Laravel Scout

Introduction

Laravel Scout provides a simple, driver based solution for adding full-text search to your Eloquent models. Using model observers, Scout will automatically keep your search indexes in sync with your Eloquent records.

Currently, Scout ships with an Algolia driver; however, writing custom drivers is simple and you are free to extend Scout with your own search implementations.

Installation

First, install Scout via the Composer package manager:

```
composer require laravel/scout
```

After installing Scout, you should publish the Scout configuration using the `vendor:publish` Artisan command. This command will publish the `scout.php` configuration file to your `config` directory:

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

Finally, add the `Laravel\Scout\Searchable` trait to the model you would like to make searchable. This trait will register a model observer to keep the model in sync with your search driver:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;
}
```

Queueing

While not strictly required to use Scout, you should strongly consider configuring a queue driver before using the library. Running a queue worker will allow Scout to queue all operations that sync your model information to your search indexes, providing much better response times for your application's web interface.

Once you have configured a queue driver, set the value of the `queue` option in your `config/scout.php` configuration file to `true`:

```
'queue' => true,
```

Driver Prerequisites

Algolia

When using the Algolia driver, you should configure your Algolia `id` and `secret` credentials in your `config/scout.php` configuration file. Once your credentials have been configured, you will also need to install the Algolia PHP SDK via the Composer package manager:

```
composer require algolia/algoliasearch-client-php:^2.2
```

Configuration

Configuring Model Indexes

Each Eloquent model is synced with a given search "index", which contains all of the searchable records for that model. In other words, you can think of each index like a MySQL table. By default, each model will be persisted to an index matching the model's typical "table" name. Typically, this is the plural form of the model name; however, you are free to customize the model's index by overriding the `searchableAs` method on the model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * Get the index name for the model.
     *
     * @return string
     */
    public function searchableAs()
    {
        return 'posts_index';
    }
}
```

Configuring Searchable Data

By default, the entire `toArray` form of a given model will be persisted to its search index. If you would like to customize the data that is synchronized to the search index, you may override the `toSearchableArray` method on the model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class Post extends Model
{
    use Searchable;

    /**
     * Get the indexable data array for the model.
     *
     * @return array
     */
    public function toSearchableArray()
    {
        $array = $this->toArray();

        // Customize array...

        return $array;
    }
}
```

Configuring The Model ID

By default, Scout will use the primary key of the model as the unique ID stored in the search index. If you need to customize this behavior, you may override the `getScoutKey` method on the model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Laravel\Scout\Searchable;

class User extends Model
{
    use Searchable;

    /**
     * Get the value used to index the model.
     *
     * @return mixed
     */
    public function getScoutKey()
    {
        return $this->email;
    }
}
```

Indexing

Batch Import

If you are installing Scout into an existing project, you may already have database records you need to import into your search driver. Scout provides an `import` Artisan command that you may use to import all of your existing records into your search indexes:

```
php artisan scout:import "App\Post"
```

The `flush` command may be used to remove all of a model's records from your search indexes:

```
php artisan scout:flush "App\Post"
```

Adding Records

Once you have added the `Laravel\Scout\Searchable` trait to a model, all you need to do is `save` a model instance and it will automatically be added to your search index. If you have configured Scout to use queues this operation will be performed in the background by your queue worker:

```php
$order = new App\Order;

// ...

$order->save();
```

Adding Via Query

If you would like to add a collection of models to your search index via an Eloquent query, you may chain the `searchable` method onto an Eloquent query. The `searchable` method will chunk the results of the query and add the records to your search index. Again, if you have configured Scout to use queues, all of the chunks will be added in the background by your queue workers:

```php
// Adding via Eloquent query...
App\Order::where('price', '>', 100)->searchable();

// You may also add records via relationships...
$user->orders()->searchable();

// You may also add records via collections...
$orders->searchable();
```

The `searchable` method can be considered an "upsert" operation. In other words, if the model record is already in your index, it will be updated. If it does not exist in the search index, it will be added to the index.

