UNIVERSITÀ DEGLI STUDI DI MODENA E

REGGIO EMILIA

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea Triennale in Ingegneria Informatica

Implementazione di Funzionalità Aggiuntive per l’Applicazione Web Missioni Basata sul Web Framework Django

Tutor:

Prof. Federico Bolelli

Candidato:

Riccardo Masetti

Matricola 165910

ANNO ACCADEMICO 2023/2024

Indice

Elenco delle figure

Elenco dei Codici

Introduzione

La gestione delle spese sostenute durante le missioni universitarie, con le conseguenti richieste di rimborso all'organizzazione committente, può risultare complessa e dispendiosa in termini di tempo.

"Missioni" è un'applicazione web esistente progettata per semplificare tale processo, offrendo uno strumento semplice ed intuitivo per la compilazione semi-automatica dei documenti di richiesta di rimborso, sulla base delle spese effettuate durante le missioni.

Il presente elaborato si propone di illustrare le modifiche introdotte nell'applicazione, con particolare attenzione all'implementazione di nuove funzionalità volte a migliorare l'esperienza utente su dispositivi mobili e a offrire servizi aggiuntivi precedentemente non disponibili. Le modifiche apportate al progetto si articolano in tre aree principali: la prima riguarda l'adattamento dell'interfaccia grafica per garantire una visualizzazione ottimale di tutte le pagine dell'applicazione su dispositivi mobili. La seconda parte si occupa dell'integrazione di una funzione che permette di allegare immagini degli scontrini, facilitando una documentazione efficace delle spese effettuate. Infine, un terzo aspetto fondamentale riguarda la migrazione dei dati dal vecchio al nuovo formato, garantendo la continuità e l'integrità delle informazioni esistenti.

L'elaborato è suddiviso in quattro capitoli, che partono da un'analisi iniziale delle tecnologie utilizzate e delle caratteristiche dell'applicazione, per arrivare alla descrizione dettagliata delle nuove implementazioni. Nello specifico, la struttura è organizzata come segue:

- Nel **Capitolo 1** viene fornita una panoramica generale su **Python**, illustrando le sue caratteristiche principali, le sue funzionalità e le sue potenzialità nello sviluppo Web.
- Il **Capitolo 2** è dedicato all'approfondimento delle principali tecnologie impiegate nel progetto. In particolare, verrà trattato il framework **Django**, con una descrizione dei suoi componenti, del suo funzionamento e dei vantaggi che offre per lo sviluppo di applicazioni web. Inoltre, verrà brevemente presentato **Bootstrap**, framework open source per lo sviluppo di interfacce web responsive.

- Il **Capitolo 3** descrive la struttura del progetto, mettendo in luce le caratteristiche principali dell'applicativo e i punti critici che hanno motivato le modifiche apportate.
- Nel **Capitolo 4** vengono infine riportate e descritte le **modifiche**. Verranno illustrate con precisione le nuove funzionalità, spiegando quali sono state le scelte implementative per il raggiungimento del risultato finale.

1. Python

Python [1] è un linguaggio di programmazione interpretato, orientato agli oggetti, di alto livello e con semantica dinamica. Si distingue per la semplicità della sua sintassi e la leggibilità del codice, caratteristiche che, combinate con le strutture dati integrate ad alto livello, la tipizzazione dinamica e il binding dinamico, lo rendono particolarmente efficace. Tali caratteristiche hanno contribuito in maniera significativa alla crescente popolarità di Python, in particolare negli ultimi anni.

1.1 Storia

La storia [2] di Python inizia alla fine degli anni '80, quando Guido van Rossum iniziò a lavorare ad esso come un progetto hobbistico. Python è considerato il successore del linguaggio di programmazione ABC, un linguaggio che van Rossum aveva contribuito a creare all'inizio della sua carriera. ABC possedeva alcune caratteristiche innovative, come la gestione delle eccezioni e una sintassi chiara e leggibile, ma anche altrettanti difetti. Van Rossum, riconoscendo i punti di forza di ABC, ne prese le caratteristiche migliori, risolvendo al contempo le sue problematiche. Fu così che nel 1991 fu rilasciato per la prima volta Python[1]. Attualmente, lo sviluppo di Python viene gestito dall'organizzazione no-profit *Python Software Foundation*. Con il rilascio della versione 3.0 di Python nel 2008 viene semplificato il linguaggio introducendo diversi miglioramenti, come ad esempio l'uso predefinito di stringhe Unicode.

1.2 Funzionalità e Caratteristiche

Python viene generalmente presentato come un **linguaggio interpretato**[2], anche se in realtà combina caratteristiche sia dei linguaggi interpretati che di quelli compilati. In particolare, un interprete si occupa di analizzare il codice sorgente e, se sintatticamente corretto, compilarlo in formato *byte-*

[1] Il nome deriva dalla commedia *Monty Python's Flying Circus*, in onda sulla BBC nel corso degli anni Settanta.

[2] Linguaggio il cui codice sorgente viene eseguito direttamente da un interprete, senza la necessità di una compilazione preliminare

code per poi eseguirlo. Questo processo permette di mantenere la semplicità e la flessibilità tipiche dei linguaggi interpretati, garantendo al contempo buone prestazioni grazie alla pre-compilazione in bytecode.

Un'altra importante caratteristica di Python è che si presenta come un linguaggio **multi-paradigma** in quanto supporta diversi tipi di programmazione, tra cui: programmazione strutturale, programmazione funzionale e programmazione ad oggetti. Ciò lo rende un linguaggio molto flessibile e adatto all'utilizzo in numerosi contesti.

Python è un linguaggio tipizzato, il che significa che ad ogni variabile è associato un tipo che descrive il tipo di dato che essa può contenere. Tuttavia, a differenza di molti altri linguaggi come C e Java, Python utilizza una **tipizzazione dinamica**. In altre parole, il tipo di una variabile non deve essere dichiarato esplicitamente, ma viene assegnato dinamicamente a runtime in base ai dati che la variabile stessa contiene.

Un altro aspetto molto apprezzato in Python riguarda la gestione della memoria. Essa avviene attraverso il sistema di **Garbage Collection** [3], un meccanismo automatizzato che gestisce l'allocazione e il rilascio della memoria, assicurando che la memoria occupata da oggetti non più utilizzati venga recuperata. Python utilizza un algoritmo di conteggio dei riferimenti per tracciare il numero di riferimenti attivi a ciascun oggetto. Quando il conteggio dei riferimenti di un oggetto scende a zero, l'oggetto viene automaticamente rilasciato. Questo meccanismo consente di risolvere vari problemi, tra cui la liberazione prematura della memoria o l'eventuale omissione nel rilascio della stessa.

1.3 Potenzialità

Python è attualmente uno dei linguaggi di programmazione più diffusi al mondo, grazie alla sua estensibilità offerta da un vasto numero di librerie e framework di terze parti. Questa caratteristica lo rende uno strumento estremamente versatile e valido in numerosi ambiti di applicazione. Ad esempio, librerie come **NumPy** e **Pandas** sono essenziali per l'analisi dei dati e la manipolazione di grandi dataset, mentre **TensorFlow** e **PyTorch** sono ampiamente utilizzati per applicazioni di machine learning e intelligenza artificiale. Inoltre, **Django** e **Flask** sono framework popolari per

lo sviluppo web, che permettono di creare applicazioni web robuste e scalabili con relativa facilità. Python è anche molto apprezzato nella comunità scientifica grazie a strumenti come **SciPy** e **Matplotlib**, che facilitano la modellazione, la simulazione e la visualizzazione dei dati scientifici. Il grande successo di Python è ben testimoniato da uno studio condotto nel 2023 da *GitHub* relativamente ai linguaggi più utilizzati sulla piattaforma [4].

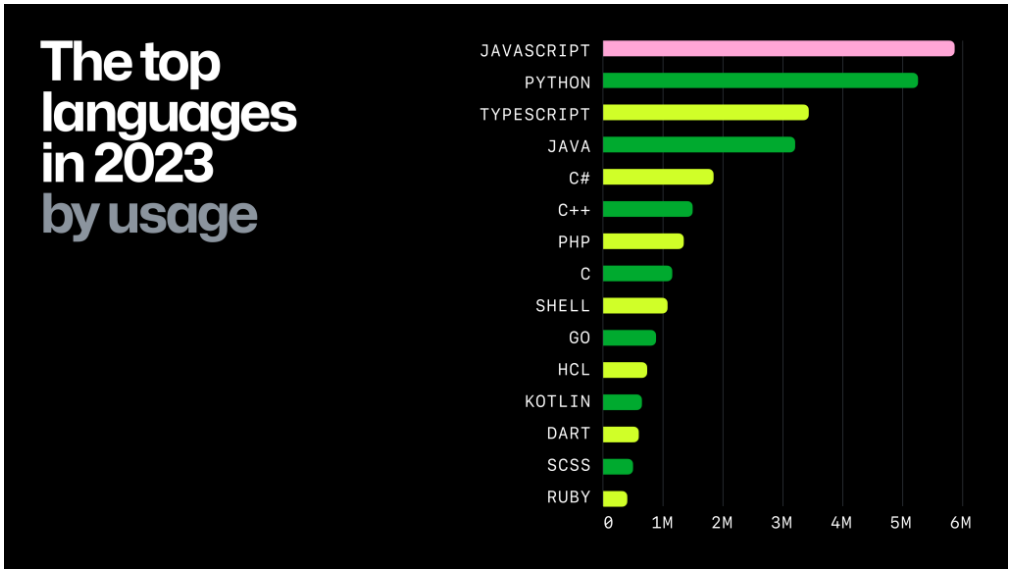

Figura 1.1: Top languages in 2023 by usage (GitHub Survey).

1.4 Web Development

Nel 2018, JetBrains ha condotto uno studio intervistando numerosi sviluppatori per comprendere i principali impieghi del linguaggio [5]. I risultati hanno rivelato che lo **sviluppo web** è al secondo posto tra le applicazioni più comuni di Python, sottolineando la sua rilevanza in questo settore.

Come già detto, i due framework più popolari per il web development sono Django e Flask:

- **Django**: è un framework ad alto livello che fornisce numerose funzionalità pronte all'uso, come l'autenticazione degli utenti, la gestione del database e la creazione di interfacce amministrative; permettendo agli sviluppatori di costruire rapidamente applicazioni web robuste e scalabili;

- **Flask**: è un microframework leggero che offre maggiore flessibilità, consentendo agli sviluppatori di scegliere e integrare solo i componenti di cui hanno bisogno.

Entrambi i framework facilitano la creazione di applicazioni web moderne e sicure, promuovendo pratiche di sviluppo rapide ed efficienti.

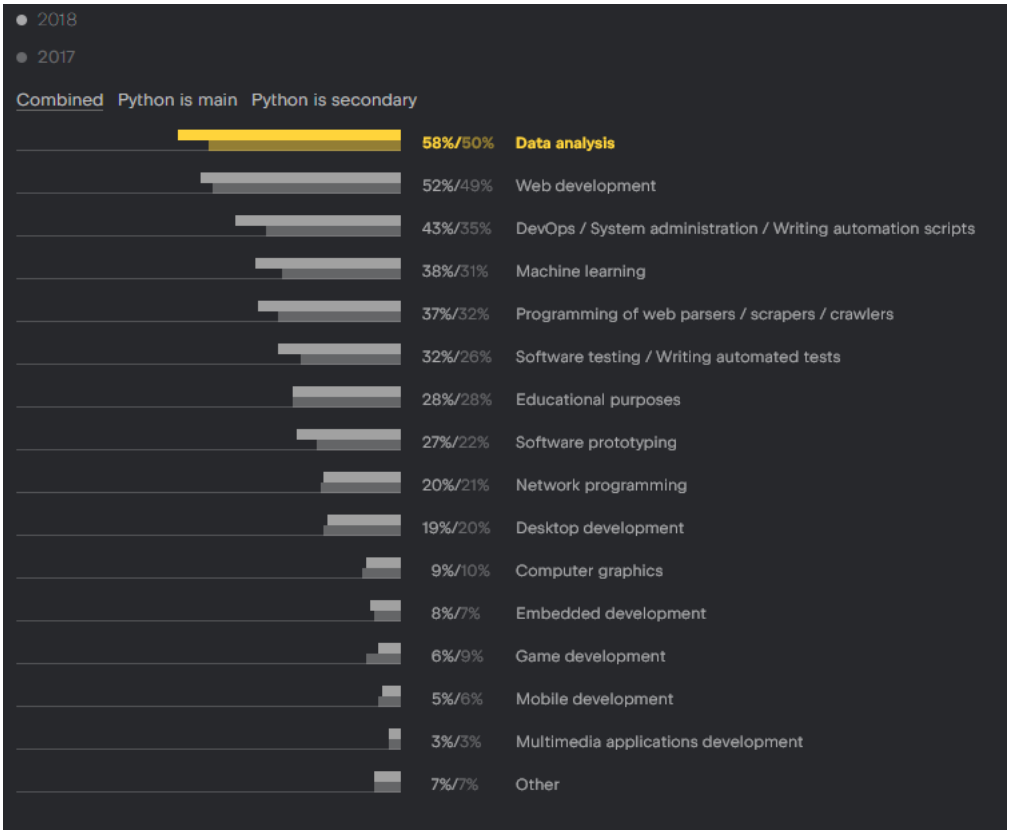

Figura 1.2: Main areas of application for Python (Python Developers Survey 2018).

2. Tecnologie e Strumenti

La seguente sezione è dedicata a un'analisi delle tecnologie impiegate nello sviluppo dell'applicazione. Verranno esaminati i principali strumenti e framework utilizzati, con particolare attenzione a Django e Bootstrap.

2.1 Django

Nato nel 2005, Django è un framework web di alto livello basato su Python, ideato per semplificare la creazione di applicazioni web complesse e scalabili. Gestito dalla *Django Software Foundation*, Django è un progetto gratuito e open source con una comunità attiva e in continua crescita. La sua eccellente documentazione e le numerose opzioni di supporto, sia gratuite che a pagamento, contribuiscono a renderlo uno degli strumenti più potenti e versatili nel panorama dello sviluppo web.

