Milli-Django: Documentazione Tecnica e Architetturale

1. Introduzione

Questo documento fornisce una panoramica dettagliata del progetto MilliPW-Django, una modernizzazione basata su Python di un'applicazione originale in PHP. L'obiettivo era ricostruire le funzionalità dell'applicazione su uno stack tecnologico robusto, sicuro e scalabile, sfruttando le pratiche di sviluppo moderne.

L'applicazione funge da pannello di controllo amministrativo per un sistema ospedaliero, consentendo agli utenti di visualizzare e gestire dati relativi a ospedali, pazienti (cittadini), condizioni mediche (patologie) e ricoveri.

Funzionalità Principali

- **Homepage:** Una pagina di benvenuto.
- **Ricerca e Visualizzazione:** Accesso in sola lettura con potenti funzionalità di ricerca per Cittadini, Patologie e Ricoveri.
- **CRUD Completo:** Funzionalità complete di Creazione, Lettura, Aggiornamento ed Eliminazione (CRUD) per gli Ospedali.
- **Relazioni tra Dati:** Visualizzazione delle connessioni tra le entità, come il numero di ricoveri per una patologia o le patologie specifiche associate a un ricovero.

Stack Tecnologico

- **Backend:** Python 3.11+ con il framework Django 5.2+.
- **Database:** PostgreSQL 15+.
- **Frontend:** HTML standard, CSS e JavaScript puro (vanilla).
- **Gestione dell'Ambiente:** `python-dotenv` per la configurazione e `venv` per l'isolamento delle dipendenze.

---

2. Progetto Architetturale

Il progetto è basato sul pattern **Model-View-Template (MVT)** di Django, che promuove una netta separazione delle responsabilità tra dati, logica e presentazione.

2.1. Pattern Model-View-Template (MVT)

- **Model (`hospital/models.py`):** Il layer dei dati. Definisce la struttura del database attraverso classi Python. L'Object-Relational Mapper (ORM) di Django utilizza questi modelli per eseguire query sul database in modo "Pythonico", astraendo il SQL grezzo.
- **View (`hospital/views.py`):** Il layer della logica. Le viste gestiscono le richieste HTTP in arrivo, interagiscono con i modelli per recuperare o manipolare i dati e passano tali dati a un template per il rendering.
- **Template (`hospital/templates/hospital/`):** Il layer di presentazione. I template sono file HTML che utilizzano il Template Language di Django (DTL) per visualizzare dinamicamente i dati forniti dalle viste.

2.2. Struttura del Codice e delle Directory

Il progetto è organizzato in componenti logici:

- `millipw_django/`: La directory di configurazione principale del progetto Django. Contiene `settings.py` e il file `urls.py` radice.
- `hospital/`: Un'"app" Django che incapsula tutte le funzionalità principali del sistema di gestione ospedaliera. Questo design modulare rende le funzionalità dell'app auto-contenute.
- `static/`: Contiene tutte le risorse frontend come CSS e JavaScript, che vengono servite direttamente al client.
- `templates/`: Contiene i template HTML utilizzati per il rendering dell'interfaccia utente.
- `db_scripts/`: Contiene script SQL supplementari, principalmente per il popolamento iniziale dei dati.

2.3. Modelli Dati e Relazioni (`hospital/models.py`)

La struttura dei dati è definita da cinque modelli chiave:

1. **`Cittadino`**: Rappresenta un cittadino/paziente. Utilizza il `cssn` (Codice Fiscale) come chiave primaria.
2. **`Ospedale`**: Rappresenta un ospedale. Presenta una relazione `OneToOneField` con `Cittadino` per il `codice_sanitario_direttore`. Questo impone la regola di business secondo cui una persona può essere direttore di al massimo un ospedale. La regola `on_delete=models.PROTECT` impedisce l'eliminazione di un `Cittadino` se è assegnato come direttore, garantendo l'integrità dei dati.
3. **`Patologia`**: Un semplice modello che rappresenta una patologia o condizione medica.
4. **`Ricovero`**: Rappresenta un ricovero ospedaliero. Collega gli altri modelli:
    - Una `ForeignKey` a `Cittadino` (il paziente).
    - Una `ForeignKey` a `Ospedale` (il luogo del ricovero).
    - Una `ManyToManyField` a `Patologia`, consentendo a un ricovero di essere associato a più patologie.
5. **`RicoveroPatologie`**: Questo è un modello "through" esplicito per la `ManyToManyField` tra `Ricovero` e `Patologia`. Rispecchia direttamente la tabella di giunzione del database originale, con una chiave primaria composita (`unique_together`) sulle chiavi esterne di `ricovero` e `patologia`.