Updating Records

To update a searchable model, you only need to update the model instance's properties and `save` the model to your database. Scout will automatically persist the changes to your search index:

```
$order = App\Order::find(1);

// Update the order...

$order->save();
```

You may also use the `searchable` method on an Eloquent query to update a collection of models. If the models do not exist in your search index, they will be created:

```
// Updating via Eloquent query...
App\Order::where('price', '>', 100)->searchable();

// You may also update via relationships...
$user->orders()->searchable();

// You may also update via collections...
$orders->searchable();
```

Removing Records

To remove a record from your index, `delete` the model from the database. This form of removal is even compatible with soft deleted models:

```
$order = App\Order::find(1);

$order->delete();
```

If you do not want to retrieve the model before deleting the record, you may use the `unsearchable` method on an Eloquent query instance or collection:

```
// Removing via Eloquent query...
App\Order::where('price', '>', 100)->unsearchable();

// You may also remove via relationships...
$user->orders()->unsearchable();

// You may also remove via collections...
$orders->unsearchable();
```

Pausing Indexing

Sometimes you may need to perform a batch of Eloquent operations on a model without syncing the model data to your search index. You may do this using the `withoutSyncingToSearch` method. This method accepts a single callback which will be immediately executed. Any model operations that occur within the callback will not be synced to the model's index:

```
App\Order::withoutSyncingToSearch(function () {
    // Perform model actions...
});
```

Conditionally Searchable Model Instances

Sometimes you may need to only make a model searchable under certain conditions. For example, imagine you have `App\Post` model that may be in one of two states: "draft" and "published". You may only want to allow "published" posts to be searchable. To accomplish this, you may define a `shouldBeSearchable` method on your model:

```
public function shouldBeSearchable()
{
    return $this->isPublished();
}
```

The `shouldBeSearchable` method is only applied when manipulating models through the `save` method, queries, or relationships. Directly making models or collections searchable using the `searchable` method will override the result of the `shouldBeSearchable` method:

```
// Will respect "shouldBeSearchable"...
App\Order::where('price', '>', 100)->searchable();

$user->orders()->searchable();

$order->save();

// Will override "shouldBeSearchable"...
$orders->searchable();
```

```
$order->searchable();
```

Searching

You may begin searching a model using the `search` method. The search method accepts a single string that will be used to search your models. You should then chain the `get` method onto the search query to retrieve the Eloquent models that match the given search query:

```
$orders = App\Order::search('Star Trek')->get();
```

Since Scout searches return a collection of Eloquent models, you may even return the results directly from a route or controller and they will automatically be converted to JSON:

```
use Illuminate\Http\Request;

Route::get('/search', function (Request $request) {
    return App\Order::search($request->search)->get();
});
```

If you would like to get the raw results before they are converted to Eloquent models, you should use the `raw` method:

```
$orders = App\Order::search('Star Trek')->raw();
```

Search queries will typically be performed on the index specified by the model's `searchableAs` method. However, you may use the `within` method to specify a custom index that should be searched instead:

```
$orders = App\Order::search('Star Trek')
    ->within('tv_shows_popularity_desc')
    ->get();
```

Where Clauses

Scout allows you to add simple "where" clauses to your search queries. Currently, these clauses only support basic numeric equality checks, and are primarily useful for scoping search queries by a tenant ID. Since a search index is not a relational database, more advanced "where" clauses are not currently supported:

```
$orders = App\Order::search('Star Trek')->where('user_id', 1)->get();
```

Pagination

In addition to retrieving a collection of models, you may paginate your search results using the `paginate` method. This method will return a `Paginator` instance just as if you had paginated a traditional Eloquent query:

```
$orders = App\Order::search('Star Trek')->paginate();
```

