Pagination

Introduction

In other frameworks, pagination can be very painful. Laravel makes it a breeze. Laravel can quickly generate an intelligent "range" of links based on the current page, and the generated HTML is compatible with the Bootstrap CSS framework.

Basic Usage

Paginating Query Builder Results

There are several ways to paginate items. The simplest is by using the `paginate` method on the query builder or an Eloquent query. The `paginate` method provided by Laravel automatically takes care of setting the proper limit and offset based on the current page being viewed by the user. By default, the current page is detected by the value of the `?page` query string argument on the HTTP request. Of course, this value is automatically detected by Laravel, and is also automatically inserted into links generated by the paginator.

First, let's take a look at calling the `paginate` method on a query. In this example, the only argument passed to `paginate` is the number of items you would like displayed "per page". In this case, let's specify that we would like to display `15` items per page:

```php
<?php

namespace App\Http\Controllers;

use DB;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show all of the users for the application.
     *
     * @return Response
     */
    public function index()
    {
        $users = DB::table('users')->paginate(15);

        return view('user.index', ['users' => $users]);
    }
}
```

Note: Currently, pagination operations that use a `groupBy` statement cannot be executed efficiently by Laravel. If you need to use a `groupBy` with a paginated result set, it is recommended that you query the database and create a paginator manually.

"Simple Pagination"

If you only need to display simple "Next" and "Previous" links in your pagination view, you have the option of using the `simplePaginate` method to perform a more efficient query. This is very useful for large datasets if you do not need to display a link for each page number when rendering your view:

```php
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You may also paginate Eloquent queries. In this example, we will paginate the `User` model with `15` items per page. As you can see, the syntax is nearly identical to paginating query builder results:

```php
$users = App\User::paginate(15);
```

Of course, you may call `paginate` after setting other constraints on the query, such as `where` clauses:

```php
$users = User::where('votes', '>', 100)->paginate(15);
```

You may also use the `simplePaginate` method when paginating Eloquent models:

```php
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

Manually Creating A Paginator

Sometimes you may wish to create a pagination instance manually, passing it an array of items. You may do so by creating either an `Illuminate\Pagination\Paginator` or `Illuminate\Pagination\LengthAwarePaginator` instance, depending on your needs.

The `Paginator` class does not need to know the total number of items in the result set; however, because of this, the class does not have methods for retrieving the index of the last page. The `LengthAwarePaginator` accepts almost the same arguments as the `Paginator`; however, it does require a count of the total number of items in the result set.

In other words, the `Paginator` corresponds to the `simplePaginate` method on the query builder and Eloquent, while the `LengthAwarePaginator` corresponds to the `paginate` method.

When manually creating a paginator instance, you should manually "slice" the array of results you pass to the paginator. If you're unsure how to do this, check out the array_slice PHP function.

Displaying Results In A View

When you call the `paginate` or `simplePaginate` methods on a query builder or Eloquent query, you will receive a paginator instance. When calling the `paginate` method, you will receive an instance of `Illuminate\Pagination\LengthAwarePaginator`. When calling the `simplePaginate` method, you will receive an instance of `Illuminate\Pagination\Paginator`. These objects provide several methods that describe the result set. In addition to these helpers methods, the paginator instances are iterators and may be looped as an array.

So, once you have retrieved the results, you may display the results and render the page links using Blade:

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{!! $users->render() !!}
```

The `render` method will render the links to the rest of the pages in the result set. Each of these links will already contain the proper `?page` query string variable. Remember, the HTML generated by the `render` method is compatible with the Bootstrap CSS framework.

Note: When calling the `render` method from a Blade template, be sure to use the `{!! !!}` syntax so the HTML links are not escaped.

Customizing The Paginator URI

The `setPath` method allows you to customize the URI used by the paginator when generating links. For example, if you want the paginator to generate links like `http://example.com/custom/url?page=N`, you should pass `custom/url` to the `setPath` method:

```
Route::get('users', function () {
    $users = App\User::paginate(15);

    $users->setPath('custom/url');

    //
});
```

Appending To Pagination Links

You may add to the query string of pagination links using the `appends` method. For example, to append `&sort=votes` to each pagination link, you should make the following call to `appends`:

```
{!! $users->appends(['sort' => 'votes'])->render() !!}
```

If you wish to append a "hash fragment" to the paginator's URLs, you may use the `fragment` method. For example, to append `#foo` to the end of each pagination link, make the following call to the `fragment` method:

```
{!! $users->fragment('foo')->render() !!}
```

Additional Helper Methods

You may also access additional pagination information via the following methods on paginator instances:

- `$results->count()`
- `$results->currentPage()`
- `$results->hasMorePages()`
- `$results->lastPage() (Not available when using simplePaginate)`
- `$results->nextPageUrl()`
- `$results->perPage()`
- `$results->previousPageUrl()`
- `$results->total() (Not available when using simplePaginate)`
- `$results->url($page)`

Converting Results To JSON

The Laravel paginator result classes implement the `Illuminate\Contracts\Support\JsonableInterface` contract and expose the `toJson` method, so it's very easy to convert your pagination results to JSON.

You may also convert a paginator instance to JSON by simply returning it from a route or controller action:

```
Route::get('users', function () {
    return App\User::paginate();
});
```

The JSON from the paginator will include meta information such as `total`, `current_page`, `last_page`, and more. The actual result objects will be available via the `data` key in the JSON array. Here is an example of the JSON created by returning a paginator instance from a route:

Example Paginator JSON

```
{
   "total": 50,
   "per_page": 15,
   "current_page": 1,
   "last_page": 4,
   "next_page_url": "http://laravel.app?page=2",
   "prev_page_url": null,
   "from": 1,
   "to": 15,
   "data":[
        {
            // Result Object
        },
        {
            // Result Object
        }
   ]
}
```