Laravel

Practical 3

James Ooi

Feb 2018

Contents

1 Inserting & Updating Related Models

Recall that in Practical 01, you have simply saved the **division_id** attribute of the **Member** when storing or updating the model. This will associate that particular **Member** model with the appropriate **Division** model as the **division_id** attribute is a foreign key that references the **id** attribute on the **divisions** table.

Eloquent ORM provides us with many handy methods for saving and updating related models. We shall briefly go through the various methods available in the following sections.

1.1 The *save* Method

Assume that in a photo gallery app, we want to insert a new **Photo** for an **Album** model. While we can manually set the **album_id** attribute on the **Photo** model, we can actually insert the **Photo** directly from the relationship's **save** method:

```
$photo = new App\Photo(['description' =>
    'Chinese New Year decoration at Pavilion KL: Dream Garden of Prosperity']);

$album = App\Album::find(333);

$album->photos()->save($photo);
```

In the above example, we called the **photos** relation method to obtain an instance of the relationship. The **save** method will automatically add the appropriate **album_id** value to the new **Photo** model.

To save multiple related models, we can use the **saveMany** method:

```
$album = App\Album::find(303);

$album->photos()->saveMany([
    new App\Photo(['description' =>
        'Cuddling a koala at Cohunu Koala Park, Western Australia.']),
    new App\Photo(['description' =>
        'A koala joey at Caversham Wildlife Park, Western Australia.']),
]);
```

1.2 The *create* Method

We can also use the **create** method, which accepts an array of attributes instead of a model, creates a model, and inserts it into the database:

```
$album = App\Album::find(333);

```

```
$photo = $album->photos()->create([
    'description' =>
        'Chinese␣New␣Year␣decoration␣at␣Pavilion␣KL:␣Dream␣Garden␣of␣Prosperity',
]);
```

Or use the **createMany** method for multiple related models:

```
$album = App\Album::find(303);

$album->photos()->createMany([
    [
        'description' =>
            'Cuddling␣a␣koala␣at␣Cohunu␣Koala␣Park,␣Western␣Australia.'
    ],
    [
        'description' =>
            'A␣koala␣joey␣at␣Caversham␣Wildlife␣Park,␣Western␣Australia.'
    ],
]);
```

1.3 Belongs To Relationships

In the previous sections, the **save** and **create** methods are applicable for **hasMany** relationships. How do we save related model for **belongsTo** relationship? For this scenario, we will use the **associate** method. However, this may only be used for updating a related model, i.e. to set the foreign key on the child model:

```
$album = App\Album::find(368);
$photo = App\Photo::find(2977);

$album->photos()->associate($photo);

$album->save();
```

This effectively updates the **Photo** model's **album_id** attribute with the value **368**. To an end user, this is in effect associating the specified photo to an album.

To remove a **belongsTo** relationship, use the **dissociate** method. This method will set the relationship's foreign key to **null**:

```
$album->photos()->dissociate();

$album->save();
```

Of course in a photo gallery app, we would not be disassociating a photo from an album as photos are always associated with an album in a photo gallery app. The above example is only used to illustrate the concept.

1.4 Many To Many Relationships

Remember that in *many to many* relationships, we require the use of a *pivot* table. Now, how did you attach a **Group** to a **Member** in the previous practical task? Similarly, how did you attach a **Member** to a **Group**?

1.4.1 The *attach* and *detach* Methods

To attach a **Member** to a **Group** by inserting a record in the pivot table that joins the models, use the **attach** method:

```
$group = App\Member::find($group_id);

$group->members()->attach($member_id);
```

Note that when using the **attach** method, we attach the value of the **id** attribute of **Member** model, NOT the **Member** model itself. In effect, the **attach** method only add records to the pivot table.

