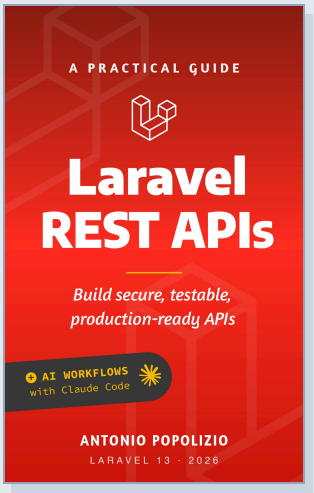
FREE CHAPTER · NOT FOR SALE

You're about to read chapter 6 of Laravel REST APIs



It's the chapter on eager loading and the N+1 problem, taken straight from the printed book.

The book builds one API from an empty Laravel install to a zero-downtime deploy on a VPS. It's called BookShelf, a deliberately boring catalog of books, authors and genres: you shouldn't have to decode a clever domain while you're trying to learn how an API is put together.

WHERE YOU'RE JUMPING IN

- `Book`, `Author` and `Genre` models: a book belongs to an author (`belongsTo`) and has many genres (`belongsToMany`);
- CRUD endpoints whose responses go through API Resources, with relationships wrapped in `whenLoaded()`;
- validation in Form Requests;
- a controller that just started calling `with()` to load relationships. Why that call matters is exactly where this chapter picks up.

You'll meet a couple of references to earlier chapters. That's what they point to. Everything else is self-contained: if you know PHP, you can follow along even if BookShelf isn't sitting on your disk.

CHAPTER 6

Eager Loading and the N+1 problem

In the previous chapter we added authors and genres to books and in the controller we used `with()` to load the relationships. In this chapter we understand *why* that `with()` is so important, what happens when you forget it and how to give the client the option to choose which relationships it wants in the response.

The N+1 problem

Imagine you have 15 books per page and you want to show the author of each. Without eager loading, the code might look like this:

```
public function index()
{
    $books = Book::paginate(15);

    return BookResource::collection($books);
}
```

And in `BookResource`:

```
'author' => new AuthorResource($this->author),
```

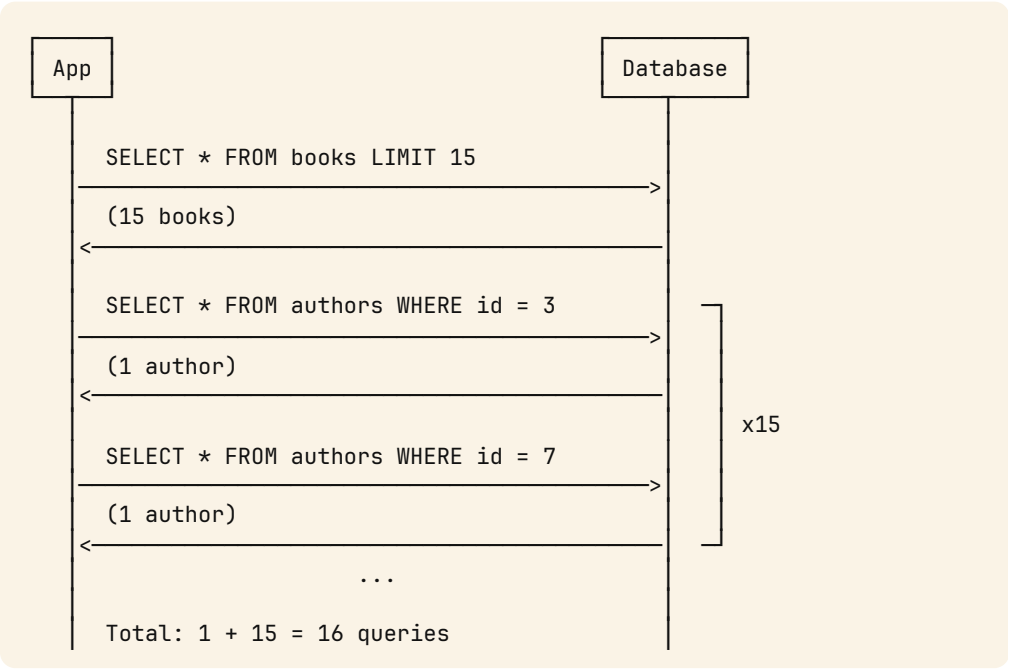
Without `whenLoaded()` in the resource and without `with()` in the controller. What happens?

