

SUPSI

Gestione Federazione Svizzera di Twirling

Studente

Peter Catania

Relatore

Brocco Amos

Correlatore

Committente

Fulvio Frapolli

Anna Compaore

Corso di laurea

Ingegneria Informatica TP

Codice Progetto

C10627

Anno

2022/2023

Indice

Abstract (Italiano)	6
Abstract (Inglese)	7
Introduzione	8
1 Motivazione e Contesto	9
1.1 Obbiettivi	9
1.2 Requisiti	10
1.3 Struttura di questo documento	10
2 Stato dell'arte	11
2.1 Gestione tramite file Excel	12
2.2 Django	13
2.2.1 Autenticazione	13
2.2.2 Infrastruttura	14
2.2.3 Ciclo di richiesta e risposta HTTP	16
2.2.4 Suddivisione in Applicazione Distinte	17
2.2.5 Le Diverse View	17
2.2.6 Dispatch by Method	18
2.2.7 Controllo di base	18
2.2.8 Object Relational Mapper	18
2.2.9 Sistema Autenticazione Django	19
2.2.10 Testing	19
2.2.11 Test con Unittest	19
2.2.12 La Classe TestCase	20
2.3 Costruzione Pagine Responsive	20
2.4 HTMX e Single-page applications	21
2.5 Risultati ricerca	22
2.5.1 Tecnologie adottate per il Backend	22
2.5.2 Tecnologie adottate per il Frontend	23
3 Soluzione	24
3.1 Pianificazione	25
3.2 Database	26
4 Implementazione	27
4.1 Scripts	27
4.1.1 Script per la Traduzione	28
4.1.2 Script per l'inserimento dei dati di default del database	28

4.1.3	Script per il reset del database	29
4.1.4	Script per il deploy su PythonAnywhere	30
4.2	Struttura Base	31
4.3	Evitare il CSRF	31
4.4	Traduzione di testo	31
4.4.1	Traduzioni con Rosetta	31
4.4.2	Traduzione di Testo all'interno di files Java Script	32
4.4.3	Traduzioni Datatable	32
4.5	Verifica dell'identità dell'utente autenticato	32
4.6	Relazioni Modelli Django	33
4.7	Creazione di permessi personalizzati	33
4.8	Modelli Astratti ed Ereditarietà	34
4.9	Fornire Variabili Globali a tutti i Template	35
4.10	Risoluzione Errori	35
5	Testing	36
5.1	Metodologia di testing	36
5.2	Librerie Scelte per Testing	36
5.3	Moking	37
5.3.1	Decorator Patch	38
6	Deployment	39
7	Conclusioni	40
7.1	Risultati Ottenuti	40
7.2	Consuntivo	40
7.3	Considerazioni Personali	41
7.4	Possibili sviluppi futuri	41
7.5	Conclusioni finali	42
	Bibliografia	43
	Appendici	44
A	Funzionalità base di Django	44
B	Traduzione con Django	56
C	Verifica dell'identità dell'utente autenticato	63
D	Context Processor	65

E	Risoluzioni Errori	66
F	Deployment	68
Glossario		76

Indice Delle figure

<i>Figura 1 - Excel Gestione Membri e Club</i>	12
<i>Figura 2 - MVTU</i>	14
<i>Figura 3 - MVC</i>	14
<i>Figura 4 - Struttura Progetto Django</i>	15
<i>Figura 5 - Flusso di richieste e risposte Django</i>	16
<i>Figura 6 - Gerarchia classe django.test.TestCase</i>	20
<i>Figura 7 - Gantt Pianificazione Iniziale</i>	25
<i>Figura 8 - Progettazione iniziale database</i>	26
<i>Figura 9 - Gerarchia Cartelle contenuti i comandi/ scripts</i>	27
<i>Figura 10 - Cartella locale</i>	57
<i>Figura 11 - Pannello di Traduzione di Rosetta</i>	61
<i>Figura 12 - Tasto di Suggerimento Traduzione</i>	61
<i>Figura 13 - Configurazione Staticfiles PythonAnywhere</i>	69
<i>Figura 14 - PythonAnywhere Risultato finale della Configurazione Virtualenv</i>	70
<i>Figura 15 - PythonAnywhere Aprire Nuova Console</i>	71
<i>Figura 16 - PythonAnywhere Accesso al Ambiente Virtuale e alla root del progetto</i>	71
<i>Figura 17 - Aggiunta Permesso FSTB Admin</i>	74
<i>Figura 18 - PythonAnywhere Ricaricare Applicazione Web</i>	75

Abstract (Italiano)

Lo sviluppo web moderno è diventato notevolmente più complesso rispetto ai giorni in cui HTML, CSS e Vanilla JS erano sufficienti. L'utilizzo di framework front-end come React e Angular è diventato lo standard dell'industria per realizzare applicazioni web altamente scalabili e complesse. Tuttavia, questo progetto esplora un'alternativa, dimostrando come Django, possa essere una soluzione efficace e moderna.

La Federazione Svizzera di Twirling Baton deve gestire efficacemente i suoi membri e i circa 20 club affiliati. Attualmente, questo compito viene svolto utilizzando numerosi file Excel, alcuni dei quali di dimensioni considerevoli. Purtroppo, questa soluzione comporta complicazioni nel reperimento e nella modifica di informazioni specifiche, rendendo il processo dispendioso in termini di tempo e suscettibile a errori.

Per questo motivo è stata ideata un'applicazione web basata su Django, per risolvere le problematiche causate dall'utilizzo di file Excel. Si è sviluppata utilizzando metodologie agili, così da consegnare un'applicazione che semplifichi e ottimizzi il più possibile le operazioni chiave come la registrazione dei club e la gestione dei membri. Questa nuova soluzione è stata progettata per essere semplice, intuitiva, efficiente e scalabile, dimostrando come Django, integrato con alcune librerie front-end specifiche, possa rispondere alle esigenze del moderno sviluppo web.

La piattaforma permette l'autenticazione degli utenti e mette a disposizione strumenti per la registrazione e la gestione sia di club che di membri. Queste funzionalità sono accessibili sia da un amministratore centrale che dai responsabili di singoli club. Inoltre, il sistema include meccanismi di verifica che permettono all'amministratore centrale di evitare l'applicazione di modifiche ritenute inappropriate o errate da parte dei responsabili dei club. Tutto è stato realizzato con la massima attenzione alla facilità d'uso e all'efficienza del sistema, rendendo questo progetto un esempio di come Django possa essere utilizzato per soddisfare requisiti complessi.

Nel complesso, questo progetto offre un'alternativa valida ai framework front-end più complessi, dimostrando che Django è non solo capace di gestire la logica lato server, ma può anche offrire una soluzione completa per le esigenze di sviluppo web moderno.

Abstract (Inglese)

Modern web development has become considerably more complex than in the days when HTML, CSS and Vanilla JS were sufficient. The use of front-end frameworks such as React and Angular has become the industry standard for building highly scalable and complex web applications. However, this project explores an alternative, demonstrating how Django, can be an effective and modern solution.

The Swiss Baton Twirling Federation needs to effectively manage its members and approximately 20 affiliated clubs. Currently, this task is accomplished using numerous Excel files, some of them of considerable size. Unfortunately, this solution leads to complications in retrieving and editing specific information, making the process time-consuming and prone to errors.

For this reason, a Django-based web application was designed to solve the problems caused by using Excel files. It was developed using agile methodologies so as to deliver an application that simplifies and streamlines key operations such as club registration and membership management as much as possible. This new solution was designed to be simple, intuitive, efficient, and scalable, demonstrating how Django, integrated with some specific front-end libraries, can meet the needs of modern Web development.

The platform allows user authentication and provides tools for registering and managing both clubs and members. These features are accessible both by a central administrator and by individual club managers. In addition, the system includes verification mechanisms that allow the central administrator to prevent the application of changes deemed inappropriate or erroneous by club managers. Everything has been implemented with a focus on ease of use and system efficiency, making this project an example of how Django can be used to meet complex requirements.

Overall, this project offers a viable alternative to more complex front-end frameworks, demonstrating that Django is not only capable of handling server-side logic, but can also offer a complete solution for modern Web development needs.

Introduzione

Il Twirling è uno sport che combina capacità ginniche, senso del ritmo e della danza con l'abilità di lanciare e far volteggiare un'asta di acciaio cromato. Questo sport utilizza un bastone di 70 centimetri di lunghezza con pomelli di gomma alle estremità.

Il Twirling ha iniziato a evolversi come disciplina sportiva dalla metà del secolo scorso. È stato introdotto in Giappone e si è diffuso in Europa a partire dal 1961. La Francia è stata il primo paese a utilizzare bastoni di legno. In Svizzera, negli anni '70, le Majorettes sono state trasformate in atlete che offrivano spettacoli agonistici di alta qualità. Nel 1972 è stata fondata la Federazione svizzera di Sport Twirling Baton (FSTB), che successivamente si è associata alla Federazione europea (CETB) e alla Federazione mondiale (WBTF). Nel 1980, la FSTB è stata ammessa nell'Associazione svizzera dello sport.

Negli Stati Uniti, il Twirling coinvolge un milione di giovani praticanti, mentre in Giappone ha una diffusione più ampia, considerandosi una disciplina sportiva scolastica. Globalmente, il numero di appassionati di questo sport si aggira intorno ai cinque milioni e continua a crescere. In Svizzera, il Twirling ha una presenza più limitata, con circa una trentina di partecipanti a Bellinzona. È rilevante notare che questa pratica non è riservata esclusivamente alle donne, ma coinvolge anche giovani uomini che possono partecipare in diverse discipline e categorie.

Attualmente, la FSTB affronta una sfida significativa nella gestione dei propri membri e club. Questo processo si basa principalmente su numerosi file Excel, alcuni dei quali sono di dimensioni considerevoli. Di conseguenza, il reperimento e la modifica di determinate informazioni risultano estremamente complessi e dispendiosi in termini di tempo. Questo sistema frammentato porta a inefficienze nella gestione complessiva, con il rischio di omissioni ed errori.

Al fine di superare questa problematica, è stata formulata una soluzione basata su tecnologie Web. La proposta si concentra sul miglioramento dell'intero processo di gestione delle informazioni all'interno della FSTB. L'approccio prevede l'implementazione di un'applicazione web basata sul framework Django. Questa applicazione web consentirà un accesso rapido e semplice alle informazioni da diversi dispositivi, ovunque vi sia una connessione internet disponibile. L'obiettivo principale è quello di prevenire gli errori commessi dai responsabili e dagli amministratori durante la gestione delle informazioni, garantendo al contempo una piattaforma centralizzata e affidabile per la gestione dei membri e dei club all'interno della FSTB.

Questo progetto ha anche lo scopo di concludere il percorso di Bachelor in ingegneria informatica presso la SUPSI, fungendo da dimostrazione pratica delle competenze acquisite dagli studenti. La tesi è stata selezionata da un elenco di proposte disponibili, offrendo l'opportunità di applicare le conoscenze teoriche in un contesto reale.

1 Motivazione e Contesto

La FSTB si trova attualmente di fronte a un ostacolo notevole per quanto riguarda la gestione efficiente dei suoi membri e club. L'amministrazione corrente si appoggia ampiamente su un insieme di fogli Excel, alcuni dei quali sono abbastanza voluminosi. Questo approccio rende il recupero e la modifica delle informazioni una procedura lenta e che necessita di molto sforzo. Il sistema attuale aumenta inoltre la possibilità che si verificano errori e omissioni, poiché alcuni responsabili possiedono una limitata familiarità nell'uso di dispositivi digitali.

In Svizzera, esistono circa una ventina di club sotto la supervisione della FSTB. Ogni club ha almeno un amministratore dedicato alla gestione delle attività e delle informazioni dei membri. Inoltre, ci sono responsabili a livello federale che hanno il compito aggiuntivo di supervisionare e amministrare l'intera rete di club.

Questo progetto punta a risolvere questi problemi, offrendo un modo più semplice ed efficiente di gestire le informazioni sia per i singoli club che per la FSTB nel suo complesso.

1.1 Obiettivi

L'obiettivo principale di questo progetto è la creazione di un gestionale web basato su Django, per i membri della FSTB. Gli utilizzatori del sistema saranno gli amministratori della federazione e i responsabili dei club svizzeri.

Il progetto mira a realizzare un sistema per la gestione di membri, club e competizioni all'interno della FSTB. La prima fase include la raccolta e l'organizzazione delle necessità del progetto, con l'obiettivo definire con chiarezza i requisiti funzionali e non-funzionali.

Per l'amministratore della FSTB, il sistema deve permettere di gestire i membri, come atleti e membri del comitato. Questo significa che l'amministratore potrà aggiungere nuovi membri, eliminare quelli vecchi e aggiornare le informazioni di quelli esistenti. Inoltre, dovrà essere in grado di creare e gestire eventi sportivi e aggiornare le presenze e le assenze.

I responsabili dei club dovranno essere in grado di iscrivere nuovi membri e apportare modifiche ai membri esistenti. Il sistema dovrà anche permettere di iscrivere gli atleti alle competizioni, verificando che siano rispettati i requisiti necessari.

Un altro obiettivo importante è rendere il sistema disponibile in diverse lingue, in modo che gli utenti possano utilizzarlo nella lingua che preferiscono.

Il sistema dovrà gestire le competizioni. Questo include la creazione di competizioni con dettagli come il nome, la posizione e le date, l'aggiornamento delle presenze e assenze, e la gestione del responsabile della competizione.

In generale il progetto ha lo scopo di creare un sistema completo che possa gestire membri e competizioni, supportare diverse lingue e soddisfare le esigenze di diversi utenti, come l'amministratore della FSTB e i responsabili dei club. Il tutto dovrà essere realizzato seguendo un approccio agile e fornendo una documentazione chiara e dettagliata.

Infine, il committente ha chiaramente specificato la preferenza di evitare l'utilizzo di framework JavaScript. Ha inoltre indicato che MySQL dovrebbe essere il database di scelta e che il sito sarà caricato e ospitato sulla piattaforma di hosting PythonAnywhere.

Gestione federazione svizzera di Twirling

1.2 Requisiti

L'obiettivo del progetto è di creare un gestionale web dei membri della Federazione svizzera di Twirling (di seguito FSTB).

Gli utilizzatori di tale gestionale saranno da una parte l'amministratore della federazione e dall'altra i responsabili dei vari clubs svizzeri.

L'amministratore della FSTB deve poter gestire:

1. I membri della FSTB (atleti, membri di comitato, monitori, ...).
 - a. validazione nuovi membri
 - b. eliminazione vecchi membri
 - c. modifiche dei dati dei membri già inseriti nel sistema
2. la creazione di competizioni (campionato Svizzero, Coppa svizzera, ...)
3. l'aggiornamento delle presenze/assenze ai vari campionati

I responsabili dei club devono poter gestire:

1. L'annuncio di nuovi membri e/o modifiche a membri esistenti (atleti, membri di comitato, monitori, ...)
2. L'iscrizione di atleti alle varie competizioni (il sistema controlla che i requisiti necessari all'iscrizione siano validi)

1.3 Struttura di questo documento

La documentazione del progetto segue un approccio logico e strutturato per descrivere il lavoro svolto è quello che è si è appreso. In questo primo capitolo, viene presentato il problema e lo scopo del progetto, fornendo una motivazione dettagliata e spiegando il contesto in cui si inserisce.

Il secondo capitolo è dedicato alla valutazione delle varie alternative per risolvere il problema. Qui viene descritta la prima attività di ricerca svolta per capire i vantaggi e gli svantaggi di certe tecnologie web disponibili.

Nel terzo capitolo, vengono elencate le scelte effettivamente fatte e viene spiegato il motivo per cui si è optato per quelle specifiche.

All'interno del quarto capitolo si esamina la fase di implementazione, dettagliando i risultati raggiunti durante lo sviluppo.

Nel quinto capitolo si discute la metodologia dei test adottata, specificando l'approccio consigliato in ambiente Django.

Invece nel sesto capitolo si illustrano le procedure per il deploy del sistema django su PythonAnywhere.

Infine, il settimo e ultimo capitolo fornisce le conclusioni del lavoro svolto, facendo il punto della situazione con i risultati ottenuti e discutendo le considerazioni personali, i limiti del sistema e i possibili sviluppi futuri.

2 Stato dell'arte

Nell'attuale panorama tecnologico, il web è una piattaforma in continua evoluzione, arricchita da una vasta gamma di siti e applicazioni web. Per lo sviluppo, gli strumenti più diffusi sono i framework e le librerie, che offrono componenti pre-costruiti o facilitano la creazione di nuovi elementi. Tra queste soluzioni, i framework frontend basati su JavaScript dominano il mercato, poiché permettono la creazione agile di interfacce utente interattive.

Tuttavia, nel contesto di questo progetto, si opta per l'uso di Django. Sebbene sia possibile integrare un framework frontend, ciò potrebbe aggiungere una complessità indesiderata. L'obiettivo è quindi esplorare alternative tecnologiche che possano sostituire efficacemente un framework frontend, permettendo allo stesso tempo la costruzione di un'interfaccia utente responsive con la massima semplicità. Questa fase servirà anche a comprendere le potenzialità di Django e come esse possano essere integrate nel sistema.

2.1 Gestione tramite file Excel

Per gestire la registrazione dei membri, gli amministratori della FSTB hanno originariamente impiegato file Excel. In questi documenti sono catalogate tutte le informazioni rilevanti dei membri, quali nome, cognome, indirizzo e data di nascita. Ogni foglio all'interno del file è dedicato a un club specifico e include anche informazioni essenziali relative al club stesso.

1	LICENCES 2023 - BELLINZONA				
2					
3					
4	Nom	Prénom	Adresse	Localité	Date de naissance
5					
6	CLUB				
7	ATHLETES ACTIVES				
8	Mustro	Silvia	Via El Stradun 28	6500 Bellinzona	07.04.11
9	Paretta	Sofia	Via Roccio 9	6517 Arbedo	21.01.10
10	Scopello	Amy	Via San Bernardino 1	6533 Lumino	17.11.08
11	Dinoia	Marco	Via Stradun 58	6532 Castione	17.09.2017
12	Ertori	Marisa	Via Pomelli 4	6702 Claro	22.01.2017
13	Zouzzi	Maya	Via Struzzi 32B	6500 Bellinzona	21.01.10
14	Montagna	Tommaso	Al Monastro 43	6702 Claro	22.01.2017
15	Capri	Gino	Via San Bernardino 41	6500 Bellinzona	15.05.2015
16	Tarast	Bianca	Via San Giovanni 4	6517 Arbedo	05.09.2016
17	MONITEURS				
18	Olbiani	Alessandro	Via Stradone Vecchio Sud 40	6500 Bellinzona	17.10.89
19	Aurora	Valentina	Via Cantonale 14	6532 Castione	01.02.89
20	AIDE / AUTRE				
21					
22	COMITE				
23	Postizzo	Carla	Vicolo Strutto 7	6710 Biasca	11.02.98
24					

Figura 1 - Excel Gestione Membri e Club

Dato che il sistema esistente presenta una certa complessità, l'uso di un framework diventa necessario. Poiché i requisiti non funzionali specificano l'impiego di Django, si opterà per questo framework per lo sviluppo del progetto.