To remove a many-to-many relationship record, use the **detach** method. The **detach** method only removes the appropriate record out of the pivot table:

```
// Detach a member from the group...
$group->members()->detach($member_id);

// Detach all members from the group...
$group->members()->detach();
```

When attaching, if the pivot table contains additional attributes, pass an array of additional data to be inserted into the intermediate table:

```
$student->courses()->attach($course_id, ['trimester' => '201801']);
```

1.4.2 Syncing Associations

We can also use the **sync** method to construct many-to-many associations. The **sync** method accepts an array of **id** values to place on the pivot table. Any **id** values that are not in the given array will be removed from the pivot table. So, after this operation is complete, only the **id** values in the given array will exist in the pivot table:

```
$group->members()->sync([111, 112, 113]);
```

To sync with additional attributes:

```
$student->courses()->sync([
  25 => ['trimester' => '201801'],
  69 => ['trimester' => '201705'],
  82 => ['trimester' => '201710'],
]);
```

We can also sync without detaching existing associations by using the **syncWithoutDetaching** method:

```
$group->members()->syncWithoutDetaching([18, 22, 31]);
```

1.4.3 Toggling Associations

Use the **toggle** method to toggles the attachment status of the given **id** values. If the given **id** value is currently attached, it will be detached, and vice versa:

```
$group->members()->toggle([59, 68, 86]);
```

1.4.4 Updating A Record On A Pivot Table

To update an existing row in the pivot table, use the **updateExistingPivot** method. This method accepts the pivot record foreign key and an array of attributes to update:

```
$student = App\Student::find(11);

$student->courses()->updateExistingPivot($course_id, ['trimester' => '201801']);
```

1.5 Touching Parent Timestamps

When a model **belongsTo** or **belongsToMany** another model, such as a **Photo** which belongs to an **Album**, it is sometimes helpful to update the parent's timestamp when the child model is updated.

For example, when a **Photo** model is updated, we may want to automatically *touch* the **updated_at** timestamp of the containing **Album**. Eloquent makes it easy. Just add a **touches** property containing the names of the relationships to the child model:

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Photo extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['album'];

    /**
     * Get the album that the photo belongs to.
     */
    public function album()
    {
        return $this->belongsTo('App\Album');
    }
}
```

When we update a **Photo**, the owning **Album** will have its **updated_at** column updated as well:

```
$photo = App\Photo::find(8681);

$photo->description =
    'Dynamic:␣The␣theme␣of␣Pavilion␣Bukit␣Bintang␣MRT␣station';

$photo->save();
```

2 Querying Relations

We can call the relation methods that we defined in our Eloquent model classes to obtain an instance of the relationship to perform queries. In addition, all types of Eloquent relationships also serve as query builders, allowing us to continue to chain constraints onto the relationship query before finally executing the SQL against your database.

For example, in the Travelife app that we are developing, we can query for all **Member** models that belong to a particular **Division**. In the example below, we retrieve all female members that belong to the division with the **id** having the value **16**:

```
$division = App\Division::find(16);

$members = $division->members()->where('gender', 'F')->get();
```

2.1 Relationship Methods vs Dynamic Properties

If we do not need to add additional constraints to an Eloquent relationship query, we may access the relationship as if it were a property

For example, we may access all of a division's members as follows:

```
$division = App\Division::find(16);

foreach ($division->members as $member) {
    //
}
```

However, dynamic properties are using *lazy loading* technique, meaning they will only load their relationship data when you actually access them. Because of this, developers often use *eager loading* to pre-load relationships they know will be accessed after loading the model. *Eager loading* provides a significant reduction in SQL queries that must be executed to load a model's relations.

2.2 Experimenting with Querying Relations

In this section, we will be experimenting with querying relations. Note that, for this task, we will be writing some temporary code in our route file using **Closure** to simply experiment with the various ways of querying relations.

Define a route as follows in the **routes/web.php** file:

```
Route::get('/foobar', function () {

});
```

We will be performing our experiment by writing code in the **Closure** above.

2.2.1 Get all related models

Let's begin by retrieving all related models (**Member**) of a **Division**. In the example below, we **find** a **Division** model with the **id** having the value **3**. Try it with different values and see the results:

```
$division = App\Division::find(3);

$members = $division->members()->get();
```

Add a **return** statement to return the data in **$members** as JSON. Observe the JSON output.

2.2.2 Chaining Query Builder Methods

Now, we try chaining a query builder method to add some condition to our query:

```
$division = App\Division::find(3);

$members = $division->members()->where('gender', 'F')->get();
```

Of course, we shall return the data and observe the output.

2.2.3 Querying Relationship Existence

When accessing the records for a model, we may wish to limit our results based on the existence of a relationship.

For example, imagine we want to retrieve all divisions that have at least one member. To do so, we pass the name of the relationship to the **has** and **orHas** methods:

