Mastering Doctrine Relationships in Symfony 3

With <3 from SymfonyCasts

Chapter 1: Create Genus Note

It's you again! Welcome back friend! In this tutorial, we're diving back into Doctrine: this time to master database relations. And to have fun of course - databases are *super* fun.

Like usual, you should code along with me or risk 7 years of bad luck. Sorry. To do that, download the code from the course page and use the start directory. I already have the code - so I'll startup our fancy, built-in web server:

```
$ ./bin/console server:run
```

You may also need to run composer install and a few other tasks. Check the README in the download for those details.

When you're ready, pull up the genus list page at http://localhost:8000/genus. Nice!

Create the GenusNote Entity

Click to view a specific genus. See these notes down here? These are loaded via a ReactJS app that talks to our app like an API. But, the notes themselves are still hardcoded right in a controller. Bummer! Time to make them dynamic - time to create a second database table to store genus notes.

To do that, create a new GenusNote class in the Entity directory. Copy the ORM use statement from Genus that all entities need and paste it here:

80 lines | src/AppBundle/Entity/GenusNote.php

```
... lines 1 - 2
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
... lines 6 - 10
class GenusNote
{
... lines 13 - 78
}
```

With that, open the "Code"->"Generate" menu - or Cmd + N on a Mac - and select "ORM Class":

80 lines | src/AppBundle/Entity/GenusNote.php

```
... lines 1 - 2
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="genus_note")
 */
class GenusNote
{
... lines 13 - 78
}
```

Bam! This is now an entity!

Next: add the properties we need. Let's see... we need a username, an userAvatarFilename, notes and a createdAt property:

80 lines | src/AppBundle/Entity/GenusNote.php

```php
... lines 1 - 10
class GenusNote
{
... lines 13 - 17
    private $id;
... lines 19 - 22
    private $username;
... lines 24 - 27
    private $userAvatarFilename;
... lines 29 - 32
    private $note;
... lines 34 - 37
    private $createdAt;
... lines 39 - 78
}
```

When we add a user table later - we'll replace username with a relationship to that table. But for now, keep it simple.

Open the "Code"->"Generate" menu again and select "ORM Annotation". Make sure each field type looks right. Hmm, we probably want to change $note to be a text type - that type can hold a lot more than the normal 255 characters:

80 lines | src/AppBundle/Entity/GenusNote.php

```php
... lines 1 - 10
class GenusNote
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $username;

    /**
     * @ORM\Column(type="string")
     */
    private $userAvatarFilename;

    /**
     * @ORM\Column(type="text")
     */
    private $note;

    /**
     * @ORM\Column(type="datetime")
     */
    private $createdAt;
... lines 39 - 78
}
```

Finally, go back to our best friend - the "Code"->"Generate" menu - and generate the getter and setters for every field - except for id. You don't usually want to set the id, but generate a getter for it:

```
85 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 10
class GenusNote
{
... lines 13 - 39
    public function getId()
    {
        return $this->id;
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function setUsername($username)
    {
        $this->username = $username;
    }

    public function getUserAvatarFilename()
    {
        return $this->userAvatarFilename;
    }

    public function setUserAvatarFilename($userAvatarFilename)
    {
        $this->userAvatarFilename = $userAvatarFilename;
    }

    public function getNote()
    {
        return $this->note;
    }

    public function setNote($note)
    {
        $this->note = $note;
    }

    public function getCreatedAt()
    {
        return $this->createdAt;
    }

    public function setCreatedAt($createdAt)
    {
        $this->createdAt = $createdAt;
    }
}
```

Generate Migrations

Entity done! Well, *almost* done - we still need to somehow *relate* each GenusNote to a Genus. We'll handle that in a second.

But first, don't forget to generate a migration for the new table:

```
$ ./bin/console doctrine:migrations:diff
```

Open up that file to make sure it looks right - it lives in app/DoctrineMigrations. CREATE TABLE genus_note - it looks great! Head back to the console and run the migration:

```
$ ./bin/console doctrine:migrations:migrate
```

Adding Fixtures

Man, that was *easy*. We'll want some good dummy notes too. Open up the fixtures.yml file and add a new section for AppBundle\Entity\GenusNote. Start just like before: genus.note_ and - let's create 100 notes - so use 1..100:

15 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml

```
... lines 1 - 8
AppBundle\Entity\GenusNote:
    genus.note_{1..100}:
... lines 11 - 15
```

Next, fill in each property using the Faker functions: username: <username()> and then userAvatarFilename: Ok, eventually users might upload their *own* avatars, but for now, we have two hardcoded options: leanna.jpeg and ryan.jpeg. Let's select one of these randomly with a sweet syntax: 50%? leanna.jpeg : ryan.jpeg. That's Alice awesomeness:

15 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml

```
... lines 1 - 7

AppBundle\Entity\GenusNote:
    genus.note_{1..100}:
        username: <userName()>
        userAvatarFilename: '50%? leanna.jpeg : ryan.jpeg'
... lines 13 - 15
```

The rest are easy: note: <paragraph()> and createdAt: <dateTimeBetween('-6 months', 'now')>:

15 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml

```
... lines 1 - 8
AppBundle\Entity\GenusNote:
    genus.note_{1..100}:
        username: <userName()>
        userAvatarFilename: '50%? leanna.jpeg : ryan.jpeg'
        note: <paragraph()>
        createdAt: <dateTimeBetween('-6 months', 'now')>
```

Ok, run the fixtures!

```
$ ./bin/console doctrine:fixtures:load
```

Double-check them with a query:

```
$ ./bin/console doctrine:query:sql 'SELECT * FROM genus_note'
```

So awesome! Ok team, we have two entities: let's add a relationship!

Chapter 2: The King of Relations: ManyToOne

Selecting between ManyToOne and ManyToMany

Each genus will have *many* genus_notes. But, each genus_note that someone adds will relate to only *one* genus. There are only *two* possible types of relationships, and this is by far the most common. It's called a *ManyToOne* association.

The second type is called a *ManyToMany* association. To use a different example, this would be if each product had many tags, but also each tag related to many products.

And when it comes to Doctrine relations - don't trust the Internet! Some people will try to confuse you with other relationships like OneToMany, OneToOne and some garbage about unidirectional and bidirectional associations. Gross. Ignore it all. I guarantee, all of that will make sense really soon.

So your first job is simple: decide if you have a ManyToOne or ManyToMany relationship. And it's easy. Just answer this question:

> Do either of the sides of the relationship belong to only *one* of the other?

Each genus_note belongs to only *one* genus, so we have a classic ManyToOne relationship.

Setting up a ManyToOne Relation

Forget about Doctrine: just think about the database. If every genus_note should belong to exactly *one* genus, How would you set that up? You'd probably add a genus_id column to the genus_note table. Simple!

Since we need to add a new column to GenusNote, open that entity class. You *probably* feel like you want to add a $genusId integer property here. That makes sense. But don't! Instead, add a $genus property and give it a ManyToOne annotation. Inside that, add targetEntity="Genus":

100 lines | src/AppBundle/Entity/GenusNote.php

```
... lines 1 - 10
class GenusNote
{
... lines 13 - 39
    /**
     * @ORM\ManyToOne(targetEntity="Genus")
     */
    private $genus;
... lines 44 - 98
}
```

Tip

You can also use the *full* namespace: AppBundle\Entity\Genus - and that's required if the two entities do not live in the same namespace/directory.

Umm, guys? That's it. Relationship finished. Seriously.

Chapter 3: Saving a Relationship

Doctrine will create a genus_id integer column for this property and a foreign key to genus.

Use the "Code"->"Generate" menu to generate the getter and setter:

100 lines | src/AppBundle/Entity/GenusNote.php

```
... lines 1 - 10
class GenusNote
{
... lines 13 - 89
    public function getGenus()
    {
        return $this->genus;
    }

    public function setGenus($genus)
    {
        $this->genus = $genus;
    }
}
```

Add a Genus type-hint to setGenus():

100 lines | src/AppBundle/Entity/GenusNote.php

```
... lines 1 - 10
class GenusNote
{
... lines 13 - 94
    public function setGenus(Genus $genus)
    {
        $this->genus = $genus;
    }
}
```

Yes, when we call setGenus(), we'll pass it an entire Genus *object* not an ID. More on that soon.

Generate the Migration

Generate the migration for the change:

```
$ ./bin/console doctrine:migrations:diff
```

And then go check it out... Wow - look at this!

```
33 lines | app/DoctrineMigrations/Version20160207091756.php
... lines 1 - 10
class Version20160207091756 extends AbstractMigration
{
    public function up(Schema $schema)
    {
... lines 15 - 17
        $this->addSql("ALTER TABLE genus_note ADD genus_id INT DEFAULT NULL");
        $this->addSql("ALTER TABLE genus_note ADD CONSTRAINT FK_6478FCEC85C4074C FOREIGN KEY (genus_id) REFERENC
        $this->addSql("CREATE INDEX IDX_6478FCEC85C4074C ON genus_note (genus_id)");
    }
... lines 22 - 31
}
```

Even though we called the property genus, it sets up the database *exactly* how you would have normally: with a genus_id integer column and a foreign key. And we did this with basically 2 lines of code.

Run the migration to celebrate!

```
$ ./bin/console doctrine:migrations:migrate
```

Now, how do we actually *save* this relationship?