2.1.1 Principi di Sviluppo

Django si basa su una serie di principi fondamentali [6] che guidano lo sviluppo delle applicazioni e l'architettura del framework:

- **Loose Coupling:** Un obiettivo chiave di Django è il disaccoppiamento stretto. I vari strati del framework dovrebbero *conoscersi* solo quando strettamente necessario. Ad esempio, il sistema di template non sa nulla delle richieste web, il layer del database non conosce la visualizzazione dei dati e il sistema di view non si preoccupa del sistema di template utilizzato dal programmatore. Sebbene Django offra uno stack completo per comodità, i componenti sono indipendenti l'uno dall'altro ovunque possibile;
- **Less Code:** Le applicazioni Django dovrebbero utilizzare il meno codice possibile, evitando ridondanza nelle varie sezioni;

- **Quick Development:** Lo scopo di un framework web è rendere rapide le attività tediose dello sviluppo web. Django consente uno sviluppo web incredibilmente veloce;
- **Don't Repeat Yourself (DRY):** Ogni concetto o dato univoco dovrebbe essere definito in un unico luogo nel codice. La duplicazione è da evitare perché aumenta la complessità e il rischio di errore;
- **Explicit is Better than Implicit:** Seguendo un principio fondamentale di Python, Django cerca di evitare la *magia*[1] nascosta nel codice. Ciò significa che il comportamento del codice dovrebbe essere chiaro e comprensibile, senza operazioni nascoste o implicite;
- **Consistency:** Il framework deve essere coerente a tutti i livelli, dalla codifica Python a basso livello all'esperienza complessiva di utilizzo di Django.

2.1.2 Potenzialità

Sulla base dei principi sopra riportati, Django è stato progettato con l'obiettivo di essere:

- **Completo:** Ogni componente funziona coerentemente con gli altri, segue principi di progettazione software e dispone di una documentazione ampia e aggiornata;
- **Versatile:** Può essere utilizzato per costruire qualsiasi tipo di sito web, dai sistemi di gestione dei contenuti ai social network, funzionando con qualsiasi framework lato client e fornendo contenuti in vari formati utili per lo sviluppo web, come HTML, RSS, JSON e XML;
- **Sicuro:** Abilita misure di protezione predefinite contro numerose vulnerabilità, come *SQL Injection* (inserimento di codice SQL malizioso), *Cross-Site Scripting* (XSS, iniezione di script dannosi in pagine web), *Cross-Site Request Forgery* (CSRF, esecuzione non autorizzata di azioni su un'applicazione), e *Clickjacking* (induzione a cliccare su elementi ingannevoli);
- **Scalabile:** Ogni componente è indipendente dalle altre, permettendo di aggiungere hardware a qualsiasi livello in base alle necessità di traffico;

[1] Termine usato per riferirsi a quei comportamenti del codice che avvengono in modo automatico e nascosto.

- **Mantenibile:** Il codice segue il principio DRY, evitando duplicazioni inutili. Raggruppato in moduli riutilizzabili secondo il modello Model-View-Controller (MVC), è facilmente mantenibile per lo sviluppatore;
- **Portabile:** Scritto in Python, le applicazioni web possono essere eseguite su varie versioni di Linux, Windows, Mac OS X e provider di web hosting.

2.1.3 Il Paradigma MVC in Django

L'architettura di Django si basa sul design pattern **MVC**, acronimo di *Model-View-Controller*. Lo scopo di questa scelta di design è la suddivisione della logica in tre parti fondamentali favorendo modularità, sviluppo collaborativo e riutilizzo del codice. In particolare, il pattern si propone di separare la rappresentazione del modello dei dati (**Model**), l'interfaccia utente (**View**) e la logica applicativa (**Controller**), anche detta logica di *"business"*.

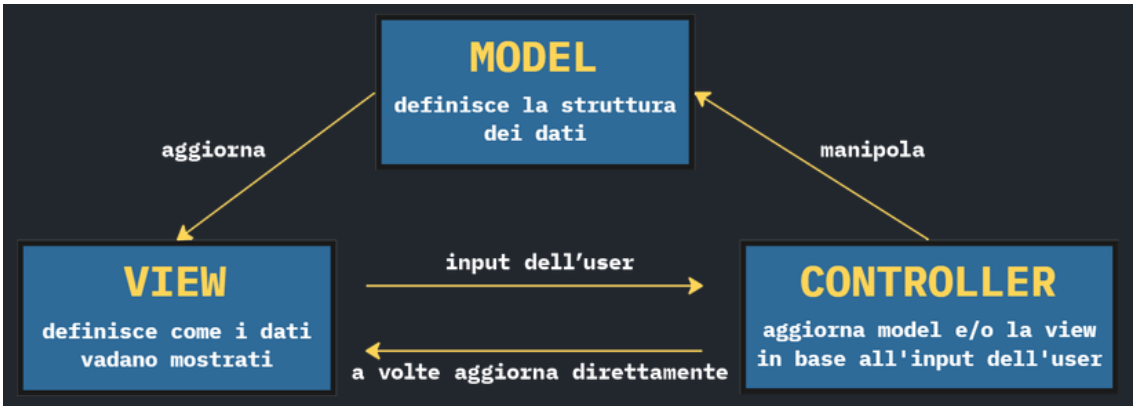

Figura 2.1: Pattern Model-View-Controller (MVC).

Django interpreta il pattern MVC a modo suo, in modo tale che il Controller risulti in realtà il framework in sé, le View non scelgano come i dati vadano mostrati bensì quali dati mostrare e il come mostrarli viene in realtà deciso nei Template. Questa reinterpretazione del modello MVC prende il nome di *Model-Template-View* (**MTV**) [7].

- **Model**: Il *Model layer* rappresenta i dati e lo schema del database dell'applicazione. Definisce la struttura delle tabelle, inclusi campi, relazioni e vincoli. In Django, i modelli sono

definiti come classi Python che estendono la classe *django.db.models.Model*, dove ogni attributo rappresenta un campo del database. Questa caratteristica è particolarmente apprezzata perché consente di gestire il salvataggio dei dati direttamente in Python. Django utilizza un *Object-Relational Mapping* (**ORM**) per mappare le classi Python alle tabelle del database, eliminando la necessità di scrivere manualmente le query SQL. Tutti i modelli sono contenuti nel file `models.py` nella directory dell'applicazione.

- **Template**: Il *Template layer* è responsabile di definire come i dati devono essere presentati all'utente. I template sono file HTML che contengono dei *placeholders*[2] per i contenuti dinamici. Django utilizza un motore di template integrato che permette incorporare codice Python al loro interno e di inserire dati provenienti dal livello View utilizzando tag e filtri specifici. Questo consente di generare file HTML dinamici in modo efficiente e flessibile.
- **View**: Il *View layer* funge da ponte tra il livello Model e il livello Template ed implementa la componente *Controller* del pattern MVC. Le viste sono funzioni o classi Python che gestiscono le richieste HTTP, recuperano i dati dai modelli e li passano al Template per la renderizzazione. Le viste definiscono la logica dietro le pagine web e determinano quali dati vengono visualizzati ed utilizzando quale template. L'utente interagisce con le view, contenute nel `views.py`, tramite URL, che vengono associati direttamente a ciascuna di esse.

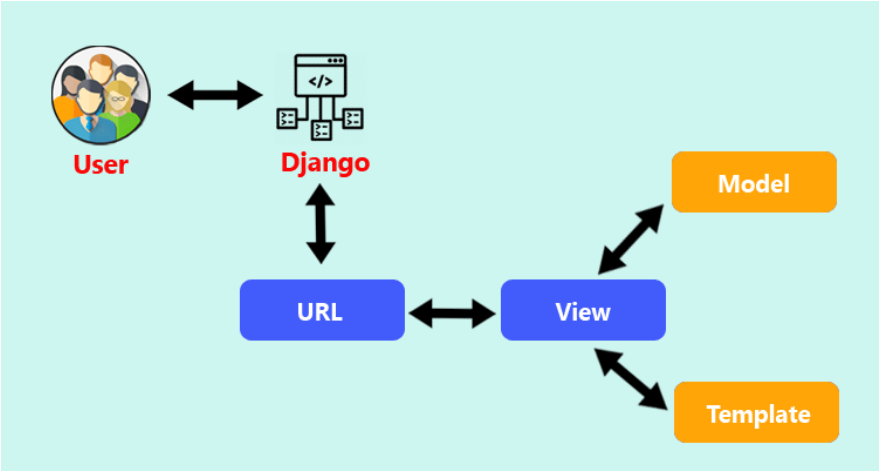

Figura 2.2: Pattern Model-Template-View in Django (MTV).

[2]segnaposto all'interno dei file HTML che vengono sostituiti con contenuti dinamici durante la renderizzazione della pagina.

2.1.4 Struttura di un Progetto Django

Un progetto Django [8] è composto da una serie di file e directory che seguono una struttura ben definita, progettata per facilitare lo sviluppo, la manutenzione e la scalabilità delle applicazioni web. Subito dopo la creazione del progetto viene generata una struttura di directory simile alla seguente:

```
myproject/
    manage.py
    myproject/
        __init__.py
        settings.py
        urls.py
        wsgi.py
        asgi.py
```

Codice 2.1: Struttura di un progetto Django.

- `manage.py`: Script che funge da gateway per vari comandi di gestione di Django. È lo strumento attraverso cui si avvia il server di sviluppo, si creano applicazioni, si eseguono migrazioni e altro ancora;
- `myproject/`: É la directory del progetto principale, che contiene le configurazioni globali del progetto;
  - `__init__.py`: File vuoto che indica a Python che questa directory dovrebbe essere trattata come un pacchetto. É essenziale per organizzare e importare moduli all'interno del progetto;
  - `settings.py`: Contiene tutte le configurazioni del progetto, come le impostazioni del database, le applicazioni installate, le impostazioni di sicurezza e altre configurazioni globali;
  - `urls.py`: Definisce i percorsi URL per il progetto e li mappa alle viste corrispondenti;
  - `wsgi.py`: Abbreviazione di *Web Server Gateway Interface*, wsgi.py funge da punto di ingresso per l'applicazione quando viene distribuita su un server di produzione;

- `asgi.py`: simile a `wsgi.py`, `asgi.py` è il punto di ingresso per i server web asincroni. Sta per *Asynchronous Server Gateway Interface* e facilita la gestione delle richieste HTTP asincrone.

Un progetto Django è ulteriormente suddiviso in diverse applicazioni, ciascuna delle quali rappresenta un modulo funzionale indipendente (Loose Coupling). Ogni applicazione ha una struttura più o meno standard:

```
myapp/
    migrations/
        __init__.py
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    views.py
    templates/
    static/
```

Codice 2.2: Struttura di un'applicazione Django.

In particolare sono presenti i file principali per la configurazione dell'applicazione e per l'implementazione del modello MTV:

- `models.py`: File in cui si definiscono le strutture dati utilizzando l'ORM (Object-Relational Mapping) di Django. Ogni classe modello rappresenta una tabella nel database. Questo file costituisce la base della gestione dei dati dell'applicazione;
- `views.py`: File che racchiude la logica che definisce come l'applicazione interagisce con le richieste degli utenti. Le viste gestiscono l'elaborazione dei dati, la renderizzazione dei template e le risposte alle azioni. Questo file trasforma le interazioni degli utenti in risposte concrete;
- `tests.py`: File in cui vengono inseriti i test per garantire che i componenti dell'applicazione funzionino come previsto;

- `admin.py`: File che configura come i modelli dell'applicazione vengono presentati nell'**interfaccia di amministrazione** di Django. L'interfaccia di amministrazione è un'applicazione web integrata nel framework che consente agli sviluppatori di gestire e amministrare facilmente i dati. Viene generata automaticamente e offre un'interfaccia grafica intuitiva per eseguire operazioni CRUD[3] sui modelli definiti nell'applicazione;
- `migrations`: Questa directory contiene le *migrations*, file che tengono traccia delle modifiche apportate ai modelli dell'applicazione, utilizzati per gestire e applicare le modifiche al database;
- Altri File: Potrebbero emergere file aggiuntivi in base alle esigenze dell'applicazione. Ad esempio, `forms.py` contiene le classi dei form per l'inserimento dei dati, `urls.py` mappa gli URL alle viste, e `apps.py` gestisce le configurazioni specifiche dell'applicazione;
- `templates`: É la directory che contiene i file HTML che definiscono la struttura e il layout delle pagine web;
- `static`: É la directory che contiene i file statici necessari per il funzionamento del sito web, come file CSS, JavaScript, immagini e altri asset.

2.1.5 Approfondimento sulla Gestione del Database

Come già riportato, Django permette di gestire il database in modo molto semplice attraverso il meccanismo ORM (Object-Relational Mapping), che consente di definire i modelli di dati come classi Python, successivamente tradotte in tabelle del database. Ciò avviene attraverso il meccanismo di migrazioni. Quando si apportano modifiche ai modelli, come l'aggiunta di un nuovo campo o la modifica di uno esistente, Django utilizza il comando `makemigrations` per generare il **file di migrazione** che descrive queste modifiche. Successivamente, il comando `migrate` applica queste modifiche traducendole in linguaggio SQL, aggiornando così la struttura del database in modo incrementale e sicuro. Django supporta attualmente cinque database: PostgreSQL, MariaDB, MySQL, Oracle e lo standard SQLite.

[3] Operazioni di tipo Create, Read, Update, Delete

2.2 Bootstrap

Bootstrap [9] è un framework front-end open source, originariamente sviluppato da Mark Otto e Jacob Thornton presso Twitter, che facilita la creazione di siti web reattivi e mobile-first. Rilasciato per la prima volta nel 2011, Bootstrap ha guadagnato rapidamente popolarità grazie alla sua facilità d'uso, flessibilità e vasta documentazione. Questo framework è stato progettato per aiutare gli sviluppatori a costruire interfacce utente consistenti e di alta qualità utilizzando componenti predefiniti e uno stile CSS comune.

2.2.1 Struttura del Framework

Bootstrap è composto da un insieme di strumenti CSS, JavaScript e HTML che permettono di costruire pagine web in modo rapido ed efficiente. La struttura del framework, in particolare, si basa su quattro componenti principali:

- **Grid System**: Il sistema a griglia di Bootstrap è uno degli elementi più distintivi del framework. Utilizza una struttura a 12 colonne flessibile che consente di creare layout reattivi che si adattano automaticamente alle dimensioni dello schermo. Questo sistema è altamente personalizzabile e può essere manipolato tramite **classi CSS** per gestire la visibilità e il posizionamento dei contenuti;
- **Componenti CSS**: Bootstrap include una vasta gamma di componenti CSS predefiniti, come pulsanti, modali, barre di navigazione, badge, e molto altro. Questi componenti sono progettati per essere facilmente integrabili e personalizzabili;
- **Componenti JavaScript**: Oltre ai componenti CSS, Bootstrap fornisce una serie di plugin JavaScript che aggiungono funzionalità interattive alle pagine web. Questi plugin includono elementi come *carousels*[4], finestre di dialogo modali, tooltips[5] e finestre di avviso. L'integrazione dei plugin è semplice e può essere gestita tramite attributi HTML o chiamate JavaScript dirette;

[4]Componente di presentazione per scorrere gli elementi

[5]Finestra pop-up che appare quando l'utente passa il cursore su un elemento dell'interfaccia, fornendo informazioni aggiuntive o descrizioni sull'elemento stesso.