```
// Retrieve all divisions that have at least one member...
$divisions = App\Division::has('members')->get();
```

We can also specify an operator and count to further customize the query:

```
// Retrieve all divisions that have three or more members...
$divisions = App\Division::has('members', '>=', 3)->get();
```

Nested **has** statements may also be constructed using *dot notation* to specify multiple relationships.

For example, in a blogging platform application, we can retrieve all posts that have at least one comment and vote:

```
// Retrieve all posts that have at least one comment with votes...
$posts = App\Post::has('comments.votes')->get();
```

We can use the **whereHas** and **orWhereHas** methods to put *where* conditions on the **has** queries. These methods allow us to add customized constraints to a relationship constraint:

```
// Retrieve all divisions with at least one member whose name containing words like %
$divisions = App\Division::whereHas('members', function ($query) {
    $query->where('name', 'like', '%li%');
})->get();
```

2.2.4 Querying Relationship Absence

When accessing the records for a model, we may wish to limit our results based on the absence of a relationship.

For example, if we want to retrieve all groups that don't have any members, we pass the name of the relationship to the **doesntHave** or **orDoesntHave** methods:

```
$groups = App\Group::doesntHave('members')->get();
```

2.2.5 Counting Related Models

To count the number of results from a relationship without actually loading them we may use the **withCount** method, which will place a **{relation}_count** column on the resulting models:

```
$groups = App\Group::withCount('members')->get();

foreach ($groups as $group) {
    var_dump($group->members_count);
}
```

We can also alias the relationship count result, allowing multiple counts on the same relationship:

```
$groups = App\Group::withCount([
    'members',
    'members␣as␣female_count' => function ($query) {
        $query->where('gender', 'F');
    }
])->get();

foreach ($groups as $group) {
    var_dump($group->name);
    var_dump($group->members_count);
    var_dump($group->female_count);
}
```

2.2.6 Eager Loading

When accessing Eloquent relationships as properties, the relationship data is *lazy loaded*. This means the relationship data is not actually loaded until we first access the property.

However, Eloquent can *eager load* relationships at the time you query the parent model.

Eager loading alleviates the N + 1 query problem. To illustrate:

```
$members = App\Member::all();

foreach ($members as $member) {
    var_dump($member->division->code);
}
```

In the above code, the loop will execute 1 query to retrieve all of the members on the table, then another query for each member to retrieve the division. Hence, if we have 100 members, this loop would run 101 queries.

Use eager loading to reduce this operation to just 2 queries. When querying, we specify which relationships should be eager loaded using the **with** method:

```
$members = App\Member::with('division')->get();

foreach ($members as $member) {
    var_dump($member->division->code);
}
```

For this operation, only two queries will be executed.

2.2.6.1 Eager Loading Multiple Relationships

Sometimes we need to eager load several different relationships in a single operation. To do so, just pass the relation names as an array to the **with** method:

```
$members = App\Member::with(['division', 'groups'])->get();
```

2.2.6.2 Nested Eager Loading

To eager load nested relationships, you may use *dot syntax*. For example, let's eager load all of the group members and all of the members division in one statement:

```
$groups = App\Group::with('members.division')->get();
```

2.2.6.3 Eager Loading Specific Columns

We may not always need every column from the relationships we are retrieving. For this reason, Eloquent allows us to specify which columns of the relationship we would like to retrieve:

```
$members = App\Member::with('division:id,code,name')->get();
```

IMPORTANT: Take note that the **id** column of the related model is compulsory. Failing to include it will result in a **null** value for the related model.

2.2.7 Constraining Eager Loads

Sometimes we may wish to eager load a relationship, but also specify additional query constraints for the eager loading query:

```
$members = App\Member::with(['groups' => function ($query) {
    $query->where('name', 'like', '%adventure%');
}])->get();
```

In this example, Eloquent will only eager load groups where the **name** column contains the word ***adventure***.

We can also call other query builder methods to further customize the eager loading operation:

```
$members = App\Member::with(['groups' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

3 Pagination

Laravel's paginator is integrated with the *query builder* and *Eloquent ORM* and provides convenient, easy-to-use pagination of database results out of the box. The HTML generated by the paginator is compatible with the *Bootstrap CSS framework*.

