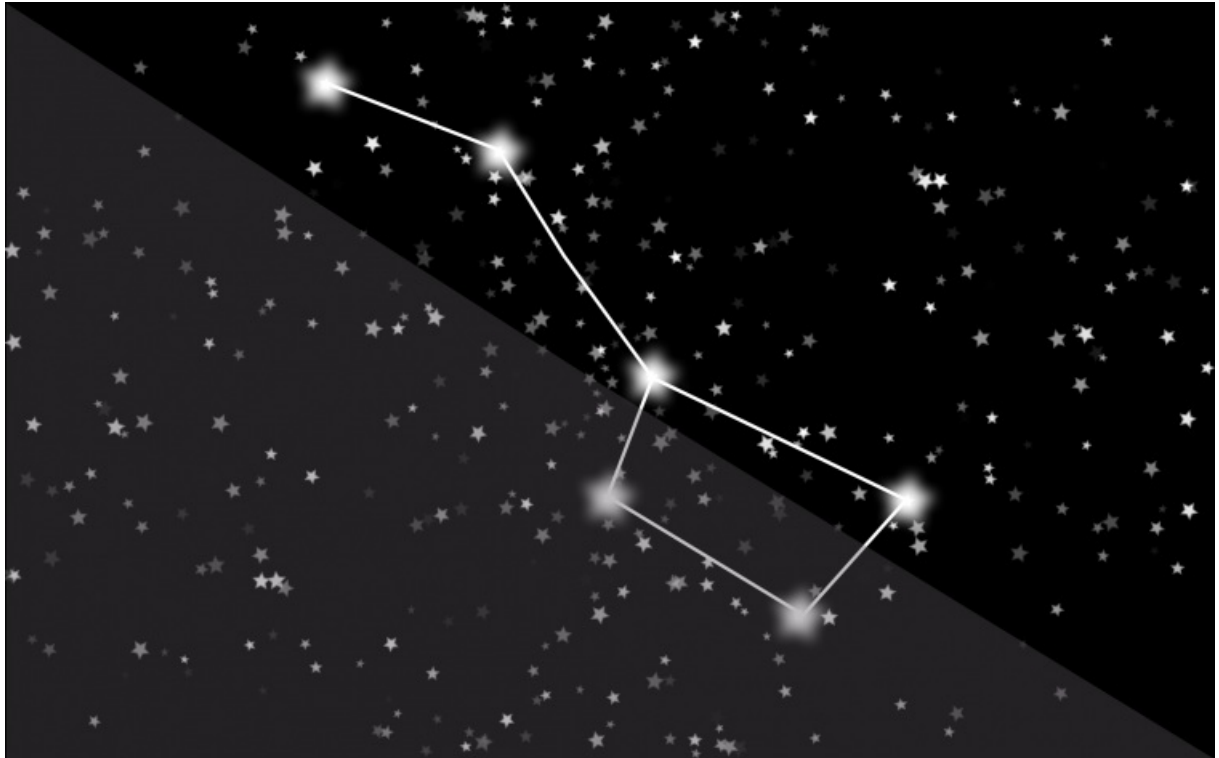


Mastering Doctrine Relations!



With <3 from SymfonyCasts

Chapter 1: Adding a Comment Entity

Hey friends! I mean, hello fellow space-traveling developer... friends. Welcome, to part *two* of our Doctrine tutorial where we talk *all* about... relationships. Oh, I *love* relationships, and there are so *many* beautiful types in the universe! Like, the relationship between two old friends, as they high-five after a grueling trip between solar systems. Or, the complex relationship between a planet and a moon: a perfect gravitational dance between BFF's. And of course, the most *incredible* type of relationship in *all* of the galaxy... database relationships.

Sure, we learned a *ton* about Doctrine in the first tutorial, but we *completely* avoided this topic! And it turns out, database relationships are *pretty* darn important if you want to build one of those "real" applications. So let's *crush* them.

Project Setup

As always, to have the *best* possible relationship with Doctrine, you should totally code along with me. Download the course code from this page. After you unzip the file, you'll find a `start/` directory that will have the same code you see here. Check out the `README.md` file for setup instructions, and the answer to this KnpU space riddle:

My name sounds white & fluffy, but I'm not! And instead of *blocking* the sun, I orbit it.

Need to know the answer? Then download the course code! Anyways, the last setup step will be to open a terminal, move into the project directory and run:

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

to start the built-in web server. Then, celebrate by finding your browser, and loading `http://localhost:8000`. Hello: The Space Bar! Our hot new app that helps spread *real* news to curious astronauts across the galaxy.

And thanks to the *last* tutorial, these articles are being loaded dynamically from the database. But... these comments at the bottom? Yea, *those* are still hardcoded. We need to fix that! And this will be our first relationship: each Article can have *many* Comments. But, more about that later.

Creating the Comment Entity

In the `src/Entity` directory, the *only* entity we have so far is Article:

```
159 lines | src/Entity/Article.php
... lines 1 - 8
9  /**
10 * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
11 */
12 class Article
13 {
... lines 14 - 157
158 }
```

So *before* we can talk about relationships, we *first* need to build a Comment entity. We *could* create this by hand, but the generator is so much nicer:

Open a new terminal tab and run:

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

Name the entity Comment. Then, for the fields, we need one for the author and one for the actual comment. Add `authorName` as a string field. And yea, *someday*, we might have a User table. And then, this could be a *relationship* to that table. But for

now, keep it as a simple string.

Next, add content as a text field, and also say no to nullable. Hit enter one more time to finish up.

Oh, but *before* we generate the migration, go open the new Comment class:

```
58 lines | src/Entity/Comment.php
... lines 1 - 2
3  namespace App\Entity;
4
5  use Doctrine\ORM\Mapping as ORM;
6
7  /**
8   * @ORM\Entity(repositoryClass="App\Repository\CommentRepository")
9   */
10 class Comment
11 {
12     /**
13      * @ORM\Id()
14      * @ORM\GeneratedValue()
15      * @ORM\Column(type="integer")
16      */
17     private $id;
18
19     /**
20      * @ORM\Column(type="string", length=255)
21      */
22     private $authorName;
23
24     /**
25      * @ORM\Column(type="text")
26      */
27     private $content;
28
29     public function getId()
30     {
31         return $this->id;
32     }
33
34     public function getAuthorName(): ?string
35     {
36         return $this->authorName;
37     }
38
39     public function setAuthorName(string $authorName): self
40     {
41         $this->authorName = $authorName;
42
43         return $this;
44     }
45
46     public function getContent(): ?string
47     {
48         return $this->content;
49     }
50
51     public function setContent(string $content): self
```

```
52     {
53         $this->content = $content;
54     }
55     return $this;
56 }
57 }
```

No surprises: id, authorName, content and some getter & setter methods. At the top of the class, let's add use TimestampableEntity:

```
61 lines | src/Entity/Comment.php
... lines 1 - 5
6 use Gedmo\Timestampable\Traits\TimestampableEntity;
7
8 /**
9  * @ORM\Entity(repositoryClass="App\Repository\CommentRepository")
10 */
11 class Comment
12 {
13     use TimestampableEntity;
14     ... lines 14 - 59
60 }
```

That will give us \$createdAt and \$updatedAt fields.

Now head back to your terminal and run:

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

When that finishes, go find the new file. We *just* want to make sure that this doesn't contain any surprises. For example, if you're working on multiple branches, then your database may be out-of-sync *before* you run make:migration. If that happens, the migration file would contain *extra* changes that you'll want to remove. In this case, it looks *great*:

```

29 lines | src/Migrations/Version20180426184910.php
... lines 1 - 2
3  namespace DoctrineMigrations;
4
5  use Doctrine\DBAL\Migrations\AbstractMigration;
6  use Doctrine\DBAL\Schema\Schema;
7
8  /**
9   * Auto-generated Migration: Please modify to your needs!
10  */
11  class Version20180426184910 extends AbstractMigration
12  {
13      public function up(Schema $schema)
14      {
15          // this up() migration is auto-generated, please modify it to your needs
16          $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
17
18          $this->addSql('CREATE TABLE comment (id INT AUTO_INCREMENT NOT NULL, author_name VARCHAR(255) NOT NULL, comment VARCHAR(255) NOT NULL, INDEX(id))');
19      }
20
21      public function down(Schema $schema)
22      {
23          // this down() migration is auto-generated, please modify it to your needs
24          $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
25
26          $this->addSql('DROP TABLE comment');
27      }
28  }

```

Go back to your terminal and, migrate!

```

$ php bin/console doctrine:migrations:migrate

```

Perfect! We have an article table and *now* a comment table. But, they are *not* friends yet. Time to add a relation!


Chapter 2: Adding the ManyToOne Relation

Hmm. We want each Article to have many Comments... and we want each Comment to *belong* to one Article. Forget about Doctrine for a minute: let's think about how this should look in the database. Because each Comment should *belong* to one article, this means that the comment table needs an `article_id` column.

So far, in order to add a new column to a table, we add a new property to the corresponding entity. And, at first, adding a relationship column is no different: we need a new property on Comment.

Generating the Relationship

And, just like before, when you want to add a new field to your entity, the *easiest* way is to use the generator. Run:



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

Type Comment so we can add the new field to it. But then, wait! This is a *very* important moment: it asks us for the new property's name. If you think that this should be something like `articleId`... that makes sense. But, surprise! It's wrong!

Instead, use `article`. I'll explain *why* soon. For the field type, we can use a "fake" option here called: `relation`: that will start a special wizard that will guide us through the relation setup process.

The first question is:

What class should this entity be related to?

Easy: Article. Now, it explains the *four* different types of relationships that exist in Doctrine: `ManyToOne`, `OneToMany`, `ManyToMany` and `OneToOne`. If you're not sure which relationship you need, you can read through the descriptions to find the one that fits best.

Check out the `ManyToOne` description:

Each comment relates to one Article

That sound perfect! And then:

Each Article can have many Comment objects

Brilliant! *This* is the relationship we need. In fact, it's the "king" of relationships: you'll probably create more `ManyToOne` relationships than any other.

Answer with: `ManyToOne`.

Now, it asks us if the `article` property on Comment is allowed to be null. Basically, it's asking us if it should be legal for a Comment to be saved to the database that is *not* related to an Article, so, with an `article_id` set to null. A Comment *must* have an article, so let's say no.

Generating the Other (Inverse) Side of the Relation

This next question is *really* important: do we want to add a new property to Article? Here's the deal: you can look at every relationship from two different sides. You could look at a Comment and ask for its one related Article. *Or*, you could look at an Article, and ask for its many related *comments*.

No matter *what* we answer here, we *will* be able to get or set the Article for a Comment object. But, if we *want*, the generator can *also* map the *other* side of the relationship. This is *optional*, but it means that we will be able to say `$article->getComments()` to get all of the Comments for an Article. There's no real downside to doing this, except having extra code if you *don't* need this convenience. But, this sounds pretty useful. In fact, we can use it to render the comments on the article page!

If this is making your head spin, don't worry! We'll talk more about this later. But most of the time, because it makes life easier,

you *will* want to generate both sides of a relationship. So let's say yes.


Then, for the *name* of this new property in Article, use the default: comments.

Finally, it asks you about something called orphanRemoval. Say no here. This topic is a bit more advanced, and you probably don't need orphanRemoval unless you're doing something complex with Symfony form collections. Oh, and we can easily update our code later to add this.

And... it's done! Hit enter one more time to exit. We did it!

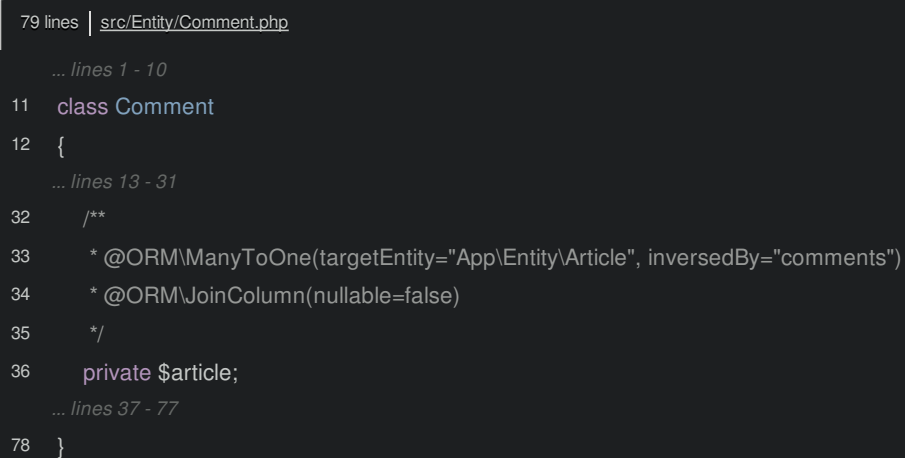
[Looking at the Entities](#)

Because I committed all of my changes before recording, I'll run:



```
$ git status
```

to see what this did. Cool! It updated *both* Article and Comment. Open the Comment class first:



```
79 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 31
32 /**
33  * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34  * @ORM\JoinColumn(nullable=false)
35  */
36 private $article;
... lines 37 - 77
78 }
```

Awesome! It added a new property called article, but instead of the normal @ORM\Column, it used @ORM\ManyToOne, with some options that point to the Article class. Then, at the bottom, we have getter and setter methods like normal:



```
79 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 66
67 public function getArticle(): ?Article
68 {
69     return $this->article;
70 }
71
72 public function setArticle(?Article $article): self
73 {
74     $this->article = $article;
75
76     return $this;
77 }
78 }
```

Now, check out the other side of the relationship, in Article entity. This has a new comments property:

```

202 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 60
61 /**
62  * @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="article")
63  */
64 private $comments;
... lines 65 - 200
201 }

```

And, near the bottom, *three* new methods: `getComments()`, `addComment()` and `removeComment()`:

```

202 lines | src/Entity/Article.php
... lines 1 - 5
6 use Doctrine\Common\Collections\Collection;
... lines 7 - 13
14 class Article
15 {
... lines 16 - 170
171 /**
172  * @return Collection|Comment[]
173  */
174 public function getComments(): Collection
175 {
176     return $this->comments;
177 }
178
179 public function addComment(Comment $comment): self
180 {
181     if (!$this->comments->contains($comment)) {
182         $this->comments[] = $comment;
183         $comment->setArticle($this);
184     }
185
186     return $this;
187 }
188
189 public function removeComment(Comment $comment): self
190 {
191     if ($this->comments->contains($comment)) {
192         $this->comments->removeElement($comment);
193         // set the owning side to null (unless already changed)
194         if ($comment->getArticle() === $this) {
195             $comment->setArticle(null);
196         }
197     }
198
199     return $this;
200 }
201 }

```

You *could* also add a `setComments()` method: but `addComment()` and `removeComment()` are usually more convenient:

[The ArrayCollection Object](#)

Oh, and there's one *little*, annoying detail that I need to point out. Whenever you have a relationship that holds a *collection* of items - like how an Article will relate to a *collection* of comments, you need to add a `__construct()` method and initialize that property to a new `ArrayCollection()`:

```
202 lines | src/Entity/Article.php
... lines 1 - 4
5   use Doctrine\Common\Collections\ArrayCollection;
... lines 6 - 13
14  class Article
15  {
... lines 16 - 65
66      public function __construct()
67      {
68          $this->comments = new ArrayCollection();
69      }
... lines 70 - 200
201 }
```

The generator took care of that for us. And, this looks scarier, or at least, more important than it really is. *Even* though the comments are set to an `ArrayCollection` object, I want you to think of that like a normal array. In fact, you can count, loop over, and pretty much treat the `$comments` property *exactly* like a normal array. The `ArrayCollection` is simply needed by Doctrine for internal reasons.

[ManyToOne Versus OneToMany](#)

Now, remember, we generated a *ManyToOne* relationship. We can see it inside `Comment`: the `article` property is a *ManyToOne* to `Article`. But, if you look at `Article`, huh. *It* has a *OneToMany* relationship back to `Comment`:

```
202 lines | src/Entity/Article.php
... lines 1 - 13
14  class Article
15  {
... lines 16 - 60
61      /**
62       * @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="article")
63       */
64      private $comments;
... lines 65 - 200
201 }
```

This is a really important thing. In reality, *ManyToOne* and *OneToMany* do *not* represent two different types of relationships! Nope, they describe the *same, one* relationship, just viewed from different sides.

[Generating the Migration](#)

Enough talking! Let's finally generate the migration. Find your terminal and run:

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

Go back to your editor and open that new migration file. Woh! Awesome! The end-result is *super* simple: it adds a new `article_id` column to the `comment` table along with a foreign key constraint to the `article`'s `id` column:

```

33 lines | src/Migrations/Version20180426185536.php
... lines 1 - 2
3 namespace DoctrineMigrations;
4
5 use Doctrine\DBAL\Migrations\AbstractMigration;
6 use Doctrine\DBAL\Schema\Schema;
7
8 /**
9  * Auto-generated Migration: Please modify to your needs!
10 */
11 class Version20180426185536 extends AbstractMigration
12 {
13     public function up(Schema $schema)
14     {
15         // this up() migration is auto-generated, please modify it to your needs
16         $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
17
18         $this->addSql('ALTER TABLE comment ADD article_id INT NOT NULL');
19         $this->addSql('ALTER TABLE comment ADD CONSTRAINT FK_9474526C7294869C FOREIGN KEY (article_id) REFERENCES article (id)');
20         $this->addSql('CREATE INDEX IDX_9474526C7294869C ON comment (article_id)');
21     }
22
23     public function down(Schema $schema)
24     {
25         // this down() migration is auto-generated, please modify it to your needs
26         $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
27
28         $this->addSql('ALTER TABLE comment DROP FOREIGN KEY FK_9474526C7294869C');
29         $this->addSql('DROP INDEX IDX_9474526C7294869C ON comment');
30         $this->addSql('ALTER TABLE comment DROP article_id');
31     }
32 }

```

So even though, in Comment, we called the property article:

```

79 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
13     ... lines 13 - 31
14
15     /**
16      * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
17      * @ORM\JoinColumn(nullable=false)
18      */
19     private $article;
20     ... lines 37 - 77
21 }

```

In the database, this creates an `article_id` column! Ultimately, the database looks *exactly* like we expected in the beginning! But in PHP, guess what? When we set this article property, we will set an entire Article *object* on it - *not* the Article's ID. More about that next.

The migration looks prefect. So find your terminal, and run it!



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

Ok, time to create a Comment object and learn how to relate it to an Article.

Chapter 3: Saving Relations

Our Comment entity has an article property and an article_id column in the database:

```
79 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 31
32 /**
33  * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34  * @ORM\JoinColumn(nullable=false)
35  */
36 private $article;
... lines 37 - 77
78 }
```

So, the question *now* is: how do we actually *populate* that column? How can we relate a Comment to an Article?

The answer is both very easy, and also, quite possibly, at first, weird! Open up the ArticleFixtures class. Let's hack in a new comment object near the bottom: `$comment1 = new Comment()`:

```
71 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 5
6 use App\Entity\Comment;
... lines 7 - 8
9 class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28 public function loadData(ObjectManager $manager)
29 {
30     $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 56
57         $article->setAuthor($this->faker->randomElement(self::$articleAuthors))
... lines 58 - 59
60         ;
61
62         $comment1 = new Comment();
... lines 63 - 65
66     });
... lines 67 - 68
69 }
70 }
```

Then, `$comment1->setAuthorName()`, and we'll go copy our *favorite*, always-excited astronaut commenter: Mike Ferengi:

```

71 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 61
62          $comment1 = new Comment();
63          $comment1->setAuthorName('Mike Ferengi');
... lines 64 - 65
66      });
... lines 67 - 68
69  }
70 }

```

Then, `$comment1->setContent()`, and use one of our hardcoded comments:

```

71 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 61
62          $comment1 = new Comment();
63          $comment1->setAuthorName('Mike Ferengi');
64          $comment1->setContent('I ate a normal rock once. It did NOT taste like bacon!');
... line 65
66      });
... lines 67 - 68
69  }
70 }

```

Perfect! Because we're creating this manually, we need to persist it to Doctrine. At the top, use the `$manager` variable:

```

71 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 65
66      });
... lines 67 - 68
69  }
70 }

```

Then, `$manager->persist($comment1)`:

```

71 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 61
62          $comment1 = new Comment();
63          $comment1->setAuthorName('Mike Ferengi');
64          $comment1->setContent('I ate a normal rock once. It did NOT taste like bacon!');
65          $manager->persist($comment1);
66      });
... lines 67 - 68
69  }
70  }

```

If we stop here, this *is* a valid Comment... but it is NOT related to *any* article. In fact, go to your terminal, and try the fixtures:

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

JoinColumn & Required Foreign Key Columns

Boom! It fails with an integrity constraint violation:

Column article_id cannot be null

It *is* trying to create the Comment, but, because we have not set the relation, it doesn't have a value for article_id:

```

79 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 31
32  /**
33   * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34   * @ORM\JoinColumn(nullable=false)
35   */
36  private $article;
... lines 37 - 77
78  }

```

Oh, and also, in Comment, see this JoinColumn with nullable=false? That's the same as having nullable=false on a property: it makes the article_id column *required* in the database. Oh, but, for whatever reason, a column *defaults* to nullable=false, and JoinColumn defaults to the opposite: nullable=true.