- **Utilities**: Bootstrap offre anche una serie di classi di utilità che consentono di applicare rapidamente stili CSS comuni senza dover scrivere codice personalizzato. Queste utilities includono classi per il margine, il padding, la tipografia, i colori, e altro ancora.

2.2.2 Vantaggi di Bootstrap

L'utilizzo di Bootstrap presenta diversi vantaggi significativi per gli sviluppatori web:

- **Reattività e Mobile-First**: Bootstrap è stato progettato con un approccio mobile-first, il che significa che le pagine web costruite con questo framework sono ottimizzate per dispositivi mobili e si adattano automaticamente a schermi di diverse dimensioni;
- **Consistenza**: Utilizzando componenti e stili predefiniti, Bootstrap garantisce un aspetto coerente e professionale per tutte le pagine di un sito web, riducendo la necessità di personalizzazioni estensive;
- **Velocità di Sviluppo**: Grazie alla disponibilità di numerosi componenti pronti all'uso, Bootstrap accelera il processo di sviluppo, permettendo agli sviluppatori di concentrarsi solo su funzionalità specifiche e personalizzazioni necessarie;
- **Comunità e Supporto**: Essendo uno dei framework più popolari al mondo, vanta una vasta comunità di sviluppatori che contribuiscono al suo miglioramento continuo e forniscono supporto attraverso forum, tutorial e documentazione dettagliata.

3. Analisi Iniziale dell'Applicazione Missioni

Nel seguente capitolo verrà descritta nel dettaglio l'applicazione Missioni [10] così com'era prima dell'implementazione delle modifiche. Verranno analizzate le caratteristiche principali e messi in luce i punti critici sui quali sono intervenuto per migliorare l'esperienza utente. Questo permetterà di comprendere le basi da cui siamo partiti e il motivo delle successive trasformazioni.

3.1 Struttura del Progetto

Il progetto è organizzato all'interno di una directory radice denominata `missioni`. All'interno di essa si trova, oltre ai file `manage.py` e `requirements.txt`, la directory `Rimborsi`, che contiene i file di configurazione del progetto come `settings.py`, `urls.py` e altri (vedi 2.1.4). Nella directory principale sono presenti due applicazioni: la prima è `shibboleth`, necessaria per l'autenticazione degli utenti tramite credenziali universitarie, e la seconda è `RimborsiApp`. Quest'ultima è l'applicazione responsabile della compilazione dei documenti e per cui quella di cui ci occuperemo da questo momento in poi. Infine, vi è una directory `template`, che contiene tutti i template per le varie applicazioni.

```
1  missioni/
2      Rimborsi/
3          __init__.py
4          settings.py
5          urls.py
6          wsgi.py
7      RimborsiApp/
8          migrations/
9          static/
10         templatetags/
11         __init__.py
12         admin.py
13         apps.py
14         compila_pdf.py
```

```
15          forms.py
16          models.py
17          storage.py
18          tests.py
19          urls.py
20          utils.py
21          views.py
22          widgets.py
23      shibboleth/
24      templates/
25          registration/
26          Rimborsi/
```

Codice 3.1: Struttura del progetto.

3.2 Funzionalità Principali

Missioni è un applicativo web progettato per semplificare la compilazione dei documenti necessari per la richiesta di rimborso delle missioni universitarie. In generale, consente all'utente di creare una missione, inserire tutte le spese ad essa correlate e successivamente compilare i documenti necessari per il rimborso. Più nello specifico, le funzionalità principali includono:

- **Creazione profilo utente:** Permette agli utenti di creare un profilo univoco con le proprie informazioni e di registrare una serie di automobili, che potranno essere segnate come mezzi di spostamento durante le missioni;
- **Creazione di una missione:** Consente l'inserimento dei dettagli relativi a una nuova missione universitaria, che viene aggiunta ad una ***lista missioni***. Quest'ultima contiene tutte le missioni attive e concluse;
- **Modifica delle missioni:** Offre la possibilità di modificare le informazioni delle missioni esistenti aggiornando gli elementi già salvati o inserendo nuove spese;
- **Eliminazione missioni:** Permette di rimuovere missioni non più necessarie;

- **Rimborso missioni:** Permette di concludere una missione specificando che essa è già stata rimborsata;
- **Resoconto missione:** Genera un resoconto della missione selezionata;
- **Compilazione documenti:** Selezionando l'opzione di compilazione dei documenti dal resoconto missione, l'applicazione automatizza l'inserimento dei dati relativi alle spese totali e alle informazioni dell'utente nei documenti di rimborso.

3.3 Struttura del Database

Come già riportato, Django si basa sul paradigma MTV (Model-Template-View). Iniziamo quindi analizzando i modelli che sono inseriti nel `models.py`. Il database utilizzato per rappresentare i modelli è MySQL, scelto per la sua capacità di gestire in modo efficiente le missioni, i profili utente e le informazioni correlate. La struttura del database è composta da diverse tabelle principali, ciascuna delle quali è collegata tramite chiavi esterne, garantendo l'integrità e la coerenza dei dati.

Come visibile nell'immagine 3.1, la tabella centrale è **Missione**, che memorizza i dettagli delle missioni assegnate agli utenti. Questa tabella include campi fondamentali come `user` (chiave esterna che collega la missione all'utente assegnato), `citta_destinazione`, `stato_destinazione` (collegato alla tabella **Stato**), `inizio`, `fine`, `tipo`, e `anticipo`; oltre a campi testuali per la descrizione delle spese e i mezzi di trasporto utilizzati. La relazione tra missioni e automobili è gestita tramite una chiave esterna verso la tabella **Automobile**, che a sua volta memorizza i dettagli dei veicoli utilizzati nelle missioni.

Il modello **Profile** estende la gestione degli utenti, aggiungendo dettagli personali e professionali come `data`, `luogo di nascita`, `residenza`, `domicilio`, `qualifica`, e altre informazioni specifiche per i dottorandi. Questa tabella è collegata alla tabella **User** di Django tramite una relazione uno-a-uno, permettendo di estendere il profilo standard degli utenti.

Per la gestione delle categorie e degli stati delle missioni, il database include le tabelle **Categoria** e **Stato**. La tabella **Stato** è collegata a **Categoria** attraverso una chiave esterna, definendo così il tipo di missione in base alla categoria (es. docenti o tecnici).

La tabella **Trasporto** è utilizzata per registrare i dettagli dei mezzi di trasporto impiegati nelle missioni, con campi come `data`, `da`, `a`, `mezzo`, `costo`, e `valuta`. Questa tabella è collegata alla tabella **Missione** attraverso una chiave esterna, permettendo di associare diversi spostamenti a una singola missione.

Il modello **ModuliMissione** gestisce i file e i moduli, come le richieste di anticipo e i documenti necessari per il completamento della missione. Ogni modulo è associato a una specifica missione tramite una relazione uno-a-uno, consentendo di caricare e gestire in modo organizzato i file richiesti.

Infine, il database include anche tabelle per gestire gli indirizzi (**Indirizzo**) e per archiviare informazioni di località rilevanti, come il comune e la provincia, necessarie per la gestione del profilo utente e delle destinazioni delle missioni.

3.3.1 Gestione delle Spese

Un aspetto sul quale vale la pena concentrarsi riguarda il trattamento delle spese associate ad ogni missione. Esse rappresentano le spese che dovranno essere rimborsate dall'università e includono pasti (registrati come scontrino), pernottamenti, convegni, altre spese e trasporti. Tra queste, solo i trasporti sono gestiti tramite una tabella dedicata. Le altre spese sono memorizzate come voci nella tabella delle missioni, utilizzando un campo `TextField`. In particolare, ogni `TextField` viene utilizzato per salvare i dati in formato JSON. Questo approccio permette di strutturare ogni voce di spesa in maniera flessibile e dettagliata, includendo vari campi specifici come la descrizione, l'importo, la data, e i relativi valori.

Sotto è riportato lo schema dettagliato che rappresenta graficamente i modelli principali utilizzati dall'applicazione, con particolare attenzione al modello "Missione", che svolge un ruolo centrale nella gestione delle informazioni e delle operazioni correlate alle missioni.

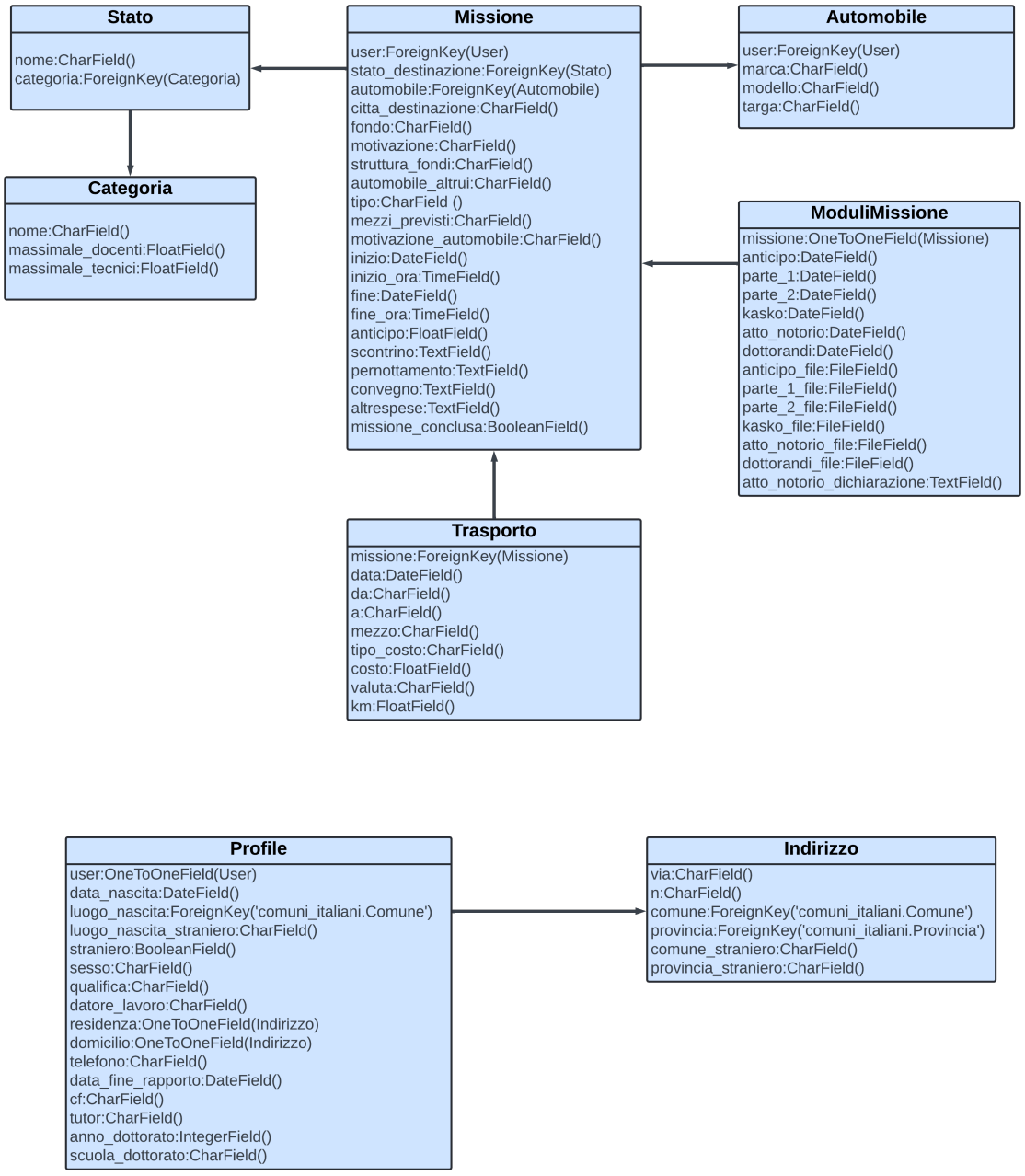

Figura 3.1: Diagramma delle tabelle del database prima delle modifiche.

3.4 Template e View

Dopo aver esaminato la logica di **Model** e la struttura del database, passiamo a descrivere come sono implementate le logiche di **Template** e **View** all'interno dell'applicazione.

3.4.1 Analisi del Livello Template

La logica dei template è organizzata attraverso diversi file HTML che gestiscono la presentazione dei dati interfacciandosi con il database tramite le view. Un elemento chiave in questa struttura sono i form, definiti nel file `forms.py`. Questi, fondamentali per l'interazione con l'utente, vengono integrati e renderizzati all'interno dei template, fungendo da interfaccia per la raccolta e la validazione dei dati. Nello specifico, in Missioni i form più rilevanti sono **MissioneForm**, **TrasportiForm**, **ProfileForm** e **ForeignProfileForm**. Essi permettono di determinare il modo in cui vengono inseriti i dati relativi ai diversi elementi dell'applicazione, come le informazioni del profilo, i dettagli delle missioni e le spese relative ai trasporti. In aggiunta, grazie al pacchetto **crispy-forms**, i form sono personalizzati per offrire un'esperienza utente coerente e funzionale oltre a poter essere configurati con specifici widget e layout.

3.4.2 Analisi del Livello View

La logica delle view è implementata nel file `views.py`, dove sono definite le funzioni che gestiscono le richieste HTTP e orchestrano la logica applicativa. Queste view recuperano i dati dal database, li elaborano secondo le necessità dell'applicazione e passano le informazioni ai template per il rendering finale. Ad esempio, la view **crea_missione** gestisce la logica per la creazione di una nuova missione, inizializzando un form `MissioneForm`, validando i dati inviati dall'utente e infine salvando una nuova istanza di `Missione` nel database. Un'altra view importante, **lista_missioni**, filtra e organizza le missioni dell'utente, distinguendo tra missioni attive e concluse, per una facile consultazione. La view **profile**, invece, gestisce la visualizzazione e l'aggiornamento dei profili utente, distinguendo tra profili italiani e stranieri. Inoltre, la view **resoconto** è responsabile della generazione di un riepilogo dettagliato delle spese e dei trasporti relativi a una missione, calcolando anche indennità e totali convertiti.