Saving a Relation

Head back to GenusController. In newAction(), create a new GenusNote - let's see *how* we can relate this to a Genus:

```
109 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 17
    public function newAction()
    {
        $genus = new Genus();
        $genus->setName('Octopus'.rand(1, 100));
        $genus->setSubFamily('Octopodinae');
        $genus->setSpeciesCount(rand(100, 99999));

        $note = new GenusNote();
... lines 26 - 37
    }
... lines 39 - 107
}
```

I'll paste in some code here to set each of the normal properties - they're *all* required in the database right now:

```
109 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 24
        $note = new GenusNote();
        $note->setUsername('AquaWeaver');
        $note->setUserAvatarFilename('ryan.jpeg');
        $note->setNote('I counted 8 legs... as they wrapped around me');
        $note->setCreatedAt(new \DateTime('-1 month'));
... lines 30 - 109
```

So how can we link *this* GenusNote to this Genus? Simple: $note->setGenus() and pass it the entire $genus object:

```
109 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 24
        $note = new GenusNote();
        $note->setUsername('AquaWeaver');
        $note->setUserAvatarFilename('ryan.jpeg');
        $note->setNote('I counted 8 legs... as they wrapped around me');
        $note->setCreatedAt(new \DateTime('-1 month'));
        $note->setGenus($genus);
... lines 31 - 109
```

That's it. Seriously! The only tricky part is that you set the entire *object*, not the ID. With Doctrine relations, you almost need to forget about ID's entirely: your job is to link one object to another. When you save, Doctrine works out the details of how this should look in the database.

Don't forget to persist the $note:

```
109 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 17
    public function newAction()
    {
... lines 20 - 32
        $em->persist($genus);
        $em->persist($note);
        $em->flush();
... lines 36 - 37
    }
... lines 39 - 107
}
```

And, you can persist in *any* order: Doctrine automatically knows that it needs to insert the genus first and then the genus_note. That's really powerful.

Defaulting the isPublished Field

And simple! Head to the browser to check it out - /genus/new. Whoops - an error: the is_published property cannot be null. My bad - that's totally unrelated.

In Genus, give the $isPublished field a default value of true:

```
95 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 10
class Genus
{
... lines 13 - 39
    /**
     * @ORM\Column(type="boolean")
     */
    private $isPublished = true;
... lines 44 - 93
}
```

Now, if you forget to set this field - it'll default to true instead of null.

Woo! No errors this time. Check out the queries for the page. Nice! Two insert queries: INSERT INTO genus and then INSERT INTO genus_note using 46: the new genus's ID.

With two lines to setup the relationship, and one line to link a GenusNote to a Genus, you've got a fantastic new relationship.

Chapter 4: JoinColumn & Relations in Fixtures

Is the relationship *required* in the database? I mean, could I save a GenusNote *without* setting a Genus on it? Actually, I could! Unlike a normal column, relationship columns - for whatever reason - are *optional* by default. But does it make sense to allow a GenusNote without a Genus? No! That's crazy talk! Let's prevent it.

Find the ManyToOne annotation and add a *new* annotation below it: JoinColumn. Inside, set nullable=false:

101 lines | src/AppBundle/Entity/GenusNote.php

```
... lines 1 - 10
class GenusNote
{
... lines 13 - 39
    /**
     * @ORM\ManyToOne(targetEntity="Genus")
     * @ORM\JoinColumn(nullable=false)
     */
    private $genus;
... lines 45 - 99
}
```

The JoinColumn annotation controls how the foreign key looks in the database. And obviously, it's optional. Another option is onDelete: that literally changes the ON DELETE behavior in your database - the default is RESTRICT, but you can also use CASCADE or SET NULL.

Anyways, we just made a schema change - so time to generate a migration!

```
$ ./bin/console doctrine:migrations:diff
```

This time, I'll be lazy and trust that it's correct. Run it!

```
$ ./bin/console doctrine:migrations:migrate
```

When Migrations Go Wrong

Ah, it explodes! Null value not allowed? Why? Think about what's happening: we have a bunch of *existing* GenusNote rows in the database, and each still has a null genus_id. We can't set that column to NOT NULL because of the data that's already in the database.

If the app were already deployed to production, we would need to fix the migration: maybe UPDATE each existing genus_note and set the genus_id to the first genus in the table.

But, alas! We haven't deployed to production yet: so there isn't any existing production database that we'll need to migrate. Instead, just start from scratch: drop the database completely, re-create it, and re-migrate from the beginning:

```
$ ./bin/console doctrine:database:drop --force
```

```
$ ./bin/console doctrine:database:create
```

```
$ ./bin/console doctrine:migrations:migrate
```

Phew! Now it works great.

Tip

If you *still* get an error while running the migration, it's because of a MySQL change! Find the details here:

http://bit.ly/migrations-tweak

Relations in Fixtures

Last step! Our fixtures are broken: we need to associate each GenusNote with a Genus. We know how to set normal properties, like username and userAvatarFilename. But how can we set relations? As usual with Alice: it's *so* nice. Use genus: @ then the internal name of one of the 10 genuses - like genus_1. That's it!