Without eager loading, Eloquent runs a query for the 15 books and then one for every author: 16 queries total.

This is the N+1 problem: one query for the collection, plus N queries for each element's relationship.

In a web application with a few dozen users you might not notice. In an API serving hundreds of clients, every extra query is more response time, more load on the database, more money. It's the kind of bug you don't see during development and that explodes in your face in production.

Without eager loading (N+1):

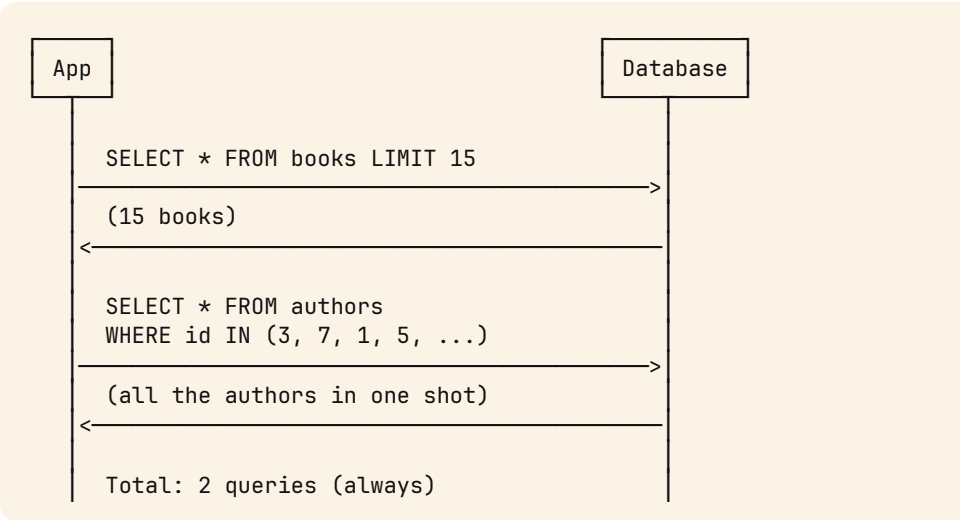


The solution is eager loading: you tell Eloquent which relationships you want *before* fetching the data and Eloquent loads them all in bulk with separate but efficient queries.

```
$books = Book::with(['author', 'genres'])->paginate(15);
```

With this line, the queries become three: one for the books, one for all the authors of the retrieved books, one for all the genres. Three queries, always. Whether the books are 15 or 500.

With eager loading:



with(), load() and $with

There are three ways to do eager loading in Laravel and each has its context.

`with()` is used on the query, before executing it:

```
// In the controller, when you build the query yourself
$books = Book::with(['author', 'genres'])->paginate(15);
```

`load()` is used on a model (or a collection) that's already been retrieved:

```
// Useful with route model binding, where the model arrives
        already loaded
public function show(Book $book)
{
    $book->load(['author', 'genres']);

    return new BookResource($book);
}
```

The `$with` property on the model automatically loads relationships every time the model is retrieved:

```
// app/Models/Book.php

protected $with = ['author'];
```

With `$with`, you don't have to remember to use `with('author')` in queries: the author is always loaded. Convenient, but dangerous. If you put too many relationships in `$with`, every single query that involves books also loads those relationships, even when not needed. For an API where the

requirements change from endpoint to endpoint, it's almost always better to be explicit with `with()` or `load()`, case by case.

