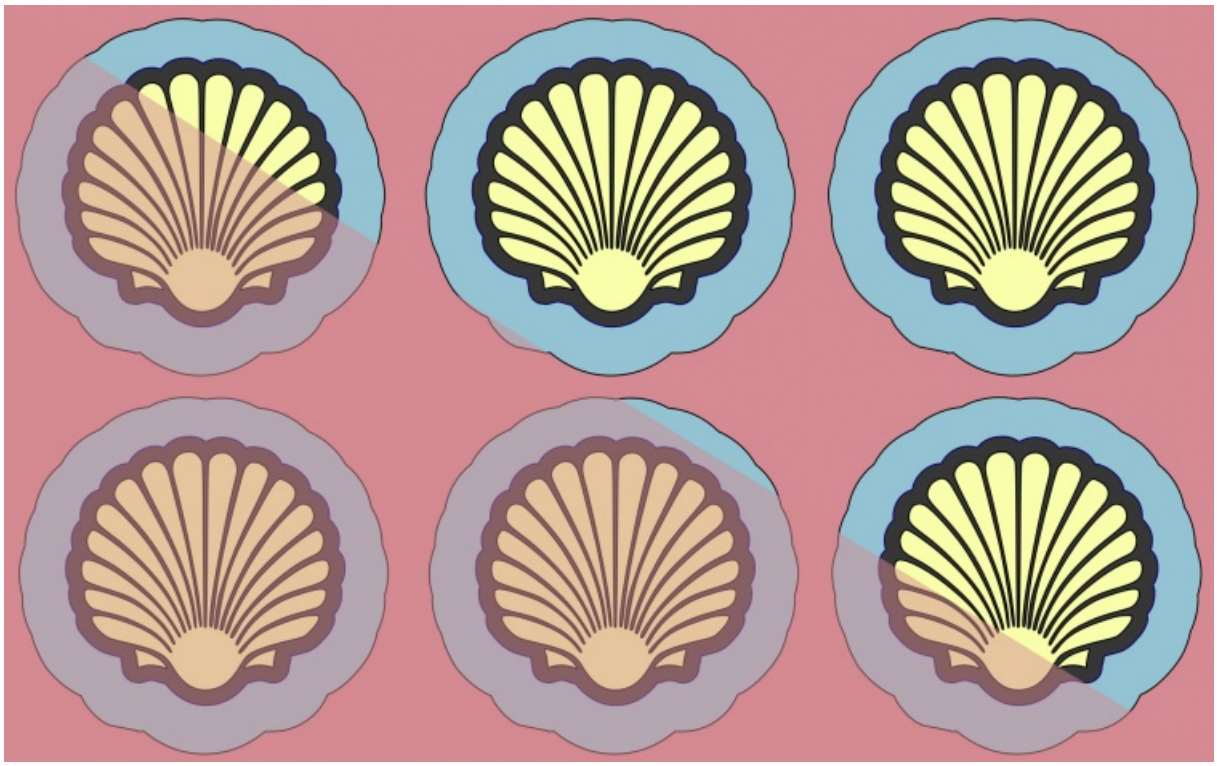
Doctrine Collections: ManyToMany, Forms & other Complex Relations



With <3 from SymfonyCasts

Chapter 1: Give me Clean URL Strings (slugs!)

Yes! Collections! Ladies and gentleman, this course is going to take us somewhere special: to the center of *two* topics that each, single-handedly, have the power to make you hate Doctrine and hate Symfony forms. Seriously, Doctrine and the form system are probably the two most *powerful* things included in the Symfony Framework... and yet... they're also the two parts that drive people insane! How can that be!?

The answer: collections. Like, when you have a database relationship where one Category is related to a collection of Products. And for forms, it's how you build a form where you can edit that category and add, remove or edit the related products all from one screen. If I may, it's a collection of chaos.

But! But, but but! I have good news: if we can understand just a *few* important concepts, Doctrine collections are going to fall into place beautifully. So let's take this collection of chaos and turn it into a collection of.. um... something awesome... like, a collection of chocolate, or ice cream. Let's do it!

Code and Setup!

You should *definitely* code along with me by downloading the course code from this page, unzipping it, and then finding the start/ directory. And don't forget to also pour yourself a fresh cup of coffee or tea: you deserve it.

That start/ directory will have the exact code that you see here. Follow the instructions in the README.md file: it will get your project setup.

The last step will be to open a terminal, move into the directory, and start the built-in PHP web server with:

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

Now, head to your browser and go to http://localhost:8000 to pull up our app: Aquanote! Head to /genus: this lists all of the *genuses* in the system, which is a type of animal classification.

Tip

The plural form of genus is actually *genera*. But irregular plural words like this can make your code a bit harder to read, and don't work well with some of the tools we'll be using. Hence, we use the simpler, genuses.

Clean, Unique URLs

Before we dive into collection stuff, I *need* to show you something else first. Don't worry, it's cool. Click one of the genuses. Now, check out the URL: we're using the *name* in the URL to identify this genus. But this has two problems. First, well, it's kind of ugly: I don't really like upper case URLs, and if a genus had a *space* in it, this would look *really* ugly - nobody likes looking at %20. Second, the name might not be unique! At least while we're developing, we might have two genuses with the same name - like Aurelia. If you click the second one... well, this is actually showing me the *first*: our query *always* finds only the first Genus matching this name.

How could I let this happen!? Honestly, it was a shortcut: I wanted to focus on more important things before. But now, it's time to right this wrong.

What we really need is a clean, unique version of the name in the url. This is commonly called a *slug*. No, no, not the slimy animal - it's just a unique name.

Create the slug Field

How can we create a slug? First, open the Genus entity and add a new property called slug:

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

```
... lines 1 - 12
class Genus
{
... lines 15 - 27
    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $slug;
... lines 32 - 163
}
```

We *will* store this in the database like any other field. The only difference is that we'll force it to be unique in the database.

Next, go to the bottom and use the "Code"->"Generate" menu, or Command+N on a Mac, to generate the getter and setter for slug:

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

```
... lines 1 - 12
class Genus
{
... lines 15 - 154
    public function getSlug()
    {
        return $this->slug;
    }

    public function setSlug($slug)
    {
        $this->slug = $slug;
    }
}
```

Finally, as always, generate a migration. I'll open a new terminal tab, and run:

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

Open that file to make sure it looks right:

37 lines | app/DoctrineMigrations/Version20160921253370.php

```
... lines 1 - 10
class Version20160921253370 extends AbstractMigration
{
... lines 13 - 15
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('ALTER TABLE genus ADD slug VARCHAR(255) NOT NULL');
        $this->addSql('CREATE UNIQUE INDEX UNIQ_38C5106E989D9B62 ON genus (slug)');
    }
... lines 24 - 27
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('DROP INDEX UNIQ_38C5106E989D9B62 ON genus');
        $this->addSql('ALTER TABLE genus DROP slug');
    }
}
```

Perfect! It adds a column, and gives it a unique index. Run it:

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

Ah, Migration Failed!

Oh no! It failed! Why!? Since we *already* have genuses in the database, when we try to add this new column... which should be unique... every genus is given the same, blank string. If we had already deployed this app to production, we would need to do a bit more work, like make the slug field *not* unique at first, write a migration to generate all of the slugs, and *then* make it unique.

Fortunately we haven't deployed this yet, so let's take the easy road. Drop the database:

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

Then recreate it, and run all of the migrations from the beginning:

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

Much better. So.... how do we actually set the slug field for each Genus?

Chapter 2: DoctrineExtensions: Sluggable

Since slug is just a *normal* field, we *could* open our fixtures file and add the slug manually here to set it:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
        name: <genus()>
... lines 4 - 37
```

LAME! There's a cooler way: what if it were automagically generated from the name? That would be awesome! Let's go find some magic!

Installing StofDoctrineExtensionsBundle

Google for a library called StofDoctrineExtensionsBundle. You can find its docs on Symfony.com. First, copy the composer require line and paste it into your terminal:

Tip

If you are on Symfony 3.2 or higher, you don't have to specify the bundle's version

```
$ composer require stof/doctrine-extensions-bundle:1.2
```

Second, plug the bundle into your AppKernel: copy the new bundle statement, open app/AppKernel.php and paste it here:

58 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 21
            new Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),
... lines 23 - 24
        );
... lines 26 - 35
    }
... lines 37 - 56
}
```

And finally, the bundle needs a little bit of configuration. But, the docs are kind of a bummer: it has a lot of not-so-important stuff near the top. It's like a treasure hunt! Hunt for a golden cold block near the bottom that shows some timestampable config.yml code. Copy this. Then, find our config.yml file and paste it at the bottom. And actually, the *only* thing we need is under the orm.default key: add sluggable: true:

81 lines | app/config/config.yml

```
... lines 1 - 75
stof_doctrine_extensions:
    default_locale: en_US
    orm:
        default:
            sluggable: true
```

This library adds *several* different magic behaviors to Doctrine, and sluggable - the automatic generation of a slug - is just one of them. And instead of turning on *all* the magic features by default, you need to activate the ones that you want. That's actually pretty nice. Another great behavior is Timestampable: an easy way to add createdAt and updatedAt fields to any entity.

The DoctrineExtensions Library

Head back to the documentation and scroll up. Near the top, find the link called DoctrineExtensions documentation and click it.

The truth is, StofDoctrineExtensionsBundle is just a small wrapper around this DoctrineExtensions library. And that means that *most* of the documentation also lives here. Open up the Sluggable documentation, and find the code example.

Adding the Sluggable Behavior

Ok cool, this is *easy*. Copy the Gedmo use statement above the entity: it's needed for the annotation we're about to add. Open Genus and paste it there:

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

```
... lines 1 - 7
use Gedmo\Mapping\Annotation as Gedmo;
... lines 9 - 168
```

Then, above the slug field, we'll add this @Gedmo\Slug annotation. Just change fields to simply name:

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

```
... lines 1 - 7
use Gedmo\Mapping\Annotation as Gedmo;
... lines 9 - 14
class Genus
{
... lines 17 - 29
    /**
     * @ORM\Column(type="string", unique=true)
     * @Gedmo\Slug(fields={"name"})
     */
    private $slug;
... lines 35 - 166
}
```

That is it! Now, when we *save* a Genus, the library will automatically generate a unique slug from the name. And that means *we* can be lazy and *never* worry about setting this field ourselves. Nice.

Reload the Fixtures

Head back to your terminal. Woh! My composer require blew up! But look closely: the library *did* install, but then it errored out when it tried to clear the cache. This is no big deal, and was just bad luck: I was *right* in the middle of adding the config.yml code when the cache cleared. If I run composer install, everything is happy.

Now, because our fixtures file sets the name property, we should just be able to reload our fixtures and watch the magic:

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

So far so good. Let's check the database. I'll use the doctrine:query:sql command:

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

Got it! The name is Balaena and the slug is the lower-cased version of that. Oh, and at the bottom, one of the slugs is trichechus-1. There are *two* genuses with this name. Fortunately, the Sluggable behavior guarantees that the slugs stay unique by adding -1, -2, -3 etc when it needs to.

So the slug magic is all done. Now we just need to update our app to use it in the URLs.

Chapter 3: Refactoring Carefully

Time to refactor our code to use the *slug* in the URLs. I'll close up a few files and then open GenusController. The "show" page we just saw in our browser comes from showAction(). And yep, it has {genusName} in the URL. Gross:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 64
    /**
     * @Route("/genus/{genusName}", name="genus_show")
     */
    public function showAction($genusName)
    {
... lines 70 - 92
    }
... lines 94 - 118
}
```

Change that to {slug}:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 64
    /**
     * @Route("/genus/{slug}", name="genus_show")
     */
... lines 68 - 111
}
```

And now, because slug is a property on the Genus entity, we *don't* need to manually query for it anymore. Instead, type-hint Genus as an argument:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 64
    /**
     * @Route("/genus/{slug}", name="genus_show")
     */
    public function showAction(Genus $genus)
    {
... lines 70 - 85
    }
... lines 87 - 111
}
```

Now, Symfony will do our job for us: I mean, query for the Genus automatically.

That means we can clean up a lot of this code. Just update the $genusName variable below to $genus->getName():

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 64
    /**
     * @Route("/genus/{slug}", name="genus_show")
     */
    public function showAction(Genus $genus)
    {
        $em = $this->getDoctrine()->getManager();

        $markdownTransformer = $this->get('app.markdown_transformer');
        $funFact = $markdownTransformer->parse($genus->getFunFact());

        $this->get('logger')
            ->info('Showing genus: '.$genus->getName());

        $recentNotes = $em->getRepository('AppBundle:GenusNote')
            ->findAllRecentNotesForGenus($genus);

        return $this->render('genus/show.html.twig', array(
            'genus' => $genus,
            'funFact' => $funFact,
            'recentNoteCount' => count($recentNotes)
        ));
    }
... lines 87 - 111
}
```

We just Broke our App!

Cool! Except, we just broke our app! By changing the wildcard from {genusName} to {slug}, we broke any code that generates a URL to this route. How can we figure out where those spots are?

My favorite way - because it's really safe - is to search the entire code base. In this case, we can search for the route name: genus_show. To do that, find your terminal and run:

```
$ git grep genus_show
```

Ok! We have 1 link in list.html.twig and we also generate a URL inside GenusController.

Search for the route in the controller. Ah, newAction() - which just holds some fake code we use for testing. Change the array key to slug set to $genus->getSlug():

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 18
    public function newAction()
    {
... lines 21 - 42
        return new Response(sprintf(
            '<html><body>Genus created! <a href="%s">%s</a></body></html>',
            $this->generateUrl('genus_show', ['slug' => $genus->getSlug()]),
            $genus->getName()
        ));
    }
... lines 49 - 111
}
```

Next, open app/Resources/views/genus/list.html.twig. Same change here: set slug to genus.slug:

27 lines | app/Resources/views/genus/list.html.twig

```
... lines 1 - 2
{% block body %}
    <table class="table table-striped">
... lines 5 - 11
        <tbody>
            {% for genus in genuses %}
                <tr>
                    <td>
                        <a href="{{ path('genus_show', {'slug': genus.slug}) }}">
                            {{ genus.name }}
                        </a>
                    </td>
... lines 20 - 21
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Project, un-broken!

There's *one* other page whose URL still uses name. In GenusController, find getNotesAction(). This is the AJAX endpoint that returns all of the notes for a specific Genus as JSON.

Change the URL to use {slug}:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 87
    /**
     * @Route("/genus/{slug}/notes", name="genus_show_notes")
     * @Method("GET")
     */
    public function getNotesAction(Genus $genus)
    {
... lines 94 - 110
    }
}
```

The automatic query will still work just like before. Now, repeat the careful searching we did before: copy the route name, find your terminal, and run:

```
$ git grep genus_show_notes
```

This is used in just *one* place. Open the genus/show.html.twig template. Change the path() argument to slug set to genus.slug:

42 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 25
{% block javascripts %}
... lines 27 - 32
    <script type="text/babel">
        var notesUrl = '{{ path('genus_show_notes', {'slug': genus.slug}) }}';
... lines 35 - 39
    </script>
{% endblock %}
```

That's it! That's everything. Go back to /genus in your browser and refresh. Now, click on Octopus. Check out that lowercase o on octopus in the URL. And since the notes are still displaying, it looks like the AJAX endpoint is working too.

So slugs are the *proper* way to do clean URLs, and they're really easy if you set them up from the beginning. You can also use {id} in your URLs - it just depends if you need them to look fancy or not.

Ok, let's get back to the point of this course: time to tackle - queue dramatic music - ManyToMany relations.

Chapter 4: ManyToMany Relationship

Let's talk about the famous, ManyToMany relationship. We already have a Genus entity and also a User entity. Before this tutorial, I updated the fixtures file. It still loads genuses, but it now loads *two* groups of users:

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

```
... lines 1 - 22
AppBundle\Entity\User:
    user_{1..10}:
        email: weaverryan+<current()>@gmail.com
        plainPassword: iliketurtles
        roles: ['ROLE_ADMIN']
        avatarUri: <imageUrl(100, 100, 'abstract')>
    user.aquanaut_{1..10}:
        email: aquanaut<current()>@example.org
        plainPassword: aquanote
        isScientist: true
        firstName: <firstName()>
        lastName: <lastName()>
        universityName: <company()> University
        avatarUri: <imageUrl(100, 100, 'abstract')>
```

The first group consists of normal users, but the second group has an isScientist boolean field set to true. In other words, our site will have many users, and some of those users happen to be scientists.

That's not really important for the relationship we're about to setup, the point is just that many users are scientists. And on the site, we want to keep track of which genuses are being studied by which scientists, or really, users. So, each User may study *many* genuses. And each Genus, may be studied by *many* Users.

This is a ManyToMany relationship. In a database, to link the genus table and user table, we'll need to add a new, *middle*, or *join* table, with genus_id and user_id foreign keys. That isn't a Doctrine thing, that's just how it's done.

Mapping a ManyToMany in Doctrine

So how do we setup this relationship in Doctrine? It's really nice! First, choose either entity: Genus or User, I don't care. I'll tell you soon why you might choose one over the other, but for now, it doesn't matter. Let's open Genus. Then, add a new private property: let's call it $genusScientists:

This could also be called users or anything else. The important thing is that it will hold the array of User objects that are linked to this Genus:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 74
    private $genusScientists;
... lines 76 - 172
}
```

Above, add the annotation: @ORM\ManyToMany with targetEntity="User".

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\ManyToMany(targetEntity="User")
     */
    private $genusScientists;
... lines 76 - 172
}
```

Doctrine ArrayCollection

Finally, whenever you have a Doctrine relationship where your property is an *array* of items, so, ManyToMany and OneToMany, you need to initialize that property in the __construct() method. Set $this->genusScientists to a new ArrayCollection():

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 76
    public function __construct()
    {
... line 79
        $this->genusScientists = new ArrayCollection();
    }
... lines 82 - 172
}
```

Creating the Join Table

Next... do nothing! Or maybe, high-five a stranger in celebration... because that is *all* you need. This is enough for Doctrine to create that middle, join table and start inserting and removing records for you.

It *can* be a bit confusing, because until now, *every* table in the database has needed a corresponding entity class. But the ManyToMany relationship is special. Doctrine says:

> You know what? I'm not going to require you to create an entity for that join table. Just map a ManyToMany relationship and I will create and manage that table for you.

That's freaking awesome! To prove it, go to your terminal, and run:

```
$ ./bin/console doctrine:schema:update --dump-sql
```

Boom! Thanks to that *one* little ManyToMany annotation, Doctrine now wants to create a genus_user table with genus_id and user_id foreign keys. Pretty dang cool.

JoinTable to control the... join table

But before we generate the migration for this, you can also control the name of that join table. Instead of genus_user, let's call ours genus_scientists - it's a bit more descriptive. To do that, add another annotation: @ORM\JoinTable. This optional annotation has just one job: to let you control how things are named in the database for this relationship. The most important is name="genus_scientist":

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\ManyToMany(targetEntity="User")
     * @ORM\JoinTable(name="genus_scientist")
     */
    private $genusScientists;
... lines 77 - 173
}
```

With that, find your terminal again and run:

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

Ok, go find and open that file!

37 lines | app/DoctrineMigrations/Version20160921164430.php

```
... lines 1 - 10
class Version20160921164430 extends AbstractMigration
{
... lines 13 - 15
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('CREATE TABLE genus_scientist (genus_id INT NOT NULL, user_id INT NOT NULL, INDEX IDX_66CF3FA885C4
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA885C4074C FOREIGN KEY (genus_id) REFERE
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA8A76ED395 FOREIGN KEY (user_id) REFEREN
    }
... lines 25 - 28
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysc

        $this->addSql('DROP TABLE genus_scientist');
    }
}
```

Woohoo!

