

# The Aulab Post L11 - USER STORY #1

## User Story #1: [🔗](#)


- Come Sara
- vorrei registrarmi in piattaforma per inserire un articolo,
- in modo tale da lavorare per The Aulab Post.

## ACCEPTANCE CRITERIA: [🔗](#)

- Utente deve potersi registrare/loggare in piattaforma
  - Bottone "inserisci articolo" in home solo a utenti loggati
  - Articolo composto da:
    - Titolo
    - Sottotitolo
    - Corpo del Testo
    - Immagine di copertina
  - Categorie dell'annuncio pre-compilate
  - La relazione tra Categoria e Annuncio è 1-a-N
  - La relazione tra Utente e Annuncio è 1-a-N
  - Ad annuncio inserito visualizzare un messaggio di conferma
- 

## SVOLGIMENTO: [🔗](#)

### LEGENDA: [🔗](#)

 Informazioni e approfondimenti sul codice scritto o sulle best practices.

 Leggi sempre gli avvertimenti: possono evitarti un sacco di problemi!

 Consigli del docente per semplificarti la vita.

## SCAFFOLDING [🔗](#)

Creiamo un nuovo progetto con il comando **laravel new**:

```
1 laravel new aulab_post_cognome1_cognome2...
```

Ci chiederà se vogliamo installare uno starter kit e selezioniamo **"No starter kit"**

```
Would you like to install a starter kit? _____  
> • No starter kit  
  ◦ Laravel Breeze  
  ◦ Laravel Jetstream
```

Ci chiederà quale framework di testing vogliamo utilizzare e selezioniamo **"PHPUnit"**

```
Which testing framework do you prefer? _____  
  ◦ Pest  
> • PHPUnit
```

Ci chiederà se vogliamo inizializzare una repository Git e selezioniamo **"No"**

```
Would you like to initialize a Git repository? _____  
  ◦ Yes / • No
```

Poi, ci chiederà se vogliamo pubblicare i file di configurazione di Laravel e selezioniamo **"No"**

```
Would you like to publish Laravel's configuration files? _____  
  ◦ Yes / • No
```

Ci chiederà quale database vogliamo utilizzare e selezioniamo **"MySQL"**

```
Which database will your application use?
> ● MySQL
  ○ MariaDB
  ○ PostgreSQL
  ○ SQLite
  ○ SQL Server
```

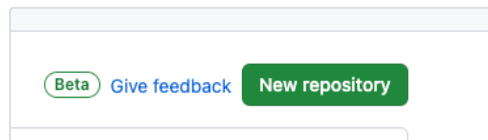
Ci chiederà se vogliamo lanciare le migrazioni di default e selezioniamo **"No"** perché dobbiamo ancora creare il nostro database.

```
Default database updated. Would you like to run the default database migrations?
○ Yes / ● No
```

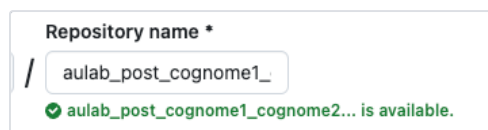
A questo punto, avremo completato l'installazione del vostro progetto Laravel. Ricordiamoci di entrare nel progetto.

```
1 cd aulab_post_cognome1_cognome2...
```

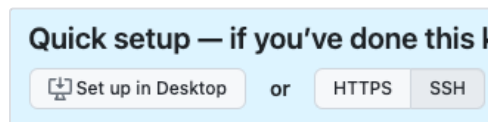
Creiamo la repository, legata al progetto, su GitHub. Andare sul sito di **GitHub**, entrare nell'organizzazione della vostra Hackademy e selezionare **"New repository"**.



In **Repository name** inserite il nome del vostro progetto.



Poi cliccare su **Create repository**. Nella schermata che ci appare, assicurarsi di aver selezionato **SSH**.



