PROGETTO EXTRACORSO TECNOLOGIE WEB

GRUPPO 50

FILIPPO MARIUCCI 1095256

GIORGIO OLIVIERI 1092407

INDICE DOCUMENTAZIONE

1 Riferimenti e contributo

Il progetto ha come obiettivo la realizzazione di un sito Web finalizzato a creare una community on-line dove gli utenti registrati si scambiano messaggi e creano blog di discussione su argomenti a scelta degli utenti registrati .Coloro che hanno progettato questo sito sono gli studenti appartenenti al GRUPPO 50 , ovvero:

• Filippo Mariucci (1095256), con un contributo del 50%

• Olivieri Giorgio (1092407), con un contributo del 50%

2 Descrizione del sito

2.1 Descrizione delle funzionalità

Attraverso questa applicazione web un navigatore può:

• visualizzare le informazioni generali sulla comunità e sulla società che gestisce il sito;

• iscriversi alla comunità stessa inserendo i suoi dati anagrafici, la sua mail , scegliere se avere un profilo pubblico o privato (in tal caso i dati completi del suo profilo sono visibili solo ai suoi amici) ed una breve descrizione.

Una volta iscritto, può:

• verificare se un altro individuo è iscritto o meno alla comunità, accedendo ad una funzione di ricerca tramite l'indicazione del nome della persona cercata. Se questa è presente e visibile a tutti, la funzione restituisce tutti i dati del membro della comunità (dati anagrafici e profilo). Se invece la persona cercata ha scelto di essere visibile solo ai suoi amici, la ricerca restituisce i dati completi se attivata da un amico, il solo nome e cognome negli altri casi;

• chiedere di entrare nel gruppo degli amici di un altro, inviando una richiesta in tal senso attraverso un sistema di messaggistica interna al sito. La richiesta può essere accettata o meno dal destinatario, e resta memorizzata in modo permanente nel sistema, assieme all'indicazione della data in cui è stata emessa. Se un utente A accetta la richiesta di amicizia fatta da un utente B, B diviene amico di A ed A di B. Ogni utente può appartenere a più gruppi;

• eliminare dal gruppo dei suoi amici chiunque quando vuole; in questo caso l'utente eliminato riceve un messaggio di notifica.

• attivare un suo blog (pagina di testo in cui esprimere opinioni su un tema a sua scelta) al quale possono accedere anche i suoi amici sia in lettura che in scrittura, per inserire commenti alle opinioni del blogger.

• ogni nuovo post sul blog è identificato dalla data, dall'ora di inserimento e dal mittente, e si aggiunge in coda ai precedenti;

• ogni volta che un post viene aggiunto ad un blog, tutti coloro che vi hanno accesso ricevono un messaggio di notifica (data, ora e mittente del post).

Relativamente a chi fa l'accesso al sito, possiamo distinguere diversi utenti, con diverse funzionalità:

• UTENTE NON REGISTRATO E NON LOGGATO : LIVELLO 1

Può visualizzare l'area pubblica del sito , cioè la parte con le informazioni generali e servizi forniti a tutti coloro che accedono al sito. Gli utenti di questa categoria potranno vedere:

- la presentazione dell'azienda, compresi i riferimenti per i contatti (telefono, e-mail, …);

- le informazioni generali sulla comunità (scopo, regole di partecipazione, …)

- le caratteristiche, le funzionalità e le condizioni d'uso del portale;

- la pagina di iscrizione alla comunità

• UTENTE REGISTRATO: LIVELLO 2

Può accedere alla propria aria riservata e può:

-modificare il proprio profilo personale;

- gestire il gruppo di amici: visualizzare i suoi amici, cercare nuovi amici, inviare e rispondere a richieste di amicizia

- creare dei blogs;

- postare sui propri blogs o su quelli dei suoi amici;

• MEMBRO STAFF: LIVELLO 3

I membri dello staff , scelti dall'amministratore, una volta loggati nella propria area riservata possono:

- visualizzare il contenuto dei blog di tutti gli utenti;

-cancellare interi blog o singoli post ritenuti non in linea con il comportamento stabilito per i membri della comunità. Ad ogni cancellazione corrisponde l'invio di un messaggio a chi ha inserito l'oggetto cancellato con l'indicazione del motivo dell'azione;

• AMMINISTRATORE : LIVELLO 4

Una volta loggato l'amministratore ,oltre a svolgere le funzioni dell'utente di livello 3, può :

- gestire (inserisce/cancella/modifica) i membri dello staff;