Now it creates a genus_scientist table with those foreign keys. Execute the migration:

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

Guys: with about 5 lines of code, we just setup a ManyToMany relationship. Next question: how do we add stuff to it? Or, read from it?

Chapter 5: Inserting into a ManyToMany

The *big* question is: who is the best superhero of all time? Um, I mean, how can we *insert* things into this join table? How can we join a Genus and a User together?

Doctrine makes this *easy*... and yet... at the same time... kind of confusing! First, you need to completely forget that a join table exists. Stop thinking about the database! Stop it! Instead, your *only* job is to get a Genus object, put one or more User objects onto its genusScientists property and then save. Doctrine will handle the rest.

Setting Items on the Collection

Let's see this in action! Open up GenusController. Remember newAction()? This isn't a real page - it's just a route where *we* can play around and test out some code. And hey, it *already* creates and saves a Genus. Cool! Let's associate a user with it!

First, find a user with $user = $em->getRepository('AppBundle:User') then findOneBy() with email set to aquanaut1@example.org:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 18
    public function newAction()
    {
... lines 21 - 38
        $user = $em->getRepository('AppBundle:User')
            ->findOneBy(['email' => 'aquanaut1@example.org']);
... lines 41 - 51
    }
... lines 53 - 115
}
```

That'll work thanks to our handy-dandy fixtures file! We have scientists with emails aquanaut, 1-10@example.org:

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

```
... lines 1 - 22
AppBundle\Entity\User:
... lines 24 - 28
    user.aquanaut_{1..10}:
        email: aquanaut<current()>@example.org
... lines 31 - 37
```

We've got a User, we've got a Genus... so how can we smash them together? Well, in Genus, the genusScientists property is private. Add a new function so we can put stuff into it: public function: addGenusScientist() with a User argument:

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 174
175     public function addGenusScientist(User $user)
176     {
... line 177
178     }
179 }
```

Very simply, add that User to the $genusScientists property. Technically, that property is an ArrayCollection object, but we can treat it like an array:

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 174
175     public function addGenusScientist(User $user)
176     {
177         $this->genusScientists[] = $user;
178     }
179 }
```

Then back in the controller, call that: $genus->addGenusScientist() and pass it $user:

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

```
... lines 1 - 13
14  class GenusController extends Controller
15  {
... lines 16 - 18
19      public function newAction()
20      {
... lines 21 - 38
39          $user = $em->getRepository('AppBundle:User')
40              ->findOneBy(['email' => 'aquanaut1@example.org']);
41          $genus->addGenusScientist($user);
... lines 42 - 51
52      }
... lines 53 - 115
116 }
```

We're done! We don't even need to persist anything new, because we're already persisting the $genus down here.

Try it out! Manually go to /genus/new. Ok, genus Octopus15 created. Next, head to your terminal to query the join table. I'll use:

```
$ ./bin/console doctrine:query:sql "SELECT * FROM genus_scientist"
```

Oh yeah! The genus id 11 is now joined - by pure coincidence - to a user who is also id 11. This successfully joined the Octopus15 genus to the aquanaut1@example.org user.

If adding new items to a ManyToMany relationship is confusing... it's because Doctrine does all the work for you: add a User to your Genus, and just save. Don't over-think it!

Avoiding Duplicates

Let's do some experimenting! What if I duplicated the addGenusScientist() line?

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 18
    public function newAction()
    {
... lines 21 - 40
        $genus->addGenusScientist($user);
        $genus->addGenusScientist($user); // duplicate is ignored!
... lines 43 - 52
    }
... lines 54 - 116
}
```

Could this *one* new Genus be related to the same User *two* times? Let's find out!

Refresh the new page again. Alright! I love errors!

> Duplicate entry '12-11' for key 'PRIMARY'

So this is saying:

> Yo! You can't insert *two* rows into the genus_scientist table for the same genus and user.

And this is *totally* by design - it doesn't make sense to relate the same Genus and User multiple times. So that's great... but I *would* like to avoid this error in case this happens accidentally in the future.

To do that, we need to make our addGenusScientist() method a *little* bit smarter. Add if $this->genusScientists->contains()... remember, the $genusScientists property is actually an ArrayCollection object, so it has some trendy methods on it, like contains. Then pass $user. If genusScientists already has this User, just return:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 174
    public function addGenusScientist(User $user)
    {
        if ($this->genusScientists->contains($user)) {
            return;
        }

        $this->genusScientists[] = $user;
    }
}
```

Now when we go back and refresh, no problems. The genus_scientist table now holds the original entry we created and this *one* new entry: no duplicates for us.

Next mission: if I have a Genus, how can I get and print of all of its related Users? AND, what if I have a User, how can I get its related Genuses? This will take us down the magical - but dangerous - road of *inverse* relationships.

Chapter 6: Fetching Items from a ManyToMany Collection

New mission! On the genus show page, I want to list all of the users that are studying this Genus. If you think about the database - which I told you *NOT* to do, but ignore me for a second - then we want to query for all users that appear in the genus_scientist join table for this Genus.

Well, it turns out this query happens automagically, and the matching users are set into the $genusScientists property. Yea, Doctrine just does it! All *we* need to do is expose this private property with a getter: public function, getGenusScientists(), then return $this->genusScientists:

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 183
184     public function getGenusScientists()
185     {
186         return $this->genusScientists;
187     }
188 }
```

Now, open up the show.html.twig genus template and go straight to the bottom of the list. Let's add a header called Lead Scientists. Next, add a list-group, then start looping over the related users. What I mean is: for genusScientist in genus.genusScientists, then endfor:

57 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
5   {% block body %}
... lines 6 - 7
8       <div class="sea-creature-container">
9           <div class="genus-photo"></div>
10          <div class="genus-details">
11              <dl class="genus-details-list">
... lines 12 - 20
21                  <dt>Lead Scientists</dt>
22                  <dd>
23                      <ul class="list-group">
24                          {% for genusScientist in genus.genusScientists %}
... lines 25 - 31
32                          {% endfor %}
33                      </ul>
34                  </dd>
35              </dl>
36          </div>
37      </div>
38      <div id="js-notes-wrapper"></div>
39  {% endblock %}
... lines 40 - 57
```

The genusScientist variable will be a User object, because genusScientists is an *array* of users. In fact, let's advertise that above the getGenusScientists() method by adding @return ArrayCollection|User[]:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 183
    /**
     * @return ArrayCollection|User[]
     */
    public function getGenusScientists()
    {
        return $this->genusScientists;
    }
}
```

We know this *technically* returns an ArrayCollection, but we *also* know that if we loop over this, each item will be a User object. By adding the |User[], our editor will give us auto-completion when looping. And that, is pretty awesome.

Inside the loop, add an li with some styling:

57 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 7
    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 20
                <dt>Lead Scientists</dt>
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item">
... lines 26 - 30
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 40 - 57
```

Then add a link. Why a link? Because before this course, I created a handy-dandy user *show* page:

81 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 11
class UserController extends Controller
{
... lines 14 - 44
    /**
     * @Route("/users/{id}", name="user_show")
     */
    public function showAction(User $user)
    {
        return $this->render('user/show.html.twig', array(
            'user' => $user
        ));
    }
... lines 54 - 79
}
```

Copy the user_show route name, then use path(), paste the route, and pass it an id set to genusScientist.id, which *we* know is a User object. Then, genusScientist.fullName:

57 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 7
    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 20
                <dt>Lead Scientists</dt>
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item">
                                <a href="{{ path('user_show', {
                                    'id': genusScientist.id
                                }) }}">
                                    {{ genusScientist.fullName }}
                                </a>
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 40 - 57
```

Why fullName? If you look in the User class, I added a method called getFullName(), which puts the firstName and lastName together:

203 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface
{
... lines 18 - 197
    public function getFullName()
    {
        return trim($this->getFirstName().' '.$this->getLastName());
    }
}
```

It's really not that fancy.

Time for a test drive! When we refresh, we get the header, but this Genus doesn't have any scientists. Go back to /genus/new to create a more interesting Genus. Click the link to view it. Boom! How many queries did *we* need to write to make this work? None! That's right - we are keeping lazy.

But now, click to go check out the *user* show page. What if we want to do the *same* thing here? How can we list all of the *genuses* that are studied by this User? Time to setup the *inverse* side of this relationship!

Chapter 7: ManyToMany: The Inverse Side of the Relationship

Our goal is clear: list all of the genuses studied by this User.

The Owning vs Inverse Side of a Relation

Back in our Doctrine Relations tutorial, we learned that *every* relationship has two different sides: a mapping, or *owning* side, and an *inverse* side. In that course, we added a GenusNote entity and gave it a ManyToOne relationship to Genus:

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
}
```

This is the *owning* side, and it's the only one that we actually needed to create.

If you look in Genus, we also mapped the *other* side of this relationship: a OneToMany back to GenusNote:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 65
    /**
     * @ORM\OneToMany(targetEntity="GenusNote", mappedBy="genus")
     * @ORM\OrderBy({"createdAt" = "DESC"})
     */
    private $notes;
... lines 71 - 187
}
```

This is the *inverse* side of the relationship, and it's optional. When we mapped the *inverse* side, it caused *no* changes to our database structure. We added it *purely* for convenience, because we decided it sure would be fancy and nice if we could say $genus->getNotes() to automagically fetch all the GenusNotes for this Genus.

With a ManyToOne relationship, we don't choose which side is which: the ManyToOne side is *always* the required, *owning* side. And that makes sense, it's the table that holds the foreign key column, i.e. GenusNote has a genus_id column.

Owning and Inverse in ManyToMany

We can *also* look at our ManyToMany relationship in two different directions. If I have a Genus object, I can say:

> Hello fine sir: please give me all Users related to this Genus.

But if I have a User object, I should also be able to say the opposite:

> Good evening madame: I would like all Genuses related to this User.

The tricky thing about a ManyToMany relationship is that you get to *choose* which side is the *owning* side and which is the *inverse* side. And, I hate choices! The choice *does* have consequences.... but don't worry about that - we'll learn why soon.

Mapping the Inverse Side

Since we only have one side of the relationship mapped now, it's the *owning* side. To map the *inverse* side, open User and add a new property: $studiedGenuses. This will *also* be a ManyToMany with targetEntity set to Genus. But also add mappedBy="genusScientists:

223 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 77
    /**
     * @ORM\ManyToMany(targetEntity="Genus", mappedBy="genusScientists")
     */
    private $studiedGenuses;
... lines 82 - 221
}
```

That refers to the property inside of Genus:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\ManyToMany(targetEntity="User")
     * @ORM\JoinTable(name="genus_scientist")
     */
    private $genusScientists;
... lines 77 - 190
}
```

Now, on *that* property, add inversedBy="studiedGenuses, which points *back* to the property we just added in User:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\ManyToMany(targetEntity="User", inversedBy="studiedGenuses")
     * @ORM\JoinTable(name="genus_scientist")
     */
    private $genusScientists;
... lines 77 - 190
}
```

When you map *both* sides of a ManyToMany relationship, this mappedBy and inversedBy configuration is how you tell Doctrine which side is which. We don't *really* know why that's important yet, but we will soon.

Back in User, remember that whenever you have a relationship that holds a collection of objects, like a collection of "studied genuses", you need to add a __construct function and initialize that to a new ArrayCollection():

223 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 4
use Doctrine\Common\Collections\ArrayCollection;
... lines 6 - 16
class User implements UserInterface
{
... lines 19 - 82
    public function __construct()
    {
        $this->studiedGenuses = new ArrayCollection();
    }
... lines 87 - 221
}
```

Finally, since we'll want to be able to access these studiedGenuses, go to the bottom of User and add a new public function getStudiedGenuses(). Return that property inside. And of course, we love PHP doc, so add @return ArrayCollection|Genus[]:

223 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 4
use Doctrine\Common\Collections\ArrayCollection;
... lines 6 - 16
class User implements UserInterface
{
... lines 19 - 214
    /**
     * @return ArrayCollection|Genus[]
     */
    public function getStudiedGenuses()
    {
        return $this->studiedGenuses;
    }
}
```

Using the Inverse Side

And *just* by adding this new property, we are - as I *love* to say - dangerous.

Head into the user/show.html.twig template that renders the page we're looking at right now. Add a column on the right side of the page, a little "Genuses Studied" header, then a ul. To loop over all of the genuses that this user is studying, just say for genusStudied in user.studiedGenuses. Don't forget the endfor:

56 lines | app/Resources/views/user/show.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
... lines 6 - 38
            <div class="col-xs-4">
                <h3>Genus Studied</h3>
                <ul class="list-group">
                    {% for genusStudied in user.studiedGenuses %}
... lines 43 - 49
                    {% endfor %}
                </ul>
            </div>
        </div>
    </div>
{% endblock %}
```

Inside, add our favorite list-group-item and then a link. Link this *back* to the genus_show route, passing slug set to genusStudied.slug. Print out genusStudied.name:

56 lines | app/Resources/views/user/show.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
... lines 6 - 38
            <div class="col-xs-4">
                <h3>Genus Studied</h3>
                <ul class="list-group">
                    {% for genusStudied in user.studiedGenuses %}
                        <li class="list-group-item">
                            <a href="{{ path('genus_show', {
                                'slug': genusStudied.slug
                            }) }}">
                                {{ genusStudied.name }}
                            </a>
                        </li>
                    {% endfor %}
                </ul>
            </div>
        </div>
    </div>
{% endblock %}
```

But will it blend? I mean, will it work? Refresh!

Hot diggity dog! There are the *three* genuses that this User studies. We did nothing to deserve this nice treatment: Doctrine is doing all of the query work for us.

In fact, click the database icon on the web debug toolbar to see what the query looks like. When we access the property, Doctrine does a SELECT from genus with an INNER JOIN to genus_scientist where genus_scientist.user_id equals this User's id: 11. That's perfect! Thanks Obama!

Ordering the Collection

The *only* bummer is that we can't control the order of the genuses. What if we want to list them alphabetically? We can't - we

would instead need to make a custom query for the genuses in the controller, and pass them into the template.

What? Just kidding! In User, add another annotation: @ORM\OrderBy({"name" = "ASC"):

224 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 77
    /**
     * @ORM\ManyToMany(targetEntity="Genus", mappedBy="genusScientists")
     * @ORM\OrderBy({"name" = "ASC"})
     */
    private $studiedGenuses;
... lines 83 - 222
}
```

Refresh that!

If you didn't *see* a difference, you can double-check the query to prove it. Boom! There's our new ORDER BY. Later, I'll show you how you can mess with the query made for collections even more via Doctrine Criteria.

But up next, the last missing link: what if a User *stops* studying a Genus? How can we remove that link?

Chapter 8: Removing a ManyToMany Item

Back on the Genus page, I want to add a little "X" icon next to each user. When we click that, it will make an AJAX call that will remove the scientist from this Genus.

How a ManyToMany Link is Removed

To link a Genus and a User, we just added the User object to the genusScientists property:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 174
    public function addGenusScientist(User $user)
    {
        if ($this->genusScientists->contains($user)) {
            return;
        }

        $this->genusScientists[] = $user;
    }
... lines 183 - 190
}
```

So guess what? To *remove* that link and delete the row in the join table, we do the exact opposite: *remove* the User from the genusScientists property and save. Doctrine will notice that the User is missing from that collection and take care of the rest.

Setting up the Template

Let's start inside the the genus/show.html.twig template. Add a new link for each user: give some style classes, and a special js-remove-scientist-user class that we'll use in JavaScript. Add a cute close icon:

71 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 - 21
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item">
... lines 26 - 31
                                <a href="#"
                                   class="btn btn-link btn-xs pull-right js-remove-scientist-user"
                                >
                                    <span class="fa fa-close"></span>
                                </a>
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 46 - 71
```

Love it! Below, in the javascripts block, add a new script tag with a $(document).ready() function:

71 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 46
{% block javascripts %}
... lines 48 - 62
    <script>
        jQuery(document).ready(function() {
... lines 65 - 67
        });
    </script>
{% endblock %}
```

Inside, select the .js-remove-scientist-user elements, and on click, add the callback with our trusty e.preventDefault():

71 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 46
{% block javascripts %}
... lines 48 - 62
    <script>
        jQuery(document).ready(function() {
            $('.js-remove-scientist-user').on('click', function(e) {
                e.preventDefault();
            });
        });
    </script>
{% endblock %}
```

The Remove Endpoint Setup

Inside, we need to make an AJAX call back to our app. Let's go set that up. Open GenusController and find some space for a new method: public function removeGenusScientistAction(). Give it an @Route() set to /genus/{genusId}/scientist/{userId}:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 117
    /**
     * @Route("/genus/{genusId}/scientists/{userId}", name="genus_scientists_remove")
... line 120
     */
    public function removeGenusScientistAction($genusId, $userId)
    {
    }
}
```

You see, the *only* way for us to identify exactly *what* to remove is to pass both the genusId and the userId. Give the route a name like genus_scientist_remove. Then, add an @Method set to DELETE:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 117
    /**
     * @Route("/genus/{genusId}/scientists/{userId}", name="genus_scientists_remove")
     * @Method("DELETE")
     */
    public function removeGenusScientistAction($genusId, $userId)
    {
    }
}
```

You don't *have* to do that last part, but it's a good practice for AJAX, or API endpoints. It's very clear that making this request will delete something. Also, in the future, we could add another end point that has the *same* URL, but uses the GET method. That would *return* data about this link, instead of deleting it.

Any who, add the genusId and userId arguments on the method:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 121
    public function removeGenusScientistAction($genusId, $userId)
    {
    }
}
```

Next, grab the entity manager with $this->getDoctrine()->getManager() so we can fetch both objects:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 121
    public function removeGenusScientistAction($genusId, $userId)
    {
        $em = $this->getDoctrine()->getManager();
... lines 125 - 145
    }
}
```

Add $genus = $em->getRepository('AppBundle:Genus')->find($genusId):

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

```
... lines 1 - 123
        $em = $this->getDoctrine()->getManager();

        /** @var Genus $genus */
        $genus = $em->getRepository('AppBundle:Genus')
            ->find($genusId);
... lines 129 - 148
```

I'll add some inline doc to tell my editor this will be a Genus object. And of course, if !$genus, we need to throw $this->createNotFoundException(): "genus not found":

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

```
... lines 1 - 123
        $em = $this->getDoctrine()->getManager();

        /** @var Genus $genus */
        $genus = $em->getRepository('AppBundle:Genus')
            ->find($genusId);

        if (!$genus) {
            throw $this->createNotFoundException('genus not found');
        }
... lines 133 - 148
```

Copy *all* of that boring goodness, paste it, and change the variable to $genusScientist. This will query from the User entity using $userId. If we don't find a $genusScientist, say "genus scientist not found":

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

```
... lines 1 - 123
        $em = $this->getDoctrine()->getManager();

        /** @var Genus $genus */
        $genus = $em->getRepository('AppBundle:Genus')
            ->find($genusId);

        if (!$genus) {
            throw $this->createNotFoundException('genus not found');
        }

        $genusScientist = $em->getRepository('AppBundle:User')
            ->find($userId);

        if (!$genusScientist) {
            throw $this->createNotFoundException('scientist not found');
        }
... lines 140 - 148
```

Deleting the Link

Now *all* we need to do is remove the User from the Genus. We don't have a method to do that yet, so right below addGenusScientist(), make a new public function called removeGenusScientist() with a User argument:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 183
    public function removeGenusScientist(User $user)
    {
... line 186
    }
... lines 188 - 195
}
```

Inside, it's *so* simple: $this->genusScientists->removeElement($user):

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 183
    public function removeGenusScientist(User $user)
    {
        $this->genusScientists->removeElement($user);
    }
... lines 188 - 195
}
```

In other words, just remove the User from the array... by using a fancy convenience method on the collection. That doesn't touch the database yet: it just modifies the array.

