

Polymorphic Relationships frightened me a little bit. I mean, who wouldn't be afraid of something called "Polymorphic Relationships." Even saying those words used to make me shiver, but that's what I hope to change for you.

In this tutorial, I want to help you grasp the concept behind polymorphic relationships. It's going to be fun because I won't explain it with a boring use-case; instead, I'll explain it with pizza.

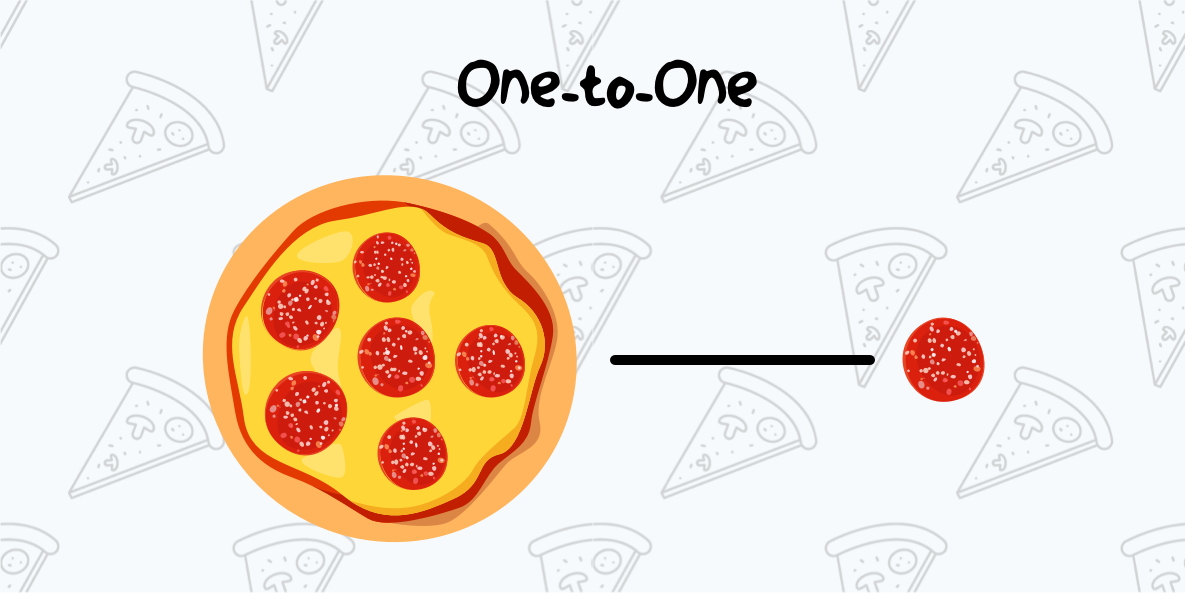
**Common Relationships**

Before explaining polymorphism, let's do a quick review of the three common relationships:

1. One-to-One
2. One-to-Many
3. Many-to-Many

**One-to-One**

This relationship ties one thing to another. **One** pizza has **one** type of topping, and inversely that **one** topping belongs to **one** pizza.



Check out the underlying table structure for this **One-to-One** relationship.