You may specify how many models to retrieve per page by passing the amount as the first argument to the `paginate` method:

```
$orders = App\Order::search('Star Trek')->paginate(15);
```

Once you have retrieved the results, you may display the results and render the page links using Blade just as if you had paginated a traditional Eloquent query:

```
<div class="container">
    @foreach ($orders as $order)
        {{ $order->price }}
    @endforeach
</div>

{{ $orders->links() }}
```

Soft Deleting

If your indexed models are soft deleting and you need to search your soft deleted models, set the `soft_delete` option of the `config/scout.php` configuration file to `true`:

```
'soft_delete' => true,
```

When this configuration option is `true`, Scout will not remove soft deleted models from the search index. Instead, it will set a hidden `__soft_deleted` attribute on the indexed record. Then, you may use the `withTrashed` or `onlyTrashed` methods to retrieve the soft deleted records when searching:

```
// Include trashed records when retrieving results...
$orders = App\Order::search('Star Trek')->withTrashed()->get();

// Only include trashed records when retrieving results...
$orders = App\Order::search('Star Trek')->onlyTrashed()->get();
```

When a soft deleted model is permanently deleted using `forceDelete`, Scout will remove it from the search index automatically.

Customizing Engine Searches

If you need to customize the search behavior of an engine you may pass a callback as the second argument to the `search` method. For example, you could use this callback to add geo-location data to your search options before the search query is passed to Algolia:

```
use Algolia\AlgoliaSearch\SearchIndex;

App\Order::search('Star Trek', function (SearchIndex $algolia, string $query, arra
    $options['body']['query']['bool']['filter']['geo_distance'] = [
        'distance' => '1000km',
        'location' => ['lat' => 36, 'lon' => 111],
    ];

    return $algolia->search($query, $options);
})->get();
```

Custom Engines

Writing The Engine

If one of the built-in Scout search engines doesn't fit your needs, you may write your own custom engine and register it with Scout. Your engine should extend the `Laravel\Scout\Engines\Engine` abstract class. This abstract class contains eight methods your custom engine must implement:

```
use Laravel\Scout\Builder;

abstract public function update($models);
abstract public function delete($models);
abstract public function search(Builder $builder);
abstract public function paginate(Builder $builder, $perPage, $page);
abstract public function mapIds($results);
abstract public function map(Builder $builder, $results, $model);
abstract public function getTotalCount($results);
abstract public function flush($model);
```

You may find it helpful to review the implementations of these methods on the `Laravel\Scout\Engines\AlgoliaEngine` class. This class will provide you with a good starting point for learning how to implement each of these methods in your own engine.

Registering The Engine

Once you have written your custom engine, you may register it with Scout using the `extend` method of the Scout engine manager. You should call the `extend` method from the `boot` method of your `AppServiceProvider` or any other service provider used by your application. For example, if you have written a `MySqlSearchEngine`, you may register it like so:

```
use Laravel\Scout\EngineManager;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
```

```
    resolve(EngineManager::class)->extend('mysql', function () {
        return new MySqlSearchEngine;
    });
}
```

Once your engine has been registered, you may specify it as your default Scout `driver` in your `config/scout.php` configuration file:

```
'driver' => 'mysql',
```

Builder Macros

If you would like to define a custom builder method, you may use the `macro` method on the `Laravel\Scout\Builder` class. Typically, "macros" should be defined within a service provider's `boot` method:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;
use Laravel\Scout\Builder;

class ScoutMacroServiceProvider extends ServiceProvider
{
    /**
     * Register the application's scout macros.
     *
     * @return void
     */
    public function boot()
    {
        Builder::macro('count', function () {
            return $this->engine->getTotalCount(
                $this->engine()->search($this)
            );
        });
    }
}
```

The `macro` function accepts a name as its first argument, and a Closure as its second. The macro's Closure will be executed when calling the macro name from a `Laravel\Scout\Builder` implementation:

```
App\Order::search('Star Trek')->count();
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.