Mastering Doctrine Relations

With <3 from SymfonyCasts

Chapter 1: The Answer Entity

Oh hey there friends! Welcome back to part 2 of our Doctrine in Symfony series... you wonderful database nerds you.

Last time we mastered the basics, but good stuff! Creating an entity, migrations, fixtures, saving, querying and making the perfect omelette... I think. *This* time, we're going to do some *mega* study on Doctrine *relations*.

Project Setup

So let's get our project rocking. To avoid foreign key constraints in your brain while watching the tutorial, I recommend downloading the course code from this page and coding along with me. After unzipping the file, you'll find a `start/` directory with all the fancy files that you see here. Check out the `README.md` file for all the fun details on how to get this project running.

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

```
$ symfony serve -d
```

I'm using the Symfony binary to start a local web server. Let's go see our site. Spin over to your browser and head to https://127.0.0.1:8000.

Oh, hey there Cauldron Overflow! This is a site where the budding industry of witches and wizards can come to ask questions... after - sometimes - prematurely shipping their spells to production... and turning their clients into small adorable frogs. It could be worse.

The questions on the homepage *are* coming from the database... we rock! We built a `Question` entity in the first tutorial. But if you click *into* a question... yea. These answers? These are *totally* hard-coded. Time to change that.

Making the Answer Entity

I want you to, for now, forget about any potential relationship between questions and answers. It's really simple: our site has answers! And so, if we want to *store* those answers in the database, we need an `Answer` entity.

At your terminal, let's generate one. Run:

```
$ symfony console make:entity
```

Now, as a reminder, `symfony console` is just a fancy way of saying `php bin/console`. *I'm* using the Docker & Symfony web server integration. That's where the Symfony web server reads your `docker-compose.yaml` file and exposes environment variables to the services inside of it. We talked about that in the first Symfony 5 tutorial. By using `symfony console` - instead of running `bin/console` directly - my commands will be able to talk to my Docker services... which for me is just a database. That's not needed for *this* command, but it will be for others.

Anyways, run this and create a new entity called `Answer`. Let's give this a few basic properties like `content` which will store the answer itself. Set this to a `text` type: the `string` type maxes out at 255 characters. Say "no" to nullable: that will make this column required in the database.

Let's also add a `username` property, which will be a string. Eventually, in the security tutorial, we'll change this to be a relationship to a `User` entity. Use the 255 length and make it not nullable.

Oh, and one more: a `votes` property that's an `integer` so that people can up vote and down vote this answer. Make this not nullable and... done! Hit enter one more time to finish.

76 lines | src/Entity/Answer.php

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

use App\Repository\AnswerRepository;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass=AnswerRepository::class)
 */
class Answer
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

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

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

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

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

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

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

        return $this;
    }

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

    public function setUsername(string $username): self
    {
        $this->username = $username;

        return $this;
    }

    public function getVotes(): ?int
    {
        return $this->votes;
    }

    public function setVotes(int $votes): self
    {
        $this->votes = $votes;

        return $this;
    }
}
```

Timestampable and Default votes Value

Before we generate the migration, go open up that class: `src/Entity/Answer.php` . So far... there's nothing special here! It looks pretty much like our other entity. Oh, but if you're using PHP 8, then the command *may* have generated PHP 8 attributes instead of annotations. That's great! They work exactly the same and you should use attributes if you can.

At the top of the class, add `use TimestampableEntity` . We talked about that in the last tutorial: it adds nice `createdAt` and `updatedAt` properties that will be set automatically.

79 lines | src/Entity/Answer.php

```
... lines 1 - 6
use Gedmo\Timestampable\Traits\TimestampableEntity;
... lines 8 - 11
class Answer
{
    use TimestampableEntity;
... lines 15 - 77
}
```

Oh, and one other thing: default the votes to zero. I made this column *not* nullable in the database. Thanks to this `= 0` , if we do *not* set the votes on a new answer, instead of getting a database error about `null` not being allowed, the `Answer` will save with `votes = 0` .

79 lines | src/Entity/Answer.php

```
... lines 1 - 11
class Answer
{
... lines 14 - 32
    /**
     * @ORM\Column(type="integer")
     */
    private $votes = 0;
... lines 37 - 77
}
```

Making the Migration

Now let's generate the migration. Find your terminal and run:

```
$ symfony console make:migration
```

As a reminder, this command is smart: it looks at all of your entities *and* your *actual* database structure, and generates the SQL needed to make them match. Go check out that new file... it's in the `migrations/` directory. And... perfect! `CREATE TABLE answer` ... and then it adds all of the columns.

32 lines | migrations/Version20210902130926.php

```
... lines 1 - 2
declare(strict_types=1);

namespace DoctrineMigrations;

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

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20210902130926 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE TABLE answer (id INT AUTO_INCREMENT NOT NULL, content LONGTEXT NOT NULL, username
    }

    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('DROP TABLE answer');
    }
}
```

Run the migration with:

```
$ symfony console doctrine:migrations:migrate
```

All good! Our database now has a `question` table *and* an `answer` table. Next, let's relate them.

Chapter 2: The ManyToOne Relation

Okay: we have a `Question` entity and table. We have an `Answer` entity and table. Yay for us! But what we *really* want to do is *relate* an `Answer` to a `Question`.

To do this... well... *forget* about Doctrine for a second. Let's just think about how this would look in a database. So: each answer belongs to a single question. We would normally model this in the database by adding a `question_id` column to the `answer` table that's a foreign key to the question's `id`. This would allow each question to have many answers and each answer to be related to exactly one question.

Ok! So... we need to add a new column to the `answer` table. The way we've done that so far in Doctrine is by adding a *property* to the entity class. And adding a relationship is *no* different.

Generating the Answer.question ManyToOne Property

So find your terminal and run:

```
$ symfony console make:entity
```

We need to update the `Answer` entity. Now, what should the new property be called... `question_id`? Actually, no. And this is one of the coolest, but trickiest things about Doctrine. Instead, call it simply `question`... because this property will hold an entire `Question` *object*... but more on that later.

For the type, use a "fake" type called `relation`. This starts a wizard that will guide us through the process of adding a relationship. What class should this new property relate to? Easy: the `Question` entity.

Ah, and *now* we see something awesome: a big table that explains the four types of relationships with an example of each of one. You can read through all of these, but the one *we* need is `ManyToOne`. Each `Answer` relates to one `Question`. And each `Question` can have many answers. That's... exactly what we want. Enter `ManyToOne`. This is actually the *king* of relationships: most of the time, this will be the one you want.

Is the `Answer.question` property allowed to be null? This is asking if we should be allowed to save an `Answer` to the database that is *not* related to a `Question`. For us, that's a "no". Every `Answer` *must* have a question... except... I guess... in the Hitchhiker's Guide to the Galaxy. Anyways, saying "no" will make the new column *required* in the database.

Mapping the "Other" Side of the Relation

This next question is *super* interesting:

> Do you want to add a new property to `Question` so you can access/update `Answer` objects from it.

Here's the deal: every relationship can have two sides. Think about it: an `Answer` is related to one `Question`. But... you can also view the relationship from the other direction and say that a `Question` has many answers.

Regardless of whether we say "yes" or "no" to this question, we *will* be able to get and set the `Question` for an `Answer`. If we *do* say "yes", it simply means that we will *also* be able to access the relationship from the *other* direction... like by saying `$question->getAnswers()` to get all the answers for a given `Question`.

And... hey! Being able to say `$question->getAnswers()` sounds pretty handy! So let's say yes. There's no downside... except that this will generate a little bit more code.

What should that new property in the `Question` entity be called? Use the default `answers`.

Finally it asks a question about `orphanRemoval`. This is a bit more advanced... and you probably don't need it. If you *do* discover later that you need it, you can enable it manually inside your entity. I'll say no.

And... done! Hit enter one more time to exit the wizard.

Checking out the Entity Changes

Let's go see what this did! I committed before recording, so I'll run

```
$ git status
```

to check things out. Ooo, *both* entities were updated. Let's open `Answer` first... and... here's the new `question` property. It *looks* like any other property except that instead of having `ORM\Column` above it, it has `ORM\ManyToOne` and targets the `Question` entity.

```
97 lines | src/Entity/Answer.php
... lines 1 - 11
class Answer
{
... lines 14 - 32
    /**
     * @ORM\Column(type="integer")
     */
    private $votes = 0;

    /**
     * @ORM\ManyToOne(targetEntity=Question::class, inversedBy="answers")
     * @ORM\JoinColumn(nullable=false)
     */
    private $question;
... lines 43 - 95
}
```

Scroll to the bottom. Down here, it generated a normal getter and setter method.