2.2 Django

Django è un framework web potente e flessibile che offre molte funzionalità predefinite, sicurezza integrata, scalabilità e facilità di manutenzione. È una scelta popolare per lo sviluppo di siti web complessi e di successo. È scritto in Python e gode di una grande popolarità e supporto dalla comunità degli sviluppatori. Offre numerosi vantaggi per lo sviluppo di siti web, come:

- **Completezza:** Django segue il principio delle "batterie incluse"¹ fornendo molte componenti predefinite e pronte all'uso. Ciò permette agli sviluppatori di concentrarsi sulla logica dell'applicazione senza dover reinventare funzionalità comuni.
- **Sicurezza:** Django è progettato con un'enfasi sulla sicurezza e offre molte funzionalità per evitare vulnerabilità comuni, come [SQL injection](#), [cross-site scripting](#) e [cross-site request forgery](#). Queste protezioni sono integrate di default nel framework.
- **Versatilità:** Django può essere utilizzato per sviluppare una vasta gamma di tipologie di siti web, tra cui CMS, social network, siti di e-commerce e altro ancora. Supporta la generazione di contenuti in diversi formati, come HTML, JSON ed RSS. Inoltre, può essere esteso tramite l'uso di componenti esterne come Django Rest Framework e supporta nativamente diversi database come MySQL, PostgreSQL e SQLite.
- **Scalabilità:** Grazie alla sua architettura, Django consente di scalare un sito web per gestire un numero molto elevato di utenti. Un esempio notevole è Instagram, che utilizza Django per gestire centinaia di milioni di utenti attivi.
- **Manutenibilità:** Django è progettato seguendo il principio DRY (Don't Repeat Yourself), che favorisce la scrittura di codice pulito e riduce la duplicazione di logica. Questo rende il codice più mantenibile nel tempo, semplificando gli aggiornamenti e le modifiche.
- **Portabilità:** Essendo scritto in Python, Django è altamente portabile e può essere utilizzato su diversi sistemi operativi senza problemi. È compatibile con i sistemi operativi più diffusi, come Windows, MacOS e Linux.

Per ulteriori dettagli su configurazioni e funzionalità di Django, si può consultare l'[Appendice A](#) in questo documento.

2.2.1 Autenticazione

L'utilizzo dell'autenticazione built-in di Django offre un modo conveniente, sicuro e scalabile per gestire l'autenticazione degli utenti all'interno delle applicazioni web, consentendo di risparmiare tempo nella fase di sviluppo e di concentrarsi sulle funzionalità specifiche dell'applicazione.

Perciò questa implementazione di base è più che sufficiente per la maggior parte dei casi d'uso comuni, anche perché offre funzionalità complete per la registrazione, l'accesso, il logout e il recupero password, assicurando una gestione sicura dei dati sensibili degli utenti.

¹ Significa che un prodotto viene fornito con componenti predefinite e pronte all'uso, evitando al consumatore di doverle costruire da zero.

2.2.2 Infrastruttura

Django utilizza un approccio chiamato Model-View-Template-URL (**MVTU**), che presenta alcune differenze rispetto al tradizionale framework Model-View-Controller (**MVC**).

- **Model** (Modello): Nella struttura MVTU di Django, il modello rappresenta la gestione dei dati e della logica di business dell'applicazione. Il modello definisce la struttura dei dati e le interazioni con il database o altri sistemi di archiviazione dei dati. Nel framework MVC tradizionale, il modello svolge un ruolo simile, ma spesso include anche la logica di accesso ai dati.
- **View** (Vista): Nell'approccio MVTU di Django, la vista si occupa di determinare quali dati devono essere inviati all'utente, ma non si occupa della loro presentazione. La vista interagisce con il modello per ottenere i dati necessari e li prepara per la visualizzazione. Nel contesto di Django, la vista è responsabile di elaborare la richiesta HTTP e restituire una risposta HTTP appropriata. Nell'MVC tradizionale, la vista gestisce la logica di presentazione dei dati all'utente e interagisce direttamente con il modello.
- **Template**: Nell'MVTU di Django, il template è responsabile di presentare i dati come HTML, insieme a eventuali stili CSS, script JavaScript e risorse statiche. Il template contiene il markup HTML e utilizza il linguaggio di template di Django (DTL) per generare dinamicamente l'output HTML in base ai dati forniti dalla vista. Nel contesto MVC, il template corrisponde alla vista, che si occupa della presentazione dei dati all'utente.
- **URL Configuration** (Configurazione degli URL): Questa è una componente specifica di Django che non è presente nel tradizionale approccio MVC. La configurazione degli URL definisce le corrispondenze tra gli URL richiesti dagli utenti e le viste che devono gestire tali richieste. In altre parole, l'URL Configuration è responsabile di instradare le richieste degli utenti verso le viste appropriate. Questa componente aiuta a mantenere l'applicazione organizzata e a gestire la navigazione tra le diverse sezioni dell'applicazione.

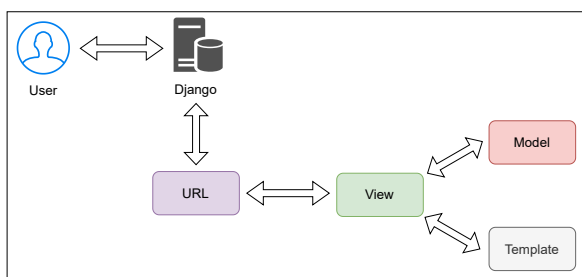


Figura 2 - MVTU

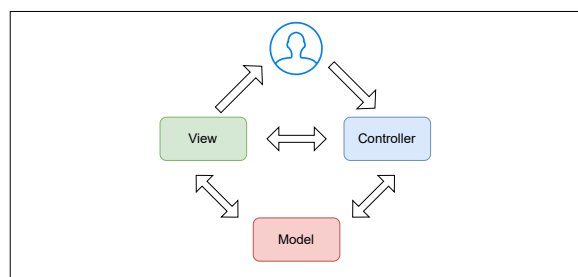


Figura 3 - MVC

Se esaminiamo la struttura del progetto seguente, possiamo identificare le diverse componenti dell'approccio MVTU. Inoltre, è possibile distinguere con chiarezza le parti relative al frontend e al backend all'interno di un'applicazione Django.

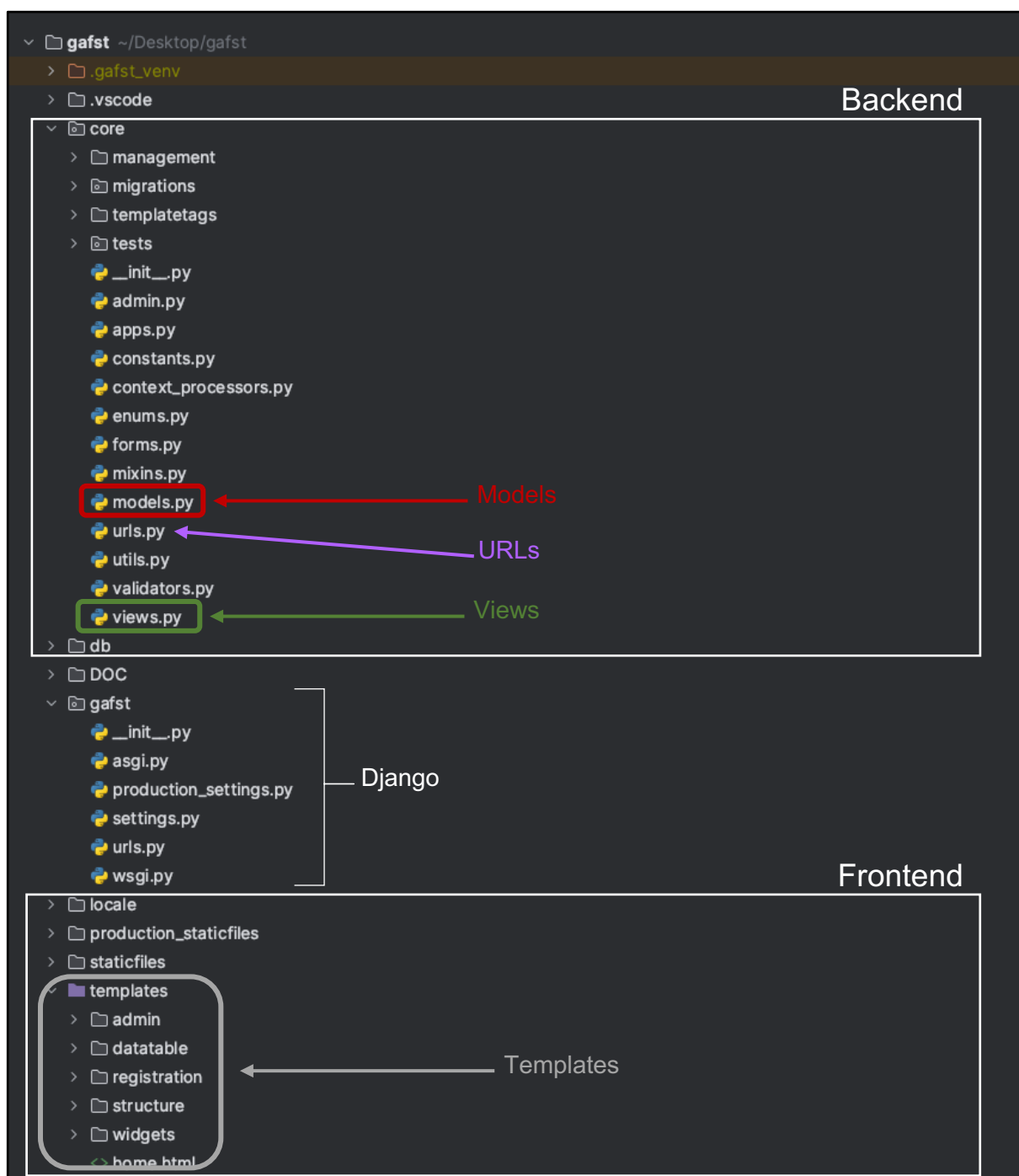


Figura 4 - Struttura Progetto Django

2.2.3 Ciclo di richiesta e risposta HTTP

Il flusso di richieste e risposte HTTP in Django coinvolge l'individuazione di un URL corrispondente, l'elaborazione dei dati tramite la vista e il modello, la generazione dell'output HTML utilizzando i template e, infine, l'invio della risposta HTTP al browser dell'utente per la visualizzazione della pagina web.

Quando si digita un URL, come ad esempio "https://djangoproject.com", nel nostro progetto Django, la prima cosa che avviene è l'attivazione del server di sviluppo Django chiamato **runserver**. Questo server aiuta Django a cercare un pattern di URL corrispondente all'interno del file **urls.py**.

Il file **urls.py** contiene una serie di pattern di URL che definiscono le possibili route che l'applicazione può gestire. Ogni pattern di URL è collegato a una singola vista definita nel file **views.py** dell'applicazione.

Una vista è una funzione o una classe che rappresenta la logica di elaborazione di una determinata richiesta HTTP. La vista accede al modello (contenuto nel file **models.py**) per ottenere i dati necessari. Il modello definisce la struttura dei dati e gestisce la logica di business dell'applicazione.

Una volta che la vista ha ottenuto i dati dal modello, può utilizzare un template HTML per combinare i dati con la presentazione. I template sono file con estensione ".html" che contengono il markup HTML insieme a tag e filtri specifici di Django. Questi template definiscono come i dati devono essere presentati all'utente.

Infine, la vista restituisce una risposta HTTP al server. Questa risposta può contenere l'output generato dal template, come HTML finale, CSS, JavaScript e altri asset statici. Il server invia quindi la risposta al browser dell'utente, che la visualizza come una pagina web.

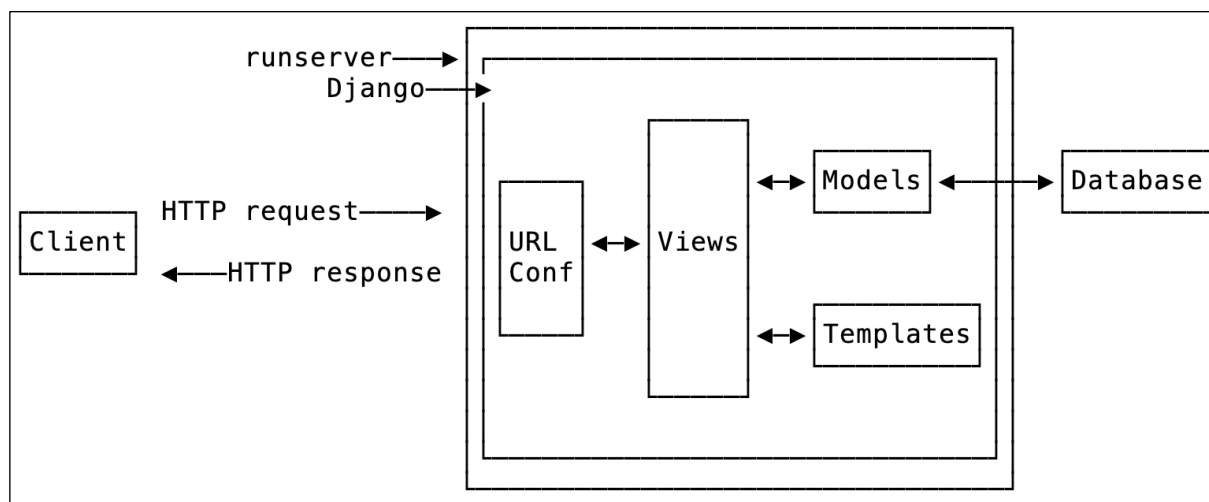


Figura 5 - Flusso di richieste e risposte Django

2.2.4 Suddivisione in Applicazione Distinte

Django è un framework web che favorisce la modularità e la riutilizzabilità del codice attraverso la suddivisione del progetto in applicazioni distinte. Ogni applicazione Django rappresenta una parte specifica del progetto e contiene i componenti necessari per gestire una determinata funzionalità.

Per aggiungere un'applicazione al progetto Django, è necessario aggiungerla alla lista **INSTALLED_APPS** nel file **settings.py** del progetto. Una volta aggiunta, l'applicazione sarà disponibile per essere utilizzata nel progetto.

La suddivisione in applicazioni consente un'organizzazione modulare, migliorando la manutenibilità e la scalabilità del codice.

2.2.5 Le Diverse View

Nel contesto di Django, le **FBV** (Function-Based Views), le **CBV** (Class-Based Views) e le **GCBV** (Generic Class-Based Views) sono tre approcci distinti per definire le viste delle applicazioni.

- **FBV** : sono le viste definite come funzioni Python. Questo approccio è stato utilizzato nella maggior parte delle versioni precedenti di Django ed è ancora ampiamente utilizzato. Nelle FBV, una funzione Python viene definita come una vista che accetta una richiesta HTTP come parametro e restituisce una risposta HTTP. All'interno di una FBV, è possibile accedere alla richiesta, elaborare i dati, accedere al modello e generare una risposta personalizzata. Le FBV offrono un controllo completo sulla logica della vista e possono essere scritte in modo semplice e diretto.
- **CBV** : sono le viste definite come classi Python. Questo approccio è stato introdotto da Django per fornire una maggiore modularità e una migliore separazione delle responsabilità. Le CBV sono più flessibili e potenti rispetto alle FBV. Le classi delle viste possono ereditare da classi di base predefinite fornite da Django, come `View` o altre classi specializzate. Le classi delle viste definiscono metodi specifici per gestire diversi tipi di richieste HTTP, come GET, POST, PUT, DELETE, ecc. Questi metodi consentono di implementare la logica della vista in modo più strutturato e leggibile. Le CBV offrono funzionalità aggiuntive, come **mixins**, per riutilizzare la logica comune tra le diverse viste.
- **GCBV** : sono una sottocategoria delle CBV e offrono un insieme di viste di base predefinite fornite da Django. Le GCBV semplificano ulteriormente lo sviluppo riducendo la quantità di codice ripetitivo. Le GCBV sono progettate per compiti comuni come visualizzare un elenco di oggetti, visualizzare i dettagli di un oggetto, creare, aggiornare o eliminare un oggetto. Forniscono funzionalità integrate per gestire queste operazioni senza dover scrivere manualmente tutta la logica della vista. È possibile personalizzare le GCBV tramite l'override di metodi specifici per adattarle alle esigenze specifiche dell'applicazione.

In generale, le **CBV** e le **GCBV** offrono un approccio più strutturato e modulare rispetto alle **FBV**. Le **CBV** consentono di scrivere codice più leggibile e riutilizzabile grazie all'utilizzo delle classi e dei metodi specifici, mentre le **GCBV** offrono viste predefinite per compiti comuni che richiedono meno codice personalizzato. Tuttavia, le **FBV** offrono ancora un controllo più diretto sulla logica della vista e possono essere più adatte per casi di utilizzo specifici o situazioni in cui si desidera massima flessibilità.

Perciò è sempre meglio utilizzare le **CBV** rispetto alle **FBV**, se non è strettamente necessario o ritenuto consono. Così da favorire uno sviluppo agile e veloce di un progetto e quindi sviluppare in modo modulare e flessibile evitando tanti refactor.

2.2.6 Dispatch by Method

Il Dispatch by Method (Dispatch per Metodo) si riferisce a una tecnica utilizzata nelle classi delle viste di Django (CBV) per instradare le richieste HTTP ai metodi appropriati in base al tipo di richiesta ricevuta (GET, POST, PUT, DELETE, ecc.). Invece di avere un unico metodo **dispatch()** che gestisce tutte le richieste, le classi delle viste utilizzano metodi separati come **get()**, **post()**, **put()**, **delete()**, ecc., che vengono richiamati automaticamente in base al metodo HTTP della richiesta. Ad esempio, una richiesta GET verrà gestita dal metodo **get()**, una richiesta POST dal metodo **post()**, e così via.

Questa tecnica consente di organizzare e gestire in modo più strutturato il codice delle viste, separando la logica di gestione delle richieste in metodi specifici per ciascun tipo di richiesta HTTP. Semplifica anche la gestione di casi d'uso diversi in base all'azione richiesta, migliorando la chiarezza e la manutenibilità del codice.

2.2.7 Controllo di base

Il controllo di base o anche comunemente chiamato *Simple Sanity Checking*, si riferisce a un approccio comune utilizzato nelle viste di Django per effettuare controlli di base sulle richieste prima di procedere con l'elaborazione. Questi controlli di base includono la verifica dell'autenticazione dell'utente, l'autorizzazione per accedere a una determinata risorsa o eseguire un'azione, la validazione dei dati inviati nella richiesta e altri controlli di sicurezza.

2.2.8 Object Relational Mapper

Django utilizza un ORM (Object-Relational Mapper) integrato e potente per la gestione delle operazioni con il database. L'ORM di Django permette agli sviluppatori di interagire con il database utilizzando oggetti Python, astratti dai dettagli specifici del database sottostante. Ciò significa che gli sviluppatori possono scrivere codice Python nel file `models.py`, che verrà automaticamente tradotto nel corrispondente SQL corretto per ciascun database supportato.

Django fornisce supporto integrato per diversi back-end di database, tra cui PostgreSQL, MySQL, MariaDB, Oracle e SQLite. Questo significa che gli sviluppatori possono utilizzare lo stesso codice Python per le operazioni con il database, indipendentemente dal back-end specifico utilizzato. La configurazione del database richiede solo l'aggiornamento della sezione **DATABASES** nel file **settings.py** del progetto Django.

Durante lo sviluppo locale, Django utilizza di default SQLite come database. SQLite è un database basato su file e, di conseguenza, è più semplice da utilizzare rispetto alle altre opzioni che richiedono un server dedicato che viene eseguito separatamente da Django stesso.

L'utilizzo dell'ORM di Django offre numerosi vantaggi, tra cui:

- **Astrazione del database:** Gli sviluppatori possono interagire con il database utilizzando oggetti Python familiari, senza dover scrivere direttamente il codice SQL.
- **Portabilità:** L'ORM di Django supporta diversi back-end di database, consentendo l'esecuzione dell'applicazione su diversi database senza modificare il codice.
- **Sicurezza:** L'ORM di Django implementa misure di sicurezza per prevenire attacchi come SQL injection.
- **Facilità di sviluppo:** L'ORM semplifica la creazione, la modifica e la gestione dello schema del database attraverso il codice Python.

2.2.9 Sistema Autenticazione Django

L'autenticazione degli utenti in Django è un sistema integrato che gestisce vari aspetti, quali account utente, gruppi, permessi e sessioni utente basate su cookie. Esso fornisce sia funzionalità di autenticazione, che verifica l'identità dell'utente, sia di autorizzazione, che determina le azioni permesse a un utente autenticato.

Sistema di autenticazione crea alcune tabelle nel database, tra cui **auth_user** per gli utenti e **auth_group** per i gruppi. Altre tabelle come **auth_permission** e **auth_user_groups** vengono utilizzate per gestire i permessi e le relazioni tra utenti e gruppi.

