IDEARIUM

Progetto per l'esame di Basi di Dati, programma dell'a.a. 2023/2024.

A cura di **Filippo di Falco** (mat. XXXXXX) e **Lorenzo Calvisi** (mat. XXXXXX)

Linee generali

Idearium è un piattaforma di blogging.

Ciascun utente, registratosi dopo aver scelto un proprio **nome utente**, **nome completo**, **e-mail**, **password** e, opzionalmente, un **avatar**, ha la possibilità di creare blog e pubblic arvi all'interno dei post.

Iscrivendosi ad un blog, la propria homepage viene popolata con gli ultimi post di quel blog. È possibile, ovviamente, iscriversi a quanti blog si desidera.

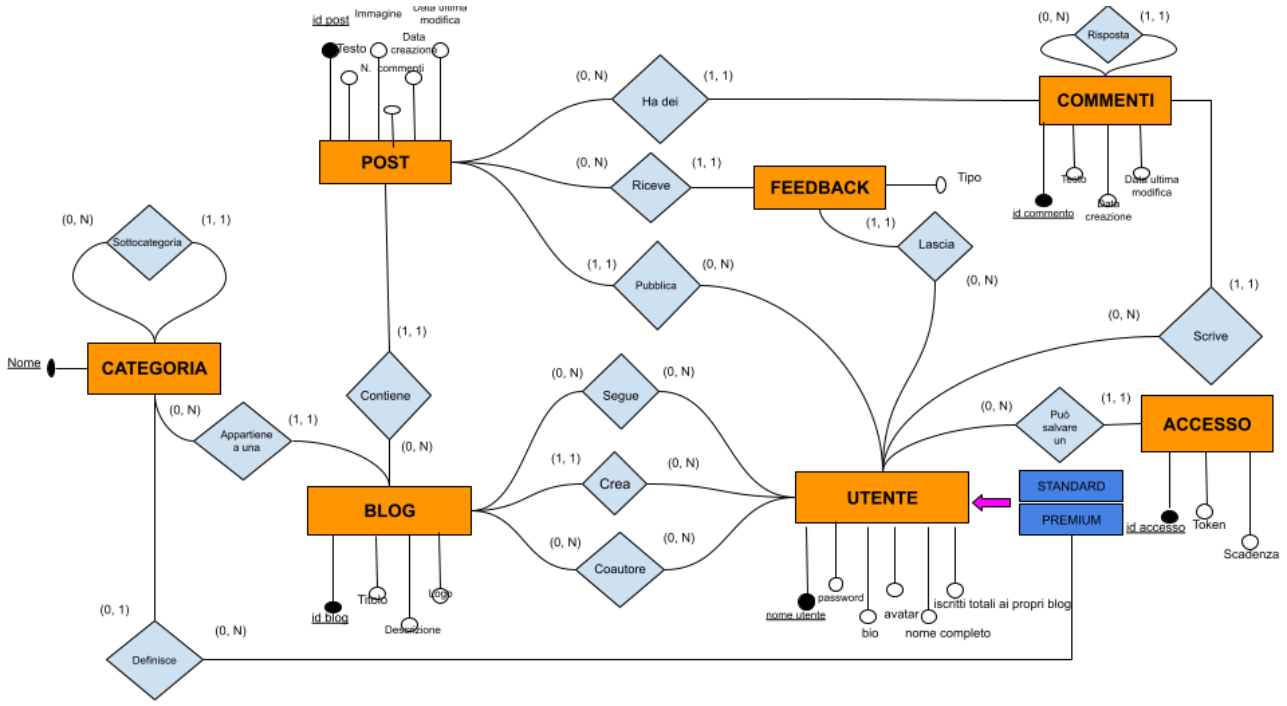
Superata la soglia dei **5 iscritti**, si diventa **utenti premium**, e si acquisisce la possibilità di visualizzare l'elenco degli iscritti ai propri blog. Una volta diventati premium, l'eventuale diminuzione di iscritti non fa perdere lo status.