Setting the Article on the Comment

ANYways, how can we relate this Comment to the Article? By calling \$comment1->setArticle(\$article):

```

72 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 61
62          $comment1 = new Comment();
63          $comment1->setAuthorName('Mike Ferengi');
64          $comment1->setContent('I ate a normal rock once. It did NOT taste like bacon!');
65          $comment1->setArticle($article);
66          $manager->persist($comment1);
67      });
... lines 68 - 69
70  }
71  }

```

And that's it! This is both the most wonderful and strangest thing about Doctrine relations! We do *not* say `setArticle()` and pass it `$article->getId()`. Sure, it will *ultimately* use the id in the database, but in PHP, we *only* think about objects: relate the Article object to the Comment object.

Once again, Doctrine wants you to pretend like there is *no* database behind the scenes. Instead, all *you* care about is that a Comment object is related to an Article object. You expect Doctrine to figure out how to save that.


Copy that entire block, paste, and use it to create a second comment to make things a bit more interesting: `$comment2`. Copy a different dummy comment and paste that for the content:

```

78 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 61
62          $comment1 = new Comment();
63          $comment1->setAuthorName('Mike Ferengi');
64          $comment1->setContent('I ate a normal rock once. It did NOT taste like bacon!');
65          $comment1->setArticle($article);
66          $manager->persist($comment1);
67
68          $comment2 = new Comment();
69          $comment2->setAuthorName('Mike Ferengi');
70          $comment2->setContent('Woohoo! I'm going on an all-asteroid diet!');
71          $comment2->setArticle($article);
72          $manager->persist($comment2);
73      });
... lines 74 - 75
76  }
77  }


```

And *now*, let's see if it works! Reload the fixtures:



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

No errors! Great sign! Let's dig into the database:



```
$ php bin/console doctrine:query:sql 'SELECT * FROM comment'
```

There it is! We have 20 comments: 2 for each article. And the `article_id` for each row is set!

This is the *beauty* of Doctrine: *we* relate objects in PHP, never worrying about the foreign key columns. But of course, when we save, it stores things exactly like it should.

Next, let's learn how to *fetch* related data, to get all of the comments for a specific Article.

Chapter 4: Fetching Relations

Yes! Each Article is now related to *two* comments in the database. So, on the article show page, it's time to get rid of this hardcoded stuff and, finally, query for the *true* comments for this Article.

In `src/Controller`, open `ArticleController` and find the `show()` action:

```
75 lines | src/Controller/ArticleController.php
... lines 1 - 16
17 class ArticleController extends AbstractController
18 {
... lines 19 - 40
41 /**
42  * @Route("/news/{slug}", name="article_show")
43  */
44 public function show(Article $article, SlackClient $slack)
45 {
46     if ($article->getSlug() === 'khaaaaaan') {
47         $slack->sendMessage('Kahn', 'Ah, Kirk, my old friend...');
48     }
49
50     $comments = [
51         'I ate a normal rock once. It did NOT taste like bacon!',
52         'Woohoo! I\'m going on an all-asteroid diet!',
53         'I like bacon too! Buy some from my site! bakinsomebacon.com',
54     ];
55
56     return $this->render('article/show.html.twig', [
57         'article' => $article,
58         'comments' => $comments,
59     ]);
60 }
... lines 61 - 73
74 }
```

This renders a single article. So, how can we find *all* of the comments related to this article? Well, we *already* know *one* way to do this.

Remember: whenever you need to run a query, step one is to get that entity's repository. And, surprise! When we generated the `Comment` class, the `make:entity` command *also* gave us a new `CommentRepository`:

```

51 lines | src/Repository/CommentRepository.php
... lines 1 - 2
3  namespace App\Repository;
4
5  use App\Entity\Comment;
6  use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
7  use Symfony\Bridge\Doctrine\RegistryInterface;
8
9  /**
10   * @method Comment|null find($id, $lockMode = null, $lockVersion = null)
11   * @method Comment|null findOneBy(array $criteria, array $orderBy = null)
12   * @method Comment[]  findAll()
13   * @method Comment[]  findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
14   */
15  class CommentRepository extends ServiceEntityRepository
16  {
17      public function __construct(RegistryInterface $registry)
18      {
19          parent::__construct($registry, Comment::class);
20      }
21      ... lines 21 - 49
50  }

```

Thanks MakerBundle!

Get the repository by adding a CommentRepository argument. Then, let's see, could we use one of the built-in methods? Try `$comments = $commentRepository->findBy()`, and pass this article set to the entire `$article` object:

```

79 lines | src/Controller/ArticleController.php
... lines 1 - 6
7  use App\Repository\CommentRepository;
... lines 8 - 17
18  class ArticleController extends AbstractController
19  {
20      ... lines 20 - 41
42      /**
43       * @Route("/news/{slug}", name="article_show")
44       */
45      public function show(Article $article, SlackClient $slack, CommentRepository $commentRepository)
46      {
47          if ($article->getSlug() === 'khaaaaaan') {
48              ... line 48
49          }
50
51          $comments = $commentRepository->findBy(['article' => $article]);
52          ... lines 52 - 63
64      }
65      ... lines 65 - 77
78  }

```

Dump these comments and die:

```

79 lines | src/Controller/ArticleController.php
... lines 1 - 17
18 class ArticleController extends AbstractController
19 {
... lines 20 - 41
42 /**
43  * @Route("/news/{slug}", name="article_show")
44  */
45 public function show(Article $article, SlackClient $slack, CommentRepository $commentRepository)
46 {
... lines 47 - 50
51     $comments = $commentRepository->findBy(['article' => $article]);
52     dump($comments);die;
... lines 53 - 63
64 }
... lines 65 - 77
78 }

```

Then, find your browser and, try it! Yes! It returns the *two* Comment objects related to this Article!

So, the weird thing is that, once again, you need to stop thinking about the *columns* in your tables, like `article_id`, and only think about the *properties* on your entity classes. That's why we use `'article' => $article`. Of course, behind the scenes, Doctrine will make a query where `article_id = the id from this Article`. But, in PHP, we think *all* about objects.

Fetching Comments Directly from Article

As nice as this was... there is a *much* simpler way! When we generated the relationship, it asked us if we wanted to add an optional comments property to the Article class, for convenience. We said yes! And thanks to that, we can literally say `$comments = $article->getComments()`. Dump `$comments` again:

```

75 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 37
38 /**
39  * @Route("/news/{slug}", name="article_show")
40  */
41 public function show(Article $article, SlackClient $slack)
42 {
... lines 43 - 46
47     $comments = $article->getComments();
48     dump($comments);die;
... lines 49 - 59
60 }
... lines 61 - 73
74 }

```

Oh, and *now*, we don't need the `CommentRepository` anymore:

75 lines | [src/Controller/ArticleController.php](#)

```
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 37
38 /**
39  * @Route("/news/{slug}", name="article_show")
40  */
41 public function show(Article $article, SlackClient $slack)
42 {
... lines 43 - 59
60 }
... lines 61 - 73
74 }
```

Cool.

[Lazy Loading](#)

Head back to your browser and, refresh! It's the *exact* same as before. Wait, what? What's this weird PersistentCollection thing?

Here's what's going on. When Symfony queries for the Article, it *only* fetches the Article data: it does *not* automatically fetch the related Comments. And, for performance, that's great! We may not even *need* the comment data! But, as *soon* as we call `getComments()` and start using that, Doctrine makes a query in the background to go get the comment data.

This is called "lazy loading": related data is not queried for until, and unless, we use it. To make this magic possible, Doctrine uses this PersistentCollection object. This is *not* something you need to think or worry about: this object looks and acts like an array.

To prove it, let's foreach over `$comments` as `$comment` and dump each `$comment` inside. Put a die at the end:

78 lines | [src/Controller/ArticleController.php](#)

```
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 37
38 /**
39  * @Route("/news/{slug}", name="article_show")
40  */
41 public function show(Article $article, SlackClient $slack)
42 {
... lines 43 - 46
47     $comments = $article->getComments();
48     foreach ($comments as $comment) {
49         dump($comment);
50     }
51     die;
... lines 52 - 62
63 }
... lines 64 - 76
77 }
```

Try it again! Boom! Two Comment objects!

[Fetching the Comments in the Template](#)

Back in the controller, we *no* longer need these hard-coded comments. In fact, we don't even need to pass comments into the

template at all! That's because we can call the `getComments()` method directly from Twig!

Remove *all* of the comment logic:

```
65 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 37
38 /**
39  * @Route("/news/{slug}", name="article_show")
40  */
41 public function show(Article $article, SlackClient $slack)
42 {
43     if ($article->getSlug() === 'khaaaaaan') {
44         $slack->sendMessage('Kahn', 'Ah, Kirk, my old friend...');
45     }
46
47     return $this->render('article/show.html.twig', [
48         'article' => $article,
49     ]);
50 }
... lines 51 - 63
64 }
```

And then, jump into `templates/article/show.html.twig`. Scroll down a little... ah, yes! First, update the count: `article.comments|length`:

```
86 lines | templates/article/show.html.twig
... lines 1 - 4
5 {% block body %}
6
7 <div class="container">
8     <div class="row">
9         <div class="col-sm-12">
10             <div class="show-article-container p-3 mt-4">
... lines 11 - 39
40         <div class="row">
41             <div class="col-sm-12">
42                 <h3><i class="pr-3 fa fa-comment"></i>{{ article.comments|length }} Comments</h3>
... lines 43 - 71
72             </div>
73         </div>
74     </div>
75 </div>
76 </div>
77 </div>
78
79 {% endblock %}
... lines 80 - 86
```

Easy! Then, below, change the loop to use for comment in `article.comments`:

86 lines | [templates/article/show.html.twig](#)

... lines 1 - 4

5 {% block body %}

6

7 <div class="container">

8 <div class="row">

9 <div class="col-sm-12">

10 <div class="show-article-container p-3 mt-4">

... lines 11 - 39

40 <div class="row">

41 <div class="col-sm-12">

42 <h3><i class="pr-3 fa fa-comment"></i>{{ article.comments|length }} Comments</h3>

... lines 43 - 57

58 {% for comment in article.comments %}

... lines 59 - 69

70 {% endfor %}

71

72 </div>

73 </div>

74 </div>

75 </div>

76 </div>

77 </div>

78

79 {% endblock %}

... lines 80 - 86

And because each comment has a dynamic author, print that with {{ comment.authorName }}. And the content is now comment.content:

87 lines | [templates/article/show.html.twig](#)

... lines 1 - 4

5 {% block body %}

6

7 <div class="container">

8 <div class="row">

9 <div class="col-sm-12">

10 <div class="show-article-container p-3 mt-4">

... lines 11 - 39

40 <div class="row">

41 <div class="col-sm-12">

42 <h3><i class="pr-3 fa fa-comment"></i>{{ article.comments|length }} Comments</h3>

... lines 43 - 57

58 {% for comment in article.comments %}

59 <div class="row">

60 <div class="col-sm-12">

... line 61

62 <div class="comment-container d-inline-block pl-3 align-top">

63 {{ comment.authorName }}

... lines 64 - 65

66 {{ comment.content }}

... line 67

68 </div>

69 </div>

70 </div>

71 {% endfor %}

72

73 </div>

74 </div>

75 </div>

76 </div>

77 </div>

78 </div>

79

80 {% endblock %}

... lines 81 - 87

Oh, and, because each comment has a `createdAt`, let's print that too, with our `trusty ago` filter:

87 lines | [templates/article/show.html.twig](#)

... lines 1 - 4

5 {% block body %}

6

7 <div class="container">

8 <div class="row">

9 <div class="col-sm-12">

10 <div class="show-article-container p-3 mt-4">

... lines 11 - 39

40 <div class="row">

41 <div class="col-sm-12">

42 <h3><i class="pr-3 fa fa-comment"></i>{{ article.comments|length }} Comments</h3>

... lines 43 - 57

58 {% for comment in article.comments %}

59 <div class="row">

60 <div class="col-sm-12">

... line 61

62 <div class="comment-container d-inline-block pl-3 align-top">

63 {{ comment.authorName }}

64 <small>about {{ comment.createdAt|ago }}</small>

... line 65

66 {{ comment.content }}

... line 67

68 </div>

69 </div>

70 </div>

71 {% endfor %}

72

73 </div>

74 </div>

75 </div>

76 </div>

77 </div>

78 </div>

79

80 {% endblock %}

... lines 81 - 87

Love it! Let's try it! Go back, refresh and... yes! Two comments, from about 17 minutes ago. And, check this out: on the web debug toolbar, you can see that there are *two* database queries. The first query selects the article data only. And the *second* selects all of the *comment* data where *article_id* matches this article's id - 112. This second query doesn't actually happen *until* we reference the comments from inside of Twig:

87 lines | [templates/article/show.html.twig](#)

... lines 1 - 4

5 {% block body %}

6

7 <div class="container">

8 <div class="row">

9 <div class="col-sm-12">

10 <div class="show-article-container p-3 mt-4">

... lines 11 - 39

40 <div class="row">

41 <div class="col-sm-12">

42 <h3><i class="pr-3 fa fa-comment"></i>{{ article.comments|length }} Comments</h3>

... lines 43 - 57

58 {% for comment in article.comments %}

... lines 59 - 70

71 {% endfor %}

72

73 </div>

74 </div>

75 </div>

76 </div>

77 </div>

78 </div>

79

80 {% endblock %}

... lines 81 - 87

That laziness is a *key* feature of Doctrine relations.

Next, it's time to talk about the *subtle*, but super-important distinction between the *owning* and *inverse* sides of a relation.

Chapter 5: Owning Vs Inverse Relations

We need to talk about a very, very important and, honestly, super-confusing part of Doctrine relations! Listen closely: this is the *ugliest* thing we need to talk about. So, let's get through it, put a big beautiful check mark next to it, then move on!

It's called the owning versus inverse sides of a relationship, and it's deals with the fact that you can always look at a single relationship from two different directions. You can either look at the Comment and say that this has one article, so, this is ManyToOne to Article:

```
79 lines | src/Entity/Comment.php
... lines 1 - 7
8  /**
9   * @ORM\Entity(repositoryClass="App\Repository\CommentRepository")
10  */
11  class Comment
12  {
13      ... lines 13 - 31
32  /**
33   * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34   * @ORM\JoinColumn(nullable=false)
35   */
36   private $article;
37      ... lines 37 - 77
78 }
```

Or you can go to Article and - for that same one relationship, you can say that this Article has many comments:

```
202 lines | src/Entity/Article.php
... lines 1 - 10
11  /**
12   * @ORM\Entity(repositoryClass="App\Repository\ArticleRepository")
13   */
14  class Article
15  {
16      ... lines 16 - 60
61  /**
62   * @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="article")
63   */
64   private $comments;
65      ... lines 65 - 200
201 }
```

So, what's the big deal then? We already know that you can *read* data from either direction. You can say `$comment->getArticle()` or `$article->getComments()`. But, can you also *set* data on both sides? Well... that's where things get interesting.

Set the Inverse Side

In ArticleFixtures, we've proven that you *can* use `$comment->setArticle()` to set the relationship:

```

78 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9 class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28 public function loadData(ObjectManager $manager)
29 {
30     $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 64
65         $comment1->setArticle($article);
... lines 66 - 70
71         $comment2->setArticle($article);
... line 72
73     });
... lines 74 - 75
76 }
77 }

```

Everything persists perfectly to the database. But now, comment those out. Instead, set the data from the *other* direction: `$article->addComment($comment1)` and `$article->addComment($comment2)`:

```

81 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9 class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28 public function loadData(ObjectManager $manager)
29 {
30     $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 64
65         //$comment1->setArticle($article);
... lines 66 - 70
71         //$comment2->setArticle($article);
... lines 72 - 73
74         $article->addComment($comment1);
75         $article->addComment($comment2);
76     });
... lines 77 - 78
79 }
80 }

```

We're adding the comments to the comments collection on Article. By the way, don't worry that this code lives *after* the call to `persist()`. That's actually fine: the code just needs to come before `flush()`.

Anyways, let's try it! Find your terminal, sip some coffee, and reload the fixtures:

```

$ php bin/console doctrine:fixtures:load

```

Ok, no errors! Check the database:

```

$ php bin/console doctrine:query:sql 'SELECT * FROM comment'

```

Yea! It *does* still work! The comments saved correctly, *and* each has its `article_id` set.

So, I guess we found our answer: you can get and *set* data from *either* side of the relationship. Well... that's not actually true!

[Synchronizing the Owning Side](#)

Hold Command or Ctrl and click the `addComment()` method to jump into it:

```
202 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 178
179 public function addComment(Comment $comment): self
180 {
181     if (!$this->comments->contains($comment)) {
182         $this->comments[] = $comment;
183         $comment->setArticle($this);
184     }
185
186     return $this;
187 }
... lines 188 - 200
201 }
```

Look closely: this code was generated by the `make:entity` command. First, it checks to make sure the comment isn't *already* in the comments collection, just to avoid duplication on that property. Then, of course, it adds the comment to the `$comments` property. But *then*, it does something very important: it calls `$comment->setArticle($this)`.

Yep, this code *synchronizes* the data to the *other* side of the relationship. It makes sure that if you add this `Comment` to this `Article`, then the `Article` is also set on the `Comment`.

Let's try something: comment out the `setArticle()` call for a moment:

```
202 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 178
179 public function addComment(Comment $comment): self
180 {
181     if (!$this->comments->contains($comment)) {
182         $this->comments[] = $comment;
183         //$comment->setArticle($this);
184     }
185
186     return $this;
187 }
... lines 188 - 200
201 }
```

Then, go back to your terminal and reload the fixtures:

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

Woh! Explosion! When Doctrine tries to save the first comment, its `article_id` is empty! The relationship is *not* being set correctly!

[Owning Versus Inverse](#)

This is *exactly* what I wanted to talk about. Every relationship has two sides. One side is known as the *owning* side of the relation and the other is known as the *inverse* side of the relation. For a ManyToOne and OneToMany relation, the *owning* side is always the ManyToOne side:

```
79 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 31
32 /**
33  * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34  * @ORM\JoinColumn(nullable=false)
35  */
36 private $article;
... lines 37 - 77
78 }
```

And, it's easy to remember: the owning side is the side where the actual *column* appears in the database. Because the comment table will have the `article_id` column, the `Comment.article` property is the owning side. And so, `Article.comments` is the *inverse* side:

```
202 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 60
61 /**
62  * @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="article")
63  */
64 private $comments;
... lines 65 - 200
201 }
```

The reason this is so important is that, when you relate two entities together and save, Doctrine *only* looks at the *owning* side of the relationship to figure out what to persist to the database. Right now, we're *only* setting the *inverse* side! When Doctrine saves, it looks at the `article` property on `Comment` - the owning side - sees that it is null, and tries to save the `Comment` with *no* `Article`!

The owning side is the *only* side where the data matters when saving. In fact, the *entire* purpose of the *inverse* side of the relationship is just... convenience! It only exists because it's useful to be able to say `$article->getComments()`. That was *particularly* handy in the template.

The Inverse Side is Optional

Heck, the inverse side of a relationship is even optional! The `make:entity` command *asked* us if we wanted to generate the inverse side. We could delete *all* of the comments stuff from `Article`, and the relationship would still exist in the database *exactly* like it does now. And, we could still use it. We wouldn't have our fancy `$article->getComments()` shortcut anymore, but everything else would be fine.

I'm explaining this so that *you* can hopefully avoid a huge WTF moment in the future. If you ever try to relate two entities together and it is *not* saving, it may be due to this problem.

