Di Napoli Nicholas **Matricola:0334000101**

Traccia d'Esame - Progetto di un Sistema Informativo per la piattaforma MySpider

Il seguente progetto riguarda l'analisi e la realizzazione di un sistema informativo per la piattaforma **"MySpider"**, un portale web ibrido pensato per la gestione e la condivisione di esperienze da parte di appassionati di tarantole.

Il progetto nasce con l'obiettivo di fornire uno strumento digitale moderno, funzionale e accessibile, che consenta agli utenti di documentare la crescita e la cura dei propri esemplari, promuovendo al contempo l'interazione tra membri della community.

L'interfaccia sarà semplice e intuitiva,costituita da un database relazionale (SQL) che sarà progettato per gestire le informazioni relative a:

- Dati personali degli utenti;
- Tarantole possedute;
- Eventi legati agli esemplari (muta, alimentazione, ecc.);
- Mi piace e commenti per evento;
- Seguaci e seguiti;
- Articoli pubblicati nella sezione "Biblioteca";
- Mi piace e commenti per articolo.

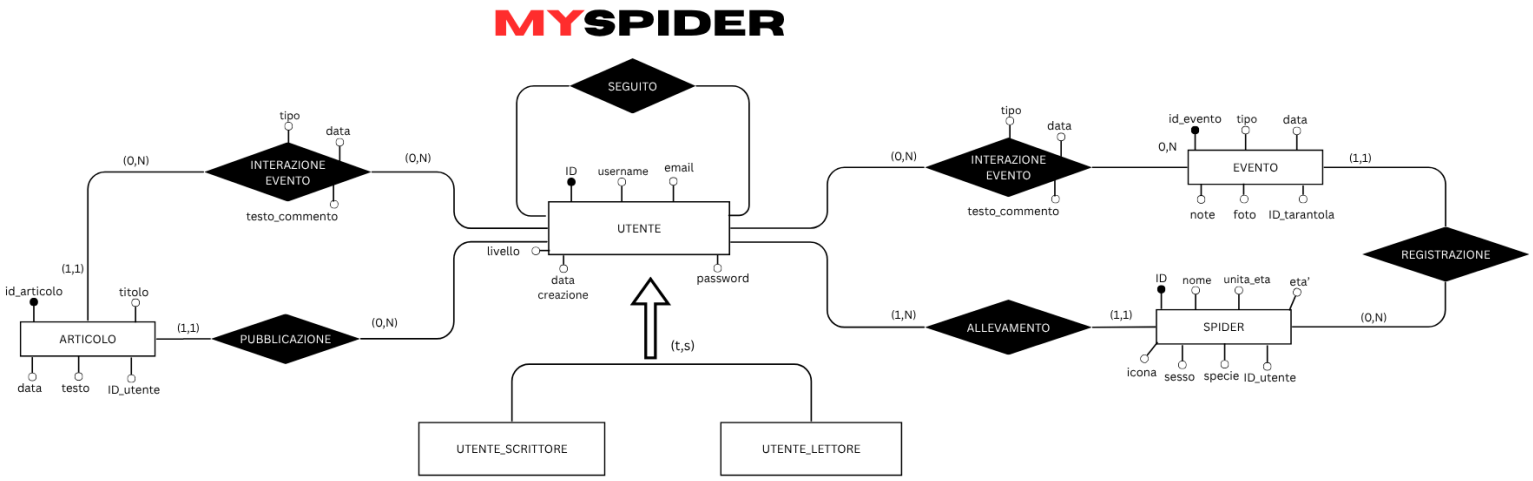
Il sistema distingue due categorie principali di utenti:

- **Utenti normali(lettori)**: potranno registrarsi, inserire esemplari, aggiornare i diari, commentare e mettere "mi piace" ai contenuti altrui;
- **Utenti avanzati (scrittori)**: raggiunto un certo livello, potranno contribuire attivamente scrivendo articoli per la community;

La progressione dell'utente avviene tramite un sistema di livello, che incrementa in base alla partecipazione attiva e alla qualità dei contributi. Ciò permette di accedere a funzioni aggiuntive, come la scrittura di articoli o la pubblicazione nella sezione "Biblioteca", riservata agli utenti più esperti.

Quest'ultima sezione e' consultabile da tutti e alimentata dagli utenti più esperti, dove saranno raccolti articoli, guide, consigli pratici e approfondimenti tematici riguardanti la cura e l'allevamento delle tarantole.

1. Analisi e progettazione concettuale: Modello E/R



Il primo modello E/R prevede una generalizzazione **totale e sovrapposta** dell'entità ***UTENTE*** in due sottoclassi logiche:

- ***UTENTE_LETTORE***: include tutti gli utenti che possono consultare eventi, articoli e contenuti della piattaforma;
- ***UTENTE_SCRITTORE***: rappresenta gli utenti abilitati anche alla pubblicazione di articoli o alla scrittura di post.

Tipo di generalizzazione:

- **Totale**: ogni istanza dell'entità **UTENTE** rientra obbligatoriamente in almeno una delle due sottoclassi.
- **Sovrapposta**: un utente può appartenere contemporaneamente a entrambe le categorie, ad esempio quando raggiunge un livello che consente di pubblicare ma continua a consultare i contenuti.

Strategie di Risoluzione della Generalizzazione

Le due strategie principali per la risoluzione della generalizzazione sono:

1. Strategia "Tutto nel padre"

In questa soluzione, si mantiene un'unica entità ***UTENTE*** che include **tutti gli attributi** e **l'informazione necessaria per distinguere i ruoli** (in questo caso, attraverso l'attributo **`livello`**).

- UTENTE(id, username, email, password, data_creazione, livello)
- Non esistono tabelle separate per ***UTENTE_LETTORE*** e ***UTENTE_SCRITTORE***.
- I privilegi sono assegnati dinamicamente in base al valore di **`livello`**.
- Le funzionalità disponibili (come la possibilità di pubblicare) sono gestite a livello applicativo.

