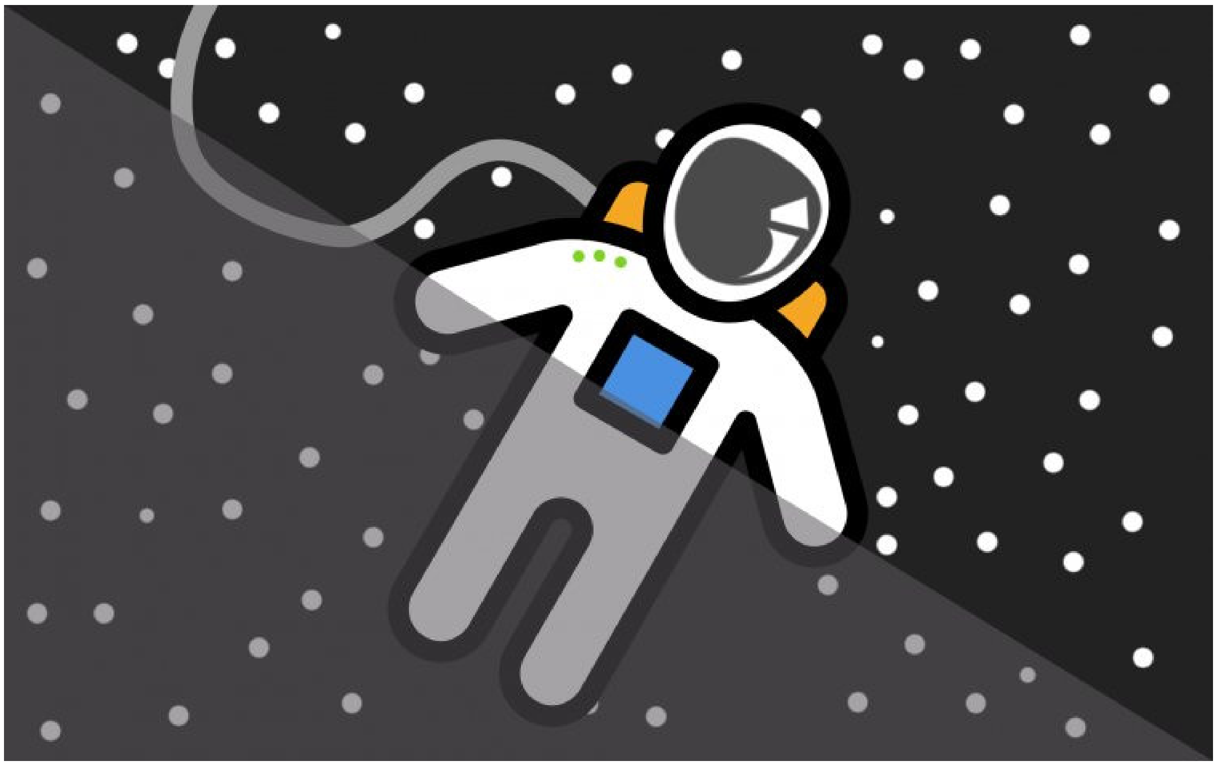
Doctrine & the Database



With <3 from SymfonyCasts

Chapter 1: Installing Doctrine

Friends! Welcome back to the *third* episode in our starting in Symfony 4 series! We've done some really cool stuff already, but it's time to explore deeper in an epic search for intelligent life... and also.. and database! Yea - what good is our cool interstellar space news site... without being able to insert and query for data?

But... actually... Symfony does *not* have a database layer. Nope, for this challenge, we're going to rely one of Symfony's BFF's: an external library called Doctrine. Doctrine has *great* integration with Symfony and is *crazy* powerful. It *also* has a reputation for being a little bit hard to learn. But, a lot has improved over the last few years.

Code with Me!

If you *also* want to be best-friends-forever with Doctrine, you should *totally* code along with me. Download the course code from this page. When you unzip it, you'll find a start/ directory that has the same code that you see here. Open the README.md file for details on how to get the project setup... and of course, a space poem.

The last step will be to open a terminal, move into the project and run:

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

to start the built in web server. Then, float over to your browser, and open http://localhost:8000 to discover... The Space Bar! Our inter-planetary, and extraterrestrial news site that spreads light on dark matters everywhere.

In the first two episodes, we already added some pretty cool stuff! But, these articles are still just hard-coded. Time to change that.

Installing Doctrine

Because Doctrine is an external library, before we do *anything* else, we need to install it! Thanks to Symfony flex, this is super easy. Open a new terminal tab and just run:

```
$ composer require doctrine
```

This will download a "pack" that contains a few libraries, including doctrine itself and also a migrations library to help manage database changes on production. More on that soon.

And... done! Hey! That's a nice message. Because we're going to be talking to a database, obviously, we will need to configure our database details somewhere. The message tells us that - no surprise - this is done in the .env file.

Configuring the Database Connection

Move over to your code and open the .env file. Nice! The DoctrineBundle recipe added a new DATABASE_URL environment variable:

22 lines | .env.dist

```
... lines 1 - 15
###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a
# For an SQLite database, use: "sqlite:///%kernel.project_dir%/var/data.db"
# Configure your db driver and server_version in config/packages/doctrine.yaml
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
###
```

Let's set this up: I use a root user with no password locally. Call the database the_spacebar:

22 lines | .env.dist

```
... lines 1 - 15
###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a
# For an SQLite database, use: "sqlite:///%kernel.project_dir%/var/data.db"
# Configure your db driver and server_version in config/packages/doctrine.yaml
DATABASE_URL=mysql://root:@127.0.0.1:3306/symfony4_space_bar
###
```

Of course, this sets a DATABASE_URL environment variable. And *it* is used in a new config/packages/doctrine.yaml file that was installed by the recipe. If you scroll down a bit... you can see the environment variable being used:

31 lines | config/packages/doctrine.yaml

```
parameters:
    # Adds a fallback DATABASE_URL if the env var is not set.
    # This allows you to run cache:warmup even if your
    # environment variables are not available yet.
    # You should not need to change this value.
    env(DATABASE_URL): ''

doctrine:
    dbal:
... lines 10 - 17
        # With Symfony 3.3, remove the `resolve:` prefix
        url: '%env(resolve:DATABASE_URL)%'
... lines 20 - 31
```

There are actually a lot of options in here, but you probably won't need to change any of them. These give you nice defaults, like using UTF8 tables:

31 lines | config/packages/doctrine.yaml

```
... lines 1 - 7
doctrine:
    dbal:
... lines 10 - 12
        charset: utf8mb4
... lines 14 - 31
```

Or, consistently using underscores for table and column names:

31 lines | config/packages/doctrine.yaml

```
... lines 1 - 7
doctrine:
... lines 9 - 19
    orm:
... line 21
        naming_strategy: doctrine.orm.naming_strategy.underscore
... lines 23 - 31
```

If you want to use something other than MySQL, you can easily change that. Oh, and you should set your server_version to the server version of MySQL that you're using on production:

```
31 lines | config/packages/doctrine.yaml
... lines 1 - 7
doctrine:
  dbal:
    # configure these for your database server
    driver: 'pdo_mysql'
    server_version: '5.7'
... lines 13 - 31
```

This helps Doctrine with a few subtle, version-specific changes.

Creating the Database

And... yea! With one composer require command and one line of config, we're setup! Doctrine can even create the database for you. Go back to your terminal and run

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

Say hello to your new database! Well, it's not *that* interesting: it's completely empty.

So let's add a table, by creating an entity class.

Chapter 2: Creating an Entity Class

Doctrine is an ORM, or object relational mapper. A fancy term for a pretty cool idea. It means that each table in the database will have a corresponding *class* in our code. So if we want to create an article table, it means that we need to create an Article *class*. You can *totally* make this class by hand - it's just a normal PHP class.

Generating with make:entity

But there's a *really* nice generation tool from MakerBundle. We installed MakerBundle in the last tutorial, and before I started coding, I updated it to the latest version to get this new command. At your terminal, run:

```
$ php bin/console make:entity
```

Stop! That word "entity": that's important. This is the word that Doctrine gives to the classes that are saved to the database. As you'll see in a second, these are just normal PHP classes. So, when you hear "entity", think:

> That's a normal PHP class that I can save to the database.

Let's call our class Article, and then, cool! We can start giving it fields right here. We need a title field. For field "type", hmm, hit "?" to see what *all* the different types are.

Notice, these are *not* MySQL types, like varchar. Doctrine has its *own* types that map to MySQL types. For example, let's use "string" and let the length be 255. Ultimately, that'll create a varchar column. Oh, and because we probably want this column to be *required* in the database, answer "no" for nullable.

Next, create a field called slug, use the string type again, and let's make it's length be 100, and no for nullable.

Next, content, set this to text and "yes" to nullable: maybe we allow articles to be drafted without content at first. And finally, a publishedAt field with a type set to datetime and yes to nullable. If this field is null, we'll know that the article has *not* been published.

When you're done, hit enter to finish. And don't worry if you make a mistake. You can always update things later, or delete the new entity class and start over.

Investigating the Entity Class

So... what did that just do? Only *one* thing: in src/Entity, this command generated a new Article class:

93 lines | src/Entity/Article.php

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

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
 */
class Article
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;
    /**
     * @ORM\Column(type="string", length=255)
     */
    private $title;

    /**
     * @ORM\Column(type="string", length=100)
     */
    private $slug;

    /**
     * @ORM\Column(type="text", nullable=true)
     */
    private $content;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $publishedAt;

    public function getId()
    {
        return $this->id;
    }

    public function getTitle(): ?string
    {
        return $this->title;
    }

    public function setTitle(string $title): self
    {
        $this->title = $title;

        return $this;
    }


    public function getSlug(): ?string
    {
        return $this->slug;
    }

    public function setSlug(string $slug): self
    {
        $this->slug = $slug;

        return $this;
    }

    public function getContent(): ?string
    {
        return $this->content;
    }

    public function setContent(?string $content): self
    public function setContent(?string $content): self
    {
        $this->content = $content;

        return $this;
    }

    public function getPublishedAt(): ?\DateTimeInterface
    {
        return $this->publishedAt;
    }

    public function setPublishedAt(?\DateTimeInterface $publishedAt): self
    {
        $this->publishedAt = $publishedAt;

        return $this;
    }
}
```

Well... to be fully honest, there is also a new ArticleRepository class, but I want you to ignore that for now. It's not important yet.

Anyways, this Article class is your *entity*. And, check it out! It's a normal, boring PHP class with a property for each column: id, title, slug, content, and publishedAt:

93 lines | src/Entity/Article.php

```
... lines 1 - 9
class Article
{
... lines 12 - 16
    private $id;
... lines 18 - 21
    private $title;
... lines 23 - 26
    private $slug;
... lines 28 - 31
    private $content;
... lines 33 - 36
    private $publishedAt;
... lines 38 - 91
}
```

What makes this class *special* are the annotations! The @ORM\Entity above the class tells Doctrine that this is an entity that should be mapped to the database:

93 lines | src/Entity/Article.php

```
... lines 1 - 4
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
 */
class Article
{
... lines 12 - 91
}
```

Then, above each property, we have some annotations that help doctrine know how to store that exact column:

```
93 lines | src/Entity/Article.php
... lines 1 - 4
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
 */
class Article
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $title;

    /**
     * @ORM\Column(type="string", length=100)
     */
    private $slug;

    /**
     * @ORM\Column(type="text", nullable=true)
     */
    private $content;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $publishedAt;
... lines 38 - 91
}
```

Actually, find your browser and Google for "doctrine annotations reference" to find a cool page. This shows you *every* annotation in Doctrine and *every* option for each one.

Back at the code, the properties are *private*. So, at the bottom of the class, the command generated getter and setter methods for each one:

```
93 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 38
    public function getId()
    {
        return $this->id;
    }

    public function getTitle(): ?string
    public function getTitle(): ?string
    {
        return $this->title;
    }

    public function setTitle(string $title): self
    {
        $this->title = $title;

        return $this;
    }


    public function getSlug(): ?string
    {
        return $this->slug;
    }

    public function setSlug(string $slug): self
    {
        $this->slug = $slug;

        return $this;
    }

    public function getContent(): ?string
    {
        return $this->content;
    }

    public function setContent(?string $content): self
    {
        $this->content = $content;

        return $this;
    }

    public function getPublishedAt(): ?\DateTimeInterface
    {
        return $this->publishedAt;
    }

    public function setPublishedAt(?\DateTimeInterface $publishedAt): self
    {
        $this->publishedAt = $publishedAt;

        return $this;
    }
}
```

There's one *really* important thing to realize: this class is 100% *your* class. Feel free to add, remove or rename any properties or methods you want.

And... yea! With one command, our entity is ready! But, the database is still empty! We need to tell Doctrine to create the corresponding article table in the database. We do this with migrations.

Chapter 3: Database Migrations

The Article entity is ready, and Doctrine already knows to save its data to an article table in the database. But... that table doesn't exist yet! So... how can we create it?

Generating a Migration

Ah, this is one of Doctrine's *superpowers*. Go back to your terminal. At the bottom of the make:entity command, it has a suggestion: run the make:migration command.

I *love* this! Try it:

```
$ php bin/console make:migration
```

The output says that it created a new src/Migrations/Version* class that we should review. Ok, find your code, open the Migrations directory and, there it is! One migration file:

29 lines | src/Migrations/Version20180413174059.php

```
... lines 1 - 2
namespace DoctrineMigrations;

use Doctrine\DBAL\Migrations\AbstractMigration;
use Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20180413174059 extends AbstractMigration
{
    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 \'mys

        $this->addSql('CREATE TABLE article (id INT AUTO_INCREMENT NOT NULL, title VARCHAR(255) NOT NULL, slug VARCHAR
    }

    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 \'mys

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

Inside, cool! It holds the MySQL code that we need!

> CREATE TABLE article...

This is *amazing*. No, seriously - it's *way* more awesome than you might think. The make:migration command actually *looked* at our database, looked at all of our entity classes - which is just one entity right now - and generated the SQL needed to *update* the database to match our entities. I'll show you an even better example in a few minutes.

Executing the Migration

This looks good to me, so close it and then go back to your terminal. To execute the migration, run:

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

This command was *also* suggested above. Answer yes to run the migrations and... done!

But now, run that same command again:

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

How Migrations Work

It does nothing! Interesting. Run:

```
$ php bin/console doctrine:migrations:status
```

Ok, this tells us a bit more about *how* the migration system works. Inside the database, the migration system automatically creates a new table called migration_versions. Then, the *first* time we ran doctrine:migrations:migrate, it executed the migration, and inserted a new *row* in that table with that migration's version number, which is the date in the class name. When we ran doctrine:migrations:migrate a *second* time, it opened the migration class, then looked up that version in the migration_versions table. Because it was already there, it knew that this migration had already been executed and did *not* try to run it again.

This is brilliant! Whenever we need to make a database change, we follow this simple two-step process: (1) Generate the migration with make:migration and (2) run that migration with doctrine:migrations:migrate. We *will* commit the migrations to our git repository. Then, on deploy, just make sure to run doctrine:migrations:migrate. The production database will have its *own* migration_versions table, so this will automatically run *all* migrations that have not been run yet on production. It's perfect.

Migration a Second Change

To see how nice this is, let's make one more change. Open the Article class. See the slug field?

93 lines | src/Entity/Article.php

```
... lines 1 - 6
/**
 * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
 */
class Article
{
... lines 12 - 23
    /**
     * @ORM\Column(type="string", length=100)
     */
    private $slug;
... lines 28 - 91
}
```

This will eventually be used to identify the article in the URL. And so, this *must* be *unique* across every article in the table.

To *guarantee* that this is unique in the database, add unique=true:

93 lines | src/Entity/Article.php

```
... lines 1 - 6
/**
 * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
 */
class Article
{
... lines 12 - 23
    /**
     * @ORM\Column(type="string", length=100, unique=true)
     */
    private $slug;
... lines 28 - 91
}
```

This option does only *one* thing: it tells Doctrine that it should create a unique *index* in the database for this column.