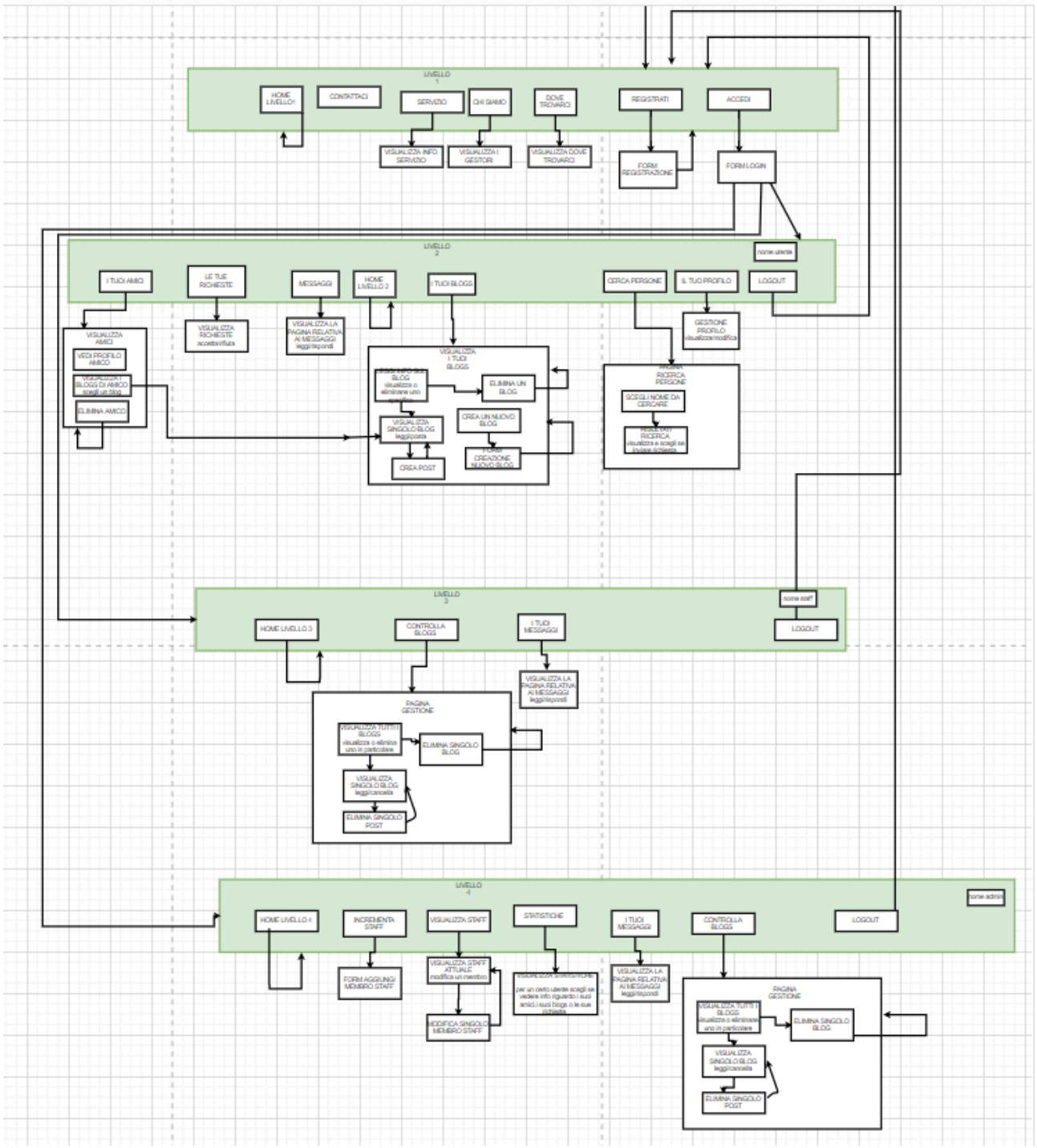
- estrarre informazioni sulla comunità quali: la composizione (nome e cognome dei membri) del gruppo di amici di un dato membro della comunità; il numero di richieste di amicizia ricevute (accettate o meno) da un dato membro della comunità; il numero totale di blog creati dai membri della comunità.

2.2 Funzionalità Opzionali

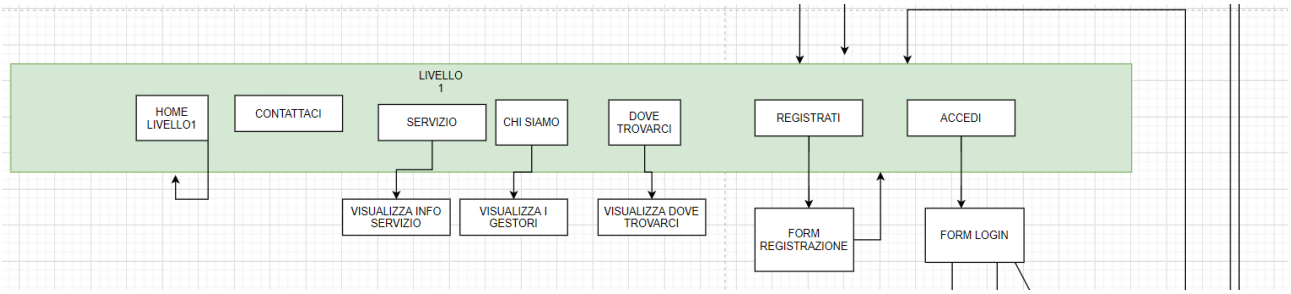
Relativamente a questo tipo di funzionalità abbiamo permesso agli utenti di creare più blogs senza però fargli selezionare un sottoinsieme del suo gruppo di amici a cui dare accesso esclusivo a un determinato blog. Quindi ogni utente può creare più blogs a cui tutti i suoi amici hanno accesso.