Back in the controller, call $genus->removeGenusScientist() and pass that the user: $genusScientist:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 121
    public function removeGenusScientistAction($genusId, $userId)
    {
        $em = $this->getDoctrine()->getManager();

        /** @var Genus $genus */
        $genus = $em->getRepository('AppBundle:Genus')
            ->find($genusId);
... lines 129 - 133
        $genusScientist = $em->getRepository('AppBundle:User')
            ->find($userId);
... lines 136 - 140
        $genus->removeGenusScientist($genusScientist);
... lines 142 - 145
    }
}
```

We're done! Just persist the $genus and flush. Doctrine will take care of the rest:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 121
    public function removeGenusScientistAction($genusId, $userId)
    {
        $em = $this->getDoctrine()->getManager();

        /** @var Genus $genus */
        $genus = $em->getRepository('AppBundle:Genus')
            ->find($genusId);
... lines 129 - 133
        $genusScientist = $em->getRepository('AppBundle:User')
            ->find($userId);
... lines 136 - 140
        $genus->removeGenusScientist($genusScientist);
        $em->persist($genus);
        $em->flush();
... lines 144 - 145
    }
}
```

Returning from the Endpoint

At the bottom, we still need to return a Response. But, there's not really any information we need to send back to our JavaScript... so I'm going to return a new Response with null as the content and a 204 status code:

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

```
... lines 1 - 11
use Symfony\Component\HttpFoundation\Response;

class GenusController extends Controller
{
... lines 16 - 121
    public function removeGenusScientistAction($genusId, $userId)
    {
        $em = $this->getDoctrine()->getManager();

        /** @var Genus $genus */
        $genus = $em->getRepository('AppBundle:Genus')
            ->find($genusId);
... lines 129 - 133
        $genusScientist = $em->getRepository('AppBundle:User')
            ->find($userId);
... lines 136 - 140
        $genus->removeGenusScientist($genusScientist);
        $em->persist($genus);
        $em->flush();

        return new Response(null, 204);
    }
}
```

This is a nice way to return a response that is successful, but has no content. The 204 status code literally means "No Content".

Now, let's finish this by hooking up the frontend.

Chapter 9: Hooking up the Scientist Removal JavaScript

Endpoint, done! Let's call this bad boy from JavaScript. Back in the template, each delete link will have a different URL to the endpoint. Add a new attribute called data-url set to path('genus_scientist_remove') and pass it genusId set to genus.id and userId set to genusScientist.id. Remember, that's a User object:

91 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 - 21
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item js-scientist-item">
... lines 26 - 31
                                <a href="#"
                                   class="btn btn-link btn-xs pull-right js-remove-scientist-user"
                                   data-url="{{ path('genus_scientists_remove', {
                                       genusId: genus.id,
                                       userId: genusScientist.id
                                   }) }}"
                                >
                                    <span class="fa fa-close"></span>
                                </a>
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 50 - 91
```

Oh, and do one more thing: give the li above its own class: js-scientist-item:

91 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 - 21
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item js-scientist-item">
... lines 26 - 40
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 50 - 91
```

That'll also help in JavaScript.

Making the AJAX Call

Scroll back to the javascripts block. I'll paste a few lines of code here to get us started:

91 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 50
{% block javascripts %}
... lines 52 - 66
    <script>
        jQuery(document).ready(function() {
            $('.js-remove-scientist-user').on('click', function(e) {
                e.preventDefault();

                var $el = $(this).closest('.js-scientist-item');

                $(this).find('.fa-close')
                    .removeClass('fa-close')
                    .addClass('fa-spinner')
                    .addClass('fa-spin');
... lines 78 - 86
            });
        });
    </script>
{% endblock %}
```

Ok, no big deal: the first line uses $(this), which is the link that was just clicked, and finds the .js-scientist-item li that is around it. We'll use that in a minute. The second chunk changes the fa-close icon into a loading spinner... ya know... because we deserve fancy things.

The *real* work - the AJAX call - is up to us. I'll use $.ajax(). Set the url key to $(this).data('url') to read the attribute we just set. And then, set method to DELETE:

91 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 50
{% block javascripts %}
... lines 52 - 66
    <script>
        jQuery(document).ready(function() {
            $('.js-remove-scientist-user').on('click', function(e) {
                e.preventDefault();

                var $el = $(this).closest('.js-scientist-item');

                $(this).find('.fa-close')
                    .removeClass('fa-close')
                    .addClass('fa-spinner')
                    .addClass('fa-spin');

                $.ajax({
                    url: $(this).data('url'),
                    method: 'DELETE'
... lines 82 - 83
                });
... lines 85 - 86
            });
        });
    </script>
{% endblock %}
```

To add a little bit *more* fancy, add a .done(). After the AJAX call finishes, call $el.fadeOut() so that the item disappears in dramatic fashion:

91 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 50
{% block javascripts %}
... lines 52 - 66
    <script>
        jQuery(document).ready(function() {
            $('.js-remove-scientist-user').on('click', function(e) {
... lines 70 - 78
                $.ajax({
                    url: $(this).data('url'),
                    method: 'DELETE'
                }).done(function() {
                    $el.fadeOut();
                });
... lines 85 - 86
            });
        });
    </script>
{% endblock %}
```

Testing time! Refresh.

Cute close icon, check! Click it! It faded away in dramatic fashion! Yes!

Checking the Delete Query

Check out the web debug toolbar's AJAX icon. Mixed in with AJAX call for notes is our DELETE call. Click the little sha, then go to the Doctrine tab. Ooh, look at this:

```
DELETE FROM genus_scientist WHERE genus_id = 11 AND user_id = 11
```

Gosh darn it that's nice. To prove it, refresh: the scientist is gone. ManyToMany? Yea, it's as simple as adding and removing objects from an array.

Well, ok, it *will* get a bit harder soon...

Chapter 10: ManyToMany & Fixtures

Head back to /genus. These genuses are coming from our fixtures, but, sadly, the fixtures don't relate any scientists to them... yet. Let's fix that!

The fixtures.yml creates some Genus objects and some User objects, but nothing links them together:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
        name: <genus()>
        subFamily: '@subfamily_*'
        speciesCount: <numberBetween(100, 100000)>
        funFact: <sentence()>
        isPublished: <boolean(75)>
        firstDiscoveredAt: <dateTimeBetween('-200 years', 'now')>
... lines 9 - 22
AppBundle\Entity\User:
    user_{1..10}:
        email: weaverryan+<current()>@gmail.com
        plainPassword: iliketurtles
        roles: ['ROLE_ADMIN']
        avatarUri: <imageUrl(100, 100, 'abstract')>
    user.aquanaut_{1..10}:
        email: aquanaut<current()>@example.org
        plainPassword: aquanote
        isScientist: true
        firstName: <firstName()>
        lastName: <lastName()>
        universityName: <company()> University
        avatarUri: <imageUrl(100, 100, 'abstract')>
```

How can we do that? Well, remember, the fixtures system is very simple: it sets each value on the given property. It also has a super power where you can use the @ syntax to reference another object:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
... line 3
        subFamily: '@subfamily_*'
... lines 5 - 37
```

In that case, that other *object* is set on the property.

Setting data on our ManyToMany is no different: we need to take a Genus object and set an *array* of User objects on the genusScientists property. In other words, add a key called genusScientists set to [] - the array syntax in YAML. Inside, use @user.aquanaut_1. That refers to one of our User objects below. And whoops, make sure that's @user.aquanaut_1. Let's add another: @user.aquanaut_5:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
... lines 3 - 8
        genusScientists: ['@user.aquanaut_1', '@user.aquanaut_5']
... lines 10 - 38
```

It's not very random... but let's try it! Find your terminal and run:

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

Ok, check out the /genus page. Now *every* genus is related to the same two users.

Smart Fixtures: Using the Adder!

But wait... that should *not* have worked. The $genusScientists property - like *all* of these properties is *private*. To set them, the fixtures library uses the setter methods. But, um, we don't have a setGenusScientists() method, we only have addGenusScientist():

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 174
    public function addGenusScientist(User $user)
    {
        if ($this->genusScientists->contains($user)) {
            return;
        }

        $this->genusScientists[] = $user;
    }
... lines 183 - 195
}
```

So that's just another reason why the Alice fixtures library *rocks*. Because it says:

> Hey! I see an addGenusScientist() method! I'll just call that twice instead of looking for a setter.

Randomizing the Users

The only way this could be more hipster is if we could make these users random. Ah, but Alice has a trick for that too! Clear out the array syntax and instead, in quotes, say 3x @user.aquanaut_*:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
... lines 3 - 8
        genusScientists: '3x @user.aquanaut_*'
... lines 10 - 38
```

Check out that wonderful Alice syntax! It says: I want you to go find *three* random users, put them into an array, and *then* try to set them.

Reload those fixtures!

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

Then head over to your browser and refresh. Cool, three random scientists for each Genus. Pretty classy Alice, pretty classy.

Chapter 11: Joining Across a ManyToMany + EXTRA_LAZY Fetch

On the genus list page, I want to add a new column that prints the *number* of scientists each Genus has. That should be simple!

Open the genus/list.html.twig template. Add the new th for number of scientists:

29 lines | app/Resources/views/genus/list.html.twig

```
... lines 1 - 2
{% block body %}
    <table class="table table-striped">
        <thead>
            <tr>
... lines 7 - 8
                <th># of scientists</th>
... line 10
            </tr>
        </thead>
... lines 13 - 26
    </table>
{% endblock %}
```

Then down below, add the td, then say {{ genus.genusScientists|length }}:

29 lines | app/Resources/views/genus/list.html.twig

```
... lines 1 - 2
{% block body %}
    <table class="table table-striped">
... lines 5 - 12
        <tbody>
            {% for genus in genuses %}
                <tr>
... lines 16 - 21
                    <td>{{ genus.genusScientists|length }}</td>
... line 23
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

In other words:

> Go out and get my array of genus scientists and count them!

And, it even works! Each genus has three scientists. Until we delete one, then only *two* scientists! Yes!

The Lazy Collection Queries

But now click the Doctrine icon down in the web debug toolbar to see how the queries look on this page. This is really interesting: we have one query that's repeated many times: it selects *all* of the fields from user and then INNER JOINs over to genus_scientist WHERE genus_id equals 29, then, 25, 26 and 27.

When we query for the Genus, it does *not* automatically *also* go fetch all the related Users. Instead, at the moment that we access the genusScientists property, Doctrine queries all of the User data for that Genus. We're seeing that query for each row in the table.

Fetching EXTRA_LAZY

Technically, that's a lot of extra queries... which *could* impact performance. But please, don't hunt down *potential* performance problems too early - there are far too many good tools - like NewRelic and Blackfire - that are far better at identifying *real* performance issues later.

But, for the sake of learning... I want to do better, and there are a few possibilities! First, instead of querying for *all* the user data *just* so we can count the users, wouldn't it be better to make a super-fast COUNT query?

Yep! And there's an awesome way to do this. Open Genus and find the $genusScientists property. At the end of the ManyToMany, add fetch="EXTRA_LAZY":

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 71
72      /**
73       * @ORM\ManyToMany(targetEntity="User", inversedBy="studiedGenuses", fetch="EXTRA_LAZY")
74       * @ORM\JoinTable(name="genus_scientist")
75       */
76      private $genusScientists;
... lines 77 - 195
196 }
```

That's it. Now go back, refresh, and click to check out the queries. We still have the same *number* of queries, but each row's query is now just a simple count.

That's freaking awesome! Doctrine knows to do this because it realizes that all we're doing is *counting* the scientists. But, if we were to actually loop over the scientists and start accessing data on each User - like we do on the genus show page - then it would make a full query for all the User data. Doctrine is really smart.

Joining for Less Queries

Another way to optimize this would be to try to *minimize* the number of queries. Instead of running a query for every row, couldn't we grab *all* of this data at once? When we originally query for the genuses, what if we joined over to the user table *then*, and fetched all of the users immediately?

That's totally possible, and while it might actually be *slower* in this case, let's find out how to do join across a ManyToMany relationship. Open GenusController and find listAction(). Right now, this controller calls a findAllPublishOrderedByRecentlyActive() method on GenusRepository to make the query:

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

```
... lines 1 - 13
14  class GenusController extends Controller
15  {
... lines 16 - 57
58      public function listAction()
59      {
... lines 60 - 61
62          $genuses = $em->getRepository('AppBundle:Genus')
63              ->findAllPublishedOrderedByRecentlyActive();
... lines 64 - 67
68      }
... lines 69 - 146
147 }
```

Go find that method! Here's the goal: modify this query to join to the middle genus_scientist table and then join again to the user table so we can select all of the user data. But wait! What's the number one rule about ManyToMany relationships? That's right: you need to pretend like the middle join table doesn't exist.

Instead, leftJoin() directly to genus.genusScientists. Alias that to genusScientist:

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

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
... lines 10 - 12
    public function findAllPublishedOrderedByRecentlyActive()
    {
        return $this->createQueryBuilder('genus')
... lines 16 - 19
            ->leftJoin('genus.genusScientists', 'genusScientist')
... lines 21 - 23
    }
}
```

When you JOIN in Doctrine, you always join on a relation property, like $genusScientists. Doctrine will automatically take care of joining across the middle table and then over to the user table.

To select the user data: addSelect('genusScientist'):

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

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
... lines 10 - 12
    public function findAllPublishedOrderedByRecentlyActive()
    {
        return $this->createQueryBuilder('genus')
... lines 16 - 19
            ->leftJoin('genus.genusScientists', 'genusScientist')
            ->addSelect('genusScientist')
... lines 22 - 23
    }
}
```

Ok, go back and refresh again! Woh, *one* query! And that query contains a LEFT JOIN to genus_scientist and another to user. Because we're fetching *all* the user data in this query, Doctrine avoids making the COUNT queries later.

If Doctrine JOINS are still a bit new to you, give yourself a head start with our Doctrine Queries Tutorial.

Chapter 12: EntityType Checkboxes with ManyToMany

Guys, we are *really* good at adding items to our ManyToMany relationship in PHP and via the fixtures. But what about via Symfony's form system? Yea, that's where things get interesting.

Go to /admin/genus and login with a user from the fixtures: weaverryan+1@gmail.com and password iliketurtles. Click to edit one of the genuses.

Planning Out the Form

Right now, we don't have the ability to change which users are studying this genus from the form.

If we wanted that, how would it look? It would probably be a list of checkboxes: one checkbox for every user in the system. When the form loads, the already-related users would start checked.

This will be *perfect*... as long as you don't have a *ton* of users in your system. In that case, creating 10,000 checkboxes won't scale and we'll need a different solution. But, I'll save that for another day, and it's not really that different.

EntityType Field Configuration

The controller behind this page is called GenusAdminController and the form is called GenusFormType. Go find it! Step one: add a new field. Since we ultimately want to change the genusScientists property, that's what we should call the field. The type will be EntityType:

60 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 7
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 9 - 16
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 22 - 44
            ->add('genusScientists', EntityType::class, [
... lines 46 - 48
            ])
        ;
    }
... lines 52 - 58
}
```

This is your go-to field type whenever you're working on a field that is mapped as *any* of the Doctrine relations. We used it earlier with subfamily. In that case, each Genus has only *one* SubFamily, so we configured the field as a select *drop-down*:

60 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 7
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 9 - 16
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... line 22
            ->add('subFamily', EntityType::class, [
                'placeholder' => 'Choose a Sub Family',
                'class' => SubFamily::class,
                'query_builder' => function(SubFamilyRepository $repo) {
                    return $repo->createAlphabeticalQueryBuilder();
                }
            ])
... lines 30 - 49
        ;
    }
... lines 52 - 58
}
```

Back on genusScientists, start with the same setup: set class to User::class. Then, because this field holds an *array* of User objects, set multiple to true. Oh, and set expanded also to true: that changes this to render as checkboxes:

60 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 7
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 9 - 16
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 22 - 44
            ->add('genusScientists', EntityType::class, [
                'class' => User::class,
                'multiple' => true,
                'expanded' => true,
            ])
        ;
    }
... lines 52 - 58
}
```

That's everything! Head to the template: app/Resources/views/admin/genus/_form.html.twig. Head to the bottom and simply add the normal form_row(genusForm.genusScientists):

26 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 21
    {{ form_row(genusForm.genusScientists) }}
... lines 23 - 24
{{ form_end(genusForm) }}
```

Guys, let's go check it out.

Choosing the Choice Label

Refresh! And... explosion!

> Catchable Fatal Error: Object of class User could not be converted to string

Wah, wah. Our form is *trying* to build a checkbox for each User in the system... but it doesn't know what field in User it should use as the display value. So, it tries - and fails *epicly* - to cast the object to a string.

There's two ways to fix this, but I like to add a choice_label option. Set it to email to use that property as the visible text:

61 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 7
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 9 - 16
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 22 - 44
            ->add('genusScientists', EntityType::class, [
... lines 46 - 48
                'choice_label' => 'email',
            ])
        ;
    }
... lines 53 - 59
}
```

Try it again. Nice!

As expected, three of the users are pre-selected. So, does it save? Uncheck Aquanaut3, check Aquanaut2 and hit save. It does! Behind the scenes, Doctrine just deleted one row from the join table and inserted another.

EntityType: Customizing the Query

Our system really has *two* types of users: plain users and *scientists*:

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

```
... lines 1 - 23
AppBundle\Entity\User:
    user_{1..10}:
        email: weaverryan+<current()>@gmail.com
        plainPassword: iliketurtles
        roles: ['ROLE_ADMIN']
        avatarUri: <imageUrl(100, 100, 'abstract')>
    user.aquanaut_{1..10}:
        email: aquanaut<current()>@example.org
        plainPassword: aquanote
        isScientist: true
        firstName: <firstName()>
        lastName: <lastName()>
        universityName: <company()> University
        avatarUri: <imageUrl(100, 100, 'abstract')>
```

Well, they're really not any different, except that some have isScientist set to true. Now technically, I really want these

checkboxes to *only* list users that are scientists: normal users shouldn't be allowed to study Genuses.

How can we filter this list? Simple! Start by opening UserRepository: create a new public function called createIsScientistQueryBuilder():

16 lines | src/AppBundle/Repository/UserRepository.php

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

use Doctrine\ORM\EntityRepository;

class UserRepository extends EntityRepository
{
    public function createIsScientistQueryBuilder()
    {
... lines 11 - 13
    }
}
```

Very simple: return $this->createQueryBuilder('user'), andWhere('user.isScientist = :isScientist') and finally, setParameter('isScientist', true):

16 lines | src/AppBundle/Repository/UserRepository.php

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

use Doctrine\ORM\EntityRepository;

class UserRepository extends EntityRepository
{
    public function createIsScientistQueryBuilder()
    {
        return $this->createQueryBuilder('user')
            ->andWhere('user.isScientist = :isScientist')
            ->setParameter('isScientist', true);
    }
}
```

This doesn't make the query: it just returns the query builder.

Over in GenusFormType, hook this up: add a query_builder option set to an anonymous function. The field will pass us the UserRepository object. That's so thoughtful! That means we can celebrate with return $repo->createIsScientistQueryBuilder():

65 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 17
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 23 - 45
            ->add('genusScientists', EntityType::class, [
... lines 47 - 50
                'query_builder' => function(UserRepository $repo) {
                    return $repo->createIsScientistQueryBuilder();
                }
            ])
        ;
    }
... lines 57 - 63
}
```

Refresh that bad boy! Bam! User list filtered.