I'm going to uncomment the `setArticle()` call:

```

202 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 178
179 public function addComment(Comment $comment): self
180 {
181     if (!$this->comments->contains($comment)) {
182         $this->comments[] = $comment;
183         $comment->setArticle($this);
184     }
185
186     return $this;
187 }
... lines 188 - 200
201 }

```

In practice, when you use the `make:entity` generator, it takes care of this ugliness automatically, by generating code that synchronizes the owning side of the relationship when you set the *inverse* side. But, keep this concept in mind: it may eventually bite you!

Back in `ArticleFixtures`, refactor things back to `$comment->setArticle()`:

```

78 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9 class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28 public function loadData(ObjectManager $manager)
29 {
30     $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 64
65         $comment1->setArticle($article);
... lines 66 - 70
71         $comment2->setArticle($article);
... line 72
73     });
... lines 74 - 75
76 }
77 }

```

But, we know that, thanks to the code that was generated by `make:entity`, we *could* instead set the inverse side.

Next, let's setup our fixtures properly, and do some *cool* stuff to generate comments and articles that are randomly related to each other.

Chapter 6: Fixture References & Relating Objects

Having just *one* fixture class that loads articles *and* comments... and eventually other stuff, is not super great for organization:

```
78 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 61
62          $comment1 = new Comment();
... lines 63 - 67
68          $comment2 = new Comment();
... lines 69 - 72
73      });
... lines 74 - 75
76  }
77 }
```

Let's give the comments their *own* home. First, delete the comment code from here:

```
66 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 60
61      });
... lines 62 - 63
64  }
65 }
```

Then, find your terminal and run:

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

Call it CommentFixture.

Flip back to your editor and open that file!

```

18 lines | src/DataFixtures/CommentFixture.php
... lines 1 - 2
3  namespace App\DataFixtures;
4
5  use Doctrine\Bundle\FixturesBundle\Fixture;
6  use Doctrine\Common\Persistence\ObjectManager;
7
8  class CommentFixture extends Fixture
9  {
10     public function load(ObjectManager $manager)
11     {
12         // $product = new Product();
13         // $manager->persist($product);
14
15         $manager->flush();
16     }
17 }

```

In the last tutorial, we made a cool base class with some extra shortcuts. Extend BaseFixture:

```

17 lines | src/DataFixtures/CommentFixture.php
... lines 1 - 6
7  class CommentFixture extends BaseFixture
8  {
... lines 9 - 15
16 }

```

Then, instead of load, we *now* need loadData(), and it should be protected. Remove the extra use statement on top:

```

17 lines | src/DataFixtures/CommentFixture.php
... lines 1 - 4
5  use Doctrine\Common\Persistence\ObjectManager;
6
7  class CommentFixture extends BaseFixture
8  {
9      protected function loadData(ObjectManager $manager)
10     {
... lines 11 - 14
15     }
16 }

```

Thanks to our custom base class, we can create a *bunch* of comments easily with `$this->createMany()`, passing it `Comment::class`, 100, and then a callback that will receive each 100 Comment objects:

24 lines | [src/DataFixtures/CommentFixture.php](#)

```
... lines 1 - 4
5 use App\Entity\Comment;
... lines 6 - 7
8 class CommentFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(Comment::class, 100, function(Comment $comment) {
... lines 13 - 18
19         });
20
21         $manager->flush();
22     }
23 }
```

Inside, let's use Faker - which we also setup in the last tutorial - to give us awesome, fake data. Start with `$comment->setContent()`. I'll use multiple lines:

24 lines | [src/DataFixtures/CommentFixture.php](#)

```
... lines 1 - 7
8 class CommentFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(Comment::class, 100, function(Comment $comment) {
13             $comment->setContent(
... line 14
15             );
... lines 16 - 18
19         });
... lines 20 - 21
22     }
23 }
```

Now, if `$this->faker->boolean`, which will be a random true or false, then either generate a random paragraph: `$this->faker->paragraph`, or generate two random sentences. Pass true to get this as text, not an array:

24 lines | [src/DataFixtures/CommentFixture.php](#)

```
... lines 1 - 7
8 class CommentFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(Comment::class, 100, function(Comment $comment) {
13             $comment->setContent(
14                 $this->faker->boolean ? $this->faker->paragraph : $this->faker->sentences(2, true)
15             );
... lines 16 - 18
19         });
... lines 20 - 21
22     }
23 }
```

Cool! Next, for the author, we can use `$comment->setAuthor()` with `$this->faker->name`, to get a random person's name:

```

24 lines | src/DataFixtures/CommentFixture.php
... lines 1 - 7
8 class CommentFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(Comment::class, 100, function(Comment $comment) {
13             $comment->setContent(
14                 $this->faker->boolean ? $this->faker->paragraph : $this->faker->sentences(2, true)
15             );
16
17             $comment->setAuthorName($this->faker->name);
18
19         });
20
21     }
22 }
23 }

```

By the way, *all* of these faker functions are covered really well in their docs. I'm seriously not just making them up.

Finally, add `$comment->setCreatedAt()` with `$this->faker->dateTimeBetween()` from -1 months to -1 seconds:

```

24 lines | src/DataFixtures/CommentFixture.php
... lines 1 - 7
8 class CommentFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(Comment::class, 100, function(Comment $comment) {
13             $comment->setContent(
14                 $this->faker->boolean ? $this->faker->paragraph : $this->faker->sentences(2, true)
15             );
16
17             $comment->setAuthorName($this->faker->name);
18             $comment->setCreatedAt($this->faker->dateTimeBetween('-1 months', '-1 seconds'));
19         });
20
21     }
22 }
23 }

```

That'll give us *much* more interesting data.

Using the Reference System

At this point, this *is* a valid Comment object... we just haven't related it to an Article yet. We know *how* to do this, but... the problem is that all of the articles are created in a totally different fixture class. How can we get access to them here?

Well, one solution would be to use the entity manager, get the ArticleRepository, and run some queries to fetch out the articles.

But, that's kinda lame. So, there's an easier way. Look again at the BaseFixture class, specifically, the createMany() method:

40 lines | [src/DataFixtures/BaseFixture.php](#)

... lines 1 - 9

```
10 abstract class BaseFixture extends Fixture
11 {
    ... lines 12 - 27
28     protected function createMany(string $className, int $count, callable $factory)
29     {
30         for ($i = 0; $i < $count; $i++) {
31             $entity = new $className();
32             $factory($entity, $i);
33
34             $this->manager->persist($entity);
35             // store for usage later as App\Entity\ClassName_#COUNT#
36             $this->addReference($className . '_' . $i, $entity);
37         }
38     }
39 }
```

It's fairly simple, but it *does* have one piece of magic: it calls `$this->addReference()` with a key, which is the entity class name, an underscore, then an integer that starts at zero and counts up for each loop. For the second argument, it passes the object itself.

This reference system is a little "extra" built into Doctrine's fixtures library. When you add a "reference" from one fixture class, you can fetch it out in *another* class. It's *super* handy when you need to relate entities. And hey, that's *exactly* what we're trying to do!

Inside `CommentFixture`, add `$comment->setArticle()`, with `$this->getReference()` and pass it one of those keys: `Article::class`, then `_0`:

26 lines | [src/DataFixtures/CommentFixture.php](#)

... lines 1 - 4

```
5 use App\Entity\Article;
    ... lines 6 - 8
9 class CommentFixture extends BaseFixture
10 {
11     protected function loadData(ObjectManager $manager)
12     {
13         $this->createMany(Comment::class, 100, function(Comment $comment) {
    ... lines 14 - 19
20             $comment->setArticle($this->getReference(Article::class.'_0'));
21         });
    ... lines 22 - 23
24     }
25 }
```

PhpStorm is complaining about a type-mismatch, but this will totally work. Try it! Find your terminal and run:

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

No errors! That's a great sign! Check out the database:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM comment'
```

Yes! 100 comments, and each is related to the exact same article.

Relating to Random Articles

So, success! Except that this isn't very interesting yet. *All* our comments are related to the *same* one article? Come on!

Let's spice things up by relating each comment to a random article. *And*, learn about when we need to implement a `DependentFixtureInterface`.

Chapter 7: Awesome Random Fixtures

Look at ArticleFixtures: we created 10 articles. So, the system has references from 0 to 9:

```
66 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 8
9  class ArticleFixtures extends BaseFixture
10 {
... lines 11 - 27
28  public function loadData(ObjectManager $manager)
29  {
30      $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 31 - 60
61      });
... lines 62 - 63
64  }
65 }
```

In CommentFixture, spice things up: replace the 0 with `$this->faker->numberBetween(0, 9)`:

```
26 lines | src/DataFixtures/CommentFixture.php
... lines 1 - 8
9  class CommentFixture extends BaseFixture
10 {
11  protected function loadData(ObjectManager $manager)
12  {
13      $this->createMany(Comment::class, 100, function(Comment $comment) {
... lines 14 - 19
20          $comment->setArticle($this->getReference(Article::class.'_'.$this->faker->numberBetween(0, 9)));
21      });
... lines 22 - 23
24  }
25 }
```

Try the fixtures again:

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

No errors! And... check the database:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM comment'
```

That is *much* better! Just like that, each comment is related to a random article!

[Making this Random Reference System Reusable](#)

I really like this idea, where we can fetch random objects in our fixtures. So, let's make it easier! In BaseFixture, add a new private property on top called `$referencesIndex`. Set that to an empty array:

62 lines | [src/DataFixtures/BaseFixture.php](#)

```
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 17
18     private $referencesIndex = [];
... lines 19 - 60
61 }
```

I'm adding this because, at the bottom of this class, I'm going to paste in a new, method that I prepared. It's a little ugly, but this new getRandomReference() does exactly what its name says: you pass it a class, like the Article class, and it will find a random Article for you:

62 lines | [src/DataFixtures/BaseFixture.php](#)

```
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 17
18     private $referencesIndex = [];
... lines 19 - 41
42     protected function getRandomReference(string $className) {
43         if (!isset($this->referencesIndex[$className])) {
44             $this->referencesIndex[$className] = [];
45
46             foreach ($this->referenceRepository->getReferences() as $key => $ref) {
47                 if (strpos($key, $className.'.') === 0) {
48                     $this->referencesIndex[$className][] = $key;
49                 }
50             }
51         }
52
53         if (empty($this->referencesIndex[$className])) {
54             throw new \Exception(sprintf('Cannot find any references for class "%s"', $className));
55         }
56
57         $randomReferenceKey = $this->faker->randomElement($this->referencesIndex[$className]);
58
59         return $this->getReference($randomReferenceKey);
60     }
61 }
```

That's super friendly!

In CommentFixture, use it: `$comment->setArticle()` with `$this->getRandomReference(Article::class)`:

26 lines | [src/DataFixtures/CommentFixture.php](#)

```
... lines 1 - 8
9  class CommentFixture extends BaseFixture
10 {
11     protected function loadData(ObjectManager $manager)
12     {
13         $this->createMany(Comment::class, 100, function(Comment $comment) {
... lines 14 - 19
20         $comment->setArticle($this->getRandomReference(Article::class));
21     });
... lines 22 - 23
24 }
25 }
```

To make sure my function works, try the fixtures one last time:

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

And, query for the comments:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM comment'
```

Brilliant!

Fixture Ordering

There is *one* last minor problem with our fixtures... they only work due to pure luck. Check this out: I'll right click on CommentFixture and rename the class to A0CommentFixture:

32 lines | [src/DataFixtures/A0CommentFixture.php](#)

```
... lines 1 - 9
10 class A0CommentFixture extends BaseFixture implements DependentFixtureInterface
11 {
... lines 12 - 30
31 }
```

Also allow PhpStorm to rename the file. Some of you *might* already see the problem. Try the fixtures now:

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

Bah! Explosion!

Cannot find any references to App\Entity\Article

The error comes from BaseFixture and it basically means that *no* articles have been set into the reference system yet!

```

62 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 41
42     protected function getRandomReference(string $className) {
... lines 43 - 52
53         if (empty($this->referencesIndex[$className])) {
54             throw new \Exception(sprintf('Cannot find any references for class "%s"', $className));
55         }
... lines 56 - 59
60     }
61 }

```

You can see the problem in the file tree. We have *not* been thinking *at all* about what *order* each fixture class is executed. By default, it loads them alphabetically. But now, this is a problem! The A0CommentFixture class is being loaded *before* ArticleFixtures... which totally ruins our cool system!

You can also see this in the terminal: it loaded A0CommentFixture first.

DependentFixtureInterface

The solution is pretty cool. As *soon* as you have a fixture class that is *dependent* on *another* fixture class, you need to implement an interface called DependentFixtureInterface:

```

32 lines | src/DataFixtures/A0CommentFixture.php
... lines 1 - 6
7 use Doctrine\Common\DataFixtures\DependentFixtureInterface;
... lines 8 - 9
10 class A0CommentFixture extends BaseFixture implements DependentFixtureInterface
11 {
... lines 12 - 30
31 }

```

This will require you to have one method. Move to the bottom, then, go to the "Code" -> "Generate" menu, or Command + N on a Mac, select "Implement Methods" and choose getDependencies(). I'll add the public before the function:

```

32 lines | src/DataFixtures/A0CommentFixture.php
... lines 1 - 9
10 class A0CommentFixture extends BaseFixture implements DependentFixtureInterface
11 {
... lines 12 - 26
27     public function getDependencies()
28     {
... line 29
30     }
31 }

```

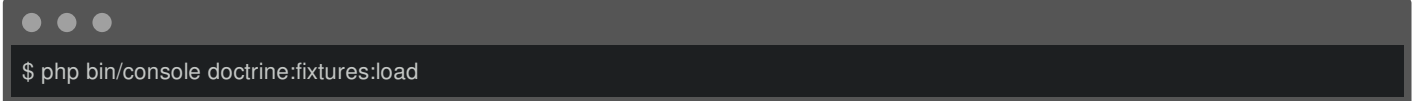
Just return an array with ArticleFixtures::class:

32 lines | [src/DataFixtures/A0CommentFixture.php](#)

... lines 1 - 9

```
10 class A0CommentFixture extends BaseFixture implements DependentFixtureInterface
11 {
    ... lines 12 - 26
27     public function getDependencies()
28     {
29         return [ArticleFixtures::class];
30     }
31 }
```

That's it! Load them again:



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

Bye bye error! It loaded ArticleFixtures *first* and then the comments below that. The fixtures library looks at all of the dependencies and figures out an order that makes sense.

With that fixed, let's rename the class *back* from this ridiculous name to CommentFixture:

32 lines | [src/DataFixtures/CommentFixture.php](#)

... lines 1 - 9

```
10 class CommentFixture extends BaseFixture implements DependentFixtureInterface
11 {
    ... lines 12 - 30
31 }
```

To celebrate, move over, refresh and... awesome! 8, random comments. We rock!

Next, let's learn about some tricks to control *how* Doctrine fetches the comments for an article, like, their *order*.

Chapter 8: OrderBy & fetch EXTRA_LAZY

It's *great* that each article now has *real*, dynamic comments at the bottom. But... something isn't right: the comments are in a, kind of, random order. Actually, they're just printing out in whatever order they were added to the database. But, that's silly! We need to print the *newest* comments on top, the oldest at the bottom.

How can we do this? Check out the template. Hmm, *all* we're doing is calling `article.comments`:

```
87 lines | templates/article/show.html.twig
... lines 1 - 6
7   <div class="container">
8     <div class="row">
9       <div class="col-sm-12">
10        <div class="show-article-container p-3 mt-4">
... lines 11 - 39
40        <div class="row">
41          <div class="col-sm-12">
... lines 42 - 57
58          {% for comment in article.comments %}
59          <div class="row">
60            <div class="col-sm-12">
61              
62              <div class="comment-container d-inline-block pl-3 align-top">
63                <span class="commenter-name">{{ comment.authorName }}</span>
64                <small>about {{ comment.createdAt|ago }}</small>
65                <br>
66                <span class="comment"> {{ comment.content }}</span>
67                <p><a href="#">Reply</a></p>
68              </div>
69            </div>
70          </div>
71          {% endfor %}
72
73        </div>
74      </div>
75    </div>
76  </div>
77 </div>
78 </div>
... lines 79 - 87
```

Which is the `getComments()` method on `Article`:

```

202 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 170
171 /**
172  * @return Collection|Comment[]
173  */
174 public function getComments(): Collection
175 {
176     return $this->comments;
177 }
... lines 178 - 200
201 }

```

The *great* thing about these relationship shortcut methods is that.... they're easy! The *downside* is that you don't have a lot of control over what's returned, like, the *order* of this article's comments.

Well... that's not entirely true. We *can* control the order, and I'll show you how. Actually, we can control a *lot* of things - but more on that later.

[@ORM\OrderBy\(\)](#)

Scroll all the way to the top and find the comments property:

```

202 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 60
61 /**
62  * @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="article")
63  */
64 private $comments;
... lines 65 - 200
201 }

```

Add a *new* annotation: `@ORM\OrderBy()` with `{"createdAt" = "DESC"}`:

```

203 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 60
61 /**
62  * @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="article")
63  * @ORM\OrderBy({"createdAt" = "DESC"})
64  */
65 private $comments;
... lines 66 - 201
202 }

```

That's it! Move over and, refresh! Brilliant! The *newest* comments are on top. This actually changed *how* Doctrine queries for the related comments.

Oh, and I want to mention *two* quick things about the syntax for annotations. First... well... the syntax can sometimes be confusing - where to put curly braces, equal sign etc. Don't sweat it: I still sometimes need to look up the correct syntax in

different situations.

Second, for better or worse, annotations *only* support double quotes. Yep, you simply *cannot* use single quotes. It just won't work.

Fetch EXTRA_LAZY

I want to show you one other trick. Go back to the homepage. It would be really nice to list the number of comments for each article. No problem! Open homepage.html.twig. Then, inside the articles loop, right after the title, add a `<small>` tag, a set of parentheses, and use `{{ article.comments|length }}` and then the word "comments":

```
59 lines | templates/article/homepage.html.twig
... lines 1 - 2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6
7              <!-- Article List -->
8
9              <div class="col-sm-12 col-md-8">
... lines 10 - 18
19          <!-- Supporting Articles -->
20
21          {% for article in articles %}
22              <div class="article-container my-1">
23                  <a href="{{ path('article_show', {slug: article.slug}) }}">
... line 24
25                      <div class="article-title d-inline-block pl-3 align-middle">
... line 26
27                          <small>({{ article.comments|length }} comments)</small>
... lines 28 - 30
31                      </div>
32                  </a>
33              </div>
34          {% endfor %}
35      </div>
... lines 36 - 55
56  </div>
57  </div>
58  {% endblock %}
```

I love it! Refresh the homepage. It works effortlessly! But... check out the queries down here on the web debug toolbar. If you click into it, there are suddenly 6 queries! The first query is what we expect: it finds all published articles.

The second query selects all of the comments for an article whose id is 176. The next is an *identical* query for article 177. As we loop over the articles, each time we call `getComments()`, at that moment, Doctrine fetches *all* of the comments for that specific Article. Then, it counts them.

This is a classic, *potential* performance issue with ORM's like Doctrine. It's called the N+1 problem. And, we'll talk about it later. But, it's basically that the cool lazy-loading of relationships can lead to an extra query per row. And this *may* cause performance issues.

But, forget about that for now, because, there's a *simpler* performance problem. We're querying for *all* of the comments for each article... simply to count them! That's insane!

This is the default behavior of Doctrine: as soon as you call `getComments()` and use that data, it makes a query at that moment to get all of the comment data, even if you eventually only need to *count* that data.

But, we can control this. In Article, at the end of the OneToMany annotation, add `fetch="EXTRA_LAZY"`:

```

203 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 60
61 /**
62  * @ORM\OneToMany(targetEntity="App\Entity\Comment", mappedBy="article", fetch="EXTRA_LAZY")
63  * @ORM\OrderBy({"createdAt" = "DESC"})
64  */
65 private $comments;
... lines 66 - 201
202 }

```

Now, go back to the page and refresh. We *still* have six queries, but go look at them. Awesome! Instead of selecting *all* of the comment data, they are super-fast COUNT queries!

Here's how this works: if you set `fetch="EXTRA_LAZY"`, and you simply *count* the result of `$article->getComments()`, then instead of querying for all of the comments, Doctrine does a quick COUNT query.

Awesome, right! You might think that it's *so* awesome that this should *always* be the way it works! But, there is *one* situation where this is *not* ideal. And actually, we have it! Go to the article show page.

Here, we count the comments first, and *then* we loop over them:

```

87 lines | templates/article/show.html.twig
... lines 1 - 4
5 {% block body %}
6
7 <div class="container">
8   <div class="row">
9     <div class="col-sm-12">
10      <div class="show-article-container p-3 mt-4">
... lines 11 - 39
40      <div class="row">
41        <div class="col-sm-12">
42          <h3><i class="pr-3 fa fa-comment"></i>{{ article.comments|length }} Comments</h3>
... lines 43 - 57
58          {% for comment in article.comments %}
... lines 59 - 70
71          {% endfor %}
72
73        </div>
74      </div>
75    </div>
76  </div>
77 </div>
78 </div>
79
80 {% endblock %}
... lines 81 - 87

```

Look at the profiler now. Thanks to EXTRA_LAZY, we have an extra query! It counts the comments... but then, right after, it queries for all of them anyways. *Before* we were using EXTRA_LAZY, this count query didn't exist.


So, sorry people, like life, everything is a trade-off. But, it's still probably a net-win for us. But *as always*, don't prematurely optimize. Deploy first, identify performance issues, and then solve them.

Chapter 9: Giving the Comments an isDeleted Flag

I want to show you a *really* cool, *really* powerful feature. But, to do that, we need to give our app a bit more depth. We need to make it possible to mark comments as *deleted*. Because, honestly, not *all* comments on the Internet are as insightful and amazing as the ones that *you* all add to KnpUniversity. You all are *seriously* the best! But, instead of *actually* deleting them, we want to keep a record of deleted comments, just in case.

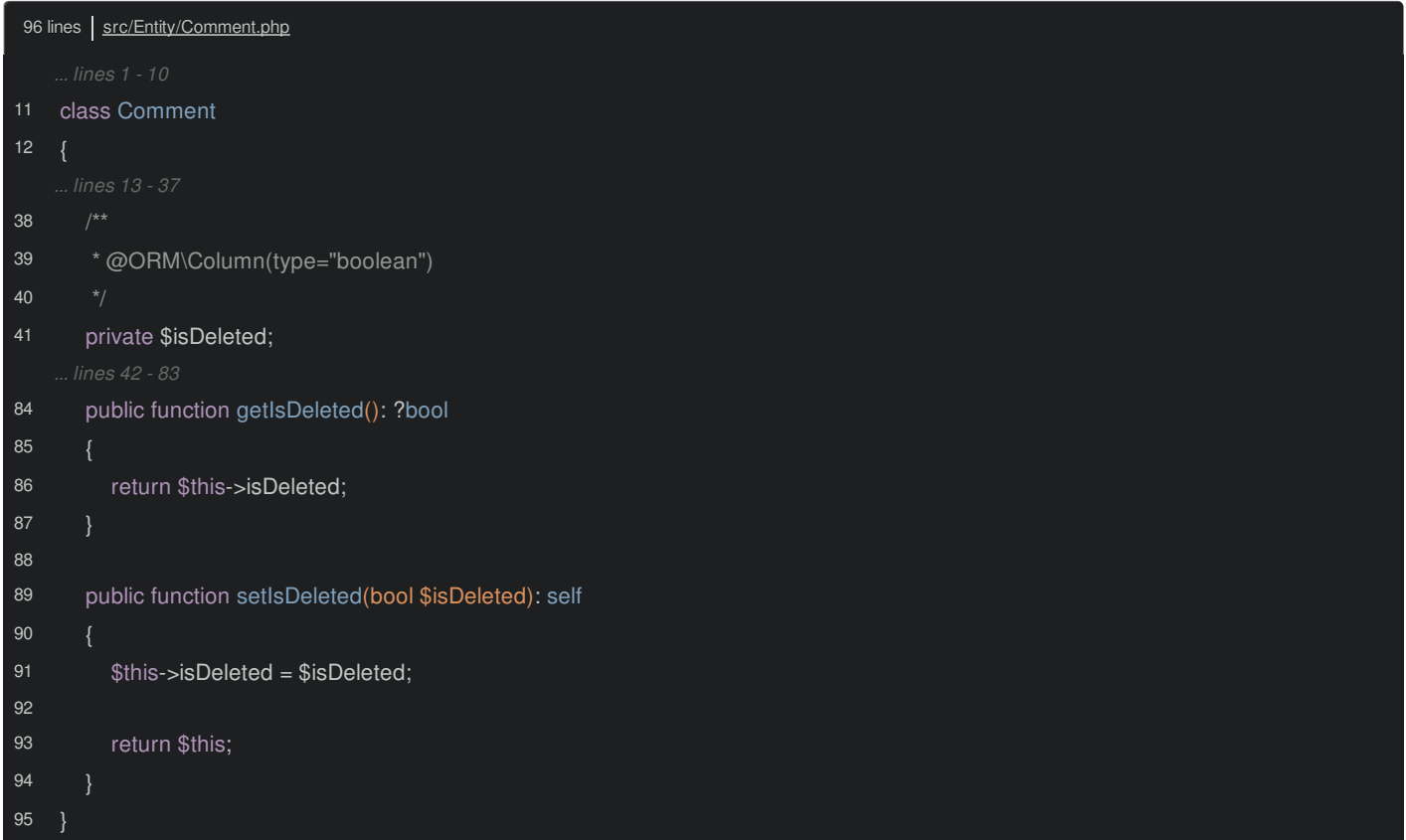
[Adding Comment.isDeleted Field](#)

Here's the setup: go to your terminal and run:




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

We're going to add a new field to the Comment entity called isDeleted. This will be a boolean type and set it to not nullable in the database:



```
96 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 37
38 /**
39  * @ORM\Column(type="boolean")
40  */
41 private $isDeleted;
... lines 42 - 83
84 public function getIsDeleted(): ?bool
85 {
86     return $this->isDeleted;
87 }
88
89 public function setIsDeleted(bool $isDeleted): self
90 {
91     $this->isDeleted = $isDeleted;
92
93     return $this;
94 }
95 }
```

When that finishes, make the migration:



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

And, you know the drill: open that migration to make sure it doesn't contain any surprises:

```

29 lines | src/Migrations/Version20180430194518.php
... lines 1 - 2
3  namespace DoctrineMigrations;
4
5  use Doctrine\DBAL\Migrations\AbstractMigration;
6  use Doctrine\DBAL\Schema\Schema;
7
8  /**
9   * Auto-generated Migration: Please modify to your needs!
10  */
11  class Version20180430194518 extends AbstractMigration
12  {
13      public function up(Schema $schema)
14      {
15          // this up() migration is auto-generated, please modify it to your needs
16          $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
17
18          $this->addSql('ALTER TABLE comment ADD is_deleted TINYINT(1) NOT NULL');
19      }
20
21      public function down(Schema $schema)
22      {
23          // this down() migration is auto-generated, please modify it to your needs
24          $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
25
26          $this->addSql('ALTER TABLE comment DROP is_deleted');
27      }
28  }

```

Oh, this is cool: when you use a boolean type in Doctrine, the value on your entity will be true or false, but in the database, it stores as a tiny int with a zero or one.

This looks good, so move back and.... migrate!

```

$ php bin/console doctrine:migrations:migrate

```

Updating the Fixtures

We're not going to create an admin interface to delete comments, at least, not yet. Instead, let's update our fixtures so that it loads some "deleted" comments. But first, inside Comment, find the new field and... default isDeleted to false:

```

96 lines | src/Entity/Comment.php
... lines 1 - 10
11  class Comment
12  {
13      ... lines 13 - 37
38      /**
39       * @ORM\Column(type="boolean")
40       */
41      private $isDeleted = false;
42      ... lines 42 - 94
95  }

```

Any new comments will *not* be deleted.

Next, in `CommentFixture`, let's say `$comment->setIsDeleted()` with `$this->faker->boolean(20)`:

```
33 lines | src/DataFixtures/CommentFixture.php
... lines 1 - 9
10 class CommentFixture extends BaseFixture implements DependentFixtureInterface
11 {
12     protected function loadData(ObjectManager $manager)
13     {
14         $this->createMany(Comment::class, 100, function(Comment $comment) {
... lines 15 - 20
21             $comment->setIsDeleted($this->faker->boolean(20));
... line 22
23         });
... lines 24 - 25
26     }
... lines 27 - 31
32 }
```

So, out of the 100 comments, approximately 20 of them will be marked as deleted.

Then, to make this a *little* bit obvious on the front-end, for now, open `show.html.twig` and, right after the date, add an if statement: if `comment.isDeleted`, then, add a close, "X", icon and say "deleted":

```
90 lines | templates/article/show.html.twig
... lines 1 - 6
7 <div class="container">
8     <div class="row">
9         <div class="col-sm-12">
10             <div class="show-article-container p-3 mt-4">
... lines 11 - 39
40         <div class="row">
41             <div class="col-sm-12">
... lines 42 - 57
58             {% for comment in article.comments %}
59             <div class="row">
60                 <div class="col-sm-12">
... line 61
62                 <div class="comment-container d-inline-block pl-3 align-top">
... line 63
64                     <small>about {{ comment.createdAt|ago }}</small>
65                     {% if comment.isDeleted %}
66                     <span class="fa fa-close"></span> deleted
67                     {% endif %}
... lines 68 - 70
71                 </div>
72             </div>
73         </div>
74         {% endfor %}
75
76     </div>
77 </div>
78 </div>
79 </div>
80 </div>
81 </div>
... lines 82 - 90
```


Find your terminal and freshen up your fixtures:

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

When that finishes, move back, refresh... then scroll down. Let's see... yea! Here's one: this article has one deleted comment.

Hiding Deleted Comments

We printed this "deleted" note *mostly* for our own benefit while developing. Because, what we *really* want to do is, of course, *not* show the deleted comments at all!

But... hmm. The problem is that, to *get* the comments, we're calling `article.comments`:

```
90 lines | templates/article/show.html.twig
... lines 1 - 6
7   <div class="container">
8     <div class="row">
9       <div class="col-sm-12">
10        <div class="show-article-container p-3 mt-4">
... lines 11 - 39
40        <div class="row">
41          <div class="col-sm-12">
... lines 42 - 57
58            {% for comment in article.comments %}
... lines 59 - 73
74            {% endfor %}
75
76          </div>
77        </div>
78      </div>
79    </div>
80  </div>
81 </div>
... lines 82 - 90
```

Which means we're calling `Article::getComments()`:

```
203 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 171
172 /**
173  * @return Collection|Comment[]
174  */
175 public function getComments(): Collection
176 {
177     return $this->comments;
178 }
... lines 179 - 201
202 }
```

This is our super-handly, super-lazy shortcut method that returns *all* of the comments. Dang! Now we need a way to return only the *non-deleted* comments. Is that possible?

Yes! One option is super simple. Instead of using `article.comments`, we could go into `ArticleController`, find the `show()` action,

create a custom query for the Comment objects we need, pass those into the template, then use that new variable. When the shortcut methods don't work, always remember that you don't *need* to use them.

But, there is *another* option, it's a bit lazier, *and* a bit more fun.

Creating Article::getNonDeletedComments()

Open Article and find the getComments() method. Copy it, paste, and rename to getNonDeletedComments(). But, for now, just return *all* of the comments:

```
211 lines | src/Entity/Article.php
... lines 1 - 13
14 class Article
15 {
... lines 16 - 179
180 /**
181  * @return Collection|Comment[]
182  */
183 public function getNonDeletedComments(): Collection
184 {
185     return $this->comments;
186 }
... lines 187 - 209
210 }
```

Then, in the show template, use this new field: in the loop, article.nonDeletedComments:

```
90 lines | templates/article/show.html.twig
... lines 1 - 4
5 {% block body %}
6
7 <div class="container">
8     <div class="row">
9         <div class="col-sm-12">
10             <div class="show-article-container p-3 mt-4">
... lines 11 - 39
40         <div class="row">
41             <div class="col-sm-12">
... lines 42 - 57
58                 {% for comment in article.nonDeletedComments %}
... lines 59 - 73
74                 {% endfor %}
75
76             </div>
77         </div>
78     </div>
79 </div>
80 </div>
81 </div>
82
83 {% endblock %}
... lines 84 - 90
```

And, further up, when we count them, *a/so* use article.nonDeletedComments:

90 lines | [templates/article/show.html.twig](#)

... lines 1 - 4

5 {% block body %}

6

7 <div class="container">

8 <div class="row">

9 <div class="col-sm-12">

10 <div class="show-article-container p-3 mt-4">

... lines 11 - 39

40 <div class="row">

41 <div class="col-sm-12">

42 <h3><i class="pr-3 fa fa-comment"></i>{{ article.nonDeletedComments|length }} Comments</h3>

... lines 43 - 75

76 </div>

77 </div>

78 </div>

79 </div>

80 </div>

81 </div>

82

83 {% endblock %}

... lines 84 - 90

Let's refresh to make sure this works so far. No errors, but, of course, we are *still* showing *all* of the comments.

[Filtering Deleted Comments in Article::getNonDeletedComments\(\)](#)

Back in Article, how can we change this method to filter out the deleted comments? Well, there is a lazy way, which is sometimes good enough. And an awesome way! The lazy way would be to, for example, create a new \$comments array, loop over \$this->getComments(), check if the comment is deleted, and add it to the array if it is not. Then, at the bottom, return a new ArrayCollection of those comments:

```
$comments = [];
```

```
foreach ($this->getComments() as $comment) {
```

```
    if (!$comment->getIsDeleted()) {
```

```
        $comments[] = $comment;
```

```
    }
```

```
}
```

```
return new ArrayCollection($comments);
```

Simple! But... this solution has a drawback... performance! Let's talk about that next, *and*, the awesome fix.

Chapter 10: Collection Magic with Criteria

Of course, if we wanted to remove any of the deleted comments from this collection, we *could* loop over *all* of the comments, check if each is deleted, and return an array of *only* the ones left. Heck, the collection object even has a `filter()` method to make this easier!

But... there's a problem. If we *did* this, Doctrine would query for *all* of the comments, even though we don't *need* all of the comments. If your collection is pretty small, no big deal: querying for a few extra comments is probably fine. But if you have a large collection, like 200 comments, and you want to return a small sub-set, like only 10 of them, that would be super, *super* wasteful!

Hello Criteria

To solve this, Doctrine has a *super* powerful and amazing feature... and yet, somehow, almost nobody knows about it! Time to change that! Once you're an expert on this feature, it'll be your job to tell the world!

The system is called "Criteria". Instead of looping over all the data, add `$criteria = Criteria - the one from Doctrine - Criteria::create()`:

```
217 lines | src/Entity/Article.php
... lines 1 - 6
7   use Doctrine\Common\Collections\Criteria;
... lines 8 - 14
15  class Article
16  {
... lines 17 - 183
184  public function getNonDeletedComments(): Collection
185  {
186      $criteria = Criteria::create()
... lines 187 - 191
192  }
... lines 193 - 215
216 }
```

Then, you can chain off of this. The Criteria object is similar to the QueryBuilder we're used to, but with a slightly different, well, slightly more *confusing* syntax. Add `andWhere()`, but instead of a string, use `Criteria::expr()`. Then, there are a bunch of methods to help create the where clause, like `eq()` for equals, `gt()` for greater than, `gte()` for greater than or equal, and so on. It's a little object-oriented builder for the WHERE expression.

In this case, we need `eq()` so we can say that `isDeleted` equals false:

```

217 lines | src/Entity/Article.php
... lines 1 - 6
7   use Doctrine\Common\Collections\Criteria;
... lines 8 - 14
15  class Article
16  {
... lines 17 - 183
184  public function getNonDeletedComments(): Collection
185  {
186      $criteria = Criteria::create()
187          ->andWhere(Criteria::expr()->eq('isDeleted', false))
... lines 188 - 191
192  }
... lines 193 - 215
216 }

```

Then, add orderBy, with createdAt => 'DESC' to keep the sorting we want:

```

217 lines | src/Entity/Article.php
... lines 1 - 6
7   use Doctrine\Common\Collections\Criteria;
... lines 8 - 14
15  class Article
16  {
... lines 17 - 183
184  public function getNonDeletedComments(): Collection
185  {
186      $criteria = Criteria::create()
187          ->andWhere(Criteria::expr()->eq('isDeleted', false))
188          ->orderBy(['createdAt' => 'DESC'])
189      ;
... lines 190 - 191
192  }
... lines 193 - 215
216 }

```

Creating the Criteria object doesn't actually *do* anything yet - it's like creating a query builder. But *now* we can say return `$this->comments->matching()` and pass `$criteria`:

```

217 lines | src/Entity/Article.php
... lines 1 - 6
7   use Doctrine\Common\Collections\Criteria;
... lines 8 - 14
15  class Article
16  {
... lines 17 - 183
184  public function getNonDeletedComments(): Collection
185  {
186      $criteria = Criteria::create()
187          ->andWhere(Criteria::expr()->eq('isDeleted', false))
188          ->orderBy(['createdAt' => 'DESC'])
189      ;
190
191      return $this->comments->matching($criteria);
192  }
... lines 193 - 215
216 }

```

Because, remember, even though we *think* of the `$comments` property as an array, it's not! This Collection return type is an interface from Doctrine, and our property will always be some object that implements that. That's a long way of saying that, while the `$comments` property will look and feel like an array, it is *actually* an object that has some extra helper methods on it.

[The Super-Intelligent Criteria Queries](#)

Anyways, ready to try this? Move over and refresh. Check it out: the 8 comments went down to 7! And the deleted comment is *gone*. But you haven't seen the *best* part yet! Click to open the profiler for Doctrine. Check out the last query: it's *perfect*. It *no* longer queries for *all* of the comments for this article. Nope, instead, Doctrine executed a super-smart query that finds all comments where the article matches this article *and* where `isDeleted` is false, or zero. It *even* did the same for the count query!

Doctrine, that's crazy cool! So, by using Criteria, we get *super* efficient filtering. Of course, it's not *always* necessary. You *could* just loop over all of the comments and filter manually. If you are removing only a *small* percentage of the results, the performance difference is minor. The Criteria system *is* better than manually filtering, but, remember! Do *not* prematurely optimize. Get your app to production, then check for issues. But if you have a *big* collection and need to return only a small number of results, you should use Criteria immediately.

[Organizing the Criteria into the Repository](#)

One thing I *don't* like about the Criteria system is that I do *not* like having query logic inside my entity. And this is important! To keep my app sane, I want to have 100% of my query logic *inside* my repository. No worries: we can move it there!

In `ArticleRepository`, create a public static function called `createNonDeletedCriteria()` that will return a Criteria object:

```

67 lines | src/Repository/ArticleRepository.php
... lines 1 - 6
7   use Doctrine\Common\Collections\Criteria;
... lines 8 - 16
17  class ArticleRepository extends ServiceEntityRepository
18  {
... lines 19 - 35
36  public static function createNonDeletedCriteria(): Criteria
37  {
... lines 38 - 41
42  }
... lines 43 - 65
66 }

```

In `Article`, copy the Criteria code, paste it here, and return:

```

67 lines | src/Repository/ArticleRepository.php
... lines 1 - 6
7  use Doctrine\Common\Collections\Criteria;
... lines 8 - 16
17 class ArticleRepository extends ServiceEntityRepository
18 {
... lines 19 - 35
36     public static function createNonDeletedCriteria(): Criteria
37     {
38         return Criteria::create()
39             ->andWhere(Criteria::expr()->eq('isDeleted', false))
40             ->orderBy(['createdAt' => 'DESC'])
41     };
42 }
... lines 43 - 65
66 }

```

These are the *only* static methods that you should ever have in your repository. It *needs* to be static simply so that we can use it from inside Article. That's because entity classes don't have access to services.

Use it with `$criteria = ArticleRepository::createNonDeletedCriteria();`

```

217 lines | src/Entity/Article.php
... lines 1 - 4
5  use App\Repository\ArticleRepository;
... lines 6 - 15
16 class Article
17 {
... lines 18 - 186
187     public function getNonDeletedComments(): Collection
188     {
189         $criteria = ArticleRepository::createNonDeletedCriteria();
190
191         return $this->comments->matching($criteria);
192     }
... lines 193 - 215
216 }

```

Side note: we *could* have also put this method into the CommentRepository. When you start working with related entities, sometimes, it's not clear exactly *which* repository class should hold some logic. No worries: do your best and don't over-think it. You can always move code around later.

Ok, go back to your browser, close the profiler and, refresh. Awesome: it still works great!

[Using the Criteria in a QueryBuilder](#)

Oh, and bonus! in ArticleRepository, what if in the future, we need to create a QueryBuilder and want to re-use the logic from the Criteria? Is that possible? Totally! Just use `->addCriteria()` then, in this case, `self::createNonDeletedCriteria();`

```

class ArticleRepository extends ServiceEntityRepository
{
    public function findAllPublishedOrderedByNewest()
    {
        $this->createQueryBuilder('a')
            ->addCriteria(self::createNonDeletedCriteria());

        return $this->addIsPublishedQueryBuilder()
            ->orderBy('a.publishedAt', 'DESC')
            ->getQuery()
            ->getResult()
        ;
    }
}

```

These Criteria *are* reusable.

Updating the Homepage

To finish this feature, go back to the homepage. These comment numbers are still including deleted comments. No problem! Open homepage.html.twig, find where we're printing that number, and use `article.nonDeletedComments`:

```

59 lines | templates/article/homepage.html.twig
... lines 1 - 2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6
7              <!-- Article List -->
8
9              <div class="col-sm-12 col-md-8">
... lines 10 - 18
19          <!-- Supporting Articles -->
20
21          {% for article in articles %}
22              <div class="article-container my-1">
23                  <a href="{{ path('article_show', {slug: article.slug}) }}">
... line 24
25                      <div class="article-title d-inline-block pl-3 align-middle">
... line 26
27                          <small>({{ article.nonDeletedComments|length }} comments)</small>
... lines 28 - 30
31                      </div>
32                  </a>
33              </div>
34          {% endfor %}
35      </div>
... lines 36 - 55
56      </div>
57  </div>
58  {% endblock %}

```

Ok, go back. We have 10, 13 & 7. Refresh! Nice! Now it's 5, 9 and 5.

Next, let's take a quick detour and leverage some Twig techniques to reduce duplication in our templates.

Chapter 11: Twig Block Tricks

This tutorial is *all* about Doctrine relations... plus a few other Easter eggs along the way. And one *big* topic we need to talk about is how to create queries that *join* across these relationships. To do that properly, we're going to build a comment admin section. Well, for now, we're just going to *start* building it.

Since this will be a new section on the site, let's create a new controller class! And because I'm feeling *especially* lazy, find your terminal and run:

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

Call it CommentAdminController. This creates a new class *and* one bonus template file. Go check it out!