But, you know what? That's not awesome enough. I *really* want this to be a *random* Genus. Ok: change that genus_1 to genus_*:

16 lines | src/AppBundle/DataFixtures/ORM/fixtures.yml

```
... lines 1 - 8
AppBundle\Entity\GenusNote:
    genus.note_{1..100}:
... lines 11 - 14
        genus: '@genus_*'
```

Alice will now look at the 10 Genus objects matching this pattern and select a random one each time.

Reload the fixtures:

```
$ ./bin/console doctrine:fixtures:load
```

It's alive! Check out the results again with doctrine:query:sql:

```
$ ./bin/console doctrine:query:sql 'SELECT * FROM genus_note'
```

Every single one has a random genus. Do you love it? I love it.

Chapter 5: Controller Magic: Param Conversion

Time to *finally* make these genus notes dynamic! Woo!

Remember, those are loaded by a ReactJS app, and *that* makes an AJAX call to an API endpoint in GenusController. Here it is: getNotesAction():

```
109 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 90
    /**
     * @Route("/genus/{genusName}/notes", name="genus_show_notes")
     * @Method("GET")
     */
    public function getNotesAction($genusName)
    {
... lines 97 - 106
    }
}
```

Step 1: use the genusName argument to query for a Genus object. But you guys already know how to do that: get the entity manager, get the Genus repository, and then call a method on it - like findOneBy():

```
109 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 54
    /**
     * @Route("/genus/{genusName}", name="genus_show")
     */
    public function showAction($genusName)
    {
        $em = $this->getDoctrine()->getManager();

        $genus = $em->getRepository('AppBundle:Genus')
            ->findOneBy(['name' => $genusName]);
... lines 64 - 88
    }
... lines 90 - 107
}
```

Old news.

Let's do something *much* cooler. First, change {genusName} in the route to {name}, but don't ask why yet. Just trust me:

```
110 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 90
    /**
     * @Route("/genus/{name}/notes", name="genus_show_notes")
     * @Method("GET")
     */
    public function getNotesAction(Genus $genus)
    {
... lines 97 - 107
    }
}
```

This doesn't change the URL to this page... but it *does* break all the links we have to this route.

To fix those, go to the terminal and search for the route name:

```
$ git grep genus_show_notes
```

Oh cool! It's only used in *one* spot. Open show.html.twig and find it at the bottom. Just change the key from genusName to name:

```
40 lines | app/Resources/views/genus/show.html.twig
... lines 1 - 23
{% block javascripts %}
... lines 25 - 30
    <script type="text/babel">
        var notesUrl = '{{ path('genus_show_notes', {'name': genus.name}) }}';
... lines 33 - 37
    </script>
{% endblock %}
```

Using Param Conversion

So... doing all of this didn't change *anything*. So why did I make us do all that? Let me show you. You might *expect* me to add a $name argument. But don't! Instead, type-hint the argument with the Genus class and then add $genus:

```
110 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 90
    /**
     * @Route("/genus/{name}/notes", name="genus_show_notes")
     * @Method("GET")
     */
    public function getNotesAction(Genus $genus)
    {
... lines 97 - 107
    }
}
```

What? I just violated one of the *cardinal* rules of routing: that every argument must match the *name* of a routing wildcard. The truth is, if you type-hint an argument with an entity class - like Genus - Symfony will automatically query for it. This works as long as the wildcard has the same name as a property on Genus. *That's* why we changed {genusName} to {name}. Btw, this is called "param conversion".

Tip

Param Conversion comes from the SensioFrameworkExtraBundle.

Dump the $genus to prove it's working:

```
110 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 94
    public function getNotesAction(Genus $genus)
    {
        dump($genus);
... lines 98 - 107
    }
}
```

Go back and refresh! We don't see the dump because it's actually an AJAX call - one that happens automatically each second.

Seeing the Profiler for an AJAX Request

But don't worry! Go to /_profiler to see a list of the most recent requests, including AJAX requests. Select one of these: this is the profiler for that AJAX call, and in the Debug panel... *there's* the dump. It's alive!

So be lazy: setup your routes with a wildcard that matches a property name and use a type-hint to activate param conversion. If a genus can't be found for this page, it'll automatically 404. And if you *can't* use param conversion because you need to run a custom query: cool - just get the entity manager and query like normal. Use the shortcut when it helps!

Chapter 6: OneToMany: Inverse Side of the Relation

We have the Genus object. So how can we get the collection of related GenusNote Well, the simplest way is just to make a query - in fact, you could fetch the GenusNote repository and call findBy(['genus' => $genus]). It's really that simple.

Tip

You can also pass the Genus's *ID* in queries, instead of the entire Genus object.

