

# The Aulab Post L11 - USER STORY #3

## User Story #3: 📝

- **Come** Marta
- **vorrei** poter contare su una funzione di fact-checking
- **in modo tale** da poter controllare quali notizie pubblicare

### ACCEPTANCE CRITERIA: 📝

- Tre nuovi ruoli: Admin, Revisor, Writer
- Un utente registrato richiede di entrare a far parte del team tramite un form di "lavora con noi"
- Creazione di una dashboard per il proprietario della piattaforma per poter gestire le richieste
- Gli utenti revisori avranno una sezione a loro dedicata con tutti gli articoli da revisionare
- Bottone accetta articolo
- Bottone rifiuta articolo

---

## USER STORY 3 - ADMIN, REVISOR E WRITER 📝

### GESTIONE RUOLI NEL DATABASE 📝

Per completare questa User Story dobbiamo creare **tre nuovi ruoli** all'interno del nostro progetto:

- **admin**, l'amministratore della piattaforma, che si occuperà di gestire le richieste di lavoro e che, in futuro, potrà aggiungere e modificare le categorie;
- **revisor**, gli editors della piattaforma, che si occuperanno di decidere se un articolo può essere pubblicato o meno;
- **writer**, i redattori, che si occuperanno di scrivere gli articoli.

Per poter gestire questi ruoli, creeremo delle flags all'interno della tabella users, ovvero delle colonne che accetteranno un dato booleano.

**i** L'idea e' che i campi flag che creeremo accetteranno solo tre valori

- `false`, il valore di default - di base un utente non ha nessun ruolo;
- `null`, se l'utente ha fatto richiesta per essere assunto per un ruolo;
- `true`, se l'utente ha ricevuto un ruolo;

Creiamo una **nuova migrazione**, lanciando da terminale il comando:

```
1 php artisan make:migration add_roles_to_users_table
```

Nella **funzione `up()`**, inseriremo le nuove colonne di tipo **booleano** che gestiranno i ruoli degli utenti. Dopo di che, concateniamo altri metodi: con `after()` decidiamo il posizionamento delle colonne nella tabella 'users', con `nullable()` diciamo che queste colonne possono anche avere **NULL** come valore, con `default()`, invece, diciamo il valore che devono avere di default che, in questo caso, devono essere tutti **false**. Inoltre, direttamente nella funzione `up()`, **creiamo un nuovo utente** con nome, mail e password criptata, che avrà il ruolo di amministratore.

```
1  public function up(): void
2  {
3      Schema::table('users', function (Blueprint $table) {
4          $table->boolean('is_admin')->after('email')->nullable()->default(false);
5          $table->boolean('is_revisor')->after('is_admin')->nullable()->default(false);
6          $table->boolean('is_writer')->after('is_revisor')->nullable()->default(false);
7      });
8
9      User::create([
10         'name' => 'Admin',
11         'email' => 'admin@theaulabpost.it',
12         'password' => bcrypt('12345678'),
13         'is_admin' => true
14     ]);
15 }
```

Importiamo la classe **User**

```
1  use App\Models\User;
```

Nella funzione `down()`, invece, cancelliamo prima l'utente admin creato e poi **droppiamo le colonne**:

```
1  public function down(): void
2  {
3      User::where('email', 'admin@theaulabpost.it')->delete();
4
5      Schema::table('users', function (Blueprint $table) {
6          $table->dropColumn(['is_admin', 'is_revisor', 'is_writer']);
7      });
8 }
```

Prima di lanciare le migrazioni, ricordiamoci di aggiornare il **fillable** nel modello **User**, inserendo le nuove colonne appena aggiunte nella migrazione:

```
● ● ●
1 protected $fillable = [
2     'name',
3     'email',
4     'password',
5     'is_admin',
6     'is_revisor',
7     'is_writer'
8 ];
```

Lanciamo quindi da terminale la serie di comandi per riportare queste nuove colonne nella tabella users:

```
1 php artisan migrate
2 php artisan migrate:rollback
3 php artisan migrate
```

## FORM PER L'UTENTE E INVIO MAIL ↗

Ora occupiamoci di fornire all'utente un modo per richiedere di poter diventare admin, revisor o writer.

La logica di questo sviluppo è la seguente:

1. Un **utente loggato** può chiedere di diventare admin, revisor o writer.
2. Quando l'utente farà richiesta per un ruolo specifico, nella colonna del ruolo scelto verrà assegnato il valore **NULL**.

Il primo passo, però, è creare una sezione "lavora con noi" in cui l'utente può scegliere per quale ruolo candidarsi. Partiamo sempre dalla rotta in **web.php**:

```
● ● ●
1 Route::get('/careers', [PublicController::class, 'careers'])->name('careers');
```

Andiamo nel **PublicController** e creiamo la **funzione** `careers()` che ritornerà la **vista careers**.

```
● ● ●
1 public function careers(){
2     return view('careers');
3 }
```

Questa funzione, però, sarà protetta dal middleware **auth**, quindi, anche il **PublicController** implementerà l'**interfaccia HasMiddleware** e inseriamo il metodo `middleware()` escludendo l'homepage:

```
● ● ●  
1 class PublicController extends Controller implements HasMiddleware  
2 {  
3     public static function middleware()  
4     {  
5         return [  
6             new Middleware('auth', except: ['homepage']),  
7         ];  
8     }  
}
```

Importiamo le classi:

```
● ● ●  
1 use Illuminate\Routing\Controllers\Middleware;  
2 use Illuminate\Routing\Controllers\HasMiddleware;
```

Creiamo quindi in **resources/views** un file chiamato `careers.blade.php`, dove andremo a mostrare un **form** dove l'utente loggato potrà scegliere il ruolo per cui candidarsi.

All'interno del form inseriamo **tre input**:

- uno in cui l'utente può **scegliere il ruolo** per cui fare richiesta;
- uno per la **mail** (precompilata, essendo l'utente loggato);
- uno per il **messaggio di presentazione** che l'utente manderà all'admin.