Questa struttura, gestita dall'ORM di Django, consente di eseguire query in modo efficiente e intuitivo. Ad esempio, per ottenere tutte le patologie di un ricovero specifico, è sufficiente usare `istanza_ricovero.patologie.all()`.

2.4. Logica Applicativa e Viste (`hospital/views.py`)

Le viste sono basate su funzioni e gestiscono tutte le interazioni dell'utente:

- **Viste di Ricerca/Elenco (`cittadino_list_view`, `patologia_list_view`, `ricovero_list_view`):**
    - Queste viste recuperano tutti gli oggetti di un dato modello.
    - Costruiscono dinamicamente filtri utilizzando gli oggetti `Q` di Django basati sui parametri GET dei form di ricerca. Ciò consente criteri di ricerca flessibili e combinati (es. cercare un cittadino per nome e indirizzo).
    - **Ottimizzazione delle Prestazioni:** La `ricovero_list_view` utilizza `select_related('cittadino', 'ospedale')` e `prefetch_related('patologie')`. Questa è un'ottimizzazione critica che recupera gli oggetti correlati con un numero minimo di query al

database, prevenendo il "problema delle query N+1" che può rallentare gravemente le applicazioni. `select_related` funziona per relazioni one-to-one e foreign key (SQL JOIN), mentre `prefetch_related` funziona per relazioni many-to-many (query separate).

- **Vista CRUD (`ospedale_view`):**
  - Questa singola ed elegante vista gestisce l'elenco, la creazione e l'aggiornamento degli ospedali.
  - Se una chiave primaria (`pk`) viene passata nell'URL, la vista recupera l'istanza `Ospedale` corrispondente e opera in modalità "modifica".
  - Se non è presente alcuna `pk`, opera in modalità "creazione".
  - Questo approccio riduce la duplicazione del codice e mantiene la logica correlata in un unico posto.

- **Feedback all'Utente:** Le viste utilizzano il **Messages Framework** di Django (`messages.success`, `messages.error`) per fornire un feedback chiaro all'utente dopo un'azione (es. "Ospedale aggiunto con successo!").

2.5. Form e Validazione (`hospital/forms.py`)

- `OspedaleForm`: È un `ModelForm`, che genera automaticamente i campi del form basandosi sul modello `Ospedale`.
- **Validazione Personalizzata:** Il form include un metodo `clean_codice_sanitario_direttore`. Questa logica di validazione personalizzata controlla se il CSSN di un direttore è già assegnato a un altro ospedale, applicando il vincolo `OneToOne` a livello di applicazione prima che raggiunga il database. Ciò offre un'esperienza utente migliore, intercettando gli errori in anticipo.

2.6. Routing degli URL (`millipw_django/urls.py`, `hospital/urls.py`)

Il routing degli URL è disaccoppiato. Il file principale `millipw_django/urls.py` delega tutti gli URL specifici dell'applicazione a `hospital/urls.py` usando `include()`. Ciò rende l'app `hospital` portabile e auto-contenuta. Gli URL sono definiti con un `name`, il che permette di farvi riferimento programmaticamente nei template (es. `{% url 'ospedale_list' %}`), evitando URL codificati a mano (hardcoded).

2.7. Design del Frontend (`static/` e `templates/`)

- **Template:** Il frontend è renderizzato utilizzando il motore di template di Django.
  - **Ereditarietà:** Un template `base.html` definisce il layout comune (navigazione, footer). Altre pagine come `cittadini.html` e `ospedale.html` `{% extend %}`-ono questa base, sovrascrivendo blocchi specifici come `{% block content %}`. Questo approccio è DRY (Don't Repeat Yourself).
  - **Partials:** Elementi comuni dell'interfaccia utente come la barra di navigazione (`_nav.html`) e il footer (`_footer.html`) sono inclusi con `{% include %}`.
- **Stile:** Il CSS è usato per lo stile, con `style.css` che contiene le regole principali e `font.css` che importa i Google Fonts.
- **JavaScript (`static/js/script.js`):** Un singolo file JavaScript puro migliora l'esperienza utente. La sua funzione principale è implementare una **conferma di eliminazione a due clic**. Quando un utente clicca per la prima volta sull'icona di eliminazione, questa si trasforma in un segno di spunta. Un secondo clic entro 3 secondi conferma l'eliminazione; altrimenti, l'icona torna normale. Questo previene la perdita accidentale di dati.