Le view sono anche responsabili della gestione delle operazioni CRUD (Create, Read, Update, Delete) per le entità principali dell'applicazione, come missioni, profili e spese. Tra queste rientrano **salva_trasporti**, **salva_pasti**, **salva_pernottamenti**, **salva_convegni**, **clona_missione** e **crea_missione**. Queste view integrano la logica backend con la presentazione frontend gestita dai template, garantendo una coerenza tra l'elaborazione dei dati e la loro visualizzazione.

3.5 Valutazione delle Criticità Esistenti

L'applicazione "Missioni" soddisfa le esigenze di base per la gestione delle missioni universitarie, ma presenta alcune criticità che ne compromettono l'efficacia complessiva. Una delle principali problematiche è la visualizzazione su dispositivi mobili. Sebbene Bootstrap offra un certo grado di scalabilità, molti degli elementi e delle strutture utilizzati nell'applicazione non si adattano adeguatamente a schermi di piccole dimensioni. Questo problema rende l'esperienza utente da mobile poco agevole, soprattutto nelle pagine più cruciali come quelle dedicate alla creazione e modifica del profilo, alla gestione della lista delle missioni, alla creazione di nuove missioni e al resoconto delle spese.

La seconda grande limitazione su cui abbiamo deciso di intervenire è l'assenza della possibilità di inserire immagini nell'applicazione. In particolare, sarebbe utile prevedere campi specifici per l'inserimento degli scontrini delle spese, così da poter certificare le diverse spese associandole al relativo documento fiscale.

4. Implementazione Funzionalità Aggiuntive

In questo capitolo verranno descritte in dettaglio le modifiche apportate all'applicazione, con l'obiettivo di migliorare l'esperienza utente e aggiungere nuove funzionalità. Questi cambiamenti sono stati realizzati in risposta alle problematiche evidenziate precedentemente e mirano a ottimizzare l'utilizzo dell'applicazione in contesti diversi.

Il capitolo è strutturato in tre sezioni principali. Nella prima verrà spiegato come l'applicazione è stata adattata per essere compatibile con dispositivi mobili, rendendo le varie pagine adatte a schermi di diverse dimensioni. Saranno descritte le modifiche alle classi e alle funzioni, evidenziando le ragioni di tali interventi e i benefici che ne derivano. Nella seconda parte, verrà illustrata l'aggiunta della funzionalità che consente di inserire immagini associate alle spese delle missioni, con particolare attenzione alle modifiche apportate al database. Infine, l'ultima sezione analizzerà gli script di migrazione, fondamentali per trasportare i dati dal vecchio al nuovo formato.

4.1 Responsive Design e Adattamento per Mobile

Lo strumento fondamentale utilizzato per rendere il design adattabile alle variazioni di dimensione dello schermo è il framework Bootstrap. Le modifiche iniziali si sono quindi concentrate sul front-end dell'applicazione, integrando classi e componenti specifiche di Bootstrap nelle parti di codice relative alla visualizzazione dell'interfaccia.

4.1.1 Layout Responsivi per i Form di Inserimento Dati del Profilo

Il primo aspetto trattato riguarda i form per l'inserimento delle informazioni relative al profilo utente e alle missioni. I layout sono specificati nel `forms.py` utilizzando il modulo **django-crispy-forms**. Prendendo in esame la classe `ProfileForm`, la disposizione è organizzata in righe e colonne di dimensioni definite dalle classi Bootstrap, per garantire una presentazione chiara e responsiva dei campi. Ad esempio, l'uso della classe `col-6` di Bootstrap fa sì che le colonne occupino 6 delle 12 unità disponibili, distribuendo così gli elementi in maniera equilibrata all'interno della riga.

Nella versione originale, i campi si adattavano alle dimensioni dello schermo, ma mantenevano sempre la loro dimensione proporzionata, così come visibile nell'immagine 4.1.

Figura 4.1: Campi di inserimento del profilo prima delle modifiche.

La soluzione è stata semplice grazie a una funzionalità di Bootstrap che consente di assegnare diverse classi a uno stesso elemento, selezionando automaticamente la più appropriata in base alla dimensione dello schermo. In particolare, è possibile utilizzare delle *options* che agiscono come breakpoint, applicando una classe specifica solo quando lo schermo raggiunge o supera la dimensione del breakpoint stesso.

Nel codice sottostante ho utilizzato le opzioni [11] `col-sm-*`, `col-md-*`, e `col-lg-*`, che entrano in utilizzo rispettivamente per schermi di piccole, medie e grandi dimensioni. Queste consentono di definire un layout responsivo che si adatta automaticamente alle diverse larghezze degli schermi.

```
1 Row(Column('nome', css_class='col-sm-6'),
2     Column('cognome', css_class='col-sm-6')),
3 Row(Column('data_nascita', css_class='col-lg-3 col-sm-6'),
4     Column('luogo_nascita_straniero', css_class='col-lg-3 col-sm-6'),
5     Column('cf', css_class='col-lg-3 col-sm-6'),
```

Codice 4.1: Applicazione delle class-options Bootstrap al layout del profilo.

Questa metodologia, applicata a tutti i campi del profilo, ha permesso alla fine di ottenere un risultato come quello in figura 4.2 e 4.3. Prendendo ad esempio il campo `nome`: su schermi di dimensioni medio-grandi, questo campo occuperà 6 delle 12 colonne disponibili nella griglia, mantenendo così una disposizione bilanciata. Su schermi più piccoli, invece, il campo si espanderà per occupare tutte e 12 le colonne, garantendo che l'interfaccia rimanga leggibile e ben organizzata anche su dispositivi mobili.

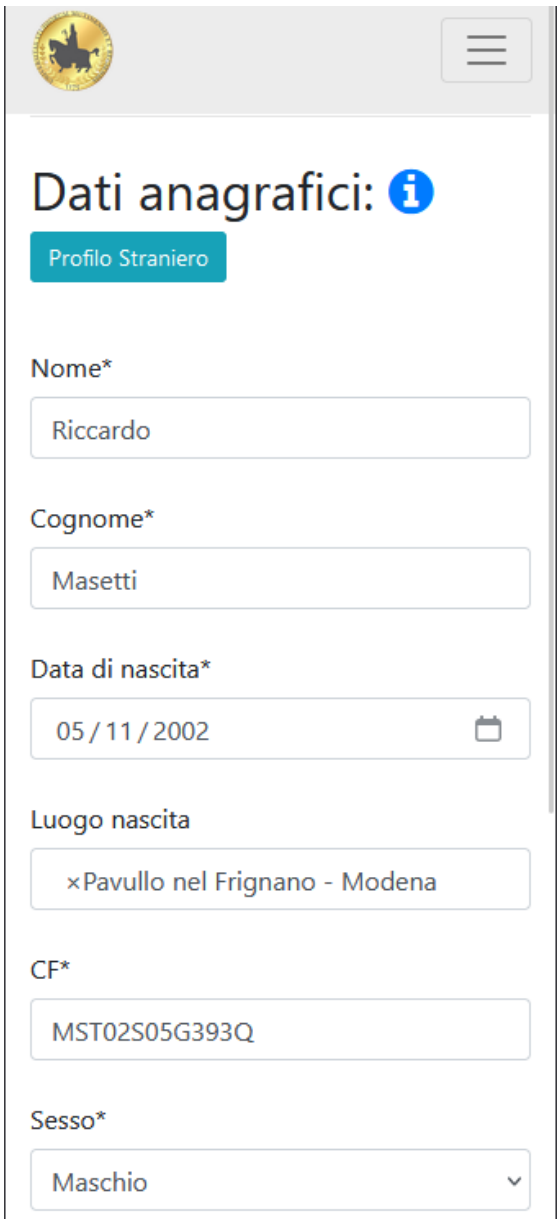

Figura 4.2: Campi di inserimento dei dati anagrafici dopo le modifiche.

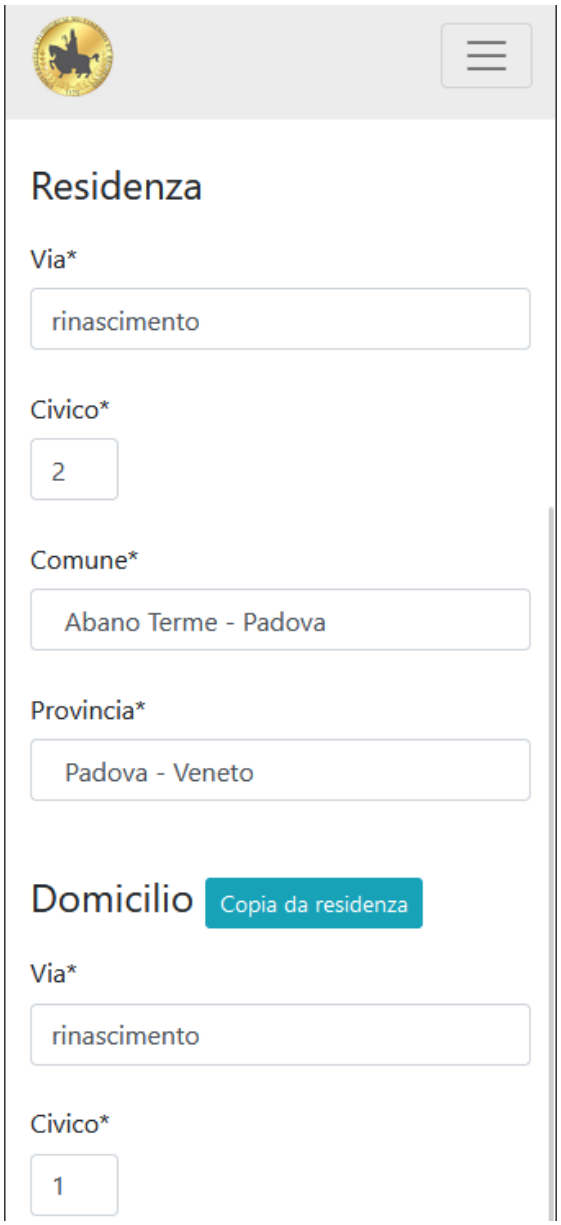

Figura 4.3: Campi di inserimento dei dati di residenza dopo le modifiche.

4.1.2 Layout Responsivo per il Form Missione

Sebbene per il profilo, così come per la versione straniera `ForeignProfile`, le modifiche siano state minime grazie all'uso di Bootstrap, la pagina di inserimento delle informazioni e delle spese relative alla missione ha richiesto un intervento più complesso. In particolare, ogni missione è composta da un form iniziale, definito nel `forms.py` utilizzando il modulo **django-crispy-forms** [12], seguito da una serie di formset dinamici che gestiscono l'inserimento dei dettagli specifici, come pasti, pernottamenti, trasporti, iscrizioni a convegni e altre spese.

Per quel che riguarda il primo aspetto, la modifica è stata pressochè identica al campo profilo, aggiungendo al layout le classi Bootstrap per personalizzarlo sulla base della dimensione schermo.

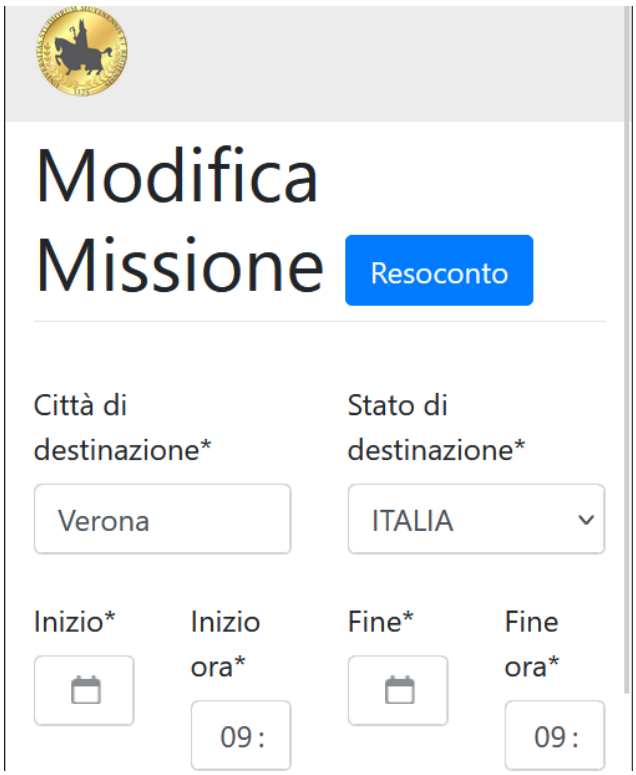

Figura 4.4: Campi di inserimento dei dati missione prima delle modifiche.

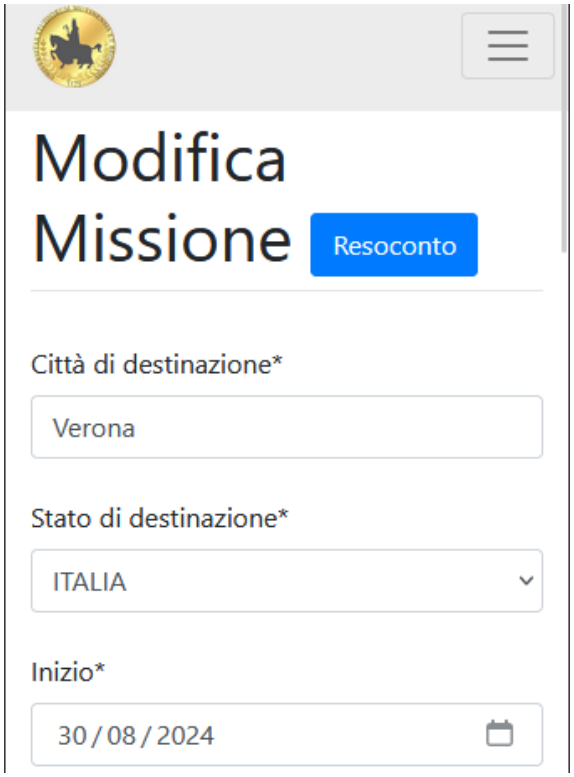

Figura 4.5: Campi di inserimento dei dati missione dopo delle modifiche.