3 Schema dei link

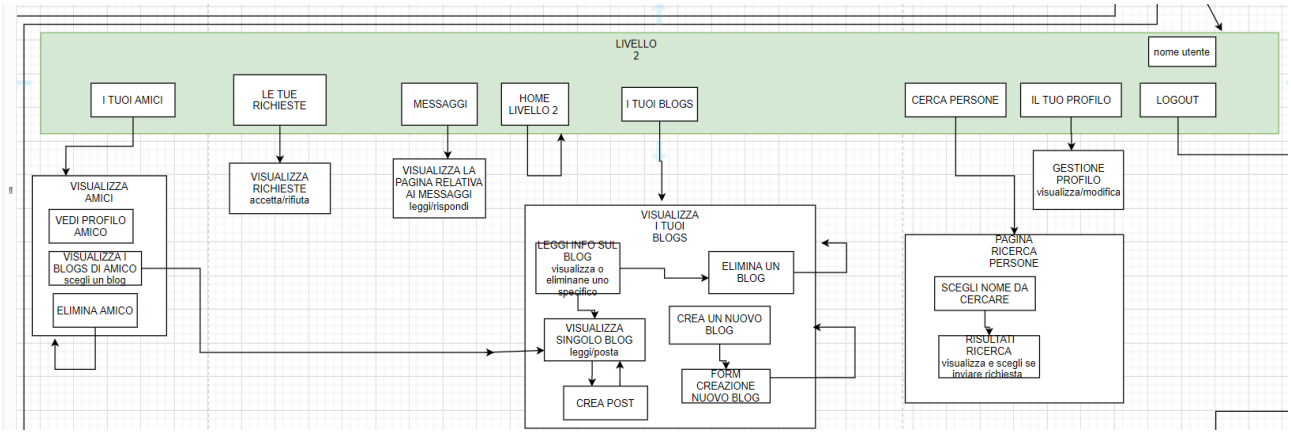
Di seguito riportiamo lo schema dei link che è stato articolato per livelli di utenza. Per questioni di spazio mostriamo prima quello completo e poi ,per rendere meglio visibili i singoli livelli, gli screenshot mostrano che meglio i vari dettagli dei singoli livelli.


LIVELLO 1
HOME LIVELLO1
CONTATTACI
SERVIZIO
CHI SIAMO
DOVE TROVARCI
REGISTRATI
ACCEDI
VISUALIZZA INFO SERVIZIO
VISUALIZZA I GESTORI
VISUALIZZA DOVE TROVARCI
FORM REGISTRAZIONE
FORM LOGIN
LIVELLO 2
nome utente
I TUOI AMICI
LE TUE RICHIESTE
MESSAGGI
HOME LIVELLO 2
I TUOI BLOGS
CERCA PERSONE
IL TUO PROFILO
LOGOUT
VISUALIZZA AMICI
VEDI PROFILO AMICO
VISUALIZZA BLOGS DI AMICO scegli un blog
ELIMINA AMICO
VISUALIZZA RICHIESTE accetta/rifiuta
VISUALIZZA LA PAGINA RELATIVA AI MESSAGGI leggi/rispondi
VISUALIZZA I TUOI BLOGS
ELIMINA UN BLOG
VISUALIZZA SINGOLO BLOG leggi/posta
CREA UN NUOVO BLOG
CREA POST
PAGINA CREAZIONE NUOVO BLOG
GESTIONE PROFILO visualizza/modifica
PAGINA RICERCA PERSONE
SCEGLI NOME DA CERCARE
RISULTATI RICERCA visualizza e scegli se inviare richiesta
LIVELLO 3
nome staff
HOME LIVELLO 3
CONTROLLA BLOGS
I TUOI MESSAGGI
LOGOUT
VISUALIZZA LA PAGINA RELATIVA AI MESSAGGI leggi/rispondi
PAGINA GESTIONE
VISUALIZZA TUTTI I BLOGS visualizza o elimina uno in particolare
ELIMINA SINGOLO BLOG
VISUALIZZA SINGOLO BLOG leggi/cancella
ELIMINA SINGOLO POST
LIVELLO 4
nome admin
HOME LIVELLO 4
INCREMENTA STAFF
VISUALIZZA STAFF
STATISTICHE
I TUOI MESSAGGI
CONTROLLA BLOGS
LOGOUT
FORM AGGIUNGI MEMBRO STAFF
VISUALIZZA STAFF ATTUALE modifica un membro
MODIFICA SINGOLO MEMBRO STAFF
per un certo utente scegli se vedere info riguardo i suoi amici,i suoi blogs o le sue richieste
VISUALIZZA LA PAGINA RELATIVA AI MESSAGGI leggi/rispondi
PAGINA GESTIONE
VISUALIZZA TUTTI I BLOGS visualizza o elimina uno in particolare
ELIMINA SINGOLO BLOG
VISUALIZZA SINGOLO BLOG leggi/cancella
ELIMINA SINGOLO POST