Django non fornisce alcune funzionalità comuni ad altri framework come la verifica della forza della password, il throttling dei tentativi di accesso o l'autenticazione tramite terze parti (es. OAuth). Tuttavia, è possibile estendere queste funzionalità attraverso pacchetti di terze parti.

2.2.10 Testing

Nel contesto di Django, testare il codice è fondamentale per individuare rapidamente errori e confermare che funzioni correttamente. Il testing automatizzato è un'arma chiave per gli sviluppatori web. Si può utilizzare una test suite per garantire che nuove implementazioni o modifiche al codice esistente non influenzino inaspettatamente l'applicazione.

Django offre un framework di esecuzione dei test e strumenti per simulare richieste e verificare l'output dell'applicazione. Il metodo consigliato per scrivere test è utilizzare il modulo **unittest** di Python, ma Django supporta anche altri framework di test attraverso specifiche API e strumenti.

2.2.11 Test con Unittest

In Django, i test unitari utilizzano il modulo unittest della libreria standard di Python, definendo i test attraverso un approccio basato su classi.

Quando si eseguono i test, l'utility predefinita trova tutti i casi di test in file il cui nome inizia con **"test"**, costruendo e eseguendo automaticamente una suite di test.

Sebbene il template startapp crei un file **tests.py**, è consigliabile strutturarlo in un pacchetto di test suddiviso in sotto moduli man mano che la suite di test cresce (es: **test_models.py**, **test_views.py**, ...).

Se i test interagiscono con il database, è preferibile usare **django.test.TestCase** invece di **unittest.TestCase** per evitare comportamenti variabili e potenziali fallimenti a seconda dell'ordine di esecuzione.

2.2.12 La Classe TestCase

La classe *TestCase* di Django è un'estensione della classe base *unittest.TestCase* di *Python*, progettata specificamente per i test all'interno di progetti basati su Django. Essa permette di creare test unitari, di integrazione e funzionali all'interno dell'ambiente Django in modo efficace.

Convertendo una classe di test normale basata su *unittest.TestCase* in una sottoclasse di *TestCase* di Django, si ottiene accesso a tutte le funzionalità standard di Python per i test unitari, insieme a ulteriori miglioramenti e funzionalità specifiche fornite da Django. Questo approccio semplifica la scrittura e l'esecuzione dei test nell'ambito di un progetto Django.

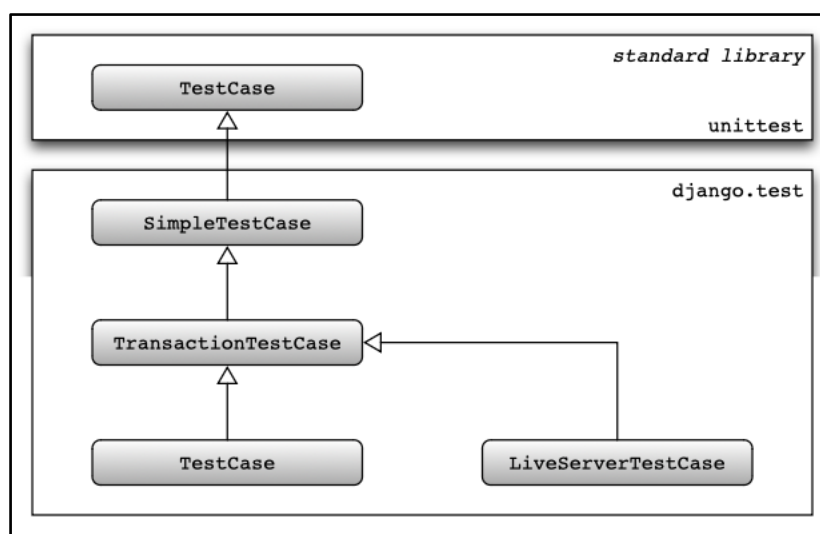


Figura 6 - Gerarchia classe django.test.TestCase

2.3 Costruzione Pagine Responsive

Se si desidera utilizzare Django senza l'adozione di un framework frontend, è necessario esplorare le alternative compatibili, tenendo presente la necessità di una soluzione che non aumenti ulteriormente la complessità del sistema. Dopo aver effettuato una ricerca approfondita, le migliori opzioni identificate sono:

- Acquisire un template preesistente di un sito web e adattarlo per l'uso con Django.
- Sviluppare manualmente un template HTML/CSS sfruttando i tag template e le funzionalità di Django. Questo approccio offre maggiore controllo e flessibilità, ma comporta tempi di sviluppo più lunghi.
- Creare componenti riutilizzabili nel progetto Django utilizzando i tag template di Django. Questa strategia, simile a quella impiegata in certi framework frontend, facilita la separazione tra logica e presentazione, rendendo più semplice sia la manutenzione che la creazione di nuove pagine. L'adozione di librerie come Bootstrap può ulteriormente agevolare la costruzione di componenti e fornire elementi predefiniti.

2.4 HTMX e Single-page applications

HTMX è una libreria JavaScript che permette di accedere a funzionalità avanzate come AJAX, transizioni CSS, WebSockets e Server Sent Events direttamente nell'HTML con l'utilizzo di attributi HTML.

Può essere utilizzato per creare Single-Page Applications (SPA) grazie alla sua capacità di effettuare richieste HTTP senza necessità di ricaricare l'intera pagina. Essa va oltre i metodi HTTP GET e POST standard e gli eventi come "click" e "submit", permettendo di creare molte interazioni in modo semplice e flessibile.

Tali funzionalità risultano estremamente vantaggiose, in quanto possono essere utilizzate per aggiornare in modo semplice sia componenti grafici personalizzati che elementi forniti da librerie come Bootstrap. Ciò consente di adottare una metodologia simile a quella utilizzata in alcuni framework frontend, favorendo una più efficace separazione tra le diverse componenti visive del progetto.

2.5 Risultati ricerca

Nel corso dell'analisi si sono esplorate in profondità le diverse funzionalità offerte da Django e la sua architettura sottostante, identificando anche le librerie più adatte da aggiungere. Questo ha permesso di acquisire una comprensione chiara su come raggiungere gli obiettivi stabiliti dal progetto.

In conclusione, ciò ha guidato la selezione delle tecnologie più adatte da integrare nel sistema Django, al fine di sviluppare un'applicazione all'avanguardia che soddisfi pienamente le esigenze del committente.

2.5.1 Tecnologie adottate per il Backend

Per le funzioni di base dell'applicazione web, è stata adottata la standard library di Django. Al fine di migliorare l'esperienza utente, sono stati integrati diversi pacchetti aggiuntivi. Tra questi, *FontAwesome* per l'implementazione di icone aggiuntive, *Widget-Tweaks* per la personalizzazione dei form di inserimento dati, e *Rosetta* per una gestione più flessibile delle traduzioni direttamente dall'interfaccia web.

Inoltre, è stata integrata la libreria *pycountry* per fornire un elenco predefinito di nazionalità. Questo facilita la scelta dell'utente e minimizza il rischio di errori nell'inserimento dei dati.

Scelta Pacchetti Python	
Nome	Scopo
rosetta	Facilitare la traduzione
fontawesomefree	Aggiunta di icone espressive
widget_tweaks	Creazione form personalizzati, utilizzando CBV
pycountry	Ottenere la lista degli acronimi delle nazionalità

2.5.2 Tecnologie adottate per il Frontend

Per la parte di costruzione delle interfacce utente è stato scelto di utilizzare i componenti *Bootstrap* incluse le relative icone. Invece Per la creazione di tabelle dinamiche, la scelta è ricaduta sulla libreria *Datatable*, che offre una vasta gamma di funzionalità già implementate e utilizzate da un'ampia community.

Per la gestione delle interazioni utente, si è optato per l'utilizzo di HTMX. Questa libreria facilita l'integrazione di AJAX direttamente nell'HTML, permettendo aggiornamenti grafici locali senza dover ricaricare l'intera pagina.

Librerie Web	
Nome	Scopo
Bootstrap	Facilitare la traduzione
Bootstrap Icons	Icone stile bootstrap
HTMX	Creazione interfaccia e Web App stile Single Application
Fontawesomefree	Icone Aggiuntive non presenti in Bootstrap Icons
Datatables	Libreria per la creazione di Datatable

3 Soluzione

L'architettura scelta per il progetto si fonda su Django, in conformità con i vincoli imposti. Durante il ciclo di sviluppo agile dell'applicazione web, si è cercato di ottimizzare l'architettura, imparando e adattandosi nel corso del tempo.

Poiché il committente ha espresso la preferenza per un approccio senza l'uso di *framework JavaScript* come React.js, si è andati alla ricerca di alternative valide. In questa fase, si è scoperto HTMX, una libreria che ha facilitato la creazione di pagine web interattive mediante l'uso di AJAX e CSS Transitions. Questa scelta ha reso possibile la modifica dinamica del contenuto senza la necessità di un refresh completo della pagina, contribuendo così a un'esperienza utente fluida e efficiente.

In sintesi, l'architettura basata su Django, unita all'uso di HTMX, AJAX e HTMX, offre una soluzione equilibrata che soddisfa i vincoli e le esigenze del committente. L'approccio agile adottato ha permesso un miglioramento costante dell'architettura, assicurandosi che il sistema si flessibile e sicuro.

In termini di tecnologie, il progetto utilizza una combinazione di soluzioni all'avanguardia supportate da un' ampia *community* per massimizzare efficienza, stabilità e sicurezza.

MySQL è scelto come database per la sua robustezza e performance elevate ma anche perché è stato richiesto dal committente, ed è ideale per applicazioni che richiedono accesso ai dati in tempo reale.

PythonAnywhere viene utilizzato per l'*hosting*, offrendo un ambiente cloud ottimizzato per *Python*, facilitando così la distribuzione e la manutenzione.

Django serve come framework sia per il backend che per il frontend, noto per la sua architettura simile a MVC e funzionalità integrate come l'autenticazione, rendendo più veloce lo sviluppo.

Le tecnologie frontend come HTML, JS, CSS e HTMX sono utilizzate per creare un'interfaccia utente reattiva e interattiva. Infine, il *framework* di Unit Test di Django, basato su Unittest, garantisce che il sistema sia affidabile e mantenibile.

La scelta di queste tecnologie è motivata dal loro ampio utilizzo e dalla loro dimostrata efficacia durante lo sviluppo nell'assicurare la sicurezza e l'efficienza del sistema.

Servizio	Tecnologia
Database	MySQL
Host	pythonanywhere
Backend Framework	Django
Frontend Framework	Django
Frontend	HTML, JS, CSS, Django template language, HTMX
Test	Unit Test Django basato sulla libreria Unittest

3.1 Pianificazione

La pianificazione per la costruzione della Web App per la gestione della federazione e dei club svizzeri di Twirling è stata sviluppata seguendo un ordine di priorità. Si tiene conto delle attività di test automatici e nonché delle correzioni necessarie durante il percorso di sviluppo.

La tempistica prevista è stata compressa al fine di disporre di un margine di tempo supplementare per affrontare eventuali imprevisti, apportare correzioni necessarie e magari permettere delle pause, come le vacanze.

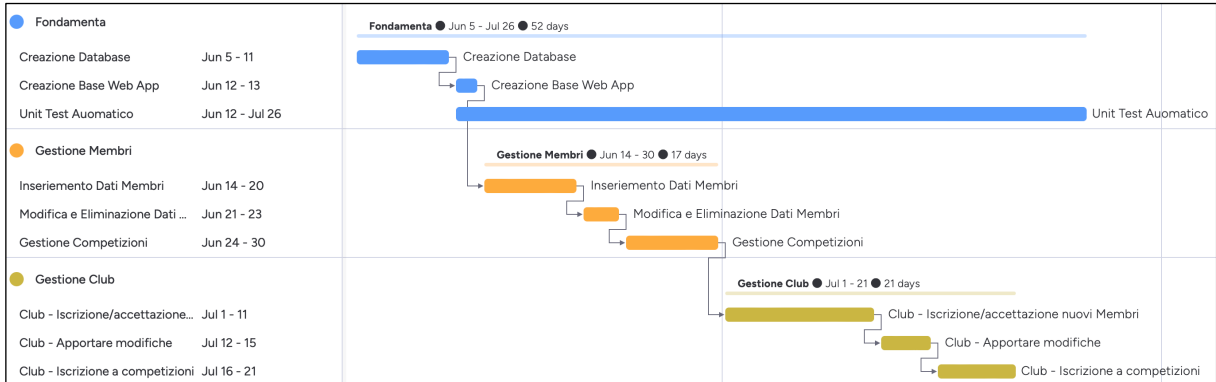


Figura 7 – Gantt Pianificazione Iniziale

3.2 Database

Quando un membro viene trasferito da un club a un altro, si inserisce una nuova riga nella tabella **Membership** per registrare il trasferimento e si aggiunge la **transfer_date** nel record di **Membership** che indica l'appartenenza del membro al vecchio club. Ciò consente di mantenere una cronologia dei trasferimenti e tenere traccia del club corrente a cui il membro appartiene.

Per effettuare il *discharge* di un club, si imposta la **discharge_date** per indicare la data di sospensione o rimozione del club. Inoltre, si può utilizzare la colonna **possible_resume_date** per segnalare una possibile data futura di ripresa delle attività del club. Queste informazioni permettono di tenere traccia dello stato attuale del club e di pianificare eventuali azioni future di ripristino.

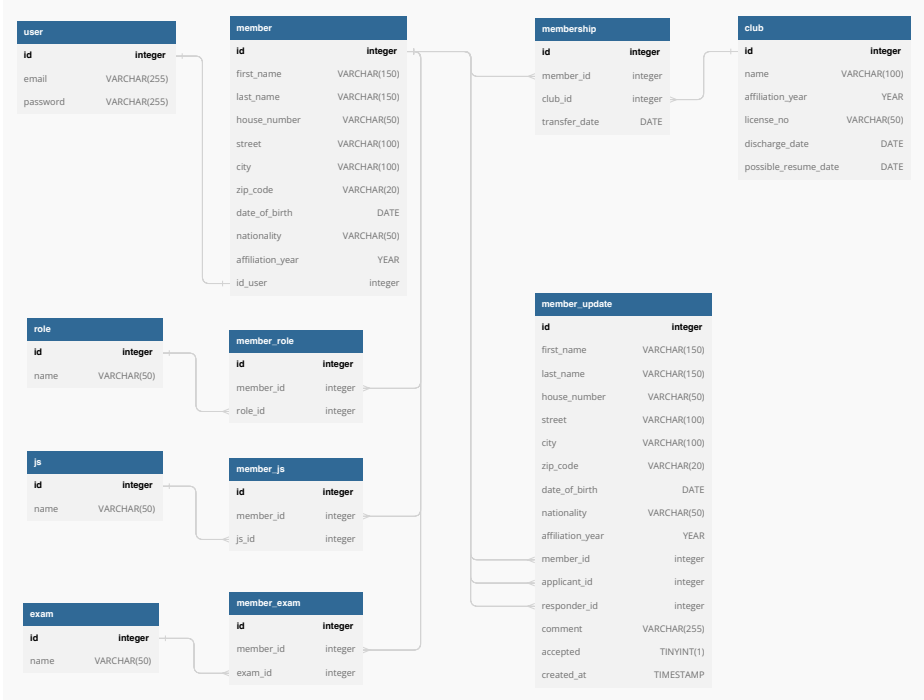


Figura 8 - Progettazione iniziale database

4 Implementazione

In questo capitolo viene illustrato il processo di implementazione del progetto, analizzando in modo dettagliato i componenti utilizzati sia nel backend che nel frontend. Si descrivono le strutture architettoniche adottate e le librerie utilizzate, evidenziando le motivazioni dietro le scelte fatte.

4.1 Scripts

Durante la fase di implementazione, si sono sviluppati *script* automatizzati per eseguire operazioni ripetitive, al fine di ottimizzare il ciclo di sviluppo. Questi *script* sono stati progettati anche per facilitare eventuali aggiornamenti alla piattaforma web, permettendo a committenti o sviluppatori di apportare modifiche in modo più efficiente, senza la necessità di una comprensione approfondita della struttura sottostante.

Gli script sono stati sviluppati sfruttando la funzionalità "*Commands*" di Django, che permette di creare pezzi di codice eseguibili mediante un comando specifico, il quale corrisponde al nome del file in cui è contenuto il codice. All'interno di ciascuno di questi comandi è presente il metodo ***handle***, il quale rappresenta la funzione eseguita al momento della chiamata del comando. Tale metodo è ereditato dalla classe ***BaseCommand***.

Per eseguire gli script, è necessario accedere all'ambiente virtuale Python e posizionarsi nella directory *root* del progetto Django (identificata come 'gafst'). Questo garantisce un accesso completo alle funzionalità richieste per l'operazione.

Tutti gli script sono ubicati nella directory "*management/commands*", che fa parte della struttura standard di Django. Questa directory si trova all'interno della seguente gerarchia:

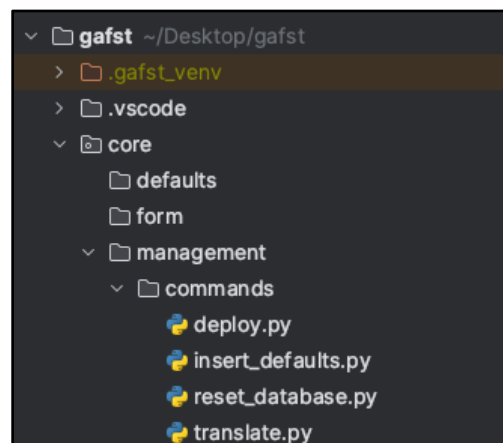


Figura 9 - Gerarchia Cartelle contenenti i comandi/scripts

4.1.1 Script per la Traduzione

Lo script *translate.py* è stato progettato per automatizzare il processo di traduzione delle stringhe testuali presenti nel codice sorgente. Esso effettua una scansione di tutti i file all'interno del progetto e aggiorna i file di traduzione con estensione **.po**. Successivamente, compila tali file generando file **.mo**, rendendo in tal modo le traduzioni prontamente accessibili alla piattaforma web.

Per eseguire il comando basta eseguire questo codice all'interno di una shell configurata correttamente:

```
python manage.py translate
```

4.1.2 Script per l'inserimento dei dati di default del database

Basandosi sulle enumerazioni (*enum*) create in Django, lo script *insert_defaults.py* automatizza l'inserimento dei dati predefiniti nel database. Questo strumento non solo semplifica il ripristino del database, ma facilita anche la generazione di test che coinvolgono i dati predefiniti memorizzati nel database stesso.

Lo script non fa altro che prendere i valori degli enum e metterli all'interno delle rispettive tabelle del database, con questa semplice funzione:

```
def insert_defaults_by_enum(model, enum):  
    for e in enum:  
        model.objects.get_or_create(name=e.value)
```

In aggiunta, durante l'inserimento dei gruppi predefiniti per l'autenticazione Django, lo script assegna automaticamente i relativi permessi a entrambi i gruppi.

```
fstb_admin_group = Group.objects.create(name=GroupEnum.FSTB_ADMIN.value)  
fstb_admin_group.permissions.set([fstb_admin_permissions])  
fstb_admin_group.permissions.set([club_admin_permissions])  
  
club_admin_group = Group.objects.create(name=GroupEnum.CLUB_ADMIN.value)  
club_admin_group.permissions.set([club_admin_permissions])
```

Per eseguire il comando basta eseguire questo codice all'interno di una shell configurata correttamente:

```
python manage.py insert_defaults
```

4.1.3 Script per il reset del database

Lo script denominato *reset_database.py* è progettato per semplificare e accelerare il processo di reimpostazione del database. Esso elimina tutti i dati attualmente memorizzati, ricostruisce il database basandosi sull'insieme delle migrazioni effettuate durante la fase di sviluppo e infine lo ripopola con i valori di default tramite il comando *'insert_defaults'*.