Tuttavia, la parte più significativa della modifica ha riguardato la gestione dei form delle spese all'interno della missione. Nella versione originale, i formset relativi alle spese erano incapsulati nel file html `missione.html` all'interno di tabelle (**table**) con l'aggiunta di classi di opzione Boo-

tstrap. Questo approccio, sebbene funzionale per una presentazione ordinata dei dati, presentava limitazioni significative in termini di responsività, specialmente su schermi di piccole dimensioni. Prendendo in considerazione le spese relative ai trasporti, il problema è ben visibile in figura 4.6, dove la table esce dai bordi dello schermo, risultando di difficile lettura e poco pratica da utilizzare.

Figura 4.6: Formset di inserimento trasporti prima delle modifiche.

Per risolvere questo problema, ho apportato modifiche direttamente al file `missione.html`, sostituendo la struttura basata su **table** con l'uso di contenitori di classe **card**[1] di Bootstrap [13]. In questo modo, ogni formset relativo alle spese, come quello per i trasporti preso ad esempio, è stato riorganizzato in un layout che si adatta meglio alle diverse dimensioni dello schermo. Le cards offrono infatti una maggiore flessibilità nella gestione dello spazio e consentono di presentare i dati in modo più compatto, senza i problemi di overflow tipici delle tabelle su schermi piccoli.

Con riferimento alla gestione delle spese di trasporto, il codice modificato riportato di seguito è stato applicato in modo simile anche a tutte le altre tipologie di spesa, con l'unica differenza nei formset specifici per ciascuna categoria e nella view collegata al template.

```
1  <form action="{% url `RimborsiApp:salva_trasporti'  id=missione.id %}"
       method="post" enctype="multipart/form-data">{% csrf_token %}
2     {{ trasporti_formset.management_form }}
3         <div id="trasporti-formset-container">
4             {% for form in trasporti_formset.forms %}
```

[1]Una "card" è un contenitore flessibile per vari contenuti (testi, immagini, link) organizzati in blocchi separati. Offre un layout standardizzato, personalizzabile e responsivo, adattandosi automaticamente alle dimensioni dello schermo.

```
5                 <div class="card-lg-inline mb-5 mb-md-3
                      trasporti_formset_row">
6                     <div class="card-body d-lg-flex">
7                         {% for field in form.visible_fields %}
8                             <div class="form-group mx-1 flex-grow-1">
9                                 {% if forloop.first %}
10                                    {% for hidden in form.hidden_fields
                                          %}
11                                    {{ hidden }}
12                                {% endfor %}
13                            {% endif %}
14                            <label for="{{ field.id_for_label }}">
                                  {{ field.label|capfirst }}</label>
15                            {{ field.errors.as_ul }}
16                            {{ field }}</div>
17                        {% endfor %}</div>
18                    </div>
19            {% endfor %}
20        </div>
21    </form>
```

Codice 4.2: Template della spesa `trasporti` aggiornato utilizzando le cards di Bootstrap.

Grazie alle modifiche apportate, il layout ora si adatta perfettamente a diverse dimensioni dello schermo con i campi che si riposizionano verticalmente quando lo spazio orizzontale è limitato.

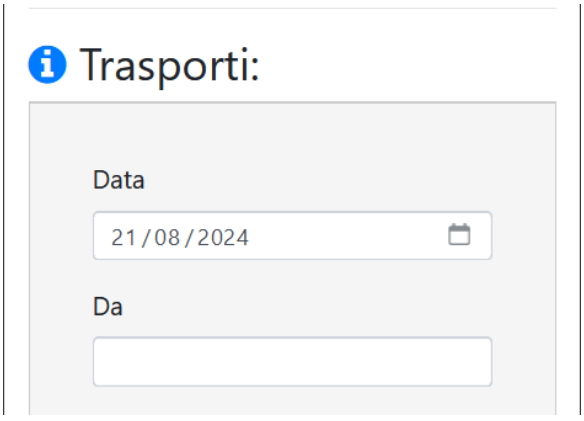

Figura 4.7: Formset trasporti post-modifiche.

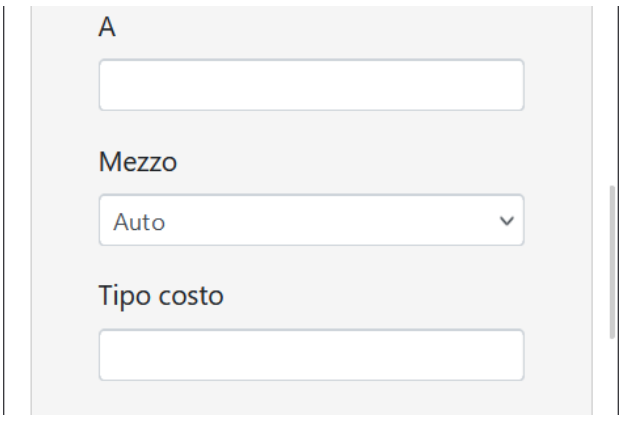

Figura 4.8: Formset trasporti post-modifiche.

4.1.3 Interfaccia Adattiva per la Lista delle Missioni

La struttura a tabelle era ampiamente utilizzata nell'applicazione originale, non solo per le missioni, ma anche in altre sezioni. In particolare, la pagina che elenca tutte le missioni, così come il resoconto che verrà analizzato nel prossimo paragrafo, soffriva di problemi di adattamento alle diverse dimensioni dello schermo a causa dell'uso estensivo di **table**.

Perciò, nel file `lista_missioni.html`, ho rivisto la struttura del layout sia per le missioni attive che per quelle concluse. Per le prime, ho mantenuto lo stesso formato per gli elementi chiave, come il titolo e i pulsanti "Resoconto", "Modifica", "Clona", "Rimborsata", "Cancella", ma li ho riorganizzati all'interno di contenitori con classe **card**. Questo approccio ha migliorato la responsività, permettendo ai pulsanti di riposizionarsi in maniera funzionale quando la dimensione della pagina diminuisce, come visibile in figura 4.10.

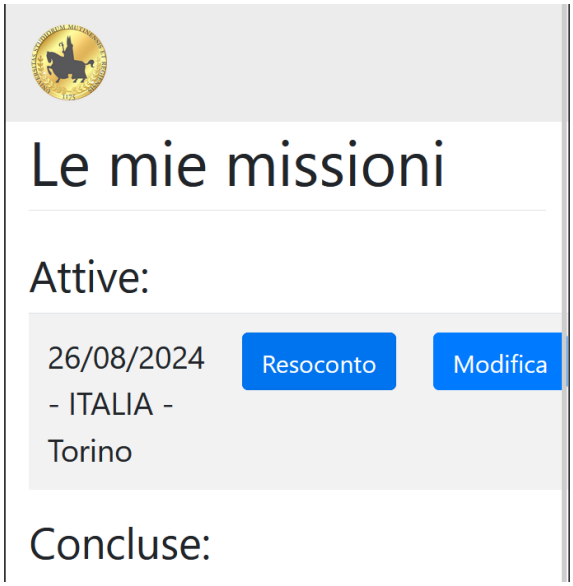

Figura 4.9: Interfaccia della lista missioni attive prima delle modifiche al layout.

Figura 4.10: Interfaccia della lista missioni attive dopo le modifiche al layout.

Per le missioni concluse, la modifica è molto simile (anche dal punto di vista del codice), ma con una differenza: solo i pulsanti "Resoconto", "Clona" e "Cancella" sono stati mantenuti all'interno delle cards, poiché gli altri non erano necessari per questa sezione.

Per via delle somiglianze, di seguito è riportato solo il codice delle missioni attive:

```
<h3>Attive:</h3>
{% if missioni_attive %}
    <div class="container-fluid px-0">
        {% for m in missioni_attive %}
            <div class="card mb-3 mx-0" style="background-color:
                whitesmoke; width: 100%;">
                <div class="card-body d-flex flex-column flex-md-row
                    flex-wrap align-items-center align-items-md-center
                    justify-content-between">
                    <h5 class="card-title mb-2 mb-md-0">
                        {{m.inizio|date:"d/m/Y"}} - {{
                            m.stato_destinazione}} - {{
                            m.citta_destinazione}}</h5>
                    <div class="btn-group flex-wrap
                        justify-content-center align-items-center">
                        <a class="btn btn-primary btn-md mb-md-0 mb-2
                            mx-1 rounded"
                            href="{% url 'RimborsiApp:resoconto' id=m.id
                                %}"
                            role="button">Resoconto</a>
                        <a class="btn btn-primary btn-md mb-md-0 mb-2
                            mx-1 rounded"
                            href="{% url 'RimborsiApp:missione' id=m.id
                                %}"
                            role="button">Modifica</a>
                        <a class="btn btn-primary btn-md mb-md-0 mb-2
                            mx-1 rounded"
                            href="{% url 'RimborsiApp:clona_missione'
                                id=m.id %}"
                            role="button">Clona</a>
                        <a class="btn btn-success btn-md mb-md-0 mb-2
                            mx-1 rounded"
                            href="#" data-href="{% url '
                                RimborsiApp:concludi_missione' id=m.id %}
                                "
```

```
21                              data-target="#confirm-concludi"
22                              data-toggle="modal"
23                              role="button">Rimborsata</a>
24                          <a class="btn btn-danger btn-md mb-md-0 mb-2
                                mx-1 rounded"
25                              href="#" data-href="{% url '
                                    RimborsiApp:cancella_missione' id=m.id%}"
26                              data-target="#confirm-delete"
27                              data-toggle="modal"
28                              role="button">Cancella</a>
29                      </div>
30                  </div>
31              </div>
32          {% endfor %}
33      </div>
34  {% else %}
35      <p>Non ci sono missioni attive</p>
36  {% endif %}
```

Codice 4.3: Implementazione della lista missioni tramite utilizzo delle cards di Bootstrap.

4.1.4 Interfaccia Adattiva per il Resoconto Missione

Analogamente alle modifiche apportate alla lista delle missioni, anche la pagina di resoconto missione è stata riorganizzata per migliorare la compatibilità con dispositivi mobile.

Ancora una volta, ho ridisegnato il layout sostituendo le tabelle (utilizzate per organizzare i dati dei moduli di rimborso) con dei contenitori di classe **card**, riorganizzando le informazioni all'interno di questi container per garantire una migliore adattabilità. I dati sono suddivisi in moduli specifici, ciascuno rappresentato da una card separata. Ognuna contiene un form corrispondente a un modulo di missione, quali:

- **Richiesta Anticipo**: per la gestione delle richieste di anticipo.
- **Missione Parte I**: per la richiesta di autorizzazione, con dati personali, destinazione, motivazione e dettagli del trasporto.

- **Missione Parte II**: per la richiesta di rimborso, con dettaglio delle spese suddivise per categoria.
- **Kasko**: per la gestione della copertura assicurativa.
- **Autorizzazione Dottorando**: presente solo se l'utente è un dottorando, per l'autorizzazione alle missioni.
- **Atto Notorio**: per la gestione degli atti notarili e dichiarazioni correlate.

All'interno di ciascuna card, ho inserito i campi dei form dei vari moduli di rimborso, utilizzando **django-crispy-forms** per garantire una gestione ordinata e stilizzata. Oltre ad essi, ogni card contiene anche un link per scaricare il file associato, se disponibile.

Le cards sono state configurate per adattarsi a diverse dimensioni di schermo grazie all'utilizzo delle classi di Bootstrap come `col-md-6` e `col-lg-3`, che garantiscono si dispongano su più colonne su schermi più grandi e si allineino verticalmente su dispositivi piccoli.

Di seguito è riportato un estratto del codice aggiornato. Ho incluso solo la parte relativa alla richiesta di anticipo, poiché le altre sezioni seguono una struttura identica:

```
1  <h3>Moduli precompilati</h3>
2  <form action="{% url 'RimborsiApp:genera_pdf' id=missione.id %}"
       method="post"> {% csrf_token %}
3      <div class="card m-0 p-0">
4          <div class="card-body p-0">
5              {{ moduli_missione_form.management_form }}
6                  <div class="row no-gutters">
7                      <div class="col-md-6 col-lg-3 d-flex">
8                          <div class="table-danger p-4 w-100 d-flex
                               flex-column">
9                              {{ moduli_missione_form.anticipo|
                                   as_crispy_field }}
10                             {% if missione.modulimissione.anticipo_file
                                   %}
11                             <a class="download" href="{% url '
                                   RimborsiApp:download' missione.id '
                                   anticipo_file' %}">
```

```
12                                      <i class="fa fa-file-word-o fa-2x">
                                            </i> Richiesta Anticipo
13                                  </a>
14                              {% else %}
15                                  <p>Richiesta Anticipo</p>
16                              {% endif %}
17                          </div>
18                      </div>
```

Codice 4.4: Implementazione resoconto missione tramite l'utilizzo delle cards di Bootstrap.

Le figure 4.11 e 4.12 mostrano rispettivamente l'interfaccia dei moduli precompilati prima e dopo le modifiche, evidenziando i miglioramenti ottenuti in termini di leggibilità e usabilità.

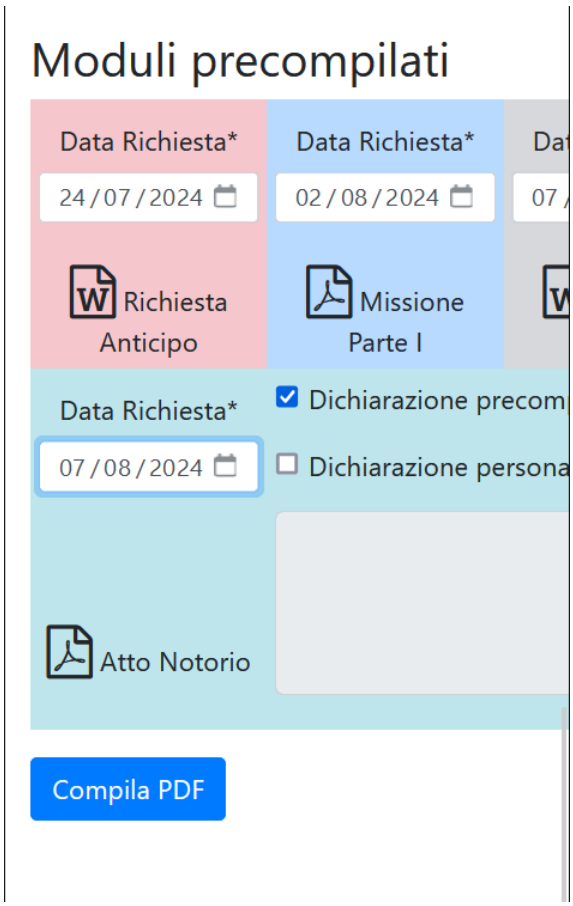

Figura 4.11: Interfaccia moduli precompilati prima delle modifiche al layout.