Anche in questo form, gli input avranno gli attributi `name`, l'helper `old()` e la direttiva `@error`.

```

1  <x-layout>
2      <div class="container-fluid p-5 bg-secondary-subtle text-center">
3          <div class="row justify-content-center">
4              <div class="col-12">
5                  <h1 class="display-1">Lavora con noi</h1>
6              </div>
7          </div>
8      </div>
9      <div class="container my-5">
10         <div class="row">
11             <div class="col-12 col-md-6">
12                 <form action="#" method="" class="card p-5 shadow">
13                     <div class="mb-3">
14                         <label for="role" class="form-label">Per quale ruolo ti stai candidando?</label>
15                         <select name="role" id="role" class="form-control">
16                             <option value="" selected disabled>Seleziona il ruolo</option>
17                             <option value="admin">Amministratore</option>
18                             <option value="revisor">Revisore</option>
19                             <option value="writer">Redattore</option>
20                         </select>
21                         @error('role')
22                             <span class="text-danger">{{ $message }}</span>
23                         @enderror
24                     </div>
25                     <div class="mb-3">
26                         <label for="email" class="form-label">Email</label>
27                         <input type="email" class="form-control" id="email" name="email" value="{{Auth::user()->email}}" disabled>
28                         @error('email')
29                             <span class="text-danger">{{ $message }}</span>
30                         @enderror
31                     </div>
32                     <div class="mb-3">
33                         <label for="message" class="form-label">Perché vuoi candidarti per questo ruolo? Raccontacelo!</label>
34                         <textarea name="message" id="message" cols="30" rows="7" class="form-control">{{old('message')}}</textarea>
35                         @error('message')
36                             <span class="text-danger">{{ $message }}</span>
37                         @enderror
38                     </div>
39                     <div class="mt-3 d-flex justify-content-center">
40                         <button type="submit" class="btn btn-outline-secondary">Invia candidatura</button>
41                     </div>
42                 </form>
43             </div>
44             <div class="col-12 col-md-6 p-5">
45                 <h2>Lavora come amministratore</h2>
46                 <p>Scegliendo di lavorare come amministratore, ti occuperai di gestire le richieste di lavoro e di aggiungere e modificare le categorie.</p>
47                 <h2>Lavora come revisore</h2>
48                 <p>Scegliendo di lavorare come revisore, deciderai se un articolo può essere pubblicato o meno in piattaforma.</p>
49                 <h2>Lavora come redattore</h2>
50                 <p>Scegliendo di lavorare come redattore, potrai scrivere gli articoli che saranno pubblicati.</p>
51             </div>
52         </div>
53     </div>
54 </x-layout>

```

Per raggiungere questa vista, inseriamo il **link nella navbar**, visibile anche da utenti non autenticati.

```

1  <li class="nav-item">
2      <a class="nav-link active" aria-current="page" href="{{route('careers')}}">Lavora con noi</a>
3  </li>

```

Adesso, andiamo a creare la **rotta post** che gestirà le informazioni inserite nel form. In **web.php**:

```

1  Route::post('/careers/submit', [PublicController::class, 'careersSubmit'])->name('careers.submit');

```

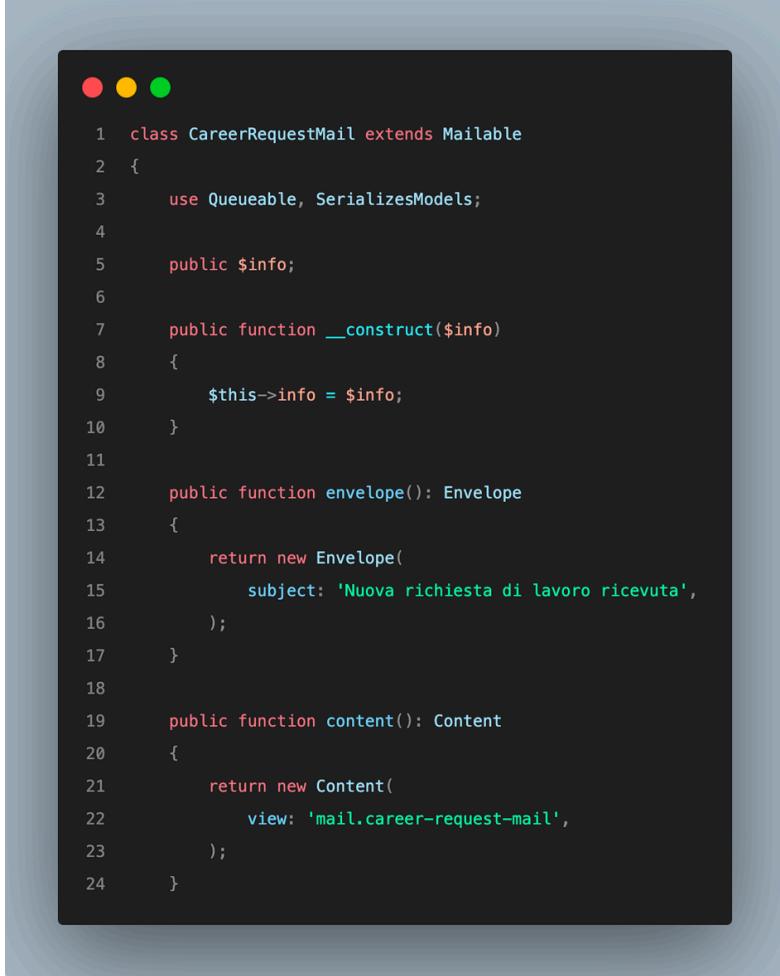
Andiamo a gestire la **funzione** `careersSubmit()` all'interno del **PublicController**, che prenderà i dati dalla **Request** e fermiamoci a gestire solo le **regole di validazione**:

```
1 public function careersSubmit(Request $request){  
2     $request->validate([  
3         'role' => 'required',  
4         'email' => 'required|email',  
5         'message' => 'required'  
6     ]);  
7 }
```

Questa funzione si occuperà di inviare una mail in cui verranno mostrate tutte le informazioni inviate dall'utente che ha compilato il form. Prima di completare questa funzione, quindi, dobbiamo creare una classe **Mailable**. Dal terminale, lanciamo il comando:

```
1 php artisan make:mail CareerRequestMail
```

All'interno della classe, andiamo prima di tutto ad inizializzare una **proprietà pubblica** che conterrà i dati dell'utente, poi gestiamo tutto il resto della logica:



```
1  class CareerRequestMail extends Mailable
2  {
3      use Queueable, SerializesModels;
4
5      public $info;
6
7      public function __construct($info)
8      {
9          $this->info = $info;
10     }
11
12     public function envelope(): Envelope
13     {
14         return new Envelope(
15             subject: 'Nuova richiesta di lavoro ricevuta',
16         );
17     }
18
19     public function content(): Content
20     {
21         return new Content(
22             view: 'mail.career-request-mail',
23         );
24     }

```

Cosa stiamo facendo:

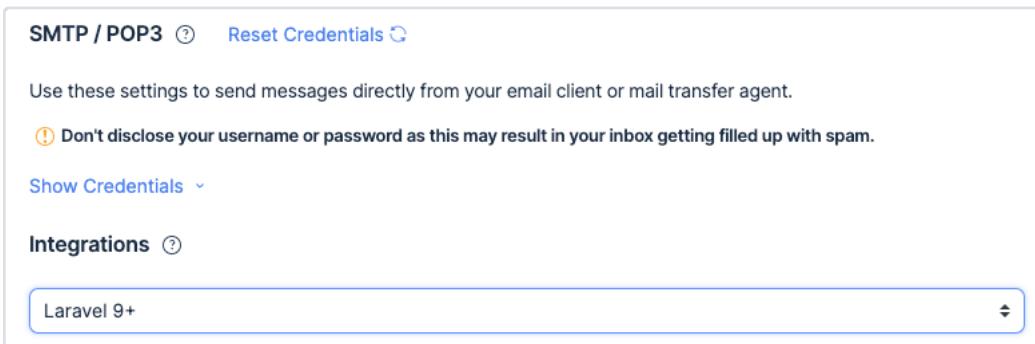
- nella funzione `__construct()` andiamo a specificare come la classe deve gestire le informazioni dell'utente;
- nella funzione `envelope()`, busta, andiamo a inserire le informazioni riguardo i dettagli della mail (come oggetto, cc, ccn ecc...);
- nella funzione `content()`, contenuto, andiamo a specificare quale vista vogliamo che l'utente visualizzi quando riceve la mail.