Vantaggi:

- Schema semplificato.
- Nessuna ridondanza.
- Facilita la gestione di utenti che cambiano ruolo nel tempo (es. da lettore a scrittore).

Svantaggi:

- La distinzione formale tra i ruoli non è visibile nel modello concettuale se non attraverso la logica applicativa.

2. Strategia "Tutto nelle figlie"

Questa soluzione prevede la creazione di due entità separate ***UTENTE_LETTORE*** e ***UTENTE_SCRITTORE***, ciascuna collegata all'entità padre **`UTENTE`** tramite una relazione 1:1. Le tabelle figlie ereditano gli attributi del padre o mantengono solo il riferimento al padre.

UTENTE(id, username, email, password, data_creazione, livello)

UTENTE_LETTORE(id_utente) → FK verso UTENTE
UTENTE_SCRITTORE(id_utente) → FK verso UTENTE

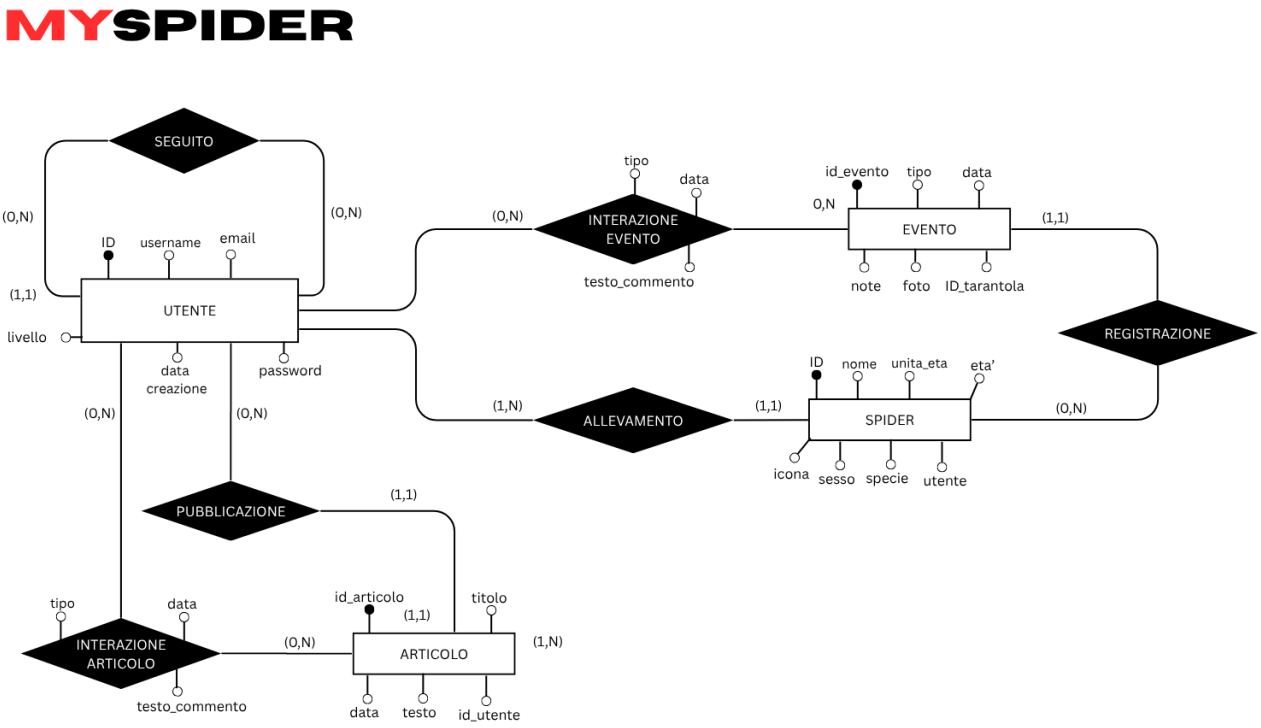
Vantaggi:

- La distinzione tra ruoli è esplicita nel modello e nella base di dati.
- Utile se esistessero attributi o relazioni esclusive per ciascun tipo di utente.

Svantaggi:

- Non adatta in assenza di attributi distintivi.
- Introduce complessità non necessaria.
- Poco flessibile nel caso in cui un utente debba evolvere da lettore a scrittore.

Strategia adottata



Nel progetto **MySpider**, è stata adottata la **strategia "Tutto nel padre"**. Questo perché:

- Non esistono attributi distintivi per ***`UTENTE_LETTORE`*** e ***`UTENTE_SCRITTORE`***;
- La **differenza tra i ruoli è dinamica** e dipende esclusivamente dal valore dell'attributo **`livello`**;
- Gli utenti possono **evolvere nel tempo** e acquisire nuovi privilegi in base alle interazioni sul sito;
- Il sistema è progettato per gestire i permessi direttamente a livello di applicazione in modo flessibile.

2. Progettazione Logica

La progettazione logica del database è stata sviluppata a partire dal modello concettuale E/R, adottando una **generalizzazione totale e sovrapposta** per l'entità **`Utente`**, che rappresenta gli utenti registrati al sistema. La generalizzazione non comporta differenze di attributi tra le specializzazioni (**`Lettore`** e **`Scrittore`**), ma si basa esclusivamente sull'attributo **`livello`**, che determina i permessi accessibili nel sistema (es. possibilità di pubblicare articoli).

`Utenti:``(id_utente, username UNIQUE, email UNIQUE, password, data_creazione, livello)`

- Rappresenta gli utenti registrati sulla piattaforma.

`Seguito:``(id_seguito, id_utente_seguace:Utenti, id_utente_seguito:Utenti)`

- Relazione di follow tra utenti.

`Spider:``(id, nome, specie, sesso, unita_eta, eta, icona, id_utente:Utenti)`

- Ogni tarantola è posseduta da un utente.

`Evento:``(id_evento, tipo, data, note, foto, id_spider:Spider)`

- Rappresenta eventi legati a una tarantola.