```
97 lines | src/Entity/Answer.php
... lines 1 - 11
class Answer
{
... lines 14 - 84
    public function getQuestion(): ?Question
    {
        return $this->question;
    }

    public function setQuestion(?Question $question): self
    {
        $this->question = $question;

        return $this;
    }
}
```

Let's go look at the `Question` entity. If we scroll... beautiful: this now has an `answers` property, which is a `OneToMany` relationship.

178 lines | src/Entity/Question.php

```
... lines 1 - 14
class Question
{
... lines 17 - 51
    /**
     * @ORM\OneToMany(targetEntity=Answer::class, mappedBy="question")
     */
    private $answers;
... lines 56 - 176
}
```

And... all the way at the bottom, it generated a getter and setter method. Oh, well, instead of `setAnswers()`, it generated `addAnswer()` and `removeAnswer()`, which are just a bit more convenient, especially in Symfony if you're using the form component or the serializer.

178 lines | src/Entity/Question.php

```
... lines 1 - 6
use Doctrine\Common\Collections\Collection;
... lines 8 - 14
class Question
{
... lines 17 - 147
    /**
     * @return Collection|Answer[]
     */
    public function getAnswers(): Collection
    {
        return $this->answers;
    }

    public function addAnswer(Answer $answer): self
    {
        if (!$this->answers->contains($answer)) {
            $this->answers[] = $answer;
            $answer->setQuestion($this);
        }

        return $this;
    }

    public function removeAnswer(Answer $answer): self
    {
        if ($this->answers->removeElement($answer)) {
            // set the owning side to null (unless already changed)
            if ($answer->getQuestion() === $this) {
                $answer->setQuestion(null);
            }
        }

        return $this;
    }
}
```

The ArrayCollection Object

Head back up near the top of this class. The command *also* generated a constructor method so that it could initialize the `answers` property to some `ArrayCollection` object.

```
178 lines | src/Entity/Question.php

... lines 1 - 5
use Doctrine\Common\Collections\ArrayCollection;
... lines 7 - 14
class Question
{
... lines 17 - 56
    public function __construct()
    {
        $this->answers = new ArrayCollection();
    }
... lines 61 - 176
}
```

Ok, so we know that each `Question` will have many answers. So we know that the `answers` property will be an array... or some sort of collection. In Doctrine... for internal reasons, instead of setting the `answers` property to an array, it sets it to a `Collection` object. That's... not *too* important: the object looks an acts like an array - like, you can `foreach` over it. But it *does* have a few extra useful methods on it.

Anyways, whenever you have a relationship that holds a "collection" of other items, you need to initialize that property to an `ArrayCollection` in your constructor. If you use the `make:entity` command, this will always be done for you.

ManyToOne vs OneToMany

Oh, and I want to point something out. We generated a `ManyToOne` relationship. We can see this in the `Answer` entity. But... in the `Question` entity, it says `OneToMany`.

This is a *key* thing to understand: a `ManyToOne` relationship and a `OneToMany` relationship are *not* actually two different types of relationships. Nope: they described the *same* relationship... just from the two different sides.

Think about it: from the perspective of a `Question`, we have a "one question relates to many answers" relationship - a `OneToMany`. From the perspective of the `Answer` entity, that *same* relationship would be described as "many answers can relate to one question": a `ManyToOne`.

The point is: when you see these two relationships, realize that they are *not* two different things: they're the same *one* relation seen from opposite sides.

The answer_id Foreign Key Column

Anyways, we ran `make:entity` and it added one property to each class and a few methods. Nothing fancy. Time to generate the migration for this:

```
$ symfony console make:migration
```

Let's go peek at the new file! How cool is this???

36 lines | migrations/Version20210902132832.php

```
... lines 1 - 2
declare(strict_types=1);

namespace DoctrineMigrations;

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

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20210902132832 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE answer ADD question_id INT NOT NULL');
        $this->addSql('ALTER TABLE answer ADD CONSTRAINT FK_9474526C1E27F6BF FOREIGN KEY (question_id) REFERENC
        $this->addSql('CREATE INDEX IDX_9474526C1E27F6BF ON answer (question_id)');
    }