Thanks to our ManyToMany relationship, hooking up this field was easy: it just *works*. But now, let's go the *other* direction: find a user form, and add a list of genus checkboxes. That's where things are going to go a bit crazy.

Chapter 13: Saving the Inverse Side of a ManyToMany

Back on the main part of the site, click one of the genuses, and then click one of the users that studies it. This page has a little edit button: click that. Welcome to a very simple User form.

Building the Field

Ok, same plan: add checkboxes so that I can choose which genuses are being studied by this User. Open the controller: UserController and find editAction():

81 lines | src/AppBundle/Controller/UserController.php

```
... lines 1 - 5
use AppBundle\Form\UserEditForm;
... lines 7 - 11
class UserController extends Controller
{
... lines 14 - 57
    public function editAction(User $user, Request $request)
    {
        $form = $this->createForm(UserEditForm::class, $user);
... lines 61 - 78
    }
}
```

This uses UserEditForm, so go open that as well.

In buildForm(), we'll do the *exact* same thing we did on the genus form: add a new field called studiedGenuses - that's the property name on User that we want to modify:

41 lines | src/AppBundle/Form/UserEditForm.php

```
... lines 1 - 6
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 8 - 14
class UserEditForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 20 - 24
            ->add('studiedGenuses', EntityType::class, [
... lines 26 - 29
            ])
        ;
    }
... lines 33 - 39
}
```

Keep going: use EntityType::class and then set the options: class set now to Genus::class to make Genus checkboxes. Then, multiple set to true, expanded set to true, and choice_label set to name to display that field from Genus:

41 lines | src/AppBundle/Form/UserEditForm.php

```
... lines 1 - 6
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 8 - 14
class UserEditForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 20 - 24
            ->add('studiedGenuses', EntityType::class, [
                'class' => Genus::class,
                'multiple' => true,
                'expanded' => true,
                'choice_label' => 'name',
            ])
        ;
    }
... lines 33 - 39
}
```

Next! Open the template: user/edit.html.twig. At the bottom, use form_row(userForm.studiedGenuses):

25 lines | app/Resources/views/user/edit.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-xs-8">
... lines 7 - 8
                {{ form_start(userForm) }}
... lines 10 - 16
                    {{ form_row(userForm.studiedGenuses) }}
... lines 18 - 19
                {{ form_end(userForm) }}
            </div>
        </div>
    </div>
{% endblock %}
```

That's it.

Try it! Refresh! Cool! This User is studying five geniuses: good for them! Let's uncheck one genus, check a new one and hit Update.

It didn't Work!!! Inverse Relationships are Read-Only

Wait! It didn't work! The checkboxes just reverted back! What's going on!?

This is the moment where someone who doesn't know what we're about to learn, starts to *hate* Doctrine relations.

Earlier, we talked about how every relationship has two sides. You can start with a Genus and talk about the genus scientist users related to it:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\ManyToMany(targetEntity="User", inversedBy="studiedGenuses", fetch="EXTRA_LAZY")
     * @ORM\JoinTable(name="genus_scientist")
     */
    private $genusScientists;
... lines 77 - 195
}
```

Or, you can start with a User and talk about its studied genuses:

224 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 77
    /**
     * @ORM\ManyToMany(targetEntity="Genus", mappedBy="genusScientists")
     * @ORM\OrderBy({"name" = "ASC"})
     */
    private $studiedGenuses;
... lines 83 - 222
}
```

Only one of these side - in this case the Genus - is the *owning* side. So far, that hasn't meant anything: we can easily *read* data from either direction. BUT! The owning side has one special power: it is the *only* side that you're allowed to change.

What I mean is, if you have a User object and you add or remove genuses from its studiedGenuses property and save... Doctrine will do *nothing*. Those changes are completely ignored.

And it's not a bug! Doctrine is built this way on purpose. The data about which Genuses are linked to which Users is stored in *two* places. So Doctrine needs to choose *one* of them as the official source when it saves. It uses the *owning* side.

For a ManyToMany relationship, we chose the owning side when we set the mappedBy and inversedBy options. The owning side is also the only side that's allowed to have the @ORM\JoinTable annotation.

This is a *long* way of saying that if we want to update this relationship, we *must* add and remove users from the $genusScientists property on Genus:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\ManyToMany(targetEntity="User", inversedBy="studiedGenuses", fetch="EXTRA_LAZY")
     * @ORM\JoinTable(name="genus_scientist")
     */
    private $genusScientists;
... lines 77 - 195
}
```

Adding and removing genuses from the User object will do nothing. And that's exactly what our form just did.

No worries! We can fix this, with just a *little* bit of really smart code.

Chapter 14: Form by_reference + Adder and Remover

Head back to our form. We have a field called studiedGenuses:

41 lines | src/AppBundle/Form/UserEditForm.php

```
... lines 1 - 14
class UserEditForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 20 - 24
            ->add('studiedGenuses', EntityType::class, [
                'class' => Genus::class,
                'multiple' => true,
                'expanded' => true,
                'choice_label' => 'name',
            ])
        ;
    }
... lines 33 - 39
}
```

Because *all* of our properties are private, the form component works by calling the setter method for each field. I mean, when we submit, it takes the submitted email and calls setEmail() on User:

224 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 30
    private $email;
... lines 32 - 137
    public function setEmail($email)
    {
        $this->email = $email;
    }
... lines 142 - 222
}
```

But wait... we *do* have a field called studiedGenuses... but we do *not* have a setStudiedGenuses method:

224 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 81
    private $studiedGenuses;
... lines 83 - 222
}
```

Shouldn't the form component be throwing a huge error about that?

The by_reference Form Option

In theory... yes! But, the form is being *really* sneaky. Remember, the studiedGenuses property is an ArrayCollection object:

224 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 81
    private $studiedGenuses;

    public function __construct()
    {
        $this->studiedGenuses = new ArrayCollection();
    }
... lines 88 - 215
    /**
     * @return ArrayCollection|Genus[]
     */
    public function getStudiedGenuses()
    {
        return $this->studiedGenuses;
    }
}
```

When the form is building, it calls getStudiedGenuses() so that it knows which checkboxes to check. Then on submit, *instead* of trying to call a setter, it simply *modifies* that ArrayCollection. Basically, since ArrayCollection is an object, the form realizes it can be lazy: it adds and removes genuses directly from the object, but never sets it back on User. It doesn't need to, because the object is linked to the User by reference.

This *ultimately* means that our studiedGenuses property *is* being updated like we expected... just in a fancy way.

So... why should we care? We don't really... except that by *disabling* this fancy functionality, we will uncover a way to fix *all* of our problems.

How? Add a new option to the field: by_reference set to false:

42 lines | src/AppBundle/Form/UserEditForm.php

```
... lines 1 - 14
class UserEditForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 20 - 24
            ->add('studiedGenuses', EntityType::class, [
... lines 26 - 29
                'by_reference' => false,
            ])
        ;
    }
... lines 34 - 40
}
```

It says:

> Stop being fancy! Just call the setter method like normal!

Go refresh the form, and submit!

The Adder and Remover Methods

Ah! It's yelling at us! This is the error we expected all along:

> Neither the property studiedGenuses nor one of the methods - and then it lists a bunch of potential methods, including setStudiedGenuses() - exist and have public access in the User class.

In less boring terms, the form system is trying to say:

> Hey! I can't set the studiedGenuses back onto the User object unless you create one of these public methods!

So, should we create a setStudiedGenuses() method like it suggested? Actually, no. Another option is to create adder & remover methods.

Create a public function addStudiedGenus() with a Genus argument:

```
238 lines | src/AppBundle/Entity/User.php
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 223
    public function addStudiedGenus(Genus $genus)
    {
... lines 226 - 230
    }
... lines 232 - 236
}
```

Here, we'll do the same type of thing we did back in our Genus class: if $this->studiedGenuses->contains($genus), then do nothing. Otherwise $this->studiedGenuses[] = $genus:

```
238 lines | src/AppBundle/Entity/User.php
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 223
    public function addStudiedGenus(Genus $genus)
    {
        if ($this->studiedGenuses->contains($genus)) {
            return;
        }

        $this->studiedGenuses[] = $genus;
    }
... lines 232 - 236
}
```

After that, add the remover: public function removeStudiedGenus() also with a Genus argument. In here, say $this->studiedGenuses->removeElement($genus):

238 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 231

    public function removeStudiedGenus(Genus $genus)
    {
        $this->studiedGenuses->removeElement($genus);
    }
}
```

Perfect!

Go back to the form. Uncheck one of the genuses and check a new one. When we submit, it *should* call addStudiedGenus() once for the new checkbox and removeStudiedGenus() once for the box we unchecked.

Ok, hit update! Hmm, it *looked* successful... but it still didn't actually work. And that's expected! We just setup a cool little system where the form component calls our adder and remover methods to update the studiedGenuses property. But... this hasn't really changed anything: we're still not setting the *owning* side of the relationship.

But, we're just *one* small step from doing that.

Chapter 15: Synchronizing Owning & Inverse Sides

Ultimately, when we submit... we're still *only* updating the studiedGenuses property in User, which is the *inverse* side of this relationship:

238 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 77
    /**
     * @ORM\ManyToMany(targetEntity="Genus", mappedBy="genusScientists")
... line 80
     */
    private $studiedGenuses;
... lines 83 - 236
}
```

So, nothing actually saves.

How can we set the *owning* side? Why not just do it inside the adder and remover methods? At the bottom of addStudiedGenus(), add $genus->addGenusScientist($this):

240 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 223
    public function addStudiedGenus(Genus $genus)
    {
        if ($this->studiedGenuses->contains($genus)) {
            return;
        }

        $this->studiedGenuses[] = $genus;
        $genus->addGenusScientist($this);
    }
... lines 233 - 238
}
```

Booya, we just set the owning side of the relationship!

In removeStudiedGenus(), do the same thing: $genus->removeGenusScientist($this):

240 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 233
    public function removeStudiedGenus(Genus $genus)
    {
        $this->studiedGenuses->removeElement($genus);
        $genus->removeGenusScientist($this);
    }
}
```

So.... yea, that's all we need to do. Go back to the form, uncheck a genus, check a genus and hit update. It's alive!!!

We didn't need to add a lot of code to get this to work... but this situation has caused *many* developers to lose countless hours trying to get their relationship to save. To summarize: if you're modifying the inverse side of a relationship, set the by_reference form option to false, create an adder and remover function, and make sure you set the *owning* side of the relationship in each. That is it.

Synchronizing Both Sides

So, we're done! Well, technically we *are* done, but there is one last, tiny, teeny detail that I'd like to *perfect*. In Genus, when we call addGenusScientist(), it would be nice if we also updated this User to know that this Genus is now being studied by it. In other words, it would be nice if we called $user->addStudiedGenus($this):

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 174
    public function addGenusScientist(User $user)
    {
        if ($this->genusScientists->contains($user)) {
            return;
        }

        $this->genusScientists[] = $user;
        // not needed for persistence, just keeping both sides in sync
        $user->addStudiedGenus($this);
    }
... lines 185 - 199
}
```

I'm also going to add a note: this is *not* needed for persistence, but it might save you from an edge-case bug. Suppose we called $genus->addGenusScientist() to link a User to the Genus. Then later, during the *same* request - that's important - we have that same User object, and we call getStudiedGenuses(). We would want the Genus that was just linked to be in that collection. This does that! We're guaranteeing that both sides of the relationship stay synchronized.

Do the same thing down in the remover: $user->removeStudiedGenus($this):

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 185
    public function removeGenusScientist(User $user)
    {
        $this->genusScientists->removeElement($user);
        // not needed for persistence, just keeping both sides in sync
        $user->removeStudiedGenus($this);
    }
... lines 192 - 199
}
```

Have Fun and Avoid Infinite Recursion!

That's great! Oh, except for one thing I just introduced: **infinite recursion**! When we call removeStudiedGenus(), that calls removeGenusScientist(), which calls removeStudiedGenus(), and so on... forever. And we are too busy to let our scripts run forever!

The fix is easy - I was being lazy. Add an if statement in the remove functions, like if !$this->studiedGenuses->contains($genus), then just return:

244 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 233
    public function removeStudiedGenus(Genus $genus)
    {
        if (!$this->studiedGenuses->contains($genus)) {
            return;
        }

        $this->studiedGenuses->removeElement($genus);
        $genus->removeGenusScientist($this);
    }
}
```

In other words, if the $genus is not in the studiedGenuses array, there's no reason to try to remove it.

Inside Genus, do the exact same thing: if !$this->genusScientists->contains($user), then return:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 185
    public function removeGenusScientist(User $user)
    {
        if (!$this->genusScientists->contains($user)) {
            return;
        }

        $this->genusScientists->removeElement($user);
        // not needed for persistence, just keeping both sides in sync
        $user->removeStudiedGenus($this);
    }
... lines 196 - 203
}
```

Bye bye recursion.

Head back: uncheck a few genuses, check a few more and.... save! It works perfectly. We don't really notice this last perfection... but it may help us out in the future.

Chapter 16: ManyToMany with Extra Fields

Head back to /genus and click into one of our genuses. Thanks to our hard work, we can link genuses and users. So I know that Eda Farrell is a User that studies this Genus.

But, hmm, what if I need to store a little extra data on that relationship, like the number of *years* that each User has studied the Genus. Maybe Eda has studied this Genus for 10 years, but Marietta Schulist has studied it for only 5 years.

In the database, this means that we need our join table to have *three* fields now: genus_id, user_id, but also years_studied. How can we add that extra field to the join table?

The answer is simple, you can't! It's not possible. Whaaaaat?

You see, ManyToMany relationships only work when you have *no* extra fields on the relationship. But don't worry! That's by design! As soon as your join table need to have even *one* extra field on it, you need to build an entity class for it.

Creating the GenusScientist Join Entity

In your Entity directory, create a new class: GenusScientist. Open Genus and steal the ORM use statement on top, and paste it here:

71 lines | src/AppBundle/Entity/GenusScientist.php

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

use Doctrine\ORM\Mapping as ORM;
... lines 6 - 10
class GenusScientist
{
... lines 13 - 70
}
```

Next, add some properties: id - we could technically avoid this, but I like to give every entity an id - genus, user, and yearsStudied:

71 lines | src/AppBundle/Entity/GenusScientist.php

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

use Doctrine\ORM\Mapping as ORM;
... lines 6 - 10
class GenusScientist
{
... lines 13 - 17
    private $id;
... lines 19 - 23
    private $genus;
... lines 25 - 29
    private $user;
... lines 31 - 34
    private $yearsStudied;
... lines 36 - 70
}
```

Use the "Code"->"Generate" menu, or Command+N on a Mac, and select "ORM Class" to generate the class annotations:

71 lines | src/AppBundle/Entity/GenusScientist.php

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

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="genus_scientist")
 */
class GenusScientist
{
... lines 13 - 70
}
```

Oh, and notice! This generated a table name of genus_scientist: that's perfect! I want that to match our existing join table: we're going to migrate it to this new structure.

Go back to "Code"->"Generate" and this time select "ORM Annotation". Generate the annotations for id and yearsStudied:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;
... lines 19 - 31
    /**
     * @ORM\Column(type="string")
     */
    private $yearsStudied;
... lines 36 - 70
}
```

Perfect!

So how should we map the genus and user properties? Well, think about it: each is now a classic ManyToOne relationship. Every genus_scientist row should have a genus_id column and a user_id column. So, above genus, say ManyToOne with targetEntity set to Genus Below that, add the optional @JoinColumn with nullable=false:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
... lines 13 - 19
    /**
     * @ORM\ManyToOne(targetEntity="Genus")
     * @ORM\JoinColumn(nullable=false)
     */
    private $genus;
... lines 25 - 70
}
```

Copy that and put the same thing above user, changing the targetEntity to User:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
... lines 13 - 25
    /**
     * @ORM\ManyToOne(targetEntity="User")
     * @ORM\JoinColumn(nullable=false)
     */
    private $user;
... lines 31 - 70
}
```

And... that's it! Finish the class by going back to the "Code"->"Generate" menu, or Command+N on a Mac, selecting Getters and choosing id:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
... lines 13 - 36
    public function getId()
    {
        return $this->id;
    }
... lines 41 - 70
}
```

Do the same again for Getters and Setters: choose the rest of the properties:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
... lines 13 - 41
    public function getGenus()
    {
        return $this->genus;
    }

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

    public function getUser()
    {
        return $this->user;
    }

    public function setUser($user)
    {
        $this->user = $user;
    }

    public function getYearsStudied()
    {
        return $this->yearsStudied;
    }

    public function setYearsStudied($yearsStudied)
    {
        $this->yearsStudied = $yearsStudied;
    }
}
```

Entity, done!

Updating the Existing Relationships

Now that the join table has an entity, we need to update the relationships in Genus and User to point to it. In Genus, find the genusScientists property. Guess what? This is *not* a ManyToMany to User anymore: it's now a OneToMany to GenusScientist. Yep, it's now the *inverse* side of the ManyToOne relationship we just added. That means we need to change inversedBy to mappedBy set to genus. And of course, targetEntity is GenusScientist:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(targetEntity="GenusScientist", mappedBy="genus", fetch="EXTRA_LAZY")
     */
    private $genusScientists;
... lines 76 - 202
}
```

You *can* still keep the fetch="EXTRA_LAZY": that works for any relationship that holds an array of items. But, we *do* need to remove the JoinTable: annotation: both JoinTable and JoinColumn can only live on the *owning* side of a relationship.

There are more methods in this class - like addGenusScientist() that are now totally broken. But we'll fix them later. In GenusScientist, add inversedBy set to the genusScientists property on Genus:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
... lines 13 - 19
    /**
     * @ORM\ManyToOne(targetEntity="Genus", inversedBy="genusScientists")
     * @ORM\JoinColumn(nullable=false)
     */
    private $genus;
... lines 25 - 70
}
```

Finally, open User: we need to make the exact same changes here.

For studiedGenuses, the targetEntity is now GenusScientist, the relationship is OneToMany, and it's mappedBy the user property inside of GenusScientist:

243 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 77
    /**
     * @ORM\OneToMany(targetEntity="GenusScientist", mappedBy="user")
     */
    private $studiedGenuses;
... lines 82 - 241
}
```

The OrderBy doesn't work anymore. Well, technically it *does*, but we can only order by a field on GenusScientist, not on User. Remove that for now.