Ogni blog deve appartenere ad una categoria, ma si può scegliere se assegnarlo ad una categoria principale o a una sottocategoria (ad esempio, si può creare un blog nella categoria *Animali* così come uno nella sottocategoria *Cani*).

È stato scelto di consentire l'utilizzo del sito solamente ad utenti registrati e loggati, come, in varia misura, già avviene in numerosi forum o piattaforme di social network.

Parte I: La base di dati

I. Lo schema ER



Nello schema sono presenti due ridondanze:

- quella relativa al numero totale di iscrizioni ai blog di un utente.
- quella relativa al numero di commenti di un post

Prima ridondanza

Stimando a 5 la media di blog creati da un utente e 10 quella degli iscritti di ciascun blog, senza ridondanza, per conoscere il numero di iscritti totali di un utente, bisogna:

- Accedere all'utente (1 accesso a Utente)
- Conoscere tutti i blog da lui creati (5 accessi a Blog e 5 a Crea)
- Conoscere tutti gli iscritti di ciascun blog (10 accessi a Segue e 10 a Utente)

Ovvero 1 + 5 + 5 + 10 + 10 = 31 accessi. Supponiamo di aver bisogno di questo dato 1000 volte al giorno.

Se il costo operazionale è pari a

$$C(O)lett = F(O) * w * (a * NAw * NAr)$$

Assumendo w = 1 e a = 2, il costo operazionale senza ridondanza sarà

$$C(O)lett = 1000 * (2 * 0 + 31) = 31.000$$

Viceversa, nello scenario con ridondanza per conoscere il numero di iscritti dell'utente, è sufficiente 1 accesso in lettura

$$C(O)lett = 1000 * (2 * 0 + 1) = 1000$$

Per un costo totale di C(O)tot = 1000.

A fronte di un aumento dell'utilizzo di memoria di 4 byte per utente (il numero di iscritti viene salvato come INT) si ha un risparmio di costi operazionali pari a circa il 96,77%.

Seconda ridondanza

Supponendo una media di 100 commenti per post, il calcolo del numero di commenti totali esigerà 1 accesso in lettura a Post (E), 100 accessi a Ha dei (R) e 100 a Commenti (E).

Avendo dunque da effettuare, per ogni operazione

- 1: Un accesso in lettura all'entità Post (per recuperare i dati del post stesso).
- 100: Accessi alla relazione Ha dei (R), che collega i post ai commenti (mediamente 100 commenti per post).
- 100: Accessi all'entità Commenti (E), per leggere i dati di ciascun commento.

Ammettendo che sia necessario calcolare il numero di commenti per ogni post 1000 volte al giorno, il costo operazionale sarà

$$C(O)lett_{\,senza\ ridondanza} = 1000 * (1 + 100 + 100) = 201.000$$

Con l'opzione con ridondanza, dove il numero dei commenti è mantenuto in un contatore aggiornato direttamente in Post, il costo operativo scende a

$$C(O)lett_{\,ridondanza} = 1000 * 1 = 1000$$

Questo rappresenta un risparmio del 99.5%,

II. Lo schema logico

Il database finale, ridotto alla terza forma normale, si struttura in undici tabelle:

- **Autori**(id_autore*,id_blog*):
- **Blog**(id_blog, titolo_visualizzato, categoria*, descrizione, nome_amministratore_blog*, data_creazione, logo):
- **Categoria**(nome_categoria): Elenco delle categorie ammesse
- **Commenti**(id_commento, nome_autore_commento* id_post_commentato*, contenuto, creazione_commento, ultima_modifica)
- **FeedbackPost**(id_feedback ,id_post_riferimento*, username_autore_feedback*, feedback_type)
- **Iscrizioni** (id_iscrizione, nome_utente_iscritto*, id_blog_iscrizione*)
- **Post** (id_post, titolo_post, nome_autore_post*, id_blog_appartenenza*, creazione, ultima_modifica, contenuto_post, immagine_post, numero_commenti)
- **RicordaUtenti** (id_login_salvato, *nome_utente**, hashed_cookie_token, scadenza_del_cookie)
- **RisposteCommenti** (id_commento_risposta*, id_commento_riferimento*)
- **Sottocategoria** (categoria_principale*, sotto_categoria*)
- **Utenti** (nome_utente, nome_visualizzato, id_autorepassword, email, bio, avatar, premium, data_ora_iscrizione, totale_iscritti_ai_propri_blog)