> **Best practice**: avoid `$with` on the model unless the relationship is *truly always* needed. In APIs it's rare. Prefer `with()` in the controller, where you know exactly what's needed for that response.

On-demand relationship inclusion

There's a pattern widely used in APIs: let the client choose which relationships it wants in the response through a query parameter.

```
GET /api/books?include=author
GET /api/books?include=author,genres
GET /api/books       (no relationships)
```

The advantage is that the client doesn't pay the "price" of relationships when it doesn't need them. If it has to display a list of titles, it doesn't need the author. If it has to display the detail, it does.

Let's implement it. The idea is simple: we read the `include` parameter, verify the requested relationships are among the allowed ones and load them with `with()`.

We need two pieces. For queries we register a macro on the Builder in `AppServiceProvider`:

```
// app/Providers/AppServiceProvider.php

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Http\Request;

public function boot(): void
{
    Builder::macro('withIncludes', function (
        array $allowed,
        ?Request $request = null
    ): Builder {
        $request = $request ?? request();

        $requested = $request->has('include')
            ? explode(',', $request->query('include'))
            : [];

        $valid = array_intersect($requested, $allowed);

        if (filled($valid)) {
            $this->with($valid);
        }

        return $this;
    });
}
```

For models we create a trait (the Builder supports macros, the Model doesn't):

```
php artisan make:trait HasDynamicIncludes
```

The file is created at `app/Traits/HasDynamicIncludes.php`. We add the `loadIncludes()` method:

```php
// app/Traits/HasDynamicIncludes.php

namespace App\Traits;

use Illuminate\Http\Request;

trait HasDynamicIncludes
{
    public function loadIncludes(
        array $allowed,
        ?Request $request = null
    ): static {
        $request = $request ?? request();

        $requested = $request->has('include')
            ? explode(',', $request->query('include'))
            : [];

        $valid = array_intersect($requested, $allowed);

        if (filled($valid)) {
            $this->load($valid);
        }

        return $this;
    }
}
```

Add the trait to the models that need to support it. For now Book and Author:

```php
// app/Models/Book.php

use App\Traits\HasDynamicIncludes;

class Book extends Model
{
    use HasFactory;
    use HasDynamicIncludes;
    // ...
}
```

```php
// app/Models/Author.php

use App\Traits\HasDynamicIncludes;

class Author extends Model
{
    use HasFactory;
    use HasDynamicIncludes;
    // ...
}
```

`withIncludes` is used on queries (eager loading before running the query), `loadIncludes` on already-loaded models (lazy eager loading). The `$request` parameter is optional: if you don't pass it, the current request is used. The `$allowed` array is the whitelist: only the listed relationships can be loaded. If the client asks for `?include=author,secretStuff`, only `author` passes the filter.

Here's the updated `BookController`:

```
// app/Http/Controllers/BookController.php

namespace App\Http\Controllers;

use App\Models\Book;
use App\Http\Resources\BookResource;
use App\Http\Requests\StoreBookRequest;
use App\Http\Requests\UpdateBookRequest;

class BookController extends Controller
{
    public function index()
    {
        $books = Book::query()
            ->withIncludes(['author', 'genres'])
            ->paginate(15);

        return BookResource::collection($books);
    }

    public function show(Book $book)
    {
        $book->loadIncludes(['author', 'genres']);

        return new BookResource($book);
    }

    // store, update, destroy as in chapter 5
}
```

The list of allowed relationships is right in the call: the controller reads like a sentence.

> **Note**: `withIncludes` and `loadIncludes` are for **dynamic** includes, the ones the client chooses via `?include=`. In the `store` from chapter 5 we use `$book->load(['author', 'genres'])` directly, because there we **always** load the relationships to return them in the response. Two different things: fixed load vs optional include.