`Articolo:``(id_articolo, titolo, testo, data, id_utente:Utenti)`

- Articolo pubblicato da un utente.

`Interazione_Articolo:``(id_interazione, id_utente:Utente, id_articolo:Articolo, tipo, testo_commento, data)`

- Interazioni degli utenti con gli articoli (like o commenti).

`Interazione_Evento``:(id_interazione, id_utente:Utente, id_evento:Evento, tipo, testo_commento, data)`

- Interazioni degli utenti con gli eventi (like o commenti).

UTENTE

Nome campo	Descrizione	Tipo dati	Lunghezza	Vincoli
id_utente	Identificatore univoco utente	INT		PK, AUTO_INCREMENT
username	Nome utente	VARCHAR	50	NOT NULL, UNIQUE
email	Indirizzo email	VARCHAR	100	NOT NULL, UNIQUE
password	Password cifrata	VARCHAR	255	NOT NULL
data_signup	Data di registrazione	DATE		NOT NULL
livello	livello dell'utente	INT		NOT NULL,DEFAULT 1, CHECK (livello > 0)

Vincoli:

- **`username`** e **`email`** devono essere univoci.
- **`livello`** deve essere maggiore di 0; valore di default: 1.

SEGUITO

Nome campo	Descrizione	Tipo dati	Lunghezza	Vincoli
id_seguito	Identificatore relazione	INT		PK, AUTO_INCREMENT
id_utente_seguace	FK utente che segue	INT		NOT NULL, FK → Utente(id_utente)
id_utente_seguito	FK utente seguito	INT		NOT NULL, FK → Utente(id_utente), CHECK (id_utente_seguace <> id_utente_seguito)
(**UNIQUE coppia**)	Unicità seguace/seguito			UNIQUE(id_utente_seguace, id_utente_seguito)

Vincoli:

- Un utente non può seguire sé stesso: **`CHECK (id_utente_seguace <> id_utente_seguito)`**.
- Unicità della coppia **`(id_utente_seguace, id_utente_seguito)`**.

SPIDER

Nome campo	Descrizione	Tipo dati	Lunghezza	Vincoli
id_spider	Identificatore tarantola	INT		PK, AUTO_INCREMENT
nome	Nome assegnato	VARCHAR	50	NOT NULL
specie	Specie della tarantola	VARCHAR	100	NOT NULL
sesso	Sesso (M/F)	CHAR	1	NULLABLE, CHECK (sesso IN ('M','F'))
unita_eta	Unità dell'età	VARCHAR	10	NULLABLE, CHECK (unita_eta IN ('giorni', 'mesi', 'anni'))
eta	Età numerica	INT		NULLABLE, CHECK (eta ≥ 1)
icona	Percorso/URL immagine	VARCHAR	255	NOT NULL
id_utente	FK proprietario	INT		NOT NULL, FK → Utente(id_utente)

Vincoli:

- **`sesso`**, **`unita_eta`**, **`eta`** sono opzionali (NULLABLE).
- **`sesso`**, se specificato, deve appartenere all'insieme **`{'M', 'F'}`.**

- **`unita_eta`**, se specificata, deve essere in **`{'giorni', 'mesi', 'anni'}`**.
- **`eta`**, se specificata, deve essere ≥ 1.
- **`icona`** è obbligatoria (NOT NULL).
- **`id_utente`** è chiave esterna verso **`Utente`**.

EVENTO

Nome campo	Descrizione	Tipo dati	Lunghezza	Vincoli
id_evento	Identificatore evento	INT		PK, AUTO_INCREMENT
tipo	Tipo evento	VARCHAR	20	NOT NULL, CHECK (tipo IN ('alimentazione','muta','accoppiamento',’morte’,'altro'))
data	Data evento	DATE		NOT NULL
note	Annotazioni	TEXT		NULLABLE
foto	Foto evento	VARCHAR	255	NULLABLE
id_spider	FK Spider	INT		NOT NULL, FK → Spider(id_spider)

Vincoli:

- **`tipo`** deve appartenere all’insieme **`{'alimentazione', 'muta', 'accoppiamento', 'morte', 'altro'}`**.
- **`note`** e **`foto`** sono opzionali (NULLABLE).
- **`id_spider`** è chiave esterna verso **`Spider`**.

INTERAZIONE EVENTO

Nome campo	Descrizione	Tipo dati	Lunghezza	Vincoli
id_interazione	Identificatore interazione	INT		PK, AUTO_INCREMENT
id_utente	FK utente	INT		NOT NULL, FK → Utente(id_utente)
id_evento	FK evento	INT		NOT NULL, FK → Evento(id_evento)
tipo	Tipo interazione	VARCHAR	10	NOT NULL, CHECK (tipo IN ('like','commento'))
testo_commento	Contenuto commento	TEXT		NULLABLE se tipo='like', NOT NULL se tipo='commento'
data	Data interazione	DATE		NOT NULL
(UNIQUE LIKE)	Like univoco			UNIQUE(id_utente, id_evento) WHERE tipo = 'like' (vincolo parziale)

Vincoli:

- **`tipo`** deve appartenere all'insieme **`{'like', 'commento'}`**.
- Se **`tipo = 'commento'`**, allora **`testo_commento`** `IS NOT NULL`.

- Se **`tipo = 'like'`**, allora **`testo_commento`** `IS NULL`.
- Un utente può mettere **al massimo un like per evento**: vincolo `UNIQUE` **`(id_utente, id_evento)`** dove **`tipo = 'like'`**.