Cinque trigger, per il cui codice si rimanda direttamente al database:

- Uno per rendere l'utente premium al raggiungimento dei 5 iscritti
- Due per aggiornare il numero di iscritti:
  - Uno attivato al momento dell'iscrizione di un utente a un blog
  - Uno attivato al momento della disiscrizione di un utente da un blog
- Due per aggiornare il numero di commenti ad un post
  - Uno attivato al momento della creazione di un commento.
  - Uno attivato al momento della cancellazione di un commento.

E un evento (*elimina_cookie_scaduti*) che, una volta al giorno, si assicura di eliminare i le righe di *RicordaUtenti* rappresentati accessi scaduti (scadenza a 30 giorni dalla creazione del cookie).

Parte II: **Il sito web**

Scelte di progettazione e programmazione

Il codice, front-end quanto back-end, è stato scritto in ottica *clean code*, scegliendo ovvero come criteri guida la leggibilità e la facilità di manutenzione. Sono dunque state evitate funzioni anonime, *magic values* o livelli di annidamento eccessivi, mentre si è applicato ovunque possibile in formato *snake_case* nella nomenclatura di funzioni, variabili e costanti, in minuscolo le prime due e in maiuscolo le ultime, evitando abbreviazioni di qualunque tipo e mantenendo una certa coerenza nella scelta dei nomi, come ad esempio il prefisso *is_* per funzioni o variabili che restituiscono o

contengono un valore booleano e il prefisso *get_* per funzioni che ne restituiscono uno non-booleano.

Lato client

Il criterio generale è stato quello di mantenere ben separati i livelli dei dati, della presentazione e della logica applicativa: ecco che dunque il front-end, costruito secondo i dettami dell'HTML5, non mescola mai HTML e Javascript, né HTML e CSS.

Per quanto riguarda l'HTML, a guidarci nella scelta dei tag sono state le raccomandazioni del W3C intorno al **web semantico**, applicate in funzione della **SEO** e dell'**accessibilità**. Ecco dunque che ogni immagine è dotata di un attributo *alt* descrittivo, che vengono usate opportuni tag `<label>` per identificare i campi di input in un form, che la pagina non viene strutturata adoperando delle tabelle, che tag deprecati quali `<b>`, `<i>`, `<font>` ecc… sono stati evitati, e che non sono stati usati tag privi di valore semantico come `<br />`. Gli unici due tag non semantici adoperati sono stati `<div>` e `<span>`, utili a identificare delle parti di pagina cui applicare degli stili CSS.

Per quanto riguarda il CSS, tutte le direttive sono contenute in un unico file, *style.css*, che usa come unici selettori i nomi di tag e le classi. La grafica è stata completamente implementata in CSS, sia per semplicità di sviluppo, sia per ottenere migliori prestazioni, sia per mantenere ben separato l'aspetto di presentazione della pagina dagli aspetti logici. L'organizzazione della pagina è realizzata attraverso *grid*, e ci si è assicurati, ancora una volta, di mantenere il più possibile lo stile del codice coerente e organizzato.

