Laravel Eloquent Relationships Explained with Examples

One of the most powerful features of **Laravel 12** is its **Eloquent ORM**. With Eloquent, you don't have to write complex SQL joins to connect your data models. Instead, you define **relationships** between models, and Laravel makes querying related data simple and expressive.

In this guide, we'll cover the five major Eloquent relationships — **one-to-one**, **one-to-many**, **many-to-many**, **has-many-through**, and **polymorphic** — with full examples including migrations, model methods, queries, and Blade UI snippets. After each code block, you'll find a clear explanation so beginners won't get lost.

1 – One to One Relationship

A **one-to-one** relationship means one record in a table is related to exactly one record in another table. Example: every `User` has exactly one `Profile`.

```php
// Migration: create_profiles_table.php
Schema::create('profiles', function (Blueprint $table) {
    $table->id();
    $table->foreignId('user_id')->constrained()->onDelete('cascade');
    $table->string('bio')->nullable();
    $table->timestamps();
});
```
Code language: PHP (php)

This migration creates a `profiles` table with a `user_id` foreign key. `constrained()` sets the reference to `users.id`. `onDelete('cascade')` removes the profile when its user is deleted to keep data consistent.

```
// app/Models/User.php
public function profile()
{
    return $this->hasOne(Profile::class);
}

// app/Models/Profile.php
public function user()
{
    return $this->belongsTo(User::class);
}
```
Code language: PHP (php)

`hasOne` defines the "owner" side (User → Profile). `belongsTo` defines the inverse (Profile → User). Eloquent infers the foreign key `user_id` from the model names.

```
// Query + Blade
$user = User::with('profile')->find(1);
$bio = optional($user->profile)->bio;

// Blade
<p>Bio: {{ optional($user->profile)->bio }}</p>
```
Code language: PHP (php)

`with('profile')` eager-loads the related profile to avoid N+1 queries. `optional()` prevents errors if a user has no profile yet.

2 - One to Many Relationship

A **one-to-many** relationship is when one record can have multiple related records. Example: a `Post` has many `Comment` entries.

```
// Migration: create_comments_table.php
Schema::create('comments', function (Blueprint $table) {
```

```php
    $table->id();
    $table->foreignId('post_id')->constrained()->onDelete('cascade');
    $table->text('body');
    $table->timestamps();
});
```
Code language: PHP (php)

Each comment points to its post using `post_id`. When a post is deleted, its comments are cleaned up automatically.

```php
// app/Models/Post.php
public function comments()
{
    return $this->hasMany(Comment::class);
}

// app/Models/Comment.php
public function post()
{
    return $this->belongsTo(Post::class);
}
```
Code language: PHP (php)

`hasMany` indicates a parent with multiple children. `belongsTo` defines the child pointing back to the parent. Eloquent assumes `post_id` as the foreign key.

```php
// Query + Blade
$post = Post::with('comments')->find(1);

@foreach($post->comments as $comment)
  <p>{{ $comment->body }}</p>
@endforeach
```
Code language: PHP (php)

We eager-load comments to avoid a query per comment. The Blade loop renders all related comments efficiently.

3 – Many to Many Relationship

A **many-to-many** relationship allows multiple records on both sides to be related. Classic example: `Post` ↔ `Tag`. A post can have many tags, and a tag can belong to many posts, typically using a pivot table `post_tag`.

```
// Migration: create_tags_and_post_tag_tables.php
Schema::create('tags', function (Blueprint $table) {
    $table->id();
    $table->string('name')->unique();
    $table->timestamps();
});

Schema::create('post_tag', function (Blueprint $table) {
    $table->id();
    $table->foreignId('post_id')->constrained()->onDelete('cascade');
    $table->foreignId('tag_id')->constrained()->onDelete('cascade');
    $table->timestamps();
});Code language: PHP (php)
```

We create the `tags` table and a pivot table `post_tag` that references both `posts` and `tags`. The pivot holds the relationships.

```
// app/Models/Post.php
public function tags()
{
    return $this->belongsToMany(Tag::class);
}

// app/Models/Tag.php
public function posts()
{
    return $this->belongsToMany(Post::class);
}Code language: PHP (php)
```

`belongsToMany` tells Eloquent to use a pivot table. By convention it expects `post_tag`. If you use a different name, pass it as the second argument.

```
// Attaching / syncing tags
```

```
$post = Post::find(1);
$post->tags()->attach([1, 3]);       // add relationships
$post->tags()->sync([2, 3, 5]);      // make the set exactly [2,3,5]
$post->tags()->detach([1]);          // remove relationships

// Query + Blade
$post = Post::with('tags')->find(1);

@foreach($post->tags as $tag)
  <span class="badge bg-secondary">{{ $tag->name }}</span>
@endforeachCode language: HTML, XML (xml)
```

`attach` adds new pivot rows; `detach` removes them; `sync` replaces the entire set. Eager-loading keeps queries minimal.

