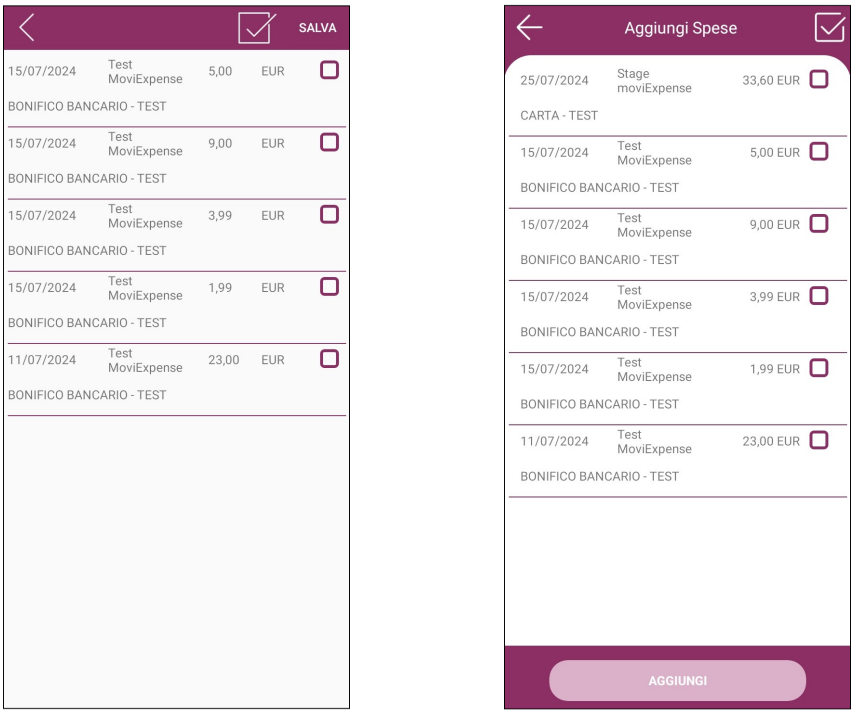


Figura 4.12: Pagina di aggiunta delle spese prima (sinistra) e dopo (destra) il *restyling*

4.4 Verifica e validazione

L'attività di verifica svolta in parallelo alla codifica si è basata principalmente sull'analisi statica del codice effettuata da Visual Studio con strumenti quali il controllo della sintassi e la segnalazione della presenza di parti di programma non raggiungibili. Insieme a questi strumenti, sono stati effettuati test manuali per verificare che le funzionalità e le aggiunte implementate svolgessero il compito previsto.
A termine dell'attività di codifica, mi è stato fornito dall'amministratore di VISIONEIMPRESA un piano di test specifico per moviEXPENSE. Questo piano prevedeva test manuali svolti eseguendo l'applicazione, per verificare che tutte le richieste effettuate fossero state soddisfatte e che fossero state implementate correttamente. Non è stata dunque necessaria la creazione di test automatici.
Per quanto riguarda invece la validazione del prodotto, periodicamente aggiornavo l'amministratore aziendale riguardo l'avanzamento del lavoro, e contestualmente mostravo ciò che avevo implementato o corretto, richiedendo quindi la sua approvazione. Al termine dello *stage* ho svolto una presentazione riassuntiva di tutto il lavoro svolto, supportata dalla documentazione realizzata, in particolare un documento tecnico riguardante le modifiche effettuate e le novità apportate, e delle *slide* aventi come oggetto il *restyling*.

Capitolo 5

Conclusioni

In questo capitolo vengono presentate le conclusioni tratte da questa esperienza di stage. La prima parte del capitolo si concentra sull'organizzazione e gli obiettivi del progetto, mentre la seconda parte è incentrata sulle riflessioni svolte a termine dell'esperienza.