```
20 lines | src/Controller/CommentAdminController.php
... lines 1 - 2
3  namespace App\Controller;
4
5  use Symfony\Component\Routing\Annotation\Route;
6  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
7
8  class CommentAdminController extends Controller
9  {
10     /**
11      * @Route("/comment/admin", name="comment_admin")
12      */
13     public function index()
14     {
15         return $this->render('comment_admin/index.html.twig', [
16             'controller_name' => 'CommentAdminController',
17         ]);
18     }
19 }
```

Ok, nice start! Hmm, but let's change that URL to /admin/comment:

```
20 lines | src/Controller/CommentAdminController.php
... lines 1 - 7
8  class CommentAdminController extends Controller
9  {
10     /**
11      * @Route("/admin/comment", name="comment_admin")
12      */
13     public function index()
14     {
15         ... lines 15 - 17
18     }
19 }
```

Let's see what we have so far. Open a new browser tab and go to <http://localhost:8000/admin/comment>. Awesome! The template even tells us where the source code lives!

[Building the Comment Admin Template](#)

Let's open that template and get to work!

```
21 lines | templates/comment_admin/index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Hello {{ controller_name }}!{% endblock %}
4
5  {% block body %}
6    <style>
7      .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
8      .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
9    </style>
10
11    <div class="example-wrapper">
12      <h1>Hello {{ controller_name }}! </h1>
13
14      This friendly message is coming from:
15      <ul>
16        <li>Your controller at <code><a href="{{ 'src/Controller/CommentAdminController.php'|file_link(0) }}">src/Controller/CommentAdm
17        <li>Your template at <code><a href="{{ 'templates/comment_admin/index.html.twig'|file_link(0) }}">templates/comment_admin/inc
18      </ul>
19    </div>
20  {% endblock %}
```

This already overrides the title block, which is cool! Change it to say "Manage Comments". Then, delete all the body code:

```
21 lines | templates/comment_admin/index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Manage Comments{% endblock %}
4
5  {% block body %}
6    ... lines 6 - 19
20  {% endblock %}
```

To make the page look nice, open show.html.twig: we need to steal some markup from this. Copy the first 6 divs. Back in index.html.twig, paste, close each of those 6 divs and... back in the middle, add Manage Comments:

21 lines | [templates/comment_admin/index.html.twig](#)

... lines 1 - 4

```
5 {% block body %}
6
7 <div class="container">
8   <div class="row">
9     <div class="col-sm-12">
10      <div class="show-article-container p-3 mt-4">
11        <div class="row">
12          <div class="col-sm-12">
13            <h1>Manage Comments</h1>
14          </div>
15        </div>
16      </div>
17    </div>
18  </div>
19 </div>
20 {% endblock %}
```

Try that again in your browser: refresh. Ok! Those 6 divs give us this white box that you also see on the article show page.

[Creating a Sub-Layout](#)

Hmm. If you think about it, it's probably going to be *really* common for us to want a page where we have some nice margin and a white box. The homepage doesn't need this, but, I bet a lot of internal pages *will*.

So, needing to duplicate these six divs on *each* page is pretty *lame*. Fortunately, Twig comes to the rescue! Go Twig! We can isolate this markup into a *new* base layout.

In the templates/ directory, create a new file: content_base.html.twig. What's *cool* is that, we can extend the *normal* base.html.twig and then just add the *extra* markup we need:

13 lines | [templates/content_base.html.twig](#)

```
1 {% extends 'base.html.twig' %}
... lines 2 - 13
```

To do that, override block body just like we would in a normal template. Then, steal the first four divs: these are the divs that *really* give us the structure. Paste them here, and type a ton of closing div tags:

13 lines | [templates/content_base.html.twig](#)

```
1 {% extends 'base.html.twig' %}
2
3 {% block body %}
4   <div class="container">
5     <div class="row">
6       <div class="col-sm-12">
7         <div class="show-article-container p-3 mt-4">
... line 8
9       </div>
10    </div>
11  </div>
12 </div>
13 {% endblock %}
```

Next, and here's the key, in the middle, which is where we want the *content* to go, create a *new* block called content_body and {% endblock %}:

```

13 lines | templates/content_base.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6              <div class="col-sm-12">
7                  <div class="show-article-container p-3 mt-4">
8                      {% block content_body %}{% endblock %}
9                  </div>
10             </div>
11         </div>
12     </div>
13 {% endblock %}

```

I just invented that name.

And, that's it! Let's go use it! In index.html.twig, change the extends to content_base.html.twig:

```

12 lines | templates/comment_admin/index.html.twig
1  {% extends 'content_base.html.twig' %}
... lines 2 - 12

```

Now, we do *not* want to override the block body. Nope, we want to override content_body. Thanks to this, we should get the normal base layout *plus* the extra markup from content_base.html.twig.

Remove the 4 divs, their closing tags, then clean things up a bit:

```

12 lines | templates/comment_admin/index.html.twig
1  {% extends 'content_base.html.twig' %}
2
3  {% block title %}Manage Comments{% endblock %}
4
5  {% block content_body %}
6      <div class="row">
7          <div class="col-sm-12">
8              <h1>Manage Comments</h1>
9          </div>
10      </div>
11  {% endblock %}

```

Ok, try it! Go back to the admin tab and refresh. Yay! A nice layout with *no* work. Repeat this in show.html.twig: extend content_base.html.twig, change the block to content_body, and remove the 4 divs on top, the 4 closing tags on the bottom and... un-indent a few times so this looks decent:

```

81 lines | templates/article/show.html.twig
1  {% extends 'content_base.html.twig' %}
2
3  {% block title %}Read: {{ article.title }}{% endblock %}
4
5  {% block content_body %}
6      <div class="row">
7          <div class="col-sm-12">
... lines 8 - 20
21      </div>
22  </div>
... lines 23 - 34
35      <div class="row">
36          <div class="col-sm-12">
... lines 37 - 70
71      </div>
72  </div>
73
74  {% endblock %}
... lines 75 - 81

```

And unless we forgot something... nope! It still looks perfect! We now have a *super* easy way to create new pages.

[Adding a Custom Class to One Template](#)

Except... there's *one* small design change I want to make... but I want to make it to the comment admin section *only*. Our project already has a public/css/styles.css file. On your browser, "Inspect Element" on the white box. *One* of the CSS classes in styles.css is show-article-container-border-green.

If you add this, you get a nice green border on top! And according to our hard-working and very particular design team, they want this on the "Manage Comments" page, but they do *not* want it on the article page. Apparently some of our alien readers associate the color green with fictional, untrustworthy content. Well, we can't have that!

But, dang! This ruins everything! The class needs to live on the show-article-container div... but *that* lives inside content_base.html.twig. How can we change this for *only* one of the children templates?

The answer is... drumroll... blocks! Blocks are *almost* always the answer when you need to do cool things with Twig inheritance.

In content_base.html.twig, surround all of the classes with a new block: call it content_class. After the classes, use endblock:

```

13 lines | templates/content_base.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6              <div class="col-sm-12">
7                  <div class="{% block content_class %}show-article-container p-3 mt-4{% endblock %}">
... line 8
9              </div>
10          </div>
11      </div>
12  </div>
13  {% endblock %}

```

This defines a new block that has *default content*. If nobody overrides the block, it will have all of these classes. But, in the comments template, we *can* override this: {% block content_class %}. But, we don't *really* want to fully replace it: we want to *add* to the block's content. No problem: use {{ parent() }} to print the existing classes, then

show-article-container-border-green with {% endblock %}:

14 lines | [templates/comment_admin/index.html.twig](#)

```
1  {% extends 'content_base.html.twig' %}
```

... lines 2 - 4

```
5  {% block content_class %}{{ parent() }} show-article-container-border-green{% endblock %}
```

... lines 6 - 14

I love it! To make sure I'm not lying: find your browser and refresh the manage comments page. Looks great! But the article show page... nope! No green border: our alien readers will continue to trust us.

Next, let's finish our comment admin page by listing them in a table. And, we'll install a cool library called "Twig Extension".

Chapter 12: The Twig Extensions Library

Let's bring this section to life by listing *all* of the comments in the system. Ah man, with all our tools, this is going to be really easy! First, to query for the comments, add the `CommentRepository $repository` argument:

```
23 lines | src/Controller/CommentAdminController.php
... lines 1 - 4
5  use App\Repository\CommentRepository;
... lines 6 - 8
9  class CommentAdminController extends Controller
10 {
11     /**
12      * @Route("/admin/comment", name="comment_admin")
13      */
14     public function index(CommentRepository $repository)
15     {
... lines 16 - 20
21     }
22 }
```

Then, `$comments = $repository->`, and we *could* use `findAll()`, but I'll use `findBy()` passing this an empty array, then `'createdAt' => 'DESC'` so that we get the newest comments on top:

```
23 lines | src/Controller/CommentAdminController.php
... lines 1 - 8
9  class CommentAdminController extends Controller
10 {
11     /**
12      * @Route("/admin/comment", name="comment_admin")
13      */
14     public function index(CommentRepository $repository)
15     {
16         $comments = $repository->findBy([], ['createdAt' => 'DESC']);
... lines 17 - 20
21     }
22 }
```

Clear out the `render()` variables: we only need to pass one: `comments` set to `$comments`:

23 lines | [src/Controller/CommentAdminController.php](#)

... lines 1 - 8

```
9 class CommentAdminController extends Controller
10 {
11     /**
12      * @Route("/admin/comment", name="comment_admin")
13      */
14     public function index(CommentRepository $repository)
15     {
16         $comments = $repository->findBy([], ['createdAt' => 'DESC']);
17
18         return $this->render('comment_admin/index.html.twig', [
19             'comments' => $comments,
20         ]);
21     }
22 }
```

Perfect! Next, to the template! Below the h1, I'll paste the beginning of a table that has some Bootstrap classes and headers for the article name, author, the comment itself and when it was created:

45 lines | [templates/comment_admin/index.html.twig](#)

... lines 1 - 6

```
7 {% block content_body %}
8     <div class="row">
9         <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <table class="table table-striped">
13                 <thead>
14                     <tr>
15                         <th>Article</th>
16                         <th>Author</th>
17                         <th>Comment</th>
18                         <th>Created</th>
19                     </tr>
20                 </thead>
21                 <tbody>
22
23 ... lines 22 - 39
40             </tbody>
41         </table>
42     </div>
43 </div>
44 {% endblock %}
```

No problem! In the tbody, let's loop: for comment in comments, and {% endfor %}:

45 lines | [templates/comment_admin/index.html.twig](#)

... lines 1 - 6

```
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <table class="table table-striped">
13                 <thead>
14                     <tr>
15                         <th>Article</th>
16                         <th>Author</th>
17                         <th>Comment</th>
18                         <th>Created</th>
19                     </tr>
20                 </thead>
21                 <tbody>
22                     {% for comment in comments %}
... lines 23 - 38
39                     {% endfor %}
40                 </tbody>
41             </table>
42         </div>
43     </div>
44 {% endblock %}
```

Add the `<tr>`, then let's print some data! In the first td, we need the article name. But, to make it *more* awesome, let's make this a *link* to the article. Add the `a` tag with `href=""`, but keep that blank for a moment. Inside, hmm, we have a `Comment` object, but we want to print the article's title. No problem! We can use our relationship: `comment.article` - that gets us to the `Article` object - then `.title`.

For the href, use the `path()` function from Twig. Here, we need the *name* of the route that we want to link to. Open `ArticleController`. Ah! There it is: `name="article_show"`:

65 lines | [src/Controller/ArticleController.php](#)

... lines 1 - 13

```
14 class ArticleController extends AbstractController
15 {
... lines 16 - 37
38     /**
39      * @Route("/news/{slug}", name="article_show")
40      */
41     public function show(Article $article, SlackClient $slack)
42     {
... lines 43 - 49
50     }
... lines 51 - 63
64 }
```

Close that and, back in the template, use `article_show`. This route needs a slug parameter so add that, set to `comment.article.slug`:

45 lines | [templates/comment_admin/index.html.twig](#)

... lines 1 - 6

```
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <table class="table table-striped">
13                 <thead>
14                     <tr>
15                         <th>Article</th>
16                         <th>Author</th>
17                         <th>Comment</th>
18                         <th>Created</th>
19                     </tr>
20                 </thead>
21                 <tbody>
22                     {% for comment in comments %}
23                         <tr>
24                             <td>
25                                 <a href="{{ path('article_show', {'slug': comment.article.slug}) }}">
26                                     {{ comment.article.title }}
27                                 </a>
28                             </td>
... lines 29 - 37
38                         </tr>
39                     {% endfor %}
40                 </tbody>
41             </table>
42         </div>
43     </div>
44 {% endblock %}
```

Dang, those relationships are handy!

Let's keep going! Add another td and print comment.authorName:

... lines 1 - 6

```
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <table class="table table-striped">
... lines 13 - 20
21                 <tbody>
22                     {% for comment in comments %}
23                         <tr>
24                             <td>
25                                 <a href="{{ path('article_show', {'slug': comment.article.slug}) }}">
26                                     {{ comment.article.title }}
27                                 </a>
28                             </td>
29                             <td>
30                                 {{ comment.authorName }}
31                             </td>
... lines 32 - 37
38                         </tr>
39                     {% endfor %}
40                 </tbody>
41             </table>
42         </div>
43     </div>
44 {% endblock %}
```

Give the next td a style="width: 20%" so it doesn't get too big. Then, print comment.content:

45 lines | [templates/comment_admin/index.html.twig](#)

... lines 1 - 6

```
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <table class="table table-striped">
... lines 13 - 20
21                 <tbody>
22                     {% for comment in comments %}
23                         <tr>
24                             <td>
25                                 <a href="{{ path('article_show', {'slug': comment.article.slug}) }}">
26                                     {{ comment.article.title }}
27                                 </a>
28                             </td>
29                             <td>
30                                 {{ comment.authorName }}
31                             </td>
32                             <td style="width: 20%;">
33                                 {{ comment.content }}
34                             </td>
... lines 35 - 37
38                         </tr>
39                     {% endfor %}
40                 </tbody>
41             </table>
42         </div>
43     </div>
44 {% endblock %}
```

Finally, add a td with comment.createdAt|ago:

45 lines | [templates/comment_admin/index.html.twig](#)

... lines 1 - 6

```
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <table class="table table-striped">
... lines 13 - 20
21                 <tbody>
22                     {% for comment in comments %}
23                         <tr>
24                             <td>
25                                 <a href="{{ path('article_show', {'slug': comment.article.slug}) }}">
26                                     {{ comment.article.title }}
27                                 </a>
28                             </td>
29                             <td>
30                                 {{ comment.authorName }}
31                             </td>
32                             <td style="width: 20%;">
33                                 {{ comment.content }}
34                             </td>
35                             <td>
36                                 {{ comment.createdAt|ago }}
37                             </td>
38                         </tr>
39                     {% endfor %}
40                 </tbody>
41             </table>
42         </div>
43     </div>
44 {% endblock %}
```

Cool! Let's see if we made any mistakes. Find your browser, refresh and... boom! A big, beautiful list of *all* of the comments on the site. Oh, but eventually on production, this will be a *huge* number of results. Let's put it on our todo list to add pagination.

[The N+1 Query Problem](#)

Hmm, it works, but check out the web debug toolbar: 11 queries. This is that same annoying N+1 problem that we talked about earlier. The first query is the one we expect: SELECT all of the comments from the system. But, as we loop over each comment and fetch data from its related article, an *extra* query is made to get that data.

Like, this query fetches the data for article id 181, this does the same for article id 186, and so on. We get 11 queries because we have 1 query for the comments and 10 more queries for the 10 related articles.

Hence, the N+1 problem: 10 related object plus the 1 original query. So, the question is, how can we solve this, right? Well, actually, a better question is: *should* we solve this? Here's the point: *you* need to be aware of the fact that Doctrine's nice relationship lazy-loading magic makes it easy to accidentally make *many* queries on a page. But, it is *not* something that you always need to solve. Why? Well, a lot of times, having 10 extra queries - especially on an admin page - is no big deal! On the other hand, maybe 100 extra queries on your homepage, well, that probably *is* a problem. As I *always* like to say, deploy first, then see where you have problems. Using a tool like Blackfire.io makes it *very* easy to find *real* issues.

Anyways, we *will* learn how to fix this in a few minutes. But, ignore it for now.

[Installing Twig Extensions](#)


Because... we have a minor, but more immediate problem: some comments will probably be pretty long. So, printing the *entire* comment will become a problem. What I *really* want to do is show some sort of preview, maybe the first 30 characters of a comment.

Hmm, can Twig do that? Go to twig.symfony.com and click on the Documentation. Huh, there is actually *not* a filter or function that can do this! We could easily add one, but instead, search for "Twig extensions" and click on the [documentation for some Twig extensions library](#).

We know that if we need to create a custom Twig function or filter, we create a class called a Twig extension. We did it in an earlier tutorial. But *this* is something different: this is an open source library *called* "Twig Extensions". It's simply a collection of pre-made, useful, Twig extension classes. Nice!

For example, one Twig extension - called Text - has a filter called *truncate*! Bingo! That's *exactly* what we need. Click on the "Text" extension's documentation, then click the link to install it. Perfect! Copy that composer require line.

Then, find your terminal and, paste!




```
$ composer require twig/extensions
```

[Activating the Twig Extension](#)

While we're waiting for this to install, I want to point out something important: we're installing a PHP *library*, not a Symfony *bundle*. What's the difference? Well, a PHP *library* simply contains *classes*, but does *not* automatically integrate into your app. Most importantly, this means that while this library *will* give us some Twig extension PHP classes, it will *not* register those as services or make our Twig service aware of them. *We* will need to configure things by hand.


Go back to the terminal and, oh! Let's play a thrilling game of Pong while we wait. Go left side, go left side, go! Boooo!

Anyways, ooh! This installed a *recipe*! I committed my changes before I started recording. So let's run:



```
$ git status
```

to see what changed. Beyond the normal Composer files and symfony.lock, the recipe created a *new* file: config/packages/twig_extensions.yaml. Ah, go check it out!



