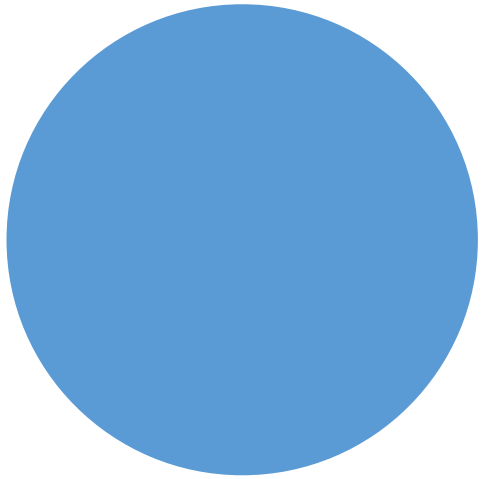
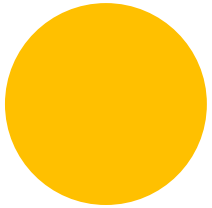



Model Relationships

Recap

- Models are responsible for managing data
- In Laravel we mainly define our models in database migration files
- We also have a PHP class representing each model, that class implements the Active Record design pattern and is responsible for – interacting with the table as a whole, representing an individual row and managing its own persistence
- Using the Eloquent library we can build database queries without directly writing SQL statements
- To test our application we can automatically seed data into our database and make it realistic using the faker library

Building Relationships between Models

- The next step is to start building relationships between our Models
- You might want to read over your database notes!
- It will get a little complex but essentially:
  - First decide what type of relationship you need
  - Define the database key relationships in the migration files
  - And create methods in our model classes to automatically perform the relationship based queries

Types of Relationships

- One to One: Example – student id to exam results
- One to Many: Example – tutors to student ids
- Many to Many: Example – student ids to enrolled modules

One to One Relationships

Animals and Emergency Contacts

First we will need to create an EmergencyContact model…

php artisan make:model EmergencyContact -m

One to One

- The goal of the relationship is that each animal has exactly one emergency contact, and each emergency contact is only used for one animal.
- We need to add the animal id to the emergency contact as a foreign key
- We need to create methods in the animal and emergency contact models to return relationships

One to One: Migration

First we create/amend the migration of the Emergency Contacts table:

```
public function up()
{
    Schema::create('emergency_contacts', function (Blueprint $table)
        {
        $table->increments('id');
        $table->string('name');
        $table->string('phone_number');
        $table->unsignedInteger('animal_id');
        $table->timestamps();

        $table->foreign('animal_id')->references('id')->
            on('animals')->onDelete('cascade')-> onUpdate('cascade');
    });
}
```

This will hold the id of the animal for the emergency contact.

By default Laravel expects this to be named 'animal_id'. You can change it, but you have to specify more 'links' later on.

This last (double) line sets up the referential integrity constraint for the database. Many online tutorials will leave this out. Adding this in might catch and prevent errors later on.

Referential Integrity Constraints

```
$table->foreign('enclosure_id')->references('id')->
    on('enclosures');
```

- Adding this line sets up a constraint which will maintain the correctness of data in our database
- We could also set up cascades and other conditions to perform on data deletion…

Referential Integrity Constraints - Improved

You can add more to the constraint line:

```
$table->foreign('animal_id')->references('id')->
    on('animals')->onDelete('cascade')-> onUpdate('cascade');
```

This will setup the constraint as before.

But now if you delete an enclosure, all animals will be deleted too.

Also if you change the id of an enclosure then the animal's enclosure ids will be updated too.

Instead of cascade you can also use the word restrict to prevent the operation if it causes the violations of the constraint. But this is the default behaviour anyway.

One to One: Updating the Animal Model

```php
class Animal extends Model
{
    public function emergencyContact()
    {
        return $this->hasOne('App\EmergencyContact');
    }
}
```

- The next step is to define a function which will allow us to use the Animal class to query it’s emergency contact
- This relies on you following Laravel’s conventions with naming foreign keys!

One to One: Emergency Contact Model

```php
class EmergencyContact extends Model
{
    public function animal()
    {
        return $this->belongsTo('App\Animal');
    }
}
```

- We also want to setup the inverse relationship
- Again we are defining a function for us to access the Animal related to a particular Emergency Contact

Some gotchas before you migrate!