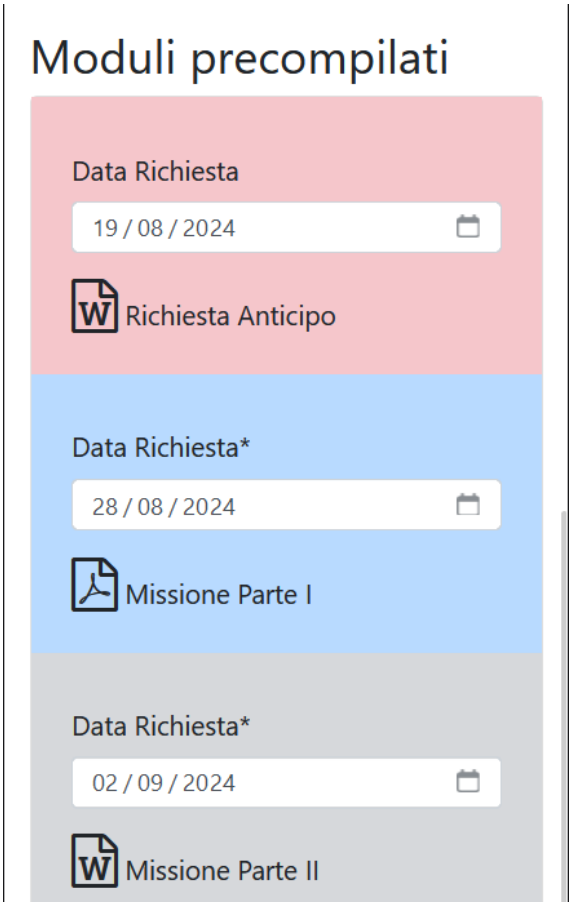

Figura 4.12: Interfaccia moduli precompilati dopo le modifiche al layout.

4.2 Servizio di Upload Immagini per le Spese di Missione

Dopo aver esaminato le modifiche al frontend, si passa ora all'analisi delle modifiche apportate lato backend. In particolare, il paragrafo corrente illustrerà gli interventi apportati sui modelli e sulle viste, volti a implementare la possibilità di associare a ogni spesa l'immagine dello scontrino che la certifica.

Le spese sono suddivise in categorie: pasti, trasporti, pernottamenti, convegni e altre spese. Sebbene alcune di esse condividano campi simili, altre presentano caratteristiche uniche, richiedendo quindi un trattamento differenziato.

4.2.1 Aggiornamenti al Database

Nella versione originale del progetto, solo la spesa relativa ai trasporti era gestita con una tabella dedicata nel database, mentre tutte le altre spese erano memorizzate come campi di tipo `TextField` all'interno del modello `Missione`. Ogni missione includeva un campo per i pasti (`scontrino`), uno per i pernottamenti, uno per i convegni e uno per altre spese. Questi campi contenevano stringhe in formato JSON, che racchiudevano tutte le informazioni sulle spese corrispondenti. Tuttavia, questo approccio non consentiva l'inserimento di immagini, poiché la loro codifica in formato testuale avrebbe reso i campi troppo pesanti. Di conseguenza, è stato necessario creare nuovi modelli per gestire in modo più efficiente tutte le categorie di spesa.

Modello Pasti

Per iniziare, ho sviluppato un modello dedicato per i pasti, ispirandomi alla struttura del modello `trasporti` già esistente. Questo modello è stato collegato al modello Missione tramite una chiave esterna, permettendo di associare in modo chiaro ogni pasto a una specifica missione. Inoltre, l'opzione `on_delete=models.CASCADE` sulla chiave esterna garantisce che i pasti vengano automaticamente rimossi nel caso in cui la missione corrispondente venga eliminata, mantenendo così l'integrità del database. La tabella relativa ai pasti include un campo per la data e tre set di campi[2]

[2]Ogni istanza di pasto è pensata per rappresentare una singola giornata, con campi specifici per colazione, pranzo e cena

per importo, valuta[3], descrizione e immagine.

```
class Pasti(models.Model):
    missione = models.ForeignKey(Missione, on_delete=models.CASCADE)
    data = models.DateField()
    importo1 = models.FloatField(null=True, blank=True)
    valuta1 = models.CharField(max_length=3, choices=VALUTA_CHOICES,
        default='EUR')
    descrizione1 = models.CharField(max_length=255, null=True, blank=True)
    img_scontrino1 = models.ImageField(upload_to=pasti_path, null=True,
        blank=True)
    importo2 = models.FloatField(null=True, blank=True)
    valuta2 = models.CharField(max_length=3, choices=VALUTA_CHOICES,
        default='EUR')
    descrizione2 = models.CharField(max_length=255, null=True, blank=True)
    img_scontrino2 = models.ImageField(upload_to=pasti_path, null=True,
        blank=True)
    importo3 = models.FloatField(null=True, blank=True)
    valuta3 = models.CharField(max_length=3, choices=VALUTA_CHOICES,
        default='EUR')
    descrizione3 = models.CharField(max_length=255, null=True, blank=True)
    img_scontrino3 = models.ImageField(upload_to=pasti_path, null=True,
        blank=True)
    class Meta:
        verbose_name = "Pasto"
        verbose_name_plural = "Pasti"
```

Codice 4.5: Implementazione Modello Pasti.

Modelli Spesa e SpesaMissione

Per quanto riguarda i pernottamenti, i convegni e le altre spese, poiché condividevano gli stessi campi, ho applicato il principio **DRY**[4] evitando di creare un modello separato per ciascuna tipologia. Invece, ho creato un modello generico `Spesa`, che include i campi per la data, l'importo, la

[3]La variabile `valuta` può assumere valori predefiniti che sono specificati nella variabile `VALUTA_CHOICES`, una lista di tuple contenente le diverse valute disponibili.

[4]Don't Repeat Yourself.

valuta, la descrizione e l'immagine. Successivamente, per collegare ogni spesa alla missione corrispondente e definirne la categoria, ho sviluppato un modello `SpesaMissione`, che contiene una chiave esterna sia alla missione che alla spesa, oltre a un campo `tipo` che ne specifica la tipologia.

```
class Spesa(models.Model):
    data = models.DateField()
    importo = models.FloatField()
    valuta = models.CharField(max_length=3, choices=VALUTA_CHOICES,
       default='EUR')
    descrizione = models.CharField(max_length=100, null=True, blank=True)
    img_scontrino = models.ImageField(upload_to=profile_type_path, null=
       True, blank=True)
    class Meta:
        verbose_name = 'Spesa'
        verbose_name_plural = 'Spese'

class SpesaMissione(models.Model):
    missione = models.ForeignKey(Missione, on_delete=models.CASCADE)
    spesa = models.ForeignKey(Spesa, on_delete=models.CASCADE)
    tipo = models.CharField(max_length=13, choices=TIPO_SCONTRINO_CHOICES)

    class Meta:
        verbose_name = 'Spesa Missione'
        verbose_name_plural = 'Spese Missione'
```

Codice 4.6: Implementazione Modelli Spesa e SpesaMissione.

Una volta implementato il modello generico per le spese, è stato necessario apportare modifiche anche al modello `Missione`. Oltre alla rimozione dei precedenti campi `TextField`, ho introdotto tre nuovi campi: `pernottamenti`, `convegni` e `altre_spese`, ciascuno dei quali è un `ManyToManyField` [14]. Questo tipo di campo consente di creare una relazione molti-a-molti tra le missioni e le spese attraverso le istanze di `SpesaMissione`, utilizzando il modello come intermediario per gestire questa relazione.

```
pernottamenti = models.ManyToManyField(Spesa, through='
   PernottamentoMissione', related_name='pernottamenti_missioni')
```

```
convegni = models.ManyToManyField(Spesa, through='ConvegnoMissione',
    related_name='convegni_missioni')
altre_spese = models.ManyToManyField(Spesa, through='AltreSpeseMissione',
    related_name='altrespese_missioni')
```

Codice 4.7: Relazioni ManyToMany per le spese associate alle missioni.

Tuttavia, poiché non è possibile creare diverse relazioni many-to-many utilizzando la stessa classe intermedia, è stato necessario introdurre dei sotto-modelli **proxy**[5] di `SpesaMissione`. Queste sottoclassi permettono di distinguere e gestire in modo specifico le diverse tipologie all'interno del codice, pur mantenendo una singola tabella nel database.

```
class PernottamentoMissione(SpesaMissione):
    class Meta:
        proxy = True
class ConvegnoMissione(SpesaMissione):
    class Meta:
        proxy = True
class AltreSpeseMissione(SpesaMissione):
    class Meta:
        proxy = True
```

Codice 4.8: Definizione di modelli proxy per la gestione specializzata delle spese nelle missioni.

Campi ImageField

Come si può osservare dai frammenti di codice precedenti, per consentire l'inserimento degli allegati visivi, è stato aggiunto un campo `ImageField` a ciascun modello, incluso il modello `Trasporti`. Un campo `ImageField` in Django è specifico per la gestione dei file immagine, permettendo di caricarli e memorizzarli nel database, associandoli ai relativi record. Inoltre, questi file vengono salvati in cartelle organizzate sulla base dell'utente, della missione e della tipologia della spesa associata.

[5] In Django, un Proxy Model è una sottoclasse di un modello esistente che non introduce nuove tabelle nel database, ma permette di modificare o aggiungere funzionalità al modello originale

Questa organizzazione permette di mantenere un'archiviazione ordinata e facilmente accessibile, poiché ogni immagine è memorizzata in un percorso del tipo `users/user_id/id_missione/tipo_scontrino/filename`.

Struttura del database aggiornata

La figura 4.13 evidenzia i cambiamenti apportati alla struttura del database rispetto a quanto illustrato nella figura 3.1, con particolare attenzione alle tabelle per la gestione delle missioni e delle spese collegate.

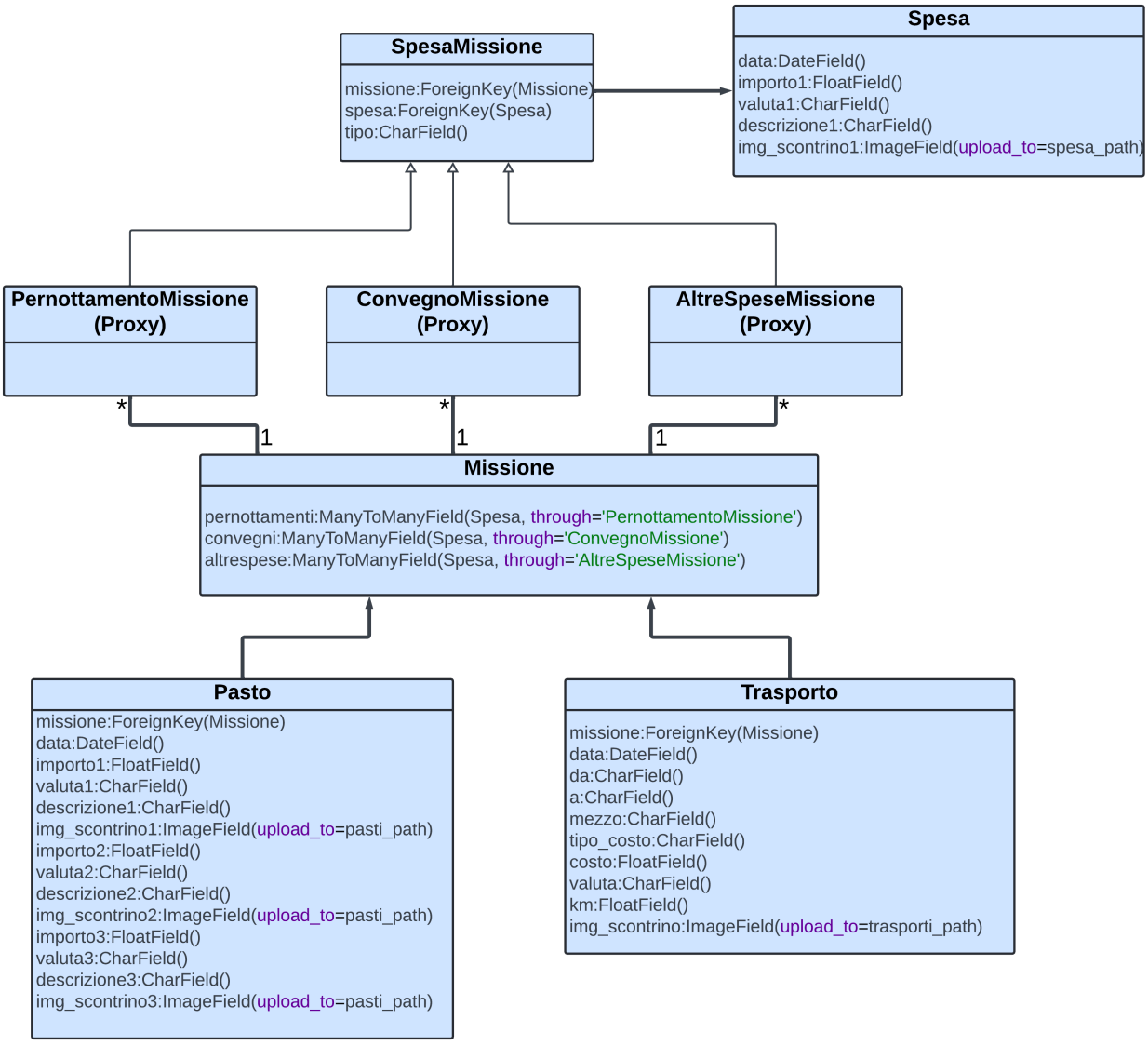

Figura 4.13: Diagramma delle tabelle del database dopo le modifiche.

4.2.2 Modifiche ai Form

Dopo aver aggiornato la struttura del database, è stato necessario adeguare anche il file `forms.py` per garantire una corretta gestione dei dati e delle immagini. In particolare, i vari form associati ai modelli sono stati modificati o creati ex novo per riflettere le nuove tabelle e campi.

Nel `TrasportoForm`, esistente già nel progetto, è stata aggiunta una singola linea di codice per includere il campo di input per l'immagine, dato che il modello `Trasporto` aveva già una struttura solida e ben definita. Questo piccolo cambiamento ha permesso di estendere la funzionalità del form per accogliere le immagini degli scontrini senza stravolgerne l'organizzazione.

