The Aulab Post L11 - USER STORY #4

User Story #4:

- **Come** Lorenzo
- **vorrei** poter cercare tra gli articoli
- **in modo tale** da visualizzare subito quello che mi interessa

ACCEPTANCE CRITERIA:

- Implementazione della ricerca full-text
  - Ricerca per titolo
  - Ricerca per sottotitolo
  - Ricerca per categoria

USER STORY 4 - RICERCA FULL-TEXT

INSTALLAZIONE LARAVEL SCOUT & TNT-SEARCH

In questa User Story andremo a creare un sistema per permettere ai visitatori di poter **cercare gli articoli** all'interno della nostra piattaforma: implementeremo una ricerca definita di tipo **full-text**. In questo tipo di ricerca, un sistema indicizza un set di documenti con del testo e crea un indice. In pratica, il nostro sistema creerà un indice in cui salverà tutte le parole contenute nei titoli, nei sottotitoli e nel contenuto degli articoli e, accanto ad ogni parola, scriverà tutti gli id dei documenti in cui viene utilizzata quella parola.

Per poter creare questa funzionalità, utilizzeremo una libreria che si chiama **Laravel Scout**. Per installare Scout lanciamo il comando da terminale:

```
composer require laravel/scout
```

Poi lanciare, sempre da terminale, questo comando che renderà disponibile il file `ScoutServiceProvider.php` all'interno del nostro progetto: sarà in questo file che andremo a configurare Laravel Scout:

```
php artisan vendor:publish --provider="Laravel\Scout\ScoutServiceProvider"
```

Importiamo nel nostro progetto anche la libreria `tntsearch` che sarà il nostro **driver per la ricerca**, con il comando:

```
composer require teamtnt/laravel-scout-tntsearch-driver
```

Adesso andiamo in `config/scout.php` e nelle funzione `env()` cambiamo 'algolia' con `'tntsearch'`

Poi, restando in questo file, andiamo alla fine e inseriamo le seguenti righe di codice:

```php
'tntsearch' => [
    'storage'  => storage_path(),
    'fuzziness' => env('TNTSEARCH_FUZZINESS', true),
    'fuzzy' => [
        'prefix_length' => 2,
        'max_expansions' => 50,
        'distance' => 2,
    ],
    'asYouType' => false,
    'searchBoolean' => env('TNTSEARCH_BOOLEAN', false),
    'maxDocs' => env('TNTSEARCH_MAX_DOCS', 500),
],
```

⚠️ Inserisci questo snippet all'interno di `scout.php` senza toccare il codice originale. Fai attenzione alle chiusure dei vari array.

Cosa abbiamo scritto:

- La chiave `storage` indica dove verranno salvati i file che contengono l'indicizzazione del nostro db.
- La chiave `fuzziness` con valore true, serve per poter cercare gli articoli del nostro blog anche se commettiamo degli errori di scrittura.
- Tramite la chiave `distance`, indichiamo quanti sono gli errori che possiamo commettere.

Per evitare che gli indici del nostro database vengano pushati, è **importante andare ad aggiornare immediatamente** il file `.gitignore`:

```
...
/storage/*.index
```

⚠️ Questo passaggio e' FONDAMENTALE: dimenticarsene vuol dire affrontare merge su merge per tutto il resto del progetto.

ℹ️ Inserire `/storage/*.index` all'interno del file `.gitignore` e' anche una forma di protezione: all'interno degli indici, infatti, potrebbero esserci dei dati sensibili.

Una volta configurato `tntsearch`, dobbiamo indicare al nostro modello **Article** quali sono i suoi campi che devono essere indicizzati e che, quindi, sono **ricercabili**. Otteniamo questo risultato andando nel modello Article e inserendo al suo interno un **trait** chiamato `Searchable`, il quale ci dà accesso a una funzione che restituirà un array con la specifica di quali campi vogliamo indicizzare e quali sono i loro valori:

```php
class Article extends Model
{
    use HasFactory, Searchable;
    ...
    public function toSearchableArray(){
        return [
            'id' => $this->id,
            'title' => $this->title,
            'subtitle' => $this->subtitle,
            'body' => $this->body,
            'category' => $this->category,
        ];
    }
```

Con `$this->category` stiamo richiamando la funzione di relazione con `Category`, non il nome di una colonna nella tabella.

Importare:

```php
use Laravel\Scout\Searchable;
```

Una volta configurati questi parametri, **dobbiamo permettere a Laravel Scout di entrare nel nostro database** e di indicizzarlo, lanciando il comando da terminale:

```
php artisan scout:flush "Percorso\Del\Modello"
php artisan scout:import"Percorso\Del\Modello"
```

Nel nostro caso quindi:

```
php artisan scout:flush "App\Models\Article"
php artisan scout:import "App\Models\Article"
```

Il comando `php artisan scout:flush` rimuove tutti i record di un modello da un indice di ricerca; il comando `php artisan scout:import` invece importa tutti i record di un modello in un indice di ricerca.