Tip

You should also add the inversedBy="studiedGenuses" to the user property in GenusScientist:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
... lines 13 - 25
    /**
     * @ORM\ManyToOne(targetEntity="User", inversedBy="studiedGenuses")
     * @ORM\JoinColumn(nullable=false)
     */
    private $user;
... lines 31 - 70
}
```

It didn't hurt anything, but I forgot that!

The Truth About ManyToMany

Woh! Ok! Step back for a second. Our ManyToMany relationship is now *entirely* gone: replaced by 3 entities and 2 classic ManyToOne relationships. And if you think about it, you'll realize that a ManyToMany relationship is nothing more than two ManyToOne relationships in disguise. All along, we could have mapped our original setup by creating a "join" GenusScientist entity with only genus and user ManyToOne fields. A ManyToMany relationship is just a convenience layer when that join table doesn't need any extra fields. But as soon as you *do* need extra, you'll need this setup.

Generating (and Fixing) the Migration

Last step: generate the migration:

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

Tip

If you get a

> There is no column with name id on table genus_scientist

error, this is due to a bug in doctrine/dbal 2.5.5. It's no big deal, as it just affects the *generation* of the migration file. There are 2 possible solutions until the bug is fixed:

1) Downgrade to doctrine/dbal 2.5.4. This would mean adding the following line to your composer.json file:

```
"doctrine/dbal": "2.5.4"
```

Then run composer update

2) Manually rename genus_scientist to something else (e.g. genus_scientist_old) and then generate the migration. Then, rename the table back. The generated migration will be *incorrect*, because it will think that you need to create a genus_scientist table, but we do not. So, you'll need to manually update the migration code by hand and test it.

Look in the app/DoctrineMigrations directory and open that migration:

48 lines | app/DoctrineMigrations/Version20161017160251.php

```
... lines 1 - 10
class Version20161017160251 extends AbstractMigration
{
... lines 13 - 15
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysq

        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA885C4074C');
        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA8A76ED395');
        $this->addSql('ALTER TABLE genus_scientist DROP PRIMARY KEY');
        $this->addSql('ALTER TABLE genus_scientist ADD id INT AUTO_INCREMENT NOT NULL, ADD years_studied VARCHAR(255)
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA885C4074C FOREIGN KEY (genus_id) REFERE
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA8A76ED395 FOREIGN KEY (user_id) REFEREN
        $this->addSql('ALTER TABLE genus_scientist ADD PRIMARY KEY (id)');
    }
... lines 29 - 32
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysq

        $this->addSql('ALTER TABLE genus_scientist MODIFY id INT NOT NULL');
        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA885C4074C');
        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA8A76ED395');
        $this->addSql('ALTER TABLE genus_scientist DROP PRIMARY KEY');
        $this->addSql('ALTER TABLE genus_scientist DROP id, DROP years_studied');
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA885C4074C FOREIGN KEY (genus_id) REFERE
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA8A76ED395 FOREIGN KEY (user_id) REFEREN
        $this->addSql('ALTER TABLE genus_scientist ADD PRIMARY KEY (genus_id, user_id)');
    }
}
```

So freakin' cool! Because we already have the genus_scientist join table, the migration does *not* create any new tables. Nope, it simply modifies it: drops a couple of foreign keys, adds the id and years_studied columns, and then re-adds the foreign keys. Really, the only thing that changed of importance is that we now have an id primary key, and a years_studied column. But otherwise, the table is still there, just the way it always was.

If you try to run this migration...it will blow up, with this rude error:

> Incorrect table definition; there can be only one auto column...

It turns out, Doctrine has a bug! Gasp! The horror! Yep, a bug in its MySQL code generation that affects this *exact* situation: converting a ManyToMany to a join entity. No worries: it's easy to fix... and I can't think of *any* other bug like this in Doctrine... and I use Doctrine *a lot*.

Take this last line: with ADD PRIMARY KEY id, copy it, remove that line, and then - after the id is added in the previous query - paste it and add a comma:

47 lines | app/DoctrineMigrations/Version20161017160251.php

```
... lines 1 - 10
class Version20161017160251 extends AbstractMigration
{
... lines 13 - 15
    public function up(Schema $schema)
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysq

        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA885C4074C');
        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA8A76ED395');
        $this->addSql('ALTER TABLE genus_scientist DROP PRIMARY KEY');
        $this->addSql('ALTER TABLE genus_scientist ADD id INT AUTO_INCREMENT NOT NULL, ADD PRIMARY KEY (id), ADD years_
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA885C4074C FOREIGN KEY (genus_id) REFERE
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA8A76ED395 FOREIGN KEY (user_id) REFEREN
    }
... lines 28 - 31
    public function down(Schema $schema)
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != 'mysql', 'Migration can only be executed safely on \'mysq

        $this->addSql('ALTER TABLE genus_scientist MODIFY id INT NOT NULL');
        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA885C4074C');
        $this->addSql('ALTER TABLE genus_scientist DROP FOREIGN KEY FK_66CF3FA8A76ED395');
        $this->addSql('ALTER TABLE genus_scientist DROP PRIMARY KEY');
        $this->addSql('ALTER TABLE genus_scientist DROP id, DROP years_studied');
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA885C4074C FOREIGN KEY (genus_id) REFERE
        $this->addSql('ALTER TABLE genus_scientist ADD CONSTRAINT FK_66CF3FA8A76ED395 FOREIGN KEY (user_id) REFEREN
        $this->addSql('ALTER TABLE genus_scientist ADD PRIMARY KEY (genus_id, user_id)');
    }
}
```

MySQL needs this to happen all in one statement.

But now, our migrations are in a *crazy* weird state, because this one *partially* ran. So let's start from scratch: drop the database fully, create the database, and then make sure all of our migrations can run from scratch:

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

Success!

Now that we have a different type of relationship, our app is broken! Yay! Let's fix it and update our forms to use the CollectionType.

Chapter 17: Join Entity App Refactoring

In some ways, not much just changed. Before, we had a genus_scientist table with genus_id and user_id columns. And... we still have that, just with two new columns:

71 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 10
class GenusScientist
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;
... lines 19 - 31
    /**
     * @ORM\Column(type="string")
     */
    private $yearsStudied;
... lines 36 - 70
}
```

But, in our app, a ton just changed. That's my nice way of saying: we just broke everything!

Collection of GenusScientists, not Users

For example, before, genusScientists was a collection of User objects, but now it's a collection of GenusScientist objects:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(targetEntity="GenusScientist", mappedBy="genus", fetch="EXTRA_LAZY")
     */
    private $genusScientists;
... lines 76 - 202
}
```

The same thing is true on User:

243 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 77
    /**
     * @ORM\OneToMany(targetEntity="GenusScientist", mappedBy="user")
     */
    private $studiedGenuses;
... lines 82 - 241
}
```

Wherever our code was using the studiedGenuses property - to get the collection or change it - well, that code is done broke.

Let's clean things up! And see some cool stuff along the way.

Creating new Join Entity Links

First, because we just emptied our database, we have no data. Open the fixtures file and temporarily comment-out the genusScientists property:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
... lines 3 - 8
        # genusScientists: '3x @user.aquanaut_*'
... lines 10 - 38
```

We can't simply set a User object on genusScientists anymore: this *now* accepts GenusScientist objects. We'll fix that in a second.

But, run the fixtures:

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

While that's working, go find GenusController and newAction(). Let's once again use this method to hack together and save some interesting data.

First, remove the two addGenusScientist lines:

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

```
... lines 1 - 13
class GenusController extends Controller
{
... lines 16 - 18
    public function newAction()
    {
... lines 21 - 40
        $genus->addGenusScientist($user);
        $genus->addGenusScientist($user); // duplicate is ignored!
... lines 43 - 52
    }
... lines 54 - 146
}
```

These don't make any sense anymore!

How can we add a new row to our join table? Just create a new entity: $genusScientist = new GenusScientist(). Then, set $genusScientist->setGenus($genus), $genusScientist->setUser($user) and $genusScientist->setYearsStudied(10). Don't

forget to $em->persist() this new entity:

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

```
... lines 1 - 6
use AppBundle\Entity\GenusScientist;
... lines 8 - 14
class GenusController extends Controller
{
... lines 17 - 19
    public function newAction()
    {
... lines 22 - 42
        $genusScientist = new GenusScientist();
        $genusScientist->setGenus($genus);
        $genusScientist->setUser($user);
        $genusScientist->setYearsStudied(10);
        $em->persist($genusScientist);
... lines 48 - 57
    }
... lines 59 - 151
}
```

There's nothing fancy going on anymore: GenusScientist is a normal, boring entity.

Using the new Collections

In your browser, try it: head to /genus/new. Genus created! Click the link to see it! Explosion! That's no surprise: our template code is looping over genusScientists and expecting a User object. Silly template! Let's fix that and the fixtures next.

Chapter 18: Using the new OneToMany Collections

Open up the genus/show.html.twig template. Actually, let's start in the Genus class itself. Find getGenusScientists():

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 195
196     /**
197      * @return ArrayCollection|User[]
198      */
199     public function getGenusScientists()
200     {
201         return $this->genusScientists;
202     }
203 }
```

This method is lying! It does not return an array of User objects, it returns an array of GenusScientist objects!

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 195
196     /**
197      * @return ArrayCollection|GenusScientist[]
198      */
199     public function getGenusScientists()
200     {
201         return $this->genusScientists;
202     }
203 }
```

In the template, when we loop over genus.genusScientists, genusScientist is *not* a User anymore. Update to genusScientist.user.fullName, and above, for the user_show route, change this to genusScientist.user.id:

92 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 7
    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 21
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item js-scientist-item">
                                <a href="{{ path('user_show', {
                                    'id': genusScientist.user.id
                                }) }}">
                                    {{ genusScientist.user.fullName }}
... line 30
                                </a>
... lines 32 - 41
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 51 - 92
```

Then, in the link, let's show off our new yearsStudied field: {{ genusScientist.yearsStudied }} then years:

92 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 7
    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 21
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item js-scientist-item">
                                <a href="{{ path('user_show', {
                                    'id': genusScientist.user.id
                                }) }}">
                                    {{ genusScientist.user.fullName }}
                                    ({{ genusScientist.yearsStudied }} years)
                                </a>
... lines 32 - 41
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 51 - 92
```

We still need to fix the remove link, but let's see how it looks so far!

Refresh! It's way less broken! Well, until you click to view the user!

Updating the User Template

To fix this, start by opening User and finding getStudiedGenuses(). Change the PHPDoc to advertise that this *now* returns an array of GenusScientist objects:

243 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 214
    /**
     * @return ArrayCollection|GenusScientist[]
     */
    public function getStudiedGenuses()
    {
        return $this->studiedGenuses;
    }
... lines 222 - 241
}
```

Next, go fix the template: user/show.html.twig. Hmm, let's rename this variable to be a bit more clear: genusScientist, to

match the type of object it is. Now, update slug to be genusScientist.genus.slug. And print genusScientist.genus.name:

56 lines | app/Resources/views/user/show.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
... lines 6 - 38
            <div class="col-xs-4">
                <h3>Genus Studied</h3>
                <ul class="list-group">
                    {% for genusScientist in user.studiedGenuses %}
                        <li class="list-group-item">
                            <a href="{{ path('genus_show', {
                                'slug': genusScientist.genus.slug
                            }) }}">
                                {{ genusScientist.genus.name }}
                            </a>
                        </li>
                    {% endfor %}
                </ul>
            </div>
        </div>
    </div>
{% endblock %}
```

Try it! Page is alive!

Updating the Delete Link

Back on the genus page, the other thing we need to fix is this remove link. In the show.html.twig template for genus, update the userId part of the URL: genusScientist.user.id:

92 lines | app/Resources/views/genus/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 7
    <div class="sea-creature-container">
        <div class="genus-photo"></div>
        <div class="genus-details">
            <dl class="genus-details-list">
... lines 12 - 21
                <dd>
                    <ul class="list-group">
                        {% for genusScientist in genus.genusScientists %}
                            <li class="list-group-item js-scientist-item">
... lines 26 - 32
                                <a href="#"
                                   class="btn btn-link btn-xs pull-right js-remove-scientist-user"
                                   data-url="{{ path('genus_scientists_remove', {
                                       genusId: genus.id,
                                       userId: genusScientist.user.id
                                   }) }}"
                                >
                                    <span class="fa fa-close"></span>
                                </a>
                            </li>
                        {% endfor %}
                    </ul>
                </dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 51 - 92
```

Next, find this endpoint in GenusController: removeGenusScientistAction():

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

```
... lines 1 - 14
class GenusController extends Controller
{
... lines 17 - 126
    public function removeGenusScientistAction($genusId, $userId)
    {
        $em = $this->getDoctrine()->getManager();

        /** @var Genus $genus */
        $genus = $em->getRepository('AppBundle:Genus')
            ->find($genusId);

        if (!$genus) {
            throw $this->createNotFoundException('genus not found');
        }

        $genusScientist = $em->getRepository('AppBundle:User')
            ->find($userId);

        if (!$genusScientist) {
            throw $this->createNotFoundException('scientist not found');
        }

        $genus->removeGenusScientist($genusScientist);
        $em->persist($genus);
        $em->flush();

        return new Response(null, 204);
    }
}
```

It's about to get *way* nicer. Kill the queries for Genus and User. Replace them with $genusScientist = $em->getRepository('AppBundle:GenusScientist') and findOneBy(), passing it user set to $userId and genus set to $genusId:

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

```
... lines 1 - 14
class GenusController extends Controller
{
... lines 17 - 126
    public function removeGenusScientistAction($genusId, $userId)
    {
        $em = $this->getDoctrine()->getManager();

        $genusScientist = $em->getRepository('AppBundle:GenusScientist')
            ->findOneBy([
                'user' => $userId,
                'genus' => $genusId
            ]);
... lines 136 - 140
    }
}
```

Then, instead of removing this link from Genus, we simply delete the entity: $em->remove($genusScientist):

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

```
... lines 1 - 14
class GenusController extends Controller
{
... lines 17 - 126
    public function removeGenusScientistAction($genusId, $userId)
    {
        $em = $this->getDoctrine()->getManager();

        $genusScientist = $em->getRepository('AppBundle:GenusScientist')
            ->findOneBy([
                'user' => $userId,
                'genus' => $genusId
            ]);

        $em->remove($genusScientist);
        $em->flush();

        return new Response(null, 204);
    }
}
```

And celebrate!

Go try it! Quick, delete that scientist! It disappears in dramatic fashion, *and*, when we refresh, it's *definitely* gone.

Phew! We're almost done. By the way, you can see that this refactoring takes some work. If you know that your join table will probably need extra fields on it, you can save yourself this work by setting up the join entity from the very beginning and avoiding ManyToMany. But, if you definitely won't have extra fields, ManyToMany is way nicer.

Updating the Fixtures

The *last* thing to fix is the fixtures. We won't set the genusScientists property up here anymore. Instead, scroll down and add a new AppBundle\Entity\GenusScientist section:

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

```
... lines 1 - 38
AppBundle\Entity\GenusScientist:
... lines 40 - 44
```

It's simple: we'll just build new GenusScientist objects ourselves, just like we did via newAction() in PHP code earlier. Add genus.scientist_{1..50} to create 50 links. Then, assign user to a random @user.aquanaut_* and genus to a random @genus_*. And hey, set yearsStudied to something random too: <numberBetween(1, 30)>:

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

```
... lines 1 - 38
AppBundle\Entity\GenusScientist:
    genus.scientist_{1..50}:
        user: '@user.aquanaut_*'
        genus: '@genus_*'
        yearsStudied: <numberBetween(1, 30)>
```

Nice! Go find your terminal and reload!

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

Ok, go back to /genus... and click one of them. We have scientists!

So our app is fixed, right? Well, not so fast. Go to /admin/genus: you might need to log back in - password iliketurtles. Our genus form is still *totally* broken. Ok, no error: but it doesn't even make sense anymore: our relationship is now more complex than checkboxes can handle. For example, how would I set the yearsStudied?

Time to take this form up a level.

Chapter 19: Embedded Form: CollectionType

Now that we've added the yearsStudied field to each GenusScientist, I'm not too sure that checkboxes make sense anymore. I mean, if I want to show that a User studies a Genus, I need to select a User, but I also need to tell the system how many *years* they have studied. How *should* this form look now?

Here's an idea, and one that works really well the form system: embed a collection of GenusScientist *subforms* at the bottom, one for each user that studies this Genus. Each subform will have a User drop-down and a "Years Studied" text box. We'll even add the ability to add or delete subforms via JavaScript, so that we can add or delete GenusScientist rows.

Creating the Embedded Sub-Form

Step one: we need to build a form class that represents *just* that little embedded GenusScientist form. Inside your Form directory, I'll press Command+N - but you can also right-click and go to "New" - and select "Form". Call it GenusScientistEmbeddedForm. Bah, remove that getName() method - that's not needed in modern versions of Symfony:

37 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 2
namespace AppBundle\Form;
... lines 4 - 8
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class GenusScientistEmbeddedForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 17 - 26
    }

    public function configureOptions(OptionsResolver $resolver)
    {
... lines 31 - 33
    }


}
```

Yay!

In configureOptions(), add $resolver->setDefaults() with the classic data_class set to GenusScientist::class:

37 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 4
use AppBundle\Entity\GenusScientist;
... lines 6 - 12
class GenusScientistEmbeddedForm extends AbstractType
{
... lines 15 - 28
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => GenusScientist::class
        ]);
    }
... lines 35 - 36
}
```

We *will* ultimately embed this form into our main genus form... but at this point... you can't tell: this form looks exactly like any other. And it will ultimately give us a GenusScientist object.

For the fields, we need two: user and yearsStudied:

37 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 12
class GenusScientistEmbeddedForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('user', EntityType::class, [
... lines 19 - 23
            ])
            ->add('yearsStudied')
        ;
    }
... lines 28 - 36
}
```

We do *not* need a genus dropdown field: instead, we'll automatically set that property to whatever Genus we're editing right now.

The user field should be an EntityType dropdown. In fact, let's go to GenusFormType and steal the options from the genusScientists field - it'll be *almost* identical. Set this to EntityType::class and then paste the options:

37 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 5
use AppBundle\Entity\User;
use AppBundle\Repository\UserRepository;
... lines 8 - 12
class GenusScientistEmbeddedForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('user', EntityType::class, [
                'class' => User::class,
                'choice_label' => 'email',
                'query_builder' => function(UserRepository $repo) {
                    return $repo->createIsScientistQueryBuilder();
                }
            ])
... line 25
        ;
    }
... lines 28 - 36
}
```

And make sure you re-type the last r in User and auto-complete it to get the use statement on top. Do the same for UserRepository. The only thing that's different is that this will be a drop-down for just *one* User, so remove the multiple and expanded options.