- Your foreign key must match the type of the primary key it refers to – depending on your Laravel version that might be bigIncrements
- You will need to think about the order of database migrations. If one table needs to set up a foreign key with another then that other table needs to exist already

Seeding One to One relationships

- When seeding emergency contacts we must provide an animal id
- In this case it will just be the first animal

```php
class EmergencyContactsTableSeeder extends Seeder
{
    public function run()
    {
        $e = new EmergencyContact;
        $e->name = 'Max';
        $e->phone_number = '201-886-0269';
        $e->animal_id = 1; // Leo
        $e->save();
    }
}
```

Order of Seeding

- The order of seeding also matters
- In this case we need the animals to exist before we can set the relationship in the emergency contact

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UsersTableSeeder::class);
        $this->call(AnimalsTableSeeder::class);
        $this->call(EmergencyContactsTableSeeder::class);
    }
}
```

One to One: Testing in Tinker

You should always test in tinker!

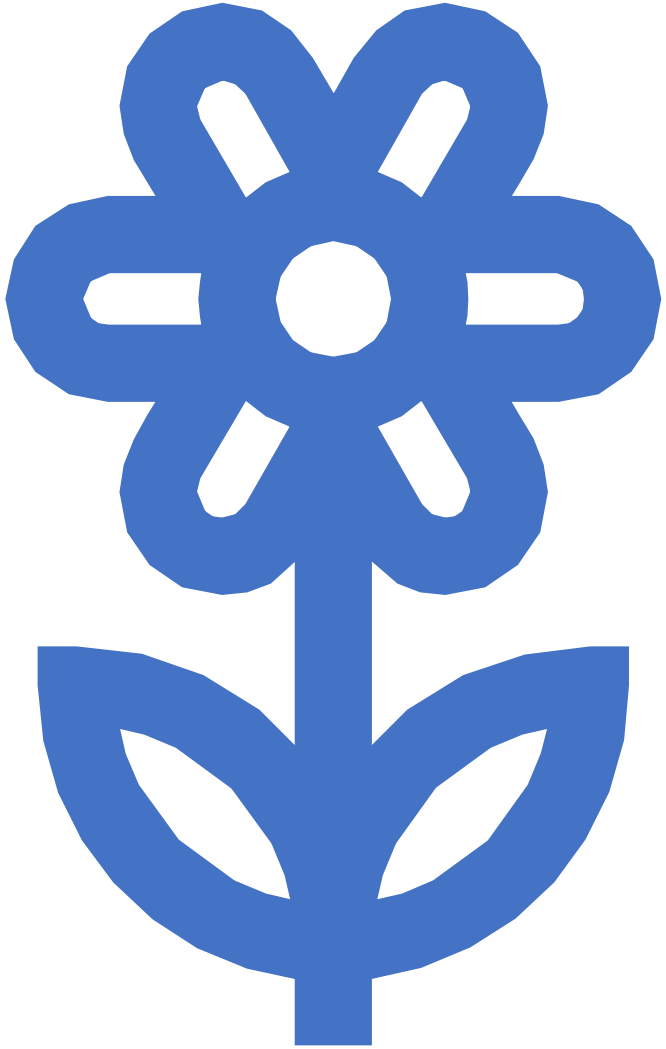
```
vagrant@homestead:~/Laravel/swanseazoo$ artisan tinker
Psy Shell v0.9.8 (PHP 7.2.9-1+ubuntu18.04.1+deb.sury.org+1 — cli) by Justin Hileman
>>> App\Animal::find(1)->emergencyContact
=> App\EmergencyContact {#2912
     id: 1,
     name: "Max",
     phone_number: "201-886-0269",
     animal_id: 1,
     created_at: "2018-10-16 19:48:35",
     updated_at: "2018-10-16 19:48:35",
   }
>>> App\Animal::find(1)->emergencyContact->animal
=> App\Animal {#2914
     id: 1,
     name: "Leo",
     weight: 351.6,
     date_of_birth: null,
     created_at: "2018-10-16 19:48:35",
     updated_at: "2018-10-16 19:48:35",
   }
>>>
```

Getting the emergency contact from an animal

If we then get the animal of the emergency contact we should end up back where we started – with the original animal.

One to Many

Enclosures to Animals



One to Many

- Our goal is to have enclosures which contain many animals
- …and each animal belongs to a single enclosure
- So we need to add an enclosure id to the animal model
- Then create methods in the animal and enclosure models to return those relationships

One to Many: Migration

First we create/amend the migration of the animals table:

```php
public function up()
{
    Schema::create('animals', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->double('weight', 8, 2);
        $table->dateTime('date_of_birth')->nullable();
        $table->unsignedInteger('enclosure_id');
        $table->timestamps();

        $table->foreign('enclosure_id')->references('id')->
            on('enclosures')->onDelete('cascade')->
            onUpdate('cascade');
    });
}
```

Similar to One-To-One. This will hold the id of the enclosure for animal.

By default Laravel expects this to be named 'enclosure_id'. You can change it, but you have to specify more 'links' later on.

This last (double) line sets up the referential integrity constraint for the database. Many online tutorials will leave this out. Adding this in might catch and prevent errors later on.

One to Many: Migration (Cont.)

We had better create the enclosures table too:

```php
public function up()
{
    Schema::create('enclosures', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->timestamps();
    });
}
```

One to Many: Enclosure Model

- Now we need to add the method to the enclosure model which gets its animals
- Notice the function name is plural – because we will be getting Many animals

```php
class Enclosure extends Model
{
    /**
     * Get the animals that live in the enclosure.
     */
    public function animals()
    {
        return $this->hasMany('App\Animal');
    }
}
```

One to Many: Animal Model

We should also setup the inverse relationship in Animal:

```php
class Animal extends Model
{
    public function emergencyContact()
    {
        return $this->hasOne('App\EmergencyContact');
    }

    /**
     * Get the enclosure that this animal lives in.
     */
    public function enclosure()
    {
        return $this->belongsTo('App\Enclosure');
    }
}
```

Now we can get the enclosure of a particular animal.

One to Many: Enclosures & Animal Seeders

Again, seeding is a little tricky

```php
class EnclosuresTableSeeder extends Seeder
{
    public function run()
    {
        $e = new Enclosure;
        $e->name = 'Central Enclosure';
        $e->save();
    }
}
```

```php
class AnimalsTableSeeder extends Seeder
{
    public function run()
    {
        $a = new Animal;
        $a->name = "Leo";
        $a->weight = 351.6;
        $a->enclosure_id = 1; // Central Enclosure
        $a->save();

        factory(App\Animal::class, 50)->create();
    }
}
```

Seeding Order

The enclosures seeder should be run before the animals seeder

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        // $this->call(UsersTableSeeder::class);
        $this->call(EnclosuresTableSeeder::class);
        $this->call(AnimalsTableSeeder::class);
        $this->call(EmergencyContactsTableSeeder::class);
    }
}
```

One to Many: Testing in Tinker

You should always test in tinker!

```
vagrant@homestead:~/Laravel/swanseazoo$ artisan tinker
Psy Shell v0.9.8 (PHP 7.2.9-1+ubuntu18.04.1+deb.sury.org+1 — cli) by Justin Hileman
>>> App\Animal::find(1)->enclosure
=> App\Enclosure {#2910
     id: 1,
     name: "Central Enclosure",
     created_at: "2018-10-16 20:27:10",
     updated_at: "2018-10-16 20:27:10",
   }
>>> App\Animal::find(1)->enclosure->animals
=> Illuminate\Database\Eloquent\Collection {#2962
     all: [
       App\Animal {#2963
         id: 1,
         name: "Leo",
         weight: 351.6,
         date_of_birth: null,
         enclosure_id: 1,
         created_at: "2018-10-16 20:27:10",
         updated_at: "2018-10-16 20:27:10",
       },
       App\Animal {#2964
         id: 2,
         name: "Dixie",
         weight: 312.23,
         date_of_birth: null,
         enclosure_id: 1,
         created_at: "2018-10-16 20:27:11",
         updated_at: "2018-10-16 20:27:11",
       },
       App\Animal {#2965
         id: 3,
```

Getting the enclosure of an animal

If we then get the animals of that enclosure we get a list of animals. Our original animal should be part of the list.