But what if we could be even lazier? What if we were able to just say $genus->getNotes()? That'd be cool! Let's hook it up!

Setting up the OneToMany Side

Open up GenusNote. Remember, there are only two types of relationships: ManyToOne and ManyToMany. For this, we needed ManyToOne.

But actually, you can think about any relationship in two directions: each GenusNote has *one* Genus. Or, each Genus has many GenusNote. And in Doctrine, you can *map* just one side of a relationship, *or* both. Let me show you.

Open Genus and add a new $notes property:

```
111 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 11
class Genus
{
... lines 14 - 48
    private $notes;
... lines 50 - 109
}
```

This is the *inverse* side of the relationship. Above this, add a OneToMany annotation with targetEntity set to GenusNote and a mappedBy set to genus - that's the *property* in GenusNote that forms the main, side of the relation:

```
111 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 11
class Genus
{
... lines 14 - 45
    /**
     * @ORM\OneToMany(targetEntity="GenusNote", mappedBy="genus")
     */
    private $notes;
... lines 50 - 109
}
```

But don't get confused: there's still only *one* relation in the database: but now there are two ways to access the data on it: $genusNote->getGenus() and now $genus->getNotes().

Add an inversedBy set to notes on this side: to point to the other property:

```
101 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 10
class GenusNote
{
... lines 13 - 39
    /**
     * @ORM\ManyToOne(targetEntity="Genus", inversedBy="notes")
     * @ORM\JoinColumn(nullable=false)
     */
    private $genus;
... lines 45 - 99
}
```

I'm not sure why this is *also* needed - it feels redundant - but oh well.

Next, generate a migration! Not! This is *super* important to understand: this didn't cause any changes in the database: we just added some sugar to our Doctrine setup.

Add the ArrayCollection

Ok, one last detail: in Genus, add a __construct() method and initialize the notes property to a new ArrayCollection:

```
111 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 11
class Genus
{
... lines 14 - 50
    public function __construct()
    {
        $this->notes = new ArrayCollection();
    }
... lines 55 - 109
}
```

This object is like a PHP array on steroids. You can loop over it like an array, but it has other super powers we'll see soon. Doctrine always returns one of these for relationships instead of a normal PHP array.

Finally, go to the bottom of the class and add a getter for notes:

```
111 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 11
class Genus
{
... lines 14 - 105
    public function getNotes()
    {
        return $this->notes;
    }
}
```

Time to try it out! In getNotesAction() - just for now - loop over $genus->getNotes() as $note and dump($note):

112 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 94
    public function getNotesAction(Genus $genus)
    {
        foreach ($genus->getNotes() as $note) {
            dump($note);
        }
... lines 100 - 109
    }
}
```

Head back and refresh! Let the AJAX call happen and then go to /_profiler to find the dump. Yes! A *bunch* of GenusNote objects.

Oh, and look at the Doctrine section: you can see the extra query that was made to fetch these. This query doesn't happen until you *actually* call $genus->getNotes(). Love it!

Owning and Inverse Sides

That was pretty easy: *if* you want this shortcut, just add a few lines to map the *other* side of the relationship.

But actually, you just learned the *hardest* thing in Doctrine. Whenever you have a relation: start by figuring out which entity should have the foreign key column and then add the ManyToOne relationship there first. *This* is the only side of the relationship that you *must* have - it's called the "owning" side.

Mapping the *other* side - the OneToMany *inverse* side - is always optional. I don't map it until I *need* to - either because I want a cute shortcut like $genus->getNotes() or because I want to join in a query from Genus to GenusNote - something we'll see in a few minutes.

Tip

ManyToMany relationships - the only other *real* type of relationship - also have an owning and inverse side, but you can *choose* which is which. We'll save that topic for later.

Now, there is *one* gotcha. Notice I did *not* add a setNotes() method to Genus. That's because you cannot *set* data on the inverse side: you can only set it on the *owning* side. In other words, $genusNote->setGenus() will work, but $genus->setNotes() would *not* work: Doctrine will ignore that when saving.

So when you setup the inverse side of a relation, do yourself a favor: do *not* generate the setter function.

Chapter 7: Order By with a OneToMany

Let's finish this! Ultimately, we need to create the same $notes structure, but with the *real* data. Above the foreach add a new $notes variable. Inside, add a new entry to that and start populating it with id => $note->getId():

117 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 94
    public function getNotesAction(Genus $genus)
    {
        $notes = [];

        foreach ($genus->getNotes() as $note) {
            $notes[] = [
                'id' => $note->getId(),
            ];
        }
... lines 104 - 114
    }
}
```

Hey! Where's my autocompletion on that method!? Check out the getNotes() method in Genus. Ah, there's no @return - so PhpStorm has no idea what that returns. Sorry PhpStorm - my bad. Add some PhpDoc with @return ArrayCollection|GenusNote[]:

114 lines | src/AppBundle/Entity/Genus.php

```
... lines 1 - 11
class Genus
{
... lines 14 - 105
    /**
     * @return ArrayCollection|GenusNote[]
     */
    public function getNotes()
    {
        return $this->notes;
    }
}
```

This will autocomplete any methods from ArrayCollection *and* auto-complete from GenusNote if we loop over these results.