ARTICOLO

Nome campo	Descrizione	Tipo dati	Lunghezza	Vincoli
id_articolo	Identificatore articolo	INT		PK, AUTO_INCREMENT
titolo	Titolo dell'articolo	VARCHAR	100	NOT NULL
testo	Contenuto completo	TEXT		NOT NULL
data	Data pubblicazione	DATE		NOT NULL
id_utente	FK utenti	INT		NOT NULL, FK → Utente(id_utente)

INTERAZIONE ARTICOLO

Nome campo	Descrizione	Tipo dati	Lunghezza	Vincoli
id_interazione	Identificatore interazione	INT		PK, AUTO_INCREMENT
id_utente	FK utente	INT		NOT NULL, FK → Utente(id_utente)
id_articolo	FK articolo	INT		NOT NULL, FK → Articolo(id_articolo)
tipo	Tipo interazione	VARCHAR	10	NOT NULL, CHECK (tipo IN ('like','commento'))
testo_commento	Contenuto commento	TEXT		NULLABLE se tipo='like', NOT NULL se tipo='commento'
data	Data interazione	DATE		NOT NULL
(UNIQUE LIKE)	Like univoco			UNIQUE(id_utente, id_articolo) WHERE tipo = 'like' (vincolo parziale)

Vincoli:

- **`tipo`** deve appartenere all'insieme **`{'like', 'commento'}`**.
- Se **`tipo = 'commento'`**, allora **`testo_commento IS NOT NULL`**.
- Se **`tipo = 'like'`**, allora **`testo_commento IS NULL`**.
- Un utente può mettere **al massimo un like per articolo**: vincolo `UNIQUE` (**`id_utente, id_articolo`**) dove **`tipo = 'like'`**.

Vincoli interrelazionali

- **Pubblicazione articoli riservata ad utenti esperti**

Un utente può pubblicare un articolo solo se il suo livello >= 10

- **Un solo like per articolo**

Ogni utente può mettere al massimo un like per articolo, ma può scrivere più commenti.
→ Coinvolge: Interazione_Articolo.
→ Gestibile con vincolo UNIQUE parziale su (id_utente, id_articolo) dove tipo = 'like'.

- **Un solo like per evento**

Stesso vincolo del punto precedente, ma riferito agli eventi.
→ Coinvolge: Interazione_Evento.
→ Anche qui: UNIQUE parziale su (id_utente, id_evento) per tipo = 'like'.

- **Divieto di auto-follow**

Un utente non può seguire sé stesso.
→ Coinvolge: Seguito, con due FK sulla stessa tabella Utente.
→ Gestito con vincolo CHECK (id_utente_seguace <> id_utente_seguito).

- **Cancellazione a cascata di tarantole ed eventi**

Se un utente viene eliminato, devono essere eliminati automaticamente tutti gli spider associati a quell'utente e tutti gli eventi collegati a quegli spider.

→ Coinvolge: *Utenti*, *Spider*, *Evento*.
→ Implementabile tramite **`ON DELETE CASCADE`** sulle **FK `Spider.id_utente`** e **`Evento.id_spider`.**

- **Cancellazione a cascata di eventi associati a uno spider**

Se uno spider viene eliminato, devono essere eliminati automaticamente tutti gli eventi collegati a quello spider.

→ Coinvolge: *Spider*, *Evento*.
→ Implementabile tramite **`ON DELETE CASCADE`** sulla FK **`Evento.id_spider`**.

3. Implementazione del sistema informativo

L'implementazione del sistema informativo MySpider è stata sviluppata utilizzando il framework Django, selezionato per la sua scalabilità, sicurezza e semplicità di manutenzione.

Il sistema organizza i dati secondo il modello logico definito, garantendo integrità e coerenza, e risponde efficacemente alle esigenze della community, assicurando correttezza nelle interazioni tra gli utenti.

✦ Funzionalità principali del sistema MySpider

1. Registrazione e accesso sicuro

- MySpider offre un sistema di autenticazione semplice e protetto.
  Durante la registrazione, l'utente deve inserire una password che contenga almeno 8 caratteri, un numero e un carattere speciale.
- Le credenziali vengono criptate con algoritmo SHA-256 per garantire sicurezza.
- Ogni nuovo utente parte da **livello 1**.
- Il login è rapido e permette di accedere al proprio profilo in pochi secondi.

2. Diario personale e gestione tarantole

- Ogni utente ha un diario dove può inserire, modificare e visualizzare i propri esemplari.
- Il form consente di aggiungere nome, specie, sesso, età, unità di misura e icona.
- A ogni nuova tarantola creata, l'utente guadagna livello.
- Se una tarantola viene eliminata, anche tutti i suoi eventi vengono cancellati in automatico.

3. Eventi associati agli esemplari

- È possibile registrare eventi per ogni tarantola, come mute, alimentazioni o note particolari.
- Gli eventi sono inseriti tramite un modulo e visibili anche in una sezione dedicata.
- Ogni evento creato determina un aumento di livello.
- Gli eventi possono essere eliminati quando necessario.

4. Biblioteca e articoli della community

- Tutti possono leggere gli articoli pubblicati nella sezione Biblioteca.
- Solo chi ha **livello ≥ 10** può scrivere nuovi articoli, tramite un semplice form con titolo e contenuto. Ogni articolo mostra autore e data di pubblicazione.
- L'autore può eliminare solo i propri articoli: il pulsante "Elimina" appare solo se si è loggati come autore.

5. Like e commenti agli articoli

- Gli articoli possono ricevere like e commenti da qualsiasi utente.
- Ogni interazione fa aumentare il livello sia di chi commenta/mette like, sia di chi ha scritto l'articolo, incentivando la partecipazione attiva.

6. Follow e feed aggiornato

- Gli utenti possono seguire altri membri.
- Nel feed personale compaiono le nuove tarantole e gli eventi degli utenti seguiti, rendendo facile rimanere aggiornati sulle attività della propria rete.