Using Eloquent rather than ids

- You don’t need to work with the model ids to deal with relationships
- Find out more at https://laravel.com/docs/eloquent-relationships#inserting-and-updating-related-models

```
>>> $e = new App\Enclosure
=> App\Enclosure {#3015}
>>> $e->name = "Eastern Enclosure"
=> "Eastern Enclosure"
>>> $e->save()
=> true
>>>
>>>
>>> $a = App\Animal::find(42)
=> App\Animal {#3007
     id: 42,
     name: "Adan",
     weight: 361.61,
     date_of_birth: null,
     enclosure_id: 1,
     created_at: "2018-10-16 21:18:07",
     updated_at: "2018-10-16 21:18:07",
   }
>>>
>>> $e->animals
=> Illuminate\Database\Eloquent\Collection {#2918
     all: [],
   }
>>> $e->animals()->save($a)
=> App\Animal {#3007
     id: 42,
     name: "Adan",
     weight: 361.61,
     date_of_birth: null,
     enclosure_id: 2,
     created_at: "2018-10-16 21:18:07",
     updated_at: "2018-10-16 21:32:08",
   }
```

Eager vs Lazy Loading

- When accessing Eloquent relationships as properties the relationship data is not actually loaded until you first access the property – this is lazy loading
- Eloquent can “eager load” relationships at the time you query the parent model

Lazy Loading

```php
$books = App\Book::all();


foreach ($books as $book) {
    echo $book->author->name;
}
```

Eager Loading

```php
$books = App\Book::with('author')->get();


foreach ($books as $book) {
    echo $book->author->name;
}
```

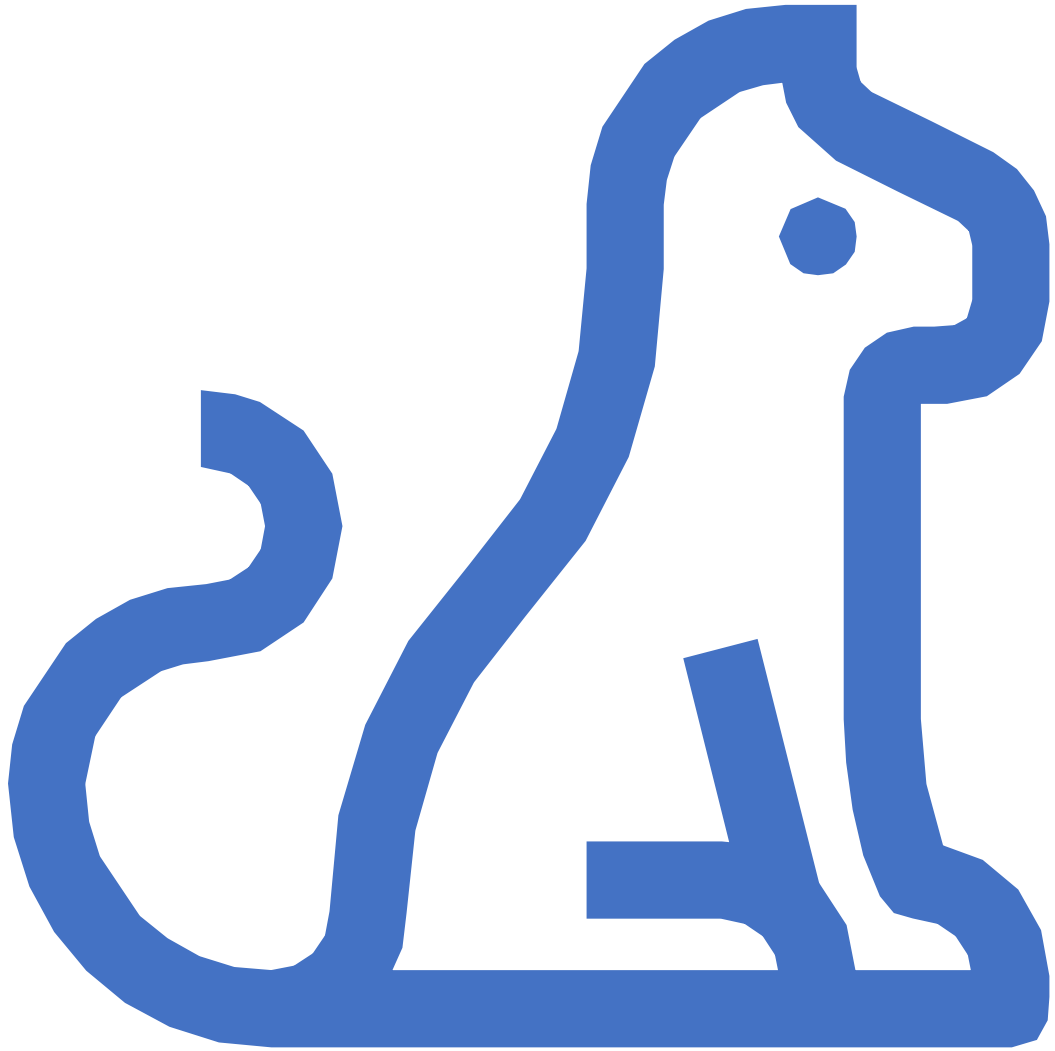
- Imagine we have a Book model related to Author where each Book belongs to one Author
- Let's retrieve all books and their authors
- With Lazy Loading we do 1 query to get all books, then another query for each book to retrieve the author. So N+1 queries (where N is the number of books)
- With Eager Loading we can reduce it to just 2 queries

```sql
select * from books


select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Many to Many Relationships

Animals and their Keepers



Many to Many: Goal

- We want our data to reflect that each animal is looked after by a number of keepers, and each keeper looks after several animals.
- To do this we need to create a pivot table
- A pivot table contains a record for each animal-keeper link which has two values:
  - animal_id
  - keeper_id
- This means we don’t need to change the animal or keeper table to reflect this relationship

Many to Many: Keepers Table

First we create the simple keepers table:

```
public function up()
{
    Schema::create('keepers', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->timestamps();
    });
}
```

We already have the animals table.

Many to Many: Creating the Pivot Table

- We need to create a pivot table for the relationship
- Laravel's convention is that the pivot table is named animal_keeper
  - It removes the pluralisation and joins them in alphabetical order with an underscore
- If your design requires multiple pivot tables for the same two models (e.g. secondary keepers for the animals) then you'll need to name the table differently and specify links manually. See the documentation for information how to do this.
- Create the migration with artisan - php artisan make:migration create_animal_keeper_table

Many to Many: Pivot Table Migration (Cont.)

Creating the pivot table:

```
public function up()
{
    Schema::create('animal_keeper', function (Blueprint $table) {
        // $table->increments('id');
        $table->primary(['animal_id', 'keeper_id']);
        $table->unsignedInteger('animal_id');
        $table->unsignedInteger('keeper_id');
        $table->timestamps();

        $table->foreign('animal_id')->references('id')->
            on('animals')->onDelete('cascade')->
            onUpdate('cascade');

        $table->foreign('keeper_id')->references('id')->
            on('keepers')->onDelete('cascade')->
            onUpdate('cascade');
    });
}
```

You do not want to use the auto generated id column. Why? Think back to database design. We want a composite key.

Here we state that the primary key will be a composite key of animal_id and keeper_id.

Many to Many: Pivot Table Migration (Cont.)

```php
public function up()
{
    Schema::create('animal_keeper', function (Blueprint $table) {
        // $table->increments('id');
        $table->primary(['animal_id', 'keeper_id']);
        $table->unsignedInteger('animal_id');
        $table->unsignedInteger('keeper_id');
        $table->timestamps();

        $table->foreign('animal_id')->references('id')->
            on('animals')->onDelete('cascade')->
            onUpdate('cascade');

        $table->foreign('keeper_id')->references('id')->
            on('keepers')->onDelete('cascade')->
            onUpdate('cascade');
    });
}
```

We still have to create the columns. The primary method doesn't do this for us.

We should always setup referential integrity.

Many to Many: Animal Model

Next, we update the Animal Model:

```php
class Animal extends Model
{
    public function emergencyContact(){...}

    public function enclosure() {...}

    /**
     * The keepers assigned to this animal.
     */
    public function keepers()
    {
        return $this->belongsToMany('App\Keeper');
    }
}
```

We add a method to get the keeper from via the many to many relationship.

So from an animal we can access its keepers.

Many to Many: Keeper Model

Next, we update the Keeper Model:

So from a keeper we can access their animals.

```php
class Keeper extends Model
{
    /**
     * The animals that this keeper is assigned to.
     */
    public function animals()
    {
        return $this->belongsToMany('App\Animal');
    }
}
```

Many to Many: Keepers and Seeding

Again, seeding is a little tricky

```php
class KeepersTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $k = new Keeper;
        $k->name = "Lisa";
        $k->save();
        $k->animals()->attach(1) ; // Leo
        $k->animals()->attach(12) ; // Random animal
    }
}
```

Create a new keeper

Now add some animals into their care.

Laravel's Eloquent system provides the methods attach (and detach) for working with pivot tables.

Unfortunately, you have to provide the ids (and not the models).

When seeding make sure your animal and keeper tables are created before the pivot table – because the pivot table references them!

You should always test in tinker!

Here we find our keeper Lisa.

Next we check she is caring for the 2 animals.

Next we check that if we take her first animal and ask for it’s keepers, then Lisa should be in the list.