Per reimpostare il database, lo script effettua il 'drop' del database esistente e lo ricrea utilizzando specifici comandi SQL. Per fare questo, si connette al server MySQL attraverso le configurazioni del database specificate nel file 'settings.py'.

```
connection = connections["default"]
db_name = connection.settings_dict["NAME"]

with connection.cursor() as cursor:
    cursor.execute(f"DROP DATABASE IF EXISTS {db_name}")
    cursor.execute(f"CREATE DATABASE {db_name}")
```

Per eseguire il comando basta eseguire questo codice all'interno di una shell configurata correttamente:

```
python manage.py reset_database
```

4.1.4 Script per il deploy su PythonAnywhere

Lo script *deploy.py* è stato sviluppato per semplificare il processo di deployment su PythonAnywhere, riducendo così il rischio di errori nella sequenza di operazioni di deployment. Questo ha notevolmente velocizzato il deployment, consentendo una visualizzazione immediata delle modifiche apportate al sito web e minimizzando la probabilità di errori.

Per effettuare il deployment, lo script esegue una sequenza di operazioni programmata. Durante questo processo, l'utente ha la possibilità di confermare o declinare alcune azioni specifiche, come il reset del database. Ogni operazione è gestita da una funzione dedicata all'interno del comando.

```
def handle(self, *args, **options):
    self.stdout.write(
        self.style.WARNING(
            "Deploying the application...\n"
            + "- remember to run this command in the right virtual environment"
            + "\n- remember to fetch and pull the latest changes from git ..."
            + "\n- remember that this will modify the settings.py file"
            + "\n\nPRESS ENTER TO CONTINUE"
        )
    )

    input()

    try:
        self.install_requirements()
        self.collect_static_files()
        self.setup_production_settings()
        self.setup_db()
        self.create_superuser()

    except Exception as e:
        raise CommandError(str(e))
```

```
def setup_db(self):
    reset_database = input(
        self.style.WARNING("Do you want to reset the database? (y/n) ")
    )
    if reset_database.lower() == "y":
        run_command("python manage.py reset_database")
```

Per eseguire il comando basta eseguire questo codice all'interno di una shell configurata correttamente:

```
python manage.py deploy
```

4.2 Struttura Base

La struttura di base del sito web prodotto è messa in piedi utilizzando Django, un unico framework backend. Il frontend viene implementato utilizzando esclusivamente i Template e View di Django, con l'ausilio di Bootstrap e librerie aggiuntive come HTMX, per costruire template responsive.

4.3 Evitare il CSRF

Questo codice o template tag viene utilizzato per garantire la sicurezza del sito web e prevenire attacchi CSRF (Cross-Site Request Forgery).

```
{% csrf_token %}
```

Quando il tag "{% csrf_token %}" non viene utilizzato o il token CSRF manca o è invalido, Django risponderà tipicamente con un errore **HTTP 403 Forbidden**. Questo errore indica che il server ha compreso la richiesta ma si rifiuta di autorizzarla a causa dell'attivazione del meccanismo di protezione CSRF.

4.4 Traduzione di testo

Django offre un metodo per la traduzione di testi in modo semplice e veloce. Con una breve configurazione, è possibile ottenere traduzioni utilizzando le funzioni standard di Django. Per fare ciò, si fa uso del modulo di **internazionalizzazione** di Django e, se necessario, anche del sistema di **localizzazione**.

Questa funzionalità permette di tradurre le stringhe di testo presenti nel codice in diverse lingue, senza dover modificare il codice stesso. Per ulteriori dettagli su come la traduzione di testo è configurata e opera in Django, si può consultare l'[Appendice B](#) in questo documento.

4.4.1 Traduzioni con Rosetta

Per facilitare il processo di traduzione, è stato scelto di installare il pacchetto *Python Rosetta*. Questa libreria offre un metodo semplice per modificare le traduzioni, attraverso un'interfaccia basata sull'implementazione del pannello di amministrazione fornito da Django. In questo modo, se il committente identifica traduzioni inadeguate o incorrette, ha la possibilità di segnalarle o modificarle autonomamente.

Se Rosetta viene impiegato in un ambiente di produzione, è fondamentale essere consapevoli di alcune possibili problematiche. Ad esempio, nell'uso specifico con PythonAnywhere, è necessario effettuare un reload dell'applicazione web per rendere effettive le nuove traduzioni. Tale passaggio è cruciale poiché Rosetta compila i file di traduzione aggiornati, che restano inattivi fino al completamento del reload. Tuttavia, va notato che questa esigenza potrebbe non essere universale per tutti gli ambienti di produzione.

Per ulteriori dettagli su come la traduzione tramite rosetta è configurata e opera in Django, si può consultare l'[Appendice B.1](#) in questo documento.

4.4.2 Traduzione di Testo all'interno di files Java Script

Django risolve le problematiche di traduzione nel codice *JavaScript* fornendo la *View JavaScriptCatalog*. Questa genera una libreria *JavaScript* che replica le funzionalità dell'interfaccia *gettext* e include un *array* di stringhe di traduzione. Di default, il dominio di traduzione utilizzato è **djangojs**, che contiene le stringhe da includere nell'output della vista.

Per ulteriori dettagli su come la traduzione all'interno di file Java Script è configurata e opera in Django, si può consultare l'[Appendice B.2](#) in questo documento.

4.4.3 Traduzioni Datatable

Per tradurre le stringhe di default della libreria Datatable, è stato necessario aggiungere l'opzione di inizializzazione **language** specificando tutte le stringhe che si desidera tradurre. È possibile trovare la lista completa di queste stringhe nella documentazione online della libreria. In questo progetto specifico, l'inizializzazione è stata effettuata nel file **main.js**:

```
language: {  
  "decimal":      gettext("."),  
  "emptyTable":   gettext("No data available in table"),  
  "info":         gettext("Showing _START_ to _END_ of _TOTAL_ entries"),  
  ...  
},
```

4.5 Verifica dell'identità dell'utente autenticato

Django offre un sistema di autenticazione robusto e altamente personalizzabile. Questa flessibilità ha facilitato la creazione di due categorie di utenti: FTSB Admin e Club Admin. Per ognuna di queste, sono stati creati gruppi di autenticazione con permessi specifici.

Tale configurazione consente un controllo agevole sull'identità dell'utente all'interno dell'applicazione, permettendo di adeguare sia la logica applicativa che la presentazione in base al tipo di utente autenticato.

Per ulteriori dettagli su come viene effettuata la verifica dell'identità e come le categorie di utenti sono configurate e sono utilizzate in Django, si può consultare l'[Appendice C](#) in questo documento.

4.6 Relazioni Modelli Django

In Django, la presenza di vari tipi di relazioni tra i modelli facilita notevolmente la strutturazione del database, specialmente quando si tratta di rappresentare connessioni complesse tra entità. Questi tipi di relazioni sono stati implementati in vari modelli all'interno del progetto.

Queste sono le tipologie di relazioni utilizzate all'interno dei *Model*:

- **ForeignKey**: Stabilisce una relazione di tipo “molti-a-uno” e permette a un modello di riferirsi a un'istanza di un altro modello. Questo tipo di relazione è vantaggioso quando un'entità necessita di formare più legami con un'altra entità e fornisce la capacità di accedere facilmente all'elenco di oggetti correlati.
- **OneToOneField**: Configura una relazione “uno-a-uno”. Sebbene simile al ForeignKey, assicura che ogni modello nella relazione abbia un legame univoco con l'altro. Facilita l'accesso bidirezionale all'oggetto associato.
- **ManyToManyField**: Implementa una relazione “molti-a-molti”, ideale quando diverse entità necessitano di essere associate in maniera bidirezionale. Questo campo offre la flessibilità di creare tabelle ponte separate. Tale funzionalità è stata inizialmente sperimentata, ma successivamente rimossa poiché si è rivelata superflua.

4.7 Creazione di permessi personalizzati

Al fine di facilitare la gestione di permessi personalizzati non associati a specifici modelli, è stata sviluppata la classe *PermissionsSupport*. In quanto modello non gestito in Django (*managed = False*), questa classe non conduce alla creazione di una tabella correlata nel *database*, eludendo così eventuali operazioni di creazione o rimozione di tabelle. La classe inoltre annulla la creazione dei permessi *CRUD* standard di Django, quali “aggiungi”, “modifica”, “elimina” e “visualizza”.

La *PermissionsSupport* introduce due categorie di permessi personalizzati, denominati *fstb_admin_permissions* e *club_admin_permissions*. Tali permessi sono associati a gruppi utente, facilitando così l'identificazione e la gestione degli utenti all'interno del sistema.

```
class PermissionsSupport(models.Model):
    """This model is used to add permissions to the User model"""

    class Meta:
        managed = False # No database table creation or deletion \
        # operations will be performed for this model.

        default_permissions = () # disable "add", "change", "delete"
        # and "view" default permissions

        permissions = (
            ("fstb_admin_permissions", "Have FSTB Admin Permissions"),
            ("club_admin_permissions", "Have Club Admin Permissions"),
        )
```

4.8 Modelli Astratti ed Ereditarietà

I modelli Django sono intuitivi da configurare, velocizzando notevolmente la costruzione del database. Offrono la possibilità di creare classi astratte, che possono poi essere ereditate da altri modelli, evitando duplicazioni e rendendo il codice più pulito.

Questa ereditarietà ha facilitato la creazione di vari modelli con campi o funzionalità comuni. Inoltre, le proprietà e metodi ereditati possono essere modificati o estesi, incrementando la flessibilità del codice.

Queste caratteristiche sono state sfruttate per sviluppare tabelle progettate per gestire richieste di aggiornamento da parte dei *Club Admin*.

```
class BaseMember(models.Model):
    ...

class Member(BaseMember):
    pass

class ChangeModel(models.Model):
    ...

class MemberChange(BaseMember, ChangeModel):

    member = models.ForeignKey(
        Member,
        on_delete=models.CASCADE,
        related_name="member_changes",
        null=True,
        blank=True,
    )

    pass
```

4.9 Fornire Variabili Globali a tutti i Template

Per consentire a ogni Template di accedere a variabili globali, è stata implementata una soluzione basata su un ***Custom Context Processor***.

Un *Context Processor* è un componente di Django che permette di definire una serie di variabili che saranno disponibili globalmente in tutti i template. Questo meccanismo è particolarmente utile quando si ha la necessità di rendere disponibili certi dati o funzioni in ogni pagina del sito, senza doverli definire manualmente in ogni singola View. Per esempio, Django fornisce un Context Processor per l'autenticazione, che rende facilmente accessibili i dati dell'utente autenticato e i suoi permessi.

Per ulteriori dettagli su come un *Context Processor* personalizzato è configurato e opera all'interno del progetto, si può consultare l'[Appendice D](#) in questo documento.

4.10 Risoluzione Errori

Durante la fase di implementazione, sono stati individuati vari errori nel codice. Per affrontare queste sfide, sono state ricercate e applicate soluzioni appropriate. La documentazione di queste soluzioni è risultata essenziale, soprattutto per prevenire la duplicazione di sforzi nel risolvere problemi già incontrati.

Questo approccio si è rivelato particolarmente efficace, poiché l'apprendimento di meccanismi specifici in un framework come Django spesso comporta la comprensione di elementi molto simili. Pertanto, la documentazione ha giocato un ruolo fondamentale nel facilitare il processo di apprendimento durante lo sviluppo del progetto.

Per ulteriori dettagli su come sono stati risolti i problemi incontrati, si può consultare l'[Appendice E](#) in questo documento.

5 Testing

5.1 Metodologia di testing

All'interno di questo progetto basato su Django, è stata adottata una solida metodologia di *testing* per garantire l'affidabilità e il corretto funzionamento delle applicazioni. Questa metodologia è in linea con le *best practice* di testing di Django, che sfrutta appieno la sua libreria di testing integrata chiamata `unittest`.

Django, come noto, è un *framework* di sviluppo web scritto in *Python*, progettato per agevolare l'effettuazione di test in modo efficiente. La strategia di testing si fonda principalmente sulla libreria `unittest`, che offre una vasta gamma di strumenti per creare e condurre test unitari, di integrazione e funzionali.

Nell'ambito di questo progetto, la metodologia di testing si sviluppa in diversi passaggi chiave. In primo luogo, l'organizzazione dei test è gestita attraverso una struttura gerarchica basata sulle classi e sui metodi di `unittest`. I test unitari consentono di esaminare le singole componenti dell'applicazione in isolamento, verificando che ciascuna funzione o metodo agisca come previsto.

Per i test di integrazione, viene sfruttata la classe **Client** fornita da Django. Questo strumento simula le richieste HTTP verso le viste dell'applicazione, consentendo di valutare il comportamento dell'applicazione in risposta a diverse interazioni utente.

Un aspetto fondamentale della metodologia è l'utilizzo dei **Mocks**. I *Mocks* sono utilizzati per simulare parti esterne all'applicazione, come servizi o funzioni, in modo da isolare il codice in test da dipendenze esterne. Questo approccio assicura che il comportamento delle dipendenze venga controllato e prevedibile, consentendo di focalizzarsi sull'analisi delle specifiche funzionalità dell'applicazione.

5.2 Librerie Scelte per Testing

È stata adottata una strategia di testing che combina la libreria standard di testing `unittest` di Python con i moduli forniti dalla libreria `django.test` di Django.

Questo approccio permette di creare test unitari, di integrazione e funzionali in modo efficace. La libreria `unittest` fornisce strumenti per strutturare e condurre i test, mentre i moduli `django.test` permettono di testare in modo semplice modelli, viste e template.

Questo permette l'uso del client per simulare richieste *http* e la possibilità di isolare componenti esterni tramite l'uso di *mocks*. L'approccio complessivo garantisce test accurati e affidabili, adattandosi alle specifiche esigenze del framework Django.

5.3 Moking

Il *moking* rappresenta una pratica fondamentale per isolare parti specifiche del codice durante i test. Il *mocking* consente di creare oggetti simulati che replicano il comportamento di componenti esterni, come funzioni, oggetti o servizi, all'interno delle funzioni che si desidera testare. Questa tecnica è particolarmente utile per concentrarsi sui comportamenti specifici del codice in isolamento, senza dipendere da implementazioni reali o esterne.

Il seguente esempio illustra l'utilizzo del "mocking" per testare il codice compilato. Nello specifico, il codice crea un test-case **TestGetSuccessResponse** che effettua il test di una funzione **get_success_response** utilizzando il modulo di test *unittest.mock*.

Nell'esempio, la funzione *setUp* viene utilizzata per preparare l'ambiente di test. Sono create istanze di oggetti simulati (Mock) per la vista (*self.view*) e per un'istanza di modello (*self.instance*). Questi oggetti simulati vengono configurati per simulare il comportamento di oggetti reali all'interno della funzione in test. Inoltre, è preparato un formato di messaggio (*self.message_format*) che verrà utilizzato all'interno della funzione.

La funzione **test_get_success_response** esegue il test vero e proprio. Viene richiamata la funzione *get_success_response* con gli oggetti simulati precedentemente creati. Successivamente, vengono eseguite una serie di asserzioni per verificare se la risposta ottenuta dalla funzione è conforme alle aspettative.

```
class TestGetSuccessResponse(TestCase):
    def setUp(self):
        self.view = Mock(model=Mock(__name__="Member"))
        self.instance = Mock(__str__=Mock(return_value="Gina Doe"))
        self.message_format = ADDED_MESSAGE

    def test_get_success_response(self):
        expected_changed_event = "memberListChanged"
        expected_message = "Gina Doe added"

        response = get_success_response(self.view, self.instance,
self.message_format)

        self.assertIsInstance(response, HttpResponse)
        self.assertEqual(response.status_code, 204)

        response_headers = json.loads(response["HX-Trigger"])
        self.assertEqual(
            response_headers,
            {
                expected_changed_event: None,
                SHOW_MESSAGE: expected_message,
            },
        )
```

5.3.1 Decorator Patch

Il decorator **@patch** è uno strumento utilizzato nell'ambito dei test unitari in *Python* ed è contenuto nel modulo *unittest*. Esso consente di sostituire temporaneamente una variabile, una funzione o un oggetto con un suo **mock** (cioè una simulazione o imitazione) al fine di controllare il comportamento durante l'esecuzione dei test.

Per garantire il corretto funzionamento dei *mock*, è fondamentale assicurarsi di utilizzare la giusta *path* di importazione all'interno del *patch decorator*. Prendendo ad esempio il seguente metodo di test, se le funzioni **run_command** e **Popen** sono definite nello stesso file, allora il percorso di patch dovrebbe fare riferimento a quel modulo (File python).

Nel nostro contesto specifico, se le definizioni si trovano in moduli distinti, è indispensabile garantire che la patch di **Popen** punti al modulo in cui è definita **run_command**, anziché al modulo in cui è definito **Popen**. (cioè *patch* dovrebbe puntare al modulo **core.utils**, che contiene la funzione *run_command* e viene importata la funzione *Popen*).

```
@patch("core.utils.Popen")
def test_run_command(self, MockPopen):
    process_mock = Mock()
    process_mock.communicate.return_value = (b"output", b"")
    process_mock.returncode = 0
    MockPopen.return_value = process_mock

    run_command("echo Hello")

    # Assert that Popen was called
    MockPopen.assert_called_once()

    # Asserts
    process_mock.communicate.assert_called_with(
        b"input data that is passed to subprocess' stdin"
    )
    MockPopen.assert_called_with(
        "echo Hello", shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE
    )
```

6 Deployment

Per effettuare con successo il *deployment* dell'applicazione web sviluppata utilizzando il framework Django, è stato fondamentale apportare modifiche alle configurazioni generali di Django ed eseguire specifici comandi. Questi passaggi ottimizzano il framework Django per la fase di produzione, garantendo una risoluzione più rapida delle richieste e un funzionamento corretto e sicuro. Questo processo riveste un'importanza cruciale poiché consente all'applicazione sviluppata di essere accessibile agli utenti finali.

Per realizzare questa operazione, è stata utilizzata una piattaforma di hosting e sviluppo web nota come *PythonAnywhere*. Questa piattaforma offre un ambiente virtuale che consente agli sviluppatori di ospitare, eseguire e gestire le loro applicazioni *Python*, semplificando notevolmente il processo di deployment.

Per informazioni più dettagliate riguardo alla procedura di *deployment* adottata e, di conseguenza, su come eseguire il *deploy* di nuove funzionalità, nonché sulla configurazione e il funzionamento in produzione di Django, è possibile fare riferimento all'[Appendice F](#) presente in questo documento.

7 Conclusioni

In quest'ultimo capitolo viene effettuata un'analisi della situazione attuale, valutando i risultati ottenuti a seguito dell'implementazione. In particolare, viene valutato se gli obiettivi prefissati sono stati raggiunti e se lo scopo generale del progetto è stato soddisfatto.

Ulteriormente, questo capitolo fornisce un'analisi dei limiti dell'applicazione, identificando le aree che potrebbero beneficiare di sviluppi futuri. Questa sezione non si limita a esporre i fatti, ma include anche commenti riflessivi e personali sullo sviluppo di questo progetto.

Infine, il capitolo si conclude con le conclusioni finali, riassumendo i punti chiave del progetto e le considerazioni generali sulle lezioni apprese e le eventuali implicazioni per il futuro.

7.1 Risultati Ottenuti

Nella fase di realizzazione di questo progetto, la maggior parte degli obiettivi prefissati sono stati soddisfatti con successo. Tuttavia, si deve notare che non è stato possibile implementare la gestione delle competizioni e delle iscrizioni. Tale mancanza è dovuta principalmente a una questione di tempo, in quanto lo studio e la risoluzione di errori e bug hanno richiesto più tempo del previsto.

Riguardo all'interfaccia utente, è stata creata un'interfaccia semplice e intuitiva, composta da componenti familiari presenti in altri siti, rendendo così il sito più accessibile e comprensibile. Oltre alle interfacce sviluppate ex novo, sono inclusi anche il *pannello admin di Django*, utilizzato per la gestione degli utenti, gruppi e permessi, e il *pannello di traduzione di Rosetta*, un'UI che sfrutta i pannelli admin di Django e permette di cambiare le traduzioni in modo semplice.