Adesso possiamo procedere con il lanciare da terminale i comandi suggeriti.

```
1 git init
2 git add .
3 git commit -m "Start project"
4 git branch -M main
5 git remote add origin .... [usa quelli indicati da GitHub]
6 git push -u origin main
```

Fatto il primo push, apriamo il progetto con Visual Studio Code

```
1 code .
```

Sempre da terminale, possiamo ora procedere creando il **database** che andremo ad utilizzare, scrivendo:

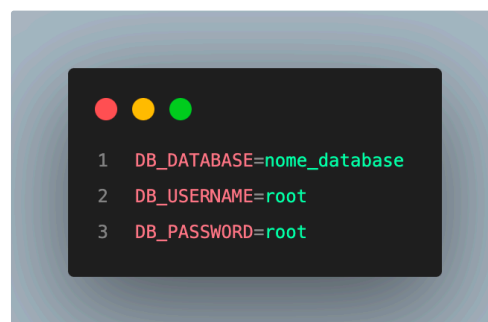
```
1 //per i windows
2 winpty mysql -u root -p
3 //per i mac
4 mysql -u root -p
```

e inseriamo come password per windows **root**, per mac **rootroot**.

Dentro il terminale di MySQL, creiamo il database con il comando

```
1 create database nome_database;
```

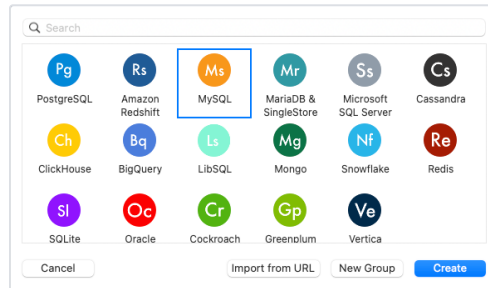
Torniamo sul progetto e nel file `.env` aggiorniamo i dati del database:



Creiamo la connessione su **TablePlus**. Aprirlo, cliccare sul +

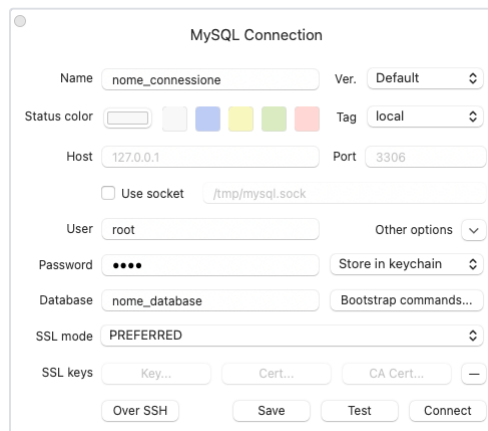


Selezionare **MySQL** e cliccare su **Create**



La visualizzazione su Windows e' ad elenco

Compilare il campo **Name** con il nome della connessione, a **User** e **Password** mettere **root** e a **database** mettere il nome che abbiamo dato al nostro database. Cliccare prima su **Test** per verificare che sia tutto ok e poi su **Connect**.



Adesso che abbiamo settato tutto, possiamo procedere con il codice. Torniamo sul progetto. Per prima cosa, ricordiamo di lanciare da terminale le **migrazioni**

```
1 php artisan migrate
```

Creiamo il **PublicController** nel terminale con il comando

```
1 php artisan make:controller PublicController
```

Andiamo poi in **web.php** e aggiorniamo la **rotta**, ricordiamoci però di **importare il controller**.



Gestiamo questa rotta nel **controller** di riferimento:



Adesso installiamo **Bootstrap**. Nel terminale lanciamo il comando:

```
1 npm install bootstrap
```

⚠ Attenzione allo spelling del pacchetto! E' facile scrivere male per la fretta (es. *bootstrap*)

Una volta terminato, andiamo in `resources/css/` e creiamo un nuovo file `style.css`.

Aggiorniamo poi il file `app.css`:

```
1 @import 'bootstrap';
2 @import './style';
```

Spostiamoci poi in `resources/js/` e creiamo un nuovo file `script.js`.

Aggiorniamo poi il file `app.js`:

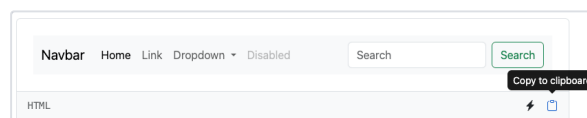
```
1 import './bootstrap';
2 import 'bootstrap';
3 import './script';
```

Adesso, occupiamoci dei **componenti**. In `resources/views/` creiamo una cartella chiamata `components`. Al suo interno creiamo due file: `layout.blade.php` e `navbar.blade.php`.