pizzas

id - integer

toppings

id - integer

name - string

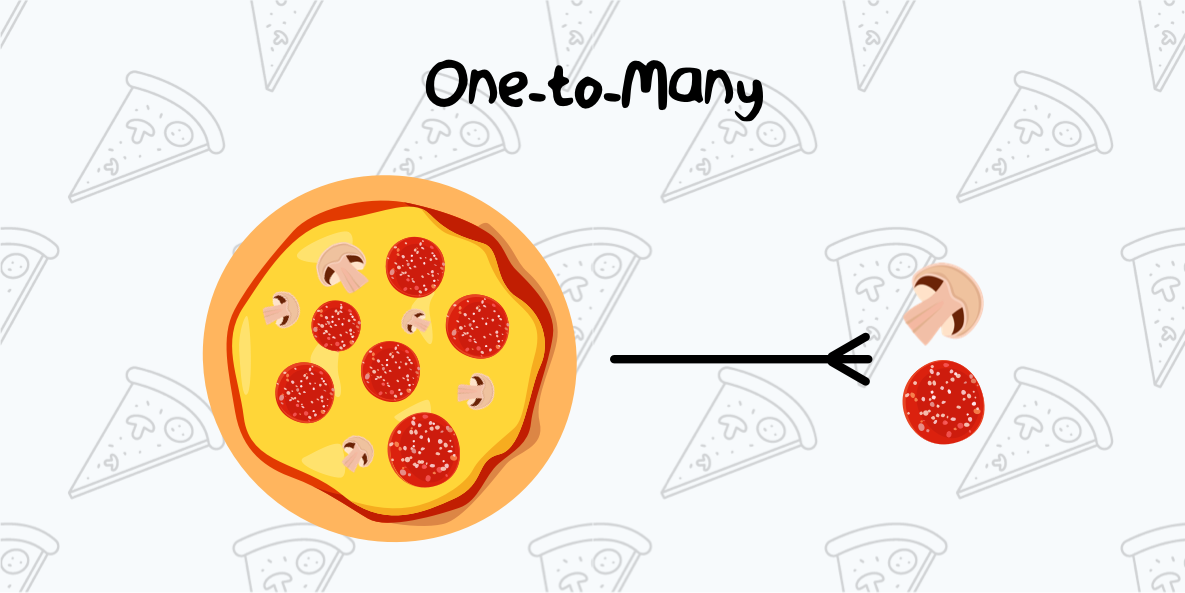
pizza\_id - integer

The toppings table has a *unique* foreign key pizza\_id that references the id from the pizzas table, creating our **One-to-One** relationship.

Of course, a pizza with only one topping seems kind'a crazy. In most cases a pizza will have many toppings. This brings us to the next relationship.

**One-to-Many**

This relationship ties one thing with many other things. From our scenario, **one** pizza can have **many** toppings, and inversely **many** toppings belong to **one** pizza.



Here is the simple database structure for this table:

pizzas

id - integer

toppings

id - integer

name - string

pizza\_id - integer

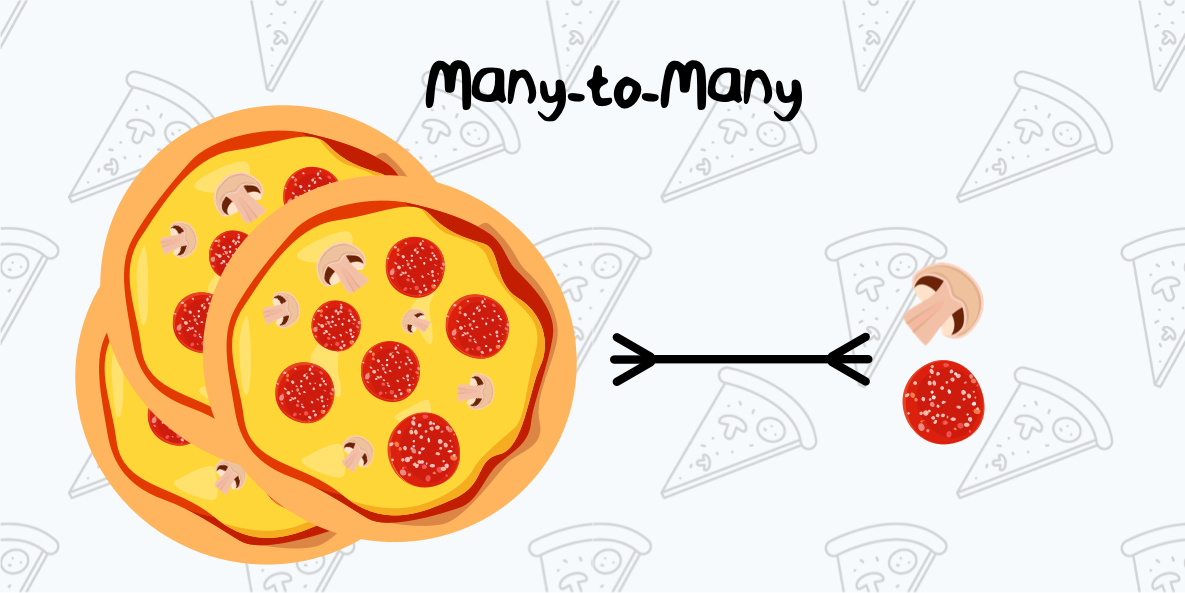
You may have thought that this is the same structure as our previous relationship, and you're right!