L'impiego di Javascript è limitato all'implementazione di AJAX attraverso JQuery. Le scelte relative all'utilizzo di AJAX sono discusse più avanti, mentre qui ci limitiamo a osservare che l'interazione con il DOM avviene esclusivamente attraverso gli **event listener**, evitando completamente gli **event handlers** tradizionali e gli **attributi event handler HTML**, e che si è evitato per quanto possibile di adoperare funzioni di callback anonime: per quanto questa sia una pratica assai diffusa in Javascript e in particolare nell'uso di JQuery, riteniamo che contribuisca a rendere più ardua la lettura del codice.

Accessibilità

Rispetto all'accessibilità, la preoccupazione principale è stata quella di garantire un buon contrasto dei colori e di strutturare il sito web in maniera tale da rendere l'interfaccia chiara e intuitiva. Il sito è perfettamente navigabile da tastiera, sono stati adoperati opportuni attributi "alt" per le poche immagini presenti e si è cercato di mantenere il sito il più possibile leggero e reattivo, al fine di non pregiudicarne l'uso da parte di utenti con connessioni più lente della norma.

AJAX

AJAX è stato adoperato ogni volta in cui fosse possibile o sensato aggiornare un contenuto senza dover ricaricare la pagina. Implementano AJAX in particolare la **live search**, le funzioni per **lasciare o togliere un feedback a un post**, quelle per **seguire un blog**, quelle per **pubblicare, cancellare o modificare un commento**.

AJAX è stato sempre implementato attraverso JQuery, in ragione della sua semplicità di implementazione rispetto al Javascript puro, e adoperando, per le medesime ragioni, sempre JSON per lo scambio di dati in luogo di XML.

Lato server

Come da specifiche, il backend è stato sviluppato adoperando PHP puro.

L'interazione col database è realizzata attraverso **PDO**, prediletto rispetto a mysqli in ragione della sua sintassi più semplice e pulita e della sua indipendenza dal database sottostante, garantendo così una miglior astrazione rispetto al livello della base di dati.

Nella definizione di metodi e funzioni, sono state usate tecniche, quali il ***type hinting***, disponibili solo da PHP 7.0 in poi e, dove possibile, da PHP 8.0 in su, in maniera tale da rompere volutamente la compatibilità del codice con versioni di PHP meno aggiornate e sicure.

I commenti sono in generale confinati ai blocchi **PHPDoc** che accompagnano ogni metodo e funzioni e, occasionalmente, pure alcune costanti. Si è cercato di rendere il codice il più *self-documenting* possibile, limitando i commenti non-PHPDoc a brevi esplicitazioni delle logiche implementate (espongono il *perché*, non il *cosa*) quando non si è riusciti a raggiungere una chiarezza sufficiente con il solo codice.

Sono sempre stati preferiti, ove disponibili, funzioni built-in di PHP rispetto a una implementazione ex-novo di determinate logiche. Ecco allora che gli array non vengono mai iterati con un ciclo `for` o un `while` regolari, ma sempre con un `foreach`, che il controllo del valore *null* di una variabile non avviene mai con `$var === null`, ma sempre con `is_null($var)`, mentre per accertarsi che una stringa sia puramente numerica non viene mai usata una regex, ma sempre con `ctype_digit()`.

Per quanto riguarda i tipi delle variabili, si è cercato, per quanto possibile in un linguaggio tipizzato dinamicamente, di mettere i limiti più rigidi possibile ai tipi di dato ammessi in una certa operazione, sia in ottica di sicurezza - vedasi la sezione successiva - sia di prevedibilità del comportamento del sito. Ciò è stato realizzato fondamentalmente attraverso il *type hinting* di metodi e funzioni e l'utilizzo dello *strict equality operator* (`===`) in luogo del semplice *equality operator* (`==`) per assicurarsi che le variabili confrontate *siano dello stesso tipo* ***e*** *dello stesso valore*, evitando dunque conversioni dagli effetti poco prevedibili.

La gestione degli errori è interamente realizzata attraverso le **eccezioni**.

Gestione delle immagini

Gli unici file che un utente può caricare su Idearium sono delle immagini. Queste sono salvate nella directory *images* indipendentemente dal loro impiego.