    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE answer DROP FOREIGN KEY FK_9474526C1E27F6BF');
        $this->addSql('DROP INDEX IDX_9474526C1E27F6BF ON answer');
        $this->addSql('ALTER TABLE answer DROP question_id');
    }
}
```

It's adding a `question_id` column to the `answer` table! Doctrine is smart: we added a `question` property to the `Answer` entity. But in the database, it added a `question_id` column that's a foreign key to the `id` column in the `question` table. In other words, the table structure looks *exactly* like we expected!

The tricky, but honestly *awesome* thing, is that, in PHP, to relate an `Answer` to a `Question`, we're *not* going to set the `Answer.question` property to an integer `id`. Nope, we're going to set it to an entire `Question` *object*. Let's see exactly how to do that next.

Chapter 3: Saving Relations

Our `answer` table has a new `question_id` column. Cool... but how do we *populate* that column? How do we relate an `Answer` to a `Question`? This is actually pretty easy... but it might feel weird if you're used to working with databases *directly*.

Open up `src/DataFixtures/AppFixtures.php`. We're using Foundry to add rich fixtures, or fake data, into our project.

```
 25 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 2
namespace App\DataFixtures;

use App\Entity\Question;
use App\Factory\QuestionFactory;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        QuestionFactory::createMany(20);

        QuestionFactory::new()
            ->unpublished()
            ->many(5)
            ->create()
        ;

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

But to see how relationships work, let's do some good ol' fashioned manual coding.

Creating some Dummy Question and Answer Objects

Start by creating a new `Answer` object... and populate it with enough data to get it to save. Repeat this to create a new `Question` object... and *also* give that some data.

```
 37 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 10
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 15 - 22
        $answer = new Answer();
        $answer->setContent('This question is the best? I wish... I knew the answer.');
        $answer->setUsername('weaverryan');

        $question = new Question();
        $question->setName('How to un-disappear your wallet.');
        $question->setQuestion('... I should not have done this...');
... lines 30 - 34
    }
}
```

Save these boring objects to the database by calling `$manager->persist()` on both of them.

```
 37 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 10
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 15 - 22
        $answer = new Answer();
        $answer->setContent('This question is the best? I wish... I knew the answer.');
        $answer->setUsername('weaverryan');

        $question = new Question();
        $question->setName('How to un-disappear your wallet.');
        $question->setQuestion('... I should not have done this...');

        $manager->persist($answer);
        $manager->persist($question);
... lines 33 - 34
    }
}
```

Cool. If we stop now, these objects won't be related... and the `Answer` won't even save! Try it:

```
$ symfony console doctrine:fixtures:load
```

And... woh! My bad! We generated a migration in the last chapter, and then I totally forgot to run it! Time to do that:

```
$ symfony console doctrine:migrations:migrate
```

Now try the fixtures:

```
$ symfony console doctrine:fixtures:load
```

JoinColumn

That's the error I expected:

> `question_id` cannot be null on the `answer` table

That's because we made `question_id` required: it was one of the questions that `make:entity` command asked us. Oh, and I can show you where this is configured. Open up the `Answer` class and find the `question` property. It's this `JoinColumn(nullable=false)`: that makes the `question_id` column required.

```
37 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 10
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 15 - 22
        $answer = new Answer();
        $answer->setContent('This question is the best? I wish... I knew the answer.');
        $answer->setUsername('weaverryan');

        $question = new Question();
        $question->setName('How to un-disappear your wallet.');
        $question->setQuestion('... I should not have done this...');

        $manager->persist($answer);
        $manager->persist($question);
... lines 33 - 34
    }
}
```

Relating an Answer to a Question

Anyways, the thing we want to know is: how can I relate this `Answer` to this `Question`? How do we say that the `Answer` belongs to the `Question`? It's as simple as `$answer->setQuestion($question)`.

```
39 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 10
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 15 - 28
        $question->setQuestion('... I should not have done this...');

        $answer->setQuestion($question);

        $manager->persist($answer);
... lines 34 - 36
    }
}
```

Notice that we do *not* say `$question->getId()`. We're not passing the *ID* to the `question` property, we're setting the entire `Question` *object* onto the property. Doctrine will be smart enough to save these in the correct order: it'll save the `question` first, grab its new id, and use that to save the `Answer`.