3. Gestione del Database

3.1. Migrazioni dello Schema

Il progetto utilizza il sistema di migrazione integrato di Django per gestire lo schema del database.

- `python manage.py makemigrations`: Questo comando analizza `models.py` in cerca di modifiche e genera un nuovo file di migrazione nella directory `hospital/migrations/`.
- `python manage.py migrate`: Questo comando applica al database tutte le migrazioni non ancora applicate, creando o modificando le tabelle secondo necessità.

Questo sistema fornisce un controllo di versione per lo schema del database, rendendo facile tracciare le modifiche e collaborare.

3.2. Popolamento dei Dati (Seeding)

I dati iniziali vengono caricati nel database utilizzando un comando di gestione personalizzato:

- **`python manage.py seed_db`**: Questo comando legge il SQL grezzo da `db_scripts/load_postgres_data.sql` e lo esegue sul database configurato.
- **Motivazione:** La creazione di un comando di gestione automatizza il processo di seeding, rendendolo ripetibile e facile da integrare negli script di installazione. È più robusto rispetto a richiedere all'utente di eseguire manualmente un file SQL in un client di database separato. Il comando include anche un flag `--no-input` per bypassare la conferma dell'utente, essenziale per gli script automatizzati.

---

4. Installazione ed Esecuzione

Il progetto è progettato per essere facile da installare sia automaticamente (su Windows) sia manualmente.

4.1. Prerequisiti

- Python 3.11+
- PostgreSQL 15+ (il server deve essere in esecuzione)
- Un database PostgreSQL vuoto creato per il progetto.

4.2. Installazione Automatizzata (Windows)

Lo script `setup_and_run.bat` fornisce un processo di installazione interattivo e completamente automatizzato:

1. **Controllo Python:** Verifica che Python sia installato e presente nel PATH di sistema.
2. **Configurazione Database:** Chiede all'utente l'host, la porta, l'utente, il nome del database e la password di PostgreSQL.
3. **Creazione .env:** Genera un file `.env` con le credenziali fornite nel formato `DATABASE_URL` richiesto.
4. **Ambiente Virtuale:** Crea un ambiente virtuale Python in una cartella `venv/` per isolare le dipendenze.
5. **Installazione Dipendenze:** Installa tutti i pacchetti richiesti da `requirements.txt`.
6. **Migrazione Database:** Esegue `manage.py migrate` per creare le tabelle del database.
7. **Popolamento Database:** Esegue `manage.py seed_db` per popolare le tabelle.

8. **Avvio Server:** Esegue `manage.py runserver` per avviare il server di sviluppo.

4.3. Installazione Manuale (Tutte le Piattaforme)

1. **Clona/Scarica:** Ottieni i file del progetto.
2. **Ambiente Virtuale:** Crea e attiva un ambiente virtuale:

```
python -m venv venv
# Windows:
.\venv\Scripts\activate
# macOS/Linux:
source venv/bin/activate
```

3. **Installa Dipendenze:** `pip install -r requirements.txt`
4. **Configura Ambiente:** Copia `.env.example` in `.env` e inserisci la tua stringa di connessione `DATABASE_URL`.
5. **Migra Database:** `python manage.py migrate`
6. **Popola Dati:** `python manage.py seed_db`
7. **Esegui Server:** `python manage.py runserver`

4.4. Esecuzione dell'Applicazione

Dopo l'installazione iniziale, l'applicazione può essere avviata:

- **Windows:** Eseguendo lo script `run.bat`.
- **Tutte le Piattaforme:** Attivando l'ambiente virtuale ed eseguendo `python manage.py runserver`.

L'applicazione sarà disponibile all'indirizzo **`http://127.0.0.1:8000`**.