In `layout.blade.php`, nell'`head`, inserire la direttiva `@vite()` con il percorso dei file `app.css` e `app.js`, modificare il `title` con `"The Aulab Post"` e, nel `body`, richiamare il componente `navbar` e inserire un `div` che contiene la pseudo-variabile `!!slot`

```
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
7   @vite(['resources/css/app.css', 'resources/js/app.js'])
8
9   <title>The Aulab Post</title>
10 </head>
11 <body>
12   <x-navbar />
13
14   <div class="min-vh-100">
15     {!! slot !!}
16   </div>
17
18 </body>
19 </html>
```

Nel file `navbar.blade.php`, inseriamo una navbar presa da Bootstrap. Andiamo su **Bootstrap**, cerchiamo **navbar**, copiamoci il codice della prima navbar che ci compare e lo incolliamo nel file.



Andiamo adesso nella pagina `welcome.blade.php`, cancelliamo il contenuto, richiamiamo il `layout` e all'interno inseriamo una sezione per il titolo.

```
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">The Aulab Post</h1>
6       </div>
7     </div>
8   </div>
9 </x-layout>
```

Nel componente **navbar**, aggiungiamo il link per raggiungere la vista **welcome**

```
1 <li class="nav-item">
2   <a class="nav-link active" aria-current="page" href="{{route('homepage')}}">Home</a>
3 </li>
```

Creiamo anche un file `footer.blade.php` e creiamo un footer che completi la nostra pagina web.

💡 Il footer e' sempre un po' ostico da fare. Possiamo prendere spunto dal sito [MDB](#) o da altre risorse online.

Adesso controlliamo che tutto si visualizzi e funzioni correttamente. In un terminale lanciamo il comando per il **bundling degli assets**:

```
1 npm run dev
```

In un altro, invece, avviamo il server di **artisan**

```
1 php artisan serve
```

📘 Da adesso, quando cominciamo a lavorare, avremo sempre due terminali attivi: uno con il server e uno per il bundling degli assets.

Con questo, abbiamo completato lo start del nostro progetto. Possiamo pushare queste modifiche:

```
1 git add .
2 git commit -m "Scaffolding"
3 git push
```

## USER STORY 1 - AUTHENTICATION E CREAZIONE ARTICOLO [↗](#)

### AUTHENTICATION [↗](#)

Per l'autenticazione, utilizziamo la libreria **Laravel Fortify**. Per installarla lanciamo il comando sul terminale:

```
1 composer require laravel/fortify
```

Dopo aver scaricato Fortify tra le nostre dipendenze, lanciamo il comando:

```
1 php artisan fortify:install
```

per poter pubblicare tutti i file che poi utilizzeremo, tra i quali il `FortifyServiceProvider` per poter richiamare le funzioni di login e register.

Fatti questi passaggi, lanciamo la serie di comandi:

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

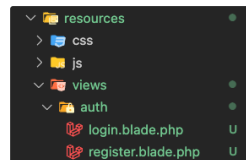
per poter migrare le tabelle che servono a Fortify per funzionare.

Fortify gestisce autonomamente il backend dell'autenticazione, ma dobbiamo provvedere noi a istruirlo su come gestire le viste di login e register, ovvero le viste dove l'utente potrà accedere o registrarsi in piattaforma. Nella cartella **app/Providers**, andare nel **FortifyServiceProvider**.

Nel metodo `boot()`, includiamo le **funzioni** che ritornano la vista login e register.



Dopo aver istruito Fortify su come restituire le viste, creiamo una nuova **cartella** in `resources/views/` che chiameremo `auth`. All'interno di questa cartella, creiamo 2 file: **login.blade.php** e **register.blade.php**



All'interno della vista login inseriremo il form di login, mentre all'interno della vista register inseriremo il form di register. Andiamo quindi a gestire prima la pagina `register.blade.php`, prestando particolarmente attenzione al form. Dalla documentazione di Fortify, possiamo leggere che nel form **register** dobbiamo inserire nel tag form la **action** con la **rotta register** e il **method POST**, il **@csrf** token e quattro input con i seguenti `name`:

- **name**
- **email**
- **password**
- **password\_confirmation**

⚠ Mi raccomando, il name dev'essere necessariamente `password_confirmation`. Ricordati di inserire la visualizzazione degli errori di validazione nei form.

