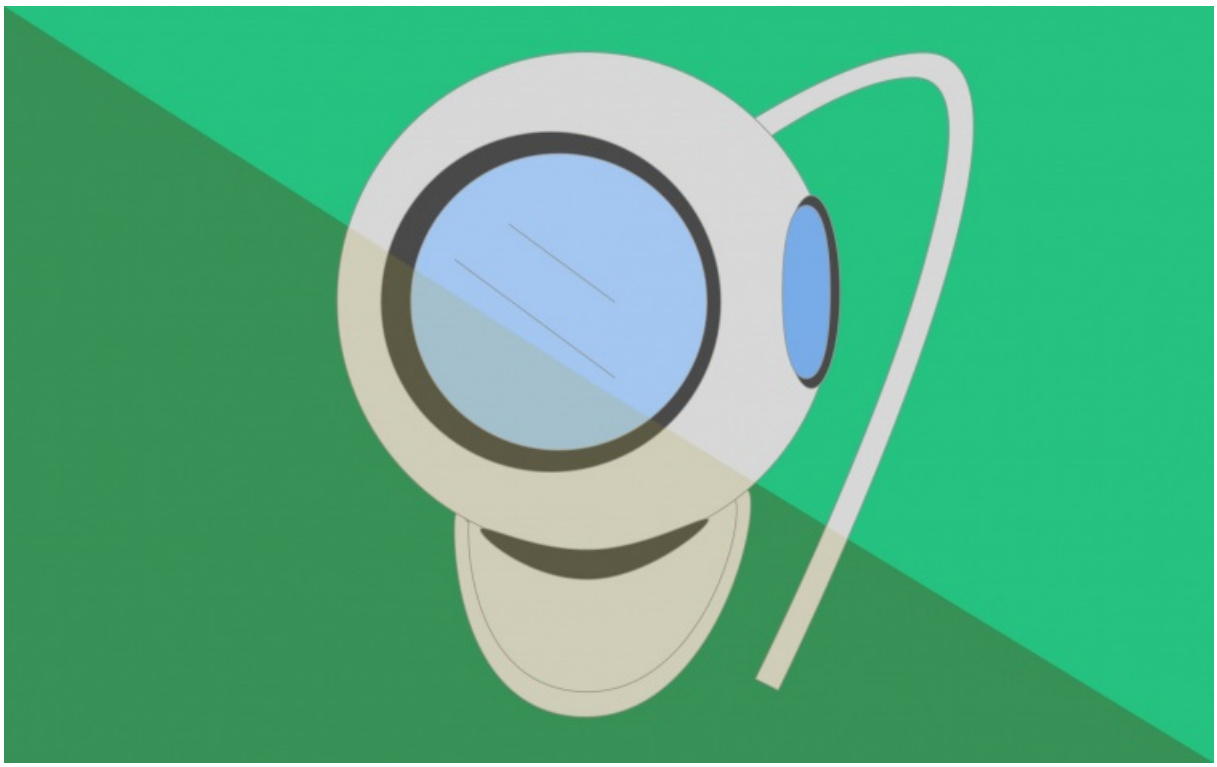
Symfony 3: Doctrine & the Database



With <3 from SymfonyCasts

Chapter 1: Creating an Entity Class

Yo guys! Time to level-up our project in a *big* way. I mean, *big*. This series is *all* about an incredible library called Dog-trine. Wait, that's not right - it's *Doctrine*. But anyways, Doctrine is *kinda* like a dog: a dog fetches a bone, Doctrine fetches data from the database. But Doctrine does *not* pee in the house. That's part of what makes it awesome.

But back up: is Doctrine part of Symfony? Nope. Symfony doesn't care *how* or *if* you talk to a database at all. You could use a direct PDO connection, use Doctrine, or do something else entirely. As usual, you're in control.

If you want to code along - which you *should* - then download the code from the screencast page and move into the directory called start. I already have the start code downloaded, so I'll go straight to opening up a new terminal and starting the built-in sever with:

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

Tip

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

Perfect!

Doctrine is an ORM

Doctrine is an ORM: object relational mapper. In short, that means that every table - like genus - will have a corresponding PHP class that we will create. When you query the genus table, Doctrine will give you a Genus object. Every property in the class maps to a column in the table. Keep this simple idea in mind as we go along: this mapping between a table and a PHP class is Doctrine's main goal.

Oh, and before we hop in, I want you to remember something very important: all the tools in Symfony are *optional*, including Doctrine. If Doctrine - or any tool - does more harm than good while solving a problem, skip it and do something simpler. Tools are meant to serve *you*, not the other way around.

Your First Entity Class

Our sweet app displays information about different ocean-living genuses... but so far, all that info is hardcoded. That's so sad.

Instead, let's create a genus table in the database and load all of this dynamically from there. How do you create a database table with Doctrine? You don't! Your job is to create a class, then Doctrine will create the table *based* on that class. It's pretty sweet. Oh, and the *whole* setup is going to take about 2 minutes and 25 lines of code. Watch.

Create an Entity directory in AppBundle and then create a normal class inside called Genus:

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

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

class Genus
{
}
```

You're going to hear this word - *entity* - a lot with Doctrine. Entity - it sounds like an alien parasite. Fortunately, it's less scary than that: an entity is just a class that Doctrine will map to a database table.

Configuration with... Annotations!

To do that - Doctrine needs to know two things: what the table should be called and what columns it needs. To help it out, we're going to use... drumroll... annotations! Remember, whenever you use an annotation, you need a use statement for it.

This will look weird, but add a use for a Column class and let it auto-complete from Doctrine\ORM\Mapping. Remove the Column part and add as ORM:

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

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

use Doctrine\ORM\Mapping as ORM;
... lines 6 - 15
```

Tip

You can also configure Doctrine with YAML, XML or PHP, instead of annotations. Check Add Mapping Information to see how to configure it.

Every entity class will have that *same* use statement. Next, put your cursor inside the class and open up the "Code"->"Generate" menu - cmd+N on a Mac. Ooh, one of the options is ORM Class. Click that... and boom! It adds two annotations - @ORM\Entity and @ORM\Table above the class:

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

```
... lines 1 - 6
/**
 * @ORM\Entity
 * @ORM\Table(name="genus")
 */
class Genus
{

}
```

Doctrine now knows this class should map to a table called genus.