Le pagine costruite sono state progettate con un design responsive e non richiedono l'aggiornamento dell'intero contenuto per mostrare un aggiornamento. Grazie all'utilizzo di HTMX, è stato possibile costruire pagine dinamiche e creare single-page applications. Inoltre, il templating language di Django ha facilitato la riutilizzo del codice HTML, semplificando la costruzione di pagine aggiuntive.

Invece riguardo al backend, sono state adottate le standard practice e le best practice di Django, seguendo attentamente la documentazione ufficiale che si è rivelata completa e dettagliata. Questo approccio ha consentito la costruzione di un backend solido e sicuro, supportato dall'attuazione di test su ogni singola componente.

In conclusione, i risultati conseguiti riflettono l'impegno nella realizzazione degli obiettivi, l'accurata attenzione dedicata all'implementazione dell'interfaccia utente e l'adozione consapevole delle ottimali pratiche nel settore dello sviluppo Web. Questa combinazione di fattori ha contribuito a un progetto ben organizzato e funzionale.

7.2 Consuntivo

All'inizio del progetto è stata redatta una pianificazione, in cui erano organizzate le attività necessarie per raggiungere il completamento del progetto. Tale pianificazione è stata in parte rispettata, in quanto le attività sono state svolte nell'ordine programmato, ma la durata preventivata non è stata estesa per consentire uno sviluppo agile e iterativo.

Alcune attività sono state sviluppate in fasi distinte, permettendo così l'implementazione e l'estensione delle funzionalità in modo flessibile ed efficiente. Questo approccio ha contribuito a ridurre eventuali modifiche necessarie a causa di una carenza di informazioni.

7.3 Considerazioni Personali

Inizialmente, comprendere il funzionamento di Django può apparire complicato, ma una volta acquisita e messa in pratica la conoscenza di base, è possibile estenderne le funzionalità con relativa semplicità e un ulteriore studio mirato. Pertanto, l'apprendimento pratico riveste un ruolo cruciale nell'assimilare Django.

Secondo me è benefico applicare immediatamente ciò che si è appreso, piuttosto che dedicarsi a lunghi periodi di studio teorico, poiché ciò consente una comprensione più profonda e intuitiva del framework.

Durante la realizzazione di questo progetto, ho avuto l'opportunità di sperimentare la costruzione di un'applicazione web utilizzando principalmente Django. Posso affermare con certezza che si tratta di un framework estremamente robusto, che ha offerto una solida base per lo sviluppo dell'applicazione.

Ho scoperto inoltre che non è necessario impiegare un *framework* specifico come *React.js* per costruire un *frontend* robusto e facilmente manutenibile. La scelta degli strumenti giusti può rendere l'intero processo più snello senza sacrificare la qualità.

Grazie all'utilizzo della libreria HTMX, unitamente al linguaggio di *templating* fornito da Django all'interno dei file *HTML*, sono stato in grado di creare una *single-page application* senza ricorrere a un *framework* dedicato. Questa esperienza mi ha dimostrato la versatilità e l'efficacia degli strumenti a disposizione, sottolineando la possibilità di realizzare progetti complessi con un approccio semplificato ma efficace.

7.4 Possibili sviluppi futuri

Durante la fase di implementazione, sono emerse numerose opportunità di miglioramento che potrebbero essere progettate e implementate nell'applicazione web. Tuttavia, tali miglioramenti non sono stati realizzati a causa di vincoli di tempo e della necessità di concentrarsi sulle funzionalità prioritarie richieste dal committente.

Un'area significativa di miglioramento riguarda gli aspetti di stile, UX e UI. Ad esempio, una modifica potrebbe includere la trasformazione dell'icona in alto in una "X" quando la barra laterale è visibile, rendendo più chiaro che la barra può essere chiusa. Inoltre, se lo schermo non permette di visualizzare correttamente l'header, alcune componenti potrebbero essere automaticamente spostate nella barra laterale, migliorando così la navigazione.

Un altro miglioramento potenziale riguarda l'aggiunta di un pulsante di reset nei form. Questa funzione permetterebbe di riportare il form alle informazioni di partenza con un semplice clic, migliorando l'interazione con l'utente.

In aggiunta, l'introduzione di test parametrizzati attraverso l'installazione di librerie aggiuntive, come ad esempio *parameterized*, potrebbero rendere i test più flessibili e veloci.

Infine, l'inclusione delle preferenze dell'utente migliorerebbe l'esperienza complessiva, offrendo la possibilità di personalizzare le proprie impostazioni all'interno del sito, come la scelta della lingua desiderata. Questa personalizzazione potrebbe rendere l'interazione con il sito più gradevole e coerente con le aspettative individuali degli utenti.

Gestione federazione svizzera di Twirling

7.5 Conclusioni finali

Gli obiettivi iniziali sono stati per la maggior parte raggiunti, prendendo in considerazione le esigenze del committente e con un accurato studio di Django implementare soluzioni innovative e familiari. Nonostante ciò, alcuni aspetti, come la gestione delle competizioni e delle iscrizioni, sono rimasti inesplorati, principalmente a causa di vincoli di tempo.

L'interfaccia utente, con la sua semplicità e intuitività, dimostra un'attenzione particolare all'utente. L'uso combinato di componenti standard come il pannello admin di Django e il pannello di traduzione di Rosetta, insieme alle pagine responsive realizzate con HTMX, ha contribuito a creare un'esperienza utente gradevole e familiare.

Sul lato tecnico, l'adozione delle best practice di Django e l'attenzione ai dettagli nella documentazione ufficiale hanno portato alla realizzazione di un backend solido e affidabile. Il test completo di ogni componente ha ulteriormente garantito la qualità e la sicurezza del progetto.

Tuttavia, il progetto non è stato sviluppato senza incontrare limitazioni e sfide. La necessità di dedicare tempo allo studio e alla risoluzione di bug ha influenzato l'implementazione di alcune funzionalità desiderate. Questo sottolinea l'importanza di una pianificazione accurata e dell'allocazione di risorse sufficienti per le fasi impreviste dello sviluppo.

Per concludere, il progetto dimostra un'efficace sinergia tra design, funzionalità e solida solidità tecnica. Le lezioni apprese, le realizzazioni raggiunte e le aree identificate per ulteriori sviluppi forniscono una base solida per future iterazioni e miglioramenti. L'implementazione di questo progetto ha permesso l'esplorazione e la conferma della validità dell'approccio all'implementazione di un'applicazione web utilizzando Python e Django.

Bibliografia

<http://www.fstb-twirling.ch/la-fstb.php> (01/06/2023)

<https://www.pythonanywhere.com> (01/06/2023)

<https://djangoforbeginners.com> (12/06/2023)

<https://docs.djangoproject.com/en/4.2/ref/class-based-views/base/> (1/06/2023)

<https://docs.djangoproject.com/en/4.2/topics/testing/> (15/07/2023)

<https://docs.djangoproject.com/en/4.2/topics/i18n/translation/> (18/07/23)

<https://htmx.org> (18/07/23)

<https://htmx.org/docs/#response-headers> (20/07/2023)

<https://docs.djangoproject.com/en/4.2/ref/settings/#staticfiles-dirs-prefixes> (20/07/2023)

<https://docs.djangoproject.com/en/4.2/howto/static-files/> (20/07/2023)

<https://django-rosetta.readthedocs.io/> (22/07/2023)

<https://datatables.net/manual/i18n> (28/07/2023)

<https://datatables.net/reference/option/language> (28/07/2023)

<https://docs.djangoproject.com/en/4.2/topics/testing/tools/> (30/07/2023)

<https://docs.python.org/3/library/unittest.html#assert-methods> (30/07/2023)

<https://docs.python.org/3.10/library/unittest.mock.html> (31/07/2023)

<https://www.toptal.com/python/an-introduction-to-mocking-in-python> (2/08/2023)

<https://ralphmcneal.com/unit-testing-with-pythons-patch-decorator/> (2/08/2023)

<https://docs.djangoproject.com/en/4.2/topics/testing/advanced/> (2/08/2023)

<https://docs.djangoproject.com/en/4.2/howto/static-files/deployment/> (3/08/2023)

<https://docs.djangoproject.com/en/4.2/topics/auth/default/> (9/08/2023)

<https://docs.djangoproject.com/en/4.2/ref/contrib/auth> (10/08/2023)

<https://web.archive.org/web/20130205133031/http://parand.com/say/index.php/2010/02/19/django-using-the-permission-system/> (16/08/2023)

<https://openclassrooms.com/en/courses/7107341-intermediate-django/7265147-assign-permissions-using-groups> (16/08/2023)

<https://stackoverflow.com/questions/13932774/how-can-i-use-django-permissions-without-defining-a-content-type-or-model> (18/08/2023)

Appendici

A Funzionalità base di Django

A.1 Creare Un Superuser

Per creare un super user basta utilizzare un semplice comando di django. Questo comando permette di definire un nome utente, indirizzo email e password per il nuovo superutente.

```
python manage.py createsuperuser
```

Un superuser in Django non è altro che un `auth_user` con `is_superuser=True` e che possiede tutti i permessi di default di django.

A.2 Pannello Admin

Il pannello amministrativo di Django, chiamato Django Admin, fornisce un'interfaccia predefinita per la gestione e l'amministrazione dei dati nel tuo progetto Django. Ti permette di gestire facilmente i modelli di database, visualizzare, modificare ed eliminare i record.

A.3 Cambio Lingua

Django ha un impressionante supporto per il multilingue. Se si desidera visualizzare l'interfaccia amministrativa, i moduli e altri messaggi predefiniti in una lingua diversa dall'inglese, è possibile regolare la configurazione **LANGUAGE_CODE** nel file `django_project/settings.py`. Questa opzione è impostata automaticamente per l'inglese americano, `en-us`.

Per cambiare la lingua predefinita, seguire i passaggi seguenti:

1. Aprire il file **settings.py** nel progetto Django.
2. Trovare la variabile **LANGUAGE_CODE**.
3. Modificare il valore della variabile **LANGUAGE_CODE** con il codice della lingua desiderata. Ad esempio, per l'italiano, impostare il valore su `'it'`.
4. Salvare il file **settings.py**.

A.4 Deploy dei file Statici

Il deploy dei file statici avviene tramite la configurazione dell'opzione **STATIC_ROOT**. Quando si esegue l'applicazione in modalità di sviluppo e **DEBUG** è True, Django cerca i file statici utilizzando vari "finder" definiti nell'impostazione **STATICFILES_FINDERS**. Questi "finder" includono il `FileSystemFinder`, che cerca i file statici nella directory elencata in **STATICFILES_DIRS**, e l'`AppDirectoriesFinder`, che cerca i file statici nelle directory statiche di ciascuna applicazione Django installata.

Tuttavia, per migliorare le prestazioni, quando **DEBUG** è False, Django si limita a cercare i file statici nella directory **STATIC_ROOT**. Per garantire che tutti i file statici siano presenti in **STATIC_ROOT**, è necessario utilizzare il comando `collectstatic`, che copierà tutti i file individuati dai "finder" elencati in **STATICFILES_FINDERS** in **STATIC_ROOT**.

Disabilitare **DEBUG** è importante per evitare che dati segreti vengano rivelati nell'applicazione in produzione. Quando **DEBUG** è True, la ricerca dei file statici può essere più lenta in quanto Django deve attraversare diverse directory prima di trovare un singolo file. Quando **DEBUG** è False, questa ricerca viene disattivata e Django si limita a cercare i file nella directory **STATIC_ROOT**, ottimizzando le prestazioni.

A.5 Notifica di errori

Quando **DEBUG** è impostato su False, Django invia notifiche via email alle persone specificate nella lista **ADMINS** in caso di eccezioni sollevate nel ciclo di richiesta/risposta. Questo avviene quando è configurato l'`AdminEmailHandler` nel **LOGGING** (che viene fatto per impostazione predefinita).

L'impostazione **ADMINS** è una lista di tuple, dove ogni tupla rappresenta una persona da notificare. Ogni tupla deve contenere il nome completo e l'indirizzo email della persona. Ad esempio:

```
ADMINS = [
    ('Nome completo', 'indirizzo@email.com'),
]
```

Quando si verifica un'eccezione in modalità non di debug, Django invierà automaticamente un'email contenente i dettagli dell'errore alle persone elencate nella lista **ADMINS**. Questo può essere utile per monitorare gli errori e prendere provvedimenti tempestivi per risolverli.

A.6 Validazione

La validazione di un **form** avviene durante la pulizia dei dati. Se si desidera personalizzare questo processo, ci sono vari punti in cui apportare modifiche, ognuno con uno scopo diverso. Durante il processo di elaborazione del form vengono eseguiti tre tipi di metodi di pulizia. Di solito vengono eseguiti quando si chiama il metodo `is_valid()` su un form. Esistono altre situazioni che possono innescare la pulizia e la validazione (accesso all'attributo **errors** o chiamata diretta a `full_clean()`), ma di solito non sono necessarie.

A.7 Errore Di Validazione

La exception **ValidationError** viene sollevata durante il processo di validazione di un form in Django. Viene utilizzata per segnalare errori o problemi nei dati inseriti dall'utente.

```
from django.core.exceptions import ValidationError

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError("Il numero deve essere pari.")
```

In generale, qualsiasi metodo di pulizia può sollevare una `ValidationError` se si riscontra un problema con i dati che sta elaborando. Se non venisse sollevata alcuna `ValidationError`, il metodo dovrebbe restituire i dati puliti (normalizzati) come oggetto Python.

La `ValidationError` viene sollevata quando si verificano situazioni in cui i dati non soddisfano i criteri di validazione definiti. Ad esempio, se un campo richiede una stringa di lunghezza minima e viene fornita una stringa troppo corta, è opportuno sollevare una `ValidationError` per indicare che il campo non è valido.

Quando viene sollevata una `ValidationError` durante il processo di validazione di un form o di un modello, Django cattura l'eccezione e la associa al campo corrispondente. In questo modo, l'errore può essere visualizzato all'utente o gestito in altri modi appropriati. Ad esempio mostrato tramite:

```
<div class="invalid-feedback">{{ form.field_name.errors }}</div>
```

A.8 Validators

I [validators](#) in Django sono funzioni o classi che consentono di effettuare la validazione dei dati. Essi verificano se i dati inseriti in un campo di un form soddisfano determinate condizioni o requisiti.

Le funzioni validatori sono semplici funzioni Python che prendono un singolo argomento (il valore da validare) e sollevano una **ValidationError** se l'input non è valido. Le classi validatori sono oggetti invocabili che implementano un metodo chiamato `__call__()`, che viene eseguito quando il validator viene chiamato. Anche le classi validatori devono sollevare una **ValidationError** se l'input non è valido.

In Django, sono disponibili diversi validatori predefiniti che coprono una vasta gamma di casi comuni, come la validazione del formato di un indirizzo email o di un URL, la verifica di un valore minimo o massimo, ecc. Tuttavia, è anche possibile definire validatori personalizzati per adattarsi alle specifiche esigenze del progetto.

```
from django.core.validators import EmailValidator

email_validator = EmailValidator(message="Inserisci un indirizzo email valido.")

class MyForm(forms.Form):
    email = forms.EmailField(validators=[email_validator])
```

```
from django.core.validators import MinLengthValidator

min_length_validator = MinLengthValidator(8, message="La lunghezza minima è 8.")

class MyForm(forms.Form):
    password = forms.CharField(validators=[min_length_validator],
                               widget=forms.PasswordInput)
```

```
from django.core.validators import MaxLengthValidator

max_length_validator = MaxLengthValidator(100, message="La lunghezza max è 100.")

class MyForm(forms.Form):
    description = forms.CharField(validators=[max_length_validator],
                                  widget=forms.Textarea)
```

In questi esempi, i validatori predefiniti [EmailValidator](#), [MinLengthValidator](#) e [MaxLengthValidator](#) vengono utilizzati per effettuare diverse verifiche. Questi validatori sono già inclusi in Django e possono essere utilizzati direttamente specificando il validator come argomento nella definizione del campo del form.

Esempi di funzioni validatori: [validate_email](#), [validate_slug](#), [validate_image_file_extension](#).

A.9 Esempio Funzione Validatrice Custom

Per esempio, si può creare una funzione per verificare se il numero inserito nel form è pari:

```
from django.core.exceptions import ValidationError
from django.utils.translation import gettext_lazy as _

def validate_even(value):
    if value % 2 != 0:
        raise ValidationError(
            _("%(value)s is not an even number"),
            params={"value": value},
        )
```

Poi questo validatore si può aggiungere a un campo di un modello tramite l'argomento `validators` del campo:

```
from django.db import models

class MyModel(models.Model):
    even_field = models.IntegerField(validators=[validate_even])
```

Poi questo validatore si può aggiungere a un campo di un modello tramite l'argomento `validators` del campo:

```
from django import forms

class MyForm(forms.Form):
    even_field = forms.IntegerField(validators=[validate_even])
```


A.10 Passaggi Validazione

La validazione di un form è suddivisa in diversi passaggi, che possono essere personalizzati o sovrascritti:

- 1) Il metodo **to_python()** su un campo è il primo passaggio in ogni validazione. Coerce il valore in un tipo di dati corretto e solleva una `ValidationError` se ciò non è possibile. Questo metodo accetta il valore grezzo dal widget e restituisce il valore convertito. Ad esempio, un `FloatField` trasformerà i dati in un float Python o solleverà una `ValidationError` nel caso di problemi.
- 2) Il metodo **validate()** su un campo gestisce la validazione specifica del campo che non è adatta per un validatore. Prende un valore che è stato convertito in un tipo di dati corretto e solleva una `ValidationError` in caso di errore. Questo metodo non restituisce nulla e non dovrebbe modificare il valore. Dovresti sovrascriverlo per gestire la logica di validazione che non puoi o non vuoi inserire in un validatore.
- 3) Il metodo **run_validators()** su un campo esegue tutti i validatori del campo e aggrega tutti gli errori in una singola `ValidationError`. Non dovresti avere bisogno di sovrascrivere questo metodo.
- 4) Il metodo **clean()** su una sottoclasse di `Field` è responsabile per l'esecuzione di `to_python()`, `validate()` e `run_validators()` nell'ordine corretto e per propagare i loro errori. Se, in qualsiasi momento, uno dei metodi solleva una `ValidationError`, la validazione si interrompe e quell'errore viene sollevato. Questo metodo restituisce i dati puliti, che vengono quindi inseriti nel dizionario **cleaned_data** del form.
- 5) Il metodo **clean_<field name>()** viene chiamato su una sottoclasse di form - dove **<field name>** viene sostituito con il nome dell'attributo del campo del form. Questo metodo effettua la pulizia specifica dell'attributo, indipendentemente dal tipo di campo che è. Questo metodo non riceve alcun parametro. È necessario cercare il valore del campo in **self.cleaned_data** e ricordare che sarà un oggetto Python in questo punto, non la stringa originale inviata nel form (sarà presente in **cleaned_data** perché il metodo `clean()` generale del campo ha già pulito i dati una volta).
- 6) Il metodo **clean()** della sottoclasse di form può eseguire la validazione che richiede l'accesso a più campi del form. È qui che è possibile inserire controlli come "se il campo A viene fornito, il campo B deve contenere un indirizzo email valido". Questo metodo può restituire un dizionario completamente diverso, se lo desidera, che verrà utilizzato come **cleaned_data**. Poiché i metodi di validazione del campo sono stati eseguiti al momento in cui viene chiamato `clean()`, è possibile accedere anche all'attributo **errors** del form, che contiene tutti gli errori generati dalla pulizia dei singoli campi.

Eventuali errori sollevati dall'override di `clean()` del form non sono associati a un campo specifico. Vengono inseriti in un campo speciale (chiamato **__all__**), che è possibile accedere tramite il metodo **non_field_errors()** se necessario. Se si desidera associare gli errori a un campo specifico del form, è necessario chiamare **add_error()**.

```
<div class="invalid-feedback">{{ form.non_field_errors }}</div>
```

A.11 Valori di Traduzione Fuzzy

Nel contesto di localizzazione e internazionalizzazione in Django, un valore "fuzzy" indica una stringa tradotta che potrebbe non essere accurata o completa. Questo stato "fuzzy" può essere impostato automaticamente quando, ad esempio, la stringa originale viene modificata, e il sistema non è sicuro se la traduzione esistente sia ancora valida.