Embedding Using CollectionType

This form is now perfect. Time to embed! Remember, our goal is *still* to modify the genusScientists property on Genus, so our form field will *still* be called genusScientists. But clear out all of the options and set the type to CollectionType::class. Set its entry_type option to GenusScientistEmbeddedForm::class:

60 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 11
use Symfony\Component\Form\Extension\Core\Type\CollectionType;
... lines 13 - 18
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 24 - 46
            ->add('genusScientists', CollectionType::class, [
                'entry_type' => GenusScientistEmbeddedForm::class
            ])
        ;
    }
... lines 52 - 58
}
```

Before we talk about this, let's see what it looks like! Refresh!

Woh! This Genus is related to *four* GenusScientists... which you can see because it built an embedded form for each one! Awesome! Well, it's mostly ugly right now, but it works, and it's free!

Try updating one, like 26 to 27 and hit Save. It even saves!

Rendering the Collection... Better

But let's clean this up - because the form looks *awful*... even by my standards.

Open the template: app/Resources/views/admin/genus/_form.html.twig:

```
26 lines | app/Resources/views/admin/genus/_form.html.twig
{{ form_start(genusForm) }}
... lines 2 - 21
    {{ form_row(genusForm.genusScientists) }}
... lines 23 - 24
{{ form_end(genusForm) }}
```

This genusScientists field is *not* and actual field anymore: it's an *array* of fields. In fact, each of *those* field is *itself* composed of more sub-fields. What we have is a fairly complex form tree, which is something we talked about in our Form Theming Tutorial.

To render this in a more controlled way, delete the form_row. Then, add an h3 called "Scientists", a Bootstrap row, and then loop over the fields with for genusScientistForm in genusForm.genusScientists:

```
34 lines | app/Resources/views/admin/genus/_form.html.twig
{{ form_start(genusForm) }}
... lines 2 - 22
    <h3>Scientists</h3>
    <div class="row">
    {% for genusScientistForm in genusForm.genusScientists %}
... lines 26 - 28
    {% endfor %}
    </div>
... lines 31 - 32
{{ form_end(genusForm) }}
```

Yep, we're *looping* over each of those four embedded forms.

Add a column, and then call form_row(genusScientistForm) to print both the user and yearsStudied fields at once:

```
34 lines | app/Resources/views/admin/genus/_form.html.twig
{{ form_start(genusForm) }}
... lines 2 - 22
    <h3>Scientists</h3>
    <div class="row">
    {% for genusScientistForm in genusForm.genusScientists %}
        <div class="col-xs-4">
            {{ form_row(genusScientistForm) }}
        </div>
    {% endfor %}
    </div>
... lines 31 - 32
{{ form_end(genusForm) }}
```

So this should render the same thing as before, but with a bit more styling. Refresh! Ok, it's better... but what's up with those zero, one, two, three labels?

This genusScientistForm is actually an entire form full of several fields. So, it prints out a label for the entire form... which is zero, one, two, three, and four. That's not helpful!

Instead, print each field by hand. Start with form_errors(genusScientistForm), just in case there are any validation errors that are attached at this form level:

36 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 22
    <h3>Scientists</h3>
    <div class="row">
    {% for genusScientistForm in genusForm.genusScientists %}
        <div class="col-xs-4">
            {{ form_errors(genusScientistForm) }}
... lines 28 - 29
        </div>
    {% endfor %}
    </div>
... lines 33 - 34
{{ form_end(genusForm) }}
```

It's not common, but possible. Then, simply print form_row(genusScientistForm.user) and form_row(genusScientistForm.yearsStudied):

36 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 22
    <h3>Scientists</h3>
    <div class="row">
    {% for genusScientistForm in genusForm.genusScientists %}
        <div class="col-xs-4">
            {{ form_errors(genusScientistForm) }}
            {{ form_row(genusScientistForm.user) }}
            {{ form_row(genusScientistForm.yearsStudied) }}
        </div>
    {% endfor %}
    </div>
... lines 33 - 34
{{ form_end(genusForm) }}
```

Try it! Much better!

But you know what we *can't* do yet? We can't actually *remove* - or *add* - new scientists. all *we* can do is edit the existing ones. That's silly! So let's fix it!

Chapter 20: Collection Delete & allow_delete

Right now, this Genus is related to four GenusScientists. Cool... but what if one of those users *stopped* studying the Genus - how could we *remove* that one?

The Delete UI & JavaScript

Let's plan out the UI first: I want to be able click a little x icon next to each embedded form to make it disappear from the page. Then, when we submit, it should fully delete that GenusScientist record from the database. Cool?

Inside the embedded form, add a new class to the column: js-genus-scientist-item:

39 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
    ... lines 2 - 24
25      {% for genusScientistForm in genusForm.genusScientists %}
26          <div class="col-xs-4 js-genus-scientist-item">
    ... lines 27 - 32
33          </div>
34      {% endfor %}
    ... lines 35 - 37
38  {{ form_end(genusForm) }}
```

We'll use that in JavaScript in a second. Below that, add a little link with its own js-remove-scientist class... and put the cute little "x" icon inside:

39 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
    ... lines 2 - 24
25      {% for genusScientistForm in genusForm.genusScientists %}
26          <div class="col-xs-4 js-genus-scientist-item">
27              <a href="#" class="js-remove-scientist pull-right">
28                  <span class="fa fa-close"></span>
29              </a>
    ... lines 30 - 32
33          </div>
34      {% endfor %}
    ... lines 35 - 37
38  {{ form_end(genusForm) }}
```

Brilliant!

Time to hook up some JavaScript! Since this template is included by edit.html.twig and new.html.twig, I can't override the javascripts block from here. Instead, open edit.html.twig and override the block javascripts there:

32 lines | app/Resources/views/admin/genus/edit.html.twig

```
    ... lines 1 - 2
3   {% block javascripts %}
4       {{ parent() }}
    ... lines 5 - 18
19  {% endblock %}
    ... lines 20 - 32
```

We'll worry about adding JS to the new template later.

Start with the always-exciting document.ready function:

32 lines | app/Resources/views/admin/genus/edit.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script>
        jQuery(document).ready(function() {
... lines 8 - 16
        });
    </script>
{% endblock %}
... lines 20 - 32
```

Oh, but back in _form.html.twig, add one more class to the row that's around the entire section called js-genus-scientist-wrapper:

39 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 23
    <div class="row js-genus-scientist-wrapper">
... lines 25 - 34
    </div>
... lines 36 - 37
{{ form_end(genusForm) }}
```

Ok, back to the JavaScript! Add var $wrapper = then use jQuery to select that wrapper element. Register a listener on click for any .js-remove-scientist element - that's the delete link. Start that function with my favorite e.preventDefault():

32 lines | app/Resources/views/admin/genus/edit.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 5
    <script>
        jQuery(document).ready(function() {
            var $wrapper = $('.js-genus-scientist-wrapper');

            $wrapper.on('click', '.js-remove-scientist', function(e) {
                e.preventDefault();
... lines 12 - 15
            });
        });
    </script>
{% endblock %}
... lines 20 - 32
```

Then... what next? Well, forget about Symfony and the database: just find the .js-genus-scientist-item element that's around this link and... remove it!

32 lines | app/Resources/views/admin/genus/edit.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 5
    <script>
        jQuery(document).ready(function() {
            var $wrapper = $('.js-genus-scientist-wrapper');

            $wrapper.on('click', '.js-remove-scientist', function(e) {
                e.preventDefault();

                $(this).closest('.js-genus-scientist-item')
                    .fadeOut()
                    .remove();
            });
        });
    </script>
{% endblock %}
... lines 20 - 32
```

Simple! Refresh the page, click that "x", and be amazed.

Missing Fields: The allow_delete Option

But this is superficial: it didn't delete anything from the database nor can we submit the form and expect something to magically delete this GenusScientist, *just* because we removed it from the page. Or can we?

Submit! Well, I guess not. Huge error from the database!

> UPDATE genus_scientist SET years_studied and user_id to null.

Hmm. So our form is *not* expecting this embedded form to simply disappear. Instead, because the fields are missing from the submitted data, it thinks that we want to set that Genus Scientist's yearsStudied and user fields to null! No! I want to *delete* that entire object from the database!

How can we do that? First, in GenusFormType, we need to tell the genusScientists field that it's *ok* if one of the embedded form's fields is missing from the submit. Set a new allow_delete option to true:

61 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 18
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 24 - 46
            ->add('genusScientists', CollectionType::class, [
... line 48
                'allow_delete' => true,
            ])
        ;
    }
... lines 53 - 59
}
```

This tells the CollectionType that it's *ok* if one of the GenusScientist forms is missing when we submit. *And*, if a GenusScientist form is missing, it should remove that GenusScientist from the genusScientists array property. In other words, when we remove a GenusScientist form and submit, the final array will have *three* GenusScientist objects in it, instead of

four.

Ready? Submit!

Hmm, no error... but it still doesn't work. Why not? Hint: we already know the answer... and it relates to Doctrine's inverse relationships. Let's fix it.

Chapter 21: Deleting an Item from a Collection: orphanRemoval

When we delete one of the GenusScientist forms and submit, the CollectionType is now smart enough to *remove* that GenusScientist from the genusScientists array on Genus. So, why doesn't that make any difference to the database?

The problem is that the genusScientists property is now the *inverse* side of this relationship:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(targetEntity="GenusScientist", mappedBy="genus", fetch="EXTRA_LAZY")
     */
    private $genusScientists;
... lines 76 - 202
}
```

In other words, if we remove or add a GenusScientist from this array, it doesn't make any difference! Doctrine ignores changes to the inverse side.

Setting the Owning Side: by_reference

How to fix it? We already know how! We did it back with our ManyToMany relationship! It's a two step process.

First, in GenusFormType, set the by_reference option to false:

62 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 18
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 24 - 46
            ->add('genusScientists', CollectionType::class, [
... lines 48 - 49
                'by_reference' => false,
            ])
        ;
    }
... lines 54 - 60
}
```

Remember this?

Without this, the form component never calls setGenusScientists(). In fact, there *is* no setGenusScientists method in Genus. Instead, the form calls getGenusScientists() and then modifies that ArrayCollection object by reference:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 195
    /**
     * @return ArrayCollection|GenusScientist[]
     */
    public function getGenusScientists()
    {
        return $this->genusScientists;
    }
}
```

But by setting it to false, it's going to give us the flexibility we need to set the owning side of the relationship.

Setting the Owning Side: Adder & Remover

With *just* that change, submit the form. Error! But look at it closely: the error happens when the form system calls removeGenusScientist(). That's perfect! Well, not the error, but when we set by_reference to false, the form started using our adder and remover methods. *Now*, when we delete a GenusScientist form, it calls removeGenusScientist():

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 184
    public function removeGenusScientist(User $user)
    {
        if (!$this->genusScientists->contains($user)) {
            return;
        }

        $this->genusScientists->removeElement($user);
        // not needed for persistence, just keeping both sides in sync
        $user->removeStudiedGenus($this);
    }
... lines 195 - 202
}
```

The only problem is that those methods are *totally* outdated: they're still written for our old ManyToMany setup.

In removeGenusScientist(), change the argument to accept a GenusScientist object. Then update $user to $genusScientist in one spot, and then the other:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 184
    public function removeGenusScientist(GenusScientist $genusScientist)
    {
        if (!$this->genusScientists->contains($genusScientist)) {
            return;
        }

        $this->genusScientists->removeElement($genusScientist);
... lines 192 - 193
    }
... lines 195 - 202
}
```

For the last line, use $genusScientist->setGenus(null). Let's update the note to say the *opposite*:

> Needed to update the owning side of the relationship!

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 184
    public function removeGenusScientist(GenusScientist $genusScientist)
    {
        if (!$this->genusScientists->contains($genusScientist)) {
            return;
        }

        $this->genusScientists->removeElement($genusScientist);
        // needed to update the owning side of the relationship!
        $genusScientist->setGenus(null);
    }
... lines 195 - 202
}
```

Now, when we remove one of the embedded GenusScientist forms and submit, it will call removeGenusScientist() and that will set the owning side: $genusScientist->setGenus(null).

If you're a bit confused how this will ultimately *delete* that GenusScientist, hold on! Because you're right! But, submit the form again.

The Missing Link: orphanRemoval

Yay! Another error!

> UPDATE genus_scientist SET genus_id = NULL

Huh... that makes *perfect* sense. Our code is not *deleting* that GenusEntity. Nope, it's simply setting its genus property to null. This update query makes sense!

But... it's *not* what we want! We want to say:

> No no no. If the GenusScientist is no longer set to this Genus, it should be deleted entirely from the database.

And Doctrine has an option for *exactly* that. In Genus, find your genusScientists property. Let's reorganize the OneToMany

annotation onto multiple lines: it's getting a bit long. Then, add one magical option: orphanRemoval = true:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(
     *     targetEntity="GenusScientist",
     *     mappedBy="genus",
     *     fetch="EXTRA_LAZY",
     *     orphanRemoval=true
     * )
     */
    private $genusScientists;
... lines 81 - 207
}
```

That's the key. It says:

> If one of these GenusScientist objects suddenly has their genus set to null, just delete it entirely.

Tip

If the GenusScientist.genus property is set to a *different* Genus, instead of null, it will *still* be deleted. Use orphanRemoval only when that's not going to happen.

Give it a try! Refresh the form to start over. We have four genus scientists. Remove one and hit save.

Woohoo! That *fourth* GenusScientist was just deleted from the database.

I know this was a bit tricky, but we didn't write a lot of code to get here. There are just two things to remember.

First, if you're ever modifying the *inverse* side of a relationship in a form, set by_reference to false, create adder and remover methods, and set the *owning* side in each. And second, for a OneToMany relationship like this, use orphanRemoval to delete that related entity for you.

This was a *big* success! Next: we need to be able to add *new* genus scientists in the form.

Chapter 22: CollectionType: Adding New with the Prototype

So, how can we add a *new* scientist to a Genus?

Here's the plan: I want to add a button called "Add New Scientist", and when the user clicks it, it will render a new blank, embedded GenusScientist form. After the user fills in those fields and saves, we will insert a new record into the genus_scientist table.

The allow_add Option

Let's start with the front end first. Open GenusFormType. After the allow_delete option, put a new one: allow_add set to true:

63 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 18
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 24 - 46
            ->add('genusScientists', CollectionType::class, [
... lines 48 - 49
                'allow_add' => true,
                'by_reference' => false,
            ])
        ;
    }
... lines 55 - 61
}
```

Remember: allow_delete says:

> It's ok if one of the genus scientists' fields are missing from the submitted data.

And when one *is* missing, the form should remove it from the genusScientists array.

The allow_add option does the opposite:

> If there is suddenly an *extra* set of GenusScientist form data that's submitted, that's great!

In this case, it will create a *new* GenusScientist object and set it on the genusScientists array.

JavaScript Setup!

So, cool! Now open the _form.html.twig template. Add a link and give it a class: js-genus-scientist-add. Inside, give it a little icon - fa-plus-circle and say "Add Another Scientist":

46 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 23
    <div class="row js-genus-scientist-wrapper"
... lines 25 - 26
    >
... lines 28 - 37
        <a href="#" class="js-genus-scientist-add">
            <span class="fa fa-plus-circle"></span>
            Add Another Scientist
        </a>
    </div>
... lines 43 - 44
{{ form_end(genusForm) }}
```

Love it! Time to hook up the JavaScript: open edit.html.twig. Attach another listener to $wrapper: on click of the .js-genus-scientist-add link. Add the amazing e.preventDefault():

52 lines | app/Resources/views/admin/genus/edit.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script>
        jQuery(document).ready(function() {
            var $wrapper = $('.js-genus-scientist-wrapper');
... lines 9 - 17
            $wrapper.on('click', '.js-genus-scientist-add', function(e) {
                e.preventDefault();
... lines 20 - 35
            });
        });
    </script>
{% endblock %}
... lines 40 - 52
```

So... what exactly are we going to do in here? We somehow need to *clone* one of the embedded GenusScientist forms and insert a new, blank version onto the page.

Using... the prototype!

No worries! Symfony's CollectionType has a crazy thing to help us: the prototype.

Google for "Symfony form collection" and open the How to Embed a Collection of Forms document on Symfony.com. This page has some code that's ripe for stealing!

First, under the "Allowing New" section, find the template and copy the data-prototype attribute code. Open our form template, and add this to the wrapper div. Update the variable to genusForm.genusScientists.vars.prototype:

46 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 23
    <div class="row js-genus-scientist-wrapper"
        data-prototype="{{ form_widget(genusForm.genusScientists.vars.prototype)|e('html_attr') }}"
... line 26
    >
... lines 28 - 41
    </div>
... lines 43 - 44
{{ form_end(genusForm) }}
```

Oh, add one other thing while we're here: I promise I'll explain all of this in a minute: data-index set to genusForm.genusScientists|length:

46 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 23
    <div class="row js-genus-scientist-wrapper"
        data-prototype="{{ form_widget(genusForm.genusScientists.vars.prototype)|e('html_attr') }}"
        data-index="{{ genusForm.genusScientists|length }}"
    >
... lines 28 - 41
    </div>
... lines 43 - 44
{{ form_end(genusForm) }}
```

That will count the number of embedded forms that the form has right now.

Don't touch anything else: let's refresh the page to see what this looks like... because it's kind of crazy.

Wait, oh damn, I have three "Add New Scientist" links. Make sure your link is *outside* of the for loop. This link is great... but not so great that I want it three times. Oh, and fix the icon class too - get it together Ryan!

46 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 23
    <div class="row js-genus-scientist-wrapper"
        data-prototype="{{ form_widget(genusForm.genusScientists.vars.prototype)|e('html_attr') }}"
        data-index="{{ genusForm.genusScientists|length }}"
    >
    {% for genusScientistForm in genusForm.genusScientists %}
... lines 29 - 36
    {% endfor %}
        <a href="#" class="js-genus-scientist-add">
            <span class="fa fa-plus-circle"></span>
            Add Another Scientist
        </a>
    </div>
... lines 43 - 44
{{ form_end(genusForm) }}
```

Refresh again. Much better!

Checking out the prototype: __name__

View the HTML source and search for wrapper to find our js-genus-scientist-wrapper element. That big mess of characters is

the *prototype*. Yep, it looks *crazy*. This is a blank version of one of these embedded forms... after being escaped with HTML entities so that it can safely live in an attribute. This is *great*, because we can read this in JavaScript when the user clicks "Add New Scientist".

Oh, but check out this __name__ string: it shows up in a bunch of places inside the prototype. Scroll down a little to the embedded GenusScientist forms. If you look closely, you'll see that the fields in each of these forms have a different index *number*. The first is index zero, and it appears in a few places, like the name and id attributes. The next set of fields use one and then two.

When Symfony renders the prototype, instead of hard coding a number there - like zero, one or two - it uses __name__. It then expects *us* - in JavaScript - to change that to a unique index number, like three.

The Prototype JavaScript

Let's do it! Back on the Symfony documentation page: a lot of the JavaScript we need lives here. Find the addTagForm() function and copy the *inside* of it. Back in edit.html.twig, paste this inside our click function.

And let's make some changes. First, update $collectionHolder to $wrapper: that's the element that has the data-prototype attribute. We also read the data-index attribute... which is important because it tells us what number to use for the index. This is used to *replace* __name__ with that number. And then, each time we add another form, this index goes up by one.

Finally, at the very bottom: put this new sub-form onto the page: $(this) - which is the "Add another Scientist" link, $(this).before(newForm):

52 lines | app/Resources/views/admin/genus/edit.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script>
        jQuery(document).ready(function() {
            var $wrapper = $('.js-genus-scientist-wrapper');
... lines 9 - 17
            $wrapper.on('click', '.js-genus-scientist-add', function(e) {
                e.preventDefault();

                // Get the data-prototype explained earlier
                var prototype = $wrapper.data('prototype');

                // get the new index
                var index = $wrapper.data('index');

                // Replace '__name__' in the prototype's HTML to
                // instead be a number based on how many items we have
                var newForm = prototype.replace(/__name__/g, index);

                // increase the index with one for the next item
                $wrapper.data('index', index + 1);

                // Display the form in the page before the "new" link
                $(this).before(newForm);
            });
        });
    </script>
{% endblock %}
... lines 40 - 52
```

I think we are ready! Find your browser and refresh! Hold your breath: click "Add Another Scientist". It works! Well, the styling isn't quite right... but hey, this is a victory! And yea, we'll fix the styling later.

Add one new scientist, and hit save. Ah! It blows up! Obviously, we have a *little* bit more work to do.

Chapter 23: Adding to a Collection: Cascade Persist

After adding a new GenusScientist sub-form and submitting, we're greeted with this wonderful error!