Ed è proprio ciò che andremo a gestire ora, infatti andiamo in `resources/views/` e creiamo una nuova cartella `mail` dove creeremo un file `career-request-mail.blade.php`. In questa vista non possiamo utilizzare i componenti e Bootstrap, e possiamo solo inserire CSS e JS inline o embedded. Per questo andremo ad inserire un **template HTML** con del semplice testo per visualizzare le informazioni che l'utente ha inserito nel form.



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Document</title>
7 </head>
8 <body>
9   <h1>È arrivata una richiesta per il ruolo di {{$info['role']}}</h1>
10  <p>Ricevuta da {{$info['email']}}</p>
11  <h4>Messaggio:</h4>
12  <p>{{$info['message']}}</p>
13 </body>
14 </html>
```

Successivamente, dobbiamo settare il file `.env` inserendo le nostre credenziali per l'invio della mail tramite il nostro provider di email, **Mailtrap**. Andare quindi sul sito di [Mailtrap](#) (se non siete ancora iscritti, fatelo, è gratis), nella sezione **Email Testing**, cliccare sulla vostra **inbox**. Nella sezione **Integrations** aprire il menu a tendina e selezionare **Laravel 9+**.



**SMTP / POP3** ⓘ [Reset Credentials](#) ⓘ

Use these settings to send messages directly from your email client or mail transfer agent.

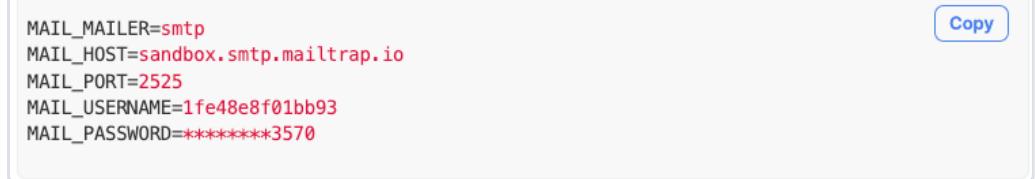
⚠ Don't disclose your username or password as this may result in your inbox getting filled up with spam.

Show Credentials ⓘ

**Integrations** ⓘ

Laravel 9+ ⏺

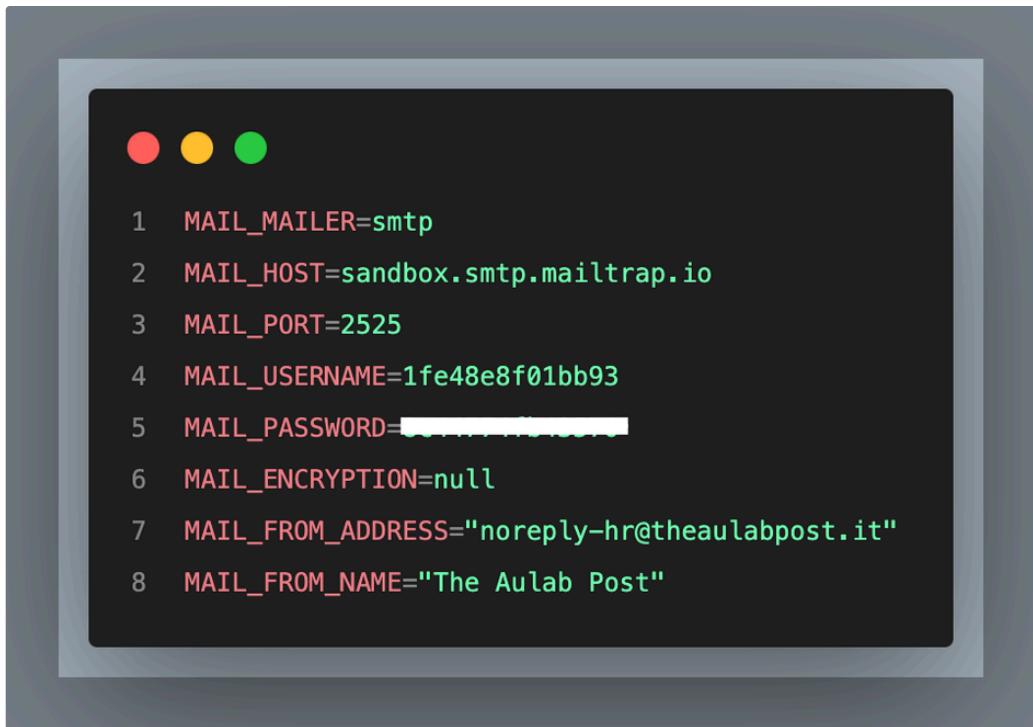
Vi compariranno le credenziali del vostro account di MailTrap da inserire nel vostro progetto.



```
MAIL_MAILER=smtp
MAIL_HOST=sandbox.smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=1fe48e8f01bb93
MAIL_PASSWORD=*****3570
```

⚠ Usa il bottone “Copy” per copiare le credenziali, altrimenti non verranno copiate correttamente.

Inserire le credenziali copiate nella sezione apposita nel file `.env`:



```
1 MAIL_MAILER=smtp
2 MAIL_HOST=sandbox.smtp.mailtrap.io
3 MAIL_PORT=2525
4 MAIL_USERNAME=1fe48e8f01bb93
5 MAIL_PASSWORD=[REDACTED]
6 MAIL_ENCRYPTION=null
7 MAIL_FROM_ADDRESS="noreply-hr@theaulabpost.it"
8 MAIL_FROM_NAME="The Aulab Post"
```

Adesso, torniamo nella funzione `careersSubmit()` nel **PublicController** e la completiamo per permettere l'invio della mail.

Dopo le regole di validazione, per prima cosa accediamo ai dati dell'**utente loggato** che fa la richiesta, poi prendiamo le informazioni che inserisce nel form tramite la **request** e tramite il metodo `compact()` le "compattiamo" in un'unica variabile. Poi, inviamo la **mail** con le info che ci ha inviato l'utente all'indirizzo di un amministratore della piattaforma.

Inoltre, in base al ruolo selezionato dall'utente, tramite uno `switch()` settiamo a `NULL` la colonna corrispondente nella tabella users. Infine, aggiorniamo il record dell'utente e lo rimandiamo all'**homepage** con un **messaggio di successo**.