3.1 Paginating Eloquent Results

In this example, we will paginate the **Member** model with **10** items per page:

```
$members = App\Member::paginate(10);
```

Of course, we can call **paginate** after setting other constraints on the query, such as **where** clauses:

```
$members = App\Member::where('gender', 'F')->paginate(10);
```

If we only need to display simple "Next" and "Previous" links in the pagination view, we may use the **simplePaginate** method to perform a more efficient query. This is very useful for large datasets when we do not need to display a link for each page number when rendering the view:

```
$members = App\Member::simplePaginate(10);
```

3.2 Displaying Pagination Results

When calling the **paginate** method, we will receive an instance of **Illuminate\Pagination\LengthAwarePaginator**. When calling the **simplePaginate** method, we 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. Once we have retrieved the results, we may display the results and render the page links using Blade:

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

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

The **links** 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. The HTML generated by the **links** method is compatible with the *Bootstrap CSS framework*.

3.3 Converting Results to JSON

The Laravel paginator result classes implement the **Illuminate\Contracts\Support\Jsonable** interface contract and expose the **toJson** method, so it's very easy to convert the pagination results to JSON. We can also convert a paginator instance to JSON by returning it from a route or controller action:

```
Route::get('users', function () {
    return App\Member::paginate(10);
});
```

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:

```
{
    "current_page": 2,
    "data": [
        {
            "id": 3,
            "membership_no": "1001550224",
            "nric": "860620046226",
            "name": "Ng Pei Li",
            "gender": "F",
            "address": "1 Jalan Hang Jebat",
            "postcode": "75000",
            "city": "Melaka",
            "state": "04",
            "phone": "016-6067979",
            "division_id": 3,
            "created_at": "2018-02-10 08:57:30",
            "updated_at": "2018-02-10 08:57:30",
            "division": {
                "id": 3,
                "code": "DMR",
                "name": "Damansara"
            }
        },
        {
            "id": 4,
            "membership_no": "1522365623",
            "nric": "871019016211",
            "name": "Lim Li Li",
            "gender": "F",
            "address": "100 Jalan Haji Abu",
            "postcode": "84000",
            "city": "Muar",
            "state": "01",
            "phone": "012-7556232",
            "division_id": 3,
            "created_at": "2018-02-10 08:58:40",
            "updated_at": "2018-02-10 08:58:40",
            "division": {
                "id": 3,
                "code": "DMR",
                "name": "Damansara"
            }
        }
    ],
    "first_page_url": "http://localhost:8216/member?page=1",
    "from": 3,
    "last_page": 8,
    "last_page_url": "http://localhost:8216/member?page=8",
    "next_page_url": "http://localhost:8216/member?page=3",
    "path": "http://localhost:8216/member",
    "per_page": 2,
    "prev_page_url": "http://localhost:8216/member?page=1",
    "to": 4,
    "total": 15
}
```

3.4 Exercise

- Implement pagination for the listings of divisions that you have developed in the **index** method of **DivisionController**. Each page shall display up to 10 divisions. Pagination links shall include links to individual pages as well as the previous and next links.
- Implement pagination for the listings of members that you have developed in the **index** method of **MemberController**. Each page shall display up to 20 members. Pagination links shall include links to individual pages as well as the previous and next links.
- Implement pagination for the listings of groups that you have developed in the **index** method of **GroupController**. Each page shall display up to 10 groups. Pagination links shall include links to individual pages as well as the previous and next links.

4 Searching & Filtering

In our **index** methods of the three controllers that we have developed so far, we have implemented pagination to display up to a certain number of results per page, such as 10 results per page. However, end users would like to be able to perform searches and filters to the results.

Suppose that in the **index** method of **MemberController**, we would like to provide users with the following searching & filtering capabilities:

- Search by **membership_no**.
- Search by **nric**.
- Pattern search by **name**.
- Filter by **gender**.
- Filter by **division_id**.

We will need to create the required user interface and use the necessary *query builder* methods to implement our searches and filters.