But of course, the database didn't just magically update to have this index. We need a migration. No problem! Find your terminal and do step 1: run:

```
$ php bin/console make:migration
```

Ha! I even misspelled the command: Symfony figured out what I meant. This created a *second* migration class: the first creates the table and the second... awesome! It creates the unique index:

29 lines | src/Migrations/Version20180413174154.php

```
... lines 1 - 2
namespace DoctrineMigrations;

use Doctrine\DBAL\Migrations\AbstractMigration;
use Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20180413174154 extends AbstractMigration
{
    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 \'mys

        $this->addSql('CREATE UNIQUE INDEX UNIQ_23A0E66989D9B62 ON article (slug)');
    }

    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 \'mys

        $this->addSql('DROP INDEX UNIQ_23A0E66989D9B62 ON article');
    }
}
```

This is the Doctrine magic I mentioned earlier: the make:migration command looked at the entity, looked at the database, determined the *difference* between the two, then generated the SQL necessary to *update* the database.

Now, for step (2), run:

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

It sees the *two* migration classes, notices that the *first* has already been executed, and only runs the *second*.

Ok! Our database is setup, our Article entity is ready, and we already have a killer migration system. So let's talk about how to *save* articles to the table.

Chapter 4: Saving Entities

Put on your publishing hat, because it's time to write some thoughtful space articles and insert some rows into our article table! And, good news! This is probably one of the *easiest* things to do in Doctrine.

Let's create a new controller called ArticleAdminController. We'll use this as a place to add new articles. Make it extend the normal AbstractController:

```
19 lines | src/Controller/ArticleAdminController.php
... lines 1 - 2
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
... lines 6 - 8
class ArticleAdminController extends AbstractController
{
... lines 11 - 17
}
```

And create a public function new():

```
19 lines | src/Controller/ArticleAdminController.php
... lines 1 - 8
class ArticleAdminController extends AbstractController
{
... lines 11 - 13
    public function new()
    {
... line 16
    }
}
```

Above, add the @Route() - make sure to auto-complete the one from Symfony Components so that PhpStorm adds the use statement. For the URL, how about /admin/article/new:

```
19 lines | src/Controller/ArticleAdminController.php
... lines 1 - 6
use Symfony\Component\Routing\Annotation\Route;

class ArticleAdminController extends AbstractController
{
    /**
     * @Route("/admin/article/new")
     */
    public function new()
    {
... line 16
    }
}
```

We're not *actually* going to build a real page with a form here right now. Instead, I just want to write some code that saves a dummy article to the database.

But first, to make sure I haven't screwed anything up, return a new Response: the one from HttpFoundation with a message:

space rocks... include comets, asteroids & meteoroids

```
19 lines | src/Controller/ArticleAdminController.php
... lines 1 - 5
use Symfony\Component\HttpFoundation\Response;
... lines 7 - 8
class ArticleAdminController extends AbstractController
{
    /**
     * @Route("/admin/article/new")
     */
    public function new()
    {
        return new Response('space rocks... include comets, asteroids & meteoroids');
    }
}
```

Now, we *should* be able to find the browser and head to /admin/article/new. Great!

Creating the Article Object

So, here's the big question: *how* do you save data to the database with Doctrine? The answer... is beautiful: just create an Article object with the data you need, then ask Doctrine to put it into the database.

Start with $article = new Article():

```
48 lines | src/Controller/ArticleAdminController.php
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 9
class ArticleAdminController extends AbstractController
{
... lines 12 - 14
    public function new()
    {
        $article = new Article();
... lines 18 - 45
    }
}
```

For this article's data, go back to the "Why Asteroids Taste like Bacon" article: we'll use this as our dummy news story. Copy the article's title, then call $article->setTitle() and paste:

```
48 lines | src/Controller/ArticleAdminController.php
... lines 1 - 9
class ArticleAdminController extends AbstractController
{
... lines 12 - 14
    public function new()
    {
        $article = new Article();
        $article->setTitle('Why Asteroids Taste Like Bacon')
... lines 19 - 45
    }
}
```

This is one of the setter methods that was automatically generated into our entity:

93 lines | src/Entity/Article.php

```
... lines 1 - 9
class Article
{
... lines 12 - 18
    /**
     * @ORM\Column(type="string", length=255)
     */
    private $title;
... lines 23 - 48
    public function setTitle(string $title): self
    {
        $this->title = $title;

        return $this;
    }
... lines 55 - 91
}
```

Oh, and the generator *also* made all the setter methods return $this, which means you can chain your calls, like: ->setSlug(), then copy the last part of the URL, and paste here. Oh, but we need to make sure this is unique... so just add a little random number at the end:

48 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 9
class ArticleAdminController extends AbstractController
{
... lines 12 - 14
    public function new()
    {
        $article = new Article();
        $article->setTitle('Why Asteroids Taste Like Bacon')
            ->setSlug('why-asteroids-taste-like-bacon-'.rand(100, 999))
... lines 20 - 45
    }
}
```

Then, ->setContent(). And to get this, go back to ArticleController, copy *all* of that meaty markdown and paste here. Ah, make sure the content is completely *not* indented so the multi-line text works:

```
48 lines | src/Controller/ArticleAdminController.php
... lines 1 - 9
class ArticleAdminController extends AbstractController
{
... lines 12 - 14
    public function new()
    {
        $article = new Article();
        $article->setTitle('Why Asteroids Taste Like Bacon')
            ->setSlug('why-asteroids-taste-like-bacon-'.rand(100, 999))
            ->setContent(<<<EOF
Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
**turkey** shank eu pork belly meatball non cupim.

Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
occaecat lorem meatball prosciutto quis strip steak.

Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
fugiat.
EOF
        );
... lines 39 - 45
    }
}
```

Much better! The last field is publishedAt. To have more interesting data, let's only publish *some* articles. So, if a random number between 1 to 10 is greater than 2, publish the article: $article->setPublishedAt() with new \DateTime() and sprintf('-%d days') with a bit more randomness: 1 to 100 days old:

```
48 lines | src/Controller/ArticleAdminController.php
... lines 1 - 9
class ArticleAdminController extends AbstractController
{
... lines 12 - 14
    public function new()
    {
        $article = new Article();
        $article->setTitle('Why Asteroids Taste Like Bacon')
            ->setSlug('why-asteroids-taste-like-bacon-'.rand(100, 999))
            ->setContent(<<<EOF
Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
**turkey** shank eu pork belly meatball non cupim.

Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
occaecat lorem meatball prosciutto quis strip steak.

Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
fugiat.
EOF
        );

        // publish most articles
        if (rand(1, 10) > 2) {
            $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
        }

        return new Response('space rocks... include comets, asteroids & meteoroids');
    }
}
```

Perfect! Now... stop. I want you to notice that *all* we've done is create an Article object and set data on it. This is normal, boring, PHP code: we're not using Doctrine at *all* yet. That's really cool.

Saving the Article

To *save* this, we just need to find Doctrine and say:

> Hey Doctrine! Say hi to Jon Wage for us! Also, can you please save this article to the database. You're the best!

How do we do this? In the last Symfony tutorial, we talked about how the *main* thing that a bundle gives us is more *services*. DoctrineBundle gives us one, *very* important service that's used for both saving to *and* fetching from the database. It's called the DeathStar. No, no, it's the EntityManager. But, missed opportunity...

Find your terminal and run:

```
$ php bin/console debug:autowiring
```

Scroll to the the top. There it is! EntityManagerInterface: that's the type-hint we can use to fetch the service. Go back to the top of the new() method and add an argument: EntityManagerInterface $em:

56 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 5
use Doctrine\ORM\EntityManagerInterface;
... lines 7 - 10
class ArticleAdminController extends AbstractController
{
... lines 13 - 15
    public function new(EntityManagerInterface $em)
    {
... lines 18 - 53
    }
}
```

Now that we have the all-important entity manager, saving is a two-step process... and it *may* look a bit weird initially. First, $em->persist($article), then $em->flush():

56 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 5
use Doctrine\ORM\EntityManagerInterface;
... lines 7 - 10
class ArticleAdminController extends AbstractController
{
... lines 13 - 15
    public function new(EntityManagerInterface $em)
    {
... lines 18 - 40
        // publish most articles
        if (rand(1, 10) > 2) {
            $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
        }

        $em->persist($article);
        $em->flush();
... lines 48 - 53
    }
}
```

It's *always* these two lines. Persist simply says that you would *like* to save this article, but Doctrine does *not* make the INSERT query yet. That happens when you call $em->flush(). Why two separate steps? Well, it gives you a bit more flexibility: you could create ten Article objects, called persist() on each, then flush() just *one* time at the end. This helps Doctrine optimize saving those ten articles.

At the bottom, let's make our message a bit more helpful, though, I thought my message about space rocks was *at least* educational. Set the article id to some number and the slug to some string. Pass: $article->getId() and $article->getSlug():

```
56 lines | src/Controller/ArticleAdminController.php
... lines 1 - 5
use Doctrine\ORM\EntityManagerInterface;
... lines 7 - 10
class ArticleAdminController extends AbstractController
{
... lines 13 - 15
    public function new(EntityManagerInterface $em)
    {
... lines 18 - 45
        $em->persist($article);
        $em->flush();

        return new Response(sprintf(
            'Hiya! New Article id: #%d slug: %s',
            $article->getId(),
            $article->getSlug()
        ));
    }
}
```

Oh, and this is important: *we* never set the id. But when we call flush(), Doctrine will insert the new row, get the new id, and put that onto the Article *for* us. By the time we print this message, the Article will have its new, fancy id.

Ok, are you ready? Let's try it: go back to /admin/article/new and... ha! Article id 1, then 2, 3, 4, 5, 6! Our news site is alive!

If you want to be *more* sure, you can check this in your favorite database tool like phpMyAdmin or whatever the cool kids are using these days. *Or*, you can use a helpful console command:

```
$ php bin/console doctrine:query:sql "SELECT * FROM article"
```

This is article with a *lowercase* "a", because, thanks to the default configuration, Doctrine creates snake case table and column names.

And... yes! There are the new, 6 results.

We have successfully put stuff *into* the database! Now it's time to run some queries to fetch it back out.

Chapter 5: Querying for Data!

Hey! There are rows in our article table! So let's update the news page to *not* show this hard-coded article, but instead to query the database and print *real*, dynamic data.

Open ArticleController and find the show() method:

```
90 lines | src/Controller/ArticleController.php
... lines 1 - 13
class ArticleController extends AbstractController
{
... lines 16 - 33
    /**
     * @Route("/news/{slug}", name="article_show")
     */
    public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack)
    {
        if ($slug === 'khaaaaaan') {
            $slack->sendMessage('Kahn', 'Ah, Kirk, my old friend...');
        }

        $comments = [
            'I ate a normal rock once. It did NOT taste like bacon!',
            'Woohoo! I\'m going on an all-asteroid diet!',
            'I like bacon too! Buy some from my site! bakinsomebacon.com',
        ];

        $articleContent = <<<EOF
Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
**turkey** shank eu pork belly meatball non cupim.

Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
occaecat lorem meatball prosciutto quis strip steak.

Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
fugiat.
EOF;

        $articleContent = $markdownHelper->parse($articleContent);

        return $this->render('article/show.html.twig', [
            'title' => ucwords(str_replace('-', ' ', $slug)),
            'slug' => $slug,
            'comments' => $comments,
            'articleContent' => $articleContent,
        ]);
    }
... lines 77 - 88
}
```

This renders that page. As I mentioned earlier, DoctrineBundle gives us one service - the EntityManager - that has the power to save *and* fetch data. Let's get it here: add another argument: EntityManagerInterface $em:

```
101 lines | src/Controller/ArticleController.php
... lines 1 - 7
use Doctrine\ORM\EntityManagerInterface;
... lines 9 - 15
class ArticleController extends AbstractController
{
... lines 18 - 35
    /**
     * @Route("/news/{slug}", name="article_show")
     */
    public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 86
    }
... lines 88 - 99
}
```

When you want to *query* for data, the first step is always the same: we need to get the *repository* for the entity: $repository = $em->getRepository() and then pass the entity class name: Article::class:

```
101 lines | src/Controller/ArticleController.php
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack, EntityManagerInterface $em)
    {
        if ($slug === 'khaaaaaan') {
            $slack->sendMessage('Kahn', 'Ah, Kirk, my old friend...');
        }

        $repository = $em->getRepository(Article::class);
... lines 46 - 86
    }
... lines 88 - 99
}
```

This repository object knows *everything* about how to query from the article table. We can use it to say $article = $repository->. Oh, nice! It has some built-in methods, like find() where you can pass the $id to fetch a single article. Or, findAll() to fetch *all* articles. With the findBy() method, you can fetch *all* articles where a field matches some value. And findOneBy() is the same, but only returns *one* Article. Let's use that: ->findOneBy() and pass it an array with 'slug' => $slug:

```
101 lines | src/Controller/ArticleController.php
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 44
        $repository = $em->getRepository(Article::class);
... line 46
        $article = $repository->findOneBy(['slug' => $slug]);
... lines 48 - 86
    }
... lines 88 - 99
}
```

This will fetch *one* row where the slug field matches this value. These built-in find methods are nice... but they can't do much more than this. But, don't worry! We will *of course* learn how to write custom queries soon.

Above this line, just to help my editor, I'll tell it that this is an Article object:

```
101 lines | src/Controller/ArticleController.php
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 44
        $repository = $em->getRepository(Article::class);
        /** @var Article $article */
        $article = $repository->findOneBy(['slug' => $slug]);
... lines 48 - 86
    }
... lines 88 - 99
}
```

And... hold on, that's important! When you query for something, Doctrine returns *objects*, not just an associative arrays with data. That's really the whole point of Doctrine! You need to stop thinking about inserting and selecting rows in a database. Instead, think about saving and fetching *objects*... almost as if you didn't know that a database was behind-the-scenes.

Handling 404's

At this point, it's *possible* that there is *no* article in the database with this slug. In that case, $article will be null. How should we handle that? Well, in the real world, this should trigger a 404 page. To do that, say if !$article, then, throw $this->createNotFoundException(). Pass a descriptive message, like: No article for slug "%s" and pass $slug:

```
101 lines | src/Controller/ArticleController.php
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 44
        $repository = $em->getRepository(Article::class);
        /** @var Article $article */
        $article = $repository->findOneBy(['slug' => $slug]);
        if (!$article) {
            throw $this->createNotFoundException(sprintf('No article for slug "%s"', $slug));
        }
... lines 51 - 86
    }
... lines 88 - 99
}
```

I want to dig a *little* bit deeper to see how this work. Hold Command on a Mac - or Ctrl otherwise - and click this method. Ah, it comes from a trait that's used by the base AbstractController. Fascinating! It just throws an exception!

In Symfony, to trigger a 404, you just need to throw this very special exception class. That's why, in the controller, we *throw* $this->createNotFoundException(). The message can be as descriptive as possible because it will only be shown to you: the developer.

After all of this, let's dump() the $article to see what it looks like and die:

```
101 lines | src/Controller/ArticleController.php
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, MarkdownHelper $markdownHelper, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 44
        $repository = $em->getRepository(Article::class);
        /** @var Article $article */
        $article = $repository->findOneBy(['slug' => $slug]);
        if (!$article) {
            throw $this->createNotFoundException(sprintf('No article for slug "%s"', $slug));
        }

        dump($article);die;
... lines 53 - 86
    }
... lines 88 - 99
}
```

Head back to your browser and first, refresh. Ok! *This* is the 404 page: there's nothing in the database that matches this slug: all the *real* slugs have a random number at the end. *We* see the helpful error message because *this* is what the 404 page looks like for *developers*. But of course, when you switch into the prod environment, your users will see a different page that you can customize.

We're not going to talk about *how* to customize error pages... because it's super friendly and easy. Just Google for "Symfony customize error pages" and... have fun! You can create separate pages for 404 errors, 403 errors, 500 errors, or whatever your heart desires.

To find a *real* slug, go back to /admin/article/new. Copy that slug, go back, paste it and... it works! There is our *full*, *beautiful*, well-written, inspiring, Article object... with fake content about meat. Having an *object* is *awesome*! We are now... dangerous.

Rendering the Article Data: Twig Magic

Back in the controller, remove the dump():

76 lines | src/Controller/ArticleController.php

```
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 44
        $repository = $em->getRepository(Article::class);
        /** @var Article $article */
        $article = $repository->findOneBy(['slug' => $slug]);
        if (!$article) {
            throw $this->createNotFoundException(sprintf('No article for slug "%s"', $slug));
        }
... lines 51 - 61
    }
... lines 63 - 74
}
```

Keep the hardcoded comments for now. But, remove the $articleContent:

76 lines | src/Controller/ArticleController.php