Now we get autocompletion for getId(). Next, add username => $note->getUsername() and I'll paste in the other fields: avatarUri, note and createdAt. Ok, delete that hardcoded stuff!

116 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 94
    public function getNotesAction(Genus $genus)
    {
        $notes = [];

        foreach ($genus->getNotes() as $note) {
            $notes[] = [
                'id' => $note->getId(),
                'username' => $note->getUsername(),
                'avatarUri' => '/images/'.$note->getUserAvatarFilename(),
                'note' => $note->getNote(),
                'date' => $note->getCreatedAt()->format('M d, Y')
            ];
        }

        $data = [
            'notes' => $notes
        ];

        return new JsonResponse($data);
    }
}
```

Deep breath: moment of truth. Refresh! Ha! There are the 15 beautiful, random notes, courtesy of the AJAX request, Alice and Faker.

Ordering the OneToMany

But wait - the order of the notes is weird: these should really be ordered from newest to oldest. That's the *downside* of using the $genus->getNotes() shortcut: you can't customize the query - it just happens magically in the background.

Well ok, I'm lying a *little* bit: you *can* control the order. Open up Genus and find the $notes property. Add another annotation: @ORM\OrderBy with {"createdAt"="DESC"}:

115 lines | src/AppBundle/Entity/Genus.php

```
... lines 1 - 11
class Genus
{
... lines 14 - 45
    /**
     * @ORM\OneToMany(targetEntity="GenusNote", mappedBy="genus")
     * @ORM\OrderBy({"createdAt" = "DESC"})
     */
    private $notes;
... lines 51 - 113
}
```

I know, the curly-braces and quotes look a little crazy here: just Google this if you can't remember the syntax. I do!

Ok, refresh! Hey! Newest ones on top, oldest ones on the bottom. So we do have *some* control. But if you need to go further - like only returning the GenusNotes that are less than 30 days old - you'll need to do a little bit more work.

Chapter 8: Tricks with ArrayCollection

Oh man, the project manager just came to me with a new challenge. Showing all the notes below is great, but they want a new section on top to easily see how many notes have been posted during the past 3 months.

Hmm. In showAction(), we need to somehow count all the recent notes for this Genus. We *could* start with $recentNotes = $genus->getNotes()... but that's *everything*. Do we need to finally stop being lazy and make a custom query? Not necessarily.

Remember: getNotes() returns an ArrayCollection object and it has some tricks on it - like a method for filtering! Chain a call to the filter() method and pass this an anonymous function with a GenusNote argument. The ArrayCollection will call this function for *each* item. If we return true, it stays. If we return false, it disappears.

Easy enough! Return $note->getCreatedAt() > new \DateTime('-3 months');:

122 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 57
    public function showAction($genusName)
    {
... lines 60 - 85
        $recentNotes = $genus->getNotes()
            ->filter(function(GenusNote $note) {
                return $note->getCreatedAt() > new \DateTime('-3 months');
            });
... lines 90 - 94
    }
... lines 96 - 120
}
```

Next, pass a new recentNoteCount variable into twig that's set to count($recentNotes):

122 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 57
    public function showAction($genusName)
    {
... lines 60 - 90
        return $this->render('genus/show.html.twig', array(
            'genus' => $genus,
            'recentNoteCount' => count($recentNotes)
        ));
    }
... lines 96 - 120
}
```

In the template, add a new dt for Recent Notes and a dd with {{ recentNoteCount }}:

```
42 lines | app/Resources/views/genus/show.html.twig
... lines 1 - 4
{% block body %}
    <h2 class="genus-name">{{ genus.name }}</h2>

    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 17
                <dt>Recent Notes</dt>
                <dd>{{ recentNoteCount }}</dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 25 - 42
```

All right - give it a try! Refresh. Six notes - perfect: we clearly have a lot more than six in total.

The ArrayCollection has lots of fun methods on it like this, including contains(), containsKey(), forAll(), map() and other goodies.

Don't Abuse ArrayCollection

Do you see any downsides to this? There's one big one: this queries for *all* of the notes, even though we don't need them all. If you know you'll only ever have a few notes, no big deal. But if you may have *many* notes: don't do this - you *will* feel the performance impact of loading up hundreds of extra objects.

So what's the *right* way? Finally making a custom query that only returns the GenusNote objects we need. Let's do that next.

Chapter 9: Querying on a Relationship

We need to create a query that returns the GenusNotes that belong to a specific Genus *and* are less than 3 months old. To keep things organize, custom queries to the GenusNote table should live in a GenusNoteRepository. Ah, but we don't have one yet! No problem: copy GenusRepository.php to GenusNoteRepository.php, rename the class and clear it out:

22 lines | src/AppBundle/Repository/GenusNoteRepository.php