This relationship will have the same structure; but, there is a small difference. In a **one-to-one** relationship the pizza\_id foreign key is *unique*. In a **one-to-many** relationship, the foreign key is not *unique*, allowing the pizza to have many toppings.

Now, what if we wanted to have many pizzas with many toppings? The best way to accomplish this is with a **many-to-many** relationship.

**Many-to-Many**

With a **many-to-many** relationship, we use a lookup (or intermediate) table to define the relationship between pizzas and their toppings.



Take a look at the table structure for our **many-to-many** relationship:

pizzas

id - integer

pizza\_toppings

pizza\_id - integer

topping\_id - integer

toppings

id - integer

name - string

The pizza\_toppings lookup table has two foreign keys that reference the *pizzas* and the *toppings* tables. This lookup table creates our relationship between **many** pizzas and **many** toppings.

Those are the three most common types of relationships.

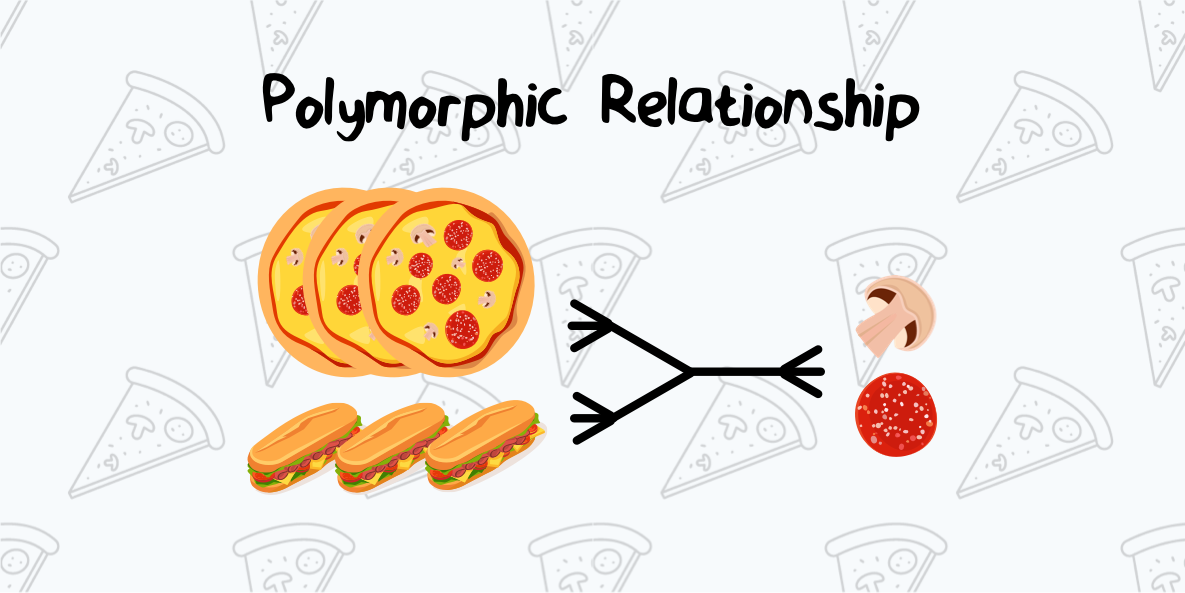
Now that we understand those, we can jump into Polymorphic Relationships.

**Hello Polymorphism**

What if we introduced a new food item into our scenario? Say, for instance, Hot Sandwiches! If we want the sandwiches to include the same toppings as a pizza, how could we do that in our database?