```
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 44
        $repository = $em->getRepository(Article::class);
        /** @var Article $article */
        $article = $repository->findOneBy(['slug' => $slug]);
        if (!$article) {
            throw $this->createNotFoundException(sprintf('No article for slug "%s"', $slug));
        }

        $comments = [
            'I ate a normal rock once. It did NOT taste like bacon!',
            'Woohoo! I\'m going on an all-asteroid diet!',
            'I like bacon too! Buy some from my site! bakinsomebacon.com',
        ];

        return $this->render('article/show.html.twig', [
... lines 59 - 60
        ]);
    }
... lines 63 - 74
}
```

Let's also remove the markdown parsing code and the now-unused argument:

```
76 lines | src/Controller/ArticleController.php
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 61
    }
... lines 63 - 74
}
```

We'll process the markdown in the template in a minute: Back down at render(), instead of passing title, articleContent and slug, *just* pass article:

```
76 lines | src/Controller/ArticleController.php
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 38
    public function show($slug, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 41 - 57
        return $this->render('article/show.html.twig', [
            'article' => $article,
            'comments' => $comments,
        ]);
    }
... lines 63 - 74
}
```

Now, open that template! With the Symfony plugin, you can cheat and hold Command or Ctrl and click to open it. Or, it's just in templates/article.

Updating the template is a *dream*. Instead of title, print article.title:

```
83 lines | templates/article/show.html.twig
{% extends 'base.html.twig' %}

{% block title %}Read: {{ article.title }}{% endblock %}
... lines 4 - 83
```

Oh, and in *many* cases... but not always... you'll get auto-completion based on the methods on your entity class!

But look closely: it's auto-completing getTitle(). But when I hit tab, it just prints article.title. Behind the scenes, there is some serious Twig magic happening. When you say article.title, Twig first looks to see if the class has a title *property*:

```
93 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 21
    private $title;
... lines 23 - 91
}
```

It does! But since that property is *private*, it can't use it. No worries! It then looks for a getTitle() method. And because *that* exists:

```
93 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 21
    private $title;
... lines 23 - 43
    public function getTitle(): ?string
    {
        return $this->title;
    }
... lines 48 - 91
}
```

It calls it and prints that value.

This is *really* cool because our template code can be simple: Twig figures out what to do. If you were printing a boolean field, something like article.published, Twig would also look for isPublished() a hasPublished() methods. *And*, if article were an array, the dot syntax would just fetch the keys off of that array. Twig: you're the bomb.

Let's update a few more places: article.title, then, article.slug, and finally, for the content, article.content, but then |markdown:

83 lines | templates/article/show.html.twig

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

{% block title %}Read: {{ article.title }}{% endblock %}

{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
                            <span class="show-article-title ">{{ article.title }}</span>
... lines 16 - 18
                            <span class="pl-2 article-details">
... line 20
                                <a href="{{ path('article_toggle_heart', {slug: article.slug}) }}" class="fa fa-heart-o like-article js-like-article"></a>
                            </span>
                        </div>
                    </div>
                </div>
                <div class="row">
                    <div class="col-sm-12">
                        <div class="article-text">
                            {{ article.content|markdown }}
                        </div>
                    </div>
                </div>
... lines 33 - 71
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 78 - 83
```

The KnpMarkdownBundle gives us a markdown filter, so that we can just process it right here in the template.

Ready to try it? Move over, deep breath, refresh. Yes! It works! Hello dynamic title! Hello dynamic bacon content!

See your Queries in the Profiler

Oh, and I have a *wonderful* surprise! The web debug toolbar now has a database icon that tells us how many database queries this page executed and how long they took. But wait, there's more! Click the icon to go into the profiler. Yes! This actually *lists* every query. You can run "EXPLAIN" on each one or view a runnable query. I use this to help debug when a particularly complex query isn't returning the results I expect.

So, um, yea. This is awesome. Next, let's take a quick detour and have some fun by creating a custom Twig filter with a Twig extension. We need to do this, because our markdown processing is *no* longer being cached. Boo.

Chapter 6: Fun with Twig Extensions!

Head back to the article show page because... there's a little, bitty problem that I just introduced. Using the markdown filter from KnpMarkdownBundle works... but the process is not being cached anymore. In the previous tutorial, we created a cool MarkdownHelper that used the markdown object from KnpMarkdownBundle, but added caching so that we don't need to re-parse the *same* markdown content over and over again:

44 lines | src/Service/MarkdownHelper.php

```php
... lines 1 - 2
namespace App\Service;

use Michelf\MarkdownInterface;
use Psr\Log\LoggerInterface;
use Symfony\Component\Cache\Adapter\AdapterInterface;

class MarkdownHelper
{
    private $cache;
    private $markdown;
    private $logger;
    private $isDebug;

    public function __construct(AdapterInterface $cache, MarkdownInterface $markdown, LoggerInterface $markdownLogger, bool $is
    {
        $this->cache = $cache;
        $this->markdown = $markdown;
        $this->logger = $markdownLogger;
        $this->isDebug = $isDebug;
    }

    public function parse(string $source): string
    {
        if (stripos($source, 'bacon') !== false) {
            $this->logger->info('They are talking about bacon again!');
        }

        // skip caching entirely in debug
        if ($this->isDebug) {
            return $this->markdown->transform($source);
        }

        $item = $this->cache->getItem('markdown_'.md5($source));
        if (!$item->isHit()) {
            $item->set($this->markdown->transform($source));
            $this->cache->save($item);
        }

        return $item->get();
    }
}
```

Basically, we want to be able to use a markdown filter in Twig, but we want it to use *our* MarkdownHelper service, instead of the uncached service from the bundle.

So... how can we do this? Let's create our *own* Twig filter, and make it do exactly what we want. We'll call it, cached_markdown.

Generating a Twig Extension

To create a custom function, filter or to extend Twig in any way, you need to create a Twig *extension*. These are *super* fun. Find your terminal and run:

```
$ php bin/console make:twig-extension
```

It suggests the name AppExtension, which I'm actually going to use. I'll call it AppExtension because I typically create just *one* extension class that will hold *all* of the custom Twig functions and filters that I need for my entire project. I do this instead of having multiple Twig extensions... because it's easier.

Let's go check out our new AppExtension file!

```php
30 lines | src/Twig/AppExtension.php
... lines 1 - 2
namespace App\Twig;

use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;
use Twig\TwigFunction;

class AppExtension extends AbstractExtension
{
    public function getFilters(): array
    {
        return [
            new TwigFilter('filter_name', [$this, 'doSomething'], ['is_safe' => ['html']]),
        ];
    }

    public function getFunctions(): array
    {
        return [
            new TwigFunction('function_name', [$this, 'doSomething']),
        ];
    }

    public function doSomething($value)
    {
        // ...
    }
}
```

Hello Twig extension! It's a normal PHP class that extends a base class, then specifies any custom functions or filters in these two methods:

```
30 lines | src/Twig/AppExtension.php
... lines 1 - 8
class AppExtension extends AbstractExtension
{
    public function getFilters(): array
    {
... lines 13 - 15
    }

    public function getFunctions(): array
    {
... lines 20 - 22
    }
... lines 24 - 28
}
```

Twig Extensions can add other stuff too, like custom operators or tests.

We need a custom filter, so delete getFunctions() and then change the filter name to cached_markdown. Over on the right, this is the *method* that will be called when the user uses the filter. Let's call our method processMarkdown. Point to that from the filter:

```
23 lines | src/Twig/AppExtension.php
... lines 1 - 5
use Twig\TwigFilter;
... lines 7 - 8
class AppExtension extends AbstractExtension
{
    public function getFilters(): array
    {
        return [
            new TwigFilter('cached_markdown', [$this, 'processMarkdown'], ['is_safe' => ['html']]),
        ];
    }

    public function processMarkdown($value)
    {
... line 20
    }
}
```

To make sure things are working, for now, in processMarkdown(), just return strtoupper($value):

```
23 lines | src/Twig/AppExtension.php
... lines 1 - 8
class AppExtension extends AbstractExtension
{
... lines 11 - 17
    public function processMarkdown($value)
    {
        return strtoupper($value);
    }
}
```

Sweet! In the Twig template, use it: |cached_markdown:

```
83 lines | templates/article/show.html.twig
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
... lines 11 - 25
                <div class="row">
                    <div class="col-sm-12">
                        <div class="article-text">
                            {{ article.content|cached_markdown }}
                        </div>
                    </div>
                </div>
... lines 33 - 71
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 78 - 83
```

Oh, and two important things. One, when you use a filter, the value to the *left* of the filter will become the first argument to your filter function. So, $value will be the article content in this case:

```
23 lines | src/Twig/AppExtension.php
... lines 1 - 8
class AppExtension extends AbstractExtension
{
... lines 11 - 17
    public function processMarkdown($value)
    {
... line 20
    }
}
```

Second, check out this options array when we added the filter. This is optional. But when you say is_safe set to html:

```
23 lines | src/Twig/AppExtension.php
... lines 1 - 8
class AppExtension extends AbstractExtension
{
    public function getFilters(): array
    {
        return [
            new TwigFilter('cached_markdown', [$this, 'processMarkdown'], ['is_safe' => ['html']]),
        ];
    }
... lines 17 - 21
}
```

It tells Twig that the result of this filter should *not* be escaped through htmlentities(). And... that's perfect! Markdown gives HTML code, and so we definitely do *not* want that to be escaped. You won't need this option on most filters, but we *do* want it here.

And... yea. We're done! Thanks to Symfony's autoconfiguration system, our Twig extension should already be registered with the Twig. So, find your browser, high-five your dog or cat, and refresh!

It works! I mean, it's *super* ugly and angry-looking... but it works!

Processing through Markdown

To make the extension use the MarkdownHelper, we're going to use good old-fashioned dependency injection. Add public function __construct() with a MarkdownHelper argument from our project:

31 lines | src/Twig/AppExtension.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 9
class AppExtension extends AbstractExtension
{
... lines 12 - 13
    public function __construct(MarkdownHelper $markdownHelper)
    {
... line 16
    }
... lines 18 - 29
}
```

Then, I'll press Alt+Enter and select "Initialize fields" so that PhpStorm creates that $helper property and sets it:

31 lines | src/Twig/AppExtension.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 9
class AppExtension extends AbstractExtension
{
    private $markdownHelper;

    public function __construct(MarkdownHelper $markdownHelper)
    {
        $this->markdownHelper = $markdownHelper;
    }
... lines 18 - 29
}
```

Down below, celebrate! Just return $this->helper->parse() and pass it the $value:

31 lines | src/Twig/AppExtension.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 9
class AppExtension extends AbstractExtension
{
... lines 12 - 25
    public function processMarkdown($value)
    {
        return $this->markdownHelper->parse($value);
    }
}
```

That's it! Go back, refresh and... brilliant! We *once* again have markdown, but now it's being cached.

Chapter 7: ago Filter with KnpTimeBundle

Ok, I just *need* to show you something fun - it deals with Twig filters. See this 4 hours ago? That's still hard coded! Find the show template and scroll up a bit to find it:

```
83 lines | templates/article/show.html.twig
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 17
                            <span class="pl-2 article-details"> 4 hours ago</span>
... lines 19 - 22
                        </div>
                    </div>
                </div>
... lines 26 - 71
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 78 - 83
```

There!

Printing a DateTime Object in Twig

The Article entity has a $publishedAt property, so let's get our act together and starting using that to print out the *real* date. Oh, but remember: the $publishedAt field might be null if the article has *not* been published yet. So let's use the fancy ternary syntax to say: {{ article.publishedAt }}, then, if it *is* published, print article.publishedAt. But, publishedAt is a DateTime *object*... and you can't just run around printing DateTime objects, and expect PHP to *not* get angry.

To fix that, pipe this through a date filter, and then say Y-m-d:

```
85 lines | templates/article/show.html.twig
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 17
                            <span class="pl-2 article-details">
                                {{ article.publishedAt ? article.publishedAt|date('Y-m-d') : 'unpublished' }}
                            </span>
... lines 21 - 24
                        </div>
                    </div>
                </div>
... lines 28 - 73
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 80 - 85
```

Most filters do not have any arguments - most are like cached_markdown. But filters *are* allowed to have arguments. If the article is *not* published, just say that: unpublished:

```
85 lines | templates/article/show.html.twig
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 17
                            <span class="pl-2 article-details">
                                {{ article.publishedAt ? article.publishedAt|date('Y-m-d') : 'unpublished' }}
                            </span>
... lines 21 - 24
                        </div>
                    </div>
                </div>
... lines 28 - 73
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 80 - 85
```

Love it! When we go back and refresh, published on March 20th.

Installing KnpTimeBundle

Cool... but it looked better when it said something like "five minutes ago" or "two weeks ago" - that was *way* more hipster. The date... it's ugly!

Fortunately, there's a really simple bundle that can convert your dates into this cute "ago" format. Search for KnpTimeBundle. Despite seeing my little face there, I did *not* create this bundle, so I take no credit for it. I just think it's great.

Scroll down to the "composer require" line, copy that, find your terminal and, paste!

```
$ composer require knplabs/knp-time-bundle
```

This installs the bundle and... interesting! It also installs symfony/translation. Behind the scenes, KnpTimeBundle uses the translator to translate the "ago" wording into other languages.

But what's *really* cool is that symfony/translation has a Flex *recipe*. Before I recorded this chapter, I committed our changes so far. So now I can run:

```
$ git status
```

to see what that sneaky translation recipe did. Interesting: we have a new config/packages/translation.yaml file and a new translations/ directory where any translation files should live... *if* we need any.

At a high level, the recipe system, like always, is making sure that everything is setup for us, automatically.

Using the ago Filter

Ok, let's use that filter! Back in the template, replace the date filter with |ago:

85 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 17
                            <span class="pl-2 article-details">
                                {{ article.publishedAt ? article.publishedAt|ago : 'unpublished' }}
                            </span>
... lines 21 - 24
                        </div>
                    </div>
                </div>
... lines 28 - 73
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 80 - 85
```

That's it. Find the page, refresh and... perfect! 27 days ago. So much nicer!

Next, I want to talk a little bit more about the AppExtension Twig extension because, for a very subtle but important reason, it has a performance problem.

Chapter 8: Service Subscriber: Lazy Performance

Our nice little Twig extension has a not-so-nice problem! And... it's subtle.

Normally, if you have a service like MarkdownHelper:

```
31 lines | src/Twig/AppExtension.php
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 9
class AppExtension extends AbstractExtension
{
    private $markdownHelper;

    public function __construct(MarkdownHelper $markdownHelper)
    {
        $this->markdownHelper = $markdownHelper;
    }
... lines 18 - 29
}
```

Symfony's container does *not* instantiate this service until and *unless* you actually use it during a request. For example, if we try to use MarkdownHelper in a controller, the container will, of course, instantiate MarkdownHelper and pass it to us.

But, in a different controller, if we *don't* use it, then that object will *never* be instantiated. And... that's perfect! Instantiating objects that we don't need would be a performance killer!

Twig Extensions: Always Instantiated

Well... Twig extensions are a special situation. *If* you go to a page that renders any Twig template, then the AppExtension will *always* be instantiated, even if we don't use any of its custom functions or filters. Twig needs to instantiate the extension so that it *knows* about those custom things.

But, in order to instantiate AppExtension, Symfony's container first needs to instantiate MarkdownHelper. So, for example, the homepage does *not* render anything through markdown. But because our AppExtension *is* instantiated, MarkdownHelper is *also* instantiated.

In other words, we are *now* instantiating an *extra* object - MarkdownHelper - on *every* request that uses Twig... even if we never actually use it! It sounds subtle, but as your Twig extension grows, this can become a real problem.

Creating a Service Subscriber

We *somehow* want to tell Symfony to pass us the MarkdownHelper, but not *actually* instantiate it until, and unless, we need it. That's totally possible.

But, it's a little bit tricky until you see the whole thing put together. So, watch closely.