Se effettui delle modifiche al modello indicizzato con scout, ricordati di lanciare in sequenza questi due comandi.

Se tutto è andato a buon fine, riceveremo un messaggio di successo con il conteggio degli elementi indicizzati come questo:

```
Imported [App\Models\Article] models up to ID: 3
All [App\Models\Article] records have been imported.
```

Per avere un'ulteriore conferma con informazioni più dettagliate, possiamo anche lanciare il comando:

```
php artisan scout:status
```

```
Analysing information from: [App\Models\Article]
1/1 [============================] 100%

 Searchable           Index            Indexed Columns            Index Records   DB Records   Records difference
 App\Models\Article   articles.index   id,title,body,category     3               3            Synchronized
```

IMPOSTARE LA RICERCA SULLA PIATTAFORMA

Adesso che Laravel Scout e TNT-Search sono correttamente installati e che il nostro modello Eloquent è indicizzato, possiamo cominciare effettivamente a gestire il funzionamento della **ricerca** sulla nostra piattaforma. Come prima cosa, andiamo in **web.php** e creiamo una rotta che gestirà i dati inseriti nella barra di ricerca:

```php
Route::get('/article/search', [ArticleController::class, 'articleSearch'])->name('article.search');
```

Andiamo nell'**ArticleController** e scriviamo la logica della funzione:

```php
public function articleSearch(Request $request){
    $query = $request->input('query');
    $articles = Article::search($query)->where('is_accepted', true)->orderBy('created_at', 'desc')->get();
    return view('article.search-index', compact('articles', 'query'));
}
```

In questo modo, stiamo dicendo al database di recuperarci tutti quegli articoli che hanno nel loro contenuto la **parola cercata dall'utente** tramite un input con name `query`, prendendo però **solo quelli effettivamente pubblicati** e ordinandoli dal più recente.

Aggiungiamo nel metodo `middleware()` la funzione per la vista della ricerca:

```php
public static function middleware()
{
    return [
        new Middleware('auth', except: ['index', 'show', 'byCategory', 'byUser', 'articleSearch']),
    ];
}
```

Il risultato della ricerca verrà quindi inviato alla vista `article/search-index.blade.php`, che dovremo creare in `resources/views/article`. In questo file, andremo sempre a ciclare tutti gli articoli mostrando le informazioni nelle card:

```blade
<x-layout>
    <div class="container-fluid p-5 bg-secondary-subtle text-center">
        <div class="row justify-content-center">
            <div class="col-12">
                <h1 class="display-1">Tutti gli articoli per {{$query}}</h1>
            </div>
        </div>
    </div>
    <div class="container my-5">
        <div class="row justify-content-evenly">
            @foreach ($articles as $article)
                <div class="col-12 col-md-3">
                    <div class="card" style="width: 18rem;">
                        <img src="{{ Storage::url($article->image) }}" class="card-img-top"
                            alt="Immagine dell'articolo: {{ $article->title }}">
                        <div class="card-body">
                            <h5 class="card-title">{{ $article->title }}</h5>
                            <p class="card-subtitle">{{ $article->subtitle }}</p>
                            <p class="small text-muted">Categoria:
                                <a href="{{route('article.byCategory', $article->category)}}" class="text-capitalize text-muted">
                                    {{ $article->category->name }}
                                </a>
                            </p>
                        </div>
                        <div class="card-footer d-flex justify-content-between align-items-center">
                            <p>Redatto il {{$article->created_at->format('d/m/Y')}} <br>
                                da <a class="text-muted" href="{{ route('article.byUser', $article->user) }}">{{$article->user->name}}</a>
                            </p>
                            <a href="{{route('article.show', $article)}}" class="btn btn-outline-secondary">Leggi</a>
                        </div>
                    </div>
                </div>
            @endforeach
        </div>
    </div>
</x-layout>
```

Ora possiamo impostare il **form** che permetterà al visitatore di effettuare una ricerca. Andiamo nel componente **navbar** e aggiungiamo la `action`, il `method GET` e il `name query` **nell'input**:

```blade
<form action="{{route('article.search')}}" method="GET" class="d-flex" role="search">
    <input class="form-control me-2" type="search" name="query" placeholder="Cerca tra gli articoli..." aria-label="Search">
    <button class="btn btn-outline-secondary" type="submit">Cerca</button>
</form>
```

Abbiamo così implementato un sistema di ricerca completo all'interno della nostra piattaforma.

Se, per qualsiasi motivo, ci trovassimo a **modificare il modello Article** o la funzione `toSearchableArray()`, ricordiamoci prima di cancellare l'indicizzazione del nostro database con questi due comandi da lanciare da terminale, **uno dopo l'altro**:

```
php artisan scout:flush "App\Models\Article"
php artisan scout:import "App\Models\Article"
```

Abbiamo completato la quarta User Story, possiamo pushare:

```
git add .
git commit -m "User Story 4 completata"
git push
```

Fine User Story 4