> Expected argument of type User, GenusScientist given

Updating the Adder Method

But, like always, look closely. Because if you scroll down a little, you can see that the form is calling the addGenusScientist() method on our Genus object:

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 178
179     public function addGenusScientist(User $user)
180     {
... lines 181 - 187
188     }
... lines 189 - 207
208 }
```

Oh yea, we expected that! But, the code in this method is still outdated.

Change the argument to accept a GenusScientist object. Then, I'll refactor the variable name to $genusScientist:

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

```
... lines 1 - 14
15  class Genus
16  {
... lines 17 - 179
180     public function addGenusScientist(GenusScientist $genusScientist)
181     {
182         if ($this->genusScientists->contains($genusScientist)) {
183             return;
184         }
185
186         $this->genusScientists[] = $genusScientist;
187     }
... lines 188 - 206
207 }
```

As you guys know, we always need to set the *owning* side of the relationship in these methods. But, don't do that... yet. For now, *only* make sure that the new GenusScientist object is added to our array.

With that fixed, go back, and refresh to resubmit the form. Yay! New error! Ooh, this is an interesting one:

> A new entity was found through the relationship Genus.genusScientists that was not configured to cascade persist operations for GenusScientist.

Umm, what? Here's what's going on: when we persist the Genus, Doctrine sees the new GenusScientist on the genusScientists array... and notices that we have not called persist on *it*. This error basically says:

> Yo! You told me that you want to save this Genus, but it's related to a GenusScientist that you have *not* told me to save. You never called persist() on this GenusScientist! This doesn't make any sense!

Cascade Persist

So what's the fix? It's simple! We just need to call persist() on any new GenusScientist objects. We *could* add some code to our controller to do that after the form is submitted:

88 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 15
class GenusAdminController extends Controller
{
... lines 18 - 34
    public function newAction(Request $request)
    {
        $form = $this->createForm(GenusFormType::class);

        // only handles data on POST
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
... lines 42 - 43
            $em = $this->getDoctrine()->getManager();
            $em->persist($genus);
            $em->flush();
... lines 47 - 53
        }
... lines 55 - 58
    }
... lines 60 - 87
}
```

Or... we could do something fancier. In Genus, add a new option to the OneToMany: cascade={"persist"}:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(
     *     targetEntity="GenusScientist",
     *     mappedBy="genus",
     *     fetch="EXTRA_LAZY",
     *     orphanRemoval=true,
     *     cascade={"persist"}
     * )
     */
    private $genusScientists;
... lines 82 - 206
}
```

This says:

> When we persist a Genus, automatically call persist on each of the GenusScientist objects in this array. In other words, *cascade* the persist onto these children.

Alright, refresh now. This is the *last* error, I promise! And this makes perfect sense: it *is* trying to insert into genus_scientist - yay! But with genus_id set to null.

The GenusScientistEmbeddedForm creates a new GenusScientist object and sets the user and yearsStudied fields:

37 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 12
class GenusScientistEmbeddedForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('user', EntityType::class, [
... lines 19 - 23
            ])
            ->add('yearsStudied')
        ;
    }
... lines 28 - 36
}
```

But, nobody is ever setting the genus property on this GenusScientist.

This is because I forced you - against your will - to temporarily *not* set the owning side of the relationship in addGenusScientist. I'll copy the same comment from the remover, and then add $genusScientist->setGenus($this):

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 179
    public function addGenusScientist(GenusScientist $genusScientist)
    {
... lines 182 - 186
        // needed to update the owning side of the relationship!
        $genusScientist->setGenus($this);
    }
... lines 190 - 208
}
```

Owning side handled!

Ok, refresh *one* last time. Boom! We now have *four* genuses: this new one was just inserted.

And yea, that's about as complicated as you can get with this stuff.

Don't Purposefully Make your Life Difficult

Oh, but before we move on, go back to /genus, click a genus, go to one of the user show pages, and then click the pencil icon. *This* form is still *totally* broken: it's still built as if we have a ManyToMany relationship to Genus. But with our new-found knowledge, we could easily fix this in the exact same way that we just rebuilt the GenusForm. But, since that's not too interesting, instead, open UserEditForm and remove the studiedGenuses field:

42 lines | src/AppBundle/Form/UserEditForm.php

```
... lines 1 - 14
class UserEditForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 20 - 24
            ->add('studiedGenuses', EntityType::class, [
                'class' => Genus::class,
                'multiple' => true,
                'expanded' => true,
                'choice_label' => 'name',
                'by_reference' => false,
            ])
        ;
    }
... lines 34 - 40
}
```

Then, open the user/edit.html.twig template and kill the render:

25 lines | app/Resources/views/user/edit.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-xs-8">
... lines 7 - 8
                {{ form_start(userForm) }}
... lines 10 - 16
                    {{ form_row(userForm.studiedGenuses) }}
... lines 18 - 19
                {{ form_end(userForm) }}
            </div>
        </div>
    </div>
{% endblock %}
```

Finally, find the User class and scroll down to the adder and remover methods. Get these outta here:

243 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 222
    public function addStudiedGenus(Genus $genus)
    {
        if ($this->studiedGenuses->contains($genus)) {
            return;
        }

        $this->studiedGenuses[] = $genus;
        $genus->addGenusScientist($this);
    }

    public function removeStudiedGenus(Genus $genus)
    {
        if (!$this->studiedGenuses->contains($genus)) {
            return;
        }

        $this->studiedGenuses->removeElement($genus);
        $genus->removeGenusScientist($this);
    }
}
```

Go back to refresh the form. Ok, better! This last task was more than just some cleanup: it illustrates an important point. If you don't need to edit the genusesStudied from this form, then you don't need all the extra code, especially the adder and remover methods. Don't make yourself do extra work. At first, whenever I map the *inverse* side of a relationship, I *only* add a "getter" method. It's only later, *if* I need to update things from this side, that I get fancy.

Oh, and also, remember that this entire *side* of the relationship is *optional*. The *owning* side of the relationship is in GenusScientist. So unless you need to be able to easily fetch the GenusScientist instances for a User - in other words, $user->getStudiedGenuses() - don't even bother mapping this side. *We* are using that functionality on the user show page, so I'll leave it.

Chapter 24: Embedded Form Validation with @Valid

We've got more work to do! So head back to /admin/genus. Leave the "Years Studied" field empty for one of the GenusScientist forms and submit.

Explosion!

```
UPDATE genus_scientist SET years_studied = NULL
```

This field is not allowed to be null in the database. That's on purpose... but we're missing validation! Lame!

But no problem, right? We'll just go into the Genus class, copy the as Assert use statement, paste it into GenusScientist and then - above yearsStudied - add @Assert\NotBlank:

73 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 5
use Symfony\Component\Validator\Constraints as Assert;
... lines 7 - 11
class GenusScientist
{
... lines 14 - 32
    /**
... line 34
     * @Assert\NotBlank()
     */
    private $yearsStudied;
... lines 38 - 72
}
```

Cool! Now, the yearsStudied field will be required.

Go try it out: refresh the page, empty out the field again, submit and... What!? It still doesn't work!?

@Valid for a Good Time

It's as if Symfony doesn't see the new validation constraint! Why? Here's the deal: our form is bound to a Genus object:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(
     *     targetEntity="GenusScientist",
     *     mappedBy="genus",
     *     fetch="EXTRA_LAZY",
     *     orphanRemoval=true,
     *     cascade={"persist"}
     * )
     */
    private $genusScientists;
... lines 82 - 208
}
```

That's the top-level object that we're modifying. And by default, Symfony reads all of the validation annotations from the top-level class... only. When it sees an embedded object, or an array of embedded objects, like the genusScientists property, it does *not* go deeper and read the annotations from the GenusScientist class. In other words, Symfony *only* validates the top-level object.

Double-lame! What the heck Symfony?

No no, it's cool, it's on purpose. You can easily *activate* embedded validation by adding a unique annotation above that property: @Assert\Valid:

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

```
... lines 1 - 6
use Symfony\Component\Validator\Constraints as Assert;
... lines 8 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(
     *     targetEntity="GenusScientist",
     *     mappedBy="genus",
     *     fetch="EXTRA_LAZY",
     *     orphanRemoval=true,
     *     cascade={"persist"}
     * )
     * @Assert\Valid()
     */
    private $genusScientists;
... lines 83 - 209
}
```

That's it! Now refresh. Validation achieved!

Preventing Duplicate GenusScientist

But there's *one* other problem. I know, I always have bad news. Set one of the users to aquanaut3. Well, that's actually a duplicate of this one... and it doesn't really make sense to have the same user listed as two different scientists. Whatever! Save right now: it's all good: aquanaut3 and aquanaut3. I want validation to prevent this!

No problem! In GenusScientist add a new annotation above the *class*: yep, a rare constraint that goes above the class instead of a property: @UniqueEntity. Make sure to auto-complete that to get a special use statement for this:

78 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 5
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 7 - 8
/**
... lines 10 - 11
 * @UniqueEntity(
... lines 13 - 14
 * )
 */
class GenusScientist
{
... lines 19 - 77
}
```

This takes a few options, like fields={"genus", "user"}:

78 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 5
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 7 - 8
/**
... lines 10 - 11
 * @UniqueEntity(
 *     fields={"genus", "user"},
... line 14
 * )
 */
class GenusScientist
{
... lines 19 - 77
}
```

This says:

> Don't allow there to be two records in the database that have the same genus and user.

Add a nice message, like:

> This user is already studying this genus.

78 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 5
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 7 - 8
/**
... lines 10 - 11
 * @UniqueEntity(
 *     fields={"genus", "user"},
 *     message="This user is already studying this genus"
 * )
 */
class GenusScientist
{
... lines 19 - 77
}
```

Great!

Ok, try this bad boy! We already have duplicates, so just hit save. Validation error achieved! But... huh... there are *two* errors and they're listed at the *top* of the form, instead of next to the offending fields.

First, ignore the *two* messages - that's simply because we allowed our app to get into an invalid state and *then* added validation. That confused Symfony. Sorry! You'll normally only see one message.

But, having the error message way up on top... that sucks! The reason why this happens is honestly a little bit complex: it has to do with the CollectionType and something called error_bubbling. The more important thing is the fix: after the message option, add another called errorPath set to user:

79 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 5
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 7 - 8
/**
... lines 10 - 11
 * @UniqueEntity(
 *     fields={"genus", "user"},
 *     message="This user is already studying this genus",
 *     errorPath="user"
 * )
 */
class GenusScientist
{
... lines 20 - 78
}
```

In a *non* embedded form, the validation error message from UniqueEntity normally shows at the top of the form... which makes a lot of sense in that situation. But when you add this option, it says:

> Yo! When this error occurs, I want you to attach it to the user field.

So refresh! Error is in place! And actually, let me get us *out* of the invalid state: I want to reset my database to *not* have any duplicates to start. *Now* if we change one back to a duplicate, it looks great... and we don't have *two* errors anymore.

Fixing CollectionType Validation Bug

There is one small bug left with our validation! And it's tricky! To see it: add 2 new scientists, immediately remove the first, leave the yearsStudied field blank, and then submit. We *should* see a validation error appearing below the yearsStudied field. Instead, it appears no the top of the form! This is actually caused by a bug in Symfony, but we can fix it easily! The following code block shows the fix and has more details:

108 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 14
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
... lines 17 - 20
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 25 - 56
        $builder->addEventListener(FormEvents::PRE_SUBMIT, array($this, 'onPreSubmit'));
    }
... lines 59 - 66
    /**
     * This fixes a validation issue with the Collection. Suppose
     * the following situation:
     *
     * A) Edit a Genus
     * B) Add 2 new scientists - don't submit & leave all fields blank
     * C) Delete the FIRST scientist
     * D) Submit the form
     *
     * The one new scientist has a validation error, because
     * the yearsStudied field was left blank. But, this error
     * shows at the *top* of the form, not attached to the form.
     * The reason is that, on submit, addGenusScientist() is
     * called, and the new scientist is added to the next available
     * index (so, if the Genus previously had 2 scientists, the
     * new GenusScientist is added to the "2" index). However,
     * in the HTML before the form was submitted, the index used
     * in the name attribute of the fields for the new scientist
     * was *3*: 0 & 1 were used for the existing scientists and 2 was
     * used for the first genus scientist form that you added
     * (and then later deleted). This mis-match confuses the validator,
     * which thinks there is an error on genusScientists[2].yearsStudied,
     * and fails to map that to the genusScientists[3].yearsStudied
     * field.
     *
     * Phew! It's a big pain :). Below, we fix it! On submit,
     * we simply re-index the submitted data before it's bound
     * to the form. The submitted genusScientists data, which
     * previously had index 0, 1 and 3, will now have indexes
     * 0, 1 and 2. And these indexes will match the indexes
     * that they have on the Genus.genusScientists property.
     *
     * @param FormEvent $event
     */
    public function onPreSubmit(FormEvent $event)
    {
        $data = $event->getData();
        $data['genusScientists'] = array_values($data['genusScientists']);
        $event->setData($data);
    }
}
```

Chapter 25: Customizing the Collection Form Prototype

There's still one *ugly* problem with our form, and I promised we would fix: when we click "Add Another Scientist"... well, it don't look right!. The new form should have the exact same styling as the existing ones.

Customizing the Prototype!

Why does it look different, anyways? Remember the data-prototype attribute?

46 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
    ... lines 2 - 23
24      <div class="row js-genus-scientist-wrapper"
25          data-prototype="{{ form_widget(genusForm.genusScientists.vars.prototype)|e('html_attr') }}"
    ... line 26
27      >
    ... lines 28 - 41
42      </div>
    ... lines 43 - 44
45  {{ form_end(genusForm) }}
```

By calling form_widget, this renders a blank GenusScientist form... by using the *default* Symfony styling. But when we render the *existing* embedded forms, we wrap them in all kinds of cool markup:

46 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
    ... lines 2 - 27
28      {% for genusScientistForm in genusForm.genusScientists %}
29          <div class="col-xs-4 js-genus-scientist-item">
30              <a href="#" class="js-remove-scientist pull-right">
31                  <span class="fa fa-close"></span>
32              </a>
33              {{ form_errors(genusScientistForm) }}
34              {{ form_row(genusScientistForm.user) }}
35              {{ form_row(genusScientistForm.yearsStudied) }}
36          </div>
37      {% endfor %}
    ... lines 38 - 44
45  {{ form_end(genusForm) }}
```

What we *really* want is to somehow make the data-prototype attribute use the markup that we wrote inside the for statement.

How? Well, there are at least two ways of doing it, and I'm going to show you the less-official and - in my opinion - easier way!

Head to the top of the file and add a *macro* called printGenusScientistRow() that accepts a genusScientistForm argument:

52 lines | app/Resources/views/admin/genus/_form.html.twig

```
    ... lines 1 - 2
3   {% macro printGenusScientistRow(genusScientistForm) %}
    ... lines 4 - 11
12  {% endmacro %}
    ... lines 13 - 52
```

If you haven't seen a macro before in Twig, it's basically a function that you create right inside Twig. It's really handy when you have some markup that you don't want to repeat over and over again.

Next, scroll down to the scientists area and copy everything inside the for statement. Delete it, and then paste it up in the macro:

52 lines | app/Resources/views/admin/genus/_form.html.twig

```
... lines 1 - 2
{% macro printGenusScientistRow(genusScientistForm) %}
    <div class="col-xs-4 js-genus-scientist-item">
        <a href="#" class="js-remove-scientist pull-right">
            <span class="fa fa-close"></span>
        </a>
        {{ form_errors(genusScientistForm) }}
        {{ form_row(genusScientistForm.user) }}
        {{ form_row(genusScientistForm.yearsStudied) }}
    </div>
{% endmacro %}
... lines 13 - 52
```

Use that Macro!

To call that macro, you actually need to import it... even though it already lives inside this template. Whatever: you can do that with {% import _self as formMacros %}:

52 lines | app/Resources/views/admin/genus/_form.html.twig

```
{% import _self as formMacros %}
... lines 2 - 52
```

The _self part would normally be the name of a *different* template whose macros you want to call, but _self is a magic way of saying, no, *this* template.

The formMacros is an alias I just invented, and it's how we will *call* the macro. For example, inside the for loop, render formMacros.printGenusScientistRow() and pass it genusScientistForm:

52 lines | app/Resources/views/admin/genus/_form.html.twig

```
... lines 1 - 13
{{ form_start(genusForm) }}
... lines 15 - 40
    {% for genusScientistForm in genusForm.genusScientists %}
        {{ formMacros.printGenusScientistRow(genusScientistForm) }}
    {% endfor %}
... lines 44 - 50
{{ form_end(genusForm) }}
```

And *now* we can do the same thing on the data-prototype attribute: formMacros.printGenusScientistRow() and pass that genusForm.genusScientists.vars.prototype. Continue to escape that that into HTML entities:

52 lines | app/Resources/views/admin/genus/_form.html.twig

```
... lines 1 - 13
{{ form_start(genusForm) }}
... lines 15 - 36
    <div class="row js-genus-scientist-wrapper"
        data-prototype="{{ formMacros.printGenusScientistRow(genusForm.genusScientists.vars.prototype)|e('html_attr') }}"
... line 39
    >
... lines 41 - 47
    </div>
... lines 49 - 50
{{ form_end(genusForm) }}
```

I love when things are this simple! Go back, refresh, and click to add another scientist. Much, much better! Obviously, we need a little styling help here with our rows but you guys get the idea.

Centralizing our JavaScript

The *last* problem with our form deals with JavaScript. Go to /admin/genus and click "Add". Well... our fancy JavaScript doesn't work here. Wah wah.

But that makes sense: we put all the JavaScript into the edit template. The fix for this is super old-fashioned... and yet perfect: we need to move all that JavaScript into its own file. Since this isn't a JavaScript tutorial, let's keep things simple: in web/js, create a new file: GenusAdminForm.js.

Ok, let's be a *little* fancy: add a self-executing block: a little function that calls itself and passes jQuery inside:

34 lines | web/js/GenusAdminForm.js

```
(function ($) {
... lines 2 - 32
})(jQuery);
```

Then, steal the code from edit.html.twig and paste it here. It doesn't *really* matter, but I'll use $ everywhere instead of jQuery to be consistent:

34 lines | web/js/GenusAdminForm.js

```
(function ($) {
    $(document).ready(function() {
        var $wrapper = $('.js-genus-scientist-wrapper');

        $wrapper.on('click', '.js-remove-scientist', function(e) {
            e.preventDefault();

            $(this).closest('.js-genus-scientist-item')
                .fadeOut()
                .remove();
        });

        $wrapper.on('click', '.js-genus-scientist-add', function(e) {
            e.preventDefault();

            // Get the data-prototype explained earlier
            var prototype = $wrapper.data('prototype');

            // get the new index
            var index = $wrapper.data('index');

            // Replace '__name__' in the prototype's HTML to
            // instead be a number based on how many items we have
            var newForm = prototype.replace(/__name__/g, index);

            // increase the index with one for the next item
            $wrapper.data('index', index + 1);

            // Display the form in the page before the "new" link
            $(this).before(newForm);
        });
    });
})(jQuery);
```

Back in the edit template, include a proper script tag: src="" and pass in the GenusAdminForm.js path:

20 lines | app/Resources/views/admin/genus/edit.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script src="{{ asset('js/GenusAdminForm.js') }}"></script>
{% endblock %}
... lines 8 - 20
```

Copy the *entire* javascripts block and then go into new.html.twig. Paste!

20 lines | app/Resources/views/admin/genus/new.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script src="{{ asset('js/GenusAdminForm.js') }}"></script>
{% endblock %}
... lines 8 - 20
```

And now, we should be happy: refresh the new form. Way better!