That's when we say, "Hello, Polymorphism!"

Polymorphic Relationships have a lookup table, similar to a many-to-many, however; it will also have an **additional column** specifying the food item.



Here is the table structure for our polymorphic relationship:

pizzas

id - integer

sandwiches

id - integer

toppings

id - integer

name - string

toppables

topping\_id - integer

toppable\_id - integer

toppable\_type - string

In this relationship, the food item (pizza or sandwich) is not directly specified; whereas, in our **many-to-many** relationship, the pizza\_toppings table directly specifies the food item (pizza).

With a Polymorphic relationship, our food can be "morphed" into different things (pizza, sandwiches, etc.). Now, any new food item we add to the menu can be toppable or have many toppings.

What's important to remember is that the tables in a polymorphic relationship can be "morphed" or dynamic.

**How Polymorphism works**

Polymorphic relationships work by using a lookup table with an additional column specifying which table the foreign key should reference.

In non-polymorphic relationships, the **foreign keys** references a **primary ID** in a specific table. On the other hand, a foreign key in a polymorphic lookup table can reference many tables.

**Polymorphism Example**

Polymorphic relationships are available in any language that utilizes a Relational Database. I will show you a few code examples of a Polymorphic Relationship using PHP and the Laravel Framework using this same table:

pizzas

id - integer

sandwiches

id - integer

toppings

id - integer

name - string

toppables

topping\_id - integer

toppable\_id - integer

toppable\_type - string

First, we will create a new Eloquent Model called Pizza which has a relationship called toppings():

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Pizza extends Model

{

/\*\*

\* Get all of the toppings for this pizza.

\*/

public function toppings()

{

return $this->morphToMany('App\Topping', 'toppable');

}

}

To get the toppings for a pizza, we can write the following code:

$pizza = App\Pizza::find(1);

foreach ($pizza->toppings as $topping) {

//

}

How cool is that!

Next, we could also retrieve the inverse of this relationship from a Topping Model:

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Topping extends Model

{

/\*\*

\* Get all of the pizzas that have a specific topping.

\*/

public function pizzas()

{

return $this->morphedByMany('App\Pizza', 'toppable');

}

/\*\*

\* Get all of the sandwiches that have a specific topping.

\*/

public function sandwiches()

{

return $this->morphedByMany('App\Sandwich', 'toppable');

}

}

And then we could say give me all the pizzas and sandwiches that have a specific topping:

$topping = App\Topping::where('name', 'onions')->first();

// Get all pizzas that have onions as a topping

foreach ($topping->pizzas as $pizza) {

//

}

// Get all sandwiches that have onions as a topping

foreach ($topping->sandwiches as $sandwich) {

//

}

Adding polymorphic relationships can make your app more efficient, flexible, and easier to program.

**Digging Deeper**

Polymorphic relationships can also be categorized into the three common relationships we covered earlier in the tutorial. A polymorphic relationship can also be **one-to-one**, **one-to-many**, and **many-to-many**.

The example relationship that we covered in this tutorial was a **Polymorphic Many-to-Many Relationship**. That's a mouthful, but as you can tell, it's not that difficult to comprehend.

If you would like to learn more about each of these Polymorphic relationships, be sure to head over to the Laravel documentation to learn about each one:

1. [Polymorphic One-to-One Relationships](https://laravel.com/docs/eloquent-relationships#one-to-one-polymorphic-relations)
2. [Polymorphic One-to-Many Relationships](https://laravel.com/docs/eloquent-relationships#one-to-many-polymorphic-relations)
3. [Polymorphic Many-to-Many Relationships](https://laravel.com/docs/eloquent-relationships#many-to-many-polymorphic-relations)

**Conclusion**

I hope this tutorial has helped you learn a little more about Polymorphic Relationships.

The best way to learn any concept is to just dive in and start creating. After implementing polymorphism into a few projects you will start to understand more and more. Eventually, the concept of utilizing "Polymorphic Relationships" will seem like a pizza cake ;)

Post By