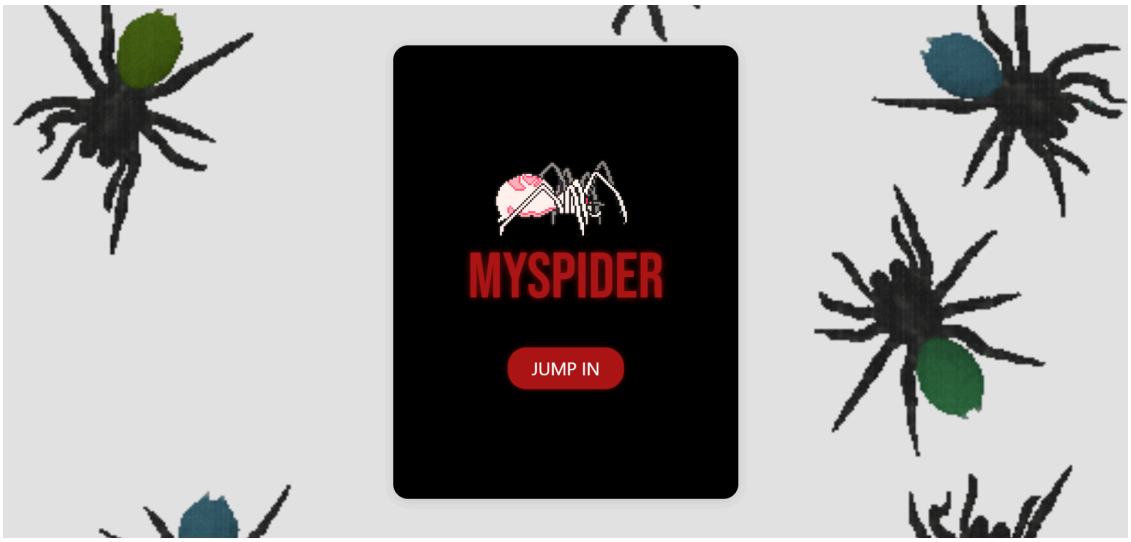
7. Interazioni sugli eventi

- Anche gli eventi possono essere commentati o apprezzati con un like.
- Come per gli articoli, ogni interazione fa salire il livello sia di chi interagisce che di chi riceve.

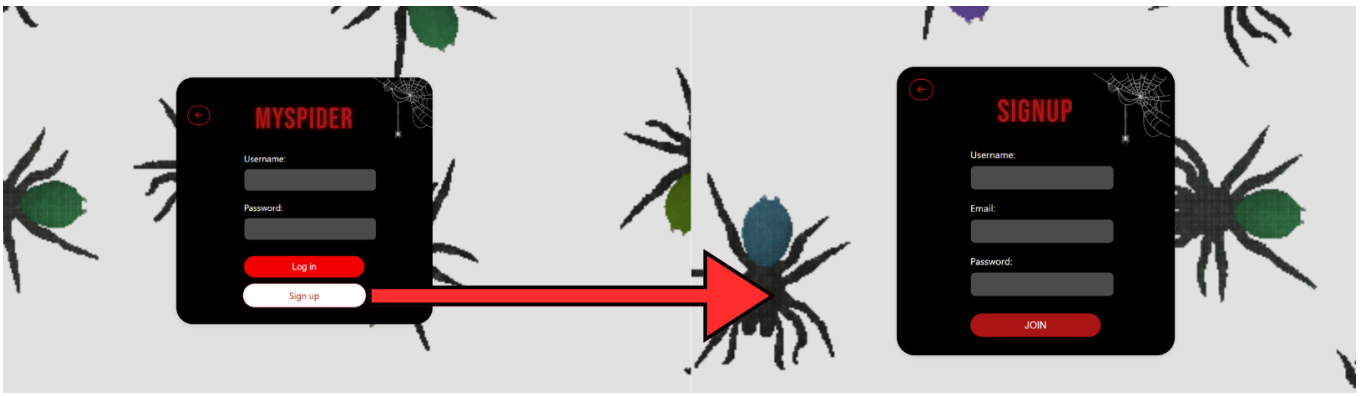
8. Ricerca utenti e articoli

- La sezione "Cerca" permette di trovare facilmente articoli nella Biblioteca o altri utenti, rendendo più semplice scoprire contenuti interessanti e fare nuove connessioni.

LOGIN/SIGNUP



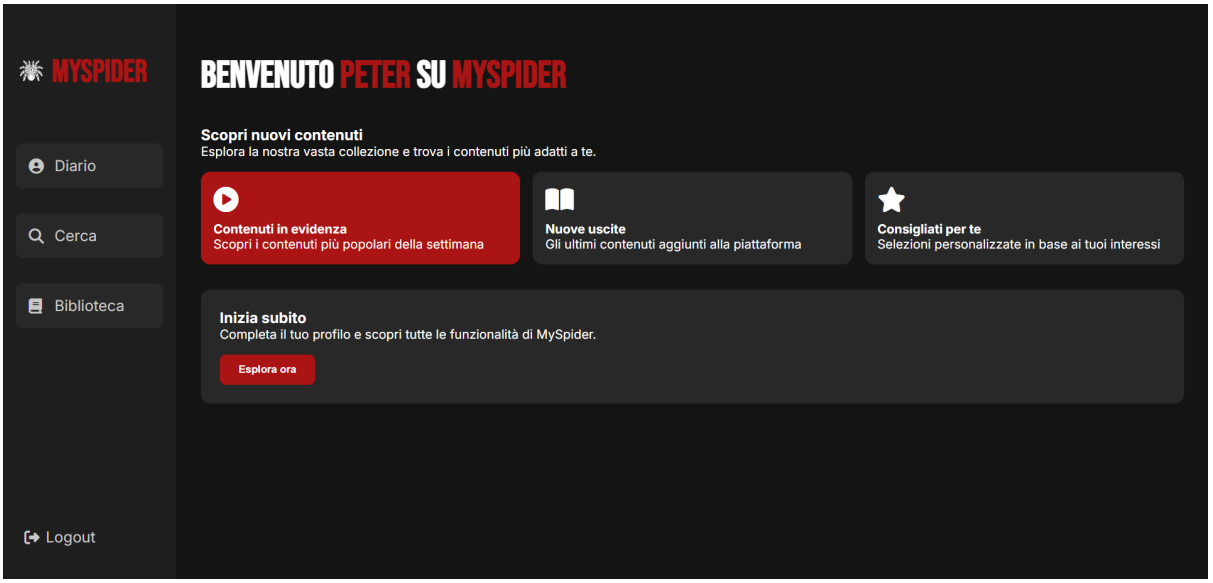
- La prima schermata che si presenta all'utente offre un pulsante che invita a entrare direttamente nell'esperienza di **MySpider**.