Configuring the Columns

But that table won't have *any* columns yet. Lame. Add two properties to get us rolling: id and name. To tell Doctrine that these should map to columns, open up the "Code"->"Generate" menu again - or cmd+N. This time, select ORM Annotation and highlight both properties. And, boom again!

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

```
... lines 1 - 6
/**
 * @ORM\Entity
 * @ORM\Table(name="genus")
 */
class Genus
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $name;
}
```

Now we have annotations above each property. The id columns is special - it will almost always look exactly like this: it basically says that id is the primary key.

After that, you'll have whatever other columns you need. Hey, look at the type option that's set to string. That's a Doctrine "type", and it will map to a varchar in MySQL. There are other Doctrine types for strings, floats and text - we'll talk about those soon!

And with just 25 lines of code, we're done! In a second, we'll ask Doctrine to create the genus table for us and we'll be ready to start saving stuff. Well, let's get to it!

Chapter 2: Database Config and Automatic Table Creation

We *described* the genus table to Doctrine via annotations, but this table doesn't exist yet. No worries - Doctrine can create it for us!

And actually, we don't even have a database yet. Doctrine can also handle this. Head to the terminal use the console to run:

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

But wait! Can Doctrine do this yet? We haven't told it *anything* about the database: not the name we want, the user or the password.

Configuring the Database

Where do we do that? The same place that *everything*, meaning all *services* are configured: app/config/config.yml. Scroll down to the doctrine key:

72 lines | app/config/config.yml

```
... lines 1 - 43
# Doctrine Configuration
doctrine:
    dbal:
        driver:   pdo_mysql
        host:     "%database_host%"
        port:     "%database_port%"
        dbname:   "%database_name%"
        user:     "%database_user%"
        password: "%database_password%"
        charset:  UTF8
        # if using pdo_sqlite as your database driver:
        #   1. add the path in parameters.yml
        #     e.g. database_path: "%kernel.root_dir%/data/data.db3"
        #   2. Uncomment database_path in parameters.yml.dist
        #   3. Uncomment next line:
        #     path:     "%database_path%"
... lines 60 - 72
```

Ah, *this* is what tells Doctrine all about your database connection.

But, the information is not hardcoded here - these are references to parameters that are defined in parameters.yml:

15 lines | app/config/parameters.yml.dist

```
... lines 1 - 3
parameters:
    database_host:     127.0.0.1
    database_port:     ~
    database_name:     symfony
    database_user:     root
    database_password: ~
    # You should uncomment this if you want use pdo_sqlite
    # database_path: "%kernel.root_dir%/data.db3"
... lines 12 - 15
```

Update the database_name to aqua_note and on my super-secure local machine, the database user is root with no

password.

Go Deeper!

Find out more about these parameters in our Symfony Fundamentals Series.

Back to the terminal! *Now* hit enter on the command:

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

Database created. To create the table, run:

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

This looks great - CREATE TABLE genus with the two columns. But this didn't *execute* the query yet - the --dump-sql option is used to preview the query if you're curious. Replace it with --force.

```
$ ./bin/console doctrine:schema:update --force
```

So hey guys, this is really cool - we can be totally lazy and let Doctrine do all the heavy database-lifting for us. This doctrine:schema:update command is actually more powerful than it looks - it's going to "wow" us in a few minutes.

But first, let's learn how to insert data into the new table.

Chapter 3: Inserting new Objects

Fearless aquanauts are constantly discovering and re-classifying deep-sea animals. If a new genus needed to be added to the system, what would that look like? Well, we would probably have a URL like /genus/new. The user would fill out a form, hit submit, and the database fairies would insert a new record into the genus table.

Sounds good to me! In GenusController, create a newAction() with the URL /genus/new. I won't give the route a name yet - that's not needed until we link to it:

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

```
... lines 1 - 11
class GenusController extends Controller
{
    /**
     * @Route("/genus/new")
     */
    public function newAction()
    {
... line 19
    }

    /**
     * @Route("/genus/{genusName}")
     */
    public function showAction($genusName)
    {
... lines 27 - 46
    }
... lines 48 - 65
}
```

Be Careful with Route Ordering!

Oh, and side-note: I put newAction() *above* showAction(). Does that matter? In this case, absolutely. Remember, routes match from top to bottom. If I had put newAction() *below* showAction(), going to /genus/new would have matched showAction() - passing the word "new" as the genusName(). To avoid this, put your most generic-matching routes near the bottom.

Go Deeper!

You can often also use route requirements to make a wildcard only match certain patterns (instead of matching everything).

Inserting a Genus

Ok, back to the database-world. Let's be *really* lazy and skip creating a form - there's a *whole* series on forms later, and, I'd just hate to spoil the fun.

Instead, insert some hardcoded data. How? Simple: start with $genus = new Genus(), put some data on that object, and tell Doctrine to save it:

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

```
... lines 1 - 4
use AppBundle\Entity\Genus;
... lines 6 - 11
class GenusController extends Controller
{
    /**
     * @Route("/genus/new")
     */
    public function newAction()
    {
        $genus = new Genus();
    }
... lines 21 - 65
}
```

Doctrine wants you to *stop* thinking about queries, and instead think about *objects*.

Right now... name is the only real field we have. And it's a private property, so we *can't* actually put data on it. To make this mutable - to use a really fancy term that means "changeable" - go to the bottom of the class and open the "Code"->"Generate" menu. Select "Getters and Setters" - we'll need the getter function later:

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 24
    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
        $this->name = $name;
    }
}
```

Great! Now use $genus->setName() and call it Octopus with a random ending to make things more fun!

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 16
    public function newAction()
    {
        $genus = new Genus();
        $genus->setName('Octopus'.rand(1, 100));
... lines 21 - 26
    }
... lines 28 - 72
}
```

Object, check! Populated with data, check! The last step is to say:

> Hey Doctrine. I want you to save this to our genus table.

Remember how *everything* in Symfony is done with a service? Doctrine is no exception: it has *one* magical service that saves *and* queries. It's called, the *entity manager*. In fact, it's so hip that it has its own controller shortcut to get it: $em = $this->getDoctrine()->getManager(). What a celebrity.