```
1  public function careersSubmit(Request $request)
2  {
3      $request->validate([
4          'role' => 'required',
5          'email' => 'required|email',
6          'message' => 'required'
7      ]);
8
9      $user = Auth::user();
10     $role = $request->role;
11     $email = $request->email;
12     $message = $request->message;
13     $info = compact('role', 'email', 'message');
14
15     Mail::to('admin@theaulabpost.it')->send(new CareerRequestMail($info));
16
17     switch ($role) {
18         case 'admin':
19             $user->is_admin = NULL;
20             break;
21         case 'revisor':
22             $user->is_revisor = NULL;
23             break;
24         case 'writer':
25             $user->is_writer = NULL;
26             break;
27     }
28
29     $user->update();
30     return redirect(route('homepage'))->with('message', 'Mail inviata con successo!');
31 }
```

Ricordiamoci di importare le classi:

```
1  use App\Mail\CareerRequestMail;
2  use Illuminate\Support\Facades\Auth;
3  use Illuminate\Support\Facades\Mail;
```

Inoltre, prima di testare che tutto funzioni, ricordiamoci di tornare nella vista **careers.blade.php** e di inserire la **action**, il **method** e il **csrf token** nel **form**.

```
● ● ●
1 <form action="{{route('careers.submit')}}" method="POST" class="card p-5 shadow">
2     @csrf
```

Per non permettere ad utente di poter fare di nuovo richiesta per un ruolo che già ricopre, possiamo inserire nella select del **form** in **careers.blade.php** un controllo nelle option per selezionare gli utenti:

```
● ● ●
1 ...
2 <select name="role" id="role" class="form-control">
3     <option value="" selected disabled>Seleziona il ruolo</option>
4     @if (!Auth::user()->is_admin)
5         <option value="admin">Amministratore</option>
6     @endif
7     @if (!Auth::user()->is_revisor)
8         <option value="revisor">Revisore</option>
9     @endif
10    @if (!Auth::user()->is_writer)
11        <option value="writer">Redattore</option>
12    @endif
13 </select>
14 ...
```

Quindi, se un utente ha già quel ruolo, la option non è visibile e quindi non è cliccabile.

Adesso, è arrivato il momento di testare tutto.

## ADMIN DASHBOARD ☰

A questo punto, però, dobbiamo permettere all'utente admin di poter vedere tutte le richieste di collaborazione fatte dagli utenti.

Procediamo quindi col creare le rotte, le viste e le funzioni con cui l'admin gestirà gli utenti. Queste rotte e queste funzioni devono essere protette però in modo che solo un utente con ruolo di admin vi possa accedere.

Abbiamo quindi la necessità di creare un **middleware** custom che controlli, appunto, se l'utente è un amministratore. da terminale lanciamo il comando:

```
1 php artisan make:middleware UserIsAdmin
```

Ciò che dobbiamo specificare nel middleware è che la request può proseguire solo se l'utente loggato ha il ruolo di admin. Nel file appena creato, quindi, modifichiamo la **funzione handle()**:

```
● ● ●  
1 public function handle(Request $request, Closure $next): Response  
2 {  
3     if(Auth::user() && Auth::user()->is_admin){  
4         return $next($request);  
5     }  
6     return redirect(route('homepage'))->with('alert', 'Non sei autorizzato');  
7 }
```

Importare:

```
● ● ●  
1 use Illuminate\Support\Facades\Auth;
```

Una volta creato il middleware, dobbiamo registrarlo in modo da renderlo disponibile e quindi utilizzabile.

Nel file `bootstrap/app.php` registriamo il nostro middleware, inserendo all'interno del metodo `withMiddleware()` un array di middleware specificando il loro **alias**, come chiave, e il percorso del middleware che vogliamo registrare, come valore.

```
● ● ●  
1 ...  
2 ->withMiddleware(function (Middleware $middleware) {  
3     $middleware->alias([  
4         'admin' => App\Http\Middleware\UserIsAdmin::class,  
5     ]);  
6 })  
7 ...
```

Ricordiamoci di inserire il messaggio da far visualizzare nell'homepage:

```
1 @if (session('alert'))
2     <div class="alert alert-danger">
3         {{ session('alert') }}
4     </div>
5 @endif
```

Per una questione di pulizia, creiamo anche un **controller dedicato alle rotte dell'Admin**, così da dividere per bene le logiche dei vari utenti. Da terminale lanciamo:

```
1 php artisan make:controller AdminController
```

Adesso vediamo un nuovo metodo per gestire le rotte, ovvero il **grouping**. Ciò che vogliamo fare è gestire un gruppo di rotte che verrà automaticamente protetto dal middleware creato, quindi senza utilizzare la funzione statica `middleware()` nel controller. Oltre a creare il gruppo, gestiamo anche una rotta che porterà l'admin alla sua dashboard personale. In `web.php` scriviamo:

```
1 Route::middleware('admin')->group(function(){
2     Route::get('/admin/dashboard', [AdminController::class, 'dashboard'])->name('admin.dashboard');
3 });
```

E importiamo il controller:

```
1 use App\Http\Controllers\AdminController;
```

Adesso nell'**AdminController**, creiamo la **funzione** `dashboard()` dove andiamo a gestire la logica dei dati da mostrare all'amministratore, ovvero tutti quegli utenti che hanno fatto richiesta per diventare amministratori, revisori o redattori:

```
1 public function dashboard(){
2     $adminRequests = User::where('is_admin', NULL)->get();
3     $revisorRequests = User::where('is_revisor', NULL)->get();
4     $writerRequests = User::where('is_writer', NULL)->get();
5     return view('admin.dashboard', compact('adminRequests', 'revisorRequests', 'writerRequests'));
6 }
```

Importiamo:

```
1 use App\Models\User;
```

Adesso andiamo a creare in `resources/views/` una directory chiamata `admin`. Al suo interno creeremo invece il file `dashboard.blade.php`. In questo file avremo **tre tabelle**, con al loro interno le richieste dei vari utenti. Essendo un elemento che si ripete, possiamo sfruttare i componenti. Andiamo quindi a creare nella cartella `resources/views/components` un file chiamato `requests-table.blade.php`. Andiamo su Bootstrap e ci prendiamo una **tabella** da incollare nel file del componente. Nella vista della `dashboard`, richiamiamo tre volte il componente della **tabella**, passandogli ogni volta due dati diversi: le richieste che ci arrivano (un tipo per ogni componente) e il ruolo.