5.1 Consuntivo orario

Settimana	**Attività**	**Ore**	**Variazione**
1	Studio	50	+10
2	Autenticazione	40	-
3-4-5	*Bug fixing* e novità	120	-
6	Interfacce	40	-
7-8	Test e documentazione	50	-10
Totale		300	±0

Tabella 5.1: Consuntivo del lavoro per settimane e ore

Come si può notare dalla tabella, ci sono state delle piccole variazioni nel numero di ore di alcune attività, ma queste non sono andate ad influenzare il monte ore finale.
Per quanto riguarda, invece, la distribuzione di tali attività nelle settimane, "*bug fixing*" e "novità" sono state inserite nella stessa riga, occupando le settimane dalla 3 alla 5. Questo perché le due attività sono state svolte insieme, e non in modo sequenziale, in quanto l'implementazione delle richieste sull'applicazione ha seguito l'ordine presente nel documento in cui tali richieste erano esposte, nel quale non era presente una divisione netta tra le richieste di *bug fixing* e di introduzione di novità.
La prima settimana di lavoro è durata più di quanto pianificato perché la configurazione dell'applicazione per la sua esecuzione in locale ha richiesto più tempo del previsto.

5.2 Obiettivi raggiunti

Al termine dello *stage* ho raggiunto tutti gli obiettivi obbligatori esposti nella Sezione 1.2.2 e quasi tutti i desiderabili. Le uniche due attività non svolte sono state la creazione del manuale utente dell'applicazione e la rivisitazione dell'interfaccia grafica per uso su *tablet*, a causa di mancanza di tempo, ma dato che avevano entrambe una bassa priorità, il livello di soddisfazione da parte dell'azienda è stato comunque alto.
Di seguito è riportata una tabella con le percentuali di raggiungimento degli obiettivi.

Categoria obiettivi	**Percentuale di raggiungimento**
Obbligatori	100%
Desiderabili	75%
Opzionali	0%
Totale	83,33%[1]

Tabella 5.2: Percentuali di raggiungimento obiettivi per categorie

5.3 Conoscenze acquisite

Grazie a questo *stage* ho potuto affacciarmi al mondo della programmazione *mobile*, argomento che non avevo trattato né all'università e nemmeno individualmente, poiché non mi attirava in modo particolare. In questo contesto ho migliorato le mie capacità di autoapprendimento legate sia allo studio di nuovi strumenti e tecnologie che all'utilizzo e lo studio di software realizzato da altri, comprendendone l'architettura, l'organizzazione delle cartelle e lo stile di codifica.
Benché il *framework* utilizzato per il *frontend* (Xamarin) non sia più supportato, ho comunque ritenuto utile il suo studio, in quanto è in ogni caso la base di partenza per il *framework* che l'ha sostituito, ovvero .NET MAUI, e perciò se in futuro mi verrà chiesto di sviluppare applicazioni utilizzando questo nuovo *framework*, avrò già una conoscenza di base che mi permetterà di apprenderlo più facilmente.

5.4 Valutazione personale

Nel complesso ho ritenuto lo *stage* un'esperienza molto utile, in quanto mi ha permesso innanzitutto di interfacciarmi con il mondo del lavoro, in particolare con la "vita d'ufficio", esperienza che non avevo mai svolto. Ritengo quindi importante anche lo svolgimento di un'esperienza di questo tipo in presenza piuttosto che da remoto, specialmente se è la prima esperienza lavorativa che si affronta, in quanto la comunicazione con i colleghi risulta più rapida ed efficiente rispetto ad una comunicazione via mail, messaggio o chiamata, nelle quali possono presentarsi diversi problemi, dalla mancata visualizzazione di un messaggio a problemi audio/video.