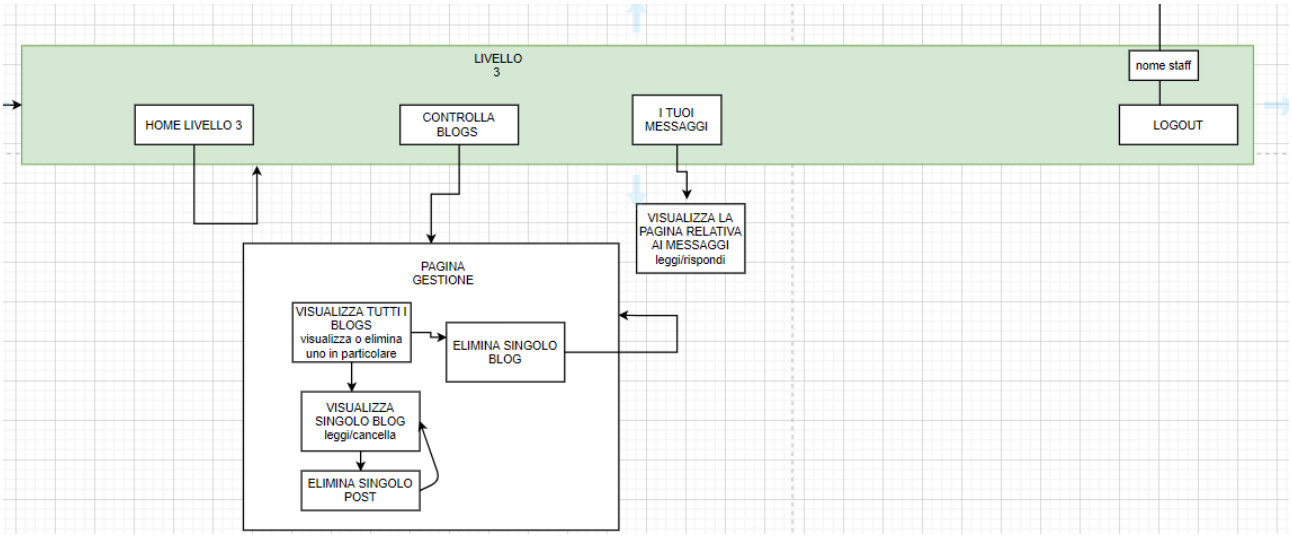
-Di seguito lo schema relativo a livello 1



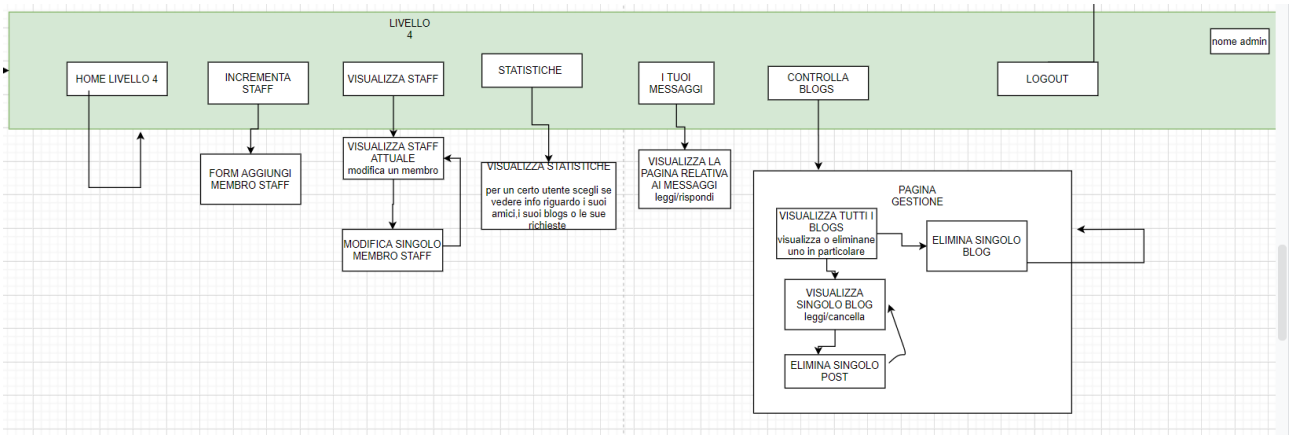
-Di seguito lo schema relativo a livello 2



-Di seguito lo schema relativo a livello 3



-Di seguito lo schema relativo a livello 4



4 Soluzioni adottate

4.1 Modello database e ricerca dei miei amici

Riteniamo necessario illustrare la scelta del modello del database perché poi si riflette sulle funzioni che abbiamo scritto per gestire i dati. In particolare ci concentriamo sulle tabelle “richieste” e “amici” che riportiamo rispettivamente di seguito con la loro struttura.

I campi “utente_riferimento” ,”amico_utente_riferimento”,”accettante” e “richiedente” fanno tutti riferimento a ‘id’ della tabella ‘users’.

Richieste

	#	Nome	Tipo
☐	1	**id**	bigint(20)
☐	2	**data_richiesta**	date
☐	3	**data_risposta**	date
☐	4	**stato**	int(11)
☐	5	**richiedente**	int(11)
☐	6	**accettante**	int(11)
☐	7	**created_at**	timestamp
☐	8	**updated_at**	timestamp

Amici

#	Nome	Tipo
1	**id**	bigint(20)
2	**utente_riferimento**	int(11)
3	**amico_utente_riferimento**	int(11)
4	**created_at**	timestamp

Quando un utente B chiede amicizia ad un utente A, nella tabella Richieste B è il richiedente mentre A è l’accettante, a prescindere dal fatto che la richiesta venga accettata o rifiutata.