First, make your class implement a new interface: ServiceSubscriberInterface:

```
42 lines | src/Twig/AppExtension.php
... lines 1 - 6
use Symfony\Component\DependencyInjection\ServiceSubscriberInterface;
... lines 8 - 11
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
... lines 14 - 40
}
```

This will force us to have one new method. At the bottom of the class, I'll go to the "Code"->"Generate" menu - or Command+N on a Mac - and implement getSubscribedServices(). Return an array from this... but leave it empty for now:

```
42 lines | src/Twig/AppExtension.php
... lines 1 - 6
use Symfony\Component\DependencyInjection\ServiceSubscriberInterface;
... lines 8 - 11
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
... lines 14 - 34
    public static function getSubscribedServices()
    {
        return [
... line 38
        ];
    }
}
```

Next, up on your constructor, *remove* the first argument and replace it with ContainerInterface - the one from Psr - $container:

```
42 lines | src/Twig/AppExtension.php
... lines 1 - 5
use Psr\Container\ContainerInterface;
... lines 7 - 11
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
... lines 14 - 15
    public function __construct(ContainerInterface $container)
    {
... line 18
    }
... lines 20 - 40
}
```

Also rename the property to $container:

42 lines | src/Twig/AppExtension.php

```
... lines 1 - 5
use Psr\Container\ContainerInterface;
... lines 7 - 11
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
    private $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }
... lines 20 - 40
}
```

Populating the Container

At this point... if you're *totally* confused... no worries! Here's the deal: *when* you make a service implements ServiceSubscriberInterface, Symfony will suddenly try to pass a *service container* to your constructor. It does this by looking for an argument that's type-hinted with ContainerInterface. So, you *can* still have *other* arguments, as long as one has this type-hint.

But, one important thing: this $container is *not* Symfony's big service container that holds hundreds of services. Nope, this is a mini-container, that holds a *subset* of those services. In fact, right now, it holds zero.

To tell Symfony *which* services you want in your mini-container, use getSubscribedServices(). Let's return the one service we need: MarkdownHelper::class:

42 lines | src/Twig/AppExtension.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 11
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
... lines 14 - 34
    public static function getSubscribedServices()
    {
        return [
            MarkdownHelper::class,
        ];
    }
}
```

When we do this, Symfony will basically autowire that service *into* the mini container, *and* make it public so that we can fetch it directly. In other words, down in processMarkdown(), we can use it with $this->container->get(MarkdownHelper::class) and then ->parse($value):

42 lines | src/Twig/AppExtension.php

```
... lines 1 - 4
use App\Service\MarkdownHelper;
... lines 6 - 11
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
... lines 14 - 27
    public function processMarkdown($value)
    {
        return $this->container
            ->get(MarkdownHelper::class)
            ->parse($value);
    }
... lines 34 - 40
}
```

At this point, this might feel like just a more complex version of dependency injection. And yea... it kinda is! Instead of passing us the MarkdownHelper directly, Symfony is passing us a *container* that *holds* the MarkdownHelper. But, the *key* difference is that, thanks to this trick, the MarkdownHelper service is *not* instantiated until and unless we fetch it out of this container.

Understanding getSubscribedServices()

Oh, and to *hopefully* make things a bit more clear, you can actually return a key-value pair from getSubscribedServices(), like 'foo' => MarkdownHelper::class:

```
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
    // ...
    public static function getSubscribedServices()
    {
        return [
            'foo' => MarkdownHelper::class,
        ];
    }
}
```

If we did this, it would *still* mean that the MarkdownHelper service is autowired into the mini-container, but we would reference it internally with the id foo.

If you just pass MarkdownHelper::class as the value, then that's also used as the key.

The end result is exactly the same as before, except MarkdownHelper is lazy! To prove it, put a die statement at the top of the MarkdownHelper constructor.

Now, go back to the article page and refresh. Not surprising: it hits the die statement when rendering the Twig template. But now, go back to the homepage. Yes! The *whole* page prints: MarkdownHelper is never instantiated.

Go back and remove that die statement.

Tip

There is a better way for creating lazy-Loaded Twig extensions since Twig v1.26. First of all, create a separate class, e.g. AppRuntime, that implements RuntimeExtensionInterface and inject MarkdownHelper object there. Also, move processMarkdown() method there:

```
namespace App\Twig;

use App\Service\MarkdownHelper;
use Twig\Extension\RuntimeExtensionInterface;

class AppRuntime implements RuntimeExtensionInterface
{
    private $markdownHelper;

    public function __construct(MarkdownHelper $markdownHelper)
    {
        $this->markdownHelper = $markdownHelper;
    }

    public function processMarkdown($value)
    {
        return $this->markdownHelper->parse($value);
    }
}
```

And then, in AppExtension, remove MarkdownHelper at all and point the cached_markdown filter to [AppRuntime::class, 'processMarkdown'] instead:

```
namespace App\Twig;

use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;

class AppExtension extends AbstractExtension
{
    public function getFilters(): array
    {
        return [
            new TwigFilter('cached_markdown', [AppRuntime::class, 'processMarkdown'], ['is_safe' => ['html']]),
        ];
    }
}
```

That's it! Now our Twig extension does not have any direct dependencies, and AppRuntime object will be created only when cached_markdown is called.

Here's the super-duper-important takeaway: I want you to use *normal* dependency injection everywhere - just pass each service you need through the constructor, *without* all this fancy service-subscriber stuff.

But then, in just a *couple* of places in Symfony, the main ones being Twig extensions, event subscribers and security voters - a few topics we'll talk about in the future - you should consider using a service subscriber instead to avoid a performance hit.

Chapter 9: All about Entity Repositories

With the article show page now dynamic, let's turn to the homepage... cause these news stories are *totally* still hardcoded. Open ArticleController and find the homepage() action:

76 lines | src/Controller/ArticleController.php

```
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 30
    public function homepage()
    {
        return $this->render('article/homepage.html.twig');
    }
... lines 35 - 74
}
```

Perfect. *Just* like before, we need to query for the articles. This means that we need an EntityManagerInterface $em argument:

81 lines | src/Controller/ArticleController.php

```
... lines 1 - 7
use Doctrine\ORM\EntityManagerInterface;
... lines 9 - 15
class ArticleController extends AbstractController
{
... lines 18 - 30
    public function homepage(EntityManagerInterface $em)
    {
... lines 33 - 38
    }
... lines 40 - 79
}
```

Next, we get the *repository* for the class: $repository = $em->getRepository(Article::class). And *then* we can say, $articles = $repository->findAll():

81 lines | src/Controller/ArticleController.php

```
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 30
    public function homepage(EntityManagerInterface $em)
    {
        $repository = $em->getRepository(Article::class);
        $articles = $repository->findAll();
... lines 35 - 38
    }
... lines 40 - 79
}
```

Nice! With this array of Article objects in hand, let's pass those into the template as a new articles variable:

81 lines | src/Controller/ArticleController.php

```
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 30
    public function homepage(EntityManagerInterface $em)
    {
        $repository = $em->getRepository(Article::class);
        $articles = $repository->findAll();

        return $this->render('article/homepage.html.twig', [
            'articles' => $articles,
        ]);
    }
... lines 40 - 79
}
```

Now, to the template! Open homepage.html.twig and scroll down *just* a little bit. Yes: here is the article list:

81 lines | templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: 'why-asteroids-taste-like-bacon'}) }}">
                        <img class="article-img" src="{{ asset('images/asteroid.jpeg') }}">
                        <div class="article-title d-inline-block pl-3 align-middle">
                            <span>Why do Asteroids Taste Like Bacon?</span>
                            <br>
                            <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('images/alien-pro
                            <span class="pl-5 article-details float-right"> 3 hours ago</span>
                        </div>
                    </a>
                </div>
... lines 32 - 56
            </div>
... lines 58 - 77
        </div>
    </div>
{% endblock %}
```

Well, there's a "main" article on top, but I'm going to ignore that for now. Down below, add for article in articles with, at the bottom, endfor:

```
58 lines | templates/article/homepage.html.twig
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                {% for article in articles %}
                <div class="article-container my-1">
... lines 23 - 31
                </div>
                {% endfor %}
            </div>
... lines 35 - 54
        </div>
    </div>
{% endblock %}
```

Then... just make things dynamic: article.slug, article.title, and for the three hours ago, if article.publishedAt is not null, print article.publishedAt|ago. If it's *not* published, do nothing. With this in place, delete the last two hardcoded articles:

```
58 lines | templates/article/homepage.html.twig
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                {% for article in articles %}
                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: article.slug}) }}">
                        <img class="article-img" src="{{ asset('images/asteroid.jpeg') }}">
                        <div class="article-title d-inline-block pl-3 align-middle">
                            <span>{{ article.title }}</span>
                            <br>
                            <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('images/alien-pro
                            <span class="pl-5 article-details float-right"> {{ article.publishedAt ? article.publishedAt|ago }}</span>
                        </div>
                    </a>
                </div>
                {% endfor %}
            </div>
... lines 35 - 54
        </div>
    </div>
{% endblock %}
```

Controlling the ORDER BY

Let's give it a try: find your browser and, refresh! Nice! You can see a mixture of published and unpublished articles. But... you can *also* see that the articles just printed out in whatever order they were created, *regardless* of the publish date. Space travellers demand fresh content! So let's print the *newest* articles first.

Head back to ArticleController. Hmm... the findAll() methods gives us *everything*... but it's pretty limited. In fact, it takes zero arguments: you can't control it at *all*:

```
81 lines | src/Controller/ArticleController.php
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 30
    public function homepage(EntityManagerInterface $em)
    {
... line 33
        $articles = $repository->findAll();
... lines 35 - 38
    }
... lines 40 - 79
}
```

But, some of the *other* methods *are* just a little bit more flexible. To control the order, use findBy() instead, pass this an empty array, and then another array with publishedAt set to DESC:

81 lines | src/Controller/ArticleController.php

```
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 30
    public function homepage(EntityManagerInterface $em)
    {
... line 33
        $articles = $repository->findBy([], ['publishedAt' => 'DESC']);
... lines 35 - 38
    }
... lines 40 - 79
}
```

The first array is where you would normally pass some criteria for a WHERE clause. If we pass nothing, we get everything!

Try it - refresh! Much better!

Hello ArticleRepository

Except... hmm... it probably does *not* make sense to show the unpublished articles on the homepage. And *this* is when things get a bit more interesting. Sure, you can pass simple criteria to findBy(), like slug equal to some value. But, in this case, we need a query that says WHERE publishedAt IS NOT NULL. That's just *not* possible with findBy()!

And so... for the *first* time, we're going to write - drumroll - a custom query!

Let me show you something cool: when we originally generated our entity, the command created the Article class, but it *also* created an ArticleRepository class in the Repository directory. Try this: dump($repository) and, refresh. Guess what? This is an instance of that ArticleRepository!

Yes, there is a *connection* between the Article and ArticleRepository classes. In fact, that connection is explicitly configured right at the top of your Article class:

93 lines | src/Entity/Article.php

```
... lines 1 - 6
/**
 * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
 */
class Article
{
... lines 12 - 91
}
```

This says: when we ask for the Article class's repository, Doctrine should give us an instance of this ArticleRepository class:

51 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 5
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
... lines 7 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 49
}
```

Oh, and the built-in find*() methods actually come from one of the parent classes of ArticleRepository.

So... why the heck are we talking about this? Because, if you want to create a custom query, you can do that by creating a custom *method* inside of this class. And, hey! It even has a couple of examples:

```
51 lines | src/Repository/ArticleRepository.php
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 21
//    /**
//     * @return Article[] Returns an array of Article objects
//     */
    /*
    public function findByExampleField($value)
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.exampleField = :val')
            ->setParameter('val', $value)
            ->orderBy('a.id', 'ASC')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult()
        ;
    }
    */

    /*
    public function findOneBySomeField($value): ?Article
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.exampleField = :val')
            ->setParameter('val', $value)
            ->getQuery()
            ->getOneOrNullResult()
        ;
    }
    */
}
```

Uncomment the first example, and rename it to findAllPublishedOrderedByNewest():

```
49 lines | src/Repository/ArticleRepository.php
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 21
    /**
     * @return Article[]
     */
    public function findAllPublishedOrderedByNewest()
    {
... lines 27 - 34
    }
... lines 36 - 47
}
```

I *love* descriptive names... or maybe I love *long* names... not sure.

Anyways, it's time to talk about *how* you actually write custom queries in Doctrine. Let's do that next!

Chapter 10: Custom Queries

How do you write custom queries in Doctrine? Well, you're already familiar with writing SQL, and, yea, it *is* possible to write raw SQL queries with Doctrine. But, most of the time, you won't do this. Instead, because Doctrine is a library that works with *many* different database engines, Doctrine has its *own* SQL-like language called Doctrine query language, or DQL.

Fortunately, DQL looks almost *exactly* like SQL. Except, instead of table and column names in your query, you'll use class and property names. Again, Doctrine *really* wants you to pretend like there is no database, tables or columns behind the scenes. It wants you to pretend like you're saving and fetching *objects* and their properties.

Introducing: The Query Builder

Anyways, to write a custom query, you can either create a DQL string directly, *or* you can do what I usually do: use the query builder. The query builder is just an object-oriented builder that helps *create* a DQL string. Nothing fancy.

And there's a *pretty* good example right here: you can add where statements order by, limits and pretty much anything else:

49 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 24
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.exampleField = :val')
            ->setParameter('val', $value)
            ->orderBy('a.id', 'ASC')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult()
        ;
    }
... lines 36 - 47
}
```

One nice thing is that you can do this all in any order - you could put the order by first, and the where statements after. The query builder doesn't care!

Oh, and see this andWhere()?

49 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 24
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.exampleField = :val')
... lines 29 - 33
        ;
    }
... lines 36 - 47
}
```

There *is* a normal where() method, but it's safe to use andWhere() even if this is the *first* WHERE clause. Again the query builder is smart enough to figure it out. I recommend andWhere(), because where() will remove any previous where clauses you may have added... which... can be a gotcha!

DQL - and so, the query builder - also uses prepared statements. If you're not familiar with them, it's a really simple idea: whenever you want to put a dynamic value into a query, instead of hacking it into the string with concatenation, put : and any placeholder name. Then, later, give that placeholder a value with ->setParameter():

49 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 24
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.exampleField = :val')
            ->setParameter('val', $value)
... lines 30 - 33
        ;
    }
... lines 36 - 47
}
```

This prevents SQL injection.

Writing our Custom Query

In our case, we won't need any arguments, and I'm going to simplify a bit. Let's say andWhere('a.publishedAt IS NOT NULL'):

```
47 lines | src/Repository/ArticleRepository.php

... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 24
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.publishedAt IS NOT NULL')
... lines 29 - 31
        ;
    }
... lines 34 - 45
}
```

You can *totally* see how close this is to normal SQL. You can even put OR statements inside the string, like a.publishedAt IS NULL OR a.publishedAt > NOW().

Oh, and what the heck does the a mean? Think of this as the table *alias* for Article in the query - just like how you can say SELECT a.* FROM article AS a.

It could be anything: if you used article instead, you'd just need to change all the references from a. to article..

Let's also add our orderBy(), with a.publishedAt, DESC:

```
47 lines | src/Repository/ArticleRepository.php

... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 24
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.publishedAt IS NOT NULL')
            ->orderBy('a.publishedAt', 'DESC')
... lines 30 - 31
        ;
    }
... lines 34 - 45
}
```

Oh, and this is a good example of how we're referencing the *property* name on the entity. The *column* name in the database is actually published_at, but we don't use that here.