To save data, use two methods $em->persist($genus) and $em->flush():

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

```
... lines 1 - 18
19          $genus = new Genus();
20          $genus->setName('Octopus'.rand(1, 100));
21
22          $em = $this->getDoctrine()->getManager();
23          $em->persist($genus);
24          $em->flush();
... lines 25 - 74
```

And yes, you need *both* lines. The first just tells Doctrine that you *want* to save this. But the query isn't made until you call flush(). What's *really* cool is that you'll use these *exact* two lines whether you're inserting a new Genus or updating an existing one. Doctrine figures out the right query to use.

Finishing the New Page

Ok, let's finish up! Do you remember what a controller must *always* return? Yep! A Response object. Skip a template and just return new Response() - the one from the HttpFoundation component - with Genus Created:

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

```
... lines 1 - 11
12  class GenusController extends Controller
13  {
... lines 14 - 16
17      public function newAction()
18      {
... lines 19 - 25
26          return new Response('Genus created!');
27      }
... lines 28 - 72
73  }
```

Deep breath. Head to /genus/new. Okay, okay - no errors. I *think* we're winning?

Debugging with the Web Debug Toolbar

The web debug toolbar has a way to see the queries that were made... but huh, it's missing! Why? Because we don't have a full, valid HTML page. That's a bummer - so go back to the controller and hack in some HTML markup into the response:

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

```
... lines 1 - 25
26          return new Response('<html><body>Genus created!</body></html>');
... lines 27 - 74
```

Try it again! Ah, there you are fancy web debug toolbar. There are actually *three* database queries. Interesting. Click the icon to enter the profiler. Ah, there's the insert query, hiding inside a transaction.

And by the way, how sweet is this for debugging? You can see a formatted query, a runnable version, or run EXPLAIN on a slow query.

Running SQL Queries in the Terminal

I still can't believe it's working - things never work on the first try! To triple-check it, head to the terminal. To run a raw SQL query, use:

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

There they are. So inserting objects with Doctrine... pretty darn easy.

Chapter 4: Adding More Columns

We're *close* to making the genus page dynamic - but it has a few fields that we don't have yet - like sub family, number of known species, and fun fact.

Let's add these... and be as lazy as possible when we do it! In Genus, create the properties first: private $subFamily, private $speciesCount and private $funFact:

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 27
    private $subFamily;
... lines 29 - 32
    private $speciesCount;
... lines 34 - 37
    private $funFact;
... lines 39 - 79
}
```

Now, just do the same thing we did before: bring up the "Code"->"Generate" menu - or command+N on a Mac - select "ORM Annotation" and choose all the fields to generate the Column annotations above each:

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 24
    /**
     * @ORM\Column(type="string")
     */
    private $subFamily;

    /**
     * @ORM\Column(type="integer")
     */
    private $speciesCount;

    /**
     * @ORM\Column(type="string")
     */
    private $funFact;
... lines 39 - 79
}
```

These tries to *guess* the right field type - but it's not always right. speciesCount should clearly be an integer - set it to that. For a full-list of *all* the built-in Doctrine types, check their docs. The most important are string, integer, text, datetime and float.

Next, head to the bottom of the class to create the getter and setter methods. But wait! Realize: you might *not* need getters and setters for *every* field, so only add them if you need them. We *will* need them, so open "Code"->"Generate" again and select "Getters and Setters":

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 49
    public function getSubFamily()
    {
        return $this->subFamily;
    }

    public function setSubFamily($subFamily)
    {
        $this->subFamily = $subFamily;
    }

    public function getSpeciesCount()
    {
        return $this->speciesCount;
    }

    public function setSpeciesCount($speciesCount)
    {
        $this->speciesCount = $speciesCount;
    }

    public function getFunFact()
    {
        return $this->funFact;
    }

    public function setFunFact($funFact)
    {
        $this->funFact = $funFact;
    }

}
```

And that's it! Create the properties, generate the annotations and generate the getters and setters if you need them.

Updating the Table Schema

Now, can we tell Doctrine to somehow *alter* the genus table and add these columns? Absolutely. In fact, it's one of the most *incredible* features of Doctrine.

In the terminal, run:

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

Look familiar? That's the *same* command we ran before. But look: it actually says, "ALTER TABLE genus" add sub_family, species_count, and fun_fact. This is *amazing*. It's able to look at the database, look at the entity, and calculating the *difference* between them.

So if we were to run this with --force, it would run that query and life would be good. Should we do it? No! Wait, hold on!

It *would* work. But imagine our app was *already* deployed and working on production with the first version of the genus table. When you deploy the new code, you'll need to run this command on production to update your table.

And that'll work 99% of the time. But sometimes, the query might not be perfect. For example, what if I rename a property?

Well, this command might *drop* the existing column and add a new one. All the data from the old column would be gone! The point is: running doctrine:schema:update is just *too* dangerous on production.

But, we're going to replace it with something *just* as good.

Chapter 5: Database Migrations

Google for DoctrineMigrationsBundle. To install it, copy the composer require line. But again, we don't need to have the version - Composer will find the best version for us:

Tip

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

```
$ composer require doctrine/doctrine-migrations-bundle:1.1
```

While Jordi is preparing that for us, let's keep busy. Copy the new statement from the docs and paste that into the AppKernel class:

56 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 20
            new Doctrine\Bundle\MigrationsBundle\DoctrineMigrationsBundle(),
... lines 22 - 23
        );
... lines 25 - 33
    }
... lines 35 - 54
}
```

Beautiful!

We already know that the *main* job of a bundle is to give us new services. But this bundle primarily gives us something different: a new set of console commands. Run bin/console with no arguments:

```
$ ./bin/console
```

Hiding in the middle is a whole group starting with doctrine:migrations. These are our new best friend.

The Migrations Workflow

Our goal is to find a way to *safely* update our database schema both locally *and* on production.

To do this right, drop the database entirely to remove all the tables: like we have a new project.