Quando A risponde alla richiesta :

-se rifiuta lo stato della richiesta passerà da ‘in attesa’ a ‘rifiutata’ (nel db poi avremo degli int) e non saranno amici

- se accetta allora lo stato della richiesta passerà da ‘in attesa’ a ‘accettata’ e ciò significa che A e B diventano amici. Quindi nella tabella Amici dobbiamo andare a memorizzare la nuova relazione di amicizia . In questo caso chi accetta la richiesta lo mettiamo nella colonna ‘utente riferimento’ mentre chi aveva mandato la richiesta nella colonna ‘amico utente riferimento’. In questo modo riesco a capire, anche solo dalla tabella Amici, chi ha chiesto a chi di diventare amici.

Adesso vogliamo spiegare l’ideazione delle funzioni che vanno a gestire le amicizie (in particolare quando un utente vuole vedere i suoi amici o quando admin vuole vedere gli amici di un determinato utente del sito) fornendo un esempio.

Dalle specifiche di progetto se A chiede a B di essere amici e B accetta , allora A è amico di B e anche B è amico di A.

Ipotizziamo che io sia l’utente con id= ‘a ’ , sono attualmente loggato e voglio vedere i miei amici. Nel db in un certo momento nella tabella amici ho la seguente situazione :

utente riferimento	amico utente riferimento
a	b
a	c
a	d
e	a

Da quanto detto prima capisco che i miei amici sono ‘b’, ’c’, ’d’ ed ’e’ . In particolare io ho richiesto a ‘e’ di essere amici mentre negli altri casi sono stato io ad accettare le richieste arrivate da altri utenti.

Se voglio conoscere TUTTI i miei amici devo ,con una query ,scorrere tutte le righe della tabella Amici, vedere dove l’id salvato è = ‘a’ in una tra queste due colonne e prendere l’id dell’utente nella colonna accanto. Se non faccio così allora rischio di non recuperare effettivamente TUTTI i miei amici. Infatti se faccio il controllo solo sulla colonna “utente_riferimento”,per esempio, recupero gli id ‘b’,’c’ e ‘d’ ma non recupero ‘e’ che è sempre un mio amico dalle specifiche di progetto.

Nei controller abbiamo implementato due versioni di una stessa funzione per cercare gli amici: una versione ‘LEFT’ e una ‘RIGHT’ dove l’unica differenza è la colonna (tra le due riportate nella foto sopra) su cui vado a controllare se l’id dell’utente loggato (che vuole vedere i suoi amici) è presente oppure no.

Di seguito mostriamo il codice delle suddette funzioni che recuperano id (e poi altre info) dei miei amici :

```
public function getmyfriendLEFT($id) {

$id=auth()->user()->id;

 $idamici= Amici::where('utente_riferimento','=',$id)->select("amico_utente_riferimento")->get()->toArray();

$check=count($idamici);

if($check==0){
$amico=['     ']   // se è vuota non gli faccio stampare nulla perchè questa è una stringa di x
}                  // spazi tra le '' quante sono le $app seguenti (infatti stampo tre campi se non è vuota)

if(($check!=0)) {
    for($i=0;$i<count($idamici);$i++){
            $app1= Users:: where('id','=',$idamici[$i])->value( "name");
            $app2= Users:: where('id','=',$idamici[$i])->value( "cognome");
            $app3= Users:: where('id','=',$idamici[$i])->value( "username");
            $app4= Users:: where('id','=',$idamici[$i])->value( "id");    // questo poi mi serve come parametro da passare alle rotte
             $app5= Users:: where('id','=',$idamici[$i])->value( "foto_profilo");
            $amico[$i]=[$app1,$app2,$app3,$app4,$app5];
        }
    }//chiude if
           return $amico;
    }
```

Nota: la parte riportata subito qui sotto come testo serve solo per gestire la corretta visualizzazione della pagina dove vedo gli amici e non serve per il recupero effettivo dei dati dei miei amici

$check=count($idamici);

if($check==0){

$amico=['   ']; } // se è vuota non gli faccio stampare nulla

```php
public function getmyfriendRIGHT($id) {

$id=auth()->user()->id;

$idamici= Amici::where('amico_utente_riferimento','=',$id)->select("utente_riferimento")->get()->toArray();


$check=count($idamici);

if($check==0){
$amicoright=['     '];   // se è vuota non gli faccio stampare nulla
}


if(($check!=0)) {
    for($i=0;$i<count($idamici);$i++){
            $app1= Users:: where('id','=',$idamici[$i])->value( "name");
            $app2= Users:: where('id','=',$idamici[$i])->value( "cognome");
            $app3= Users:: where('id','=',$idamici[$i])->value( "username");
            $app4= Users:: where('id','=',$idamici[$i])->value( "id");        // questo poi mi serve come parametro da passare alle rotte
              $app5= Users:: where('id','=',$idamici[$i])->value( "foto_profilo");
             $amicoright[$i]=[$app1,$app2,$app3,$app4,$app5];
        }
}//chiude if
    return $amicoright ;
    }
```

La scelta di utilizzare due funzioni di fatto identiche deriva dall'idea di scrivere funzioni il più semplici possibili per recuperare dalla tabella Amici gli id dei miei amici. Infatti avere un'unica funzione che fa il controllo su entrambe le colonne richiedeva una query molto più complessa da scrivere e non intuitiva; pertanto abbiamo optato per questa scelta puntando sulla semplicità del codice.