Un immagine può avere tre impieghi:

- **Avatar di un utente**
- **Logo di un blog**
- **Immagine allegata a uno o più post**

Le immagini sono salvate con il loro SHA1 come nome seguito dalla loro estensione in minuscolo. Ecco allora che una foto come *MiaFoto.JPG* avrà un nome simile a *d6706c654c6c79299ecc6415e7415469c50b022d.jpg*.

Questo approccio ha due vantaggi:

- **Nessuna necessità di cache-invalidation**: un utente che, poniamo, cambi il suo avatar, vedrà la modifica immediatamente, in quanto il browser, leggendo un nome di file nel tag `<img />` cambiato, lo caricherà nuovamente. Se invece l'avatar di un utente avesse mantenuto il medesimo nome, poniamo *mario_avatar.jpg*, sarebbe stato necessario costringere in qualche modo il browser a caricare nuovamente l'immagine piuttosto che la sua versione salvata in cache.
- **Risparmio di spazio sul disco**: se due utenti scelgono una medesima immagine come foto di profilo, questa, univocamente identificata dal suo SHA1 che ne costituisce il nome, sarà caricata una sola volta sul disco, e ad esser ripetuto sarà solo il riferimento nel record sul database.

L'unico svantaggio vero di questo approccio si manifesta nel momento della cancellazione di un contenuto (blog, post, utente) al momento di decidere se sia il caso o meno di cancellare l'immagine eventualmente presente: infatti, bisogna essere sicuri che l'immagine *sia unica*, ovvero sia utilizzata solo dal contenuto che si vuole cancellare: ad esempio, in caso di cancellazione di un post, bisognerà controllare che la stessa immagine non sia usata come immagine da altri post, che non sia usata da alcun utente come avatar e che nessun blog la stia utilizzando come logo: se anche solo una di queste condizioni è falsa, il file dell'immagine non deve essere cancellato.

Sicurezza

Le principali minacce che un sito di questa natura deve affrontare sono attacchi **XSS** e **SQL Injection**.

XSS

Per quanto riguarda XSS, la soluzione adottata è stata quella di:

1. Adoperando delle regex ove possibile (ad esempio, nella validazione dell'indirizzo e-mail). Per semplificare questa operazione, sono stati usati criteri estremamente stringenti laddove possibile (ad esempio, il nome utente ammette solo caratteri minuscoli alfanumerici; il nome completo, caratteri alfabetici maiuscoli e minuscoli più un eventuale spazio).
2. Usando come ID di post, blog e commenti valori numerici assoluti, in maniera tale da escludere agevolmente qualunque stringa anche solo potenzialmente malevola (ad esempio, per quanto riguarda gli ID dei post, se l'ID contiene dei caratteri non numerici, non viene neppure effettuata una query al database, ma viene semplicemente visualizzata una pagina che informa l'utente del fatto che il post richiesto non esiste).
3. In tutti gli altri casi, è stata adoperata la conversione dei caratteri speciali inseriti dall'utente con *htmlspecialchars*, in modo da assicurarsi di convertirli in entità HTML prima di salvarli nel database o visualizzarli nuovamente

SQL Injection.

Per quanto riguarda l'SQL Injection, dopo aver filtrato adeguatamente l'input dell'utente come sopra, ogni singola operazione è stata effettuata attraverso i ***prepared statement*** di PDO, usando dei placeholders letterali per maggior leggibilità e indicando ogni volta in modo esplicito il tipo di dato atteso.

I controlli sull'input vengono effettuati tutti lato server: effettuarli lato client, magari con JQuery, sarebbe stato inutile, in quanto l'utente avrebbe potuto disabilitare Javascript sul suo browser o modificare il codice di validazione, dunque sarebbe stato in ogni caso necessario ripetere i controlli lato server.

Password

Delle password degli utenti, è salvato nel database un hash + salt calcolato con la funzione *password_hash*, in modo da prevenire il rischio di una compromissione degli account utente nel caso di una fuga di dati.