```
vagrant@homestead:~/Laravel/swanseazoo$ artisan tinker
Psy Shell v0.9.8 (PHP 7.2.9-1+ubuntu18.04.1+deb.sury.org+1 — cli) by Justin Hileman
>>> App\Keeper::find(1)
=> App\Keeper {#2909
     id: 1,
     name: "Lisa",
     created_at: "2018-10-16 21:49:55",
     updated_at: "2018-10-16 21:49:55",
   }
>>> App\Keeper::find(1)->animals
=> Illuminate\Database\Eloquent\Collection {#2914
     all: [
       App\Animal {#2904
         id: 1,
         name: "Leo",
         weight: 351.6,
         date_of_birth: null,
         enclosure_id: 1,
         created_at: "2018-10-16 21:49:55",
         updated_at: "2018-10-16 21:49:55",
         pivot: Illuminate\Database\Eloquent\Relations\Pivot {#2913
           keeper_id: 1,
           animal_id: 1,
         },
       },
       App\Animal {#2918
         id: 12,
         name: "Vernon",
         weight: 402.01,
         date_of_birth: null,
         enclosure_id: 1,
         created_at: "2018-10-16 21:49:55",
         updated_at: "2018-10-16 21:49:55",
         pivot: Illuminate\Database\Eloquent\Relations\Pivot {#2902
           keeper_id: 1,
           animal_id: 12,
         },
       },
     ],
   }
>>> App\Keeper::find(1)->animals[0]->keepers
=> Illuminate\Database\Eloquent\Collection {#2928
     all: [
       App\Keeper {#2924
         id: 1,
         name: "Lisa",
         created_at: "2018-10-16 21:49:55",
         updated_at: "2018-10-16 21:49:55",
         pivot: Illuminate\Database\Eloquent\Relations\Pivot {#2895
           animal_id: 1,
           keeper_id: 1,
         },
       },
     ],
   }
>>>
```

Pivot Tables with Data

Our pivot table essentially looks like:

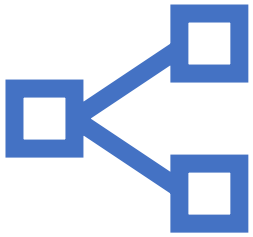
Animal_id	Keep_id
1	16
1	17
1	24

We could also add more data. E.g., the days these keepers are responsible for these aniamals. The table might then look like:

Animal_id	Keep_id	Days_responsible
1	16	Mon,Tue,Wed
1	17	Thur,Fri,Sat
1	24	Sun

Find out at:

https://laravel.com/docs/eloquent-relationships#many-to-many



Seeding Relationships

…with factories

Seeding Relationships with Factories

- Hard coding some models in our seeders allows us to easily add relationships using IDs which we will know
- But, when we start using factories to create random models we will need to do a little more…

Example – The bad way

- Each time we create an Animal model it gets a random name and weight but they all end up in the same enclosure

```php
use Faker\Generator as Faker;


$factory->define(App\Animal::class, function (Faker $faker) {
    return [
        'name' => $faker->firstName,
        'weight' => $faker->randomFloat(2, 300, 500),
        'enclosure_id' => 1, // Central Enclosure
    ];
});
```

```php
use Faker\Generator as Faker;

$factory->define(App\Animal::class, function (Faker $faker) {
    return [
        'name' => $faker->firstName,
        'weight' => $faker->randomFloat(2, 300, 500),
        'enclosure_id' => function () {
            return factory(App\Enclosure::class)->create()->id;
        }
    ];
});
```

Example – Slightly better

- In this factory each time an animal is created a function is called which:
  - Creates a new enclosure using the enclosure factory
  - Pulls out the id of that new factory
  - …and returns it to be used as the enclosure_id
- The problem now is each enclosure has only one animal
- Can you write a function to pick a random enclosure?

Tasks

1. Using the application you created last time. In turn create each of the following relationships, manually seed some data in the seeders, and test these using Tinker
    1. One to one
    2. One to many
    3. Many to many
2. Use a factory to create realistic and related test data for all of your models