Finally, let's remove the max result:

47 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 24
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.publishedAt IS NOT NULL')
            ->orderBy('a.publishedAt', 'DESC')
... lines 30 - 31
        ;
    }
... lines 34 - 45
}
```

Once you're done building your query, you always call getQuery() and then, to get the array of Article objects, getResult():

47 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 24
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.publishedAt IS NOT NULL')
            ->orderBy('a.publishedAt', 'DESC')
            ->getQuery()
            ->getResult()
        ;
    }
... lines 34 - 45
}
```

Below this method, there's an example of finding just *one* object:

47 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 34
    /*
    public function findOneBySomeField($value): ?Article
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.exampleField = :val')
            ->setParameter('val', $value)
            ->getQuery()
            ->getOneOrNullResult()
        ;
    }
    */
}
```

It's almost the same: build the query, call getQuery(), but then finish with getOneOrNullResult().

So, in all normal situations, you *always* call getQuery(), then you'll either call getResult() to return many rows of articles, or getOneOrNullResult() to return a single Article object. Got it?

Now that our new findAllPublishedOrderedByNewest() method is done, let's go use it in the controller: $repository->, and there it is!

81 lines | src/Controller/ArticleController.php

```
... lines 1 - 15
class ArticleController extends AbstractController
{
... lines 18 - 30
    public function homepage(EntityManagerInterface $em)
    {
        $repository = $em->getRepository(Article::class);
        $articles = $repository->findAllPublishedOrderedByNewest();
... lines 35 - 38
    }
... lines 40 - 79
}
```

Let's give it a try! Move over and, refresh! Perfect! The order is correct and the unpublished articles are gone.

Autowiring ArticleRepository

To make this even cooler, let me show you a trick. Instead of getting the entity manager and then calling getRepository() to get the ArticleRepository, you can take a shortcut: just type ArticleRepository $repository:

81 lines | src/Controller/ArticleController.php

```
... lines 1 - 5
use App\Repository\ArticleRepository;
... lines 7 - 16
class ArticleController extends AbstractController
... lines 18 - 31
    public function homepage(ArticleRepository $repository)
    {
        $articles = $repository->findAllPublishedOrderedByNewest();
... lines 35 - 38
    }
... lines 40 - 79
}
```

This works for a *simple* reason: all of your repositories are automatically registered as services in the container. So you can autowire them like anything else. *This* is how I actually code when I need a repository.

And when we refresh, no surprise, it works!

Custom queries are a *big* topic, and we'll continue writing a few more here and there. But if you have something particularly challenging, check out our Go Pro with Doctrine Queries tutorial. That tutorial uses Symfony 3, but the query logic is *exactly* the same as in Symfony 4.

Next, I want to show you two more tricks: one for re-using query logic between multiple queries, and another *super* shortcut to fetch any entity with *zero* work.

Chapter 11: Query Logic Re-use & Shortcuts

One of my *favorite* things about the query builder is that, with a few tricks, you can *reuse* query logic! Check this out: right now, we only have one custom method in ArticleRepository:

```
47 lines | src/Repository/ArticleRepository.php
... lines 1 - 14
class ArticleRepository extends ServiceEntityRepository
{
... lines 17 - 21
    /**
     * @return Article[]
     */
    public function findAllPublishedOrderedByNewest()
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.publishedAt IS NOT NULL')
            ->orderBy('a.publishedAt', 'DESC')
            ->getQuery()
            ->getResult()
        ;
    }
... lines 34 - 45
}
```

But, as our app grows, we'll *certainly* need to add more. And there's a *pretty* darn good chance that *another* custom query will *also* need to filter its results to only show *published* articles. In a *perfect* world, we would *share* that logic, between both custom methods. And... we can do that!

Step 1 is to isolate the query logic that we need to share into its own private method. At the bottom, create a private function addIsPublishedQueryBuilder() with a QueryBuilder type-hint - the one from Doctrine\ORM - and $qb:

```
54 lines | src/Repository/ArticleRepository.php
... lines 1 - 6
use Doctrine\ORM\QueryBuilder;
... lines 8 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 48
    private function addIsPublishedQueryBuilder(QueryBuilder $qb)
    {
... line 51
    }
}
```

Next, go up, copy that part of the query, and just return $qb->andWhere('a.publishedAt IS NOT NULL'):

```
54 lines | src/Repository/ArticleRepository.php
... lines 1 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 48
    private function addIsPublishedQueryBuilder(QueryBuilder $qb)
    {
        return $qb->andWhere('a.publishedAt IS NOT NULL');
    }
}
```

And since we're *returning* this - and each query builder method returns itself - back up top, we can say $qb = $this->createQueryBuilder('a'), and below, return $this->addIsPublishedQueryBuilder() passing it $qb:

```
54 lines | src/Repository/ArticleRepository.php
... lines 1 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 25
    public function findAllPublishedOrderedByNewest()
    {
        $qb = $this->createQueryBuilder('a');

        return $this->addIsPublishedQueryBuilder($qb)
            ->orderBy('a.publishedAt', 'DESC')
            ->getQuery()
            ->getResult()
        ;
    }
... lines 36 - 52
}
```

The rest of the query can chain off of this.

And... that's it! One important note is that you need to consistently use the same alias, like a, across all of your methods.

Fancier Re-Use

This is nice... but since I do this a lot, we can get a bit fancier. Create another private method called getOrCreateQueryBuilder() with a QueryBuilder argument like before, but make it optional:

```
58 lines | src/Repository/ArticleRepository.php
... lines 1 - 6
use Doctrine\ORM\QueryBuilder;
... lines 8 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 52
    private function getOrCreateQueryBuilder(QueryBuilder $qb = null)
    {
... line 55
    }
}
```

Here's the idea: when someone calls this method, *if* the query builder is passed, we'll just return it. Otherwise we will return a new one with $this->createQueryBuilder('a'):

58 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 52
    private function getOrCreateQueryBuilder(QueryBuilder $qb = null)
    {
        return $qb ?: $this->createQueryBuilder('a');
    }
}
```

If you're not used to this syntax, it means that if a QueryBuilder object is passed, return that QueryBuilder object. If a QueryBuilder object is not passed, then create one.

This is *cool*, because *now* we can make the argument to addIsPublishedQueryBuilder() *also* optional:

58 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 46
    private function addIsPublishedQueryBuilder(QueryBuilder $qb = null)
    {
... lines 49 - 50
    }
... lines 52 - 56
}
```

Inside, use the new method: return $this->getOrCreateQueryBuilder() passing it $qb, and then our andWhere():

58 lines | src/Repository/ArticleRepository.php

```
... lines 1 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 46
    private function addIsPublishedQueryBuilder(QueryBuilder $qb = null)
    {
        return $this->getOrCreateQueryBuilder($qb)
            ->andWhere('a.publishedAt IS NOT NULL');
    }
... lines 52 - 56
}
```

But the *real* beautiful thing is back up top. This *whole* method can now be one big chained call: return $this->addIsPublishedQueryBuilder() - and pass nothing:

```
58 lines | src/Repository/ArticleRepository.php
... lines 1 - 15
class ArticleRepository extends ServiceEntityRepository
{
... lines 18 - 25
    public function findAllPublishedOrderedByNewest()
    {
        return $this->addIsPublishedQueryBuilder()
            ->orderBy('a.publishedAt', 'DESC')
            ->getQuery()
            ->getResult()
        ;
    }
... lines 34 - 56
}
```

It will create the QueryBuilder for us.

So not only do we have really nice public functions for fetching data, we also have some private functions to help us *build* our queries. Let's make sure it works. Find your browser and, refresh! It still looks good!

ParamConverter: Automatically Querying

Ok, enough custom queries for now. Instead, I want to show you a query shortcut!

Go to ArticleController and find the show() action. Sometimes you need to query for an *array* of objects. So, we get the repository, call some method, and, done!

```
81 lines | src/Controller/ArticleController.php
... lines 1 - 5
use App\Repository\ArticleRepository;
... lines 7 - 16
class ArticleController extends AbstractController
{
... lines 19 - 31
    public function homepage(ArticleRepository $repository)
    {
        $articles = $repository->findAllPublishedOrderedByNewest();
... lines 35 - 38
    }
... lines 40 - 79
}
```

Life is good. But it's *also* really common to query for just *one* object. And in these situations, if the query you need is simple... you can make Symfony do *all* of the work:

81 lines | src/Controller/ArticleController.php

```
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 43
    public function show($slug, SlackClient $slack, EntityManagerInterface $em)
    {
... lines 46 - 50
        /** @var Article $article */
        $article = $repository->findOneBy(['slug' => $slug]);
... lines 53 - 66
    }
... lines 68 - 79
}
```

Let me show you: remove the $slug argument and replace it with Article $article:

74 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 16
class ArticleController extends AbstractController
{
... lines 19 - 40
    /**
     * @Route("/news/{slug}", name="article_show")
     */
    public function show(Article $article, SlackClient $slack)
    {
... lines 46 - 59
    }
... lines 61 - 72
}
```

Then, below, because I removed the $slug argument, use $article->getSlug():

74 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 16
class ArticleController extends AbstractController
{
... lines 19 - 40
    /**
     * @Route("/news/{slug}", name="article_show")
     */
    public function show(Article $article, SlackClient $slack)
    {
        if ($article->getSlug() === 'khaaaaaan') {
... line 47
        }
... lines 49 - 59
    }
... lines 61 - 72
}
```

We can also remove *all* of the query, and even the 404 logic:

74 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 16
class ArticleController extends AbstractController
{
... lines 19 - 40
    /**
     * @Route("/news/{slug}", name="article_show")
     */
    public function show(Article $article, SlackClient $slack)
    {
        if ($article->getSlug() === 'khaaaaaan') {
            $slack->sendMessage('Kahn', 'Ah, Kirk, my old friend...');
        }

        $comments = [
            'I ate a normal rock once. It did NOT taste like bacon!',
            'Woohoo! I\'m going on an all-asteroid diet!',
            'I like bacon too! Buy some from my site! bakinsomebacon.com',
        ];

        return $this->render('article/show.html.twig', [
            'article' => $article,
            'comments' => $comments,
        ]);
    }
... lines 61 - 72
}
```

Before we talk about this, move over and click on one of the articles. Yea! Somehow, this totally works! Back in our code, we can remove the unused EntityManagerInterface argument:

74 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 7
use App\Service\SlackClient;
... lines 9 - 16
class ArticleController extends AbstractController
{
... lines 19 - 43
    public function show(Article $article, SlackClient $slack)
    {
... lines 46 - 59
    }
... lines 61 - 72
}
```

Here's the deal. We already know that if you type-hint, a *service*, Symfony will pass you that service. In addition to that, if you *type-hint* an *entity* class, Symfony will automatically query for that entity. How? It looks at all of the route's placeholder values - which is just one in this case, {slug} - and creates a query where the slug field matches that value:

Tip

It requires sensio/framework-extra-bundle to be installed in order to automatically query for entity objects

74 lines | src/Controller/ArticleController.php

```
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 40
    /**
     * @Route("/news/{slug}", name="article_show")
     */
    public function show(Article $article, SlackClient $slack)
    {
... lines 46 - 59
    }
... lines 61 - 72
}
```

In other words, to use this trick, your routing wildcard *must* be named the same as the property on your entity, which is usually how I do things anyways. It executes the *exact* same query that we were doing before by hand! If there is *not* a slug that matches this, it *also* automatically throws a 404, before the controller is ever called.

In fact, try that - put in a bad slug. Yep, error! Something about the Article object not found by the @ParamConverter annotation. So, that's not a great error message - it makes more sense if you know that the name of this feature internally is ParamConverter.

So... yea! If you organize your route wildcards to match the property on your entity, which is a good idea anyways, then you can use this trick. If you need a more complex query, no problem! You can't use this shortcut, but it's still simple enough: autowire the ArticleRepository, and then call whatever method you need.

Chapter 12: Updating an Entity with New Fields

It's time to get back to work on the article page... because... some of this stuff is still hardcoded! Lame! Like, the author, number of hearts, and this image. There are a few possible images in our project that our dummy articles can point to.

Our mission is clear: create three new fields in Article and use those to make all of this finally dynamic! Let's go!

Open your Article entity. The simplest way to add new fields is just to... add them by hand! It's easy enough to copy an existing property, paste, rename, and configure it. Of course, if you want a getter and setter method, you'll also need to create those.

Generating New Fields into the Entity

Because of that, *my* favorite way to *add* fields is to, once again, be lazy, and generate them! Find your terminal and run the *same* command as before:

```
$ php bin/console make:entity
```

If you pass this the name of an *existing* entity, it can actually *update* that class and add new fields. Magic! First, add author, use string as the type. And yea, in the future when we have a "user" system, this field might be a database relation to that table. But for now, use a string. Say no to nullable. Reminder: when you say *no* to nullable, it means that this field *must* be set in the database. If you try to save an entity *without* any data on it, you'll get a huge database exception.

Next, add heartCount, as an integer, and say not null: this should always have a value, even if it's zero. Then, finally, the image. In the database, we'll store only the image *filename*. And, full disclosure, uploading files is a whole different topic that we'll cover in a *different* tutorial. In this example, we're going to use a few existing images in the public/ directory. But, both in this situation and in a real-file upload situation, the field on your entity looks the same: imageFilename as a string and nullable yes, because maybe the image is optional when you first start writing an article.

Ok, hit enter and, done! Let's go check out the entity! Great: three new properties on top:

```
144 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 38
    /**
     * @ORM\Column(type="string", length=255)
     */
    private $author;

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

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    private $imageFilename;
... lines 53 - 142
}
```

And of course, at the bottom, here are their getter and setter methods:

```
144 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 107
    public function getAuthor(): ?string
    {
        return $this->author;
    }

    public function setAuthor(string $author): self
    {
        $this->author = $author;

        return $this;
    }

    public function getHeartCount(): ?int
    {
        return $this->heartCount;
    }

    public function setHeartCount(int $heartCount): self
    {
        $this->heartCount = $heartCount;

        return $this;
    }

    public function getImageFilename(): ?string
    {
        return $this->imageFilename;
    }

    public function setImageFilename(?string $imageFilename): self
    {
        $this->imageFilename = $imageFilename;

        return $this;
    }
}
```

Now that we have the new fields, don't forget! We need a migration:

```
$ php bin/console make:migration
```

When that finishes, go look at the new file to make sure it doesn't have any surprises: ALTER TABLE article, and then it adds author, heart_count and image_filename:

```
29 lines | src/Migrations/Version20180414171443.php
... lines 1 - 10
class Version20180414171443 extends AbstractMigration
{
    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 \'mys

        $this->addSql('ALTER TABLE article ADD author VARCHAR(255) NOT NULL, ADD heart_count INT NOT NULL, ADD image_file
    }

    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 \'mys

        $this->addSql('ALTER TABLE article DROP author, DROP heart_count');
    }
}
```

I *love* it!

Close that, run back to your terminal, and migrate!

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

Field Default Value

Next, we need to make sure these new fields are populated on our dummy articles. Open ArticleAdminController.

Oh, but first, remember that, in the Article entity, heartCount is *required* in the database:

```
144 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 43
    /**
     * @ORM\Column(type="integer")
     */
    private $heartCount;
... lines 48 - 142
}
```

Actually, to be more clear: nullable=true means that it *is* allowed to be null in the database. If you *don't* see nullable, it uses the *default* value, which is false.

Anyways, this means that heartCount *needs* a value! But here's a cool idea: once our admin area is fully finished, when an author creates a new article, they shouldn't need to *set* the heartCount manually. I mean, it's not like we expect the form to have a "heart count" input field on it. Nope, we expect it to automatically default to zero for new articles.

So... how can we give a property a *default* value in the database? By giving it a default value in PHP: $heartCount = 0:

```
144 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 43
    /**
     * @ORM\Column(type="integer")
     */
    private $heartCount = 0;
... lines 48 - 142
}
```

Using the new Fields

Ok, back to ArticleAdminController! Add $article->setAuthor() and use the same data we had on the original, hardcoded articles:

```
61 lines | src/Controller/ArticleAdminController.php
... lines 1 - 10
class ArticleAdminController extends AbstractController
{
... lines 13 - 15
    public function new(EntityManagerInterface $em)
    {
... lines 18 - 40
        // publish most articles
        if (rand(1, 10) > 2) {
            $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
        }

        $article->setAuthor('Mike Ferengi')
... lines 47 - 58
    }
}
```

Then, ->setHeartCount() and give this a random number between, how about, 5 and 100:

```
61 lines | src/Controller/ArticleAdminController.php
... lines 1 - 10
class ArticleAdminController extends AbstractController
{
... lines 13 - 15
    public function new(EntityManagerInterface $em)
    {
... lines 18 - 45
        $article->setAuthor('Mike Ferengi')
            ->setHeartCount(rand(5, 100))
... lines 48 - 58
    }
}
```

And finally, ->setImageFilename(). The file we've been using is called asteroid.jpeg. Keep using that:

```
61 lines | src/Controller/ArticleAdminController.php
... lines 1 - 10
class ArticleAdminController extends AbstractController
{
... lines 13 - 15
    public function new(EntityManagerInterface $em)
    {
... lines 18 - 45
        $article->setAuthor('Mike Ferengi')
            ->setHeartCount(rand(5, 100))
            ->setImageFilename('asteroid.jpeg')
        ;
... lines 50 - 58
    }
}
```

Excelente! Because we already have a bunch of records in the database where these fields are *blank*, just to keep things simple, let's delete the table entirely and start fresh. Do that with:

```
$ php bin/console doctrine:query:sql "TRUNCATE TABLE article"
```

If you check out the page now and refresh... cool, it's empty. Now, go to /admin/article/new and... refresh a few times. Awesome! Check out the homepage!

We *have* articles... but actually... this author is still hardcoded in the template. Easy fix!

Updating the Templates

Open up homepage.html.twig. Let's first change the... where is it... ah, yes! The author's name: use {{ article.author }}:

58 lines | templates/article/homepage.html.twig

```
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                {% for article in articles %}
                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: article.slug}) }}">
... line 24
                        <div class="article-title d-inline-block pl-3 align-middle">
... lines 26 - 27
                            <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('images/alien-pro
... line 29
                        </div>
                    </a>
                </div>
                {% endfor %}
            </div>
... lines 35 - 54
        </div>
    </div>
{% endblock %}
```

Then, in show.html.twig, change the article's heart count - here it is - to {{ article.heartCount }}. And also update the author, just like before:

86 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 16
                            <span class="align-left article-details"><img class="article-author-img rounded-circle" src="{{ asset('images/alien-pro
... lines 18 - 20
                            <span class="pl-2 article-details">
                                <span class="js-like-article-count">{{ article.heartCount }}</span>
... line 23
                            </span>
                        </div>
                    </div>
                </div>
... lines 28 - 73
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 80 - 86
```

If you try the homepage now, ok, it looks *exactly* the same, but *we* know that these author names are now dynamic. If you click into an article.. yea! We have 88 hearts - that's definitely coming from the database.