The AuthorController doesn't use `withIncludes` for now: for an author's books we use the nested endpoint `GET /api/authors/{author}/books`. If in the future you want to add `?include=books` to the authors endpoint, the trait is already on the model.

Let's try. First a request without relationships:

```
curl -s http://localhost:8000/api/books | jq '.data[0]'
```

```
{
    "id": 1,
    "title": "Odit libero est.",
    "isbn": "9782755224665",
    "publication_year": 1970,
    "language": "it",
    "pages": 308,
    "is_recent": false,
    "created_at": "2026-03-20T14:30:00Z",
    "updated_at": "2026-03-20T14:30:00Z"
}
```

No `author` or `genres` field, because we didn't pass `include`.

Now let's ask for the author:

```
curl -s "http://localhost:8000/api/books?include=author" \
  | jq '.data[0].author'
```

```
{
  "id": 1,
  "first_name": "Robert C.",
  "last_name": "Martin",
  "full_name": "Robert C. Martin"
}
```

Notice that `bio` doesn't appear, because we're not on the `authors.show` route.

And with both relationships:

```
curl -s "http://localhost:8000/api/books?include=author,genres" \
  | jq '.data[0]'
```

```
{
  "id": 1,
  "title": "Clean Code",
  "isbn": "9780132350884",
  "pages": 464,
  "publication_year": 2008,
  "author": {
    "id": 1,
    "first_name": "Robert C.",
    "last_name": "Martin",
    "full_name": "Robert C. Martin"
  },
  "genres": [
    {
      "id": 1,
      "name": "Programming"
    },
    {
      "id": 3,
      "name": "Software Engineering"
    }
  ]
}
```

The `whenLoaded()` we put in `BookResource` in the previous chapter now makes full sense: the field appears only if the relationship has been loaded and the relationship gets loaded only if the client asks for it.

Conditional eager loading on nested relationships

Sometimes relationships have relationships of their own. For example, when we add reviews (chapter 13), a book will have reviews and every review will have a user. The client might want to include `reviews.user`.

`with()` supports dot notation:

```
Book::with(['reviews.user'])->paginate(15);
```

This loads the reviews of the books and, for every review, the user who wrote it. Three queries total: books, reviews, users.

For on-demand include, just add `'reviews.user'` to the list of allowed relationships. The client calls:

```
GET /api/books?include=author,reviews.user
```

And gets everything efficiently. Watch out, though: the more nested relationships you allow, the larger the response payload grows and the more

the database works. Put in the list only what makes sense for your clients.

Preventing N+1 with preventLazyLoading

OK, you know what the N+1 problem is and you know how to solve it. But how do you know if you have one? It's not always obvious from the code, especially when relationships are accessed in resources rather than controllers.

Laravel has a built-in solution that turns the problem from a "silent bug" into an "explicit error". In the `AppServiceProvider` file:

```
// app/Providers/AppServiceProvider.php

use Illuminate\Database\Eloquent\Model;

public function boot(): void
{
    Model::preventLazyLoading(! app()->isProduction());
}
```

With this line, in development any attempt at lazy loading (loading a relationship that hasn't been eager loaded) throws an exception. In production it stays disabled to avoid breaking anything.

It forces you to think about relationships in advance and to use `with()` or `load()` explicitly. If you forget to load a relationship and the Resource tries to access it, instead of running a silent query you get a clear error:

```
Illuminate\Database\LazyLoadingViolationException:
Attempted to lazy load [author] on model [App\Models\Book]
but lazy loading is disabled.
```

For BookShelf, let's enable it. It costs zero in production and saves hours of debugging in development.

Tools for query debugging

`preventLazyLoading` catches the problem and stops it with an exception. If instead you want to observe queries without interrupting them (to analyze execution times or find slow queries), there are other tools.

The simplest is `DB::listen()`:

```php
// In AppServiceProvider, in Tinker or in a test
DB::listen(function ($query) {
    dump($query->sql);
    // or: Log::debug($query->sql);
});

Book::with('author')->paginate(15);
```

You'll see exactly the queries executed. Two or three is what you expect. Fifteen or more, there's a problem.