Oltre a questo, possiamo anche inserire, nei vari input, l'attributo **value** con l'helper **old()** e il name dell'input. Così facendo, nel caso in cui dovessero esserci degli errori di validazione, visibili tramite la direttiva **@error** messa sotto gli input di riferimento, ci lascerebbe i campi del form già compilati. Naturalmente, questo metodo non è applicabile ai campi per la password.

```

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">Registrazione
6     </div>
7   </div>
8 </div>
9 <div class="container my-5">
10   <div class="row justify-content-center">
11     <div class="col-12 col-md-8">
12       <form action="{{route('register')}}" method="POST" class="card p-5 shadow">
13         @csrf
14         <div class="mb-3">
15           <label for="name" class="form-label">Nome utente</label>
16           <input type="text" class="form-control" id="name" name="name" value="{{old('name')}}">
17           @error('name')
18             <span class="text-danger">{{ $message }}</span>
19           @enderror
20         </div>
21         <div class="mb-3">
22           <label for="email" class="form-label">Email</label>
23           <input type="email" class="form-control" id="email" name="email" value="{{old('email')}}">
24           @error('email')
25             <span class="text-danger">{{ $message }}</span>
26           @enderror
27         </div>
28         <div class="mb-3">
29           <label for="password" class="form-label">Password</label>
30           <input type="password" class="form-control" id="password" name="password">
31           @error('password')
32             <span class="text-danger">{{ $message }}</span>
33           @enderror
34         </div>
35         <div class="mb-3">
36           <label for="password_confirmation" class="form-label">Conferma password</label>
37           <input type="password" class="form-control" id="password_confirmation" name="password_confirmation">
38         </div>
39         <div class="mb-3 d-flex justify-content-center flex-column align-items-center">
40           <button type="submit" class="btn btn-outline-secondary">Registrazione</button>
41           <p class="mt-2">Sei già registrato? <a href="{{route('login')}}" class="text-secondary">Clicca qui</a></p>
42         </div>
43       </form>
44     </div>
45   </div>
46 </div>
47 </x-layout>

```

Al submit, questo form invierà i dati presenti al suo interno all'endpoint register che in automatico prenderà questi dati e creerà un nuovo utente con cui saremo automaticamente loggati.

Nella vista login, invece, inseriamo il form di login, dove andremo ad inserire nel tag form la **action** con la **rotta login** e il **method POST**, il **@csrf** token e due input con i **name**:

- **email**
- **password**

Oltre a questo, possiamo anche inserire, nell'input della mail, l'attributo **value** con l'helper **old()** e il name dell'input e la direttiva **@error** messa sotto gli input di riferimento per mostrare eventuali errori di validazione.

```

1 <div class="container-fluid p-5 bg-secondary-subtle text-center">
2   <div class="row justify-content-center">
3     <div class="col-12">
4       <h1 class="display-3">Accedi</h1>
5     </div>
6   </div>
7 </div>
8 <div class="container my-5">
9   <div class="row justify-content-center">
10     <div class="col-12 col-md-8">
11       <form action="{{route('login')}}" method="POST" class="card p-5 shadow">
12         @csrf
13         <div class="mb-3">
14           <label form="email" class="form-label">Email</label>
15           <input type="email" class="form-control" id="email" name="email" value="{{old('email')}}">
16           @error('email')
17             <span class="text-danger">{{ $message }}</span>
18           @enderror
19         </div>
20         <div class="mb-3">
21           <label form="password" class="form-label">Password</label>
22           <input type="password" class="form-control" id="password" name="password">
23           @error('password')
24             <span class="text-danger">{{ $message }}</span>
25           @enderror
26         </div>
27         <div class="mb-3 d-flex justify-content-center flex-column align-items-center">
28           <button type="submit" class="btn btn-outline-secondary">Registrazione</button>
29           <p class="mt-2">Non sei registrato? <a href="{{route('register')}}" class="text-secondary">Clicca qui</a></p>
30         </div>
31       </form>
32     </div>
33 </div>
34 </div>
35 </div>
36 </div>

```

Avendo adesso la differenza tra utente **Guest** e **Utente Autenticato (Auth)**, abbiamo la possibilità di cambiare il nostro frontend a seconda dell'utente che sta visualizzando la pagina al momento.