Updating the Image Path

The *last* piece that's still hardcoded is this image. Go back to homepage.html.twig. The image path uses asset('images/asteroid.jpeg'):

```
58 lines | templates/article/homepage.html.twig
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                {% for article in articles %}
                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: article.slug}) }}">
                        <img class="article-img" src="{{ asset('images/asteroid.jpeg') }}">
... lines 25 - 30
                    </a>
                </div>
                {% endfor %}
            </div>
... lines 35 - 54
        </div>
    </div>
{% endblock %}
```

So... this is a *little* bit tricky, because only part of this - the asteroid.jpeg part - is stored in the database. One solution would be to use Twig's concatenation operator, which is ~, then article.imageFilename:

```
{# templates/article/homepage.html.twig #}

{# ... #}
    <img class="article-img" src="{{ asset('images/'~article.imageFilename) }}">
{# ... #}
```

You don't see the ~ much in Twig, but it works like a . in PHP.

That's fine, but a *nicer* way would be to create a new method that does this for us. Open Article and, at the bottom, create a new public function getImagePath():

```
149 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 143
    public function getImagePath()
    {
... line 146
    }
}
```

Inside, return images/ and then $this->getImageFilename():

```
149 lines | src/Entity/Article.php

... lines 1 - 9
class Article
{
... lines 12 - 143
    public function getImagePath()
    {
        return 'images/'.$this->getImageFilename();
    }
}
```

Thanks to this, in the template, we only need to say article.imagePath:

```
58 lines | templates/article/homepage.html.twig

... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">

            <!-- Article List -->

            <div class="col-sm-12 col-md-8">
... lines 10 - 18
                <!-- Supporting Articles -->

                {% for article in articles %}
                <div class="article-container my-1">
                    <a href="{{ path('article_show', {slug: article.slug}) }}">
                        <img class="article-img" src="{{ asset(article.imagePath) }}">
... lines 25 - 30
                    </a>
                </div>
                {% endfor %}
            </div>
... lines 35 - 54
        </div>
    </div>
{% endblock %}
```

And yea, imagePath is totally *not* a real property on Article! But thanks to the kung fu powers of Twig, this works fine.

Oh, and side note: notice that there is *not* an opening slash on these paths:

```
149 lines | src/Entity/Article.php

... lines 1 - 9
class Article
{
... lines 12 - 143
    public function getImagePath()
    {
        return 'images/'.$this->getImageFilename();
    }
}
```

As a reminder, you do *not* need to include the opening / when using the asset() function: Symfony will add it there automatically.

Ok, try it out - refresh! It still works! And now that we've centralized that method, in show.html.twig, it's *super* easy to make the same change: article.imagePath:

```
86 lines | templates/article/show.html.twig
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
                        <img class="show-article-img" src="{{ asset(article.imagePath) }}">
... lines 14 - 25
                    </div>
                </div>
... lines 28 - 73
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 80 - 86
```

Awesome. And when you click on the show page, it works too.

Next! Now that the heart count is stored in the database, let's make our AJAX endpoint that "likes" an article *actually* work correctly. Right now, it does nothing, and returns a random number. We can do better.

Chapter 13: Updating an Entity

In the previous tutorial, we created our heart feature! You click on the heart, it makes an Ajax request back to the server, and returns the *new* number of hearts. It's all very cute. In *theory*... when we click the heart, it would update the number of "hearts" for this article somewhere in the database.

But actually, instead of updating the database... well... it does *nothing*, and returns a new, *random* number of hearts. Lame!

Look in the public/js directory: open article_show.js:

16 lines | public/js/article_show.js

```
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
        e.preventDefault();

        var $link = $(e.currentTarget);
        $link.toggleClass('fa-heart-o').toggleClass('fa-heart');

        $.ajax({
            method: 'POST',
            url: $link.attr('href')
        }).done(function(data) {
            $('.js-like-article-count').html(data.hearts);
        })
    });
});
```

In that tutorial, we wrote some simple JavaScript that said: when the "like" link is clicked, toggle the styling on the heart, and then send a POST request to the URL that's in the href of the link. Then, when the AJAX call finishes, read the new number of hearts from the JSON response and update the page.

The href that we're reading lives in show.html.twig. Here it is:

86 lines | templates/article/show.html.twig

```
... lines 1 - 4
{% block body %}

<div class="container">
    <div class="row">
        <div class="col-sm-12">
            <div class="show-article-container p-3 mt-4">
                <div class="row">
                    <div class="col-sm-12">
... line 13
                        <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 15 - 20
                            <span class="pl-2 article-details">
... line 22
                                <a href="{{ path('article_toggle_heart', {slug: article.slug}) }}" class="fa fa-heart-o like-article js-like-article"></a>
                            </span>
                        </div>
                    </div>
                </div>
... lines 28 - 73
            </div>
        </div>
    </div>
</div>

{% endblock %}
... lines 80 - 86
```

It's a URL to some route called article_toggle_heart. And we're sending the article *slug* to that endpoint.

Open up ArticleController, and scroll down to find that route: it's toggleArticleHeart():

74 lines | src/Controller/ArticleController.php

```
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart($slug, LoggerInterface $logger)
    {
        // TODO - actually heart/unheart the article!

        $logger->info('Article is being hearted!');

        return new JsonResponse(['hearts' => rand(5, 100)]);
    }
}
```

And, as you can see... this endpoint doesn't actually do anything! Other than return JSON with a random number, which our JavaScript uses to update the page:

16 lines | public/js/article_show.js

```
$(document).ready(function() {
    $('.js-like-article').on('click', function(e) {
... lines 3 - 7
        $.ajax({
... lines 9 - 10
        }).done(function(data) {
            $('.js-like-article-count').html(data.hearts);
        })
    });
});
```

Updating the heartCount

It's time to implement this feature correctly! Or, at least, *more* correctly. *And*, for the first time, we will *update* an *existing* row in the database.

Back in ArticleController, we need to use the slug to query for the Article object. But, remember, there's a shortcut for this: replace the $slug argument with Article $article:

75 lines | src/Controller/ArticleController.php

```
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart(Article $article, LoggerInterface $logger, EntityManagerInterface $em)
    {
... lines 67 - 72
    }
}
```

Thanks to the type-hint, Symfony will automatically try to find an Article with this slug.

Then, to update the heartCount, just $article->setHeartCount() and then $article->getHeartCount() + 1:

75 lines | src/Controller/ArticleController.php

```
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart(Article $article, LoggerInterface $logger, EntityManagerInterface $em)
    {
        $article->setHeartCount($article->getHeartCount() + 1);
... lines 68 - 72
    }
}
```

Side note, it's not important for this tutorial, but in a high-traffic system, this could introduce a *race* condition. Between the time this article is queried for, and when it saves, 10 other people might have also liked the article. And that would mean that this would actually save the old, wrong number, effectively removing the 10 hearts that occurred during those microseconds.

Anyways, at the bottom, instead of the random number, use $article->getHeartCount():

75 lines | src/Controller/ArticleController.php

```
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart(Article $article, LoggerInterface $logger, EntityManagerInterface $em)
    {
        $article->setHeartCount($article->getHeartCount() + 1);
... lines 68 - 71
        return new JsonResponse(['hearts' => $article->getHeartCount()]);
    }
}
```

So, now, to the *key* question: how do we run an UPDATE query in the database? Actually, it's the *exact* same as inserting a *new* article. Fetch the entity manager like normal: EntityManagerInterface $em:

75 lines | src/Controller/ArticleController.php

```
... lines 1 - 8
use Doctrine\ORM\EntityManagerInterface;
... lines 10 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart(Article $article, LoggerInterface $logger, EntityManagerInterface $em)
    {
... lines 67 - 72
    }
}
```

Then, after updating the object, just call $em->flush():

75 lines | src/Controller/ArticleController.php

```
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart(Article $article, LoggerInterface $logger, EntityManagerInterface $em)
    {
        $article->setHeartCount($article->getHeartCount() + 1);
        $em->flush();

        $logger->info('Article is being hearted!');

        return new JsonResponse(['hearts' => $article->getHeartCount()]);
    }
}
```

But wait! I did *not* call $em->persist($article). We *could* call this... it's just not needed for updates! When you query Doctrine for an object, it *already* knows that you want that object to be saved to the database when you call flush(). Doctrine is *also* smart enough to know that it should *update* the object, instead of inserting a new one.

Ok, go back and refresh! Here is the real heart count for this article: 88. Click the heart and... yea! 89! And if you refresh, it stays! We can do 90, 91, 92, 93, and forever! And yea... this is not *quite* realistic yet. On a real site, I should only be able to like this article *one* time. But, we'll need to talk about users and security before we can do that.

Smarter Entity Method

Now that this is working, we can improve it! In the controller, we wrote some code to increment the heart count by one:

75 lines | src/Controller/ArticleController.php

```
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart(Article $article, LoggerInterface $logger, EntityManagerInterface $em)
    {
        $article->setHeartCount($article->getHeartCount() + 1);
... lines 68 - 72
    }
}
```

But, whenever possible, it's better to move code *out* of your controller. *Usually* we do this by creating a new service class and putting the logic there. But, if the logic is simple, it can sometimes live inside your *entity* class. Check this out: open Article, scroll to the bottom, and add a new method: public function incrementHeartCount(). Give it no arguments and return self, like our other methods:

```
156 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 131
    public function incrementHeartCount(): self
... lines 133 - 154
}
```

Then, $this->heartCount = $this->heartCount + 1:

```
156 lines | src/Entity/Article.php
... lines 1 - 131
    public function incrementHeartCount(): self
    {
        $this->heartCount = $this->heartCount + 1;

        return $this;
    }
... lines 138 - 156
```

Back in ArticleController, we can simplify to $article->incrementHeartCount():

```
75 lines | src/Controller/ArticleController.php
... lines 1 - 16
class ArticleController extends AbstractController
{
... lines 19 - 61
    /**
     * @Route("/news/{slug}/heart", name="article_toggle_heart", methods={"POST"})
     */
    public function toggleArticleHeart(Article $article, LoggerInterface $logger, EntityManagerInterface $em)
    {
        $article->incrementHeartCount();
... lines 68 - 72
    }
}
```

That's so nice. This moves the logic to a better place, and, it *reads* really well:

> Hello Article: I would like you to increment your heart count. Thanks!

Smart Versus Anemic Entities

And... this touches on a somewhat controversial topic related to entities. Notice that *every* property in the entity has a getter and setter method. This makes our entity *super* flexible: you can get or set any field you need.

But, sometimes, you might *not* need, or even *want* a getter or setter method. For example, do we really want a setHeartCount() method?

```
154 lines | src/Entity/Article.php
... lines 1 - 9
class Article
{
... lines 12 - 124
    public function setHeartCount(int $heartCount): self
    {
        $this->heartCount = $heartCount;

        return $this;
    }
... lines 131 - 152
}
```

I mean, should any part of the app *ever* need to change this? Probably not: they should just call our more descriptive incrementHeartCount() instead:

```
156 lines | src/Entity/Article.php
... lines 1 - 124
    public function setHeartCount(int $heartCount): self
    {
        $this->heartCount = $heartCount;

        return $this;
    }
... lines 131 - 156
```

I *am* going to keep it, because we use it to generate our fake data, but I want you to *really* think about this point.

By removing unnecessary getter or setter methods, and replacing them with more descriptive methods that fit your business logic, you can, little-by-little, give your entities more clarity. Some people take this to an extreme and have almost zero getters and setters. Here at KnpU, we tend to be more pragmatic: we *usually* have getters and setters, but we always look for ways to be more descriptive.

Next, our dummy article data is boring, and we're creating it in a hacky way:

61 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 10
class ArticleAdminController extends AbstractController
{
    /**
     * @Route("/admin/article/new")
     */
    public function new(EntityManagerInterface $em)
    {
        $article = new Article();
        $article->setTitle('Why Asteroids Taste Like Bacon')
            ->setSlug('why-asteroids-taste-like-bacon-'.rand(100, 999))
            ->setContent(<<<EOF
Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
**turkey** shank eu pork belly meatball non cupim.

Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
occaecat lorem meatball prosciutto quis strip steak.

Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
fugiat.
EOF
        );

        // publish most articles
        if (rand(1, 10) > 2) {
            $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
        }

        $article->setAuthor('Mike Ferengi')
            ->setHeartCount(rand(5, 100))
            ->setImageFilename('asteroid.jpeg')
        ;

        $em->persist($article);
        $em->flush();

        return new Response(sprintf(
            'Hiya! New Article id: #%d slug: %s',
            $article->getId(),
            $article->getSlug()
        ));
    }
}
```

Let's build an awesome fixtures system instead.

Chapter 14: Fixtures: Seeding Dummy Data!

We're creating our dummy article data in a *really*... uh... dummy way: with a, sort of, "secret" endpoint that creates an almost-identical article, each time we go there:

61 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 10
class ArticleAdminController extends AbstractController
{
    /**
     * @Route("/admin/article/new")
     */
    public function new(EntityManagerInterface $em)
    {
        $article = new Article();
        $article->setTitle('Why Asteroids Taste Like Bacon')
            ->setSlug('why-asteroids-taste-like-bacon-'.rand(100, 999))
            ->setContent(<<<EOF
Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
**turkey** shank eu pork belly meatball non cupim.

Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
occaecat lorem meatball prosciutto quis strip steak.

Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
fugiat.
EOF
        );

        // publish most articles
        if (rand(1, 10) > 2) {
            $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
        }

        $article->setAuthor('Mike Ferengi')
            ->setHeartCount(rand(5, 100))
            ->setImageFilename('asteroid.jpeg')
        ;

        $em->persist($article);
        $em->flush();

        return new Response(sprintf(
            'Hiya! New Article id: #%d slug: %s',
            $article->getId(),
            $article->getSlug()
        ));
    }
}
```

Honestly, it's not great for development: every article on the homepage pretty much looks the same.

Yep, our dummy data sucks. And, that's important! If we could load a *rich* set of random data easily, we could develop and debug faster.

Installing DoctrineFixturesBundle

To help with this, we'll use a great library called DoctrineFixturesBundle... but with our own spin to make things *really* fun.

First let's get it installed. Find your terminal and run

```
$ composer require orm-fixtures:3.0.2 --dev
```

And yep, this is a Flex *alias*, and we're using --dev because this tool will help us load *fake* data for development... which is *not* something we need in our production code. If you've ever accidentally replaced the production database with dummy data... you know what I mean.

Generating Fixture with make:fixtures

Perfect! When it finishes, generate our *first* fixture class by running:

```
$ php bin/console make:fixtures
```

Call it ArticleFixtures. It's *fairly* common to have one fixture class per entity, or sometimes, per group of entities. And... done!

Go check it out: src/DataFixtures/ArticleFixtures.php:

18 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 2
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

class ArticleFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // $product = new Product();
        // $manager->persist($product);

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

The idea behind fixtures is *dead* simple: step (1) we write code to create and save objects, and then step (2), we run a new console command that executes all of our fixture classes.