```
... lines 1 - 2
namespace AppBundle\Repository;
... lines 4 - 6
use Doctrine\ORM\EntityRepository;

class GenusNoteRepository extends EntityRepository
{
... lines 11 - 20
}
```

Add a new public function findAllRecentNotesForGenus() and give this a Genus argument:

22 lines | src/AppBundle/Repository/GenusNoteRepository.php

```
... lines 1 - 8
class GenusNoteRepository extends EntityRepository
{
    /**
     * @param Genus $genus
     * @return GenusNote[]
     */
    public function findAllRecentNotesForGenus(Genus $genus)
    {
... lines 17 - 19
    }
}
```

Excellent! And just like before - start with return $this->createQueryBuilder() with genus_note as a query alias. For now, don't add anything else: finish with the standard ->getQuery() and ->execute():

22 lines | src/AppBundle/Repository/GenusNoteRepository.php

```
... lines 1 - 8
class GenusNoteRepository extends EntityRepository
{
... lines 11 - 14
    public function findAllRecentNotesForGenus(Genus $genus)
    {
        return $this->createQueryBuilder('genus_note')
            ->getQuery()
            ->execute();
    }
}
```

Doctrine doesn't know about this new repository class yet, so go tell it! In GenusNote, find @ORM\Entity and add repositoryClass="AppBundle\Repository\GenusNoteRepository":

```
101 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 6
/**
 * @ORM\Entity(repositoryClass="AppBundle\Repository\GenusNoteRepository")
 * @ORM\Table(name="genus_note")
 */
class GenusNote
{
... lines 13 - 99
}
```

Finally, use the new method in GenusController - $recentNotes = $em->getRepository('AppBundle:GenusNote')->findAllRecentNotesForGenus() and pass it the $genus object from above:

```
120 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 57
    public function showAction($genusName)
    {
... lines 60 - 85
        $recentNotes = $em->getRepository('AppBundle:GenusNote')
            ->findAllRecentNotesForGenus($genus);
... lines 88 - 92
    }
... lines 94 - 118
}
```

Obviously, we're not done yet - but it should at least *not* break. Refresh. Ok, 100 recent comments - that's perfect: it's returning *everything*. Oh, you know what isn't perfect? My lame typo - change that to the word Recent. Embarrassing for me:

```
42 lines | app/Resources/views/genus/show.html.twig
... lines 1 - 4
{% block body %}
    <h2 class="genus-name">{{ genus.name }}</h2>

    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 17
                <dt>Recent Notes</dt>
... line 19
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 25 - 42
```

Using the Relationship in the Query

Head back to the repository. This query is pretty simple actually: add an ->andWhere('genus_note.genus = :genus'). Then, fill in :genus with ->setParameter('genus', $genus):

27 lines | src/AppBundle/Repository/GenusNoteRepository.php

```
... lines 1 - 8
class GenusNoteRepository extends EntityRepository
{
... lines 11 - 14
    public function findAllRecentNotesForGenus(Genus $genus)
    {
        return $this->createQueryBuilder('genus_note')
            ->andWhere('genus_note.genus = :genus')
            ->setParameter('genus', $genus)
... lines 20 - 22
            ->getQuery()
            ->execute();
    }
}
```

This a simple query - equivalent to SELECT * FROM genus_note WHERE genus_id = some number. The only tricky part is that the andWhere() is done on the genus property - not the genus_id column: you *always* reference property names with Doctrine.

Finish this with another andWhere('genus_note.createdAt > :recentDate') and ->setParameter('recentDate', new \DateTime('-3 months')):

27 lines | src/AppBundle/Repository/GenusNoteRepository.php

```
... lines 1 - 8
class GenusNoteRepository extends EntityRepository
{
... lines 11 - 14
    public function findAllRecentNotesForGenus(Genus $genus)
    {
        return $this->createQueryBuilder('genus_note')
            ->andWhere('genus_note.genus = :genus')
            ->setParameter('genus', $genus)
            ->andWhere('genus_note.createdAt > :recentDate')
            ->setParameter('recentDate', new \DateTime('-3 months'))
... line 22
            ->getQuery()
            ->execute();
    }
}
```

Perfect! Go back and try it - the count *should* go back to 6. There we go! But now, instead of fetching *all* the notes just to count some of them, we're only querying for the ones we need. *And*, Doctrine *loves* returning objects, but you could make this even *faster* by returning *only* the count from the query, instead of the objects. Don't optimize too early - but when you're ready, we cover that in our Going Pro with Doctrine Queries.

Chapter 10: Query across a JOIN (and Love it)

What about a JOIN query with Doctrine? Well, they're really cool.

Here's our *last* challenge. Go to /genus. Right now, this list is ordered by the speciesCount property. Instead, I want to order by which genus has the most recent note - a column that lives on an entirely different table.

In GenusRepository, the list page uses the query in findAllPublishedOrderedBySize(). Rename that to findAllPublishedOrderedByRecentlyActive():

24 lines | src/AppBundle/Repository/GenusRepository.php

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
... lines 10 - 12
    public function findAllPublishedOrderedByRecentlyActive()
    {
... lines 15 - 21
    }
}
```