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

This is the *only* time you'll need to do this. Now, re-create the database:

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

Now, instead of running doctrine:schema:update, run:

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

This created a new file in app/DoctrineMigrations. Go open that up:

29 lines | app/DoctrineMigrations/Version20160207083131.php

```
... lines 1 - 10
class Version20160207083131 extends AbstractMigration
{
    public function up(Schema $schema)
    {
        // this up() migration is autogenerated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != "mysql");

        $this->addSql("CREATE TABLE genus (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255) NOT NULL, sub_family V
    }

    public function down(Schema $schema)
    {
        // this down() migration is autogenerated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() != "mysql");

        $this->addSql("DROP TABLE genus");
    }
}
```

Check this out: the up() method executes the *exact* SQL that we would have gotten from the doctrine:schema:update command. But instead of running it, it saves it into this file. This is *our* chance to look at it and make sure it's perfect.

When you're ready, run the migration with:

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

Done! Obviously, when you deploy, you'll *also* run this command. But here's the *really* cool part: this command will *only* run the migration files that have *not* been executed before. Behind the scenes, this bundle creates a migrations_versions table that keep strack of which migration files it has already executed. This means you can safely run doctrine:migrations:migrate on every deploy: the bundle will take care of only running the new files.

Tip

You *can* run migration in reverse in case something fails. Personally, I never do this and I never worry about down() being correct. If you have a migration failure, it's a bad thing and it's better to diagnose and fix it manually.

Making Columns nullable

In newAction(), I'll add some code that sets fake data on the subFamily and speciesCount properties. But, I'll keep funFact blank: maybe some genuses just aren't very fun:

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 16
    public function newAction()
    {
        $genus = new Genus();
        $genus->setName('Octopus'.rand(1, 100));
        $genus->setSubFamily('Octopodinae');
        $genus->setSpeciesCount(rand(100, 99999));
... lines 23 - 28
    }
... lines 30 - 74
}
```

Ok, head over to /genus/new to try it out! Woh, a huge explosion!

> Integrity constraint violation: 1048 Column fun_fact cannot be null

Here's the deal: Doctrine configures *all* columns to be required in the database by default. If you *do* want a column to be "nullable", find the column and add nullable=true:

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 34
    /**
     * @ORM\Column(type="string", nullable=true)
     */
    private $funFact;
... lines 39 - 79
}
```

Creating Another Migration

Of course, just because we made this change doesn't mean that our table was automatically updated behind the scenes. Nope: we need another migration. No problem! Go back to the terminal and run:

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

Open up the new migration file: ALTER TABLE genus CHANGE fun_fact to have a default of null. This look perfect. Run it with:

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

So easy! Refresh the page again: *no* errors. Migrations are awesome.

Chapter 6: Query for a List of Genuses

Woh guys, we can *already* create new tables, add columns *and* insert or update data. There's just one big piece left: querying.

Let's create a new page that will show off *all* the genuses. Create public function listAction() and give it a route path of /genus:

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

```
... lines 1 - 30
31      /**
32       * @Route("/genus")
33       */
34      public function listAction()
35      {
... lines 36 - 40
41      }
... lines 42 - 88
```

Querying? Get the Entity Manager

Remember, *everything* in Doctrine starts with the all-powerful entity manager. Just like before, get it with $em = $this->getDoctrine()->getManager():

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

```
... lines 1 - 33
34      public function listAction()
35      {
36          $em = $this->getDoctrine()->getManager();
... lines 37 - 40
41      }
... lines 42 - 88
```

To make a query, you'll always start the same way: $genuses = $em->getRepository(). Pass this the *class* name - not the table name - that you want to query from: AppBundle\Entity\Genus. This gives us a repository object, and hey! He's *really* good at querying from the genus table. In fact, it's got a bunch of useful methods on it like findAll() and findOneBy. Use findAll():

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

```
... lines 1 - 33
34      public function listAction()
35      {
36          $em = $this->getDoctrine()->getManager();
37
38          $genuses = $em->getRepository('AppBundle\Entity\Genus')
39              ->findAll();
... line 40
41      }
... lines 42 - 88
```

What does this return exactly? Um... I don't know - so let's find out! Dump $genuses to see what it looks like:

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

```
... lines 1 - 37
        $genuses = $em->getRepository('AppBundle\Entity\Genus')
            ->findAll();
        dump($genuses);die;
... lines 41 - 88
```

Back to the browser! Go to /genus... and there's the dump! Ah, it's an array of Genus objects. That makes sense - Doctrine is obsessed with always using objects. And sure, you *can* make queries that only return *some* columns, but that's for later.

The AppBundle:Genus Alias

Back to the controller! Now change the Genus class name to just AppBundle:Genus:

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

```
... lines 1 - 37
        $genuses = $em->getRepository('AppBundle:Genus')
            ->findAll();
... lines 40 - 88
```

Wait, what? Didn't I say this should be the *class* name? What is this garbage? It's cool - this is just a shortcut. Internally, Doctrine converts this to AppBundle\Entity\Genus. You can use either form, but usually you'll see the shorter one.... ya know, because programmers are efficient... or maybe lazy.

Chapter 7: Entities, Twig and the Magic dot Syntax

Let's *finally* make this page real with a template. Return $this->render('genus/list.html.twig') and pass it a genuses variable:

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 33
    public function listAction()
    {
        $em = $this->getDoctrine()->getManager();

        $genuses = $em->getRepository('AppBundle:Genus')
            ->findAll();

        return $this->render('genus/list.html.twig', [
            'genuses' => $genuses
        ]);
    }
... lines 45 - 89
}
```

You know what to do from here: in app/Resources/views/genus, create the new list.html.twig template. Don't forget to extend base.html.twig and then override the body block. I'll paste a table below to get us started:

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

```
{% extends 'base.html.twig' %}

{% block body %}
    <table class="table table-striped">
        <thead>
            <tr>
                <th>Genus</th>
                <th># of species</th>
            </tr>
        </thead>
        <tbody>
... lines 12 - 14
        </tbody>
    </table>
{% endblock %}
```

Since genuses is an array, loop over it with {% for genus in genuses %} and add the {% endfor %}. Next, just dump out genus inside:

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