For a more complete analysis, Laravel Telescope records queries, requests, exceptions and a lot more:

```
composer require laravel/telescope --dev
php artisan telescope:install
php artisan migrate
```

Open `http://localhost:8000/telescope` and go to the Queries section. Make a call to the API and count the queries.

A lighter alternative is `beyondcode/laravel-query-detector`, which warns you in the log when it detects N+1 queries:

```
composer require beyondcode/laravel-query-detector --dev
```

You install it and forget about it until you need it.

> **In production**: Telescope shouldn't be enabled in production (it's too heavy). The query detector should be disabled. In production you monitor queries with Application Performance Monitoring (APM) tools like New Relic, Datadog or similar.

What we have now

Every endpoint that returns books now loads relationships efficiently. The client can choose which relationships it wants with the `include` parameter and the server loads only the requested ones. We have `preventLazyLoading` enabled in development to catch any N+1 before it reaches production.

The eager loading pattern is one of those things that seems like a detail but makes a huge difference in production. In an API with few users you don't notice it. In an API with thousands of requests per minute, it's the difference between a server that responds in 50ms and one that takes 2 seconds.

Checkpoint

Let's verify everything works. First, the relationships shouldn't appear if you don't ask for them:

```
curl -s http://localhost:8000/api/books \
  | jq '.data[0] | has("author")'
```

```
false
```

Now let's ask for them explicitly:

```
curl -s "http://localhost:8000/api/books?include=author" \
  | jq '.data[0] | has("author")'
```

```
true
```

Finally, let's verify that disallowed relationships are ignored without errors:

```
curl -s "http://localhost:8000/api/books?include=author,secretStuff" \
  | jq '.data[0] | keys'
```

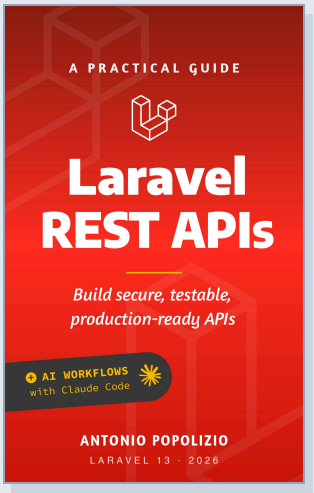
```
[
  "author",
  "created_at",
  "id",
  "is_recent",
  "isbn",
  "language",
  "pages",
  "publication_year",
  "title",
  "updated_at"
]
```

`secretStuff` isn't a relationship on the model and gets ignored. But that's not the point of the whitelist: it's there for cases where the model has real relationships you don't want to expose. If tomorrow Book has an `orders` relationship with sensitive data, the client won't be able to load it unless you add it to the whitelist.

If relationships appear only when the client asks for them and the whitelist blocks everything else, the chapter is closed. In the next one we add filters, sorting and advanced pagination.

IF YOU MADE IT HERE

That was one chapter out of twenty



The rest of the book works the same way, on the same project: REST design and error responses that tell the client what actually went wrong, validation with Form Requests, authentication with Sanctum, authorization with Policies, rate limiting, CORS and the vulnerabilities worth knowing, testing with Pest, documentation with Scribe and an Envoy deploy with zero downtime.

The last chapter is about Claude Code: how I use it on a normal working day once the fundamentals are in place. What I ask, what I re-read line by line before trusting it, when I throw the suggestion away.

> THE BOOK, ON AMAZON
> antonio.popolizio.it/laravel-rest-apis

> COMPANION CODE
> github.com/gitantonio/bookshelf

> TELL ME WHAT YOU THINK
> hello@antonio.popolizio.it, whether you liked it or not.

Il libro esiste anche in italiano: antonio.popolizio.it/api-rest-laravel