Go change it in GenusController too:

120 lines | src/AppBundle/Controller/GenusController.php

```
... lines 1 - 12
class GenusController extends Controller
{
... lines 15 - 42
    public function listAction()
    {
... lines 45 - 46
        $genuses = $em->getRepository('AppBundle:Genus')
            ->findAllPublishedOrderedByRecentlyActive();
... lines 49 - 52
    }
... lines 54 - 118
}
```

Tip

PhpStorm has a great refactoring tool to rename everything automatically. Check out the Refactoring in PhpStorm tutorial.

Adding the Join

Let's go to work! Remove the orderBy line. We need to order by the createdAt field in the genus_note table. And we know from SQL that we can't do that unless we *join* over to that table. Do that with, ->leftJoin('genus') - because that's the alias we set on line 15 - genus.notes:

24 lines | src/AppBundle/Repository/GenusRepository.php

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
... lines 10 - 12
    public function findAllPublishedOrderedByRecentlyActive()
    {
        return $this->createQueryBuilder('genus')
            ->andWhere('genus.isPublished = :isPublished')
            ->setParameter('isPublished', true)
            ->leftJoin('genus.notes', 'genus_note')
... line 19
            ->getQuery()
            ->execute();
    }
}
```

Why notes? This is the *property* name on Genus that references the relationship. And just by mentioning it, Doctrine has all the info it needs to generate the full JOIN SQL.

Joins and the Inverse Relation

Remember, this is the optional, *inverse* side of the relationship: we added this for the convenience of being able to say $genus->getNotes():

115 lines | src/AppBundle/Entity/Genus.php

```
... lines 1 - 11
class Genus
{
... lines 14 - 45
    /**
     * @ORM\OneToMany(targetEntity="GenusNote", mappedBy="genus")
     * @ORM\OrderBy({"createdAt" = "DESC"})
     */
    private $notes;
... lines 51 - 106
    /**
     * @return ArrayCollection|GenusNote[]
     */
    public function getNotes()
    {
        return $this->notes;
    }
}
```

And this is the *second* reason you might decide to map the inverse side of the relation: it's required if you're doing a JOIN in this direction.

Tip

Actually, not true! As Stof suggested in the comments on this page, it *is* possible to query over this join *without* mapping this side of the relationship, it just takes a little bit more work:

```
$this->createQueryBuilder('genus')
    // ...
    ->leftJoin(
        'AppBundle:GenusNote',
        'genus_note',
        \Doctrine\ORM\Query\Expr\Join::WITH,
        'genus = genus_note.genus'
    )
    // ...
```

Back in GenusRepository, give leftJoin() a second argument: genus_note - this is the alias we can use during the rest of the query to reference fields on the joined genus_note table. This allows us to say ->orderBy('genus_note.createdAt', 'DESC'):

24 lines | src/AppBundle/Repository/GenusRepository.php

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
... lines 10 - 12
    public function findAllPublishedOrderedByRecentlyActive()
    {
        return $this->createQueryBuilder('genus')
            ->andWhere('genus.isPublished = :isPublished')
            ->setParameter('isPublished', true)
            ->leftJoin('genus.notes', 'genus_note')
            ->orderBy('genus_note.createdAt', 'DESC')
            ->getQuery()
            ->execute();
    }
}
```

That's it! Same philosophy of SQL joining... but it takes less work.

Head back and refresh! Ok, the order *did* change. Look at the first one - the top note is from February 15th, the second genus has a note from February 11 and at the bottom, the most recent note is December 21st. I think we got it!

Question: when we added the join, did it change what the query returned? Before, it returned an array of Genus objects... but now, does it also return the joined GenusNote objects? No: a join does *not* affect *what* is returned from the query: we're still *only* selecting from the genus table. There's a lot more about that in our Doctrine Queries tutorial.

Ok, that's it! That's everything - you are truly *dangerous* with Doctrine now. Sure, there *are* some more advanced topics - like Doctrine events, inheritance and ManyToMany relations - but we'll save those for another day. Get to work on that project... or keep going with me to learn more Symfony! I promise, more bad jokes - like worse than ever.

See you next time!