```
... lines 1 - 10
        <tbody>
            {% for genus in genuses %}
                {{ dump(genus) }}
            {% endfor %}
        </tbody>
... lines 16 - 18
```

Looks like a good start - try it out!. Ok cool - this dumps out 4 Genus objects. Open up the tr. Bring this to life with a td that prints {{ genus.name }} and another that prints {{ genus.speciesCount }}. And hey, we're getting autocompletion, that's kind of nice:

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

```
... lines 1 - 10
        <tbody>
            {% for genus in genuses %}
                <tr>
                    <td>{{ genus.name }}</td>
                    <td>{{ genus.speciesCount }}</td>
                </tr>
            {% endfor %}
        </tbody>
... lines 19 - 21
```

The Magic Twig "." Notation

Refresh! Easy - it looks *exactly* how we want it. But wait a second... something cool just happened in the background. We printed genus.name... but name is a *private* property - so we should *not* be able to access it directly. How is this working?

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 22
    private $name;
... lines 24 - 39
    public function getName()
    {
        return $this->name;
    }
... lines 44 - 79
}
```

This is Twig to the rescue! Behind the scenes, Twig noticed that name was private and called getName() instead. And it does the same thing with genus.speciesCount:

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 32
    private $speciesCount;
... lines 34 - 59
    public function getSpeciesCount()
    {
        return $this->speciesCount;
    }
... lines 64 - 79
}
```

Twig is smart enough to figure out *how* to access the data - and this lets us keep the template simple.

With that in mind, I have a challenge! Add a third column to the table called "Last Updated":

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

```
... lines 1 - 4
        <thead>
            <tr>
                <th>Genus</th>
                <th># of species</th>
                <th>Last updated</th>
            </tr>
        </thead>
... lines 12 - 23
```

This won't work yet, but what I I *want* to be able to say is {{ genus.updatedAt }}. If this existed and returned a DateTime object, we could pipe it through the built-in Twig date filter to format it:

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

```
... lines 1 - 11
        <tbody>
            {% for genus in genuses %}
                <tr>
                    <td>{{ genus.name }}</td>
                    <td>{{ genus.speciesCount }}</td>
                    <td>{{ genus.updatedAt|date('Y-m-d') }}</td>
                </tr>
            {% endfor %}
        </tbody>
... lines 21 - 23
```

But this *won't* work - there is *not* an updatedAt property. We'll add one later, but we're stuck right now.

Wait! We can fake it! Add a public function getUpdatedAt() and return a random DateTime object:

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 79
    public function getUpdatedAt()
    {
        return new \DateTime('-'.rand(0, 100).' days');
    }
}
```

Try that out. It works! Twig doesn't care that there is no updatedAt property - it happily calls the getter function. Twig, you're awesome.

Chapter 8: Show them a Genus, and the 404

We have a list page! Heck, we have a show page. Let's link them together.

First, the poor show route is nameless. Give it a name - and a new reason to live - with name="genus_show":

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 45
    /**
     * @Route("/genus/{genusName}", name="genus_show")
     */
    public function showAction($genusName)
... lines 50 - 89
}
```

That sounds good.

In the list template, add the a tag and use the path() function to point this to the genus_show route. Remember - this route has a {genusName} wildcard, so we *must* pass a value for that here. Add a set of curly-braces to make an array... But this is getting a little long: so break onto multiple lines. Much better. Finish with genusName: genus.name. And make sure the text is still genus.name:

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', {'genusName': genus.name}) }}">
                            {{ genus.name }}
                        </a>
                    </td>
... lines 20 - 21
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

Cool! Refresh. Oooh, pretty links. Click the first one. The name is "Octopus66", but the fun fact and other stuff is *still* hardcoded. It's time to grow up and finally make this dynamic!

Querying for One Genus

In the controller, get rid of $funFact. We need to query for a Genus that matches the $genusName. First, fetch the entity manager with $em = $this->getDoctrine()->getManager():

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 48
    public function showAction($genusName)
    {
        $em = $this->getDoctrine()->getManager();
... lines 52 - 75
    }
... lines 77 - 94
}
```

Then, $genus = $em->getRepository() with the AppBundle:Genus shortcut. Ok now, is there a method that can help us? Ah, how about findOneBy(). This works by passing it an array of things to find by - in our case 'name' => $genusName:

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

```
... lines 1 - 50
        $em = $this->getDoctrine()->getManager();

        $genus = $em->getRepository('AppBundle:Genus')
            ->findOneBy(['name' => $genusName]);
... lines 55 - 96
```

Oh, and comment out the caching for now - it's temporarily going to get in the way:

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

```
... lines 1 - 50
        $em = $this->getDoctrine()->getManager();

        $genus = $em->getRepository('AppBundle:Genus')
            ->findOneBy(['name' => $genusName]);

        // todo - add the caching back later
        /*
        $cache = $this->get('doctrine_cache.providers.my_markdown_cache');
        $key = md5($funFact);
        if ($cache->contains($key)) {
            $funFact = $cache->fetch($key);
        } else {
            sleep(1); // fake how slow this could be
            $funFact = $this->get('markdown.parser')
                ->transform($funFact);
            $cache->save($key, $funFact);
        }
        */
... lines 69 - 96
```

Get outta here caching!

Finally, since we have a Genus *object*, we can simplify the render() call and *only* pass it:

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 48
    public function showAction($genusName)
    {
... lines 51 - 72
        return $this->render('genus/show.html.twig', array(
            'genus' => $genus
        ));
    }
... lines 77 - 94
}
```

Open up show.html.twig: we just changed the variables passed into this template, so we've got work to do. First, use genus.name and then genus.name again:

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

```
... lines 1 - 2
{% block title %}Genus {{ genus.name }}{% endblock %}

{% block body %}
    <h2 class="genus-name">{{ genus.name }}</h2>
... lines 7 - 21
{% endblock %}
... lines 23 - 40
```

Remove the hardcoded sadness and replace it with genus.subFamily, genus.speciesCount and genus.funFact. Oh, and remove the raw filter - we're temporarily not rendering this through markdown. Put it on the todo list:

40 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">
                <dt>Subfamily:</dt>
                <dd>{{ genus.subFamily }}</dd>
                <dt>Known Species:</dt>
                <dd>{{ genus.speciesCount|number_format }}</dd>
                <dt>Fun Fact:</dt>
                <dd>{{ genus.funFact }}</dd>
            </dl>
        </div>
    </div>
    <div id="js-notes-wrapper"></div>
{% endblock %}
... lines 23 - 40
```

There's *one* more spot down in the JavaScript - change this to genus.name:

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

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

Okay team, let's give it a try. Refresh. Looks awesome! The known species is the number it should be, there is no fun fact, and the JavaScript is still working.