```
11 lines | config/packages/twig_extensions.yaml
1  services:
2    _defaults:
3      public: false
4      autowire: true
5      autoconfigure: true
6
7  #Twig\Extensions\ArrayExtension: ~
8  #Twig\Extensions\DateExtension: ~
9  #Twig\Extensions\IntlExtension: ~
10 #Twig\Extensions\TextExtension: ~
```

Nice! As we just talked about, the library *simply* gives us the extension classes, but it does *not* register them as services. So, to make life easier, the Flex recipe for the library *gives* us the exact configuration we need to finish the job! Here, we can activate the extensions by uncommenting the ones we need:

```

11 lines | config/packages/twig_extensions.yaml
1  services:
2    _defaults:
3      public: false
4      autowire: true
5      autoconfigure: true
... lines 6 - 9
10  Twig\Extensions\TextExtension: ~

```

Actually - because knowledge is power! - there are a few things going on. Thanks to the `Twig\Extensions\TextExtension: ~` part, that class becomes registered as a service. Remember: each class in the `src/` directory is *automatically* registered as a service. But because this class lives in `vendor/`, we need to register it by hand. Oh, and the `~` means null: it means we don't need to configure this service in any special way. For example, we don't need to configure any arguments.

Second, thanks to the `_defaults` section on top, specifically `autoconfigure`, Symfony *notifies* this is a Twig Extension by its interface, and automatically notifies the Twig service about it, without us needing to do anything.

All of this means that in `index.html.twig`, we can now immediately add `|truncate`:

```

45 lines | templates/comment_admin/index.html.twig
... lines 1 - 6
7  {% block content_body %}
8    <div class="row">
9      <div class="col-sm-12">
10       <h1>Manage Comments</h1>
11
12       <table class="table table-striped">
... lines 13 - 20
21         <tbody>
22           {% for comment in comments %}
23             <tr>
... lines 24 - 31
32               <td style="width: 20%;">
33                 {{ comment.content|truncate }}
34               </td>
... lines 35 - 37
38             </tr>
39           {% endfor %}
40         </tbody>
41       </table>
42     </div>
43   </div>
44 {% endblock %}

```

In fact, *before* we even try it, go back to your terminal and run:

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

This nice little tool shows us *all* of the functions, filters and other goodies that exist in Twig. And, ha! We now have a filter called `truncate`!

So, try it: find your browser, go back to the Manage Comments page, and refresh! It's perfect! Oh, and don't forget about the other cool stuff this Twig Extensions library has, like `Intl` for date or number formatting and, actually, `Date`, which coincidentally has a `time_diff` filter that works like our `ago` filter.

Next! Let's add a search form to the comment admin page.

Chapter 13: Request Object & Query OR Logic

Because astronauts *love* to debate news, our site will have a *lot* of comments on production. So, let's add a search box above this table so we can find things quickly.

Open the template and, on top, I'm going to paste in a simple HTML form:

```
61 lines | templates/comment_admin/index.html.twig
... lines 1 - 6
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <form>
13                 <div class="input-group mb-3">
14                     <input type="text"
15                         name="q"
16                         class="form-control"
17                         placeholder="Search..."
18                     >
19                     <div class="input-group-append">
20                         <button type="submit"
21                             class="btn btn-outline-secondary">
22                             <span class="fa fa-search"></span>
23                         </button>
24                     </div>
25                 </div>
26             </form>
... lines 27 - 57
58     </div>
59 </div>
60 {% endblock %}
```

We're not going to use Symfony's form system because, first, we haven't learned about it yet, and second, this is a *super* simple form: Symfony's form system wouldn't help us much anyways.

Ok! Check this out: the form has one input field whose name is q, and a button at the bottom. Notice that the form has *no* `action=`: this means that the form will submit *right* back to this same URL. It *also* has no `method=`, which means it will submit with a GET request instead of POST, which is *exactly* what you want for a search or filter form.

Let's see what it looks like: find your browser and refresh. Nice! Search for "ipsam" and hit enter. No, the search won't magically work yet. But, we *can* see the `?q=` at the end of the URL.

Fetching the Request Object

Back in the controller, hmm. The *first* question is: how can we read the `?q` query parameter? Actually, let me ask some bigger questions! How could we read POST data? Or, headers? Or the content of uploaded files?

Science! Well, actually, the request! *Any* time you need to read information about the request - POST data, headers, cookies, etc - you need Symfony's Request object. How can you get it? Well... you can probably guess: add another argument with a Request type-hint:


```

25 lines | src/Controller/CommentAdminController.php
... lines 1 - 5
6 use Symfony\Component\HttpFoundation\Request;
... lines 7 - 9
10 class CommentAdminController extends Controller
11 {
12     /**
13      * @Route("/admin/comment", name="comment_admin")
14      */
15     public function index(CommentRepository $repository, Request $request)
16     {
17         ... lines 17 - 22
23     }
24 }

```

Important: get the one from HttpFoundation - there are several, which, yea, is confusing:

```

25 lines | src/Controller/CommentAdminController.php
... lines 1 - 5
6 use Symfony\Component\HttpFoundation\Request;
... lines 7 - 25

```

So far, we know of *two* "magical" things you can do with controller arguments. First, if you type-hint a service class or interface, Symfony will give you that service. And second, if you type-hint an entity class, Symfony will query for that entity by using the wildcard in the route.

Well, you *might* think that the Request falls into the *first* magic category. I mean, that the Request is a service. Well, actually... the Request object is *not* a service. And, the reasons why are technical, and honestly, not very important. The ability to type-hint a controller argument with Request is the *third* "magic" trick you can do with controller arguments. So, it's (1) type-hint services, (2) type-hint entities or (3) type-hint the Request class. There *is* other magic that's possible, but these are the 3 main cases.

Oh, side-note: while the Request object is *not* in the service container, there *is* a service called RequestStack. You can fetch it like any service and call `getCurrentRequest()` to get the Request:

```

public function index(RequestStack $requestStack)
{
    $request = $requestStack->getCurrentRequest();
}

```

Anyways, the request gives us access to *everything* about the... um, request! Add `$q = $request->query->get('q');`:

```

25 lines | src/Controller/CommentAdminController.php
... lines 1 - 9
10 class CommentAdminController extends Controller
11 {
12     /**
13      * @Route("/admin/comment", name="comment_admin")
14      */
15     public function index(CommentRepository $repository, Request $request)
16     {
17         $q = $request->query->get('q');
18         ... lines 18 - 22
23     }
24 }

```

This is how you read *query* parameters, it's like a modern `$_GET`. There are other properties for almost everything else: `$request->headers` for headers, `$request->cookies`, `$request->files`, and a few more. Basically, any time you want to use

\$_GET, \$_POST, \$_SERVER or any of those global variables, use the Request instead.

[A Custom Query with OR Logic](#)

Now that we have the search term, we need to use that to make a custom query. So, sadly, we *cannot* use `findBy()` anymore: it's not smart enough to do queries that use the LIKE keyword. No worries: inside `CommentRepository`, add a public function called `findAllWithSearch()`. Give this a *nullable* string argument called `$term`:

```
81 lines | src/Repository/CommentRepository.php
... lines 1 - 15
16 class CommentRepository extends ServiceEntityRepository
17 {
... lines 18 - 34
35     public function findAllWithSearch(?string $term)
36     {
... lines 37 - 49
50     }
... lines 51 - 79
80 }
```

I'm making this nullable because, for convenience, I want to allow this method to be called with a null term, and we'll be smart enough to just return everything.

Above the method, add some PHP doc: this will @return an array of Comment objects:

```
81 lines | src/Repository/CommentRepository.php
... lines 1 - 15
16 class CommentRepository extends ServiceEntityRepository
17 {
... lines 18 - 30
31     /**
32      * @param string|null $term
33      * @return Comment[]
34      */
35     public function findAllWithSearch(?string $term)
36     {
... lines 37 - 49
50     }
... lines 51 - 79
80 }
```

Ok: we already know how to write custom queries: `$this->createQueryBuilder()` with an alias of `c`:

```

81 lines | src/Repository/CommentRepository.php
... lines 1 - 15
16 class CommentRepository extends ServiceEntityRepository
17 {
... lines 18 - 30
31 /**
32  * @param string|null $term
33  * @return Comment[]
34  */
35 public function findAllWithSearch(?string $term)
36 {
37     $qb = $this->createQueryBuilder('c');
... lines 38 - 49
50 }
... lines 51 - 79
80 }

```

Then, *if* a \$term is passed, we need a WHERE clause. But, here's the tricky part: I want to search for the term on a couple of fields: I want WHERE content LIKE \$term OR authorName LIKE \$term.

How can we do this? Hmm, the QueryBuilder apparently has an orWhere() method. Perfect, right? No! Surprise, I *never* use this method. Why? Imagine a complex query with various levels of AND clauses mixed with OR clauses and parenthesis. With a complex query like this, you would need to be *very* careful to use the parenthesis in just the right places. One mistake could lead to an OR causing *many* more results to be returned than you expect!

To *best* handle this in Doctrine, always use andWhere() and put all the OR logic right inside:

c.content LIKE :term OR c.authorName LIKE :term. On the next line, set term to, this looks a little odd, '%'.\$term.'%':

```

81 lines | src/Repository/CommentRepository.php
... lines 1 - 15
16 class CommentRepository extends ServiceEntityRepository
17 {
... lines 18 - 30
31 /**
32  * @param string|null $term
33  * @return Comment[]
34  */
35 public function findAllWithSearch(?string $term)
36 {
37     $qb = $this->createQueryBuilder('c');
38
39     if ($term) {
40         $qb->andWhere('c.content LIKE :term OR c.authorName LIKE :term')
41             ->setParameter('term', '%' . $term . '%')
42         ;
43     }
... lines 44 - 49
50 }
... lines 51 - 79
80 }

```

By putting this all inside andWhere() - instead of orWhere() - all of that logic will be surrounded by a parenthesis. Later, if we add another andWhere(), it'll logically group together properly.

Finally, in all cases, we want to return \$qb->orderBy('c.createdAt', 'DESC') and ->getQuery()->getResult():

81 lines | [src/Repository/CommentRepository.php](#)

... lines 1 - 15

```
16 class CommentRepository extends ServiceEntityRepository
17 {
    ... lines 18 - 30
31     /**
32      * @param string|null $term
33      * @return Comment[]
34      */
35     public function findAllWithSearch(?string $term)
36     {
37         $qb = $this->createQueryBuilder('c');
38
39         if ($term) {
40             $qb->andWhere('c.content LIKE :term OR c.authorName LIKE :term')
41                 ->setParameter('term', '%' . $term . '%')
42             ;
43         }
44
45         return $qb
46             ->orderBy('c.createdAt', 'DESC')
47             ->getQuery()
48             ->getResult()
49         ;
50     }
    ... lines 51 - 79
80 }
```

Remember, `getResult()` returns an array of results, and `getOneOrNullResult()` returns just *one* row.

Phew! That looks great! Go back to the controller. Use that method: `$comments = $repository->findAllWithSearch()` passing it `$q`:

25 lines | [src/Controller/CommentAdminController.php](#)

... lines 1 - 9

```
10 class CommentAdminController extends Controller
11 {
12     /**
13      * @Route("/admin/comment", name="comment_admin")
14      */
15     public function index(CommentRepository $repository, Request $request)
16     {
17         $q = $request->query->get('q');
18         $comments = $repository->findAllWithSearch($q);
    ... lines 19 - 22
23     }
24 }
```

Moment of truth! First, remove the `?q=` from the URL. Ok, everything looks good. Now search for something very specific, like, ahem, reprehenderit. And, yes! A *much* smaller result. Try an author: Ernie: got it!

Woo! This is great! But, we can do more! Next, let's learn about a Twig global variable that can help us fill in this input box when we search. Then, it's finally time to add a *join* to our custom query.

Chapter 14: Query Joins & Solving the N+1 Problem

Let's start with a *little*, annoying problem: when we search, the term does *not* show up inside the search box. That sucks! Go back to the template. Ok, just add `value=""` to the search field. Now, hmm, how can we get that `q` query parameter? Well, of course, we could pass a new `q` variable into the template and use it. That's *totally* valid.

But, of course, there's a shortcut! In the template, use `{{ app.request.query.get('q') }}`:

```
62 lines | templates/comment_admin/index.html.twig
... lines 1 - 6
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments</h1>
11
12             <form>
13                 <div class="input-group mb-3">
14                     <input type="text"
15
16
17                     value="{{ app.request.query.get('q') }}"
18
19                 >
20
21
22
23
24
25
26             </div>
27         </form>
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59     </div>
60 </div>
61 {% endblock %}
```

Before we talk about this black magic, try it: refresh. It works! Woo!

Back in Twig, I *hope* you're now wondering: where the heck did this `app` variable come from? When you use Twig with Symfony, you get exactly *one* global variable - *completely* for free - called `app`. In fact, find your terminal, and re-run the trusty:

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

Yep! Under "Globals", we have one: `app`. And it's an *object* called, um, `AppVariable`. Ah, clever name Symfony!

Back in your editor, type Shift+Shift and search for this: `AppVariable`. Cool! Ignore the setter methods on top - these are just for setup. The `AppVariable` has a couple of handy methods: `getToken()` and `getUser()` both relate to security. Then, hey! There's our favorite `getRequest()` method, then `getSession()`, `getEnvironment()`, `getDebug()` and something called "flashes", which helps render temporary messages, usually for forms.

It's not a *huge* class, but it's *handy*! We're calling `getRequest()`, then `.query.get()`, which ultimately does the same thing as the code in our controller: go to the `query` property and call `get()`:

25 lines | [src/Controller/CommentAdminController.php](#)

```
... lines 1 - 9
10 class CommentAdminController extends Controller
11 {
... lines 12 - 14
15     public function index(CommentRepository $repository, Request $request)
16     {
17         $q = $request->query->get('q');
... lines 18 - 22
23     }
24 }
```

Cool. So now it's time for a totally *new* challenge. In addition to searching a comment's content and author name, I *also* want to search the comment's, article's title. For example, if I search for "Bacon", that should return some results.

The Twig For-Else Feature

Oh, by the way, here's a fun Twig feature. When we get *zero* results, we should probably print a nice message. On a Twig for loop, you can put an else at the end. Add a `<td colspan="4">`, a centering class, and: No comments found:

68 lines | [templates/comment_admin/index.html.twig](#)

```
... lines 1 - 6
7 {% block content_body %}
8     <div class="row">
9         <div class="col-sm-12">
... lines 10 - 28
29         <table class="table table-striped">
... lines 30 - 37
38             <tbody>
39                 {% for comment in comments %}
... lines 40 - 55
56                 {% else %}
57                     <tr>
58                         <td colspan="4" class="text-center">
59                             No comments found
60                         </td>
61                     </tr>
62                 {% endfor %}
63             </tbody>
64         </table>
65     </div>
66 </div>
67 {% endblock %}
```

Go back and try it! It works! Pff, except for my not-awesome styling skills. Use text-center:

68 lines | [templates/comment_admin/index.html.twig](#)

```
... lines 1 - 6
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
... lines 10 - 28
29      <table class="table table-striped">
... lines 30 - 37
38          <tbody>
39              {% for comment in comments %}
... lines 40 - 55
56                  {% else %}
57                      <tr>
58                          <td colspan="4" class="text-center">
59                              No comments found
60                          </td>
61                      </tr>
62                  {% endfor %}
63              </tbody>
64          </table>
65      </div>
66  </div>
67  {% endblock %}
```

That's better.

[Adding a Join](#)

Anyways, back to the main event: how can we *also* search the article's title? In SQL, if we need to reference another table inside the WHERE clause, then we need to *join* to that table first.

In this case, we want to join from comment to article: an inner join is perfect. How can you do this with the QueryBuilder? Oh, it's awesome: `->innerJoin('c.article', 'a')`:

82 lines | [src/Repository/CommentRepository.php](#)

```
... lines 1 - 15
16 class CommentRepository extends ServiceEntityRepository
17 {
... lines 18 - 34
35     public function findAllWithSearch(?string $term)
36     {
37         $qb = $this->createQueryBuilder('c')
38             ->innerJoin('c.article', 'a');
... lines 39 - 50
51     }
... lines 52 - 80
81 }
```

That's *it*. When we say `c.article`, we're actually referencing the `article` property on `Comment`:

96 lines | [src/Entity/Comment.php](#)

```
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 31
32 /**
33  * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34  * @ORM\JoinColumn(nullable=false)
35  */
36 private $article;
... lines 37 - 94
95 }
```

Thanks to that, we can be lazy! We don't need to explain to Doctrine *how* to join - we don't need an ON article.id = comment.article_id. Nah, Doctrine can figure that out on its own. The second argument - a - will be the "alias" for Article for the rest of the query.

Before we do *anything* else, go refresh the page. Nothing changes yet, but go open the profiler and click to look at the query. Yes, it's perfect! It still *only* selects from comment, but it *does* have the INNER JOIN to article!

We can now *easily* reference the article somewhere else in the query. Inside the andWhere(), add OR a.title LIKE :term:

82 lines | [src/Repository/CommentRepository.php](#)

```
... lines 1 - 15
16 class CommentRepository extends ServiceEntityRepository
17 {
... lines 18 - 34
35 public function findAllWithSearch(?string $term)
36 {
37     $qb = $this->createQueryBuilder('c')
38         ->innerJoin('c.article', 'a');
39
40     if ($term) {
41         $qb->andWhere('c.content LIKE :term OR c.authorName LIKE :term OR a.title LIKE :term')
... line 42
43     ;
44 }
... lines 45 - 50
51 }
... lines 52 - 80
81 }
```

That's all you need. Move back and refresh again. It works *instantly*. Check out the query again: this time we have the INNER JOIN *and* the extra logic inside the WHERE clause. Building queries with the query builder is not *so* different than writing them by hand.

Solving the N+1 (Extra Queries) Problem

You'll *also* notice that we still have a lot of queries: 7 to be exact. And that's because we are *still* suffering from the N+1 problem: as we loop over each Comment row, when we reference an article's data, a query is made for that article.

But wait... does that make sense anymore? I mean, if we're *already* making a JOIN to the article table, isn't this extra query unnecessary? Doesn't Doctrine *already* have *all* the data it needs from the first query, thanks to the join?

The answer is... no, or, at least not yet. Remember: while the query *does* join to article, it *only* selects data from comment. We are *not* fetching *any* article data. That's why the extra 6 queries are still needed.

But at this point, the solution to the N+1 problem is *dead* simple. Go back to CommentRepository and put ->addSelect('a'):

83 lines | [src/Repository/CommentRepository.php](#)

```
... lines 1 - 15
16 class CommentRepository extends ServiceEntityRepository
17 {
... lines 18 - 34
35     public function findAllWithSearch(?string $term)
36     {
37         $qb = $this->createQueryBuilder('c')
38             ->innerJoin('c.article', 'a')
39             ->addSelect('a');
... lines 40 - 51
52     }
... lines 53 - 81
82 }
```

When you create a QueryBuilder from inside a repository, that QueryBuilder automatically knows to select from its own table, so, from c. With this line, we're telling the QueryBuilder to select all of the comment columns *and* all of the article columns.

Try it: head back and refresh. It *still* works! But, yes! We're down to just *one* query. Go check it out: yep! It selects everything from comment *and* article.

The moral of the story is this: *if* your page has a lot of queries because Doctrine is making extra queries across a relationship, just *join* over that relationship and use addSelect() to fetch all the data you need at once.

But... there *is* one confusing thing about this. We're now selecting all of the comment data *and* all of the article data. But... you'll notice, the page still works! What I mean is, *even* though we're suddenly selecting more data, our findAllWithSearch() method *still* returns *exactly* what it did before: it returns a array of Comment objects. It does *not*, for example, now return Comment *and* Article objects.

Instead, Doctrine takes that extra article data and stores it in the background for later. But, the new addSelect() does *not* affect the return value. That's *way* different than using raw SQL.

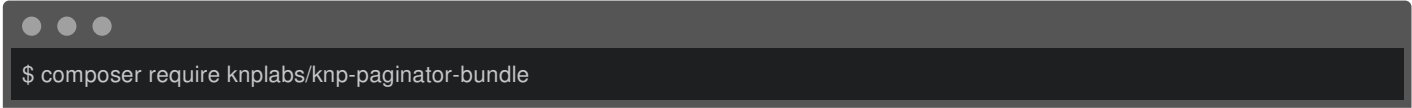
It's now time to cross off a todo from earlier: let's add pagination!

Chapter 15: Pagination

On production - because The SpaceBar is going to be a *huge* hit with a *lot* of thoughtful comments - this list will eventually become, *way* long. Not only will this page become hard to use, it will quickly slow down until it stops working. If you ever need to query and render more than 100 Doctrine entities, you're going to have slow loading times. If you try to print 1000, your page probably just won't load.

But no problem! Printing that many results is a total bummer for usability anyways! And that's why the Internet has a tried-and-true solution for this: pagination. Doctrine itself doesn't come with any pagination features. But, it doesn't need to: there are a few great libraries that *do*.

Search for KnpPaginatorBundle. As usual, my disclaimer is that I did *nothing* to help build this bundle, I just think it's great. Find the composer require line, copy that, go to your terminal and paste:



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

While that's installing, go back to its documentation. As I *love* to tell you, over and over again, the *main* thing a bundle gives you is new services. And that's 100% true for this bundle.

But before we talk more about that, notice that this has some details about enabling your bundle. That happens automatically in Symfony 4 thanks to Flex. So, ignore it.


[Paginator Usage and the Autowiring Alias](#)

Anyways, look down at the Usage example. Hmm, from a controller, it says to use `$this->get('knp_paginator')` to get some paginator service. Then, you pass that a query, the current page number, read from a `?page` query parameter, and the number of items you want per page. The paginator handles the rest! If you want 10 results per page and you're on page 3, the paginator will fetch *only* the results you need by adding a LIMIT and OFFSET to your query.

The *one* tricky thing is that the documentation is a little bit out of date. The `$this->get()` method - which is the same as saying `$this->container->get()` - is the historic way to fetch a service out of the container by using its *id*. Depending on your setup, that may or may not even be possible in Symfony 4. And, in general, it's *no* longer considered a best-practice. Instead, you should use dependency injection, which almost always means, autowiring.

But, hmm, it doesn't say anything about autowiring here. That's a problem: the bundle needs to tell us what class or interface we can use to autowire the paginator service. Ah, don't worry: we can figure it out on our own!

Go back to your terminal Excellent! The install finished. Now run:



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

Search for pager. Boom! Apparently there is a PaginatorInterface we can use to get that *exact* service. We are in business!

[Using the Paginator](#)

Back in CommentAdminController, add that as the 3rd argument: PaginatorInterface. Make sure to auto-complete this to get the use statement. Call the arg `$paginator`:

```

33 lines | src/Controller/CommentAdminController.php
... lines 1 - 5
6 use Knp\Component\Pager\PaginatorInterface;
... lines 7 - 10
11 class CommentAdminController extends Controller
12 {
... lines 13 - 15
16 public function index(CommentRepository $repository, Request $request, PaginatorInterface $paginator)
17 {
... lines 18 - 30
31 }
32 }

```

Next, go back to the docs and copy the `$pagination = section, return,` and paste:

```

33 lines | src/Controller/CommentAdminController.php
... lines 1 - 10
11 class CommentAdminController extends Controller
12 {
... lines 13 - 15
16 public function index(CommentRepository $repository, Request $request, PaginatorInterface $paginator)
17 {
... lines 18 - 21
22     $pagination = $paginator->paginate(
23         $queryBuilder, /* query NOT result */
24         $request->query->getInt('page', 1)/*page number*/,
25         10/*limit per page*/
26     );
... lines 27 - 30
31 }
32 }

```

Ok, so what should we use for `$query`? When you use a paginator, there is an important, practical change: we are no longer responsible for actually *executing* the query. Nope, we're now only responsible for *building* a query and passing it to the paginator. This `$query` variable should be a `QueryBuilder`.

So, back in `CommentRepository`, let's refactor this method to return that instead. Remove the `@returns` and, instead, use a `QueryBuilder` return type:

```

81 lines | src/Repository/CommentRepository.php
... lines 1 - 16
17 class CommentRepository extends ServiceEntityRepository
18 {
... lines 19 - 31
32 /**
33  * @param string|null $term
34  */
35 public function getWithSearchQueryBuilder(?string $term): QueryBuilder
36 {
... lines 37 - 49
50 }
... lines 51 - 79
80 }

```

Next, at the bottom, remove the `getQuery()` and `getResults()` lines:

```

81 lines | src/Repository/CommentRepository.php
... lines 1 - 16
17 class CommentRepository extends ServiceEntityRepository
18 {
... lines 19 - 31
32 /**
33  * @param string|null $term
34  */
35 public function getWithSearchQueryBuilder(?string $term): QueryBuilder
36 {
... lines 37 - 46
47     return $qb
48         ->orderBy('c.createdAt', 'DESC')
49     ;
50 }
... lines 51 - 79
80 }

```

Finally, rename the method to `getWithSearchQueryBuilder()`:

```

81 lines | src/Repository/CommentRepository.php
... lines 1 - 16
17 class CommentRepository extends ServiceEntityRepository
18 {
... lines 19 - 34
35 public function getWithSearchQueryBuilder(?string $term): QueryBuilder
36 {
... lines 37 - 49
50 }
... lines 51 - 79
80 }

```

Perfect! Back in the controller, add `$queryBuilder = $repository->getWithSearchQueryBuilder($q)`. Pass this below:

```

33 lines | src/Controller/CommentAdminController.php
... lines 1 - 10
11 class CommentAdminController extends Controller
12 {
... lines 13 - 15
16 public function index(CommentRepository $repository, Request $request, PaginatorInterface $paginator)
17 {
... lines 18 - 19
20     $queryBuilder = $repository->getWithSearchQueryBuilder($q);
21
22     $pagination = $paginator->paginate(
23         $queryBuilder, /* query NOT result */
24         $request->query->getInt('page', 1)/*page number*/,
25         10/*limit per page*/
26     );
... lines 27 - 30
31 }
32 }

```

Finally, instead of passing comments into the template, pass this pagination variable:

33 lines | [src/Controller/CommentAdminController.php](#)

```
... lines 1 - 10
11 class CommentAdminController extends Controller
12 {
... lines 13 - 15
16     public function index(CommentRepository $repository, Request $request, PaginatorInterface $paginator)
17     {
... lines 18 - 19
20         $queryBuilder = $repository->getWithSearchQueryBuilder($q);
21
22         $pagination = $paginator->paginate(
23             $queryBuilder, /* query NOT result */
24             $request->query->getInt('page', 1)/*page number*/,
25             10/*limit per page*/
26         );
27
28         return $this->render('comment_admin/index.html.twig', [
29             'pagination' => $pagination,
30         ]);
31     }
32 }
```

Open index.html.twig so we can make changes there. First, at the top, let's print the *total* number of comments because we will now only show 10 on each page. To do that, go back to the docs. Ah, this is perfect. Use: `pagination.getTotalItemCount()`:

70 lines | [templates/comment_admin/index.html.twig](#)

```
... lines 1 - 6
7 {% block content_body %}
8     <div class="row">
9         <div class="col-sm-12">
10             <h1>Manage Comments ({{ pagination.getTotalItemCount }})</h1>
... lines 11 - 66
67         </div>
68     </div>
69 {% endblock %}
```

Next, down in the loop, update this to for comment in pagination:

```

70 lines | templates/comment_admin/index.html.twig
... lines 1 - 6
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments ({{ pagination.getTotalItemCount }})</h1>
... lines 11 - 28
29         <table class="table table-striped">
... lines 30 - 37
38             <tbody>
39                 {% for comment in pagination %}
... lines 40 - 61
62                 {% endfor %}
63             </tbody>
64         </table>
... lines 65 - 66
67     </div>
68 </div>
69 {% endblock %}

```

Yes, pagination is an *object*. But, you *can* loop over it to get the comments for the current page only.

Oh, and at the bottom, we need some navigation to help the user go to the other pages. That's really easy: on the docs, copy the `kn_p_pagination_render()` line and, paste!

```

70 lines | templates/comment_admin/index.html.twig
... lines 1 - 6
7  {% block content_body %}
8      <div class="row">
9          <div class="col-sm-12">
10             <h1>Manage Comments ({{ pagination.getTotalItemCount }})</h1>
... lines 11 - 28
29         <table class="table table-striped">
... lines 30 - 37
38             <tbody>
39                 {% for comment in pagination %}
... lines 40 - 61
62                 {% endfor %}
63             </tbody>
64         </table>
65
66         {{ knp_pagination_render(pagination) }}
67     </div>
68 </div>
69 {% endblock %}

```

Phew! Let's go check it out! Yes! 100 total results, but only 10 on this page. We can click to page 2, then 3 and so-on. Heck, the search even works! Try something really common, like `est`. The URL has the `?q=` query parameter. And, if you change pages, it stays: this is page 2 of that search. Dang, that's awesome.

[Using the Bootstrap Pager Navigation Theme](#)

Of course, there's one *super* minor problem... um... dang, that navigation looks *horrible*. But, we can fix that! The bundle comes with a bunch of different *themes* for the navigation. Scroll back up to the configuration example. Obviously, you don't *need* to configure *anything* on this bundle. But, there are several options. The most important one is this: `template.pagination`. This determines which template is used to build the navigation links. *And*, it ships with one for Bootstrap 4, which is what we're using. Booya!

So, first question: where should this configuration live? Sometimes, a recipe will create a file for us, like `stof_doctrine_extensions.yaml`. But in this case, that didn't happen. And that's ok! Not every bundle *needs* to give you a config file. Just create it by hand: `knp_paginator.yaml`.

As usual, the filename matches the root config key. But, we know from previous tutorials that the filename is actually meaningless. Next, copy the config down to the pagination line, move over, paste, then remove all the stuff we don't need. Finally, copy the bootstrap v4 template name and, paste:

```
4 lines | config/packages/knp_paginator.yaml
1  knp_paginator:
2    template:
3      pagination: '@KnpPaginator/Pagination/twitter_bootstrap_v4_pagination.html.twig'
```

We're ready! Move back and refresh. Boom! It still works! It's beautiful! We rock! Our pagination is awesome! I'm super happy!

Now that this is perfect, let's turn to our last big topic: generating a *totally* new, ManyToMany relationship.

Chapter 16: The 4 (2?) Possible Relation Types

Remember *way* back when we used the `make:entity` command to generate the relationship between `Comment` and `Article`? When we did this, the command told us that there are *four* different types of relationships: `ManyToOne`, `OneToMany`, `ManyToMany` and `OneToOne`.

But, that's not really true... and the truth is a lot more interesting. For example, we quickly learned that `ManyToOne` and `OneToMany` are really two different ways to refer to the *same* relationship! `Comment` has a `ManyToOne` relationship to `Article`:

```
96 lines | src/Entity/Comment.php
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 31
32 /**
33  * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34  * @ORM\JoinColumn(nullable=false)
35  */
36 private $article;
... lines 37 - 94
95 }
```

But that same database relationship can be described as a `OneToMany` from `Article` to `Comment`.

[OneToOne: The Cousin of ManyToOne](#)

This means that there are *truly* only *three* different types of relationships: `ManyToOne`, `ManyToMany` and `OneToOne`. Um, ok, this is embarrassing. That's not true either. Yea, A `OneToOne` relationship is more or less the same as a `ManyToOne`. `OneToOne` is kind of weird. Here's an example: suppose you have a `User` entity and you decide to create a `Profile` entity that contains *more* data about that one user. In this example, each `User` has exactly one `Profile` and each `Profile` is linked to exactly one `User`.

But, in the database, this looks exactly like a `ManyToOne` relationship! For example, our `ManyToOne` relationship causes the `comment` table to have an `article_id` foreign key column. If you had a `OneToOne` relationship between some `Profile` and `User` entities, then the `profile` table would have a `user_id` foreign key to the `user` table. The *only* difference is that doctrine would make that column unique to prevent you from accidentally linking multiple profiles to the same user.

The point is, `OneToOne` relationships are kind of `ManyToOne` relationships in disguise. They also not very common, and I don't really like them.

[The 2 Types of Relationships](#)

So, *really*, if you are trying to figure out *which* relationship type to use in a situation... well... there are only *two* types: (1) `ManyToOne/OneToMany` or (2) `ManyToMany`.

For `ManyToMany`, imagine you have a `Tag` entity and you want to be able to add tags to articles. So, each article will have many tags. And, each `Tag` may be related to many articles. *That* is a `ManyToMany` relationship. And *that* is *exactly* what we're going to build.

[Building the Tag Entity](#)

Let's create the new `Tag` entity class first. Find your terminal and run:

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


Name it Tag and give it two properties: name, as a string and slug also as a string, so that we can use the tag in a URL later.

Cool! Before generating the migration, open the new class:

```
58 lines | src/Entity/Tag.php
... lines 1 - 2
3 namespace App\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 /**
8  * @ORM\Entity(repositoryClass="App\Repository\TagRepository")
9  */
10 class Tag
11 {
12     /**
13      * @ORM\Id()
14      * @ORM\GeneratedValue()
15      * @ORM\Column(type="integer")
16      */
17     private $id;
18
19     /**
20      * @ORM\Column(type="string", length=255)
21      */
22     private $name;
23
24     /**
25      * @ORM\Column(type="string", length=180)
26      */
27     private $slug;
28
29     public function getId()
30     {
31         return $this->id;
32     }
33
34     public function getName(): ?string
35     {
36         return $this->name;
37     }
38
39     public function setName(string $name): self
40     {
41         $this->name = $name;
42
43         return $this;
44     }
45
46     public function getSlug(): ?string
47     {
48         return $this->slug;
49     }
50
51     public function setSlug(string $slug): self
52     {
```

```

53     $this->slug = $slug;
54
55     return $this;
56 }
57 }

```

No surprises: name and slug. At the top, use our favorite TimestampableEntity trait:

```

63 lines | src/Entity/Tag.php
... lines 1 - 6
7     use Gedmo\Timestampable\Traits\TimestampableEntity;
... lines 8 - 11
12    class Tag
13    {
14        use TimestampableEntity;
... lines 15 - 61
62    }

```

And, just like we did in Article, configure the slug to generate automatically. Copy the slug annotation and paste that above the slug property:

```

63 lines | src/Entity/Tag.php
... lines 1 - 11
12    class Tag
13    {
... lines 14 - 27
28        /**
29         * @ORM\Column(type="string", length=180, unique=true)
30         * @Gedmo\Slug(fields={"name"})
31         */
32        private $slug;
... lines 33 - 61
62    }

```

Oh, but we need a use statement for the annotation. An easy way to add it is to temporarily type @Slug on the next line and hit tab to auto-complete it. Then, delete it: that was enough to make sure the use statement was added on top:

```

63 lines | src/Entity/Tag.php
... lines 1 - 5
6     use Gedmo\Mapping\Annotation as Gedmo;
... lines 7 - 63

```

Let's also make the slug column unique:

```

63 lines | src/Entity/Tag.php
... lines 1 - 11
12    class Tag
13    {
... lines 14 - 27
28        /**
29         * @ORM\Column(type="string", length=180, unique=true)
... line 30
31         */
32        private $slug;
... lines 33 - 61
62    }

```

Great! The entity is ready. Go back to your terminal and make that migration!

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

Whoops! My bad! Maybe you saw my mistake. Change the Slug annotation from title to name:

```
63 lines | src/Entity/Tag.php
... lines 1 - 11
12 class Tag
13 {
... lines 14 - 27
28 /**
... line 29
30  * @Gedmo\Slug(fields={"name"})
31  */
32 private $slug;
... lines 33 - 61
62 }
```

Generate the migration again:

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

Got it! Open that class to make sure it looks right:

```
29 lines | src/Migrations/Version20180501142420.php
... lines 1 - 2
3 namespace DoctrineMigrations;
4
5 use Doctrine\DBAL\Schema\Schema;
6 use Doctrine\Migrations\AbstractMigration;
7
8 /**
9  * Auto-generated Migration: Please modify to your needs!
10 */
11 class Version20180501142420 extends AbstractMigration
12 {
13     public function up(Schema $schema): void
14     {
15         // this up() migration is auto-generated, please modify it to your needs
16         $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
17
18         $this->addSql('CREATE TABLE tag (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255) NOT NULL, slug VARCHAR(255) NOT NULL);');
19     }
20
21     public function down(Schema $schema): void
22     {
23         // this down() migration is auto-generated, please modify it to your needs
24         $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
25
26         $this->addSql('DROP TABLE tag');
27     }
28 }
```

Yep: CREATE TABLE tag. Go run it:

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

Tip

If you get an error like:

Syntax error or access violation: 1071 Specified key was too long...

No worries! Just open your entity and modify the `@Column` annotation above the `$slug` property: set `length=180`. Then, remove the migration and re-run the last 2 commands. Or update MySQL to version 5.7 or higher.

Now that the entity & database are setup, we need some dummy data! Run:

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

Call it TagFixture. Then, like always, open that class so we can tweak it:

```
18 lines | src/DataFixtures/TagFixture.php
... lines 1 - 2
3  namespace App\DataFixtures;
4
5  use Doctrine\Bundle\FixturesBundle\Fixture;
6  use Doctrine\Common\Persistence\ObjectManager;
7
8  class TagFixture extends Fixture
9  {
10     public function load(ObjectManager $manager)
11     {
12         // $product = new Product();
13         // $manager->persist($product);
14
15         $manager->flush();
16     }
17 }
```

First, extend BaseFixture, rename load() to loadData() and make it protected:

```
19 lines | src/DataFixtures/TagFixture.php
... lines 1 - 7
8  class TagFixture extends BaseFixture
9  {
10     protected function loadData(ObjectManager $manager)
11     {
12         ... lines 12 - 16
17     }
18 }
```

We also don't need this use statement anymore. Call our trusty `$this->createMany()` to create 10 tags:

```

19 lines | src/DataFixtures/TagFixture.php
... lines 1 - 4
5 use App\Entity\Tag;
... lines 6 - 7
8 class TagFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(Tag::class, 10, function(Tag $tag) {
... line 13
14         });
15
16         $manager->flush();
17     }
18 }

```

For the name, use `$tag->setName()` with `$this->faker->realText()` and 20, to get about that many characters:

```

19 lines | src/DataFixtures/TagFixture.php
... lines 1 - 7
8 class TagFixture extends BaseFixture
9 {
10     protected function loadData(ObjectManager $manager)
11     {
12         $this->createMany(Tag::class, 10, function(Tag $tag) {
13             $tag->setName($this->faker->realText(20));
14         });
15
16         $manager->flush();
17     }
18 }

```

We *could* use `$this->faker->word` to get a random word, but that word would be in Latin. The `realText()` method will give us a *few* words, actually, but they will sound, at least "kinda" real.

And, that's all we need! To make sure it works, run:

```

$ php bin/console doctrine:fixtures:load

```

We are ready! Article entity, check! Tag entity, check, check! It's time to create the ManyToMany relationship.

Chapter 17: ManyToMany Relationship

We need the ability to add *tags* to each Article. And that means, we need a new relationship! Like always, we could add this by hand. But, the generator can help us. At your terminal, run:

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

But, hmm. Which entity should we update? We could add a new property to Article called tags or... I guess we could also add a new property to Tag called articles. That's really the *same* relationship, just viewed from two different sides.

And... yea! We could choose to update *either* class. The side you choose *will* make a subtle difference, and we'll learn about it soon. Let's update Article. For the property name, use tags. Remember, we need to *stop* thinking about the database and *only* think about our objects. In PHP, I want an Article object to have many Tag objects. So, the property should be called tags.

For type, use the fake relation type to activate the relationship wizard. We want to relate this to Tag and... perfect! Here is our menu of relationship options! I already hinted that this will be a ManyToMany relationship. But, let's look at the description to see if it fits. Each article can have many tags. And, each tag can relate to many articles. Yep, that's us! This is a ManyToMany relationship.

And just like last time, it asks us if we *also* want to map the *other* side of the relationship. This is optional, and is *only* for convenience. *If* we map the other side, we'll be able to say `$tag->getArticles()`. That may or may not be useful for us, but let's say yes. Call the field articles, because it will hold an array of Article objects.

And, that's it! Hit enter to finish.

[Looking at the Generating Entities](#)

Exciting! Let's see what changes this made. Open Article first:

```
247 lines | src/Entity/Article.php
... lines 1 - 15
16 class Article
17 {
... lines 18 - 68
69 /**
70  * @ORM\ManyToMany(targetEntity="App\Entity\Tag", inversedBy="articles")
71  */
72 private $tags;
... lines 73 - 245
246 }
```

Yes! Here is the new tags property: it's a ManyToMany that points to the Tag entity. And, like we saw earlier with comments, whenever you have a relationship that holds *many* objects, in your constructor, you need to initialize that property to a new ArrayCollection:

```

247 lines | src/Entity/Article.php
... lines 1 - 15
16 class Article
17 {
... lines 18 - 73
74     public function __construct()
75     {
... line 76
77         $this->tags = new ArrayCollection();
78     }
... lines 79 - 245
246 }

```

The generator did that for us.

At the bottom, instead of a getter & setter, we have a getter, adder & remover:

```

247 lines | src/Entity/Article.php
... lines 1 - 15
16 class Article
17 {
... lines 18 - 220
221     /**
222      * @return Collection|Tag[]
223      */
224     public function getTags(): Collection
225     {
226         return $this->tags;
227     }
228
229     public function addTag(Tag $tag): self
230     {
231         if (!$this->tags->contains($tag)) {
232             $this->tags[] = $tag;
233         }
234
235         return $this;
236     }
237
238     public function removeTag(Tag $tag): self
239     {
240         if ($this->tags->contains($tag)) {
241             $this->tags->removeElement($tag);
242         }
243
244         return $this;
245     }
246 }

```

There's no special reason for that: the adder & remover methods are just convenient.

Next, open Tag:

103 lines | [src/Entity/Tag.php](#)

... lines 1 - 13

14 class Tag

15 {

... lines 16 - 35

36 /**

37 * @ORM\ManyToOne(targetEntity="App\Entity\Article", mappedBy="tags")

38 */

39 private \$articles;

40

41 public function __construct()

42 {

43 \$this->articles = new ArrayCollection();

44 }

... lines 45 - 74

75 /**

76 * @return Collection|Article[]

77 */

78 public function getArticles(): Collection

79 {

80 return \$this->articles;

81 }

82

83 public function addArticle(Article \$article): self

84 {

85 if (!\$this->articles->contains(\$article)) {

86 \$this->articles[] = \$article;

87 \$article->addTag(\$this);

88 }

89

90 return \$this;

91 }

92

93 public function removeArticle(Article \$article): self

94 {

95 if (\$this->articles->contains(\$article)) {

96 \$this->articles->removeElement(\$article);

97 \$article->removeTag(\$this);

98 }

99

100 return \$this;

101 }

102 }

The code here is almost *identical*: a ManyToMany pointing back to Article and, at the bottom, getter, adder & remover methods.

Owning Versus Inverse Sides

Great! But, which side is the owning side and which is the inverse side of the relationship? Open Comment:

96 lines | [src/Entity/Comment.php](#)

```
... lines 1 - 10
11 class Comment
12 {
... lines 13 - 31
32 /**
33  * @ORM\ManyToOne(targetEntity="App\Entity\Article", inversedBy="comments")
34  * @ORM\JoinColumn(nullable=false)
35  */
36 private $article;
... lines 37 - 94
95 }
```

Remember, with a ManyToOne / OneToMany relationship, the ManyToOne side is *always* the owning side of the relation. That's easy to remember, because this is where the column lives in the database: the comment table has an article_id column.

But, with a ManyToMany relationship, well, *both* sides are ManyToMany! In Article, ManyToMany. In Tag, the same! So, which side is the *owning* side?

The answer lives in Article. See that inversedBy="articles" config?

247 lines | [src/Entity/Article.php](#)

```
... lines 1 - 15
16 class Article
17 {
... lines 18 - 68
69 /**
70  * @ORM\ManyToMany(targetEntity="App\Entity\Tag", inversedBy="articles")
71  */
72 private $tags;
... lines 73 - 245
246 }
```

That points to the articles property in Tag. On the other side, we have mappedBy="tags", which points *back* to Article:

103 lines | [src/Entity/Tag.php](#)

```
... lines 1 - 13
14 class Tag
15 {
... lines 16 - 35
36 /**
37  * @ORM\ManyToMany(targetEntity="App\Entity\Article", mappedBy="tags")
38  */
39 private $articles;
... lines 40 - 101
102 }
```


Here's the point: with a ManyToMany relationship, you *choose* the owning side by where the inversedBy versus mappedBy config lives. The generator configured things so that Article holds the owning side because that's the entity we chose to update with make:entity.

Remember, all of this owning versus inverse stuff is important because, when Doctrine saves an entity, it *only* looks at the *owning* side of the relationship to figure out what to save to the database. So, if we add tags to an article, Doctrine will save that correctly. But, if you added articles to a tag and save, Doctrine would do nothing. Well, in practice, if you use make:entity, that's not true. Why? Because the generated code *synchronizes* the owning side. If you call \$tag->addArticle(), inside, that calls \$article->addTag():

```
103 lines | src/Entity/Tag.php
... lines 1 - 13
14 class Tag
15 {
... lines 16 - 82
83     public function addArticle(Article $article): self
84     {
85         if (!$this->articles->contains($article)) {
86             $this->articles[] = $article;
87             $article->addTag($this);
88         }
89
90         return $this;
91     }
... lines 92 - 101
102 }
```

Generating the Migration

Enough of that! Let's generate the migration:



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

Cool! Go open that file:

```

31 lines | src/Migrations/Version20180501143055.php
... lines 1 - 2
3  namespace DoctrineMigrations;
4
5  use Doctrine\DBAL\Migrations\AbstractMigration;
6  use Doctrine\DBAL\Schema\Schema;
7
8  /**
9   * Auto-generated Migration: Please modify to your needs!
10  */
11  class Version20180501143055 extends AbstractMigration
12  {
13      public function up(Schema $schema)
14      {
15          // this up() migration is auto-generated, please modify it to your needs
16          $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
17
18          $this->addSql('CREATE TABLE article_tag (article_id INT NOT NULL, tag_id INT NOT NULL, INDEX IDX_919694F97294869C (article_id), INDEX IDX_919694F9BAD26311 (tag_id), PRIMARY KEY (article_id, tag_id))');
19          $this->addSql('ALTER TABLE article_tag ADD CONSTRAINT FK_919694F97294869C FOREIGN KEY (article_id) REFERENCES article (id) ON DELETE CASCADE');
20          $this->addSql('ALTER TABLE article_tag ADD CONSTRAINT FK_919694F9BAD26311 FOREIGN KEY (tag_id) REFERENCES tag (id) ON DELETE CASCADE');
21      }
22
23      public function down(Schema $schema)
24      {
25          // this down() migration is auto-generated, please modify it to your needs
26          $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\'');
27
28          $this->addSql('DROP TABLE article_tag');
29      }
30  }

```

Woh! It creates a new *table*! Of course! That's how you model a ManyToMany relationship in a relational database. It creates an `article_tag` table with only two fields: `article_id` and `tag_id`.

This is very different than anything we've seen so far with Doctrine. This is the *first* time - and really, the only time - that you will have a table in the database, that has *no* direct entity class. This table is created magically by Doctrine to help us relate tags and articles. *And*, as we'll see next, Doctrine will also automatically insert and delete records from this table as we add and remove tags from an article.

Now, run the migration:

```

$ php bin/console doctrine:migrations:migrate

```

Let's go tag some articles!

Chapter 18: Saving a ManyToMany Relation + Joins

We *now* know that a ManyToMany relationship works with the help of a join table. The question now is: how can we insert new records *into* that join table? How can I relate an Article to several tags?

The answer is *exactly* the same as our ManyToOne relation. Start by opening BaseFixture. At the bottom, I'm going to paste in a new protected function called getRandomReferences():

```
72 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 61
62     protected function getRandomReferences(string $className, int $count)
63     {
64         $references = [];
65         while (count($references) < $count) {
66             $references[] = $this->getRandomReference($className);
67         }
68
69         return $references;
70     }
71 }
```

We already have a getRandomReference() method that returns just *one* object:

```
72 lines | src/DataFixtures/BaseFixture.php
... lines 1 - 9
10 abstract class BaseFixture extends Fixture
11 {
... lines 12 - 41
42     protected function getRandomReference(string $className) {
43         if (!isset($this->referencesIndex[$className])) {
44             $this->referencesIndex[$className] = [];
45
46             foreach ($this->referenceRepository->getReferences() as $key => $ref) {
47                 if (strpos($key, $className.'_') === 0) {
48                     $this->referencesIndex[$className][] = $key;
49                 }
50             }
51         }
52
53         if (empty($this->referencesIndex[$className])) {
54             throw new \Exception(sprintf('Cannot find any references for class "%s"', $className));
55         }
56
57         $randomReferenceKey = $this->faker->randomElement($this->referencesIndex[$className]);
58
59         return $this->getReference($randomReferenceKey);
60     }
... lines 61 - 70
71 }
```

This is the same, but you can pass it a class name and how *many* of those objects you want back. The objects you get back may or may not be a unique set. Hey, my method isn't perfect, but, it's good enough.

Next, in `ArticleFixtures`, *this* is where we'll set the relationship. And *that* means, we need to make sure that `TagFixture` is loaded first so that the tags *actually* exist. At the top, add implements `DependentFixtureInterface`:

```
80 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
8  use Doctrine\Common\DataFixtures\DependentFixtureInterface;
... lines 9 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 78
79 }
```

Then, I'll go to the "Code"->"Generate" menu - or Command+N on a Mac - select "Implement Methods" and choose `getDependencies()`. We now depend on `TagFixture::class`:

```
80 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 7
8  use Doctrine\Common\DataFixtures\DependentFixtureInterface;
... lines 9 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 72
73     public function getDependencies()
74     {
75         return [
76             TagFixture::class,
77         ];
78     }
79 }
```

Above, let's *first* get some tag objects: `$tags = $this->getRandomReferences()` and pass it `Tag::class`, and then, let's fetch `$this->faker->numberBetween()` zero and five. So, find 0 to 5 random tags:

```
80 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 29
30     public function loadData(ObjectManager $manager)
31     {
32         $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 33 - 63
64             $tags = $this->getRandomReferences(Tag::class, $this->faker->numberBetween(0, 5));
... lines 65 - 67
68         });
... lines 69 - 70
71     }
... lines 72 - 78
79 }
```

And just to make sure you that this *does* give us `Tag` objects, `dump($tags)` and `die`. Now, find your terminal and run:

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

Explaining Proxies

Perfect! These *are* Tag *objects*. Oh, by the way, sometimes you may notice that your entity's class name is prefixed by this weird Proxies stuff. When you see that, ignore it. A "Proxy" is a special class that Doctrine generates and sometimes wraps *around* your real entity objects. Doctrine does this so that it can perform its relationship lazy-loading magic.

Actually, check this out: it *looks* like all the data on this Tag is null! But, that's a lie! As *soon* as you reference any data on that Tag, Doctrine will query for the data and fill it in. That's lazy-loading in action.

Let me show you: add a foreach over \$tags as \$tag:

```
80 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 29
30 public function loadData(ObjectManager $manager)
31 {
... lines 33 - 63
64     $tags = $this->getRandomReferences(Tag::class, $this->faker->numberBetween(0, 5));
65     foreach ($tags as $tag) {
... line 66
67     }
68     });
... lines 69 - 70
71 }
... lines 72 - 78
79 }
```

To help PhpStorm, I'll use some inline PHPDoc to tell it that \$tags is an array of Tag objects.

Inside the loop, just say \$tag->getName():

```
// ...
$tags = $this->getRandomReferences(Tag::class, $this->faker->numberBetween(0, 5));
foreach ($tags as $tag) {
    $tag->getName();
    dump($tag);
}
die;
// ...
```

I know, that looks weird: we're calling a method but not using it! But, calling this method is enough to make Doctrine query for the tag's real data. Below, dump(\$tag) and die after the loop.

Load the fixtures again:

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

Boom! We have data!

Anyways, this is the proxy system in action. You need to know what it is because you *will* see it from time-to-time. But mostly, it should be completely invisible: don't think about it.

Adding Tags to Article

Finally, how can we add each Tag to the Article? No surprise, it's \$article->addTag():

```

80 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 10
11 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
12 {
... lines 13 - 29
30 public function loadData(ObjectManager $manager)
31 {
32     $this->createMany(Article::class, 10, function(Article $article, $count) use ($manager) {
... lines 33 - 63
64         $tags = $this->getRandomReferences(Tag::class, $this->faker->numberBetween(0, 5));
65         foreach ($tags as $tag) {
66             $article->addTag($tag);
67         }
68     });
... lines 69 - 70
71 }
... lines 72 - 78
79 }

```

Try the fixtures again:

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

Ok, no errors. To the database!

```
$ php bin/console doctrine:query:sql 'SELECT * FROM tag'
```

Yep, 10 tags with various, weird names. Let's see what the join table looks like:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article_tag'
```

Yea! 24 rows! *Each* time we add a Tag to Article and save, Doctrine *inserts* a row in this table. If we were to *remove* an existing Tag from an Article object - with `$article->removeTag()` - and then flush, Doctrine would actually *delete* that row. For the first, and only, time, we have a table that we don't need to think about, at *all*: Doctrine inserts and deletes data for us.

All we need to do is worry about relating Article objects to Tag objects. Doctrine handles the saving.

Rendering the Tags

And *now* we can turn back to building our site: open the `article/show.html.twig` template. On this page, let's print the tags right under the article title. So, scroll down a bit. Copy the span for the heart count and paste it below.

Because the Article object holds an array of tags, use `for tag` in `article.tags`:

86 lines | [templates/article/show.html.twig](#)

... lines 1 - 4

```
5 {% block content_body %}
6     <div class="row">
7         <div class="col-sm-12">
8             
9             <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 10 - 19
20         <span class="pl-2 article-details">
21             {% for tag in article.tags %}
... line 22
23             {% endfor %}
24         </span>
25     </div>
26 </div>
27 </div>
... lines 28 - 78
79 {% endblock %}
... lines 80 - 86
```

Inside, let's create a cute little badge and print the tag name: `{{ tag.name }}`:

86 lines | [templates/article/show.html.twig](#)

... lines 1 - 4

```
5 {% block content_body %}
6     <div class="row">
7         <div class="col-sm-12">
8             
9             <div class="show-article-title-container d-inline-block pl-3 align-middle">
... lines 10 - 19
20         <span class="pl-2 article-details">
21             {% for tag in article.tags %}
22                 <span class="badge badge-secondary">{{ tag.name }}</span>
23             {% endfor %}
24         </span>
25     </div>
26 </div>
27 </div>
... lines 28 - 78
79 {% endblock %}
... lines 80 - 86
```

Super cool! Try it: find the page and refresh. We got it! Well, this Article only has one tag - boring. Find a different one: boom! Four different tags.

Repeat this on the homepage: we'll list the tags right under the title. Copy the for loop, then open homepage.html.twig. Down below, add a `
`, then paste! Wrap this in a `<small>` tag and change the class to badge-light:


```

65 lines | templates/article/homepage.html.twig
... lines 1 - 2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6
7              <!-- Article List -->
8
9              <div class="col-sm-12 col-md-8">
... lines 10 - 18
19          <!-- Supporting Articles -->
20
21          {% for article in articles %}
22              <div class="article-container my-1">
23                  <a href="{{ path('article_show', {slug: article.slug}) }}">
24                      
25                      <div class="article-title d-inline-block pl-3 align-middle">
... lines 26 - 27
28                          <br>
29                          {% for tag in article.tags %}
30                              <small>
31                                  <span class="badge badge-light">{{ tag.name }}</span>
32                              </small>
33                          {% endfor %}
... lines 34 - 36
37                      </div>
38                  </a>
39              </div>
40          {% endfor %}
41      </div>
... lines 42 - 61
62  </div>
63  </div>
64  {% endblock %}

```

This is just the same thing again: we have an article variable, which allows us to easily loop over its tags.

But notice, *before* we refresh, there are 8 queries. But now... there are 15! The page works, but we have another N+1 query problem. And, it's probably no big deal, but let's learn how to add a JOIN to a ManyToMany query so that we can fix it.

Chapter 19: ManyToMany Joins & When to Avoid ManyToMany

We have the N+1 query problem once again. Click to view those queries. The new queries are mixed in here, but you'll see 7 new queries that select from tag with an INNER JOIN so that it can find all the tags for just *one* Article. Each time we reference the tags for a new Article, it makes a new query for *that* article's tags.

This is quite possibly *not* something you need to worry about, at least, not until you can see a *real* performance issue on production. But, we *should* be able to fix it. The *first* query on this page finds all of the published articles. Could we add a join to that query to select the tag data all at once?

Totally! Open ArticleController and find the homepage() action. Right now, we're using `$articles = $repository->findAllPublishedOrderedByNewest();`:

```
65 lines | src/Controller/ArticleController.php
... lines 1 - 13
14 class ArticleController extends AbstractController
15 {
... lines 16 - 28
29 public function homepage(ArticleRepository $repository)
30 {
31     $articles = $repository->findAllPublishedOrderedByNewest();
... lines 32 - 35
36 }
... lines 37 - 63
64 }
```

Open ArticleRepository to check that out:

```
58 lines | src/Repository/ArticleRepository.php
... lines 1 - 15
16 class ArticleRepository extends ServiceEntityRepository
17 {
... lines 18 - 22
23 /**
24  * @return Article[]
25  */
26 public function findAllPublishedOrderedByNewest()
27 {
28     return $this->addIsPublishedQueryBuilder()
29         ->orderBy('a.publishedAt', 'DESC')
30         ->getQuery()
31         ->getResult()
32     ;
33 }
... lines 34 - 56
57 }
```

This custom query finds the Article objects, but does *not* do any special joins. Let's add one. But.... wait. This is weird. If you think about the database, we're going to need to join *twice*. We first need a LEFT JOIN from article to article_tag. Then, we need a another JOIN from article_tag to tag so that we can select the tag's data.

This is where Doctrine's ManyToMany relationship *really* shines. Don't think *at all* about the join table. Instead, `->leftJoin()` on

a.tags and use t as the new alias:

```
60 lines | src/Repository/ArticleRepository.php
... lines 1 - 15
16 class ArticleRepository extends ServiceEntityRepository
17 {
... lines 18 - 22
23 /**
24  * @return Article[]
25  */
26 public function findAllPublishedOrderedByNewest()
27 {
28     return $this->addIsPublishedQueryBuilder()
29         ->leftJoin('a.tags', 't')
... lines 30 - 33
34 ;
35 }
... lines 36 - 58
59 }
```

The a.tags refers to the tags property on Article. And because Doctrine knows that this is a ManyToMany relationship, it knows how to join *all* the way over to tag. To actually fetch the tag data, use ->addSelect('t'):

```
60 lines | src/Repository/ArticleRepository.php
... lines 1 - 15
16 class ArticleRepository extends ServiceEntityRepository
17 {
... lines 18 - 22
23 /**
24  * @return Article[]
25  */
26 public function findAllPublishedOrderedByNewest()
27 {
28     return $this->addIsPublishedQueryBuilder()
29         ->leftJoin('a.tags', 't')
30         ->addSelect('t')
... lines 31 - 33
34 ;
35 }
... lines 36 - 58
59 }
```

That is it. Go back to our browser. The 15 queries are... back down to 8! Open the profiler to check them out. Awesome! The query selects everything from article *and* all the fields from tag. It can do that because it has *both* joins! That's nuts!

When a ManyToMany Relationship is Not What You Need

Ok guys, there is *one* last thing we need to talk about, and, it's a warning about ManyToMany relations.

What if we wanted to start saving the *date* of when an Article was given a Tag. Well, crap! We can't do that. We could record the date that a Tag was created or the date an Article was created, but we *can't* record the date when an Article was linked to a Tag. In fact, we can't save *any* extra data about this relationship.

Why? Because that data would need to live on this article_tag table. For example, we might want a third column called created_at. The problem is, when you use a ManyToMany relationship, you *cannot* add any more columns to the join table. It's just not possible.

This means that if, in the future, you *do* need to save extra data about the relationship, well, you're in trouble.

So, here's my advice: before you set up a ManyToMany relationship, you need to think hard and ask yourself a question:

Will I ever need to store additional metadata about this relationship?

If the answer is yes, if there's even *one* extra piece of data that you want to store, then you should *not* use a ManyToMany relationship. In fact, you can't use Doctrine at all, and you need to buy a new computer.

I'm *kidding*. If you need to store extra data on the article_tag table, then, instead, create a new ArticleTag entity for that table! That ArticleTag entity would have a ManyToOne relationship to Article and a ManyToOne relationship to Tag. This would effectively give you the *exact* same structure in the database. But *now*, thanks to the new ArticleTag entity, you're free to add whatever other fields you want.

If you generated a ManyToMany relationship by mistake and want to switch, it's not the end of the world. You *can* still create the new entity class and generate a migration so that you don't lose your existing data. But, if you can configure things in the beginning... well, even better.

Ok guys, you are now Doctrine pros! Your relationship skills strike fear at the heart of your enemies, and your ability to JOIN across tables is legendary among your co-workers.

Yes, there is more to learn, like how to write even more complex queries, and there are a lot of other, cool features - like Doctrine inheritance. But, all of the *super* important stuff that you need to create a real site? Yea, you got it *down*. So go out there, SELECT * FROM world, and build something amazing with Doctrine.

Alright guys, seeya next time.