Per i pasti e le altre spese, invece, è stato necessario creare nuovi form, in quanto quelli esistenti (`ScontrinoForm` e `ScontrinoExtraForm`) erano stati pensati per gestire i dati in formato JSON, senza supportare l'inserimento di immagini. Di conseguenza, sono stati sviluppati i form `PastiForm`[6] e `SpesaForm`, specificamente progettati per integrarsi con i modelli aggiornati. Questi non solo facilitano la memorizzazione delle spese in un formato più strutturato, ma supportano anche l'associazione delle immagini a ogni singola spesa.

```
1  class PastiForm(forms.ModelForm):
2      class Meta:
3          model = Pasti
4          fields = '__all__'
5          widgets = {
6              'data': forms.DateInput(attrs={'type': 'date', 'class': 'form-
                   control form-control-sm'}),
7              'importo1': forms.NumberInput(attrs={'class': 'form-control
                   form-control-sm'}),
8              'valuta1': forms.Select(attrs={'class': 'form-control form-
                   control-sm'}),
9              'descrizione1': forms.TextInput(attrs={'class': 'form-control
                   form-control-sm'}),
10             'img_scontrino1': PastiCustomClearableFileInput(attrs={'class'
                   : 'form-control form-control-sm'}),
```

[6]Nel codice della classe `PastiForm` è presente anche un dizionario denominato `labels`, utilizzato per personalizzare i nomi dei campi in fase di visualizzazione. Tuttavia, non è stato incluso per ragioni di spazio e leggibilità.

```
            'importo2': forms.NumberInput(attrs={'class': 'form-control
                form-control-sm'}),
            'valuta2': forms.Select(attrs={'class': 'form-control form-
                control-sm'}),
            'descrizione2': forms.TextInput(attrs={'class': 'form-control
                form-control-sm'}),
            'img_scontrino2': PastiCustomClearableFileInput(attrs={'class'
                : 'form-control form-control-sm'}),
            'importo3': forms.NumberInput(attrs={'class': 'form-control
                form-control-sm'}),
            'valuta3': forms.Select(attrs={'class': 'form-control form-
                control-sm'}),
            'descrizione3': forms.TextInput(attrs={'class': 'form-control
                form-control-sm'}),
            'img_scontrino3': PastiCustomClearableFileInput(attrs={'class'
                : 'form-control form-control-sm'}),
        }

class SpesaForm(forms.ModelForm):
    class Meta:
        model = Spesa
        fields ='__all__'
        widgets = {
            'data': forms.DateInput(attrs={'type': 'date', 'class': 'form-
                control form-control-sm', 'required': 'required',}),
            'importo': forms.NumberInput(attrs={'class': 'form-control
                form-control-sm', 'required': 'required',}),
            'valuta': forms.Select(attrs={'class': 'form-control form-
                control-sm'}),
            'descrizione': forms.TextInput(attrs={'class': 'form-control
                form-control-sm'}),
            'img_scontrino': CustomClearableFileInput(attrs={'class': '
                form-control form-control-sm', 'id': 'img_scontrino_input'
                }),
        }
```

Codice 4.9: Implementazione dei form specifici per i pasti e per le spese.

Un aspetto fondamentale dell'implementazione di questi form è stata l'integrazione dei formset. In Django, un **formset** [15] è una struttura che consente di gestire dinamicamente un insieme di form identici, tutti legati allo stesso modello. Questa funzionalità è particolarmente utile quando è necessario gestire collettivamente più istanze di un modello all'interno di una singola pagina di input. Ad esempio, nel contesto del nostro progetto, il formset `pasto_formset` permette di inserire più pasti per una singola missione, ciascuno con i suoi dettagli specifici come data, importo, descrizione, e l'immagine del relativo scontrino.

L'utilizzo dei formset semplifica l'interfaccia utente, consentendo di aggiungere, modificare o eliminare più record (come pasti o spese) in modo ordinato e intuitivo. L'aggiunta e l'eliminazione dei form dai formset viene gestita nell'applicazione attraverso un file JavaScript basato su jQuery, chiamato `formset.js`. Questo file, sviluppato da Stanislaus Madueke [16], è stato rilasciato sotto la licenza BSD ed è stato integrato nel progetto per fornire una gestione dinamica e user-friendly dei formset

```
1 trasporto_formset = inlineformset_factory(Missione, Trasporto,
     TrasportoForm,
2                 extra=0, can_delete=True, fields='__all__', min_num=1)
3
4 spesa_formset = modelformset_factory(Spesa, form=SpesaForm, extra=0,
5                 can_delete=True, min_num=1)
6
7 pasto_formset = inlineformset_factory(Missione, Pasti, PastiForm, extra=0,
                   can_delete=True, min_num=1)
```

Codice 4.10: Formset specifici per le varie tipologie di spesa.

Nel codice, i formset `trasporto_formset` e `pasto_formset` sono stati creati utilizzando `inlineformset_factory`, mentre `spesa_formset` è stato generato con `modelformset_factory` [17]. Il primo metodo è progettato per creare formset di modelli collegati a un altro tramite una ForeignKey, come `Trasporto` e `Pasti`, che sono associati a `Missione`. Questo approccio consente di gestire contemporaneamente il modello principale e i modelli correlati, garantendo la coerenza dei dati. Il secondo metodo, invece, è ideale quando si lavora su più istanze di un modello indipendente, senza collegamenti diretti tramite ForeignKey ad altri.

Campo di inserimento immagine

Come evidenziato nei paragrafi precedenti, una modifica comune a tutti i form è l'aggiunta di un campo per l'upload dell'immagine dello scontrino. Per personalizzare la visualizzazione di questo campo, ho creato due classi customizzate all'interno di un nuovo file chiamato `widgets.py`. Le classi `CustomClearableFileInput` e `PastiCustomClearableFileInput` sono entrambe sottoclassi di `ClearableFileInput` di Django e permettono di sovrascrivere il `template_name` da utilizzare per il rendering.

`CustomClearableFileInput` utilizza il template `custom_clearable_file_input.html`, mentre `PastiCustomClearableFileInput` fa riferimento al template `custom_clearable_file_input2.html`. Questi file sono stati creati per migliorare l'interfaccia utente durante l'inserimento delle immagini, offrendo un'esperienza più intuitiva e visivamente gradevole.

Poiché i due template sono pressoché identici, qui viene riportato solo il codice del template `custom_clearable_file_input.html`, omettendo per brevità le classi CSS di styling:

```
1  <head>
2     <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome
          /4.7.0/css/font-awesome.min.css" rel="stylesheet">
3  </head>
4  <div class="custom-file-input-container">
5      <input type="file" class="fileInput" name="{{ widget.name }}"
          style="display: none;" data-file-input>
6      <label class="file-label">
7          <i class="fa fa-upload"></i>
8      </label>
9      <span class="file-name" data-file-name>No Selected Img</span>
10     {% if widget.is_initial %}
11         <a href="{% url 'RimborsiApp:image_preview'
              id=widget.value.instance.id  %}" target="_blank" class="
              centered-icon">
12             <i class="fa fa-picture-o" style="font-size:29px;"></i>
13         </a>
14         <button type="submit" name="{{ widget.checkbox_name }}" value="
              on" class="delete-button">
```

```
15              <td style="text-align: center;"> <i class="fa fa-trash"></i>
                    </td>
16          </button>
17      {% endif %}
18  </div>
19  <script>
20   document.querySelectorAll('.fileInput').forEach((input, index) => {
21          const label = input.nextElementSibling;
22          const fileNameDisplay = label.nextElementSibling;
23          label.setAttribute('for', `fileInput${index}`);
24          input.id = `fileInput${index}`;
25          fileNameDisplay.title = 'No Selected Img';
26          input.addEventListener('change', function() {
27              var fileName = this.files[0].name;
28              fileNameDisplay.textContent = fileName;
29              fileNameDisplay.title = fileName;        }); });
30  </script>
```

Codice 4.11: Template customizzato `custom_clearable_file_input.html`.

Il widget presenta: un'icona per l'upload, la visualizzazione del nome del file selezionato, e, se presente, un'icona per la visualizzazione dell'immagine, oltre che un pulsante per la sua rimozione. La funzionalità interattiva è gestita da uno script JavaScript, che assegna dinamicamente un identificatore univoco a ogni campo di input, aggiorna il nome del file selezionato quando ne viene caricato uno nuovo, e permette di rimuovere l'immagine selezionata. Inoltre, l'accesso all'anteprima dell'immagine è soggetto a un controllo di accesso. Cliccando sull'icona, l'utente viene reindirizzato a una view che verifica se l'utente è autenticato e autorizzato a visualizzare l'immagine; in caso affermativo, l'immagine viene visualizzata. Di seguito è riportato il risultato ottenuto:

Figura 4.14: Campo di upload immagine prima del salvataggio.

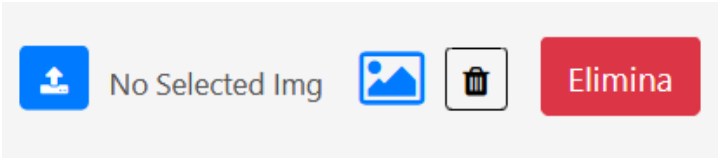

Figura 4.15: Campo di upload immagine dopo il salvataggio.

4.2.3 Modifiche alle View

La pesante modifica apportata ai modelli ha richiesto importanti interventi anche lato backend, con significative modifiche alle view. Queste sono state necessarie per adattare il codice esistente alla nuova struttura dati e per supportare le nuove funzionalità.

Nella versione originale dell'applicazione, come già detto, le spese venivano gestite tramite campi JSON memorizzati nel modello `Missione`, mentre per i trasporti esisteva già un modello separato. Di conseguenza, le view dedicate al caricamento, alla modifica e al salvataggio erano strettamente legate a questa struttura dati.

Con l'introduzione dei nuovi modelli `Spesa` e `Pasti`, è stato necessario riprogettare le view per interagire con le nuove tabelle del database e supportare il salvataggio delle immagini.

Modifiche alla view missione

Una delle prime view che ha richiesto un intervento significativo è stata la view `missione`, che gestisce la visualizzazione e la modifica dei dettagli di una missione. In precedenza, per caricare le spese associate, la vista recuperava i dati dai campi JSON memorizzati nel modello `Missione` stesso e li utilizzava per creare i formset. Nella nuova implementazione, la view popola i formset per pasti, pernottamenti, convegni e altre spese recuperando i dati direttamente dalle nuove tabelle del database, filtrandoli sulla base della missione presa in considerazione.

```
1 missione_form = MissioneForm(user=request.user, instance=missione,
2                                     initial={'automobile': missione.
                                         automobile})
3         missione_form.helper.form_action = reverse('RimborsiApp:missione',
              args=[id])
4
5         pasti_qs = Pasti.objects.filter(missione=missione).order_by('data'
             )
6         giorni = (missione.fine - missione.inizio).days
7         all_dates = [missione.inizio + datetime.timedelta(n) for n in
             range(giorni + 1)]
8         existing_pasti_dates = {pasto.data for pasto in pasti_qs}
```

```
9         missing_dates = [date for date in all_dates if date not in
              existing_pasti_dates]
10        for date in missing_dates:
11            Pasti.objects.create(missione=missione, data=date)
12
13        pasti_formset = pasto_formset(instance=missione ,queryset=pasti_qs
              )
14
15        pernottamenti_qs = Spesa.objects.filter(spesamissione__missione=
              missione, spesamissione__tipo='Pernottamento')
16        pernottamenti_formset = spesa_formset(queryset=pernottamenti_qs.
              order_by('data'), prefix='pernottamenti')
17
18        trasporti = Trasporto.objects.filter(missione=missione)
19        trasporti_formset = trasporto_formset(instance=missione, queryset=
              trasporti.order_by('data'))
20
21        convegni_qs = Spesa.objects.filter(spesamissione__missione=
              missione, spesamissione__tipo='Convegno')
22        convegni_formset = spesa_formset(queryset=convegni_qs.order_by('
              data'), prefix='convegni')
23
24        altrespese_qs = Spesa.objects.filter(spesamissione__missione=
              missione, spesamissione__tipo='Altro')
25        altrespese_formset = spesa_formset(queryset=altrespese_qs.order_by
              ('data'), prefix='altrespese')
26        response = {
27            'missione': missione,
28            'missione_form': missione_form,
29            'pasti_formset': pasti_formset,
30            'trasporti_formset': trasporti_formset,
31            'pernottamenti_formset': pernottamenti_formset,
32            'convegni_formset': convegni_formset,
33            'altrespese_formset': altrespese_formset,
34        }
```

```
35          return response
```

Codice 4.12: Estratto del codice per la popolazione dei formset nella view `missione`.

Come visibile dall'estratto di codice sopra riportato, per popolare i formset le spese vengono selezionate in base alla missione specifica. Per i pasti e i trasporti, il filtraggio avviene direttamente attraverso le relazioni tra missione e le rispettive entità. Le altre spese, che non sono direttamente collegate, vengono invece individuate tramite le istanze del modello `SpesaMissione`, che si collegano a loro volta alla missione. Infine, la view restituisce la risposta con i formset popolati.

Modifiche alle view per il salvataggio delle spese

La nuova struttura dati ha reso necessario modificare anche le view dedicate al salvataggio dei dati, che ora non sono più memorizzati in formato JSON, ma sono salvati direttamente nel database. Poiché i legami tra le spese e la missione differiscono rispetto a quelli tra pasti e trasporti, le view di salvataggio sono state adattate di conseguenza. Ad esempio, `salva_pasti` è simile a `salva_trasporti`, in quanto entrambe gestiscono dati direttamente collegati alla missione. Al contrario, `salva_pernottamenti` e le altre view richiedono un trattamento diverso, poiché le spese non sono collegate direttamente alla missione, ma al modello intermedio `SpesaMissione`.

Nelle view `salva_pasti` e `salva_trasporti`, i dati vengono semplicemente aggiornati o creati nel database utilizzando i formset corrispondenti, che, essendo stati creati da `inlineformset_factory`, operano direttamente sulle relazioni già esistenti tra la missione e le rispettive entità.

```
1  @login_required
2  def salva_pasti(request, id):
3      if request.method == 'POST':
4          missione = Missione.objects.get(id=id)
5          pasti_formset = pasto_formset(request.POST, request.FILES,
               instance=missione)
6          if pasti_formset.is_valid():
7              pasti_formset.save()
8              return redirect('RimborsiApp:missione', id)
9          else:
10             return HttpResponseServerError('Form non valido')
11     else:
```

```
12          return HttpResponseBadRequest()
```

Codice 4.13: Estratto del codice della view `salva_pasti`.