In sostanza avere delle funzioni che nella denominazione hanno 'left' e 'right' dipende da questa scelta.

Ora mostriamo una possibile alternativa che però abbiamo scartato in quanto ritenuta inferiore nel suo complesso.

Per usare una stessa funzione senza 'duplicarla' in due versioni avrei dovuto duplicare i dati nel db nella tabella Amici. Cioè avrei potuto fare la scansione su una sola colonna , ad esempio "utente_riferimento", ma per ogni relazione di amicizia tra due utenti avevo necessità di occupare due righe e non più una.

Esempio: se 'b' chiede ad 'a' di essere amici e la richiesta viene accettata io nella tabella Amici registro la nuova relazione in questo modo :

utente riferimento	amico utente riferimento
a	b
b	a

Solo facendo così riuscirei a recuperare 'a' tra gli amici di 'b' perché la scansione la faccio solo sulla colonna a sinistra. In questo modo però non riesco più a distinguere a priori chi ha fatto la richiesta e chi invece l'ha accettata.

Inoltre ci sono considerazioni non indifferenti sullo spazio di memoria che si va ad occupare per memorizzare le relazioni di amicizia. Nell'ipotesi che il sito venga utilizzato nella realtà da centinaia, migliaia o addirittura milioni di utenti , allora la tabella Amici avrebbe un numero spropositato di righe che sono di fatto copie 'inutili' di altre. Questa alternativa mi permette quindi di usare una sola funzione ma nella realtà rischia di far 'esplodere ' la tabella amici.

Da questo punto di vista conviene duplicare la funzione (che è estremamente semplice e non necessità di memoria crescente per essere mantenuta perché non varia) in due versioni piuttosto che duplicare un numero estremamente grande di righe nel db nell'ipotesi fatta riguardo la diffusione reale del sito.

4.2 Messaggi di notifica

Dalle specifiche di progetto ci sono diversi azioni che richiedono dei messaggi di notifica per informare un utente di un avvenuto evento che lo riguarda direttamente. Essendo appunto dei messaggi di notifica(dove cambiano solo i destinatari e il testo in base l tipo di evento) abbiamo scelto di rendere l'invio automatico al verificarsi dell'evento e di utilizzare testi standard per questo tipo di messaggi.

Di seguito riportiamo un solo esempio, in particolare quando l'admin o un membro dello staff eliminano un blog.

```
public function deletethisblog($idblog)
    {
    $nomeblog=Blog::where("id",$idblog)->value("titolo");
     $blog=Blog::where("id",$idblog)->first();
      $app=Blog:: where("id",$idblog)->value("utente_proprietario");

     $messaggio = new Messaggi([
            'contenuto' => "il tuo blog ".$nomeblog." è stato eliminato perchè non conforme alla nostra politica sui contenuti ",
            'data' => Carbon::now()->addHours(2),
            'mittente' => auth()->user()->id,
            'destinatario' => $blog->utente_proprietario
        ]);
        $messaggio->save();

         $blog->delete();
        return  redirect()->route('listablogs')
            ->with('status', 'blog eliminato correttamente!');
    }
```

Il mittente del blog è l'admin o il membro dello staff che effettua questa azione, mentre il destinatario è il proprietario del blog sanzionato. Il testo è sempre lo stesso perché cambia solo il nome del blog che viene eliminato.

4.3 Messaggistica

Una delle sezioni del progetto che ci ha coinvolto di più è stata la parte della messaggistica. Più in particolare l'anteprima delle conversazioni, dove le varie chat sono distinte per utente. Questa funzionalità ci ha fatto ragionare nella sezione del model della Messaggistica in quanto, oltre richiedere i dati al Database, dovevamo ricatalogarli in modo che questi siano utilizzabili nel modo più facile possibile.