Writing the Fixtures

Open ArticleAdminController: let's start stealing some code! Copy *all* of our dummy article code, go back to the fixture class, and paste! We need to re-type the e on Article and hit tab so that PhpStorm adds the use statement for us on top:

```
51 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 4
use App\Entity\Article;
... lines 6 - 8
class ArticleFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $article = new Article();
... lines 14 - 48
    }
}
```

Then, at the bottom, the entity manager variable is called $manager:

```
51 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
class ArticleFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $article = new Article();
... lines 14 - 45
        $manager->persist($article);

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

Back in the controller, just put a die('todo') for now:

```
27 lines | src/Controller/ArticleAdminController.php
... lines 1 - 10
class ArticleAdminController extends AbstractController
{
    /**
     * @Route("/admin/article/new")
     */
    public function new(EntityManagerInterface $em)
    {
        die('todo');
... lines 19 - 24
    }
}
```

Someday, we'll create a proper admin form here.

And... that's it! It's super boring and it only creates one article... but it should work! Try it: find your terminal and run a new console command:

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

This will ask if you want to continue because - important note! - the command will *empty* the database first, and *then* load fresh data. Again, *not* something you want to run on production... not saying I've done that before.

When it finishes, find your browser, and refresh. It works!

Creating Multiple Articles

But... come on! We're going to need more than *one* article! How can we create multiple? First, we're going to do it the easy... but, kinda boring way. A for loop! Say: for $i = 0; $i < 10; $i++. And, *all* the way at the bottom, add the ending curly brace. We need to call persist() in the loop, but we only need to call flush() once at the end.

Cool! Try it again:

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

Then, refresh! Awesome! Except that the articles are still *totally* boring and identical... we'll talk about that in the next chapter.

BaseFixture Class for Cooler Looping

But first, let me show you a *cooler* way to create multiple articles. In the DataFixtures directory, create a new class called BaseFixture. Make it abstract, and extend the normal class that all fixtures extend... so... Fixture:

22 lines | src/DataFixtures/BaseFixture.php

```
... lines 1 - 2
namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
... lines 6 - 7
abstract class BaseFixture extends Fixture
{
... lines 10 - 20
}
```

Here's the idea: this will *not* be a fixture class that the bundle will execute. Instead, it will be a *base* class with some cool helper methods. To start, copy the load() method and implement it here. Re-type ObjectManager to get its use statement:

22 lines | src/DataFixtures/BaseFixture.php

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

abstract class BaseFixture extends Fixture
{
... lines 10 - 14
    public function load(ObjectManager $manager)
    {
... lines 17 - 19
    }
}
```

Oh, ObjectManager is an interface implemented by EntityManager, it's not too important: just think "this is the entity manager".

Next, and this won't make sense *yet*, create a private $manager property, and set it inside the load() method:

```
22 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 7
abstract class BaseFixture extends Fixture
{
... line 10
    private $manager;
... lines 12 - 14
    public function load(ObjectManager $manager)
    {
        $this->manager = $manager;
... lines 18 - 19
    }
}
```

Finally, create an abstract protected function called loadData() with that same ObjectManager argument:

```
22 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 7
abstract class BaseFixture extends Fixture
{
... lines 10 - 12
    abstract protected function loadData(ObjectManager $manager);
... lines 14 - 20
}
```

Back in load(), call this: $this->loadData($manager):

```
22 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 7
abstract class BaseFixture extends Fixture
{
... lines 10 - 14
    public function load(ObjectManager $manager)
    {
... lines 17 - 18
        $this->loadData($manager);
    }
}
```

So far, this doesn't do anything special. Back in ArticleFixtures, extend the new BaseFixture instead. I'll also cleanup the extra use statement:

```
50 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 4
use App\Entity\Article;
use Doctrine\Common\Persistence\ObjectManager;

class ArticleFixtures extends BaseFixture
{
... lines 10 - 48
}
```

Now, instead of implementing load(), implement loadData() and make it protected:

```
50 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
... lines 12 - 47
    }
}
```

And... yea! The fixture system will call load() on BaseFixture, that will call loadData() on ArticleFixtures and... well... everything will work exactly like before.

Adding the createMany Method

So... why did we just do this? Go back to the BaseFixture class and, at the bottom, I'm going to paste in a little method that I created:

```
34 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 7
abstract class BaseFixture extends Fixture
{
... lines 10 - 21
    protected function createMany(string $className, int $count, callable $factory)
    {
        for ($i = 0; $i < $count; $i++) {
            $entity = new $className();
            $factory($entity, $i);

            $this->manager->persist($entity);
            // store for usage later as App\Entity\ClassName_#COUNT#
            $this->addReference($className . '_' . $i, $entity);
        }
    }
}
```

Oh, and to make PhpStorm happy, at the top, add some PHPDoc that the $manager property is an ObjectManager instance:

```
34 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 5
use Doctrine\Common\Persistence\ObjectManager;

abstract class BaseFixture extends Fixture
{
    /** @var ObjectManager */
    private $manager;
... lines 12 - 32
}
```

Anyways, say hello to createMany()! A simple method that we can call to create multiple instances of an object. Here's the idea: we call createMany() and pass it the class we want to create, how *many* we want to create, and a callback method that will be called each time an object is created. That'll be our chance to load that object with data.

Basically, it does the for loop for us... which is not a *huge* deal, except for two nice things. First, it calls persist() for us, so we don't have to:

```
34 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 7
abstract class BaseFixture extends Fixture
{
... lines 10 - 21
    protected function createMany(string $className, int $count, callable $factory)
    {
        for ($i = 0; $i < $count; $i++) {
... lines 25 - 27
            $this->manager->persist($entity);
... lines 29 - 30
        }
    }
}
```

Ok, cool, but not *amazing*. But, this last line *is* cool:

```
34 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 7
abstract class BaseFixture extends Fixture
{
... lines 10 - 21
    protected function createMany(string $className, int $count, callable $factory)
    {
        for ($i = 0; $i < $count; $i++) {
... lines 25 - 28
            // store for usage later as App\Entity\ClassName_#COUNT#
            $this->addReference($className . '_' . $i, $entity);
        }
    }
}
```

It won't matter yet, but in a future tutorial, we will have *multiple* fixtures classes. When we do, we will need to be able to reference objects created in one fixture class from *other* fixture classes. By calling addReference(), all of our objects are automatically stored and ready to be fetched with a key that's their class name plus the index number.

The point is: this is going to save us some serious work... but not until the *next* tutorial.

Back in ArticleFixtures, use the new method: $this->createMany() passing it Article::class, 10, and a function:

```
49 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
... lines 13 - 43
        });
... lines 45 - 46
    }
}
```

This will have two args: the Article that was just created and a count of which article this is. Inside the method, we can remove the $article = new Article(), and instead of a random number on the slug, we can use $count:

```
49 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
            $article->setTitle('Why Asteroids Taste Like Bacon')
                ->setSlug('why-asteroids-taste-like-bacon-'.$count)
                ->setContent(<<<EOF
Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
**turkey** shank eu pork belly meatball non cupim.

Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
occaecat lorem meatball prosciutto quis strip steak.

Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
fugiat.
EOF
            );

            // publish most articles
            if (rand(1, 10) > 2) {
                $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
            }

            $article->setAuthor('Mike Ferengi')
                ->setHeartCount(rand(5, 100))
                ->setImageFilename('asteroid.jpeg')
            ;
        });
... lines 45 - 46
    }
}
```

At the bottom, the persist isn't hurting anything, but it's not needed anymore.

Finish the end with a closing parenthesis and a semicolon:

49 lines | src/DataFixtures/ArticleFixtures.php

```php
// ... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
            $article->setTitle('Why Asteroids Taste Like Bacon')
                ->setSlug('why-asteroids-taste-like-bacon-'.$count)
                ->setContent(<<<EOF
Spicy **jalapeno bacon** ipsum dolor amet veniam shank in dolore. Ham hock nisi landjaeger cow,
lorem proident [beef ribs](https://baconipsum.com/) aute enim veniam ut cillum pork chuck picanha. Dolore reprehenderit
labore minim pork belly spare ribs cupim short loin in. Elit exercitation eiusmod dolore cow
**turkey** shank eu pork belly meatball non cupim.

Laboris beef ribs fatback fugiat eiusmod jowl kielbasa alcatra dolore velit ea ball tip. Pariatur
laboris sunt venison, et laborum dolore minim non meatball. Shankle eu flank aliqua shoulder,
capicola biltong frankfurter boudin cupim officia. Exercitation fugiat consectetur ham. Adipisicing
picanha shank et filet mignon pork belly ut ullamco. Irure velit turducken ground round doner incididunt
occaecat lorem meatball prosciutto quis strip steak.

Meatball adipisicing ribeye bacon strip steak eu. Consectetur ham hock pork hamburger enim strip steak
mollit quis officia meatloaf tri-tip swine. Cow ut reprehenderit, buffalo incididunt in filet mignon
strip steak pork belly aliquip capicola officia. Labore deserunt esse chicken lorem shoulder tail consectetur
cow est ribeye adipisicing. Pig hamburger pork belly enim. Do porchetta minim capicola irure pancetta chuck
fugiat.
EOF
            );

            // publish most articles
            if (rand(1, 10) > 2) {
                $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
            }

            $article->setAuthor('Mike Ferengi')
                ->setHeartCount(rand(5, 100))
                ->setImageFilename('asteroid.jpeg')
            ;
        });

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

So, it's a *little* bit fancier, and it'll save that important reference for us. Let's try it! Reload the fixtures again:

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

No errors! Refresh the homepage: ah, our same, boring list of 10 identical articles. In the next chapter, let's use an awesome library called Faker to give each article rich, *unique*, realistic data.

Chapter 15: Using Faker for Seeding Data

The problem *now* is that our dummy data is super, duper boring. It's all the *same* stuff, over, and over again. Honestly, I keep falling asleep when I see the homepage. Obviously, as good PHP developers, you guys know that we could put some random code here and there to spice things up. I mean, we *do* already have a random $publishedAt date:

49 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
... lines 13 - 35
            if (rand(1, 10) > 2) {
                $article->setPublishedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
            }
... lines 39 - 43
        });
... lines 45 - 46
    }
}
```

But, instead of creating that random data by hand, there's a *much* cooler way. We're going to use a library called Faker. Google for "Faker PHP" to find the GitHub page from Francois Zaninotto. Fun fact, Francois was the *original* documentation lead for symfony 1. He's awesome.

Anyways, this library is all about creating dummy data. Check it out: you can use it to generate random names, random addresses, random text, random letters, numbers between this and that, paragraphs, street codes and even winning lottery numbers! Basically, it's awesome.

Installing Faker

So let's get it installed. Copy the composer require line, move over and paste. But, add the --dev at the end:

```
$ composer require fzaninotto/faker --dev
```

Because we're going to use this library for our fixtures only - it's not needed on production.

Setting up Faker

When that finishes, head back to its docs so we can see how to use it. Ok: we just need to say $faker = Faker\Factory::create(). Open our BaseFixture class: let's setup Faker in this, central spot. Create a new protected $faker property:

40 lines | src/DataFixtures/BaseFixture.php

```
... lines 1 - 9
abstract class BaseFixture extends Fixture
{
... lines 12 - 15
    protected $faker;
... lines 17 - 38
}
```

And down below, I'll say, $this->faker = and look for a class called Factory from Faker, and ::create():

```
40 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 6
use Faker\Factory;
... lines 8 - 9
abstract class BaseFixture extends Fixture
{
... lines 12 - 15
    protected $faker;
... lines 17 - 19
    public function load(ObjectManager $manager)
    {
        $this->manager = $manager;
        $this->faker = Factory::create();
... lines 24 - 25
    }
... lines 27 - 38
}
```

We should also add some PHPDoc above the property to help PhpStorm know what type of object it is. Hold Command - or Ctrl - and click the create() method: let's see what this returns exactly. Apparently, it returns a Generator.

Cool! Above the property, add /** @var Generator */ - the one from Faker:

```
40 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 7
use Faker\Generator;

abstract class BaseFixture extends Fixture
{
... lines 12 - 14
    /** @var Generator */
    protected $faker;
... lines 17 - 38
}
```

Perfect! Now, using Faker will be as easy as pie! Specifically, *eating* pie, cause, that's super easy.

Generating Fake Data

Open ArticleFixtures. We already have a little bit of randomness. But, Faker can even help here: change this to if $this->faker->boolean() where the first argument is the chance of getting true. Let's use 70: a 70% chance that each article will be published:

49 lines | src/DataFixtures/ArticleFixtures.php

```
// ... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
// ... lines 13 - 34
            // publish most articles
            if ($this->faker->boolean(70)) {
// ... line 37
            }
// ... lines 39 - 43
        });
// ... lines 45 - 46
    }
}
```

And below, we had this *long* expression to create a random date. *Now* say, $this->faker->dateTimeBetween('-100 days', '-1 days'):

49 lines | src/DataFixtures/ArticleFixtures.php

```
// ... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
// ... lines 13 - 34
            // publish most articles
            if ($this->faker->boolean(70)) {
                $article->setPublishedAt($this->faker->dateTimeBetween('-100 days', '-1 days'));
            }
// ... lines 39 - 43
        });
// ... lines 45 - 46
    }
}
```

I love it! Down for heartCount, use another Faker function: $this->faker->numberBetween(5, 100):

```
49 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
... lines 13 - 39
            $article->setAuthor('Mike Ferengi')
                ->setHeartCount($this->faker->numberBetween(5, 100))
... line 42
            ;
        });
... lines 45 - 46
    }
}
```

After these few improvements, let's make sure the system *is* actually as easy as pie. Find your terminal and run:

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

No errors and... back on the browser, it works! Of course, the *big* problem is that the title, author and article images are *always* the same. Snooze.

Faker *does* have methods to generate random titles, random names and even random images. But, the *more* realistic you make your fake data, the easier it will be to build real features for your app.

Generating Controller, Realistic Data

So here's the plan: go back to ArticleFixtures. At the top, I'm going to paste in a few static properties:

```
66 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
    private static $articleTitles = [
        'Why Asteroids Taste Like Bacon',
        'Life on Planet Mercury: Tan, Relaxing and Fabulous',
        'Light Speed Travel: Fountain of Youth or Fallacy',
    ];

    private static $articleImages = [
        'asteroid.jpeg',
        'mercury.jpeg',
        'lightspeed.png',
    ];

    private static $articleAuthors = [
        'Mike Ferengi',
        'Amy Oort',
    ];
... lines 26 - 64
}
```

These represent some realistic article titles, article images that exist, and two article authors. So, instead of making *completely* random titles, authors and images, we'll randomly choose from this list.

But even here, Faker can help us. For title, say $this->faker->randomElement() and pass self::$articleTitles:

66 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
... lines 10 - 26
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
            $article->setTitle($this->faker->randomElement(self::$articleTitles))
... lines 31 - 60
        });
... lines 62 - 63
    }
}
```

We'll let Faker do all the hard work.

For setSlug(), we *could* continue to use this, but there is also a $faker->slug method:

66 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
... lines 10 - 26
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
            $article->setTitle($this->faker->randomElement(self::$articleTitles))
                ->setSlug($this->faker->slug)
... lines 32 - 60
        });
... lines 62 - 63
    }
}
```

The slug will now be totally different than the article title, but honestly, who cares?

For author, do the same thing: $this->faker->randomElement() and pass self::$articleAuthors:

66 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
... lines 10 - 26
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
... lines 30 - 56
            $article->setAuthor($this->faker->randomElement(self::$articleAuthors))
... lines 58 - 59
            ;
        });
... lines 62 - 63
    }
}
```

Copy that, and repeat it one more time for the imageFile, this time using self::$articleImages:

66 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
... lines 10 - 26
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
... lines 30 - 56
            $article->setAuthor($this->faker->randomElement(self::$articleAuthors))
... line 58
                ->setImageFilename($this->faker->randomElement(self::$articleImages))
            ;
        });
... lines 62 - 63
    }
}
```

Awesome! Let's go reload those fixtures!

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

No errors! Find your browser and, try it! Oh, it's *so* much better.

If creating nice, random data seems like a small thing, it's not! Having rich data that you can easily load will increase your ability to create new features and fix bugs fast. It's *totally* worth it.

Next! Let's install a *really* cool library with automatic slugging super-powers.

Chapter 16: Sluggable & other Wonderful Behaviors

We're using Faker to generate a random slug for each dummy article. Thanks to this, back on the homepage, look at the URLs: they're *truly* random slugs: they have no relation to the title.