Il primo passo è modificare la **navbar** inserendo una nuova sezione in base all'utente che giunge in piattaforma. Nella direttiva **@guest**, inseriremo quindi i link con le rotte **register** e **login**, mentre nella direttiva **@auth**, visualizzeremo il nome dell'utente autenticato tramite la classe **Auth::user()** e il tasto di **logout** gestito con JavaScript.

Nel tag anchor, tramite l'attributo **onclick**, stiamo dicendo a JavaScript di **prevenire il comportamento di default degli anchor** che fanno scattare solo rotte di tipo get, di **recuperare dal DOM l'elemento con id #form-logout** e di **effettuare il submit del form**. Inseriamo poi un **form** con la **action logout**, **method POST**, **id form-logout** e il **@csrf**.

```

1 <div class="navbar navbar-expand-lg bg-body-tertiary data-bs-theme="dark">
2   ...
3   @auth
4     <li class="nav-item dropdown">
5       <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown" aria-expanded="false">
6         Ciao {{ Auth::user()->name }}
7       </a>
8       <ul class="dropdown-menu">
9         <li>
10           <a class="dropdown-item" href="#" onclick="event.preventDefault(); document.querySelector('#form-logout').submit();">Logout</a>
11         </li>
12         <form action="{{route('logout')}}" method="POST" id="form-logout" class="d-none">
13           @csrf
14         </form>
15       </ul>
16     </li>
17   @endauth
18   @guest
19     <li class="nav-item dropdown">
20       <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown" aria-expanded="false">
21         Benvenuto Ospite
22       </a>
23       <ul class="dropdown-menu">
24         <li><a class="dropdown-item" href="{{route('register')}}">Registrazione</a></li>
25         <li><a class="dropdown-item" href="{{route('login')}}">Accedi</a></li>
26       </ul>
27     </li>
28   @endguest
29 </ul>
30 </div>

```

⚠ Occhio alla sintassi del contenuto dell'attributo **onclick**: ricordati di mettere il **;** dopo ogni statement.

💡 Se il logout non funziona, nel senso che vedi la pagina ricaricarsi ma senza che nulla sia successo, controlla che l'id del form e la stringa in **querySelector** combacino

Tra le direttive blade **@auth** e **@endauth**, ci sono i link che l'utente vedrà quando sarà loggato: il suo nome e il form di logout. Tra le direttive blade **@guest** e **@endguest**, invece, ci sono i link che l'utente vedrà quando non sarà loggato: i tasti che portano alla pagina di login e alla pagina di register.

Per non utilizzare entrambe le direttive, avremmo potuto anche utilizzare una delle due, intermezzata da una direttiva **@else**.

Prima di testare le varie funzioni però, andiamo nel file **fortify.php** nella cartella **config** e modifichiamo il valore della chiave **'home'** con l'uri dell'homepage per far sì che, dopo il login o la registrazione, si venga rimandati alla pagina dell'homepage.

```

1 'home' => '/',

```



Adesso, possiamo controllare che il form di register, di login e il tasto del logout funzionino perfettamente.

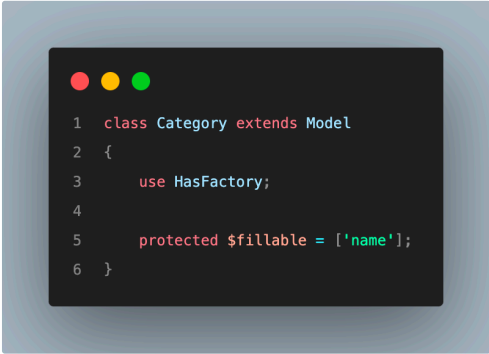
## CREAZIONE MODELLI CATEGORY E ARTICLE, RELAZIONI [↗](#)

Prima di inserire il nostro primo articolo, avremo bisogno di gestire la lista di categorie presenti sulla nostra piattaforma. Il primo passo, dunque, è quello di creare il **modello Category**:

```
1 php artisan make:model Category -m
```

così facendo stiamo creando un **Eloquent Model**, quindi un tramite tra il nostro progetto e il database che ci permette di specificare cos'è una categoria all'interno della realtà del nostro codice; aggiungendo il comando `-m`, ci viene creata anche la **migrazione** collegata.