Infatti come si può notare nella sezione di codice qui sotto elencata, dopo aver cercato tutti i messaggi dove l'utente registrato era il mittente e/o destinatario, li abbiamo uniti e poi eliminato i 'doppioni' (se 'a ' è mittente rispetto ad un messaggio per 'b' e allo stesso tempo è anche un destinatario per un certo messaggio inviato da 'b', io recupero una doppia informazione perché di fatto la chat è unica perché è sempre tra 'a' e 'b'). Di seguito la funzione 'getchat()' nel model 'Messaggistica' che si occupa di ciò. In particolare vogliamo far notare il meccanismo che abbiamo ideato per eliminare i doppioni e salvare i valori unici nel vettore 'ridotto', nome dato per alludere all'eliminazione dei duplicati.

```
public function getChat($id){       // questa funzione serve per recuperare TUTTE LE CHAT  dell'utente attualmente loggato che è identificato da $id
       //dove $id é il mittente
       $idlist1 = Messaggi::where("mittente",$id)->select("destinatario")->distinct()->get()->toArray(); //prendo i destinatari dei miei messaggi
       //dove $id è destinatario
       $idlist2 = Messaggi::where("destinatario",$id)->select("mittente")->distinct()->get()->toArray();//prendo i mittenti dei messaggi che mi arrivano
     //ora li unisco
     $idunici=array_merge($idlist1,$idlist2);

     for($i=0;$i<count($idunici);$i++){
           $app1= Users:: where('id','=',$idunici[$i])->value( "name");
           $amico[$i]=[$app1];
       }  // array con indici numerici

   $result=[];
    for($y=0;$y<count($idunici);$y++){
           $app1= Users:: where('id','=',$idunici[$y])->value( "username");
           $app2= Users:: where('id','=',$idunici[$y])->value( "id");
            $result[$y]=[$app1,$app2];   //result[0] è username     mentre  [1] è id: li uso poi nella view messaggi
      }   // qui prendo id e username per ogni persona con cui ho un messaggio,indpendentemente che io sia mittente o destinatario

ridotto=[];  // vettore che salva i valori NON duplocati all'interno di result
numero=0; // numero di valori tra gli id degli amici  che rimangono non doppiati
       for($a=0;$a<count($result);$a++){

            $trovato=0;  // lo inizializzo e lo  interpreto come booleano , ovvero non trovato

            for($b=0;$b<$numero && $trovato==0;$b++){

            if($ridotto[$b]==$result[$a]){
               $trovato=1;   // come booleano: l'ho trovato
               }
            }
                if($trovato==0){      // se non trovo duplicati allora lo salvo in ridtottoe aumento $numero che mi dice il numero di elementi non duplicati di result
                  $ridotto[$numero]=$result[$a];
                  $numero=$numero+1;
                 }
       }// fine for esterno
  return $ridotto;
   }
```

4.4 Cambio foto profilo utente

Questo 'if' all'interno della funzione 'updateProfilo' svolge il compito di andare a sostituire l'immagine profilo dell'utente con una nuova, eliminando però, prima, quella vecchia (se esistente).

```php
if ($request->hasFile('foto_profilo')) {
    $image = $request->file('foto_profilo');
    $imageName = $image->getClientOriginalName();
    $destinationPath = public_path() . '/img/foto_profilo';
    $oldImage = $destinationPath . '/' . auth()->user()->foto_profilo;
    File::delete($oldImage);
    $image->move($destinationPath, $imageName);
} else {
    $imageName = auth()->user()->foto_profilo;
}
```

4.5 Creazione di un nuovo blog

Per quanto riguarda la creazione di un blog, diamo la possibilità all'utente di crearne uno con un'apposita form dove può scegliere il titolo e la descrizione. In particolare vogliamo che questi elementi siano sempre visibili in alto nella pagina di visualizzazione di un singolo blog.La nostra scelta pertanto è stata quella di considerare il 'primo post' sul blog ,che un utente ha appena creato, la stessa descrizione che ha inserito nella form insieme al titolo. In questo modo se sono presenti molti post e scrollo la pagina per vedere quelli più in basso, io riesco sempre a vedere la descrizione che funge da primo post.

Logicamente riteniamo questa scelta conforme alle specifiche perché se avessimo ,oltre al titolo e alla descrizione, scritto il primo post come un qualunque altro post, di fatto questo sarebbe stato un'estensione della descrizione perché nessun altro utente prima del creatore avrebbe potuto postare sul blog stesso. Per rimanere conformi alle specifiche del progetto allora ad ogni creazione di un nuovo blog ci sarà un

messaggio automatico di notifica a tutti gli amici del proprietario del blog, ovvero coloro che vi hanno accesso.

Di seguito riportiamo la funzione 'storeblog' presente in UtenteController (e analoga a 'storepost') che mi permette di creare un blog e all'interno c'è anche la parte che implementa i messaggi di notifica. Facciamo notare che la divisione di idamici in left e right dipende dalla sezione 4.1 precedentemente illustrata.

```
public function storeblog(NewBlogRequest $request){
       $blog= new Blog;
       $blog->fill($request->validated());

    $utenteproprietario=auth()->user()->id;
    $blog['utente_proprietario']=$utenteproprietario;
    $nome=$blog['titolo'];
    $blog->save();

    //ora notifico in automatico a tutti i miei amici che ho creato questo blog
    $idamicileft= Amici::where('utente_riferimento','=',$utenteproprietario)->select("amico_utente_riferimento")->get()->toArray();
    $idamiciright= Amici::where('amico_utente_riferimento','=',$utenteproprietario)->select("utente_riferimento")->get()->toArray();

       $idleft=[];
       for($r=0;$r<count($idamicileft);$r++){
       $app=Users::where("id",$idamicileft[$r])->value("id");
       $idleft[$r]=$app;
       }
    //messaggi per amicileft
     for ($i=0;$i<count($idleft);$i++){
      $messaggioleft = new Messaggi([
           'contenuto' => "Ho appena creato il seguente blog: ".$nome.".Corri a vederlo! ",
           'data' => Carbon::now()->addHours(2),
           'mittente' => auth()->user()->id,
           'destinatario' => $idleft[$i]
      ]);
      $messaggioleft->save();
     }//fine for
      $idright=[];
      for($s=0;$s<count($idamiciright);$s++){
      $app2=Users::where("id",$idamiciright[$s])->value("id");
      $idright[$s]=$app2;
      }

     for ($j=0;$j<count($idright);$j++){
      $messaggioright = new Messaggi([
           'contenuto' => "Ho appena creato il seguente blog: ".$nome.".Corri a vederlo! ",
           'data' => Carbon::now()->addHours(2),
           'mittente' => auth()->user()->id,
           'destinatario' => $idright[$j]
      ]);
      $messaggioright->save();
      }//fine for

      return redirect()->route('mioblog')
          ->with('status', 'Blog creato correttamente!');
  }
```