Di default, `compilemessages`, il comando che genera i file di traduzione compilati, ignora le voci contrassegnate come "fuzzy", perciò quelle traduzioni non saranno applicate.

A.12 Eseguire i Test

Per eseguire i test scritti in Django utilizzando unittest, si usa il comando test dell'utility **manage.py** del progetto:

```
$ ./manage.py test
```

I test possono anche essere eseguiti da un IDE compatibile con Django. Di default, questo comando rileva i test in qualsiasi file chiamato `test*.py` nella directory corrente.

Per eseguire test specifici, si possono fornire "etichette di test" a `./manage.py test` o un percorso a una directory.

Se i file di test hanno nomi diversi dal pattern `test*.py`, si può specificare un pattern personalizzato con l'opzione `-p` o `--pattern`.

È consigliato eseguire i test con gli avvisi Python attivati usando `python -W manage.py test` per visualizzare eventuali avvertimenti di deprecazione e suggerimenti su possibili miglioramenti nel codice.

A.13 Testare le View

Per testare le *View* in Django, si utilizza il *test client*, una classe Python che simula un browser web. Questo permette di:

- Simulare richieste *GET* e *POST* su un *URL* e osservare la risposta, dallo status *HTTP* ai contenuti della pagina.
- Verificare la catena di reindirizzamenti e controllare l'*URL* e lo status code di ciascun passaggio.
- Assicursi che una specifica richiesta venga elaborata da un determinato template di Django con un contesto che contiene determinati valori.

Tuttavia, il test client di Django non sostituisce framework *in-browser* come Selenium. Mentre il test client verifica che il template giusto venga renderizzato con i dati corretti.

RequestFactory testa le funzioni delle *View* direttamente, ignorando il *Routing* e i *middleware*.

I framework "in-browser" come Selenium sono utili per testare l'HTML renderizzato e la funzionalità delle pagine web, come quella basata su JavaScript.

Per una suite di test completa, è consigliato utilizzare una combinazione di questi tipi di test.

A.14 Esempio come Testare le View

Per utilizzare il test client in Django e testare le *View*, si istanzia **django.test.Client** e si recuperano le pagine web:

```
>>> from django.test import Client
>>> c = Client()
>>> response = c.post("/login/", {"username": "john", "password": "smith"})
>>> response.status_code
200

>>> response = c.get("/customer/details/")
>>> response.content
b'<!DOCTYPE html...'
```
```

Il test client non necessita che il server web sia in esecuzione, poiché evita l'overhead dell'HTTP e interagisce direttamente con il framework Django, rendendo l'esecuzione dei test unitari veloce.

Quando si recuperano le pagine, è importante specificare il percorso dell'URL e non il dominio completo.

## A.15 Funzionalità Test Client

**1. Controllo CSRF:** Se si desidera che il test client esegua controlli CSRF, è possibile creare un'istanza del test client che applica questi controlli. Per fare ciò, passa l'argomento `enforce_csrf_checks` durante la costruzione del client.

```
Client(enforce_csrf_checks=True)
```

**2. Impostazione degli Header:** L'argomento `headers` permette di specificare header predefiniti che verranno inviati con ogni richiesta. Ad esempio, per impostare un header User-Agent.

```
client = Client(headers={"user-agent": "curl/7.79.1"})
```

**3. Impostazione delle Variabili WSGI:** Gli argomenti keyword arbitrari in `**defaults` impostano le variabili d'ambiente WSGI. Ad esempio, per impostare il nome dello script.

```
client = Client(SCRIPT_NAME="/app/")
```

**4. Lista di reindirizzamenti:** utilizzando il client di test di Django con il parametro ***follow=True***, si può fare quanto segue.

```
def test_redirect_list(self):
 # Ottenere la risposta dalla view
 response = self.client.get('/myapp/some_view/', follow=True)

 # Ottenere una lista di reindirizzamenti dalla risposta
 redirect_chain = response.redirect_chain

 # Stampa la lista di reindirizzamenti
 for redirect in redirect_chain:
 print(f"Reindirizzato a: {redirect[0]} - Status code: {redirect[1]}")
```

**5. Argomenti con Prefisso HTTP\_:** Gli argomenti *keyword* che iniziano con un prefisso `"HTTP_"` sono impostati come *header*, ma l'uso dell'argomento `headers` è preferibile per una maggiore leggibilità.

**6. Precedenza degli Header e Argomenti Extra:** I valori degli *headers* e degli argomenti extra passati a metodi come `get()`, `post()`, ecc., hanno la precedenza rispetto ai valori predefiniti passati al costruttore della classe. Ciò significa che è possibile sovrascrivere la configurazione della classe usando `get()` e `post()`.

**7. Controllo delle Eccezioni Durante la Richiesta:** L'argomento `raise_request_exception` controlla se le eccezioni sollevate durante la richiesta dovrebbero essere sollevate anche nel test. Il valore predefinito è `True`.

**8. Personalizzazione dell'Encoder JSON:** L'argomento `json_encoder` permette di impostare un encoder JSON personalizzato per la serializzazione JSON descritta nella documentazione di [post\(\)](#).

## A.16 Definizione Test Client e Metodi di Richiesta

Per definire un **test client**:

```
class Client(enforce_csrf_checks=False, raise_request_exception=True,
 json_encoder=DjangoJSONEncoder, *, headers=None, **defaults):
```

Per definire una richiesta **get**:

```
client.get(path, data=None, follow=False, secure=False, *, headers=None, **extra)
```

Per definire una richiesta **post**:

```
Client.post(path, data=None, content_type=MULTIPART_CONTENT, follow=False,
 secure=False, *, headers=None, **extra)
```

## A.17 Assertions in Django Test

Django nel suo pacchetto di librerie *django.test*, fornisce metodi di asserzione per facilitare il test delle applicazioni web. Questi metodi ampliano le funzionalità standard di *unittest.TestCase*. Queste sono le assertions standard di Django:

- **assertRaisesMessage** e **assertWarnsMessage**: Assicurano che una chiamata lanci un'eccezione o un avviso con un messaggio specifico.
- **assertFieldOutput**: Verifica che un campo di modulo si comporti correttamente con vari input.
- **assertFormError** e **assertFormsetError**: Verificano che un campo in un modulo o in un set di moduli restituisca errori specifici.
- **assertContains** e **assertNotContains**: Controllano che una risposta contenga o non contenga un testo specifico.
- **assertTemplateUsed** e **assertTemplateNotUsed**: Verificano che un determinato template sia stato o non sia stato utilizzato nella risposta.
- **assertURLEqual**: Confronta due URL.
- **assertRedirects**: Verifica che una risposta effettui un reindirizzamento ad un URL specifico.
- **assertHTMLEqual** e **assertHTMLNotEqual**: Confrontano due frammenti HTML considerando la semantica HTML.
- **assertXMLEqual** e **assertXMLNotEqual**: Confrontano due frammenti XML considerando la semantica XML.
- **assertInHTML**: Controlla che un frammento HTML sia contenuto in un altro.
- **assertJSONEqual** e **assertJSONNotEqual**: Confrontano frammenti JSON.
- **assertQuerySetEqual**: Confronta un queryset con un insieme di valori.
- **assertNumQueries**: Verifica che un numero specifico di query al database vengano eseguite durante una chiamata.

La maggior parte di questi metodi permette di personalizzare il messaggio di errore attraverso l'argomento ***msg\_prefix***. Questi metodi di asserzione offrono funzionalità dettagliate e specifiche per facilitare il testing in applicazioni Django.

## A.18 Assertions in Unittest

La classe *unittest.TestCase* fornisce vari metodi assert per verificare e segnalare errori. Questi metodi sono accessibili anche all'interno di *django.test.TestCase*. Queste sono le assertions standard di unittest:

- **assertEqual**: verifica che due valori siano uguali.
- **assertNotEqual**: verifica che due valori non siano uguali.
- **assertTrue** e **assertFalse**: verifica che una espressione sia vera o falsa.
- **assertIs** e **assertIsNot**: verifica che due oggetti siano o non siano lo stesso oggetto.
- **assertIsNone** e **assertIsNotNone**: verifica se un'espressione è None o non lo è.
- **assertIn** e **assertNotIn**: verifica se un elemento è o non è in un contenitore.
- **assertIsInstance** e **assertNotIsInstance**: verifica se un oggetto è o non è un'istanza di una classe o tupla di classi.
- **assertRaises** e **assertRaisesRegex**: verifica che venga sollevata un'eccezione quando viene chiamata una funzione.
- **assertWarns** e **assertWarnsRegex**: verifica che venga emesso un avvertimento quando viene chiamata una funzione.
- **assertLogs**: è un context manager per verificare che almeno un messaggio venga registrato su un logger.
- **assertNoLogs**: è un context manager per verificare che nessun messaggio venga registrato su un logger.
- **assertAlmostEqual** e **assertNotAlmostEqual**: verifica che due numeri siano approssimativamente uguali o non lo siano.
- **assertGreater**, **assertGreaterEqual**, **assertLess**, e **assertLessEqual**: verifica relazioni di maggiore, maggiore o uguale, minore, e minore o uguale tra due valori.
- **assertRegex** e **assertNotRegex**: verifica che un testo corrisponda o non corrisponda a un'espressione regolare.
- **assertCountEqual**: verifica che due sequenze contengano gli stessi elementi, indipendentemente dal loro ordine.
- **assertMultiLineEqual**: verifica che due stringhe multilinea siano uguali.
- **assertSequenceEqual**: verifica che due sequenze siano uguali.
- **assertListEqual** e **assertTupleEqual**: verifica che due liste o tuple siano uguali.
- **assertSetEqual**: verifica che due set siano uguali.
- **assertDictEqual**: verifica che due dizionari siano uguali.

Tutti questi metodi accettano un argomento "msg" che, se specificato, viene utilizzato come messaggio di errore in caso di fallimento.

## A.19 Moking In Python Unittest

La libreria **unittest.mock** in Python consente di effettuare test sostituendo parti del sistema in esame con oggetti mock e verificando come sono stati utilizzati. Fornisce una classe **Mock** centrale che elimina la necessità di creare molti *stub* nel set di test.

Dopo un'azione, è possibile verificare quali metodi o attributi sono stati utilizzati e con quali argomenti. Si possono anche definire valori di ritorno e impostare attributi come al solito.

La libreria offre inoltre un *decorator* **patch()** per modificare attributi a livello di modulo e classe durante un test. Mock è progettato per funzionare con unittest seguendo il pattern 'azione -> asserzione' anziché 'registrazione -> riproduzione' comune ad altri framework di mocking.

## A.20 Verificare che è presente un Utente autenticato

Per capire se un utente è autenticato è facile, basta utilizzare il metodo **is\_authenticated** della classe **User** di Django. In questo modo se si tratta di un Template:

```
{% if user.is_authenticated %}
...
{% endif %}
```

## A.21 Implementare Database Triggers con Django Signals

I *Signals* in Django sono un meccanismo di notifica che permette a determinate parti del codice (*handlers*) di essere notificate quando avvengono determinati eventi. Essi sono utilizzati principalmente per permettere l'esecuzione di codice in modo asincrono, quando si scatenano particolari avvenimenti. Questi possono includere operazioni come la creazione, modifica o cancellazione di un oggetto modello.

Un esempio può essere quello di eseguire certe operazioni prima che un istanza *MemberChange* venga eliminata.

```
@receiver(pre_delete, sender=MemberChange)
def do_something_before_deleting_member_change_model(sender, instance, **kwargs):
 <do something>
```

## B Traduzione con Django

1. Una delle prime cose da fare è abilitare l'internazionalizzazione nel file di configurazione **settings.py**:

```
USE_I18N = True
```

2. Nello stesso file di configurazione sono state specificate le lingue supportate dall'applicazione:

```
LANGUAGES = [
 ("en", _("English")),
 ("fr", _("French")),
 ("de", _("German")),
 ("it", _("Italian")),
]
```

Il carattere "\_" che precede le parentesi che circondano i nomi delle lingue rappresenta la denominazione standard della funzione utilizzata per la traduzione dei testi. Nel pacchetto standard di Django, questa funzione può essere **gettext\_lazy** o **gettext**, entrambe presenti nel modulo *django.utils.translation*. All'interno di **settings.py** è stata importata nel seguente modo:

```
from django.utils.translation import gettext_lazy as _
```

In **settings.py**, si utilizza **gettext\_lazy** invece di **gettext** per evitare errori, poiché **gettext\_lazy** permette la traduzione di testi in un momento successivo, garantendo che l'applicazione e le sue configurazioni siano caricate prima della traduzione.

Quindi **gettext\_lazy** viene utilizzata quando la traduzione potrebbe richiedere risorse aggiuntive o potrebbe essere influenzata da variabili d'ambiente che devono essere stabilite prima di avviare la traduzione. Inoltre consente di ottimizzare l'uso delle risorse, traducendo solo i testi effettivamente richiesti durante l'esecuzione dell'applicazione.



3. Nel file **settings.py** è specificato il percorso dei file contenenti le traduzioni, che sarà all'interno della cartella "locale" situata nella radice del progetto. Questa cartella "locale" conterrà i file di traduzione organizzati nelle rispettive sotto-cartelle.

```
LOCALE_PATHS = [
 BASE_DIR / "locale", # assuming BASE_DIR is the path to your project
]
```

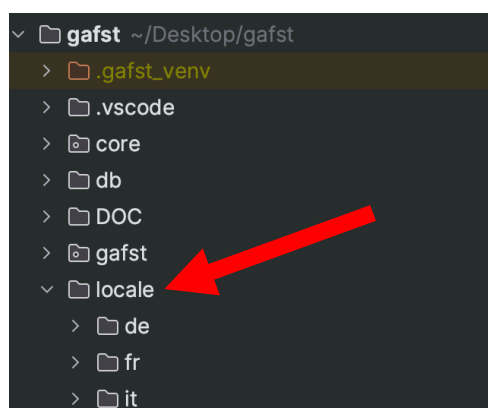


Figura 10 - Cartella locale

4. C'è anche specificato quale è la lingua predefinita, in questo caso è stato scelto l'inglese per facilità di implementazione, ma si può cambiare e mettere un'altra delle lingue supportate:

```
LANGUAGE_CODE = "en" # default language
```

5. Per attivare il sistema di traduzione in Django, è necessario integrare un *middleware* specifico, ovvero **LocaleMiddleware**, all'interno dell'elenco di Middleware nel file di configurazione. Questo middleware va posizionato dopo *SessionMiddleware*, ma prima di *CommonMiddleware*:

```
MIDDLEWARE = [
 "django.middleware.security.SecurityMiddleware",
 "django.contrib.sessions.middleware.SessionMiddleware",
 "django.middleware.locale.LocaleMiddleware",
 "django.middleware.common.CommonMiddleware",
 "django.middleware.csrf.CsrfViewMiddleware",
 "django.contrib.auth.middleware.AuthenticationMiddleware",
 "django.contrib.messages.middleware.MessageMiddleware",
 "django.middleware.clickjacking.XFrameOptionsMiddleware",
]
```

6. Per attivare la traduzione automatica delle pagine in Django, è necessario modificare gli *URL* in modo da includere un prefisso che indichi la lingua. Ad esempio, l'*URL* per la home page potrebbe cambiare da "home" a "it/home" per la versione italiana. Per implementare automaticamente questi prefissi di lingua, si utilizza la funzione ***i18n\_patterns()*** per avvolgere gli *URL* delle applicazioni Django definiti nel file ***urls.py***. All'interno di questa lista di *URL*, è importante includere anche ***django.conf.urls.i18n*** per permettere il processo di traduzione:

```
from django.conf.urls.i18n import i18n_patterns

urlpatterns = [
 path("admin/", admin.site.urls),
 path("accounts/", include("django.contrib.auth.urls")),
]

urlpatterns += i18n_patterns(
 path("i18n/", include("django.conf.urls.i18n")),
 path("", include("core.urls")),
)
```

7. Per tradurre le stringhe in Django, è necessario specificare quale testo deve essere tradotto utilizzando i metodi predefiniti. Per indicare le stringhe all'interno dei file *python*, si utilizza la funzione ***gettext***. Questa funzione è il punto chiave per identificare i testi che devono essere tradotti, e consente a Django di estrarre queste stringhe per la creazione dei file di traduzione ***.po***:

```
from django.utils.translation import gettext as _

class MembersCardsView(CardTemplateView):
 translation.activate(translation.get_language())
 model = Member
 template_name = "admin/cards/members.html"
 card_title = _("Members")
```

8. Per indicare del testo da tradurre all'interno di un file HTML, è necessario caricare il modulo di internazionalizzazione di Django utilizzando ***{% load i18n %}***, che si posiziona all'inizio del file. Successivamente, si può fare uso del tag ***translate*** di Django per indicare specificamente la stringa che necessita di traduzione:

```
{% load i18n %}

<li class="nav-item">

 <i class="bi bi-people-fill me-2"></i>
 {% translate "Members" %}


```

**9.** Per estrarre tutte le stringhe da tradurre presenti nel progetto, è necessario eseguire un comando specifico, fornendo il codice della lingua in cui si desidera effettuare la traduzione. Questo comando genererà il file **.po**, che ospiteranno le traduzioni corrispondenti:

```
python manage.py makemessages -l <language_code>
```

**10.** Una volta estratte le stringhe, si possono tradurre nei file **.po** corrispondenti alle lingue desiderate:

```
#: core/views.py:181 templates/structure/sidebar.html:22
msgid "Members"
msgstr "Membri"
```

**11.** Dopo aver estratto e tradotto le stringhe, è necessario compilare i file **.po** utilizzando il comando appropriato. Questo processo genera i file **.mo**, che Django sfrutterà per eseguire le traduzioni. Questo è il comando da eseguire:

```
python manage.py compilemessages
```

## B.1 Traduzione con Rosetta

1. Per utilizzare il pannello di traduzione di *Rosetta*, è necessario prima installare il pacchetto con il seguente comando:

```
pip install django-rosetta
```

2. Come cosa successiva è stata aggiunta l'applicazione **"rosetta"** all'interno di *INSTALLED\_APPS* del file di configurazione **settings.py**:

```
INSTALLED_APPS = [
 "django.contrib.admin",
 "django.contrib.auth",
 "django.contrib.contenttypes",
 "rosetta",
]
```

3. È anche necessario aggiungere il percorso (*path*) all'interno di *urlpatterns* nel file **urls.py**, questo permette di raggiungere le *View* di *Rosetta*:

```
if "rosetta" in settings.INSTALLED_APPS:
 urlpatterns += [re_path(r"^rosetta/", include("rosetta.urls"))]
```

4. Per visualizzare più traduzioni sulla stessa pagina, è stata aggiunta questa configurazione in **settings.py**:

```
ROSETTA_MESSAGES_PER_PAGE = 50
```

5. Per accedere al pannello di traduzione di *Rosetta* basta andare all'indirizzo:

```
https://<nome_del_sito>/rosetta
```

6. Una volta fatto l'accesso con un account amministratore si verrà portati all'interno di questa pagina:

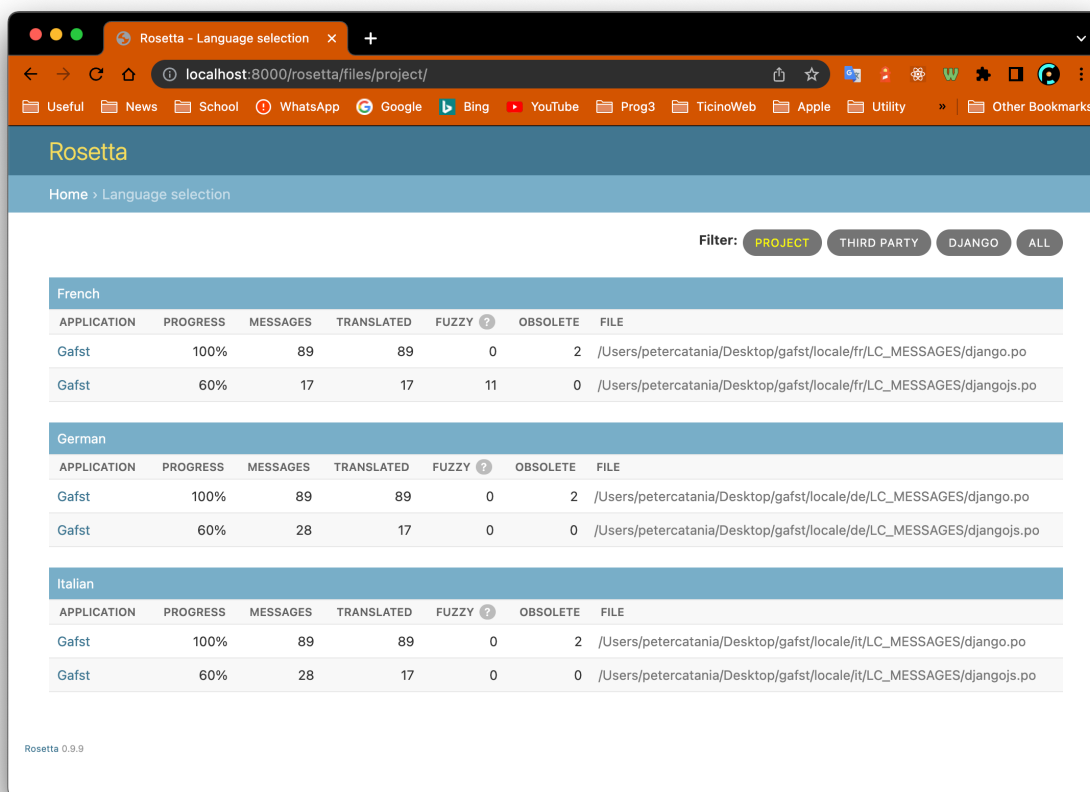


Figura 11 - Pannello di Traduzione di Rosetta

7. Per velocizzare la traduzione è stato abilitato il suggerimento di traduzione tramite **DEEPL**, una delle migliori API di traduzione che sfrutta l'intelligenza artificiale. Questo permette di avere traduzioni coerenti con un semplice click sotto la casella di traduzione.

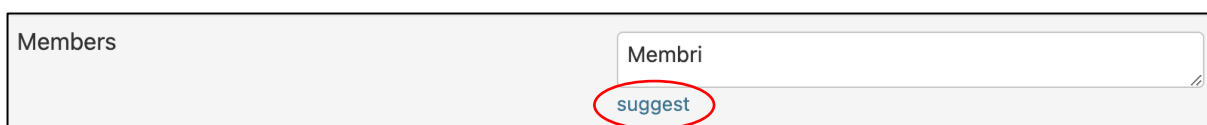


Figura 12 - Tasto di Suggerimento Traduzione

Per avere questa funzionalità si è creato un account **DEEPL** (free Api → 500,000 caratteri al mese) per avere una chiave di identificazione e sono state aggiunte queste configurazioni all'interno di **settings.py**:

```
ROSETTA_ENABLE_TRANSLATION_SUGGESTIONS = True
DEEPL_AUTH_KEY = "54f6b9f5-0aaf-3d0e-0a1f-0e53a394b2ed:fx"
```

## B.2 Traduzioni all'interno di file JS

1. Per integrare una **JavaScriptCatalog View**, è stato necessario modificare il file **urls.py** nell'applicazione generale **gafst**. In particolare, bisogna inserirla nell'elenco delle viste accessibili. Considerando che il nostro sito implementa la traduzione attraverso l'internazionalizzazione, è importante aggiungere la vista all'interno dei **i18n\_patterns**:

```
urlpatterns += i18n_patterns(
 path("i18n/", include("django.conf.urls.i18n")),
 path("jsi18n/", JavaScriptCatalog.as_view(), name="javascript-catalog"),
 path("", include("core.urls")),
)
```

2. Una volta resa disponibile la **JavaScriptCatalog View** come **URL**, per sfruttare il metodo **gettext** e gli altri metodi di traduzione, è necessario inserire il **javascript-catalog** come primo script da caricare:

```
<!-- Scripts -->
<!-- django js translations -->
<script src="{% url 'javascript-catalog' %}"></script>

<!-- bootstrap -->
<script src="{% static 'js/bootstrap5.3/bootstrap.bundle.min.js' %}"
 integrity="sha384-
geWF76RCwLtnZ8qwWowPQNguL3RmwHVBC9FhGdlKrxdiJJigb/j/68SIy3Te4Bkz"
 crossorigin="anonymous">
</script>
...
```

3. Una volta effettuata questa operazione, è possibile utilizzare il metodo **gettext** per marcare le stringhe da tradurre, in modo che possano essere successivamente tradotte e poi compilate:

```
...
buttons: [
 {
 extend: 'collection',
 text: gettext('Export'),
 buttons: ['pdf', 'excel', 'csv'],
 id: 'export',
 className: 'btn btn-primary btn-sm',
 },
],
...
```

4. Per estrarre tutte le stringhe da tradurre presenti nei **file JS**, è stato necessario eseguire un comando specifico con il dominio **djangojs**, fornendo il codice della lingua in cui si desidera effettuare la traduzione. Questo comando genererà il **file .po**, che ospiteranno le traduzioni in italiano:

```
python manage.py makemessages -d djangojs -l it
```

5. Infine, bisogna aggiornare i file **.po** con le traduzioni della lingua corrispondente e compilarli per rendere effettive le traduzioni

Gestione federazione svizzera di Twirling

## C Verifica del identità dell'utente autenticato

### C.1 Verifica in Template

**1. Verifica utilizzo contesto di autorizzazione:** Per fare il controllo che un utente autenticato appartiene ad un certo gruppo bisogna essere sicuri che nel contesto dei template sia caricato il modulo di autorizzazione di Django, cioè che nel file **settings.py** sia contenuta questa configurazione:

```
TEMPLATES = [
 {
 'OPTIONS': {
 'context_processors': [
 'django.contrib.auth.context_processors.auth',
],
 },
],
]
```

**2. Creazione permessi per ogni gruppo:** Per determinare l'appartenenza dell'utente al gruppo, si sono creati due permessi, corrispondenti a *FSTB Admin* e *Club Admin*. In questo modo si semplifica la verifica, controllando se si hanno i corrispettivi privilegi.

Per fare questo, si è creata la classe *PermissionsSupport*, che non è altro che un modello Django specificamente configurato per gestire i permessi personalizzati senza essere collegato a una tabella del database. La classe disabilita i permessi predefiniti e definisce due nuovi permessi, *fstb\_admin\_permissions* e *club\_admin\_permissions*, per distinguere i corrispettivi ruoli di *FSTB Admin* e *Club Admin*.

Questa struttura offre un controllo preciso dei permessi all'interno dell'applicazione ed è particolarmente utile per aggiungere permessi che non sono direttamente collegati a un singolo modello della banca dati. Tuttavia, è importante sottolineare che, in un contesto tipico di Django, i permessi vengono definiti all'interno dei corrispettivi modelli. In questo caso specifico, i permessi non sono associati a un modello particolare poiché riflettono autorizzazioni che si applicano su più tabelle.

```
class PermissionsSupport(models.Model):
 """This model is used to add permissions to the User model"""

 class Meta:
 managed = False # No database table creation or deletion \
 # operations will be performed for this model.

 default_permissions = () # disable "add", "change", "delete"
 # and "view" default permissions

 permissions = (
 ("fstb_admin_permissions", "Have FSTB Admin Permissions"),
 ("club_admin_permissions", "Have Club Admin Permissions"),
)
```

**3. Associazione permessi ai corrispettivi gruppi:** L'aggiunta dei permessi è effettuata all'interno dello script *insert\_defaults*. Prima vengono prendono i permessi corrispondenti, poi si creano i gruppi corrispettivi, e infine si associano i permessi a questi gruppi.

Specificamente, al gruppo *FSTB admin* vengono assegnati i permessi sia *fstb\_admin\_permissions* che di *club\_admin\_permissions*. Questo è fatto in modo che gli utenti in questo gruppo possano avere le autorizzazioni necessarie per eseguire anche le azioni che sono permesse ai club admin.

```
permissions
fstb_admin_permissions = Permission.objects.get(
 codename="fstb_admin_permissions"
)
club_admin_permissions = Permission.objects.get(
 codename="club_admin_permissions"
)

insert default groups for django auth, based on GroupEnum
fstb_admin_group = Group.objects.create(name=GroupEnum.FSTB_ADMIN.value)
fstb_admin_group.permissions.set([fstb_admin_permissions])
fstb_admin_group.permissions.set([club_admin_permissions])

club_admin_group = Group.objects.create(name=GroupEnum.CLUB_ADMIN.value)
club_admin_group.permissions.set([club_admin_permissions])
```

**4. Controllo nel template:** Per verificare che l'utente autenticato possieda i permessi corretti all'interno di un template e quindi capire a che gruppo appartiene, è stata adottata la seguente procedura.

```
{% if perms.core.fstb_admin_permissions %}
 {% include input_template with field=form.user_select ... %}
 {% include input_template with field=form.group_select ... %}
 <hr>
{% endif %}
```

## C.2 Verifica In View, Models e moduli Python

Per verificare che un utente appartiene a *FSTB Admin* o a *Club Admin*, sono state implementate due semplici funzioni. Queste funzioni operano considerando un utente, il quale, nella maggioranza dei casi, sarà l'utente autenticato.

```
def is_user_fstb_admin(user):
 if not user or not user.is_authenticated:
 return False

 return user.groups.filter(name=GroupEnum.FSTB_ADMIN.value).exists()

def is_user_club_admin(user):
 if not user or not user.is_authenticated:
 return False

 return user.groups.filter(name=GroupEnum.CLUB_ADMIN.value).exists()
```



## D Context Processor

**1. Definizione del Context Processor:** All'interno dell'app core, è stato creato un file *context\_processors.py* dove inserire i propri *Context Processor* personalizzati. In questo file, si è definita una funzione *core\_context* che restituisce un dizionario con le variabili globali desiderate.

```
from core.utils import get_user_club

def core_context(request):

 logged_in_user = request.user

 return {
 "logged_in_user_member_club": get_user_club(logged_in_user),
 }
```

**2. Aggiunta del Context Processor alle Impostazioni:** Una volta definito il *Context Processor*, è necessario informare Django di usarlo. Per fare ciò, bisogna aggiungerlo alla lista *context\_processors* nelle impostazioni di Django in *settings.py*.

```
TEMPLATES = [
 {
 "OPTIONS": {
 "context_processors": [
 "core.context_processors.core_context",
],
 },
],
]
```

**3.** In questo caso è stato utilizzato per sapere in quale club fa parte un utente che ha il privilegio di amministrare di club. Così da potere stampare il nome del club come segue:

```
{{ logged_in_user_member_club }} {% translate "Club Admin" %}
```

## E Risoluzioni Errori

### E.1 Risolvere un Circular Import

Per risolvere Questo problema, gli import problematici vengono spostati all'interno di una funzione, e vengono utilizzati solo all'interno di essa. Come in questo esempio, nel quale se importata dall'esterno la classe *Membership* creava questo tipo di errore.

```
def validator_membership_license_no(value):
 from .models import Membership

 if Membership.objects.filter(license_no=value,
transfer_date__isnull=True).exists():
 raise ValidationError("This license number is already taken.")
```

Nel contesto dell'esempio fornito, all'interno della funzione ***validator\_membership\_license\_no*** viene eseguito l'import della classe ***Membership*** dal modulo `.models` solamente quando la funzione viene effettivamente chiamata, evitando quindi un import circolare all'avvio del modulo, visto che la funzione viene utilizza all'interno dello stesso modulo dove è definita la classe *Membership*.

Quando la funzione *validator\_membership\_license\_no* è richiamata, la classe *Membership* viene importata al momento dell'esecuzione e utilizzata all'interno della stessa funzione. Poiché l'utilizzo della classe è circoscritto al contesto di questa funzione, non si verificano problemi di **import circolare**.

È importante sottolineare che questa tecnica risolve il problema di import circolare in scenari specifici in cui è necessario utilizzare classi o funzioni da moduli che potrebbero altrimenti causare tali problemi. Tuttavia, in progetti più complessi, potrebbero essere necessarie ulteriori strategie per affrontare problemi di import circolare su larga scala.

### E.2 Errori di migrazioni che sono insensati

Per affrontare errori di migrazione che non presentano apparentemente una logica, è essenziale iniziare con un'analisi accurata del problema. Se, dopo aver effettuato un'indagine approfondita, si determina che non esistono soluzioni alternative, può essere presa in considerazione questa soluzione.

Questa soluzione, sebbene sia stata dimostrata efficace in alcune circostanze, non garantisce successo in ogni situazione. La procedura in questione consiste nell'eliminare tutte le migrazioni del database precedentemente effettuate, per poi crearne una nuova che rappresenti lo stato attuale del database. Tale approccio può aiutare a risolvere conflitti o inconsistenze nelle migrazioni, ma è importante notare che potrebbe avere effetti collaterali, soprattutto quando si lavora allo sviluppo tra più persone.

- 1. Effettuare un Backup del Database:** conviene effettuare un backup dello stato attuale, se non è stato già fatto per evitare perdite.
- 2. Eliminare le Migrazioni Esistenti:** Rimuovere tutti i file di migrazione all'interno delle cartelle di migrazione di ciascuna app, escludendo i file `__init__.py`.

**3. Eliminare e Ricreare database:** Usare il comando *flush* per resettare il database e utilizzare i comandi *makemigrations* e *migrate* per generare e applicare nuove migrazioni che riflettano lo stato corrente del database.

```
python manage.py flush
python manage.py makemigrations
python manage.py migrate
```

Oppure utilizzare lo script fatto apposto per il reset del database, che oltre a resettare il database immette i dati di default:

```
python manage.py reset_database
```

## F Deployment

### F.1 Staticfiles

Per effettuare il deploy su **PythonAnywhere**, è stato necessario specificare l'*URL* per accedere ai file statici e il percorso della cartella che contiene questi file. Queste informazioni coincidono con quelle presenti nel file **settings.py** ed è importate che coincidono. Se non configurato correttamente, il sito apparirà senza alcuno stile, perdendo quindi la sua formattazione visuale.

1. Perciò come prima cosa è stata inserita la configurazione per i file statici, all'interno di **settings.py**:

```
STATIC_URL = "/static/"
STATICFILES_DIRS = [
 BASE_DIR / "staticfiles",
]

STATIC_ROOT = BASE_DIR / "production_staticfiles"
```

Il codice permette di raggruppare tutti i file statici all'interno di una singola cartella, ottimizzando la velocità di reperibilità dei file.

La variabile **STATIC\_URL** contiene l'*URL* base per i file statici, come *CSS*, *JavaScript* e immagini. Attualmente, è impostata su `"/static/"`.

La variabile **STATICFILES\_DIRS** è una lista di directory in cui Django cercherà i file statici durante lo sviluppo locale. Attualmente, è configurata per cercare la cartella **staticfiles** all'interno della directory principale (*BASE\_DIR*).

La variabile **STATIC\_ROOT** indica la directory in cui Django collezionerà tutti i file statici quando si esegue il comando di raccolta dei file statici per preparare l'applicazione per l'ambiente di produzione. Attualmente, è configurata per utilizzare la cartella **production\_staticfiles** all'interno della directory principale (*BASE\_DIR*).

2. Il comando di raccolta dei file statici viene eseguito ogni volta che si desidera effettuare il deploy di una nuova versione del sito:

```
python manage.py collectstatic
```

3. Infine, è stato necessario configurare l'applicazione web presente su PythonAnywhere in modo che sia consapevole della posizione dei file statici. Questo è stato ottenuto specificando l'*URL* per accedere ai file statici (**STATIC\_URL**) e il percorso della cartella che li contiene (**STATIC\_ROOT**).

Static files:

Files that aren't dynamically generated by your code, like CSS, JavaScript or uploaded files, can be served much faster straight off the disk if you specify them here. You need to **Reload your web app** to activate any changes you make to the mappings below.


URL	Directory	Delete
<a href="#">/static/</a>	<a href="#">/home/fstbadmin/gafst/production_staticfiles</a>	
<a href="#">Enter URL</a>	<a href="#">Enter path</a>	

Figura 13 - Configurazione Staticfiles PythonAnywhere

## F.2 Configurazione di Produzione

Quando si intende effettuare il deployment di un'applicazione Django, è fondamentale apportare alcune modifiche alle configurazioni presenti all'interno del file `settings.py`. Queste alterazioni sono necessarie per garantire che il sito, una volta reso accessibile, sia altamente sicuro e completamente operativo.

**Disabilitare modalità DEBUG:** È vitale impostare l'opzione `DEBUG` su `False` nel file di configurazione di Django (`settings.py`). Quando `DEBUG` è `True`, il sito mostra informazioni sensibili come errori dettagliati, che potrebbero essere sfruttati da un malintenzionato.

**Abilitare solo l'hostname del sito:** Si deve specificare una lista di indirizzi IP o nomi di dominio validi nella variabile `ALLOWED_HOSTS`. Questo impedisce gli attacchi HTTP Host header, garantendo che solo le richieste provenienti dagli host specificati vengano elaborate. Nel caso di questo progetto la variabile `ALLOWED_HOSTS` deve essere cambiata in questo modo:

```
ALLOWED_HOSTS = ["fstbadmin.pythonanywhere.com"]
```

## F.3 Creazione e Configurazione ambiente virtuale Python

Per eseguire il deploy su PythonAnywhere, è stato necessario creare un ambiente virtuale *Python*. Questo approccio consente l'installazione e l'utilizzo di pacchetti Python in un contesto isolato e controllato.

Si è optato per la configurazione di un ambiente virtuale basato su Python 3.10, la versione più aggiornata disponibile, allo scopo di sfruttare le ottimizzazioni e le funzionalità introdotte nelle release più recenti.

**1. Creazione ambiente virtuale:** per creare un ambiente virtuale utilizzando una console Bash, è stato sufficiente eseguire il seguente comando.