Handling 404's

But what would happen if somebody went to a genus name that *did* not exist - like FOOBARFAKENAMEILOVEOCTOPUS? Woh! We get a bad error. This is coming from Twig:

> Impossible to access an attribute ("name") on a null variable

because on line 3, genus is null - it's *not* a Genus object:

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

```
... lines 1 - 2
{% block title %}Genus {{ genus.name }}{% endblock %}
... lines 4 - 40
```

In the prod environment, this would be a 500 page. We do *not* want that - we want the user to see a nice 404 page, ideally with something really funny on it.

Back in the controller, the findOneBy() method will either return *one* Genus object or null. If it does *not* return an object, throw $this->createNotFoundException('No genus found'):

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 48
    public function showAction($genusName)
    {
        $em = $this->getDoctrine()->getManager();

        $genus = $em->getRepository('AppBundle:Genus')
            ->findOneBy(['name' => $genusName]);

        if (!$genus) {
            throw $this->createNotFoundException('genus not found');
        }
... lines 59 - 79
    }
... lines 81 - 98
}
```

Oh, and that message will only be shown to developers - not to end-users.

Head back, refresh, and *this* is a 404. In the prod environment, the user will see a 404 template that you need to setup. I won't cover how to customize the template here - it's pretty easy - just make sure it's really clever, and send me a screenshot. Do it!

Chapter 9: Fixtures: Dummy Data Rocks

It's *so* much more fun to develop when your database has real, interesting data. We *do* have a way to add some fake genuses into the database, but they're not very interesting. And when we need more dummy data - like users and genus notes - it's just not going to work well.

Nope - we can do better. I'm dreaming of a system where we can quickly re-populate our local database with a really *rich* set of fake data, or *fixtures*.

Search for DoctrineFixturesBundle. This bundle is step 1 towards my dream. Copy the composer require line and paste that into the terminal. But hold on! I *also* want to download something else: nelmio/alice. That's just a normal PHP library, not a bundle. And it's going to make our fixtures amazing:

Tip

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

```
$ composer require --dev doctrine/doctrine-fixtures-bundle:2.3.0 nelmio/alice:2.1.4
```

Tip

Be sure to install version 2 of Alice, as version 3 has many changes: $ composer require --dev nelmio/alice:2.1.4

Conditionally Load Dev Libraries

Oh, and the --dev flag isn't too important. It means that these lines will be added to the require-dev section of composer.json:

68 lines | composer.json

```
{
... lines 2 - 31
    "require-dev": {
... lines 33 - 34
        "nelmio/alice": "^2.1",
        "doctrine/doctrine-fixtures-bundle": "^2.3"
    },
... lines 38 - 66
}
```

And that's meant for libraries that are only needed for development or to run tests.

When you deploy - if you care enough - you can tell composer to *not* download the libraries in this section. But frankly, I don't bother.

While Composer is communicating with the mothership, copy the new bundle line and add it to AppKernel. But put it in the section that's inside of the dev if statement:

57 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
... lines 10 - 25
        if (in_array($this->getEnvironment(), array('dev', 'test'), true)) {
... lines 27 - 30
            $bundles[] = new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle();
        }
... lines 33 - 34
    }
... lines 36 - 55
}
```

This makes the bundle - and any services, commands, etc that it gives us - *not* available in the prod environment. That's fine for us - this is a development tool - and it keeps the prod environment a little smaller.

Creating the Fixture Class

Anyways, this bundle gives us a new console command - doctrine:fixtures:load. When we run that, it'll look for "fixture classes" and run them. And in those classes, we'll create dummy data.

Copy the example fixture class. In AppBundle, add a DataFixtures/ORM directory. Then, add a new PHP class called - well, it doesn't matter - how about LoadFixtures. Paste the example class we so aggressively stole from the docs and update its class name to be LoadFixtures:

21 lines | src/AppBundle/DataFixtures/ORM/LoadFixtures.php

```
... lines 1 - 2
namespace AppBundle\DataFixtures\ORM;
... lines 4 - 5
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;

class LoadFixtures implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
... lines 13 - 19
    }
}
```

Clear out that User code. We need to create Genuses.. and we have some perfectly good code in newAction() we can steal to do that. Paste that it:

21 lines | src/AppBundle/DataFixtures/ORM/LoadFixtures.php

```
... lines 1 - 4
use AppBundle\Entity\Genus;
... lines 6 - 8
class LoadFixtures implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        $genus = new Genus();
        $genus->setName('Octopus'.rand(1, 100));
        $genus->setSubFamily('Octopodinae');
        $genus->setSpeciesCount(rand(100, 99999));

        $manager->persist($genus);
        $manager->flush();
    }
}
```

The $manager argument passed to this function is the entity manager. Use it to persist $genus and don't forget the Genus use statement. Oh, and only one namespace - whoops!

I know this is *not* very interesting yet - stay with me. To run this, head over to the terminal and run:

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

This clears out the database and runs all of our fixture classes - we only have 1. Now, head back to the list page. Here is our *one* random genus. So it's kind of cool... but I know - totally underwhelming. Enter Alice: she makes fixtures fun again.

Chapter 10: Delightful Dummy Data with Alice

Now things are about to get fun. A few minutes ago, we installed a library called nelmio/alice - search for that and find their GitHub page.

In a nutshell, this library lets us add fixtures data via YAML files. It has an expressive syntax *and* it ships with a bunch of built-in functions for generating random data. Actually, *it* uses yet *another* library behind the scenes called Faker to do that. It's the PHP circle of life!

Creating the Fixture YAML File

Find the ORM directory and create a new file called - how about fixtures.yml. That filename lacks excitement, but at least it's clear.