4.6 Ricerca persone

La parte che consente ad un utente di cercare altre persone tra gli utenti già registrati nel sito è la funzione "cercautenti" presente in UtenteController. Un utente può inserire un nome o una sua parte (come richiesto nelle specifiche di progetto) che vuole cercare e successivamente vedrà tutti i risultati che corrispondono alla ricerca effettuata. Ad ogni utente che comparirà dai risultati della ricerca potrà inviare una richiesta di amicizia solo se non sono già amici o se c'è già una richiesta in attesa di risposta tra i due. I controlli relativi a quest'ultima parte sono realizzati nella view che mostra i riultati della ricerca grazie ai valori delle variabili $off9 e $off10 nella funzione in questione che riportiamo di seguito.

```
 public function cercautenti(Request $ricerca)  {

    $loggato=auth()->user()->id;  //mi serve per controlli in query successive
    $nomeinserito=$ricerca->name; //prendo il nome che ho inserito per poi  confrontarlo con quelli nel db

    if ( strpos($nomeinserito,"*") !== false) {        //se c'è il carattere jolly lo tolgo per fare confronto con dati presenti nel db
    $nomeinserito=str_replace("*", "", $nomeinserito);
    }
// estraggo da tutti gli utenti che corrispondono alla ricerca
$idutenti = Users::where(function($nome) use ($nomeinserito){
            $nome->where('name','LIKE', $nomeinserito.'%');
        })->select("id")->get()->toArray();

    $trovati=[]; // vettore in cui salvo info dei soli utenti che corrisponodono ai requisiti della ricerca e che quindi voglio far comparire nella view come risultati

    for($r=0;$r<count($idutenti);$r++){
     $off0= Users:: where('id','=',$idutenti[$r])->value( "name");
     $off1= Users:: where('id','=',$idutenti[$r])->value( "cognome");
     $off2= Users:: where('id','=',$idutenti[$r])->value( "username");
     $off3= Users:: where('id','=',$idutenti[$r])->value( "sesso");
     $off4= Users:: where('id','=',$idutenti[$r])->value( "data_nascita");
     $off5= Users:: where('id','=',$idutenti[$r])->value( "foto_profilo");
     $off6= Users:: where('id','=',$idutenti[$r])->value( "descrizione");
     $off7= Users:: where('id','=',$idutenti[$r])->value( "id");
     $off8= Users:: where('id','=',$idutenti[$r])->value( "visibilita");  // mi serve per capire se mostro tutto o no

     //questi dopo mi servono per dei controlli nella view
     $app3=Richieste::where("richiedente",$loggato)->where("accettante",$idutenti[$r])->where("stato",1)->count();
     $app4=Richieste::where("accettante",$loggato)->where("richiedente",$idutenti[$r])->where("stato",1)->count();
     $off9=$app3+$app4;  // se c'è gia una richiesta in stato di attesa tra me e lui ,per cui risultato sarà >0, non devo inviare ancora richiesta

     $app1=Amici::where("utente_riferimento",$loggato)->where("amico_utente_riferimento",$idutenti[$r])->count();
     $app2=Amici::where("utente_riferimento",$idutenti[$r])->where("amico_utente_riferimento",$loggato)->count();
     $off10=$app1+$app2; // se è >0 significa che il loggato è gia amico con  utente che ha id=$ridotto[$r]

     $trovati[$r]=[$off0,$off1,$off2,$off3,$off4,$off5,$off6,$off7,$off8,$off9,$off10];
    }
        return view('cercapersone')
           ->with('trovati',$trovati) ;
    }
```

5 Seeding del DB

Nel file “databaseseeder.php” sono presenti i i valori dei dati utilizzati per fornire uno scopo dimostrativo dell’applicazione web realizzata.

In particolare si trovano informazioni relative a utenti, richieste di amicizia, relazioni di amicizia, blog, post e messaggi di notifica (non sono presenti messaggi relativi a creazione/cancellazione di blog/post ma è possibile testare facilmente anche questa funzionalità).

Per testare la funzione di ricerca di utenti tramite il loro nome suggeriamo di inserire i nomi “luigi” e “pippo” o loro sottostringhe (sia in carattere minuscolo che maiuscolo che misto) perché gli utenti creati automaticamente con i seeder hanno queste parti nei loro campi “name”.