```
 1  <x-layout>
 2      <div class="container-fluid p-5 bg-secondary-subtle text-center">
 3          <div class="row justify-content-center">
 4              <div class="col-12">
 5                  <h1 class="display-1">Bentornato, Amministratore {{Auth::user()>name}}</h1>
 6              </div>
 7          </div>
 8      </div>
 9      <div class="container my-5">
10          <div class="row justify-content-center">
11              <div class="col-12">
12                  <h2>Richieste per il ruolo di amministratore</h2>
13                  <x-requests-table :roleRequests="$adminRequests" role="amministratore"/>
14              </div>
15          </div>
16      </div>
17      <div class="container my-5">
18          <div class="row justify-content-center">
19              <div class="col-12">
20                  <h2>Richieste per il ruolo di revisore</h2>
21                  <x-requests-table :roleRequests="$revisorRequests" role="revisore"/>
22              </div>
23          </div>
24      </div>
25      <div class="container my-5">
26          <div class="row justify-content-center">
27              <div class="col-12">
28                  <h2>Richieste per il ruolo di redattore</h2>
29                  <x-requests-table :roleRequests="$writerRequests" role="redattore"/>
30              </div>
31          </div>
32      </div>
33  </x-layout>
```

I tre componenti, quindi, non sono altro che tre tabelle diverse che mostrano i dati in base al tipo di richiesta. Abbiamo visto che abbiamo la possibilità di valorizzare degli attributi con dei dati semplici quali stringhe e integer (`role`, per esempio). Ma noi abbiamo la necessità di passare anche dei dati complessi a questi componenti: è per questo che utilizziamo una nuova sintassi `:roleRequests`, che consente al componente di accettare un dato più strutturato, come un array o un oggetto. Così facendo abbiamo un componente uguale per tutti ma che ogni volta utilizza dei dati diversi e dinamici.

Adesso, modifichiamo il file della tabella: per ogni richiesta **cicliamo** le righe della tabella mostrando i rispettivi dati, inserendo anche un bottone al momento non attivo.

```

1 <table class="table table-striped table-hover">
2   <thead class="table-dark">
3     <tr>
4       <th scope="col">#</th>
5       <th scope="col">Nome</th>
6       <th scope="col">Email</th>
7       <th scope="col">Azioni</th>
8     </tr>
9   </thead>
10  <tbody>
11    @foreach ($roleRequests as $user)
12      <tr>
13        <th scope="row">{{$user->id}}</th>
14        <td>{{$user->name}}</td>
15        <td>{{$user->email}}</td>
16        <td>
17          <button class="btn btn-secondary">Attiva {{$role}}</button>
18        </td>
19      </tr>
20    @endforeach
21  </tbody>
22 </table>

```

## ATTIVARE IL RUOLO SCELTO AGLI UTENTI ↗

Adesso che abbiamo ricevuto correttamente le richieste dagli utenti e che possiamo visualizzarle, è il momento di permettere all'admin di accettare (o meno) le richieste che gli arrivano. Dovremo quindi gestire tre logiche, una per ogni ruolo. Cominciamo dalla richiesta per diventare amministratore. Andiamo nel file `web.php` e nel gruppo delle rotte con `middleware admin` aggiungiamo le tre rotte che gestiscono rispettivamente le richieste per diventare amministratore, revisore e redattore:

```

1 Route::middleware('admin')->group(function(){
2   ...
3   Route::patch('/admin/{user}/set-admin', [AdminController::class, 'setAdmin'])->name('admin.setAdmin');
4   Route::patch('/admin/{user}/set-revisor', [AdminController::class, 'setRevisor'])->name('admin.setRevisor');
5   Route::patch('/admin/{user}/set-writer', [AdminController::class, 'setWriter'])->name('admin.setWriter');
6 });

```

Andiamo quindi nell'**AdminController** a inserire le funzioni:

```
1 public function setAdmin(User $user){
2     $user->is_admin = true;
3     $user->save();
4     return redirect(route('admin.dashboard'))->with('message', "Hai reso $user->name amministratore");
5 }
6
7 public function setRevisor(User $user){
8     $user->is_revisor = true;
9     $user->save();
10    return redirect(route('admin.dashboard'))->with('message', "Hai reso $user->name revisore");
11 }
12
13 public function setWriter(User $user){
14     $user->is_writer = true;
15     $user->save();
16     return redirect(route('admin.dashboard'))->with('message', "Hai reso $user->name redattore");
17 }
```

Inseriamo nella vista della **dashboard** il **messaggio** da far comparire:

```
1 <x-layout>
2     <div class="container-fluid p-5 bg-secondary-subtle text-center">
3         <div class="row justify-content-center">
4             <div class="col-12">
5                 <h1 class="display-1">Bentornato, Amministratore {{Auth::user()->name}}</h1>
6             </div>
7         </div>
8     </div>
9     @if (session('message'))
10        <div class="alert alert-success">
11            {{ session('message') }}
12        </div>
13    @endif
14    ...
15
```

Ora, possiamo modificare il **componente della tabella** rendendo cliccabili i bottoni. Utilizziamo uno **switch case** per gestire ogni volta la rotta in maniera dinamica: in questo modo, se saremo nel componente dedicato agli admin, il link ci permetterà di rendere amministratore l'utente, e così anche per gli altri ruoli.

```

1   ...
2   <tbody>
3       @foreach ($roleRequests as $user)
4           <tr>
5               <th scope="row">{$user->id}</th>
6               <td>{$user->name}</td>
7               <td>{$user->email}</td>
8               <td>
9                   @switch($role)
10                      @case('amministratore')
11                          <form action="{{route('admin.setAdmin', $user)}}" method="POST">
12                              @csrf
13                              @method('PATCH')
14                              <button type="submit" class="btn btn-secondary">Attiva {$role}</button>
15                          </form>
16                      @break
17                      @case('revisore')
18                          <form action="{{route('admin.setRevisor', $user)}}" method="POST">
19                              @csrf
20                              @method('PATCH')
21                              <button type="submit" class="btn btn-secondary">Attiva {$role}</button>
22                          </form>
23                      @break
24                      @case('redattore')
25                          <form action="{{route('admin.setWriter', $user)}}" method="POST">
26                              @csrf
27                              @method('PATCH')
28                              <button type="submit" class="btn btn-secondary">Attiva {$role}</button>
29                          </form>
30                      @break
31                  @endswitch
32              </td>
33          </tr>
34      @endforeach
35  </tbody>
36 </table>

```

Ora, quindi, l'utente admin può dare i ruoli agli utenti che hanno fatto richiesta.

Ricordiamoci di inserire un link nella **navbar** che ci permetta di raggiungere questa rotta se loggati come utente Admin, potremmo ad esempio sfruttare il dropdown della navbar all'interno della direttiva **@auth**:

```

1 @if (Auth::user()->is_admin)
2     <li><a class="dropdown-item" href="{{route('admin.dashboard')}}">Dashboard Admin</a></li>
3 @endif

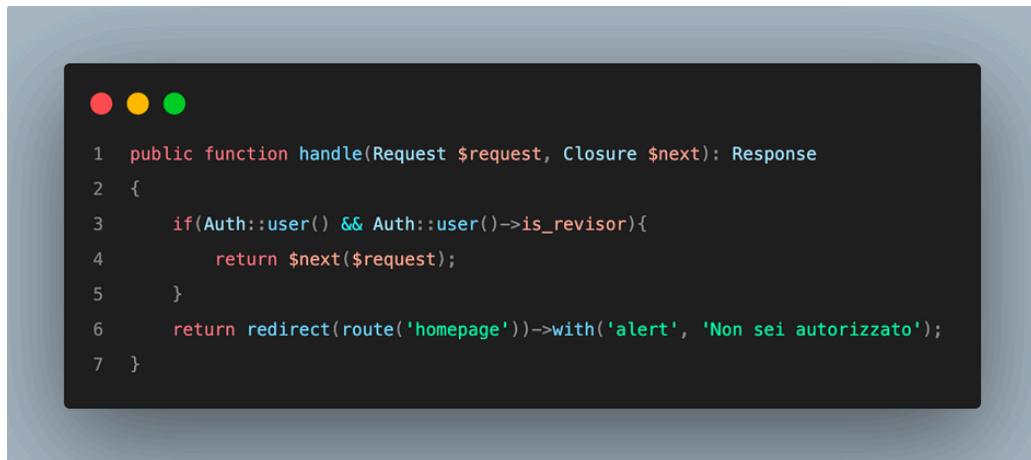
```

## REVISOR DASHBOARD ↗

Adesso concentriamoci sul ruolo di Revisore. Il compito del revisore sarà quello di controllare se gli articoli scritti dagli utenti rispettano le policy aziendali e possono essere quindi pubblicati. Così come abbiamo fatto per l'Amministratore, anche qui dobbiamo creare un middleware dedicato. Lanciamo da terminale:

```
1 php artisan make:middleware UserIsRevisor
```

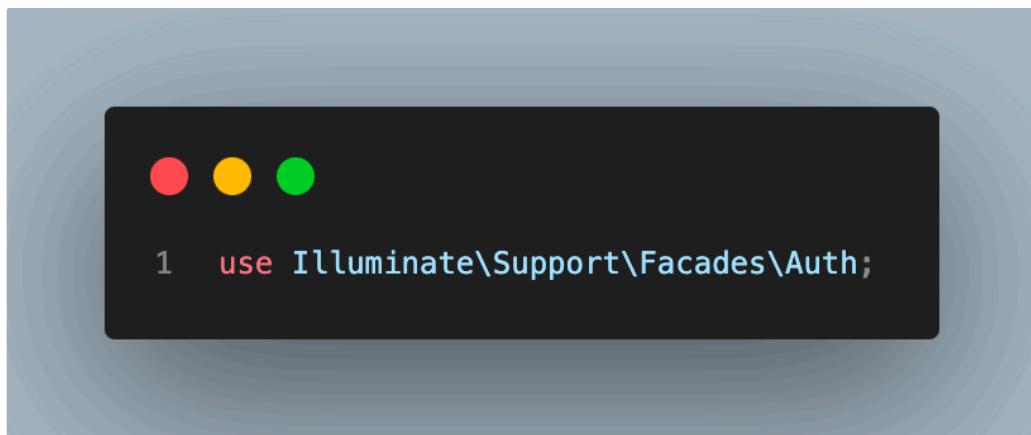
Ciò che dobbiamo specificare nel middleware è che la request può proseguire solo se l'utente loggato ha il ruolo di revisore. Nel file appena creato, quindi, modifichiamo la **funzione** `handle()`:



```
● ● ●

1 public function handle(Request $request, Closure $next): Response
2 {
3     if(Auth::user() && Auth::user()->is_revisor){
4         return $next($request);
5     }
6     return redirect(route('homepage'))->with('alert', 'Non sei autorizzato');
7 }
```

Importare:



```
● ● ●

1 use Illuminate\Support\Facades\Auth;
```

Una volta creato il middleware, dobbiamo registrarlo come abbiamo fatto per quello dell'amministratore.

Nel file `bootstrap/app.php` registriamo il nostro middleware, inserendo all'interno del metodo `withMiddleware()` una nuova chiave con il percorso del middleware che abbiamo creato.



```
● ● ●

1 ->withMiddleware(function (Middleware $middleware) {
2     $middleware->alias([
3         ...
4         'revisor' => App\Http\Middleware\UserIsRevisor::class,
5     ]);
6 })
```

L'idea è che un articolo non sia automaticamente disponibile alla sua creazione ma che, invece, passi prima dalla revisione di utenti autorizzati. Ma come facciamo a gestire questa differenza? Aggiungiamo una colonna `is_accepted` all'interno della tabella `articles` che

avrà tre possibili valori:

- **NULL**, l'articolo è in attesa di revisione;
- **true**, l'articolo è stato accettato;
- **false**, l'articolo è stato respinto.

Cominciamo con il creare la **migrazione** con il comando da terminale:

```
1 php artisan make:migration add_is_accepted_to_articles_table
```

Nella **migrazione**, nella funzione `up()`, aggiungiamo la nuova **colonna** con tipo di dato **booleano**, che si posizionerà dopo la colonna `user_id` e che può accettare come valore **NULL**. Nella funzione `down()`, invece, **droppiamo** la colonna.