4 - Has Many Through

Has-many-through lets you access a distant relationship through an intermediate model. Example: a `Country` has many `Posts` *through* `User` (Country → Users → Posts). You can fetch all posts for a country without manually joining users.

```
// Example tables
// countries: id, name
// users: id, country_id, name, ...
// posts: id, user_id, title, ...

// app/Models/Country.php
public function posts()
{
    return $this->hasManyThrough(
        Post::class,   // final model
        User::class,   // through / intermediate model
        'country_id',  // Foreign key on users table...
        'user_id',     // Foreign key on posts table...
        'id',          // Local key on countries table...
        'id'           // Local key on users table...
    );
}
```
Code language: PHP (php)

`hasManyThrough` signature is `(Final, Through, throughKey, finalKey, localKey, throughLocalKey)`. Here, a country's `id` matches users' `country_id`, and users' `id` matches posts' `user_id`.

```
// Query + Blade
$country = Country::with('posts')->find(1);

<h3>Posts from {{ $country->name }}</h3>
@foreach($country->posts as $post)
  <p>{{ $post->title }} by {{ $post->user->name }}</p>
@endforeach
```
Code language: PHP (php)

With one relationship method, you jump across the `users` table to get all posts for a country. This keeps your controllers simple.

5 – Polymorphic Relationships

Polymorphic relationships allow a model to belong to more than one type of model using a single association. Common use cases: a universal `Comment` or `Like` model that can attach to `Post`, `Video`, `Photo`, etc.

5.1 – One-to-Many Polymorphic (comments on posts & videos)

```
// Migration: create_comments_table.php
Schema::create('comments', function (Blueprint $table) {
    $table->id();
    $table->morphs('commentable'); // creates commentable_id (bigint)
& commentable_type (string)
    $table->text('body');
    $table->timestamps();
});Code language: PHP (php)
```

`morphs('commentable')` creates two columns that together define the parent model (e.g., Post or Video) and the parent id. The same table can store comments for multiple models.

```
// app/Models/Comment.php
public function commentable()
{
    return $this->morphTo();
}

// app/Models/Post.php
public function comments()
{
    return $this->morphMany(Comment::class, 'commentable');
}

// app/Models/Video.php
public function comments()
{
    return $this->morphMany(Comment::class, 'commentable');
}Code language: PHP (php)
```

`morphTo` is used on the child (`Comment`) to point to any parent. Each possible parent model (`Post`, `Video`) declares `morphMany` with the same “morph name” (`commentable`).

```
// Usage + Blade
$post = Post::with('comments')->find(1);
$video = Video::with('comments')->find(5);

@foreach($post->comments as $c)
  <p>Post comment: {{ $c->body }}</p>
@endforeach
```

```
@foreach($video->comments as $c)
  <p>Video comment: {{ $c->body }}</p>
@endforeach
```
Code language: PHP (php)

Same `comments` table, two different parents. Eloquent keeps it consistent with the `commentable_type` and `commentable_id` columns.

5.2 – Many-to-Many Polymorphic (tags for posts & videos)

```
// Migration: create_tags_and_taggables.php
Schema::create('tags', function (Blueprint $table) {
    $table->id();
    $table->string('name')->unique();
    $table->timestamps();
});

Schema::create('taggables', function (Blueprint $table) {
    $table->id();
    $table->foreignId('tag_id')->constrained()->onDelete('cascade');
    $table->morphs('taggable'); // taggable_id + taggable_type
    $table->timestamps();
});
```
Code language: PHP (php)

The `taggables` table connects a tag to any taggable model (post, video, etc.). This lets you reuse one tagging system application-wide.

```
// app/Models/Tag.php
public function posts()
{
    return $this->morphedByMany(Post::class, 'taggable');
}

public function videos()
{
    return $this->morphedByMany(Video::class, 'taggable');
}

// app/Models/Post.php
public function tags()
```

```
{
    return $this->morphToMany(Tag::class, 'taggable');
}

// app/Models/Video.php
public function tags()
{
    return $this->morphToMany(Tag::class, 'taggable');
}
```
Code language: PHP (php)

On the “thing being tagged” (Post/Video), use `morphToMany`. On the Tag side, use `morphedByMany` for each taggable type. Eloquent handles the morph columns automatically.

```
// Usage + Blade
$post = Post::with('tags')->find(1);
$video = Video::with('tags')->find(1);

@foreach($post->tags as $tag)
  <span class="badge bg-info">#{{ $tag->name }}</span>
@endforeach

@foreach($video->tags as $tag)
  <span class="badge bg-warning">#{{ $tag->name }}</span>
@endforeach
```
Code language: HTML, XML (xml)

You can attach/detach/sync tags using the relationship just like a normal many-to-many, but now polymorphic across different models.

Wrapping Up

You’ve learned the essential Eloquent relationships in Laravel 12 with practical, copy-paste examples: **one-to-one** (User–Profile), **one-to-many** (Post–Comments), **many-to-many** (Post–Tags), **has-many-through** (Country–Posts through Users), and **polymorphic** (Comments/Tags across multiple models). With these in your toolbox, you can model most real-world data cleanly and query it efficiently — while keeping your controllers and views simple.

What’s Next

- Eager Loading vs Lazy Loading in Laravel: Best Practices
- How to Use Laravel Query Scopes for Cleaner Code
- Filtering and Searching with Laravel Eloquent Query Builder