D'altro canto, le view come `salva_pernottamenti` richiedono una logica aggiuntiva per associare correttamente ogni spesa alla missione tramite il modello `SpesaMissione`. In queste viste, il processo di salvataggio prevede l'aggiornamento o la creazione di istanze di `Spesa` e la gestione esplicita del collegamento con la missione, poiché questi formset sono generati con `modelformset_factory`, che non gestisce implicitamente le relazioni tra i modelli.

```
1  @login_required
2  def salva_pernottamenti(request, id):
3      if request.method == 'POST':
4          missione = Missione.objects.get(user=request.user, id=id)
5          pernottamenti_formset = spesa_formset(request.POST, request.FILES,
               prefix='pernottamenti')
6          if pernottamenti_formset.is_valid():
7              for form in pernottamenti_formset.forms:
8                  if form.cleaned_data.get('DELETE'):
9                      form.instance.delete()
10                     SpesaMissione.objects.filter(spesa=form.instance).
                           delete()
11                 else:
12                     img_scontrino = form.instance.img_scontrino
13                     form.instance.img_scontrino = None
14                     instance = form.save(commit=False)
15                     instance.save()
16                     SpesaMissione.objects.update_or_create(missione=
                           missione, spesa=instance, tipo='PERNOTTAMENTO')
17                     if img_scontrino:
18                         instance.img_scontrino = img_scontrino
19                         instance.save()
20             return redirect('RimborsiApp:missione', id)
21         else:
22             return HttpResponseServerError('Form non valido')
23     else:
```

```
24          return HttpResponseBadRequest()
```

Codice 4.14: Estratto del codice della view `salva_pernottamenti`.

Modifiche alle view clona_mission e resoconto_data

Anche le views `clona_missione` e `resoconto_data` hanno subito importanti aggiornamenti per adeguarsi alla nuova struttura dei modelli e migliorare la gestione delle spese all'interno dell'applicazione.

La vista `clona_missione`, che in precedenza si limitava a duplicare le missioni e i trasporti associati[7], è stata ampliata per includere anche la clonazione di tutte le spese e dei relativi allegati, come le immagini degli scontrini. Questo aggiornamento garantisce che ogni missione clonata mantenga tutte le informazioni rilevanti, compresi i documenti di supporto. Durante il processo di clonazione, vengono create nuove istanze delle spese e delle immagini, mantenendo l'integrità dei dati e assicurando che la nuova missione sia una copia fedele dell'originale, ma con le proprie entità indipendenti.

La view `resoconto_data`, responsabile del calcolo e visualizzazione dei totali di una missione, è stata riscritta per sfruttare i nuovi modelli `Spesa` e `Pasti`. Invece di estrarre e sommare i dati dai campi JSON, ora la view interagisce direttamente con il database, raccogliendo le spese registrate e calcolando i totali dai campi delle tabelle.

4.2.4 Funzioni per la Compilazione dei Documenti di Rimborso

L'introduzione della nuova struttura dati e l'abbandono dei JSON per il salvataggio delle informazioni hanno reso necessaria una revisione sostanziale delle funzioni dedicate alla compilazione dei documenti di rimborso. In particolare, per quanto riguarda la gestione delle spese, i dati non vengono più estratti dai campi JSON all'interno del modello `Missione`, ma verranno recuperati direttamente dalle tabelle `Spesa` e `Pasti` nel database.

[7] Nella versione originale, solo i trasporti erano gestiti attraverso un modello indipendente, mentre gli altri campi erano dei TextField dentro il modello `missione`. Per questo la clonazione di questa tabella consentiva già di copiare tutte le spese.

Entrando più nel dettaglio del codice, la funzione `compila_parte_2`, responsabile della compilazione della seconda parte del modulo di rimborso, è stata quella che ha richiesto il maggior numero di modifiche. Come è già stato fatto in altre parti del progetto, le spese e i pasti vengono selezionati in base alla missione o alla relativa istanza di `SpesaMissione`. Una volta individuate tutte le spese pertinenti, vengono estratti gli importi e le descrizioni, che vengono poi inseriti nel documento di rimborso. Inoltre, poiché i pasti possono avere fino a tre campi di importo, è stato necessario implementare un controllo specifico per garantire che tutti i valori siano correttamente inclusi nel documento finale.

Di seguito sono riportati gli estratti di codice che illustrano le modifiche apportate. Il primo mostra come le spese vengono recuperate dal database utilizzando il modello `Missione` per filtrare i dati rilevanti:

```
1     missione = Missione.objects.get(user=request.user, id=id)
2
3     # Recupero delle spese dal database
4     pernottamenti = Spesa.objects.filter(spesamissione__missione=missione,
          spesamissione__tipo='PERNOTTAMENTO')
5     pasti = Pasti.objects.filter(missione=missione)
6     convegni = Spesa.objects.filter(spesamissione__missione=missione,
          spesamissione__tipo='CONVEGNO')
7     altre_spese = Spesa.objects.filter(spesamissione__missione=missione,
          spesamissione__tipo='ALTRO')
```

Codice 4.15: Estratto della funzione `compila_parte_2`.

Il secondo estratto mostra invece il codice responsabile dell'inserimento delle spese nel documento di rimborso. La logica differenzia i pasti dalle altre spese, gestendo correttamente l'inserimento dei campi importo e descrizione all'interno delle tabelle del documento. La parte relativa ai trasporti non è stata inclusa poiché non ha subito modifiche rispetto alla versione originale dell'applicazione.

```
1 for index, (key, queryset) in enumerate(spese_dict.items(), start=1):
2     table = document.tables[index]
3
4     row_index = 1
5     for spesa in queryset:
```

```
6         if key == 'pasto':
7             for j in range(1, 4):
8                 importo = getattr(spesa, f'importo{j}')
9                 if importo:
10                    valuta = getattr(spesa, f'valuta{j}')
11                    descrizione = getattr(spesa, f'descrizione{j}')
12                    data = spesa.data
13                    costo_str = f'{importo:.2f} {valuta}'
14                    if valuta != 'EUR':
15                        costo_in_euro = money_exchange(data, valuta,
                              importo)
16                        costo_str += f' ({costo_in_euro:.2f} EUR)'
17
18                    if row_index >= len(table.rows):
19                        table.add_row()
20
21                    table.cell(row_index, 0).text = data.strftime('%d/%m/%
                          Y')
22                    table.cell(row_index, 1).text = descrizione if
                          descrizione else ''
23                    table.cell(row_index, 2).text = costo_str
24                    table.rows[row_index].height = Cm(0.61)
25                    row_index += 1
26        else:
27            importo = spesa.importo
28            valuta = spesa.valuta
29            descrizione = spesa.descrizione
30            data = spesa.data
31            costo_str = f'{importo:.2f} {valuta}'
32            if valuta != 'EUR':
33                costo_in_euro = money_exchange(data, valuta, importo)
34                costo_str += f' ({costo_in_euro:.2f} EUR)'
35
36            if row_index >= len(table.rows):
37                table.add_row()
38
```

```
39              table.cell(row_index, 0).text = data.strftime('%d/%m/%Y')
40              table.cell(row_index, 1).text = descrizione if descrizione
                    else ''
41              table.cell(row_index, 2).text = costo_str
42              table.rows[row_index].height = Cm(0.61)
43              row_index += 1
```

Codice 4.16: Estratto della funzione `compila_parte_2`.

4.3 Migrazione dei Dati da Vecchio a Nuovo Formato

L'ultima criticità che è stato necessario risolvere prima dell'effettiva distribuzione delle modifiche riguarda la migrazione dei dati già presenti sull'applicazione. Come già menzionato, le informazioni relative a pasti, pernottamenti, convegni e altre spese erano precedentemente salvate come JSON all'interno della tabella `Missione`. Per garantire la continuità dei dati e l'integrità delle nuove funzionalità, è stato quindi necessario sviluppare degli script Python che potessero migrare queste informazioni nel nuovo formato.

Le funzioni si occupano di scorrere tutte le missioni esistenti, analizzare i `TextField` associati a pasti, pernottamenti, convegni e altre spese, e creare per ciascuna voce un'istanza corrispondente del nuovo modello. In particolare, per ogni pasto viene creata un'istanza del modello `Pasti` collegata alla missione, mentre per le altre spese viene creata un'istanza del modello `Spesa`, insieme alla corrispondente associazione `SpesaMissione`, che ne definisce il tipo e la collega alla missione.

I vari script sono stati inseriti nel file `utils.py`, che contiene quelle funzioni che non fanno parte effettivamente dell'applicazione ma hanno una qualche utilità ad essa associata. Di seguito è riportato un esempio di script utilizzato per migrare i dati relativi ai pernottamenti.

```
1  def migra_pernottamenti():
2      missioni = Missione.objects.all()
3
4      for missione in missioni:
5          if missione.pernottamento:
6              pernottamenti = json.loads(missione.pernottamento)
7              for pernottamento in pernottamenti:
8                  if not pernottamento.get('DELETE', False):
9                      data = dt.strptime(pernottamento['data'], '%Y-%m-%d').
                           date()
10                     importo = float(pernottamento['s1'])
11                     valuta = pernottamento.get("v1", "EUR")
12                     descrizione = pernottamento.get('d1', '')
13                     spesa = Spesa.objects.create(
14                         data=data,
15                         importo=importo,
```

```
                valuta=valuta,
                descrizione=descrizione
            )
            SpesaMissione.objects.create(
                missione=missione,
                spesa=spesa,
                tipo='PERNOTTAMENTO'
            )
```

Codice 4.17: Script per la migrazione dei dati relativi ai pernottamenti.

Sebbene il formato JSON sia identico per tutti i tipi di spese, lo script per i pasti presenta leggere differenze per gestire i campi `importo`, `valuta` e `descrizione`, che possono essere ripetuti fino a tre volte per ogni pasto. Inoltre, per i pasti non è stata inserita la parte di creazione delle istanze `SpesaMissione` poiché non necessaria.

5. Conclusioni

In questo elaborato sono state descritte in dettaglio le modifiche apportate all'applicazione Missioni, realizzate dopo un'accurata analisi delle sue funzionalità e delle criticità emerse. Le modifiche sono state avviate in seguito a uno studio approfondito delle tecnologie utilizzate per lo sviluppo dell'applicazione, che mi ha consentito di acquisire competenze essenziali nello sviluppo software e nella gestione dei dati, fondamentali per il progetto stesso.

Le modifiche principali si sono focalizzate su tre aree fondamentali: l'adattamento dell'applicazione per dispositivi mobili, l'integrazione di un servizio di upload delle immagini per le spese di missione e la migrazione dei dati dal vecchio al nuovo formato. L'implementazione di un design responsivo ha migliorato significativamente l'esperienza utente, rendendo l'applicazione più accessibile e fruibile su dispositivi con schermi di diverse dimensioni. In più, l'aggiunta del servizio di upload delle immagini ha ampliato le funzionalità dell'applicazione, offrendo agli utenti una gestione più completa delle spese di missione, e fornendo lo spunto per modificazioni future.

Infatti, nonostante queste modifiche abbiano risolto molte delle problematiche esistenti, il progetto offre ancora importanti margini di sviluppo. Ad esempio, potrebbe essere utile implementare un nuovo campo nel resoconto missione che includa un documento contenente tutte le immagini degli scontrini, organizzate per tipo di spesa. Questo documento fornirebbe una certificazione delle spese effettuate, migliorando la trasparenza e la tracciabilità delle operazioni di rimborso.

In conclusione, il lavoro svolto su questo progetto non solo ha contribuito a migliorare l'applicazione Missioni, ma è stato anche un'opportunità significativa per la mia crescita professionale. Ho potuto rafforzare le mie competenze nel campo dello sviluppo software, acquisendo preziose esperienze pratiche. In particolare, imparare ad approcciare un progetto già esistente, a comprenderne la logica, a valutare le soluzioni più appropriate e a confrontarmi costruttivamente con il professore; tutte qualità che considero fondamentali per la mia futura carriera professionale.

Sitografia

[1] ***Python Official Site***. *What is python.* URL: https://www.python.org/doc/essays/blurb/.

[2] ***Geeksforgeeks***. *Hystory of python.* URL: https://www.geeksforgeeks.org/history-of-python/.

[3] ***Alex DeBrie - Stackify***. *Python Garbage Collection: What It Is and How It Works.* URL: https://stackify.com/python-garbage-collection/.

[4] ***Kyle Daigle GitHub Staff***. *Octoverse: The state of open source and rise of AI in 2023.* URL: https://github.blog/news-insights/research/the-state-of-open-source-and-ai/.

[5] ***JetBrains***. *Python Developers Survey 2018 Results.* URL: https://www.jetbrains.com/research/python-developers-survey-2018/.

[6] ***Django Documentation***. *Design philosophies.* URL: https://docs.djangoproject.com/en/5.0/misc/design-philosophies/.

[7] ***Emmanuel Olakunle***. *Creating an MTV (Model-Template-View) Architecture in Django.* URL: https://python.plainenglish.io/creating-an-mtv-model-template-view-architecture-in-django-59c6502e15c8.

[8] ***Julio Cesar Siles Perez***. *Django Project Structure: A Comprehensive Guide.* URL: https://medium.com/django-unleashed/django-project-structure-a-comprehensive-guide-4b2ddbf2b6b8.

[9] ***Mark Otto, Jacob Thornton, and Bootstrap contributors***. *Bootstrap.* URL: https://getbootstrap.com/.

[10] ***Federico Bolelli and Michele Cancilla***. *Missioni Unimore.* URL: https://missioni.ing.unimore.it/.

[11] ***Bootstrap Documentation***. *Grid system.* URL: https://getbootstrap.com/docs/4.0/layout/grid/.

[12] ***Miguel Araujo, Daniel Feldroy and contributors***. *Forms have never been this crispy.* URL: https://django-crispy-forms.readthedocs.io/en/latest/.

[13] ***Bootstrap Documentation***. *Bootstrap cards.* URL:
https://getbootstrap.com/docs/4.3/components/card/.

[14] ***Django Documentation***. *Many-to-many relationships.* URL:
https://docs.djangoproject.com/en/5.1/topics/db/examples/many_to_many/.

[15] ***Geeksforgeeks***. *Django Formsets.* URL:
https://www.geeksforgeeks.org/django-formsets/.

[16] ***Stanislaus Madueke***. *Django-dynamic-formset.* URL:
https://github.com/elo80ka/django-dynamic-formset.

[17] ***Django Documentation***. *Model Form Functions.* URL:
https://docs.djangoproject.com/en/5.1/ref/forms/models/.