Start with the class name you want to create - AppBundle\Entity\Genus. Next, each genus needs an internal, unique name - it could be anything. But wait! Finish the name with 1..10:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
... lines 3 - 7
```

With this syntax, Alice will loop over and create *10* Genus objects for free. Boom!

To finish things, set values on each of the Genus properties: name: <name()>. You could just put any value here, but when using <>, you're calling a built-in *Faker* function. Next, use subFamily: <text(20)> to generate 20 characters of random text, speciesCount: <numberBetween(100, 100000)> and funFact: <sentence()>:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
        name: <name()>
        subFamily: <text(20)>
        speciesCount: <numberBetween(100, 100000)>
        funFact: <sentence()>
```

That's it team! To *load* this file, open up LoadFixtures and remove all of that boring garbage. Replace it with Fixtures - autocomplete that to get the use statement - then ::load(). Pass this __DIR__.'/fixtures.yml' and then the entity manager:

16 lines | src/AppBundle/DataFixtures/ORM/LoadFixtures.php

```
... lines 1 - 7
use Nelmio\Alice\Fixtures;

class LoadFixtures implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        $objects = Fixtures::load(__DIR__.'/fixtures.yml', $manager);
    }
}
```

Now, run the *exact* command as before:

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

I *love* when there are no errors. Refresh the list page. Voila: 10 completely random genuses. I *love* Alice.

All The Faker Functions

Well.... the genus name is actually the name of a person... which is pretty ridiculous. Let's fix that in a second.

But first, Nelmio's documentation has a *ton* of cool examples of things you can do with this library. But the *biggest* things you'll want to check out is the *Faker* library that this integrates. This shows you *all* of the built-in functions we were just using - like numberBetween, word, sentence and a ton more. There is some *great* stuff in here.

Now if we can *just* make the genus name a little more realistic.

Chapter 11: Custom Alice Faker Function

The Faker name() function gives us a *poor* genus name. Yea, I know - this is just *fake* data - but it's *so* wrong that it fails at its one job: to give us some *somewhat* realistic data to make development easy.

Here's our goal: use a new <genus()> function in Alice and have *this* return the name of a random ocean-bound genus:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
        name: <genus()>
... lines 4 - 7
```

This shouldn't work yet - but try it to see the error:

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

```
Unknown formatter "genus"
```

Faker calls these functions "formatters". Can we create our own formatter? Absolutely.

Adding a Custom Formatter (Function)

In LoadFixtures, break the load() call onto multiple lines to keep things short and civilized. Now, add a third argument - it's sort of an "options" array. Give it a key called providers - these will be additional objects that *provide* formatter functions - and set it to an array with $this:

46 lines | src/AppBundle/DataFixtures/ORM/LoadFixtures.php

```
... lines 1 - 9
class LoadFixtures implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        $objects = Fixtures::load(
            __DIR__.'/fixtures.yml',
            $manager,
            [
                'providers' => [$this]
            ]
        );
    }
... lines 22 - 45
}
```

And we're nearly done! To add a new genus formatter, add public function genus(). I've already prepared a lovely list of some fantastic genuses that live in the ocean:

46 lines | src/AppBundle/DataFixtures/ORM/LoadFixtures.php

```
... lines 1 - 9
class LoadFixtures implements FixtureInterface
{
... lines 12 - 22
    public function genus()
    {
        $genera = [
            'Octopus',
            'Balaena',
            'Orcinus',
            'Hippocampus',
            'Asterias',
            'Amphiprion',
            'Carcharodon',
            'Aurelia',
            'Cucumaria',
            'Balistoides',
            'Paralithodes',
            'Chelonia',
            'Trichechus',
            'Eumetopias'
        ];
... lines 41 - 44
    }
}
```

Finish this with $key = array_rand($genera) and then return $genera[$key]:

46 lines | src/AppBundle/DataFixtures/ORM/LoadFixtures.php

```
... lines 1 - 9
class LoadFixtures implements FixtureInterface
{
... lines 12 - 22
    public function genus()
    {
... lines 25 - 41
        $key = array_rand($genera);

        return $genera[$key];
    }
}
```

Let's give it a try:

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

No errors! Refresh! Ah, *so* much better.

New Random Boolean Column

Now, hold on, we have a *new* requirement: we need the ability to have published and *unpublished* genuses - for those times when we create a new genus, but we're still trying to think of a fun fact before it shows up on the site. With our beautiful migration and fixtures systems, this will be a breeze.

First, open Genus and add a new private property - call it $isPublished. Next, use the "Code"->"Generate" shortcut - or Ctrl+Enter - to generate the annotations:

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

```
... lines 1 - 10
class Genus
{
... lines 13 - 39
    /**
     * @ORM\Column(type="boolean")
     */
    private $isPublished = true;
... lines 44 - 89
    public function setIsPublished($isPublished)
    {
        $this->isPublished = $isPublished;
    }
}
```

Hey that was cool! Because the property started with is, PhpStorm correctly guessed that this is a boolean column. Go team!

At the bottom, generate *just* the setter function. We can add a getter function later... if we need one.

We need to update the fixtures. But first, find the command line and generate the migration:

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

Be a responsible dev and make sure the migration looks right:

35 lines | app/DoctrineMigrations/Version20160207083347.php

```
... lines 1 - 10
class Version20160207083347 extends AbstractMigration
{
    /**
     * @param Schema $schema
     */
    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 ADD is_published TINYINT(1) NOT NULL');
    }

    /**
     * @param Schema $schema
     */
    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 DROP is_published');
    }
}
```

Actually, it looks *perfect*. Run it:

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

Last step: we want to have a *few* unpublished genuses in the random data set. Open the Faker documentation and search for "boolean". Perfect! There's a built-in boolean() function *and* we can control the $chanceOfGettingTrue. In the fixtures file, add isPublished and set that to boolean(75) - so that *most* genuses are published:

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

```
AppBundle\Entity\Genus:
    genus_{1..10}:
... lines 3 - 6
        isPublished: <boolean(75)>
```

Re-run the fixtures!

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

Hey, no errors! Now, to only show the published genuses on the list page, we need a custom query.

Chapter 12: Custom Queries

Time to put that *lazy* isPublished field to work. I *only* want to show *published* genuses on the list page. Up until now, we've been the lazy ones - by using findAll() to return *every* Genus object. We've avoided writing queries.

There *are* a few other methods besides findAll() that you can use to customize things a bit, but look: someday we're going to need to grow up and write a custom query. It's time to grow up.

What is the Repository?

To query, we always use this repository object. But, uh, what *is* that object anyways? Be curious and dump $em->getRepository('AppBundle:Genus) to find out:

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 33
    public function listAction()
    {
        $em = $this->getDoctrine()->getManager();

        dump($em->getRepository('AppBundle:Genus'));
... lines 39 - 44
    }
... lines 46 - 99
}
```

Refresh! I didn't add a die statement - so the dump is playing hide-and-seek down in the web debug toolbar. Ah, it turns out this is an EntityRepository object - something from the core of Doctrine. And *this* class has the helpful methods on it - like findAll() and findOneBy().

