Eloquent relationships are defined as methods on your Eloquent model classes. Since relationships also serve as powerful [query builders](https://laravel.com/docs/10.x/queries), defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional query constraints on this posts relationship:

$user->posts()->where('active', 1)->get();

But, before diving too deep into using relationships, let's learn how to define each type of relationship supported by Eloquent.

[One To One](https://laravel.com/docs/10.x/eloquent-relationships#one-to-one)

A one-to-one relationship is a very basic type of database relationship. For example, a Usermodel might be associated with one Phonemodel. To define this relationship, we will place a phone method on the User model. The phone method should call the hasOnemethod and return its result. The hasOnemethod is available to your model via the model's Illuminate\Database\Eloquent\Modelbase class:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Model

{

/\*\*

\* Get the phone associated with the user.

\*/

public function phone(): HasOne

{

return $this->hasOne(Phone::class);

}

}

The first argument passed to the hasOnemethod is the name of the related model class. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship methods as if they were properties defined on the model:

$phone = User::find(1)->phone;

Eloquent determines the foreign key of the relationship based on the parent model name. In this case, the Phone model is automatically assumed to have a user\_idforeign key. If you wish to override this convention, you may pass a second argument to the hasOne method:

return $this->hasOne(Phone::class, 'foreign\_key');

Additionally, Eloquent assumes that the foreign key should have a value matching the primary key column of the parent. In other words, Eloquent will look for the value of the user's id column in the user\_id column of the Phone record. If you would like the relationship to use a primary key value other than id or your model's $primaryKey property, you may pass a third argument to the hasOnemethod:

return $this->hasOne(Phone::class, 'foreign\_key', 'local\_key');

[Defining The Inverse Of The Relationship](https://laravel.com/docs/10.x/eloquent-relationships#one-to-one-defining-the-inverse-of-the-relationship)

So, we can access the Phone model from our User model. Next, let's define a relationship on the Phone model that will let us access the user that owns the phone. We can define the inverse of a hasOne relationship using the belongsTo method:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Phone extends Model

{

/\*\*

\* Get the user that owns the phone.

\*/

public function user(): BelongsTo

{

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

}

}

When invoking the user method, Eloquent will attempt to find a User model that has an id which matches the user\_id column on the Phone model.

Eloquent determines the foreign key name by examining the name of the relationship method and suffixing the method name with \_id. So, in this case, Eloquent assumes that the Phone model has a user\_id column. However, if the foreign key on the Phonemodel is not user\_id, you may pass a custom key name as the second argument to the belongsTo method:

/\*\*

\* Get the user that owns the phone.

\*/

public function user(): BelongsTo

{

return $this->belongsTo(User::class, 'foreign\_key');

}

If the parent model does not use id as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the belongsTo method specifying the parent table's custom key:

/\*\*

\* Get the user that owns the phone.

\*/

public function user(): BelongsTo

{

return $this->belongsTo(User::class, 'foreign\_key', 'owner\_key');

}

[One To Many](https://laravel.com/docs/10.x/eloquent-relationships#one-to-many)

A one-to-many relationship is used to define relationships where a single model is the parent to one or more child models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by defining a method on your Eloquent model:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\Relations\HasMany;

class Post extends Model

{

/\*\*

\* Get the comments for the blog post.

\*/

public function comments(): HasMany

{

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

}

}

Remember, Eloquent will automatically determine the proper foreign key column for the Comment model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with \_id. So, in this example, Eloquent will assume the foreign key column on the Comment model is post\_id.

Once the relationship method has been defined, we can access the [collection](https://laravel.com/docs/10.x/eloquent-collections) of related comments by accessing the commentsproperty. Remember, since Eloquent provides "dynamic relationship properties", we can access relationship methods as if they were defined as properties on the model:

use App\Models\Post;

$comments = Post::find(1)->comments;

foreach ($comments as $comment) {

// ...

}

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the commentsmethod and continuing to chain conditions onto the query:

$comment = Post::find(1)->comments()

->where('title', 'foo')

->first();

Like the hasOne method, you may also override the foreign and local keys by passing additional arguments to the hasManymethod:

return $this->hasMany(Comment::class, 'foreign\_key');

return $this->hasMany(Comment::class, 'foreign\_key', 'local\_key');

[One To Many (Inverse) / Belongs To](https://laravel.com/docs/10.x/eloquent-relationships#one-to-many-inverse)

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a hasMany relationship, define a relationship method on the child model which calls the belongsTo method:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comment extends Model

{

/\*\*

\* Get the post that owns the comment.

\*/

public function post(): BelongsTo

{

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

}

}

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the post "dynamic relationship property":

use App\Models\Comment;

$comment = Comment::find(1);

return $comment->post->title;

In the example above, Eloquent will attempt to find a Post model that has an id which matches the post\_id column on the Commentmodel.

Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with a \_ followed by the name of the parent model's primary key column. So, in this example, Eloquent will assume the Post model's foreign key on the commentstable is post\_id.

However, if the foreign key for your relationship does not follow these conventions, you may pass a custom foreign key name as the second argument to the belongsTo method:

## 2/ **Object-oriented analysis (OOA):**

Is the process of looking at a problem, system, or task (that somebody wants to turn into a working software application) and identifying the objects and interactions between those objects. The analysis stage is all about what needs to be done.

**Object-oriented design (OOD)**

Is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify which objects can activate specific behaviors on other objects. The design stage is all about transforming what should be done into how it should be done.

**The output of the design stage** is an implementation specification. If we were to complete the design stage in a single step, we would have turned the requirements defined during object-oriented analysis into a set of classes and interfaces that could be implemented in (ideally) any object-oriented programming language.

**Object-oriented programming (OOP)**

Is the process of converting a design into a working program that does what the product owner originally requested.

**Data describes the object's state:**  
Let’s start with data. Data represent the individual characteristics of a certain object; its current state. A class can define specific sets of characteristics that are part of all objects that are members of that class. Any specific object can have different data values for the given characteristics. For example, the three oranges on our table (if we haven’t eaten any) could each weigh a different amount. The orange class could have a weight attribute to represent that datum. All instances of the orange class have a weight attribute, but each orange has a different value for this attribute. Attributes don’t have to be unique, though; any two oranges may weigh the same amount.