```
mkvirtualenv gafst --python=/usr/bin/python3.10
```

L'ambiente virtuale sarà generato all'interno della stessa directory in cui gli ambienti virtuali sono ospitati su PythonAnywhere, ovvero `"/home/fstbadmin/.virtualenvs/"`. Dove il nome *fstbadmin* è il nome del app.

**2.** Nella scheda "Web", nella sezione "Virtualenv", è stato configurato l'ambiente virtuale per l'applicazione web di PythonAnywhere inserendo il percorso:

```
"/home/myusername/.virtualenvs/gafst".
```

È possibile inserire solo il nome dell'ambiente virtuale ("gafst"), e il sistema dedurrà automaticamente il percorso completo dopo la conferma.

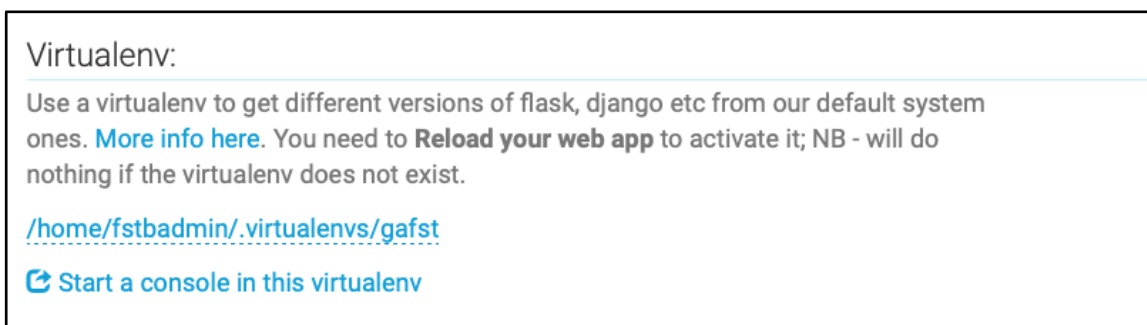


Figura 14 – PythonAnywhere Risultato finale della Configurazione Virtualenv

## F.4 Checklist

Questa *checklist* rappresenta una guida essenziale per il corretto **deploy** del sito su **PythonAnywhere**, garantendo l'esecuzione accurata di ogni *step* ed evitando l'omissione di alcun passaggio fondamentale.

- 1. Accedere al sito:** Come prima cosa bisogna accedere al sito di hosting, nel nostro caso **Pythonanywhere**, tramite l'URL all'indirizzo: <https://www.pythonanywhere.com/user/fstbadmin/>
- 2. Aprire Console:** Successivamente si deve aprire una console così che si possono eseguire tutti i comandi di *deployment*, per fare questo bisogna andare nella sezione corretta schiacciando il *tab* "Consoles", una volta all'interno della sezione giusta schiacciare su *Bash* per così aprire una nuova console Bash :

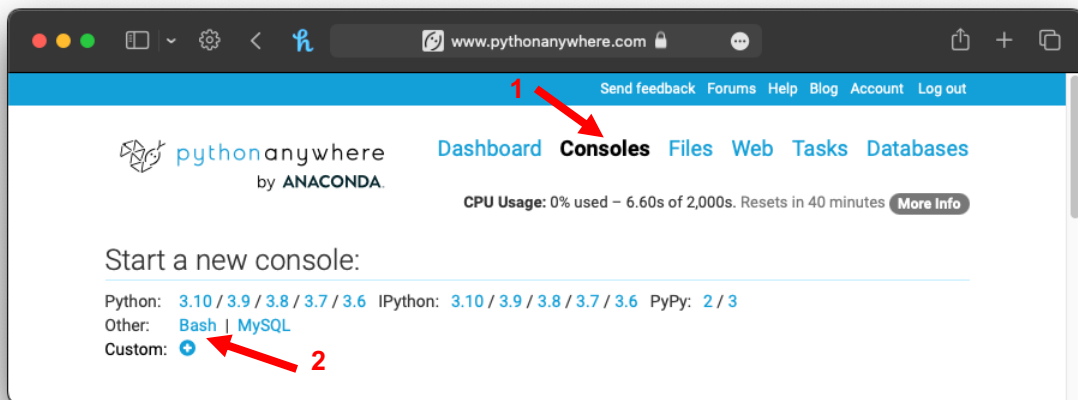


Figura 15 - PythonAnywhere Aprire Nuova Console

- 3. Preparare console:** Una volta aperta la console, il primo passo consiste nell'attivare l'ambiente virtuale di Python con il comando **workon gafst**. Successivamente, bisogna spostarsi all'interno della root del progetto usando il comando **cd gafst**.

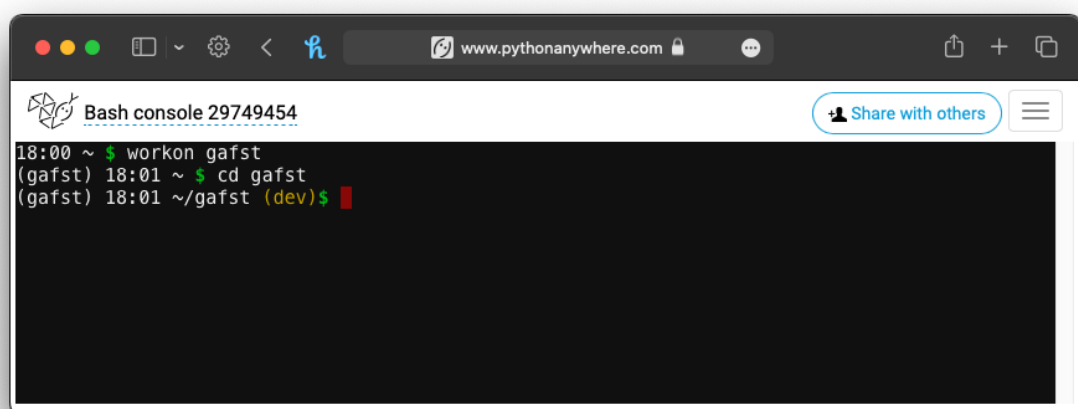


Figura 16 – PythonAnywhere Accesso al Ambiente Virtuale e alla root del progetto

Se l' ambiente virtuale python non è ancora stato creato bisogna crearlo e configurarlo come spiegato in precedenza.

**4. Verificare e aggiornare la repository git:** Una precauzione importante è assicurarsi di acquisire eventuali modifiche apportate al progetto tramite git, mediante l'utilizzo del comando *git pull*, nel caso in cui il progetto sia stato aggiornato.

Per verificare se una repository Git è aggiornata, è possibile utilizzare il comando *git fetch* seguito da *git status*.

Per eliminare le modifiche locali sulla repository, si può eseguire il comando *git checkout .* per eliminare tutte le modifiche non salvate o utilizzare *git checkout -- <nome\_file>* per ripristinare un file specifico alle sue condizioni precedenti.

**5. Raccolta file statici:** Dopo un aggiornamento della repository, è probabile che sia necessario raccogliere i file statici, per migliorare le prestazioni del sito in produzione. Se questa operazione non è mai stata eseguita in precedenza, si deve procedere alla raccolta dei file nella cartella “*STATIC\_ROOT*” utilizzando questo comando specifico:

```
python manage.py collectstatic
```

**6. Impostare configurazione di produzione:** Durante il passaggio alla produzione, è necessario effettuare alcune modifiche alle impostazioni rispetto all'ambiente di sviluppo. A tal scopo, è stato creato un file di configurazione aggiuntivo chiamato *production\_settings.py*, che contiene le impostazioni specifiche per l'ambiente di produzione della web app.

Per effettuare correttamente la transizione, bisogna sostituire il contenuto del file *settings.py* con quello di *production\_settings.py*. Questa operazione può essere eseguita mediante l'uso di un comando personalizzato chiamato “deploy”, costruito specificamente per l'ambiente Django.

Per avviare il comando deploy e inserire la configurazione della produzione, eseguire il seguente comando:

```
python manage.py deploy
```

**7. Reset del database con il comando deploy:** Se è necessario resettare il *database* basta effettuare l'esecuzione del comando *deploy*, durante la quale verrà chiesto se si vuole resettare il *database* o meno. Il reset viene effettuato eliminando e ricreando il *database* con e applicando di nuovo tutte le migrazioni (*python manage.py migrate*), e inserendo tutti i dati iniziali (*python manage.py insert\_defaults*).

Comando “deploy”, conferma reset database:

```
Do you want to reset the database? (y/n)
```



Per eseguire il reset si esegue il comando *reset\_database*, questo comando personalizzato esegue:

```
DROP DATABASE IF EXISTS <db_name>
CREATE DATABASE <db_name>

python manage.py makemigrations
python manage.py migrate

python manage.py insert_defaults
```

**8. Creazione Superuser:** Nel caso in cui il database sia stato resettato e siano stati creati un *superuser* o altri utenti, sarà necessario reinserirli. Durante l'esecuzione del comando *deploy*, verrà chiesto se si desidera effettuare il *reset* del database, e successivamente, se si vuole aggiungere un superuser. Questo passaggio è essenziale, poiché il superuser potrebbe essere stato eliminato durante il processo di reset del database e c'è il bisogno di avere un superuser per l'accesso al pannello admin di django.

Comando "deploy", conferma creazione superuser:

```
Do you want to create a superuser account? (y/n)
```

Comando eseguito per la creazione di un superuser:

```
python manage.py createsuperuser
```

**9. Dare al Superuser i permessi di FSTB Admin:** Per consentire al superuser di accedere alle pagine di amministrazione, così da creare i primi membri, è necessario seguire i seguenti passaggi:

- Accedere al pannello admin di Django all'indirizzo "<hostname\_sito>/admin" utilizzando le credenziali di amministratore.
- Una volta autenticato, selezionare la sezione "Utenti" cliccando sul tab "Users" nella barra laterale.
- Trovare l'utente superuser che di solito si chiama "admin" e aggiungere al suo profilo il gruppo "FSTB Admin" cliccando su di esso.
- Cliccare sulla freccia di aggiunta per confermare l'inclusione nel gruppo.
- Salvare le modifiche effettuate premendo il pulsante in fondo alla pagina.

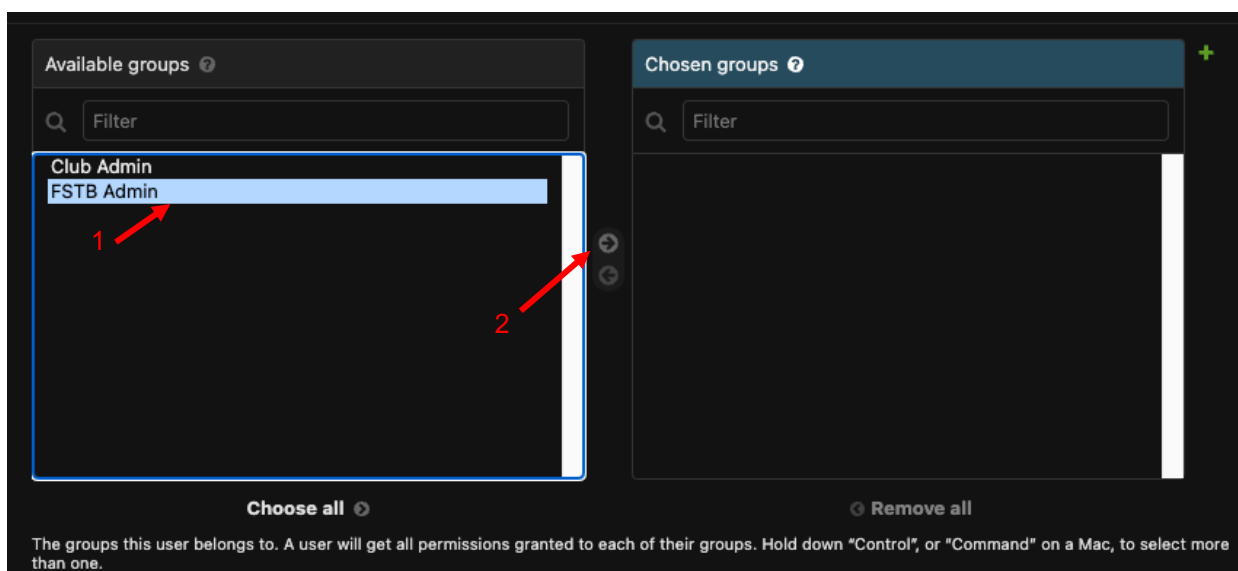


Figura 17 - Aggiunta Permesso FSTB Admin

Completando questi passaggi, l'utente superuser avrà i permessi necessari per accedere al sito con i diritti di FSTB admin.

**10. Verifica installazione dipendenze Python:** Un'altra precauzione fondamentale è verificare di avere installato tutte le dipendenze o pacchetti di Python elencati nel file "requirements.txt", poiché potrebbero esserci ancora pacchetti mancanti. È importante sottolineare che il comando "deploy" presente nel progetto Django si occupa automaticamente di gestire queste installazioni. Comunque, per fare questo basta eseguire il seguente comando:

```
pip install -r requirements.txt
```

**11. Verifica di Traduzione Delle Stringhe:** Un'altra cosa da verificare è che tutte le stringhe siano state tradotte, per fare questo basta eseguire il seguente *script* e andare a controllare sul pannello di *rosetta* se ci sono stringhe da tradurre.

```
python manage.py translate
```

**12. Riavviare Web App:** Infine, per completare il deployment, è necessario riavviare l'applicazione per garantire che stia funzionando con tutte le componenti aggiornate. Per fare ciò, basta accedere alla sezione **Web** e selezionare l'apposito tab, quindi premere il pulsante di ricaricamento.

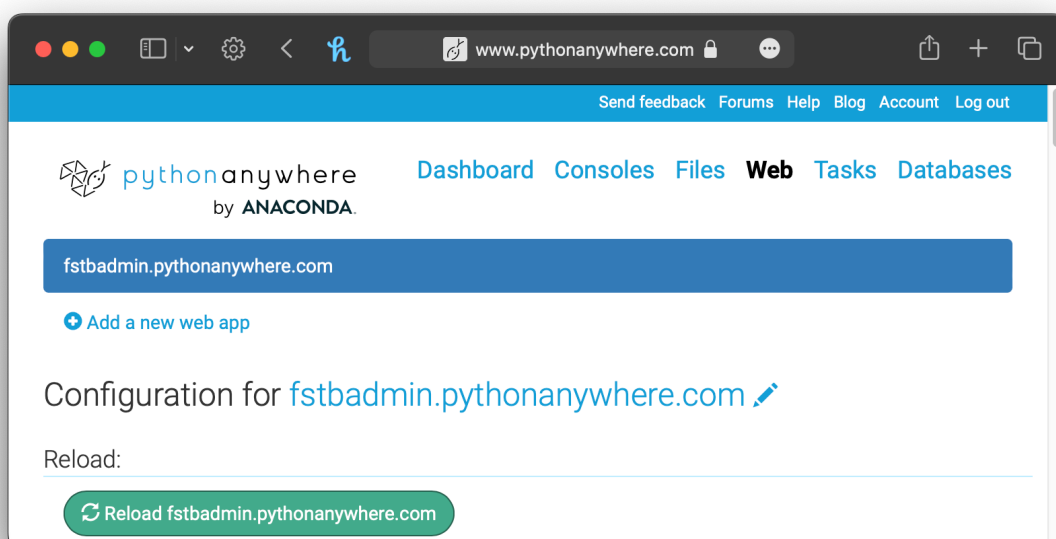


Figura 18 - PythonAnywhere Ricaricare Applicazione Web

# Glossario

**AJAX:** una tecnologia web utilizzata da HTMX che permette di effettuare richieste al server e ricevere dati in modo asincrono, senza dover aggiornare l'intera pagina. Ciò rende l'interazione utente più rapida e fluida.

**Assertion:** è una dichiarazione che verifica se una certa condizione è vera. Se l'*assertion* fallisce, il test segnalerà un errore, indicando che la condizione attesa non è stata soddisfatta. Le *assertions* sono utili per confermare che il comportamento del codice corrisponde a ciò che ci si aspetta, contribuendo a identificare errori e problemi nel software.

**Circular Import:** Nel contesto di Django, gli import circolari si verificano spesso quando due modelli presenti nello stesso file `models.py` fanno riferimento l'uno all'altro o quando modelli provenienti da diversi file `models.py` fanno riferimento tra di loro.

**CMS** (Content Management System): è un'applicazione software che consente agli utenti di creare, modificare e organizzare facilmente i contenuti di un sito web senza richiedere conoscenze tecniche avanzate. Fornisce un'interfaccia utente intuitiva e strumenti di gestione per aggiungere, modificare e pubblicare contenuti come testi, immagini e video. I CMS sono ampiamente utilizzati per gestire siti web di varie dimensioni e complessità, consentendo agli utenti di mantenere il controllo e l'aggiornamento dei propri contenuti.

**Cross-site scripting (XSS):** è una vulnerabilità di sicurezza in cui un'applicazione web permette l'esecuzione di codice di script maligno nei browser degli utenti. L'obiettivo è di rubare informazioni sensibili o compromettere il funzionamento del sito. Per prevenire l'XSS, è necessario validare e filtrare gli input e codificare correttamente i dati.

**Cross-site request forgery (CSRF):** è una vulnerabilità in cui un sito web malintenzionato sfrutta la fiducia di un utente autenticato su un altro sito per eseguire azioni indesiderate senza il suo consenso. L'attaccante inganna l'utente a eseguire azioni che sembrano legittime, consentendo di eseguire operazioni dannose. Per prevenire gli attacchi CSRF, è necessario implementare meccanismi di autenticazione dei token per verificare la legittimità delle richieste.

**CSS Transitions:** non un metodo per animare i cambiamenti di proprietà CSS su elementi web. Permettono di creare transizioni fluide tra diversi stati di un elemento senza l'uso di JavaScript, migliorando così l'esperienza utente.

**Decorator:** è una funzione speciale che viene applicata a una vista o a una funzione di test per estenderne o modificare il comportamento. I decorator nei test consentono di aggiungere funzionalità come l'autenticazione, la gestione delle autorizzazioni o la preparazione dell'ambiente prima dell'esecuzione dei test.

**Hostname:** Un hostname o nome di dominio è l'indirizzo univoco e facile da ricordare che viene utilizzato per identificare un sito web o un servizio su Internet.

**Keyword:** in Python è un termine riservato che ha un significato specifico nel linguaggio. Nelle funzioni, le keyword degli argomenti sono utilizzate per assegnare valori agli argomenti in modo esplicito, in modo che l'ordine degli argomenti non sia necessariamente importante. Questo permette di specificare solo gli argomenti che si desidera assegnare, evitando di dover ricordare la posizione esatta degli argomenti nella firma della funzione. Le keyword rendono il codice più leggibile e meno soggetto a errori.

**Middleware:** Un middleware è un componente software che agisce come un ponte o un intermediario tra differenti applicazioni, sistemi o servizi. Il middleware gestisce e controlla la comunicazione o l'interazione tra questi differenti componenti software.

Nel contesto di un framework web come Django, un middleware è un componente che processa le richieste e le risposte HTTP. Ogni volta che una richiesta HTTP viene inviata al tuo sito Django, Django la passa attraverso una serie di middleware per il processamento prima che la richiesta raggiunga la vista corrispondente. Allo stesso modo, quando una risposta viene inviata, Django la passa attraverso gli stessi middleware in ordine inverso.

**Mixins:** nel contesto di Django, sono classi che forniscono funzionalità aggiuntive e possono essere utilizzate per estendere le classi delle viste o dei modelli. I mixins sono una forma di composizione di classi, in cui è possibile riutilizzare il codice comune tra diverse classi senza dover ripetere la stessa implementazione.

**Mock:** è un oggetto simulato o fittizio creato per sostituire componenti reali durante i test. Questo consente di isolare il comportamento di una parte specifica del codice e verificare come interagisce con altre parti senza coinvolgere componenti complessi o dipendenze esterne reali.

**Path:** una "path" (o "percorso") è un indirizzo che indica la posizione di un file o di una cartella all'interno del sistema di archiviazione di un computer. Le path sono utilizzate per trovare, organizzare e collegare file e directory tra loro.

**Routing:** in Django si riferisce al processo di mappare gli URL alle azioni specifiche da eseguire all'interno di un'applicazione web. In sostanza, stabilisce quale vista o funzione deve essere chiamata quando un certo URL viene richiesto dal browser.

**RSS** (Rich Site Summary o Really Simple Syndication): è un formato standard per la distribuzione automatica di contenuti web aggiornati. Consente agli utenti di sottoscrivere ai feed RSS delle fonti di loro interesse e ricevere automaticamente gli ultimi articoli o notizie senza visitare manualmente ogni sito web. È una tecnologia ampiamente utilizzata per tenersi aggiornati sulle ultime novità online.

**Single-Page Application (SPA)**: è un tipo di applicazione web che carica dinamicamente tutto il suo contenuto su una singola pagina web, piuttosto che richiedere il caricamento di nuove pagine da parte del server. Utilizzando tecniche come l'AJAX, le SPA aggiornano solo le parti necessarie della pagina in risposta alle interazioni dell'utente, offrendo un'esperienza più fluida e reattiva. Questo approccio riduce i tempi di caricamento e migliora l'usabilità complessiva dell'applicazione.

**SQL injection**: è una vulnerabilità che si verifica quando un aggressore inserisce codice SQL dannoso all'interno di un input, permettendo loro di manipolare le query del database e ottenere accesso non autorizzato ai dati o compromettere il sistema. È una minaccia seria che richiede protezione adeguata.

**Stub**: in Python unittest, uno stub è una tecnica in cui si crea una versione semplificata di una funzione o di un oggetto per sostituire parti complesse o dipendenze esterne durante i test. Gli stub vengono utilizzati per controllare il comportamento di parti specifiche del codice e isolare il test su quella parte senza coinvolgere dettagli complicati o dipendenze reali.

**Throttling dei tentativi di accesso**: è una misura di sicurezza che limita il numero di tentativi di login falliti entro un certo periodo di tempo. Questo aiuta a prevenire attacchi di forza bruta e accessi non autorizzati.