[1] Ho utilizzato il numero di obiettivi di ogni categoria come "peso" per il calcolo della percentuale totale.

$$P_{tot} = \frac{n_1 P_1 + n_2 P_2 + n_3 P_3}{n_1 + n_2 + n_3} = \frac{7 \cdot 100 + 4 \cdot 75 + 1 \cdot 0}{7 + 4 + 1} = 83,33$$

Lo *stage* mi ha anche fatto ricredere riguardo la programmazione *mobile*, la quale è risultata molto affascinante, anche durante l'attività di *restyling*, che non avrei pensato mi sarebbe interessata particolarmente.
L'ambiente che ho trovato presso VISIONEIMPRESA, inoltre, è stato un ambiente molto accogliente, amichevole e con persone fortemente disponibili in caso avessi dubbi o difficoltà, e ciò ha contribuito a rendere piacevole questa esperienza, evitando di essere vissuta come semplice attività obbligatoria del piano di studi.
Infine ritengo che effettuare uno *stage* al termine del percorso di studi sia particolarmente utile, poiché permette di:

- avvicinarsi ad argomenti poco conosciuti e quindi apprendere strumenti, tecnologie e metodologie nuove che possono arricchire il proprio bagaglio culturale e professionale;
- confermare o meno interessi per futuri lavori o percorsi di studio;
- conoscere persone nell'ambito professionale di interesse, con cui scambiare opinioni e conoscere meglio l'ambiente lavorativo.

Acronimi e abbreviazioni

API Application Program Interface. 2, 7, 9, 12, 16–19, 22, 23

IDE Integrated Development Environment. 8, 37

JWT JSON Web Token. 14, 19

SSMS SQL Server Management Studio. vii, 23

UI User Interface (Interfaccia Utente). 26, 29

Glossario

Application Program Interface In italiano, interfaccia di programmazione di un'applicazione. Insieme di regole e di protocolli che permette la comunicazione e lo scambio di dati tra più *software* o parti di *software*. 36

Code-behind Pratica di porre in due *file* separati il codice che gestisce la struttura e presentazione di un'applicazione e quello che ne gestisce il comportamento. È una pratica utilizzata principalmente nella programmazione in ambienti Microsoft dove il codice di struttura e presentazione è definito nei *file* `.xaml`, mentre il comportamento è definito nei *file* `.cs`. 6, 7

Codebase Insieme del codice sorgente che sta alla base di un *software*. 6

Garbage collector Sistema di gestione automatica della memoria allocata da un programma in esecuzione. Rileva i blocchi di memoria non più utilizzati e li libera per migliorare le *performance* del programma che sta eseguendo. 7

Hot reload Funzionalità presente in alcuni IDE$_G$ che permette di applicare le modifiche effettuate ad un'applicazione durante la sua esecuzione, senza dover quindi chiudere l'app e rieseguirla. 8

Integrated Development Environment In italiano, ambiente di sviluppo integrato. Applicazione utilizzata per lo sviluppo *software* che offre, oltre ad un *editor* di testo, diverse funzionalità che aiutano il programmatore durante la scrittura del codice, ad esempio sistemi di automazione e *debugging*. 36

JSON Web Token Standard *web* per lo scambio di dati basato su un oggetto chiamato *token* suddiviso in tre parti: *header*, che contiene le informazioni che identificano l'algoritmo di codifica utilizzato, *payload*, che contiene i dati codificati e in formato JSON, e una *signature*, ovvero una firma che dà validità al *token*. 36

Libreria Insieme di risorse in sola lettura che possono essere incluse e utilizzate all'interno di un *software*. 7

Mockup Modello utilizzato per mostrare il *design* di un prodotto. In informatica è il risultato della progettazione di un'interfaccia grafica, utilizzato come riferimento per la sua implementazione. 3, 26

Sitografia

Siti web consultati

.NET. URL: https://learn.microsoft.com/en-us/dotnet/core/introduction.

C#. URL: https://en.wikipedia.org/wiki/C_Sharp_(programming_language).

Model-View-ViewModel. URL: https://learn.microsoft.com/en-us/previous-versions/xamarin/xamarin-forms/enterprise-application-patterns/mvvm.

SQL. URL: https://en.wikipedia.org/wiki/SQL.

VISIONEIMPRESA s.r.l. Società Benefit, pagina software. URL: https://www.vsh.it/i-nostri-software/.

Visual Studio. URL: https://it.wikipedia.org/wiki/Microsoft_Visual_Studio.

Xamarin.Forms. URL: https://learn.microsoft.com/en-us/previous-versions/xamarin/get-started/what-is-xamarin-forms.