Il primo passo però è sempre quello di inserire l'array `$fillable` all'interno del modello **Category**:



```
1 class Category extends Model
2 {
3     use HasFactory;
4
5     protected $fillable = ['name'];
6 }
```

Adesso possiamo invece gestire la sua migrazione: all'interno del file `create_categories_table`, nella **funzione up**, creiamo la **colonna** di tipo **stringa** che conterrà il nome della categoria, poi un **array di categorie** che il nostro sito avrà di default e, tramite un **foreach**, inseriamo le categorie nel database.



```
1 public function up(): void
2 {
3     Schema::create('categories', function (Blueprint $table) {
4         $table->id();
5         $table->string('name');
6         $table->timestamps();
7     });
8
9     $categories = ['politica', 'economia', 'food&drink', 'sport', 'intrattenimento', 'tech'];
10
11     foreach($categories as $category){
12         Category::create([
13             'name' => $category
14         ]);
15     }
16 }
```

Ricordiamoci di **importare il modello Category**.



```
1 use App\Models\Category;
```

Completato questo passaggio, possiamo lanciare i comandi di migrazione:

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

Controlliamo su **TablePlus** se visualizziamo correttamente la tabella categories con tutte le categorie già inserite.

Fatto questo, è tempo di passare al modello **Article**, tramite il comando:

```
1 php artisan make:model Article -mcr
```

Con il comando `-mcr` stiamo dicendo a Laravel di creare il **modello Article**, la **sua migrazione**, il **controller dedicato** e di impostare all'interno del controller le funzioni relative al **CRUD** (Create, Read, Update, Destroy) che vanno a gestire il ciclo vita dei dati.

Andiamo a gestire la **migrazione** `create_articles_table`, dove andremo a formare la tabella degli articoli. I nostri articoli avranno un titolo, un sottotitolo, un corpo ed un'immagine. Inseriamo anche le due colonne che relazionano ogni articolo con un utente ed una categoria, entrambe queste colonne sono delle **Foreign Keys**:

```

1 public function up(): void
2 {
3     Schema::create('articles', function (Blueprint $table) {
4         $table->id();
5         $table->string('title');
6         $table->string('subtitle');
7         $table->longText('body');
8         $table->string('image');
9         $table->unsignedInteger('user_id')->nullable();
10        $table->foreign('user_id')->references('id')->on('users')->onDelete('SET NULL');
11        $table->unsignedInteger('category_id')->nullable();
12        $table->foreign('category_id')->references('id')->on('categories')->onDelete('SET NULL');
13        $table->timestamps();
14    });
15 }

```

Tramite il metodo `onDelete('SET NULL')` stiamo specificando che, nel caso l'utente o la categoria venissero cancellate, quel campo assumerebbe valore `NULL`.

Possiamo adesso lanciare i comandi di migrazione:

```

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

```

Una volta creata la tabella, andiamo ad inserire l'array `$fillable` nel modello **Article**:

```

1 class Article extends Model
2 {
3     use HasFactory;
4
5     protected $fillable = [
6         'title', 'subtitle', 'body', 'image', 'user_id', 'category_id'
7     ];

```

Il passo successivo è settare le nostre relazioni nei modelli.

Abbiamo detto che un articolo è in relazione con l'utente che l'ha scritto. Un utente, quindi, può avere più articoli associati; un articolo, invece, appartiene ad un solo utente. Parliamo quindi di una relazione definita **One-to-Many** (*User 1 : N Article*). Per utilizzare le relazioni nel nostro progetto, sempre nel modello **Article** inseriamo la funzione:

```

1 class Article extends Model
2 {
3     ...
4     public function user(){
5         return $this->belongsTo(User::class);
6     }
7 }

```

Le relazioni però legano i modelli Eloquent a doppio senso, e quindi dovremo andare anche nel **modello User** e creare la funzione:

```

1 class User extends Authenticatable
2 {
3     ...
4     public function articles(){
5         return $this->hasMany(Article::class);
6     }
7 }

```

In questo modo abbiamo relazionato **Article** e **User**.

Adesso tocca relazionare anche Article e Category. Ad un articolo corrisponde una sola categoria; ad una categoria, però, corrispondono più articoli. Anche qui ci ritroviamo davanti ad una relazione **One-to-Many** (*Category 1 : N Article*).