Avoiding the Weird New Label

But... what's with that random label - "Genus scientists" - after the submit button! What the crazy!?

Ok, so the reason this is happening is a little subtle. Effectively, because there are no genus scientists on this form, Symfony sort of thinks that this genusForm.genusScientists field was never rendered. So, like all unrendered fields, it tries to render it in form_end(). And this causes an extra label to pop out.

It's silly, but easy to fix: after we print everything, add form_widget(genusForm.genusScientists). And ya know what? Let's add a note above to explain this - otherwise it looks a little crazy.

54 lines | app/Resources/views/admin/genus/_form.html.twig

```
... lines 1 - 13
{{ form_start(genusForm) }}
... lines 15 - 36
    <div class="row js-genus-scientist-wrapper"
... lines 38 - 39
    >
... lines 41 - 47
    </div>
    {# prevents weird label from showing up in new #}
    {{ form_widget(genusForm.genusScientists) }}
... lines 51 - 52
{{ form_end(genusForm) }}
```

And don't worry, this will never actually print anything. Since all of the children fields are rendered above, Symfony knows not to *re-render* those fields. This just prevents that weird label.

Refresh! Extra label gone. And if you go back and edit one of the genuses, things look cool here too.

Now, I have *one* last challenge for us with our embedded forms.

Chapter 26: Form Events: A readonly Embedded Field

Ready for the last challenge? All four of these genus scientists are already saved to the database. And while I guess it's kind of cool that I can change this scientist from one User to another, it's also a little bit weird: When would I ever change a specific scientist from one User to another? If this User weren't studying this Genus anymore, I should delete them. And if a *new* User were studying this Genus, we should probably just add a *new* GenusScientist.

So I want to update the interface: when I hit "Add Another Scientist", I *do* want the User select, just like now. But for *existing* genus scientists - the ones that are already saved to the database - I want to simply print the user's email in place of the drop-down.

In Symfony language, this means that I want to remove the user field from the embedded form if the GenusScientist behind it is already saved.

About Form Events

To do that, open the GenusScientistEmbeddedForm. Guess what? We get to try a feature that I don't get to use very often: Symfony Form Events.

Here's the idea: every form has a life cycle: the form is created, initial data is set onto the form and then the form is submitted. And we can hook into this process!

Form Event Setup!

To do it, write addEventListener() and then pass a constant FormEvents::POST_SET_DATA. After that, say array($this, 'onPostSetData'):

51 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 11
use Symfony\Component\Form\FormEvents;
... lines 13 - 14
class GenusScientistEmbeddedForm extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 20 - 27
            ->addEventListener(
                FormEvents::POST_SET_DATA,
                array($this, 'onPostSetData')
            )
        ;
    }
... lines 34 - 50
}
```

Let's break that down: the POST_SET_DATA is a constant for an event called form.post_set_data. This is called after the data behind the form is added to it: in other words, after the GenusScientist is bound to each embedded form.

When that happens, the form system will call an onPostSetData() function, which we are about to create: public function onPostSetData(). This will receive a FormEvent object:

51 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 10
use Symfony\Component\Form\FormEvent;
... lines 12 - 14
class GenusScientistEmbeddedForm extends AbstractType
{
... lines 17 - 34
    public function onPostSetData(FormEvent $event)
    {
... lines 37 - 40
    }
... lines 42 - 50
}
```

Now we're close! Inside, add an if statement: if $event->getData():This form is always bound to a GenusScientist object. So this will return the GenusScientist object bound to this form, *or* - if this is a new form - then it may return null. That's why we'll say if $event->getData() && $event->getData()->getId():

51 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 14
class GenusScientistEmbeddedForm extends AbstractType
{
... lines 17 - 34
    public function onPostSetData(FormEvent $event)
    {
        if ($event->getData() && $event->getData()->getId()) {
... lines 38 - 39
        }
    }
... lines 42 - 50
}
```

In human-speak: as long as there is a GenusScientist bound to this form and it's been saved to the database - i.e. it has an id value - then let's unset the user field from the form.

To do that, fetch the form with $form = $event->getForm(). Then, literally, unset($form['user']):

51 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php

```
... lines 1 - 14
class GenusScientistEmbeddedForm extends AbstractType
{
... lines 17 - 34
    public function onPostSetData(FormEvent $event)
    {
        if ($event->getData() && $event->getData()->getId()) {
            $form = $event->getForm();
            unset($form['user']);
        }
    }
... lines 42 - 50
}
```

This $form variable is a Form object, but you can treat it like an array, including unsetting fields.

That's it for the form! The last step is to conditionally render the user field. Because if we refresh right now, the form system yells at us:

There's no user field inside of our template at line 9.

Wrap that in an if statement: if genusScientistForm.user is defined, then print it:

58 lines | app/Resources/views/admin/genus/_form.html.twig

```
... lines 1 - 2
{% macro printGenusScientistRow(genusScientistForm) %}
    <div class="col-xs-4 js-genus-scientist-item">
... lines 5 - 8
        {% if genusScientistForm.user is defined %}
            {{ form_row(genusScientistForm.user) }}
... lines 11 - 12
        {% endif %}
... line 14
    </div>
{% endmacro %}
... lines 17 - 58
```

Else, use a strong tag and print the user's e-mail address with genusScientistForm.vars - which is something we mastered in our Form Theming tutorial - .data - which will be a GenusScientist object - .user.email:

58 lines | app/Resources/views/admin/genus/_form.html.twig

```
... lines 1 - 2
{% macro printGenusScientistRow(genusScientistForm) %}
    <div class="col-xs-4 js-genus-scientist-item">
... lines 5 - 8
        {% if genusScientistForm.user is defined %}
            {{ form_row(genusScientistForm.user) }}
        {% else %}
            <strong>{{ genusScientistForm.vars.data.user.email }}</strong>
        {% endif %}
... line 14
    </div>
{% endmacro %}
... lines 17 - 58
```

This says: find the GenusScientist object behind this form, call getUser() on it, and then call getEmail() on that.

I think it's time to celebrate! Refresh the form. It looks *exactly* like I wanted. It's like my birthday! And when we add a new one, it *still* has the drop-down. You guys are the best!

Chapter 27: Collection Filtering: The Easy Way

I have *one* last cool trick to show you. Go back to /genus.

Oh, but real quick, I need to fix two little things that I messed up before we finish.

Oh my, a Missing inversedBy()

First, see that red label on the web debug toolbar? Click it, and scroll down. It's a mapping warning:

> The field User#studiedGenuses property is on the inverse side of a bidirectional relationship, but the association on blah-blah-blah does not contain the required inversedBy().

In human-speak, this says that my User correctly has a studiedGenuses property with a mappedBy option...

223 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 77
    /**
     * @ORM\OneToMany(targetEntity="GenusScientist", mappedBy="user")
     */
    private $studiedGenuses;
... lines 82 - 221
}
```

But on GenusScientist, I forgot to add the inversedBy() that points back to this:

79 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 17
class GenusScientist
{
... lines 20 - 32
    /**
     * @ORM\ManyToOne(targetEntity="User", inversedBy="studiedGenuses")
... line 35
     */
    private $user;
... lines 38 - 78
}
```

I don't really know why Doctrine requires this... since it didn't seem to break anything, but hey! This fixes the warning.

Bad Field Type Mapping!

The second thing I need to fix is this yearsStudied field. When PhpStorm generated the annotation for us, it used type="string"... and I forgot to fix it! Change it to type="integer":

79 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 17
class GenusScientist
{
... lines 20 - 38
    /**
     * @ORM\Column(type="integer")
... line 41
     */
    private $yearsStudied;
... lines 44 - 78
}
```

It hasn't caused a problem yet... but it would if we tried to do some number operations on it inside the database.

Of course, we need a migration!

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

Just trust that it's correct - live dangerously:

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

Sweet! Now go back to /genus.

Fetching a Subset of GenusScientist Results

We're already printing the number of scientists that each Genus has. *And* thanks to a fancy query we made inside GenusRepository, that joins over and fetches the related User data all at once... this entire page is built with one query:

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

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
    /**
     * @return Genus[]
     */
    public function findAllPublishedOrderedByRecentlyActive()
    {
        return $this->createQueryBuilder('genus')
            ->andWhere('genus.isPublished = :isPublished')
            ->setParameter('isPublished', true)
            ->leftJoin('genus.notes', 'genus_note')
            ->orderBy('genus_note.createdAt', 'DESC')
            ->leftJoin('genus.genusScientists', 'genusScientist')
            ->addSelect('genusScientist')
            ->getQuery()
            ->execute();
    }
}
```

Well, except for the query that loads my security user from the database.

So this is cool! Well, its *maybe* cool - as we talked about earlier, this is fetching a lot of extra data. And more importantly, this page may not be a performance problem in the first place. Anyways, I want to show you something cool, so comment out those joins:

26 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')
//            ->leftJoin('genus.genusScientists', 'genusScientist')
//            ->addSelect('genusScientist')
            ->getQuery()
            ->execute();
    }
}
```

Refresh again! Our *one* query became a bunch! Every row now has a query, but it's a really efficient COUNT query thanks to our fetch EXTRA_LAZY option:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 71
    /**
     * @ORM\OneToMany(
... lines 74 - 75
     *     fetch="EXTRA_LAZY",
... lines 77 - 78
     * )
... line 80
     */
    private $genusScientists;
... lines 83 - 209
}
```

Here's my new wild idea: any scientist that has studied a genus for longer than 20 years should be considered an *expert*. So, in addition to the number of scientists I also want to print the number of *expert* scientists next to it.

Look inside the list template: we're printing this number by saying genus.genusScientists|length:

29 lines | app/Resources/views/genus/list.html.twig

```
... lines 1 - 2
{% block body %}
    <table class="table table-striped">
... lines 5 - 12
        <tbody>
            {% for genus in genuses %}
                <tr>
... lines 16 - 21
                    <td>{{ genus.genusScientists|length }}</td>
... line 23
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

In other words, call getGenusScientists():

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 202
    /**
     * @return ArrayCollection|GenusScientist[]
     */
    public function getGenusScientists()
    {
        return $this->genusScientists;
    }
}
```

Fetch the results, and then count them:

But how could we filter this to *only* return GenusScientist results that have studied the Genus for longer than 20 years?

It's easy! In Genus, create a new public function called getExpertScientists():

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 213
    public function getExpertScientists()
    {
... lines 216 - 218
    }
}
```

Then, we'll loop over *all* of the scientists to find the experts. And actually, we can do that very easily by saying $this->getGenusScientists()->filter(), which is a method on the ArrayCollection object. Pass *that* an anonymous function with a GenusScientist argument. Inside, return $genusScientist->getYearsStudied() > 20:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 213
    public function getExpertScientists()
    {
        return $this->getGenusScientists()->filter(function(GenusScientist $genusScientist) {
            return $genusScientist->getYearsStudied() > 20;
        });
    }
}
```

This will loop over *all* of the genus scientists and return a new ArrayCollection that only contains the ones that have studied for more than 20 years. It's perfect!

To print this in the template, let's add a new line, then {{ genus.expertScientists|length }} and then "experts":

32 lines | app/Resources/views/genus/list.html.twig

```
... lines 1 - 2
{% block body %}
    <table class="table table-striped">
... lines 5 - 12
        <tbody>
            {% for genus in genuses %}
                <tr>
... lines 16 - 21
                    <td>
                        {{ genus.genusScientists|length }}
                        ({{ genus.expertScientists|length }} experts)
                    </td>
... line 26
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Try it! Refresh! Zero! What!? Oh... I forgot my return statement from inside the filter function. Lame!

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 213
    public function getExpertScientists()
    {
        return $this->getGenusScientists()->filter(function(GenusScientist $genusScientist) {
            return $genusScientist->getYearsStudied() > 20;
        });
    }
}
```

Try it now. Yes!

Click to check out the queries. It *still* makes a COUNT query for each row... but wait: it *also* queries for *all* of the

genus_scientist results for each genus. That sucks! Even if a Genus only has two experts... we're fetching *all* of the data for *all* of its genus scientists.

Why? Well, as soon as we loop over genusScientists:

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

```
... lines 1 - 14
class Genus
{
... lines 17 - 213
    public function getExpertScientists()
    {
        return $this->getGenusScientists()->filter(function(GenusScientist $genusScientist) {
            return $genusScientist->getYearsStudied() > 20;
        });
    }
}
```

Doctrine realizes that it needs to go and query for all of the genus scientists for this Genus. Then, we happily loop over them to see which ones have more than 20 yearsStudied.

This may or may not be a huge performance problem. If every Genus always has just a few scientists, no big deal! But if a Genus has hundreds of scientists, this page will grind to a halt while it queries for and hydrates all of those extra GenusScientist objects.

There's a better way: and it uses a feature in Doctrine that - until recently - even *I* didn't know existed. And I'm super happy it does.

Chapter 28: Criteria System: Champion Collection Filtering

Filtering a collection from inside of your entity like this is really convenient... but unless you *know* that you will always have a small number of total scientists... it's likely to slow down your page big.

Ready for a better way?! Introducing, Doctrine's Criteria system: a part of Doctrine that's *so* useful... and yet... I don't think anyone knows it exists!

Here's how it looks: create a $criteria variable set to Criteria::create():

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

```
... lines 1 - 5
use Doctrine\Common\Collections\Criteria;
... lines 7 - 15
class Genus
{
... lines 18 - 214
    public function getExpertScientists()
    {
        $criteria = Criteria::create()
... lines 218 - 221
    }
}
```

Next, we'll chain off of this and build something that looks *somewhat* similar to a Doctrine query builder. Say, andWhere(), then Criteria::expr()->gt() for a greater than comparison. There are a ton of other methods for equals, less than and any other operator you can dream up. Inside gt, pass it 'yearsStudied', 20:

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

```
... lines 1 - 5
use Doctrine\Common\Collections\Criteria;
... lines 7 - 15
class Genus
{
... lines 18 - 214
    public function getExpertScientists()
    {
        $criteria = Criteria::create()
            ->andWhere(Criteria::expr()->gt('yearsStudied', 20))
... lines 219 - 221
    }
}
```

And hey! Let's show off: add an orderBy() passing it an array with yearsStudied set to DESC:

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

```
... lines 1 - 5
use Doctrine\Common\Collections\Criteria;
... lines 7 - 15
class Genus
{
... lines 18 - 214
    public function getExpertScientists()
    {
        $criteria = Criteria::create()
            ->andWhere(Criteria::expr()->gt('yearsStudied', 20))
            ->orderBy(['yearsStudied', 'DESC']);
... lines 220 - 221
    }
}
```

This Criteria describes *how* we want to filter. To use it, return $this->getGenusScientists()->matching() and pass that $criteria:

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

```
... lines 1 - 5
use Doctrine\Common\Collections\Criteria;
... lines 7 - 15
class Genus
{
... lines 18 - 214
    public function getExpertScientists()
    {
        $criteria = Criteria::create()
            ->andWhere(Criteria::expr()->gt('yearsStudied', 20))
            ->orderBy(['yearsStudied', 'DESC']);

        return $this->getGenusScientists()->matching($criteria);
    }
}
```

That is it!

Now check this out: when we go back and refresh, we get all the same results. But the queries are *totally* different. It still counts all the scientists for the first number. But then, instead of querying for all of the genus scientists, it uses a WHERE clause with yearsStudied > 20. It's now doing the filtering in the *database* instead of in PHP.

As a bonus, because we're simply *counting* the results, it ultimately makes a COUNT query. But if - in our template, for example - we wanted to loop over the experts, maybe to print their names, Doctrine would be smart enough to make a SELECT statement for that data, instead of a COUNT. But that SELECT would *still* have the WHERE clause that filters in the database.

In other words guys, the Criteria system kicks serious butt: we can filter a collection from *anywhere*, but do it efficiently. Congrats to Doctrine on this feature.

Organizing Criteria into your Repository

But, to keep my code organized, I prefer to have all of my query logic inside of repository classes, including Criteria. No worries! Open GenusRepository and create a new static public function createExpertCriteria():

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

```
... lines 1 - 8
class GenusRepository extends EntityRepository
{
... lines 11 - 26
    static public function createExpertCriteria()
    {
... lines 29 - 31
    }
}
```

Tip

Whoops! It would be better to put this method in GenusScientistRepository, since it operates on that entity.

Copy the criteria line from genus, paste it here and return it. Oh, and be sure you type the "a" on Criteria and hit tab so that PhpStorm autocompletes the use statement:

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

```
... lines 1 - 5
use Doctrine\Common\Collections\Criteria;
... lines 7 - 8
class GenusRepository extends EntityRepository
{
... lines 11 - 26
    static public function createExpertCriteria()
    {
        return Criteria::create()
            ->andWhere(Criteria::expr()->gt('yearsStudied', 20))
            ->orderBy(['yearsStudied', 'DESC']);
    }
}
```

But wait, gasp! A static method! Why!? Well, it's because I need to be able to access it from my Genus class... and that's only possible if it's static. And also, I think it's fine: this method doesn't make a query, it simply returns a small, descriptive, static value object: the Criteria.

Back inside Genus, we can simplify things $this->getGenusScientists()->matching(GenusRepository::createExpertCriteria()):

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

```
... lines 1 - 16
class Genus
{
... lines 19 - 215
    public function getExpertScientists()
    {
        return $this->getGenusScientists()->matching(
            GenusRepository::createExpertCriteria()
        );
    }
}
```

Refresh that! Sweet! It works just like before.

Criteria in Query Builder

Another advantage of building the Criteria inside of your repository is that you can use it in a query builder. Imagine that we

needed to query for *all* of the experts in the entire system. To do that we could create a new public function - findAllExperts():

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

```
... lines 1 - 8
class GenusRepository extends EntityRepository
{
... lines 11 - 29
    public function findAllExperts()
    {
... lines 32 - 35
    }
... lines 37 - 43
}
```

Tip

Once again, this method should *actually* live in GenusScientistRepository, but the idea is exactly the same :).

But, I want to *avoid* duplicating the query logic that we already have in the Criteria!

No worries! Just return $this->createQueryBuilder('genus') then, addCriteria(self::createExpertCriteria()):

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

```
... lines 1 - 8
class GenusRepository extends EntityRepository
{
... lines 11 - 29
    public function findAllExperts()
    {
        return $this->createQueryBuilder('genus')
            ->addCriteria(self::createExpertCriteria())
... lines 34 - 35
    }
... lines 37 - 43
}
```

Finish with the normal getQuery() and execute():

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

```
... lines 1 - 8
class GenusRepository extends EntityRepository
{
... lines 11 - 26
    /**
     * @return Genus[]
     */
    public function findAllExperts()
    {
        return $this->createQueryBuilder('genus')
            ->addCriteria(self::createExpertCriteria())
            ->getQuery()
            ->execute();
    }
... lines 37 - 43
}
```

How cool is that!?

Ok guys, that's it - that's everything. We just attacked the stuff that *really* frustrates people with Doctrine and Forms. Collections are hard, but if you understand the mapping and the inverse side reality, you write your code to update the mapping side *from* the inverse side, and understand a few things like orphanRemoval and cascade, everything falls into place.

Now that you guys know what to do, go forth, attack collections and create something amazing.

All right guys, see you next time.