But, really, shouldn't the slug be generated from the title? What I mean is, if I set the Article's title, *something* should automatically convert that into a slug and make sure it's unique in the database. *We* shouldn't need to worry about doing that manually.

And... yea! There's a *really* cool library that can do this, and a *bunch* of other magic! Google for StofDoctrineExtensionsBundle, and then click into its documentation.

Ok, let me explain something: there is a normal, PHP library called DoctrineExtension, which can add a lot of different *behaviors* to your entities, like sluggable, where you automatically generate the slug from another field. Other behaviors include Loggable, where each change to an entity is tracked, or Blameable, where the user who created or updated an entity is automatically recorded.

Installing StofDoctrineExtensionsBundle

This *bundle* - StofDoctrineExtensionsBundle - helps to *integrate* that library into a Symfony project. Copy the composer require line, find your terminal, and paste!

Tip

The StofDoctrineExtensionBundle does not currently support Symfony 5. But a community member has created a fork that does. Use: composer require antishov/doctrine-extensions-bundle to download the fork instead. It's the same bundle, with same configuration, but with newer releases.

```
$ composer require antishov/doctrine-extensions-bundle
```

While that's working, let's go check out the documentation. This is a *wonderful* library, but its documentation is *confusing*. So, let's navigate to the parts we need. Scroll down to find a section called "Activate the extensions you want".

As we saw, there are a *lot* of different, possible behaviors. For performance reasons, when you install this bundle, you need to *explicitly* say which behaviors you want, like timestampable, by setting it to true.

Contrib Recipes

Move back to the terminal to see if things are done. Oh, interesting! It stopped! And it's asking us if we want to install the recipe for StofDoctrineExtensionsBundle. Hmm... that's weird... because Flex has already installed *many* other recipes *without* asking us a question like this.

But! It says that the recipe for *this* package comes from the "contrib" repository, which is open to community contributions. Symfony has *two* recipe repositories. The main repository is closely controlled for quality. The second - the "contrib" repository - has some basic checks, but the community can freely contribute recipes. For security reasons, when you download a package that installs a recipe from *that* repository, it will ask you first before installing it. And, there's a link if you want to review the recipe.

I'm going to say yes, permanently. *Now* the recipe installs.

Configuring Sluggable

Thanks to this, we now have a shiny new config/packages/stof_doctrine_extensions.yaml file:

5 lines | config/packages/stof_doctrine_extensions.yaml

```yaml
# Read the documentation: https://symfony.com/doc/current/bundles/StofDoctrineExtensionsBundle/index.html
# See the official DoctrineExtensions documentation for more details: https://github.com/Atlantic18/DoctrineExtensions/tree/master/doc
stof_doctrine_extensions:
    default_locale: en_US
```

This is where we need to enable the extensions we want. We want sluggable. We can use the example in the docs as a guide. Add orm, then default. The default is referring to the *default* entity manager... because some projects can actually have *multiple* entity managers. Then, sluggable: true:

8 lines | config/packages/stof_doctrine_extensions.yaml

```yaml
# Read the documentation: https://symfony.com/doc/current/bundles/StofDoctrineExtensionsBundle/index.html
# See the official DoctrineExtensions documentation for more details: https://github.com/Atlantic18/DoctrineExtensions/tree/master/doc
stof_doctrine_extensions:
    default_locale: en_US
    orm:
        default:
            sluggable: true
```

As *soon* as we do this... drumroll... absolutely nothing will happen. Ok, behind the scenes, the bundle *is* now looking for slug fields on our entities. But, we need a *little* bit more config to activate it for Article. Open that class and find the slug property.

Now, go *back* to the documentation. Another confusing thing about this bundle is that the documentation is split in two places: this page shows you how to configure the *bundle*... but *most* of the docs are in the *library*. Scroll up and find the DoctrineExtensions Documentation link.

Awesome. Click into sluggable.md. Down a bit... it tells us that to use this feature, we need to add an @Gedmo\Slug() annotation above the slug field. Let's do it! Use @Gedmo\Slug, then fields={"title"}:

156 lines | src/Entity/Article.php

```php
// ... lines 1 - 5
use Gedmo\Mapping\Annotation as Gedmo;
// ... lines 7 - 10
class Article
{
// ... lines 13 - 24
    /**
// ... line 26
     * @Gedmo\Slug(fields={"title"})
     */
    private $slug;
// ... lines 30 - 154
}
```

That's all we need! Back in ArticleFixtures, we no longer need to set the slug manually (remove it):

```
66 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
class ArticleFixtures extends BaseFixture
{
... lines 10 - 26
    public function loadData(ObjectManager $manager)
    {
        $this->createMany(Article::class, 10, function(Article $article, $count) {
            $article->setTitle($this->faker->randomElement(self::$articleTitles))
                ->setSlug($this->faker->slug)
... lines 32 - 49
            );
... lines 51 - 60
        });
... lines 62 - 63
    }
}
```

Try it out: find your terminal, and load those fixtures!

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

No errors! That's a *really* good sign, because the slug column *is* required in the database. Go back to the homepage and... refresh! Brilliant! The slug is clean and *clearly* based off of the title! As an added benefit, look at how some of these have a number on the end. The Sluggable behavior is making sure that each slug is *unique*. So, if a slug already exists in the database, it adds a -1 , -2, -3, etc. until it finds an open space.

Hello Doctrine Events

Side note: this feature is built on top of Doctrine's *event* system. Google for "Doctrine Event Subscriber". You'll find a page on the Symfony documentation that talks about this very important topic. We're not going to create our own event subscriber, but it's a really powerful idea. In this example, they talk about how you could use the event system to automatically update a search index, each time any entity is created or updated. Behind the scenes, the sluggable features works by adding an event listener that is called *right* before saving, or "flushing", any entity.

If you ever need to do something automatically when an entity is added, updated or removed, think of this system.

Next, let's find out how to rescue things when migrations go wrong!

Chapter 17: When Migrations Fail

My *other* favorite Doctrine Extension behavior is timestampable. Go back to the library's documentation and click to view the Timestampable docs.

Oh, it's so nice: with this behavior, we can add $createdAt and $updatedAt fields to our entity, and they will be automatically set. Believe me, this will save your *butt* sometime in the future when something happens on your site you can't *quite* explain. A mystery!

Adding the createdAt & updatedAt Fields

Ok, step 1: we need those 2 new fields. We could easily add them by hand, but let's generate them instead. Run:

```
$ php bin/console make:entity
```

Update the Article entity and add createdAt, as a datetime, and say "no" to nullable: this should *always* be populated. Do the same thing for updatedAt: it should *also* always be set: it will match createdAt when the entity is first saved. Hit enter to finish adding fields:

190 lines | src/Entity/Article.php

```
... lines 1 - 10
class Article
{
... lines 13 - 55
    /**
     * @ORM\Column(type="datetime")
     */
    private $createdAt;

    /**
     * @ORM\Column(type="datetime")
     */
    private $updatedAt;
... lines 65 - 165
    public function getCreatedAt(): ?\DateTimeInterface
    {
        return $this->createdAt;
    }

    public function setCreatedAt(?\DateTimeInterface $createdAt): self
    {
        $this->createdAt = $createdAt;

        return $this;
    }

    public function getUpdatedAt(): ?\DateTimeInterface
    {
        return $this->updatedAt;
    }

    public function setUpdatedAt(?\DateTimeInterface $updatedAt): self
    {
        $this->updatedAt = $updatedAt;

        return $this;
    }
}
```

Next, you guys know the drill, run:

```
$ php bin/console make:migration
```

Awesome! Move over and open that file. Yep, this looks good: an ALTER TABLE to add created_at and updated_at:

29 lines | src/Migrations/Version20180418130337.php

```
... lines 1 - 2
namespace DoctrineMigrations;

use Doctrine\DBAL\Migrations\AbstractMigration;
use Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20180418130337 extends AbstractMigration
{
    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 \'mys

        $this->addSql('ALTER TABLE article ADD created_at DATETIME NOT NULL, ADD updated_at DATETIME NOT NULL');
    }

    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 \'mys

        $this->addSql('ALTER TABLE article DROP created_at, DROP updated_at');
    }
}
```

Go *back* to your terminal, and run it:

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

When a Migration Fails

And... great! Wait, woh! No! It exploded! Check it out:

> Incorrect datetime value: 0000-00-00

Hmm. The problem is that our database *already* has articles. So when MySQL tries to create a new datetime column that is *not* nullable, it has a hard time figuring out what value to put for those existing rows!

Yep, unfortunately, *sometimes*, migrations fail. And fixing them is a delicate process. Let's think about this. What we *really* want to do is create those columns, but *allow* them to be null... at first. Then, we can *update* both fields to today's date. And, *then* we can use another ALTER TABLE query to finally make them not null.

That's *totally* doable! And we just need to modify the migration by hand. Instead of NOT NULL, use DEFAULT NULL. Do the same for updated_at:

```
30 lines | src/Migrations/Version20180418130337.php
... lines 1 - 10
class Version20180418130337 extends AbstractMigration
{
    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 \'mys

        $this->addSql('ALTER TABLE article ADD created_at DATETIME DEFAULT NULL, ADD updated_at DATETIME DEFAULT NULL
... line 19
    }
... lines 21 - 28
}
```

Below that, call $this->addSql() with:

```
UPDATE article SET created_at = NOW(), updated_at = NOW()
```

```
30 lines | src/Migrations/Version20180418130337.php
... lines 1 - 10
class Version20180418130337 extends AbstractMigration
{
    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 \'mys

        $this->addSql('ALTER TABLE article ADD created_at DATETIME DEFAULT NULL, ADD updated_at DATETIME DEFAULT NULL
        $this->addSql('UPDATE article SET created_at = NOW(), updated_at = NOW()');
    }
... lines 21 - 28
}
```

We *still* need another query to change things *back* to not null, but don't do it yet: we can be lazy. Instead, find your terminal: let's try the migration again. But, wait! You may or may *not* be able to re-run the migration immediately. In this case, the original migration had only *one* query, and that one query failed. This means that *no* part of the migration executed successfully.

But sometimes, a migration may contain *multiple* lines of SQL. And, if the second or third line fails, then, well, we're in a *really* weird state! In that situation, if we tried to *rerun* the migration, the first line would execute for the *second* time, and it would probably fail.

Basically, when a migration fails, it's possible that your migration system is now in an invalid state. *When* that happens, you should completely drop your database and start over. You can do that with:

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

And then:

```
php bin/console doctrine:database:create
```

And *then* you can migrate. Anyways, we are *not* in an invalid state: so we can just re-try the migration:

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

And *this* time, it works! To finally make the fields *not* nullable, we can ask Doctrine to generate a new migration:

```
$ php bin/console make:migration
```

Go check it out!

29 lines | src/Migrations/Version20180418130730.php

```
... lines 1 - 2
namespace DoctrineMigrations;

use Doctrine\DBAL\Migrations\AbstractMigration;
use Doctrine\DBAL\Schema\Schema;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
class Version20180418130730 extends AbstractMigration
{
    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 \'mys

        $this->addSql('ALTER TABLE article CHANGE created_at created_at DATETIME NOT NULL, CHANGE updated_at updated_at
    }

    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 \'mys

        $this->addSql('ALTER TABLE article CHANGE created_at created_at DATETIME DEFAULT NULL, CHANGE updated_at update
    }
}
```

Ha! Nice! It simply changes the fields to be NOT NULL. Run it!

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

And we are good! Now, back to Timestampable!

Chapter 18: Activating Timestampable

Ok, let's add Timestampable! First, we need to activate it, which again, is described *way* down on the bundle's docs. Open config/packages/stof_doctrine_extensions.yaml, and add timestampable: true:

9 lines | config/packages/stof_doctrine_extensions.yaml

```yaml
... lines 1 - 2
stof_doctrine_extensions:
    default_locale: en_US
    orm:
        default:
            sluggable: true
            timestampable: true
```

Second, your entity needs some annotations. For this, go back to the *library's* docs. Easy enough: we just need @Gedmo\Timestampable.

Back in our project, open Article and scroll down to find the new fields. Above createdAt, add @Timestampable() with on="create":

192 lines | src/Entity/Article.php

```php
... lines 1 - 10
class Article
{
... lines 13 - 55
    /**
     * @ORM\Column(type="datetime")
     * @Gedmo\Timestampable(on="create")
     */
    private $createdAt;
... lines 61 - 190
}
```

Copy that, paste above updatedAt, and use on="update":

192 lines | src/Entity/Article.php

```php
... lines 1 - 10
class Article
{
... lines 13 - 61
    /**
     * @ORM\Column(type="datetime")
     * @Gedmo\Timestampable(on="update")
     */
    private $updatedAt;
... lines 67 - 190
}
```

That should be it! Find your terminal, and reload the fixtures!

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

No errors... but, let's make sure it's actually working. Run:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article'
```

Yes! They *are* set! And each time we update, the updated_at will change.

The TimestampableEntity Trait

I *love* Timestampable. Heck, I put it *everywhere*. And, fortunately, there is a *shortcut*! Yea, we did *way* too much work.

Check it out: completely delete the createdAt and updatedAt fields that we so-carefully added. And, remove the getter and setter methods at the bottom too:

192 lines | src/Entity/Article.php

```
... lines 1 - 10
class Article
{
... lines 13 - 55
    /**
     * @ORM\Column(type="datetime")
     * @Gedmo\Timestampable(on="create")
     */
    private $createdAt;

    /**
     * @ORM\Column(type="datetime")
     * @Gedmo\Timestampable(on="update")
     */
    private $updatedAt;
... lines 67 - 167
    public function getCreatedAt(): ?\DateTimeInterface
    {
        return $this->createdAt;
    }

    public function setCreatedAt(?\DateTimeInterface $createdAt): self
    {
        $this->createdAt = $createdAt;

        return $this;
    }

    public function getUpdatedAt(): ?\DateTimeInterface
    {
        return $this->updatedAt;
    }

    public function setUpdatedAt(?\DateTimeInterface $updatedAt): self
    {
        $this->updatedAt = $updatedAt;

        return $this;
    }
}
```

But now, *all* the way on top, add use TimestampableEntity:

```
159 lines | src/Entity/Article.php
... lines 1 - 6
use Gedmo\Timestampable\Traits\TimestampableEntity;
... lines 8 - 11
class Article
{
    use TimestampableEntity;
... lines 15 - 157
}
```

Yea! Hold Command or Ctrl and click to see that. *Awesome*: this contains the *exact* same code that we had before! If you want Timestampable, *just* use this trait, generate a migration and... done!

And, talking about migrations, there *could* be some slight column differences between these columns and the original ones we created. Let's check that. Run:

```
$ php bin/console make:migration
```

```
No database changes were detected
```

Cool! The fields in the trait are identical to what we had before. That means that we can already test things with:

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

Thank you TimestampableEntity!

Up Next: Relations!

Ok guys! I hope you are *loving* Doctrine! We just got a *lot* of functionality fast. We have magic - like Timestampable & Sluggable - rich data fixtures, and a rocking migration system.

One thing that we have *not* talked about yet is production config. And... that's because it's already setup. The Doctrine recipe came with its own config/packages/prod/doctrine.yaml config file, which makes sure that anything that *can* be cached easily, *is* cached:

32 lines | config/packages/prod/doctrine.yaml

```yaml
doctrine:
    orm:
        metadata_cache_driver:
            type: service
            id: doctrine.system_cache_provider
        query_cache_driver:
            type: service
            id: doctrine.system_cache_provider
        result_cache_driver:
            type: service
            id: doctrine.result_cache_provider

services:
    doctrine.result_cache_provider:
        class: Symfony\Component\Cache\DoctrineProvider
        public: false
        arguments:
            - '@doctrine.result_cache_pool'
    doctrine.system_cache_provider:
        class: Symfony\Component\Cache\DoctrineProvider
        public: false
        arguments:
            - '@doctrine.system_cache_pool'

framework:
    cache:
        pools:
            doctrine.result_cache_pool:
                adapter: cache.app
            doctrine.system_cache_pool:
                adapter: cache.system
```

This means you get nice performance, out-of-the-box.

The other *huge* topic that we have *not* talked about yet is Doctrine *relations*. But, we should *totally* talk about those - they're awesome! So let's do that in our next tutorial, with foreign keys, join queries and high-fives so that we can create a *really* rich database.

Alright guys, seeya next time.