Nel **modello Article** aggiungeremo quindi un altro metodo:

```

1 class Article extends Model
2 {
3     ...
4     public function category(){
5         return $this->belongsTo(Category::class);
6     }
7 }
8

```

Nel **modello Category**, invece, aggiungere il metodo inverso:

```

1 class Category extends Model
2 {
3     ...
4     public function articles(){
5         return $this->hasMany(Article::class);
6     }
7 }

```

## CREAZIONE DEGLI ARTICOLI [↗](#)

Occupiamoci adesso della creazione vera e propria degli articoli.

Prima di tutto, facciamo in modo che tutti i record salvati nella tabella categories siano a disposizione automaticamente di tutte le viste del nostro progetto. Andiamo quindi in **app/Providers/AppServiceProvider.php** e diciamo: se nel database c'è una tabella categories, prendi tutte le categorie e condividile in tutte le viste.

```

1 public function boot(): void
2 {
3     if(Schema::hasTable('categories')){
4         $categories = Category::all();
5         View::share(['categories' => $categories]);
6     }
7 }

```

Ricordiamoci di importare le classi, ma facciamo particolarmente attenzione a quali indichiamo:

```

1 use App\Models\Category;
2 use Illuminate\Support\Facades\Schema;
3 use Illuminate\Support\Facades\View;

```

Ora possiamo effettivamente occuparci dei nostri articoli. All'interno dell' `ArticleController` abbiamo una funzione `create()` che ha il compito di ritornare la vista che conterrà il form di creazione dell'articolo. Il primo step, quindi, è creare una rotta che richiami quella funzione. In **web.php**:

```

1 Route::get('/article/create', [ArticleController::class, 'create'])->name('article.create');

```

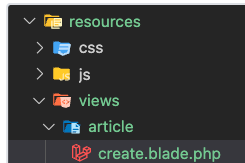
Ricordiamoci di importare l'**ArticleController**

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

Nell'**ArticleController**, nella funzione `create()`, quindi, ritorniamo la vista `article.create`:

```
1 public function create()
2 {
3     return view('article.create');
4 }
```

Creiamo quindi in `resources/views/` una nuova cartella chiamata `article` con un file al suo interno chiamato `create.blade.php`.



All'interno di questo file andremo ad inserire appunto un form con l'attributo **enctype** con valore **multipart/form-data** (per permettere al form di gestire i file, nel nostro caso le immagini), poi inseriamo:

- gli **input di type text** con i **name** per il titolo e il sottotitolo;
- un **input di type file** con il name per l'immagine;
- una **select** con il name per la categoria ed un **foreach** per ciclare le **option** per ogni categoria che abbiamo nel progetto
  - nelle option abbiamo inserito l'attributo **value** con l'**id delle categorie** che è il dato che gestiamo nel backend, mentre come contenuto abbiamo il **nome della categoria** che è il dato che visualizziamo nel frontend;
- Infine, inseriamo una **textarea** con il name per il corpo dell'articolo;

Anche in questo form, possiamo inserire, tramite la direttiva **@error**, gli eventuali errori di validazione e, con l'helper **old()**, i campi compilati se dovessero scattare degli errori appunto.

```

1 <-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">Inserisci un articolo</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 col-md-8">
12       <form class="card p-5 shadow" enctype="multipart/form-data">
13         <div class="mb-3">
14           <label for="title" class="form-label">Titolo</label>
15           <input type="text" name="title" class="form-control" id="title" value="{{old('title')}}">
16           @error('title')
17             <span class="text-danger">{{message}}</span>
18           @enderror
19         </div>
20         <div class="mb-3">
21           <label for="subtitle" class="form-label">Sottotitolo</label>
22           <input type="text" name="subtitle" class="form-control" id="subtitle" value="{{old('subtitle')}}">
23           @error('subtitle')
24             <span class="text-danger">{{message}}</span>
25           @enderror
26         </div>
27         <div class="mb-3">
28           <label for="image" class="form-label">Immagine</label>
29           <input type="file" name="image" class="form-control" id="image">
30           @error('image')
31             <span class="text-danger">{{message}}</span>
32           @enderror
33         </div>
34         <div class="mb-3">
35           <label for="category" class="form-label">Categoria</label>
36           <select name="category" id="category" class="form-control">
37             <option selected disabled>Seleziona categoria</option>
38             @foreach ($categories as $category)
39               <option value="{{ $category->id }}">{{ $category->name }}</option>
40             @endforeach
41           </select>
42           @error('category')
43             <span class="text-danger">{{message}}</span>
44           @enderror
45         </div>
46         <div class="mb-3">
47           <label for="body" class="form-label">Corpo del testo</label>
48           <textarea name="body" class="form-control" id="body" cols="30" rows="7">{{old('body')}}</textarea>
49           @error('body')
50             <span class="text-danger">{{message}}</span>
51           @enderror
52         </div>
53         <div class="mt-3 d-flex justify-content-center flex-column align-items-center">
54           <button type="submit" class="btn btn-outline-secondary">Inserisci articolo</button>
55           <a href="{{route('homepage')}}" class="text-secondary mt-2">Torna alla home</a>
56         </div>
57       </form>
58     </div>
59   </div>
60 </div>
61 </x-layout>

