Laravel Performance Pitfalls

## Collection vs. Query Builder

Here are a few methods that are shared between collections and the query builder:

1. first()
2. where()
3. max()/min()
4. count()

A database has one job. When it comes to data retrieval, your database will vastly outperform PHP. Prefer query builder methods over collection methods. Why? Collection methods first require data to be pulled in from your database; query builder methods run the operation within the database and return their result. Compare side-by-side:

Collection

User::all()->count()

Query Builder

User::count()

In the first example, we’re pulling in and instantiating every user in the database. That literally means (new User)->fill(…) for every single entry in your database.

In the second example, we’re asking our database (SQL, etc.) to let us know how many entries are contained. No actual data (except the result) are passed through PHP. Less for PHP to process = less RAM, less execution time.

Collection

User::all()->where(‘is\_active’, true);

Query Builder

User::where(‘is\_active’, true)->get()

This is the exact same, except now we’re looking at a more common pitfall. In the first example, we’re pulling in all users and then filtering them out in PHP. In the second example, we’re letting the database do the heavy lifting.

The key takeaway: leverage query builder wherever possible.

## Eager Loading (n+1)

Given this model:

class User extends Model {

public function photo() {

return $this->belongsTo(File::class);

}

}

And the following query:

User::all();

And the following Blade file:

@foreach ($users as $user)

{{ $user->name }}

{{ $user->photo->url }}

@endforeach

Spot the pitfall? Every time we call *->photo*, we’re introducing an additional round-trip database call. Easy solve? Eager loads.

User::with(‘photo’)->get();

Now, the photo will be batched into the original query (total = two queries). Much more efficient.

## Sub queries (n+1)

Given this model:

class User extends Model {

public function appointments() {

return $this->hasMany(Appointment::class);

}

}

And the following query:

User::all();

And the following Blade file:

@foreach ($users as $user)

{{ $user->name }}

{{

$user->appointments

->sortByDesc(‘created\_at’)

->first()

->created\_at

->diffForHumans()

}}

@endforeach

We’ve got two pitfalls happening here:

1. *->appointments* causes a fresh database round trip at each iteration
2. *->sortByDesc* is happening on a collection where we could’ve used the query builder

We can get rid of 1. by eager loading, but we’re still getting more data than we need.

public function latestAppointment() {

return $this->belongsTo(Appointment::class);

}

public function scopeWithLatestAppointment($builder) {

return $builder->addSelect([‘latest\_appointment\_id' => ( fn ($query) => $query->select('id’)

->from(‘appointments')

->whereColumn(‘users.id', 'appointments.user\_id')

->latest()

->limit(1)

) ])

->with(‘latestAppointment’);

}

User::withLatestAppointment()->get();

And update our Blade to be:

{{ $user->latest\_appointment->created\_at->diffForHumans() }}

Boom. (Credit: Jonathan Reinink - Eloquent Performance Patterns, Laracon 2019)

## Queries inside Blade

A good litmus test for performance smell is checking where you’re doing your queries. Mentally enforcing a strict separation of concerns between a presentation layer (Blade) and a database layer (controller) can help you easily refactor and isolate performance issues.

## Pagination

This is a free one. Pagination in Laravel is a one-liner.