Ok, wouldn't it be sweet if we could add *more* methods to this class - like findAllPublished()? Well, I think it would be cool. So let's do it!

Creating your own Repository

No no, not by hacking Doctrine's core files: we're going to create our *own* repository class. Create a new directory called Repository. Inside, add a new class - GenusRepository. None of these names are important. Keep the class empty, but make it extend that EntityRepository class so that we *still* have the original helpful methods:

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

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

use Doctrine\ORM\EntityRepository;

class GenusRepository extends EntityRepository
{

}
```

Next, we need to tell Doctrine to use *this* class instead when we call getRepository(). To do that, open Genus. At the top, @ORM\Entity is empty. Add parentheses, repositoryClass=, then the full class name to the new GenusRepository:

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

```
... lines 1 - 6
/**
 * @ORM\Entity(repositoryClass="AppBundle\Repository\GenusRepository")
 * @ORM\Table(name="genus")
 */
class Genus
... lines 12 - 95
```

That's it! Refresh! *Now* the dump shows a GenusRepository object. And *now* we can start adding custom functions that make custom queries. So, each entity that needs a custom query will have its own repository class. And every custom query you write will live inside of these repository classes. That's going to keep your queries *super* organized.

Adding a Custom Query

Add a new public function called findAllPublishedOrderedBySize():

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

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
... lines 10 - 12
    public function findAllPublishedOrderedBySize()
    {
... lines 15 - 20
    }
}
```

I'm following Doctrine's naming convention of findAllSOMETHING for an array – or findSOMETHING for a single result.

Fortunately, custom queries always look the same: start with, return $this->createQueryBuilder('genus'):

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

```
... lines 1 - 7
class GenusRepository extends EntityRepository
{
... lines 10 - 12
    public function findAllPublishedOrderedBySize()
    {
        return $this->createQueryBuilder('genus')
... lines 16 - 20
    }
}
```

This returns a QueryBuilder. His favorite things are pizza and helping you easily write queries. Because we're *in* the GenusRepository, the query already knows to select from that table. The genus part is the table alias - it's like in MySQL when you say SELECT * FROM genus g - in that case g is an alias you can use in the rest of the query. I like to make my aliases a little more descriptive.

WHERE

To add a WHERE clause, chain ->andWhere() with genus.isPublished = :isPublished:

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

```
... lines 1 - 14
        return $this->createQueryBuilder('genus')
            ->andWhere('genus.isPublished = :isPublished')
... lines 17 - 23
```

I know: the :isPublished looks weird - it's a parameter, like a placeholder. To fill it in, add ->setParameter('isPublished', true):

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

```
... lines 1 - 14
15          return $this->createQueryBuilder('genus')
16              ->andWhere('genus.isPublished = :isPublished')
17              ->setParameter('isPublished', true)
... lines 18 - 23
```

We always set variables like this using parameters to avoid SQL injection attacks. Never concatenate strings in a query.

ORDER BY

To order... well you can kind of guess. Add ->orderBy() with genus.speciesCount and DESC:

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

```
... lines 1 - 14
15          return $this->createQueryBuilder('genus')
16              ->andWhere('genus.isPublished = :isPublished')
17              ->setParameter('isPublished', true)
18              ->orderBy('genus.speciesCount', 'DESC')
... lines 19 - 23
```

Query, done!

Finishing the Query

To execute the query, add ->getQuery() and then ->execute():

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

```
... lines 1 - 14
15          return $this->createQueryBuilder('genus')
16              ->andWhere('genus.isPublished = :isPublished')
17              ->setParameter('isPublished', true)
18              ->orderBy('genus.speciesCount', 'DESC')
19              ->getQuery()
20              ->execute();
... lines 21 - 23
```

That's it! Your query will always end with either execute() - if you want an *array* of results - or getOneOrNullResult() - if you want just *one* result... or obviously null if nothing is matched.

Let's really show off by adding some PHP doc above the method. Oh, we can do better than @return mixed! *We* know this will return an array of Genus objects - so use Genus[]:

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

```
... lines 1 - 4
use AppBundle\Entity\Genus;
... lines 6 - 7
class GenusRepository extends EntityRepository
{
    /**
     * @return Genus[]
     */
    public function findAllPublishedOrderedBySize()
    {
... lines 15 - 20
    }
}
```

Using the Custom Query

Our hard work is done - using the new method is simple. Replace findAll() with findAllPublishedOrderedBySize():

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

```
... lines 1 - 11
class GenusController extends Controller
{
... lines 14 - 33
    public function listAction()
    {
... lines 36 - 37
        $genuses = $em->getRepository('AppBundle:Genus')
            ->findAllPublishedOrderedBySize();
... lines 40 - 43
    }
... lines 45 - 98
}
```

Go back, refresh... and there it is! A few disappeared because they're unpublished. And the genus with the most species is first. Congrats!

We have an entire tutorial on doing crazy custom queries in Doctrine. So if you want to start selecting only a few columns, using raw SQL or doing really complex joins, check out the Go Pro with Doctrine Queries.

Woh guys - we just *crushed* all the Doctrine basics - go build something cool and tell me about it. There's just *one* big topic we *didn't* cover - relationships. These are *beautiful* in Doctrine, but there's a lot of confusing and over-complicated information about there. So let's master that in the next tutorial. Seeya guys next time!