```

Facciamo in modo che l'utente possa arrivare in questa pagina, inserendo il link visibile solo agli utenti autenticati, nella **navbar**. Quindi, nella direttiva blade **@auth**, aggiungiamo il link con la **rotta** `article.create`.

```

1 @auth
2   <li class="nav-item">
3     <a class="nav-link" href="{{route('article.create')}}">Inserisci un articolo</a>
4   </li>
5   ...

```

Il nostro form è visibile, ma non funzionante: questo perché abbiamo bisogno di portare i dati inseriti dall'utente a una **rotta di tipo post** che non abbiamo ancora e che andremo a creare in questo momento, andando nel file **web.php**:

```

1 Route::post('/article/store', [ArticleController::class, 'store'])->name('article.store');

```

Sarà questa rotta che, al click sul tasto submit, prenderà i dati dell'utente e li salverà nel database. Andiamo quindi a gestirne la funzione `store()` nell'**ArticleController**:

```

1 public function store(Request $request)
2 {
3     $request->validate([
4         'title' => 'required|unique:articles|min:5',
5         'subtitle' => 'required|min:5',
6         'body' => 'required|min:10',
7         'image' => 'required|image',
8         'category' => 'required',
9     ]);
10
11     $article = Article::create([
12         'title' => $request->title,
13         'subtitle' => $request->subtitle,
14         'body' => $request->body,
15         'image' => $request->file('image')->store('public/images'),
16         'category_id' => $request->category,
17         'user_id' => Auth::user()->id,
18     ]);
19     return redirect(route('homepage'))->with('message', 'Articolo creato con successo');
20 }

```

In questo modo, gestiamo le **regole di validazione** per i nostri articoli e come l'articolo dev'essere **salvato nel database** prendendo i dati dalla request. Inoltre, re-indirizziamo l'utente all'homepage, dandogli anche un feedback visivo. Ricordiamoci di **importare le classi** del modello **Article** e dell'utente autenticato, **Auth**.

```

1 use App\Models\Article;
2 use Illuminate\Support\Facades\Auth;

```

Salvando le immagini nello storage, avremo bisogno di potervi accedere. Lanciamo sul terminale, dunque, il comando:

```
1 php artisan storage:link
```

Questo ci permette di creare un collegamento tra la cartella `public` (ovvero dove sono presenti tutti gli asset del nostro progetto) e la cartella `storage/app/public` dove andremo a salvare le immagini inserite dagli utenti.

Aggiorniamo quindi il **form** nella vista `article/create.blade.php`, con la **action** per la **rotta store**, **method POST** e il **@csrf** token:

```

1 <form action="{{route('article.store')}}" method="POST" class="card p-5 shadow" enctype="multipart/form-data">
2     @csrf
3     ...

```

In `welcome.blade.php`, invece, inserire il messaggio

```

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

```

Prima di provare a inserire un annuncio, però, dobbiamo essere sicuri che solo un utente loggato abbia la possibilità di inserire un annuncio. Per questo motivo, all'**ArticleController** implementeremo l'**interfaccia HasMiddleware** e inseriremo il **metodo statico middleware**. Questo metodo ritorna un array di middleware. Nel nostro caso, inseriremo il middleware `auth` a tutti i metodi nel controller, ad eccezione dei metodi `index` (visualizzazione di tutti gli articoli) e `show` (la pagina dettaglio).

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

Facciamo attenzione ad importare le classi giuste:

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

Abbiamo completato la prima User Story, possiamo pushare:

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

**Fine User Story 1**

---