4.1 The User Interface

Create a view file named **_filters.blade.php** in **resources/views/members** and type the following:

```
<?php

use App\Common;
use App\Division;

?>
<section class="filters">
    {!! Form::open([
        'route' => ['member.index'],
        'method' => 'get',
        'class' => 'form-inline'
    ]) !!}
        {!! Form::label('member-membership_no', 'Membership␣No.', [
            'class'         => 'control-label␣col-sm-3',
        ]) !!}
        {!! Form::text('membership_no', null, [
            'id'        => 'member-membership_no',
            'class'     => 'form-control',
            'maxlength' => 10,
        ]) !!}

        {!! Form::label('member-nric', 'NRIC␣No.', [
            'class'          => 'control-label␣col-sm-3',
        ]) !!}
        {!! Form::text('nric', null, [
            'id'         => 'member-nric',
            'class'      => 'form-control',
            'maxlength' => 12,
        ]) !!}

        {!! Form::label('member-name', 'Name', [
            'class'          => 'control-label␣col-sm-3',
        ]) !!}
        {!! Form::text('name', null, [
            'id'         => 'member-name',
            'class'      => 'form-control',
            'maxlength' => 100,
        ]) !!}

        {!! Form::label('member-gender', 'Gender', [
            'class'          => 'control-label␣col-sm-3',
        ]) !!}
        {!! Form::select('gender', Common::$genders, null, [
            'class'          => 'form-control',
            'placeholder'    => '-␣All␣-',
        ]) !!}

        {!! Form::label('member-division_id', 'Division', [
            'class'          => 'control-label␣col-sm-3',
        ]) !!}
        {!! Form::select('division_id',
        Division::pluck('name', 'id'),
        null, [
            'class'          => 'form-control',
            'placeholder'    => '-␣All␣-',
        ]) !!}

        {!! Form::button('Filter', [
            'type'           => 'submit',
            'class'          => 'btn␣btn-primary',
        ]) !!}

        {!! Form::close() !!}
    </div>
</section>
```

Include the above view in **resources/views/members/index.blade.php**:

```
@include('members._filters')
```

This will produce the required HTML Form to allow user to perform searches and filters as per requirement stated. Take note that the HTML Form that we built above specifies the HTTP **GET** verb. Hence, we will be accessing the user input using the **query** method of the **Request** object.

4.2 Query Builder

We have previously seen how we can use *query builder* methods such as **where** to filter results. While we can easily use the **where** method as seen before, we now face a situation if the user does not provide any search or filtering values for an attribute, we will not do any filtering on that particular attribute. For example, if a user only specify to filter by **gender**, we shall only filter our results by **gender** and ignore the filtering of other attributes.

To achieve this, we will be using *conditional clauses*. Using conditional clauses, we can apply a **where** statement if a given input value is present on the incoming request. We accomplish this using the **when** method:

```
$members = Member::with('division:id,code,name')
    ->when($request->query('gender'), function($query) use ($request) {
        return $query->where('gender', $request->query('gender'));
    })
    ->paginate(20);
```

The **when** method only executes the given **Closure** when the first parameter is **true**. If the first parameter is **false**, the **Closure** will not be executed. We can also provide another **Closure** as the third parameter which will be executed if the first parameter is **false**.

To implement the abovementioned requirements, we shall modify the **index** method of our **MemberController** as follows:

```
/**
 * Display a listing of the resource.
 *
 * @return \Illuminate\Http\Response
 */
public function index(Request $request)
{
    $members = Member::with('division:id,code,name')
        ->when($request->query('membership_no'), function($query) use ($request) {
            return $query->where('membership_no', $request->query('membership_no'));
        })
        ->when($request->query('nric'), function($query) use ($request) {
            return $query->where('nric', $request->query('nric'));
        })
        ->when($request->query('name'), function($query) use ($request) {
            return $query->where('name', 'like', '%'.$request->query('name').'%');
        })
        ->when($request->query('gender'), function($query) use ($request) {
            return $query->where('gender', $request->query('gender'));
        })
        ->when($request->query('division_id'), function($query) use ($request) {
            return $query->where('division_id', $request->query('division_id'));
        })
        ->paginate(20);

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

5 Tasks (Unguided)

- Implement functionalities for the **DivisionController** to allow user to search by **code**, **name** and **state**.
- Implement functionalities for the **GroupController** to allow user to search by **name**.
- Implement an action method named **showMembers** in the **DivisionController** to display a list of members of the specified division. You must use *querying relations* technique in your solution.
- Implement an action method named **showMembers** in the **GroupController** to display a list of members of the specified group. You must use *querying relations* technique in your solution.
- Implement an action method named **showGroups** in the **MemberController** to display a list of groups joined by the specified member. You must use *querying relations* technique in your solution.