Cliccando su questo pulsante si viene reindirizzati alla pagina di **login**.

- Se si è già registrati, basta inserire le proprie credenziali per accedere al sistema.
- In caso contrario, è possibile procedere con la registrazione cliccando sul pulsante "**Signup**", che porta alla pagina dedicata all'iscrizione.

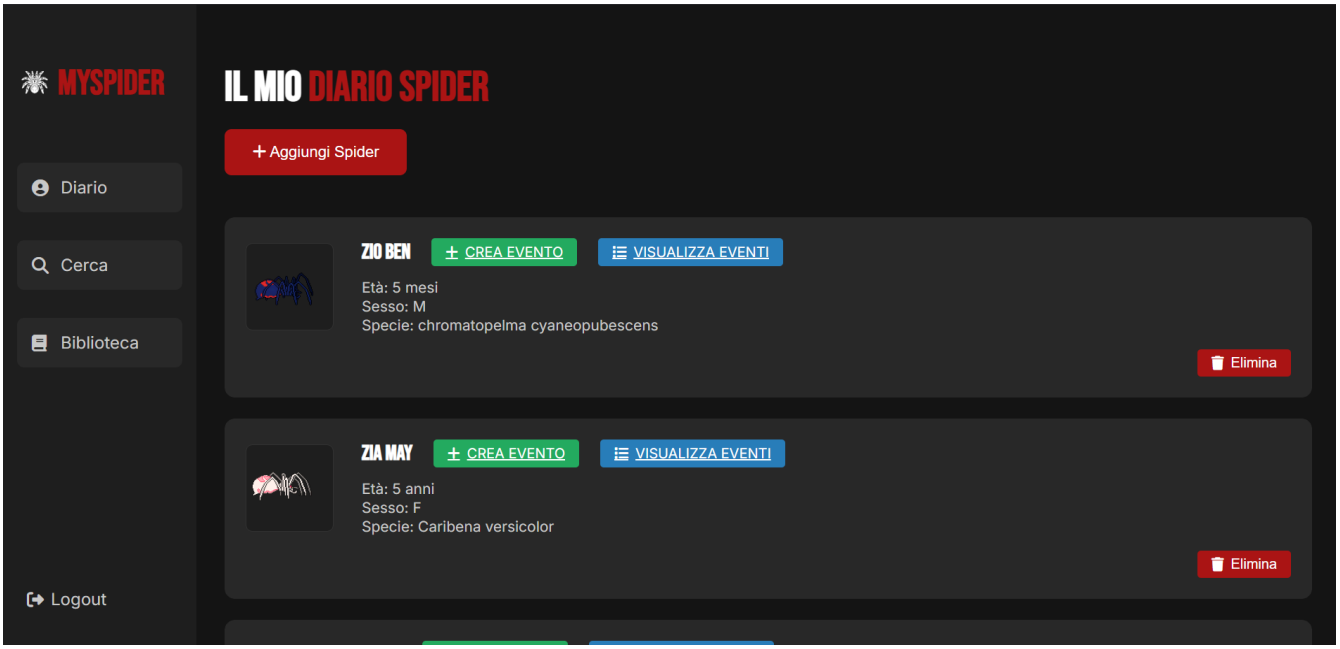
ACCOUNT



Dopo aver effettuato l'accesso, l'utente viene portato alla pagina "**Account**".

- Nella parte destra dello schermo, si trova la sezione "**Feed**", che mostra contenuti in evidenza e le attività degli utenti e delle tarantole seguite(non implementato totalmente).
- In basso a sinistra, invece, è presente il pulsante "**Logout**" per disconnettersi.
- A sinistra si trovano le sezioni **Diario**, **Cerca** e **Biblioteca**

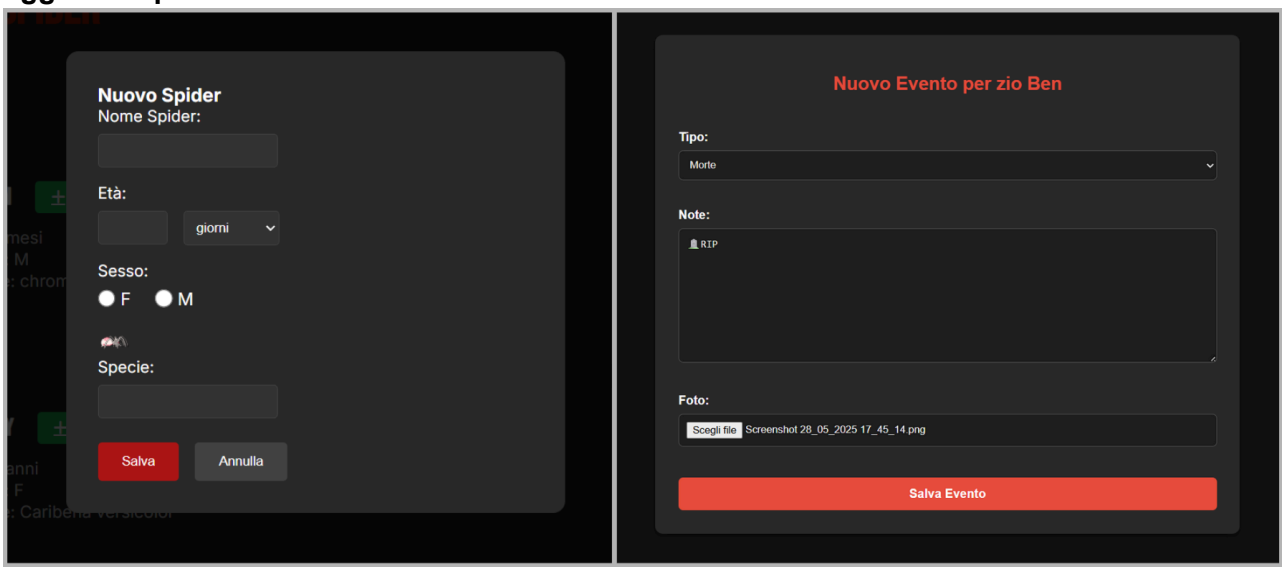
DIARIO



La sezione "**Diario**" consente di gestire completamente le proprie tarantole e i relativi eventi, con diverse funzionalità:

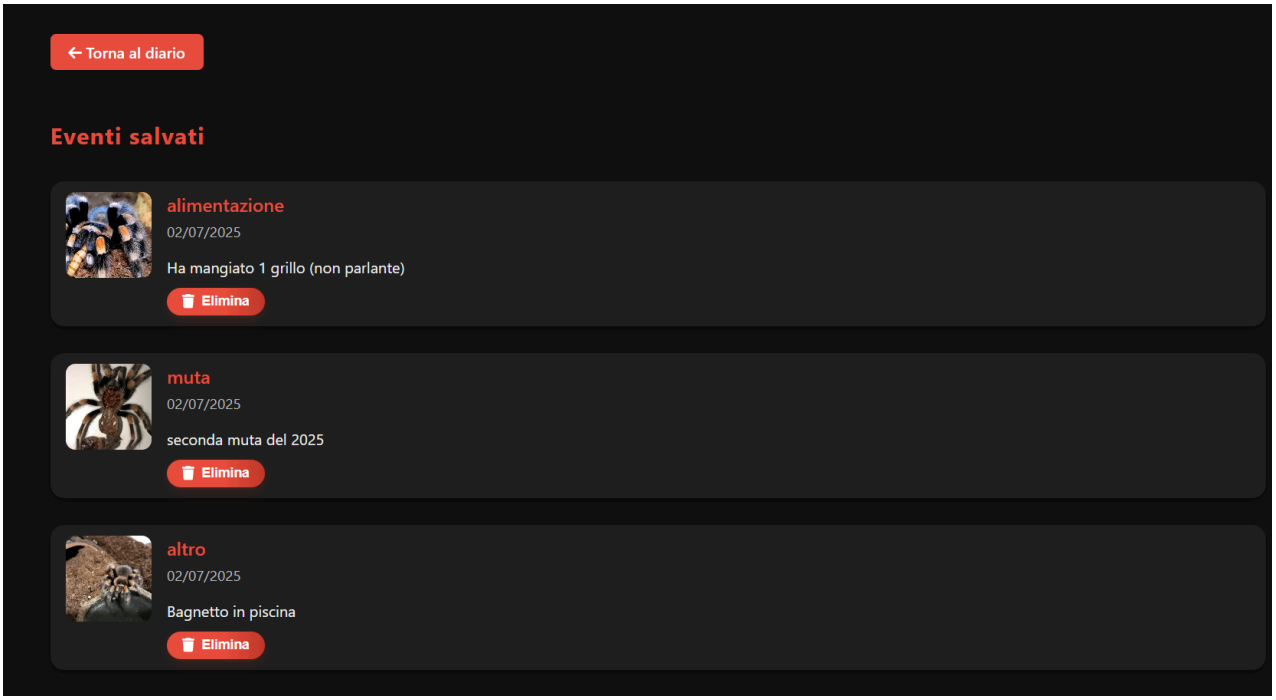
1. Aggiungere nuove tarantole tramite il pulsante “**Aggiungi Spider**” in alto
2. Visualizzare l’elenco delle proprie tarantole
3. Creare nuovi eventi associati agli esemplari
4. Consultare tutti gli eventi inseriti
5. Eliminare una tarantola insieme ai suoi eventi

Aggiunta Spider e Creazione evento



- L’aggiunta di una nuova tarantola o di un evento avviene tramite due moduli che si aprono cliccando sui rispettivi pulsanti.
- Le tarantole create appaiono nella schermata “**Diario**”

EVENTI



Gli eventi sono visibili in una pagina dedicata.

- Ogni evento mostra: il tipo di evento, un'immagine, la data di creazione e una descrizione
- È possibile eliminarli tramite un apposito pulsante.

BIBLIOTECA

La sezione **Biblioteca** raccoglie gli articoli pubblicati dagli utenti con i requisiti necessari.

- Il pulsante "**Crea articolo**" in alto a destra è visibile solo agli utenti con livello pari o superiore a 10.
- Il pulsante "**Elimina**" compare esclusivamente sugli articoli scritti dall'utente attualmente loggato.

ARTICOLI

- Cliccando su un articolo, questo si apre a schermo intero per una lettura comoda.
- Al termine, è possibile tornare facilmente alla sezione Biblioteca per esplorare altri contenuti.

Gestione delle sessioni e controllo accessi

Nel progetto **MySpider**, la gestione dello stato di autenticazione degli utenti è basata sul sistema di sessioni fornito da Django.

Le **sessioni** consentono di memorizzare dati persistenti tra una richiesta HTTP e l'altra, permettendo di mantenere attivo lo stato di login dell'utente durante la navigazione.

Quando un utente effettua il login, nel dizionario **`request.session`** vengono salvate due informazioni principali:

- **`logged_in`**: flag booleano che indica se l'utente è autenticato (**`True` o `False`**)
- **`username`**: nome utente dell'account connesso

Esempio di impostazione delle variabili di sessione durante il login:

```
request.session['logged_in'] = True
request.session['username'] = username
```

Questi dati rimangono disponibili per tutta la durata della sessione, che termina con il logout tramite chiamata a:

```
request.session.flush()
```

che cancella tutte le informazioni memorizzate nella sessione, invalidandola.

Per garantire la sicurezza e impedire accessi non autorizzati, ogni view protetta verifica la presenza del flag `logged_in` nella sessione:

```
logged_in = request.session.get('logged_in', False)
if not logged_in:
    return redirect('login')
```

Se il flag è assente o `False`, l'utente viene reindirizzato alla pagina di login, bloccando così l'accesso manuale tramite URL a contenuti riservati.