Il salvataggio degli accessi: meccanismo e aspetti relativi alla sicurezza

La principale funzione extra del progetto è quella che consente agli utenti di "Ricordare" l'accesso al sito su una certa macchina, in modo tale da non dover re-effettuare il login alla visita successiva.

1. Quando un utente decide di ricordare l'accesso, un token di 512 caratteri alfanumerici casuali viene generato. Questo token, hashato con `password_hash()`, viene salvato in un record del database. Assieme al token, vengono salvati nel database lo username dell'utente che ha scelto di salvare il login, un ID progressivo che consente di identificare univocamente il login salvato (consentendo dunque a uno stesso utente di salvare un accesso su più browser) e la data di scadenza di questo accesso, impostata a 30 giorni dal momento del salvataggio.
2. Nel corpo *value* del cookie vengono salvati gli stessi dati, con l'eccezione della scadenza, in formato JSON:

```
{
    'username': username_utente,
    'saved_login_id': id_numerico_login_salvato
    'token': token_alfanumerico_di_512_caratteri
}
```

Al momento del login, viene controllata la presenza del cookie REMEMBER_ME. Se presente:

1. Viene letto il `saved_login_id`
2. Viene selezionata nella tabella *RicordaUtenti* del database la riga identificata dall'ID letto e avente il campo *nome_utente* identico allo `username` del cookie. Se non viene trovata alcuna riga, significa che il cookie è stato manomesso, e la procedura si interrompe qui; se vengono trovate due o più righe, significa che c'è un problema nel database, e viene sollevata un'eccezione in PHP.

3. Se viene trovata una e una sola riga, viene fatto il raffronto tra il campo `token` memorizzato nel cookie e il suo hash memorizzato nel database, attraverso la funzione `password_verify()`. Se v'è corrispondenza, significa che il cookie non è stato manomesso e l'accesso è legittimo, dunque l'utente viene loggato; altrimenti, significa che v'è stata una manomissione del cookie, dunque la procedura viene interrotta e all'utente è richiesto di procedere al login regolare.

Le misure di sicurezza adottate nell'implementare tale funzionalità sono due:

1. **Raffronto dei dati del cookie con dati salvati nel database**: serve a prevenire un attacco di questo tipo: un utente, manomettendo il valore del cookie salvato, riesce a ottenere l'accesso a un account altrui (il caso più semplice sarebbe quello in cui il cookie contenesse semplicemente il nome utente dell'utente salvato, e un utente, cambiando quel valore, facesse credere al sito che a salvare l'accesso, piuttosto che *tizio*, sia stato *caio*). In questo modo, se i dati del cookie non corrispondono a quelli salvati nel database (sorta di copia di sicurezza), significa che è stata attuata una manomissione di qualche tipo del cookie dell'utente.
2. **Salvataggio del token hashato**: il salvataggio nel database dell'hash del token, piuttosto che del token in chiaro, serve, come nel caso delle password, a evitare che un attaccante che riuscisse ad accedere al database possa usare quel token per fabbricare un cookie su misura e ottenere così l'accesso all'account di un utente, facendo credere al sito che quel cookie indichi un accesso salvato dall'utente stesso. Non hashare il token avrebbe esposto gli account al medesimo rischio cui li avrebbe esposti salvare le password in chiaro.

Salvare la scadenza del token nel database consente di eliminare gli accessi scaduti automaticamente. Ciò viene effettuato una volta al giorno attraverso l'evento *elimina_cookie_scaduti*.

Per la prova

Alcuni nomi utenti con relativa password:

- *zero* (utente premium), pw: 12345678
- *giovanni*, pw: 12345678
- *hero*, pw: 12345678

Alcuni blog si trovano nella categoria *Animali* > *Acquario*, in *Arte e cultura* > *Letteratura e critica letteraria* e in *Scienza* > *Paleontologia*