```
1 public function up(): void
2 {
3     Schema::table('articles', function (Blueprint $table) {
4         $table->boolean('is_accepted')->after('user_id')->nullable();
5     });
6 }
7
8 /**
9  * Reverse the migrations.
10 */
11 public function down(): void
12 {
13     Schema::table('articles', function (Blueprint $table) {
14         $table->dropColumn('is_accepted');
15     });
16 }
```

Lanciamo quindi la serie di comandi:

```
1 php artisan migrate
2 php artisan migrate:rollback
3 php artisan migrate
```

Andiamo ad aggiornare il `fillable` nel modello **Article**, aggiungendo la nuova colonna:



```
1 protected $fillable = [
2     'title', 'subtitle', 'body', 'image', 'user_id', 'category_id', 'is_accepted'
3 ];
```

Andiamo adesso a gestire tutta la logica dietro la revisione degli articoli, quindi creiamo un **RevisorController** che gestirà un gruppo di rotte protette dal middleware `UserIsRevisor`. Da terminale:

```
1 php artisan make:controller RevisorController
```

In `web.php`, creare il gruppo delle rotte con middleware `revisor` e inserire la rotta che ci porterà alla **dashboard del revisore**:

```
1 Route::middleware('revisor')->group(function(){
2     Route::get('/revisor/dashboard', [RevisorController::class, 'dashboard'])->name('revisor.dashboard');
3 });
```

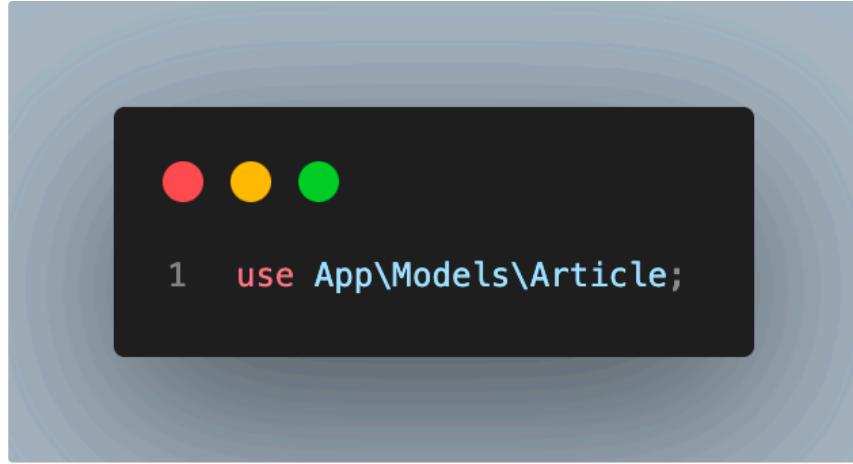
Importare il controller:

```
1 use App\Http\Controllers\RevisorController;
```

Andiamo nel **RevisorController** e creiamo la funzione dove andremo a gestire la logica degli articoli suddividendoli in articoli “da revisionare”, “accettati”, “rifiutati”, poi ritornare la dashboard del revisore passandogli tutti i dati degli articoli.

```
1 public function dashboard(){
2     $unrevisionedArticles = Article::where('is_accepted', NULL)->get();
3     $acceptedArticles = Article::where('is_accepted', true)->get();
4     $rejectedArticles = Article::where('is_accepted', false)->get();
5
6     return view('revisor.dashboard', compact('unrevisionedArticles', 'acceptedArticles', 'rejectedArticles'));
7 }
```

Importare la classe **Article**:



Adesso andiamo a creare in `resources/views/` una directory chiamata `revisor`. Al suo interno creeremo invece il file `dashboard.blade.php`. Anche in questo file avremo **tre tavole**, con al loro interno gli articoli accettati, rifiutati o da revisionare. Creiamo nella cartella `resources/views/components` un file chiamato `articles-table.blade.php`, dove inseriamo una **tavola**. Nella vista della **dashboard del revisore**, richiamiamo tre volte il componente della **tavola degli articoli**, passandogli ogni volta l'articolo a seconda se è stato accettato o meno

```
1  <x-layout>
2      <div class="container-fluid p-5 bg-secondary-subtle text-center">
3          <div class="row justify-content-center">
4              <div class="col-12">
5                  <h1 class="display-1">Bentornato, Revisore {{Auth::user()>name}}</h1>
6              </div>
7          </div>
8      </div>
9      <div class="container my-5">
10         <div class="row justify-content-center">
11             <div class="col-12">
12                 <h2>Articoli da revisionare</h2>
13                 <x-articles-table :articles="$unrevisedArticles"/>
14             </div>
15         </div>
16     </div>
17     <div class="container my-5">
18         <div class="row justify-content-center">
19             <div class="col-12">
20                 <h2>Articoli pubblicati</h2>
21                 <x-articles-table :articles="$acceptedArticles"/>
22             </div>
23         </div>
24     </div>
25     <div class="container my-5">
26         <div class="row justify-content-center">
27             <div class="col-12">
28                 <h2>Articoli respinti</h2>
29                 <x-articles-table :articles="$rejectedArticles"/>
30             </div>
31         </div>
32     </div>
33 </x-layout>
```

Adesso, modifichiamo il file della tabella: per ogni stato dell'articolo **cicliamo** le righe della tabella mostrando i rispettivi dati. Inoltre, se l'articolo è ancora da revisionare, potremo visualizzare il link che ci porterà al dettaglio, altrimenti potremo ripotare l'articolo in revisione.

```
1 <table class="table table-striped table-hover">
2   <thead class="table-dark">
3     <tr>
4       <th scope="col">#</th>
5       <th scope="col">Titolo</th>
6       <th scope="col">Sottotitolo</th>
7       <th scope="col">Redattore</th>
8       <th scope="col">Azioni</th>
9     </tr>
10    </thead>
11    <tbody>
12      @foreach ($articles as $article)
13        <tr>
14          <th scope="row">{{$article->id}}</th>
15          <td>{{$article->title}}</td>
16          <td>{{$article->subtitle}}</td>
17          <td>{{$article->user->name}}</td>
18          <td>
19            @if (is_null($article->is_accepted))
20              <a href="#" class="btn btn-secondary">Leggi l'articolo</a>
21            @else
22              <a href="#" class="btn btn-secondary">Riporta in revisione</a>
23            @endif
24          </td>
25        </tr>
26      @endforeach
27    </tbody>
28  </table>
```

Adesso ciò che c'è da fare è **gestire la logica di accettazione** degli articoli: sfrutteremo la vista di dettaglio per permettere al revisore di leggere l'articolo e vi inseriremo, solo per lui, una serie di button per accettare e rifiutare.

Per prima cosa, possiamo completare l'if nel componente `articles-table` con la rotta del dettaglio:

```
1 <td>
2   @if (is_null($article->is_accepted))
3     <a href="{{route('article.show', $article)}}" class="btn btn-secondary">Leggi l'articolo</a>
4   @else
5     <a href="#" class="btn btn-secondary">Riporta in revisione</a>
6   @endif
7 </td>
```

Adesso creiamo le **rotte**, protette dal middleware `revisor`, che permetteranno al revisore di accettare un articolo, rifiutarlo o rimandarlo in revisione. In `web.php`:

```
1 Route::middleware('revisor')->group(function(){
2     ...
3     Route::post('/revisor/{article}/accept', [RevisorController::class, 'acceptArticle'])->name('revisor.acceptArticle');
4     Route::post('/revisor/{article}/reject', [RevisorController::class, 'rejectArticle'])->name('revisor.rejectArticle');
5     Route::post('/revisor/{article}/undo', [RevisorController::class, 'undoArticle'])->name('revisor.undoArticle');
6 });
});
```

Nel **RevisorController** gestiamo le tre **funzioni** per gestire lo stato degli articoli con un messaggio di successo:

```
1 public function acceptArticle(Article $article){
2     $article->is_accepted = true;
3     $article->save();
4     return redirect(route('revisor.dashboard'))->with('message', 'Articolo pubblicato');
5 }
6
7 public function rejectArticle(Article $article){
8     $article->is_accepted = false;
9     $article->save();
10    return redirect(route('revisor.dashboard'))->with('message', 'Articolo rifiutato');
11 }
12
13 public function undoArticle(Article $article){
14     $article->is_accepted = NULL;
15     $article->save();
16     return redirect(route('revisor.dashboard'))->with('message', 'Articolo rimandato in revisione');
17 }
```

Inseriamo nella vista della **dashboard** del revisore il **messaggio** da far comparire:

```
1 <x-layout>
2     <div class="container-fluid p-5 bg-secondary-subtle text-center">
3         <div class="row justify-content-center">
4             <div class="col-12">
5                 <h1 class="display-1">Bentornato, Revisore {{Auth::user()->name}}</h1>
6             </div>
7         </div>
8     </div>
9     @if (session('message'))
10         <div class="alert alert-success">
11             {{ session('message') }}
12         </div>
13     @endif
14     ...
```

Fatto questo, modifichiamo il link nella direttiva **@else** del componente `articles-table` con un form:

```
1  <td>
2      @if ($article->is_accepted)
3          <a href="{{route('article.show', $article)}}" class="btn btn-secondary">Leggi l'articolo</a>
4      @else
5          <form action="{{route('revisor.undoArticle', $article)}}" method="POST">
6              @csrf
7              <button type="submit" class="btn btn-secondary">Riporta in revisione</button>
8          </form>
9      @endif
10     </td>
```

Per poter permettere al revisore, invece, di poter accettare o rifiutare l'articolo, inseriamo, sotto al body dell'articolo ad esempio, due **bottoni** nella vista **show** visibili solo se l'utente è un **revisore**.

```
1  ...
2  <p>$article->body</p>
3  @if (Auth::user() && Auth::user()->is_revisor)
4      <div class="container my-5">
5          <div class="row">
6              <div class="col-12 d-flex justify-content-evenly">
7                  <form action="{{route('revisor.acceptArticle', $article)}}" method="POST">
8                      @csrf
9                      <button type="submit" class="btn btn-success">Accetta l'articolo</button>
10                 </form>
11                 <form action="{{route('revisor.rejectArticle', $article)}}" method="POST">
12                     @csrf
13                     <button type="submit" class="btn btn-danger">Rifiuta l'articolo</button>
14                 </form>
15             </div>
16         </div>
17     </div>
18 @endif
19 ...
```

Abbiamo completato la logica di revisione degli articoli, ma possiamo notare che nelle pagine del nostro sito sono visibili ancora tutti gli articoli, anche quelli non ancora revisionati o rifiutati. Dobbiamo aggiornare la logica che c'è dietro le query al database.

Nel **PublicController**, nella funzione `homepage()`, prima di ordinare gli articoli e passarli alla vista, tramite il metodo `where()` ci assicuriamo di prendere solo gli articoli accettati:

```
1  public function homepage()
2  {
3      $articles = Article::where('is_accepted', true)->orderBy('created_at', 'desc')->take(4)->get();
4      return view('welcome', compact('articles'));
5 }
```

Lo stesso facciamo nelle funzioni `index()`, `byCategory()` e `byUser()` nell'**ArticleController**:

```
1 public function index()
2 {
3     $articles = Article::where('is_accepted', true)->orderBy('created_at', 'desc')->get();
4     return view('article.index', compact('articles'));
5 }
6 ...
7 public function byCategory(Category $category){
8     $articles = $category->articles()->where('is_accepted', true)->orderBy('created_at', 'desc')->get();
9     return view('article.by-category', compact('category', 'articles'));
10 }
11
12 public function byUser(User $user){
13     $articles = $user->articles()->where('is_accepted', true)->orderBy('created_at', 'desc')->get();
14     return view('article.by-user', compact('user', 'articles'));
15 }
```

Ora, quindi, l'utente revisor può gestire in autonomia tutti gli articoli inseriti in piattaforma. Ricordiamoci di inserire un link nella **navbar** che ci permetta di raggiungere questa rotta se loggati come utente Revisor. Sempre nel dropdown della navbar all'interno della direttiva `@auth`:

```
1 @if (Auth::user()->is_revisor)
2     <li><a class="dropdown-item" href="{{route('revisor.dashboard')}}">Dashboard Revisor</a></li>
3 @endif
```

**⚠️** Assicuriamoci che in ogni vista disponibile al pubblico ci siano solo articoli accettati.

## GESTIONE DEL WRITER ↗

Così come abbiamo creato un middleware dedicato ad admin e revisor, facciamo la stessa cosa per i writers. Da terminale lanciamo il comando:

```
1 php artisan make:middleware UserIsWriter
```

Ciò che dobbiamo specificare nel middleware è che la request può proseguire solo se l'utente loggato ha il ruolo di redattore. Nel file appena creato, quindi, modifichiamo la funzione `handle()`:

```
● ● ●  
1 public function handle(Request $request, Closure $next): Response  
2 {  
3     if(Auth::user() && Auth::user()->is_writer){  
4         return $next($request);  
5     }  
6     return redirect(route('homepage'))->with('alert', 'Non sei autorizzato');  
7 }
```

Importare:

```
1 use Illuminate\Support\Facades\Auth;
```

Una volta creato il middleware, dobbiamo registrarlo come abbiamo fatto per gli altri due.

Nel file `bootstrap/app.php` registriamo il nostro middleware, inserendo all'interno del metodo `withMiddleware()` una nuova chiave con il percorso del middleware che abbiamo creato.

```
● ● ●  
1 ->withMiddleware(function (Middleware $middleware) {  
2     $middleware->alias([  
3         ...  
4         'writer' => App\Http\Middleware\UserIsWriter::class,  
5     ]);  
6 })
```

Una volta che lo abbiamo creato, possiamo includere al suo interno quelle rotte che permettono la creazione di un articolo e il suo salvataggio nel database. Creiamo quindi un nuovo gruppo, e spostiamo le **rotte** `article.create` e `article.store` già create in precedenza:

```
● ● ●
```

```
1 Route::middleware('writer')->group(function(){
2     Route::get('/article/create', [ArticleController::class, 'create'])->name('article.create');
3     Route::post('/article/store', [ArticleController::class, 'store'])->name('article.store');
4 });

```

Abbiamo completato la terza User Story, possiamo pushare:

```
1 git add .
2 git commit -m "User Story 3 completata"
3 git push
```

**Fine User Story 3**

---