Blocco tentativi di login e protezione da attacchi brute force

È stata implementata una **logica di limitazione dei tentativi di accesso** per rafforzare la sicurezza dell'autenticazione e prevenire attacchi brute force e a dizionario. Il sistema utilizza la sessione dell'utente (**`request.session`**) per tracciare:

- **`login_attempts`**: numero di tentativi di login falliti.
- **`blocked_until`**: timestamp oltre il quale sarà nuovamente consentito il tentativo di accesso.

Funzionamento:

- Dopo **3 tentativi falliti consecutivi**, l'utente viene **temporaneamente bloccato**.
- La durata del blocco è di **15 minuti** dal momento dell'ultimo tentativo errato.
- Durante il periodo di blocco, **qualsiasi tentativo di accesso viene respinto** con un messaggio di avviso.
- Dopo il termine, il contatore viene **azzerato automaticamente**.

Vantaggi in termini di sicurezza:
Questa funzionalità:

- **Mitiga attacchi automatizzati** (brute force/dictionary attacks), impedendo numerosi tentativi consecutivi.
- Introduce un **ritardo forzato** (rate limiting) che rende inefficiente qualunque attacco iterativo.
- Non richiede storage persistente o modifica del database, essendo **gestita lato sessione**, rendendo l'implementazione leggera e compatibile con altri meccanismi di autenticazione.

Simulazione di SQL Injection

Supponiamo di avere un codice che gestisce il login di un utente costruendo manualmente la query SQL in questo modo:

```python
from django.db import connection

def login_vulnerabile(request):
    if request.method == "POST":
        username = request.POST["username"]
        password = request.POST["password"]
        with connection.cursor() as cursor:
            # Costruzione pericolosa della query concatenando direttamente l'input utente
            query = f"SELECT * FROM utenti WHERE username = '{username}' AND password = '{password}'"
            cursor.execute(query)
            user = cursor.fetchone()
            if user:
                # login riuscito
                pass
            else:
                # login fallito
                pass
```

Come avviene l'attacco

- Un utente malintenzionato inserisce come username:
  `admin' OR '1'='1`

- E come password:
  `qualunque_cosa`

La query risultante diventa:

SELECT * FROM utenti WHERE username = 'admin' OR '1'='1' AND password = 'qualunque_cosa'

Per la logica SQL, `'1'='1'` è sempre vera, quindi la query ritorna tutti gli utenti o almeno uno, permettendo l'accesso senza conoscere la password corretta.

Vulnerabilità introdotte

- **Concatenazione diretta**: L'input dell'utente viene inserito direttamente nella stringa SQL senza alcuna verifica o sanitizzazione.

- **Nessuna separazione dati/codice**: I dati (input utente) e il codice SQL sono mescolati, così un input malevolo diventa parte del codice eseguibile.

- **Possibilità di manipolazione della query**: L'attaccante può modificare la logica della query a suo favore.

Test dell’iniezione con SQLmap

Strumenti come **SQLmap** automatizzano gli attacchi SQL Injection. Se un’applicazione contiene una vulnerabilità come quella descritta sopra, SQLmap è in grado di:

- Individuarla analizzando la risposta del server a diversi payload.
- **Dumpare dati sensibili** (utenti, password, tabelle, ecc.)
- Effettuare escalation come accesso ai DBMS, file di sistema, ecc.

Esempio di utilizzo

Supponiamo che il form di login vulnerabile invii una richiesta **P0ST** come questa

POST /login HTTP/1.1
Host: localhost:8000
Content-Type: application/x-www-form-urlencoded
username=admin&password=1234

Salvando questa richiesta in un file **(`req.txt`),** SQLmap può essere lanciato così:

sqlmap -r req.txt --risk=3 --level=5 --dump

`-r req.txt`

- **Carica** una **richiesta HTTP completa** da un file (POST, headers, cookie, ecc.).
- Pratico quando il login ha **token o header complessi**.

`--risk=3`

- Imposta il **grado di rischio** dei test (1=basso, 2=medio, 3=alto).
- 3 esegue payload più aggressivi e dettagliati, utili in ambiente di test.

`--level=5`

- Definisce il **livello di profondità** dell’attacco (1–5).
- 5 effettua controlli su **tutti i parametri possibili** (GET, POST, headers, cookie…)

`--dump`

- Se SQLmap trova una vulnerabilità, **estrae i dati** dal database (tabelle, record, ecc.)

Risultato: Se la query è vulnerabile, l'attaccante ottiene l’accesso o scarica l’intero database.

Soluzioni SQL Injection

Uso dell'ORM Django

Django fornisce un Object Relational Mapper (ORM) che permette di interagire con il database usando metodi e oggetti Python, senza scrivere query SQL manuali.

Esempio sicuro equivalente al codice precedente:

try:

user = utenti.objects.get(username=username, password=password)

except utenti.DoesNotExist:

user = None

E' sicuro perche':

- L'ORM gestisce automaticamente l'escaping e la sanitizzazione degli input.
- I valori passati a **`get()`** non vengono mai interpretati come codice SQL, ma solo come dati da cercare.
- Anche se un utente inserisse input malevoli, questi verrebbero trattati come semplici stringhe, senza influenzare la query.

Uso di query parametrizzate

Se si deve per forza scrivere query SQL manualmente, è fondamentale usare query parametrizzate per separare il codice dai dati.

```
from django.db import connection

def login_sicuro(request):
    if request.method == "POST":
        username = request.POST["username"]
        password = request.POST["password"]
        with connection.cursor() as cursor:
            query = "SELECT * FROM utenti WHERE username = %s AND password = %s"
            cursor.execute(query, [username, password])
            user = cursor.fetchone()
            # resto del codice
```

E' sicuro perche':

- I parametri (**`username`** e **`password`**) sono passati separatamente e l'adapter del database si occupa di inserirli correttamente senza esporre la query al rischio di manipolazione.
- Nessun input utente può alterare la struttura della query.