To prove it, reload the fixtures:

```
$ symfony console doctrine:fixtures:load
```

Ok, no errors. Let's see what the database looks like. We can use the `doctrine:query:sql` command as an easy way to do this: `SELECT * FROM answer`.

```
$ symfony console doctrine:query:sql 'SELECT * FROM answer'
```

Yes! We have `one` answer in the database and its `question_id` is set to 103. Let's query for that question:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question WHERE id = 103'
```

And... there it is!

The big takeaway here is this: in PHP, we *just* think about objects. We think:

> Hey! I'd really like to relate this `Answer` object to this `Question` object.

Then, when we save these, *Doctrine* handles all the nitty gritty details of figuring out how to save that *for* us. The database is almost an implementation detail that we don't need to think about much.

Next: now that we've seen how to relate objects, let's update our fixtures to use Foundry. That will let us create a *ton* of fake questions and answers and relate them with very little code.

Chapter 4: Relations in Foundry

We're using a library called Foundry to help us generate rich fixtures data. Right now, it's creating 25 questions. Let's use Foundry to *also* add some answers.

make:factory Answer

Start by generating the factory class. At your terminal, run:

```
$ symfony console make:factory
```

Yup: we want to generate a factory for the `Answer` entity. Beautiful! Let's go check that out: `src/Factory/AnswerFactory.php` .

62 lines | src/Factory/AnswerFactory.php

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

use App\Entity\Answer;
use App\Repository\AnswerRepository;
use Zenstruck\Foundry\RepositoryProxy;
use Zenstruck\Foundry\ModelFactory;
use Zenstruck\Foundry\Proxy;

/**
 * @extends ModelFactory<Answer>
 *
... lines 14 - 27
 */
final class AnswerFactory extends ModelFactory
{
    public function __construct()
    {
        parent::__construct();

        // TODO inject services if required (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#factori
    }

    protected function getDefaults(): array
    {
        return [
            // TODO add your default values here (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#r
            'content' => self::faker()->text(),
            'username' => self::faker()->text(),
            'createdAt' => null, // TODO add DATETIME ORM type manually
            'updatedAt' => null, // TODO add DATETIME ORM type manually
        ];
    }

    protected function initialize(): self
    {
        // see https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#initialization
        return $this
            // ->afterInstantiate(function(Answer $answer) {})
        ;
    }

    protected static function getClass(): string
    {
        return Answer::class;
    }
}
```

Cool. The only work we need to do immediately is inside `getDefaults()`. The goal here is to give every *required* property a default value... and we even have Faker available here to help us generate some random stuff.

Let's see: for `username`, we can use a `userName()` faker method. And for votes, instead of a random number, use `numberBetween` -20 and 50. I'll delete `updatedAt`... but keep `createdAt` so we can fake answers with a `dateTimeBetween()` `-1 year` and now, which is the default 2nd argument. That period is a typo for future me to discover!

61 lines | src/Factory/AnswerFactory.php

```
// ... lines 1 - 28
final class AnswerFactory extends ModelFactory
{
// ... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
            'content' => self::faker()->text(),
            'username' => self::faker()->userName(),
            'createdAt' => self::faker()->dateTimeBetween('-1 year'),
            'votes' => rand(-20, 50),
        ];
    }
// ... lines 47 - 59
}
```

Head back to `AppFixtures`. Let's remove *all* of this manual `Answer` and `Question` code. Replace it with `AnswerFactory::createMany(100)` to create 100 answers.

29 lines | src/DataFixtures/AppFixtures.php

```
// ... lines 1 - 6
use App\Factory\AnswerFactory;
// ... lines 8 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        QuestionFactory::createMany(20);

        QuestionFactory::new()
            ->unpublished()
            ->many(5)
            ->create()
        ;

        AnswerFactory::createMany(100);

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

Populating the Answer.question Property

Over in `AnswerFactory` ... let's fix that typo. Notice that, in `getDefaults()`, we are *not* setting the `question` property. And so, if you spin over to your terminal and run:

```
$ symfony console doctrine:fixtures:load
```

... we get our favorite error: `question_id` column cannot be null.

To fix this, in `AppFixtures`, pass a 2nd argument to `createMany()`: an array with a `question` key set to `QuestionFactory::random()`, which is a *really* cool method.

```
 31 lines | src/DataFixtures/AppFixtures.php
 ... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
 ... lines 16 - 23
        AnswerFactory::createMany(100, [
            'question' => QuestionFactory::random(),
        ]);
 ... lines 27 - 28
    }
}
```

With this setup, when we call `createMany()`, Foundry will first call `getDefaults()`, grab that array, add `question` to it, and then will ultimately try to create the `Answer` using *all* of those values.

The `QuestionFactory::random()` method does what it sounds like: it grabs a random `Question` from the database. So yes, it *is* now important that we create the questions first and then the answers after.

Let's try this:

```
$ symfony console doctrine:fixtures:load
```

Ok... no errors. Check out the database:

```
$ symfony console doctrine:query:sql 'SELECT * FROM answer'
```

Passing a Callback to Randomize Every Answer's Data

And... sweet! We have 100 answers filled with a lot of nice random data from Faker. But... if you look closely, we have a teensy problem. This answer has `question_id` 140... and so does this one... and this one! In fact, *all* 100 answers are related to the *same* `Question`. Whoops!

Why? Because the `QuestionFactory::random()` method is called just *once*. It *did* fetch a random `Question`... and then used that same random question for all 100 answers.

If you want a different value *per* `Answer`, you need to pass a callback function to the second argument instead of an array. That function will then *return* the array of data to use. Foundry will execute the callback once for *each* `Answer`: so 100 times in total.

```
 33 lines | src/DataFixtures/AppFixtures.php
 ... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
 ... lines 16 - 23
        AnswerFactory::createMany(100, function() {
            return [
                'question' => QuestionFactory::random(),
            ];
        });
 ... lines 29 - 30
    }
}
```

Try it again: reload the fixtures:

```
$ symfony console doctrine:fixtures:load
```

Then query the `answer` table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM answer'
```

Much better! 100 answers where *each* is related to a random question.

Moving the "question" into getDefaults()

But to make life easier, we can move this `question` value directly into `AnswerFactory`. Copy the `question` line.. and then change the fixtures code back to the very simple `AnswerFactory::createMany(100)`.

```
33 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 16 - 23
        AnswerFactory::createMany(100, function() {
            return [
                'question' => QuestionFactory::random(),
            ];
        });
... lines 29 - 30
    }
}
```

Now in `AnswerFactory`, paste `question` set to `QuestionFactory::random()`. This works because the `getDefaults()` method is called 100 times, once for *each* answer.

```
62 lines | src/Factory/AnswerFactory.php
... lines 1 - 28
final class AnswerFactory extends ModelFactory
{
... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
... lines 41 - 44
                'question' => QuestionFactory::random(),
        ];
    }
... lines 48 - 60
}
```

Next: let's discover a key rule when using Foundry and relationships. A rule that, if you forget to follow it, might result in a *bunch* of random extra records in your database.

Chapter 5: Foundry: Always Pass a Factory Instance to a Relation

I love Foundry. But using Foundry with Doctrine relationships is probably the *hardest* part of this library. So let's push a bit further. Pretend that, in this situation, we want to override the `question` value. Right now it grabs *any* random `Question` from the database. But I want to randomly grab only one of these 20 *published* questions.

Overriding the question Property

No problem! And this part is pretty manual. Put our callback... back... and return an array. There actually *is* a way in Foundry, to say:

> please give me a random `Question` *where* some field matches some value.

But... in our case, we would need to say `WHERE askedAt IS NOT NULL`... which is too complex for that system to handle. But no worries! We'll just do this manually.

Above, on the `createMany()` call, add a `$questions =` before this. Back down here, add a `use` to the callback so that the `$questions` variable is accessible... then leverage `array_rand()` to grab a random item.

```
33 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $questions = QuestionFactory::createMany(20);
... lines 17 - 23
        AnswerFactory::createMany(100, function() use ($questions) {
            return [
                'question' => $questions[array_rand($questions)]
            ];
        });
... lines 29 - 30
    }
}
```

Let's make sure this works! Reload the fixtures and...

```
$ symfony console doctrine:fixtures:load
```

No errors! We can use a special query to check this:

> `SELECT DISTINCT question_id FROM answer`

```
$ symfony console doctrine:query:sql 'SELECT DISTINCT question_id FROM answer'
```

Yes! The answers are related to exactly 20 questions.

Accidentally Creating Extra Relation Objects

That was... manual but simple enough. And it was a *great* setup to show you a *really* common mistake when using Foundry with relationships.

In `AnswerFactory`, let's change the default `question` to create a new unpublished question. We can do this by saying `QuestionFactory::new()` - to create a `QuestionFactory` object - then `->unpublished()`.

There's no magic here: `unpublished()` is a method we created in the first tutorial: it changes the `askedAt` value to `null`. Then, to actually *create* the `Question` from the factory, add `->create()`.

```
62 lines | src/Factory/AnswerFactory.php
... lines 1 - 28
final class AnswerFactory extends ModelFactory
{
... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
... lines 41 - 44
            'question' => QuestionFactory::new()->unpublished()->create(),
        ];
    }
... lines 48 - 60
}
```

This is *totally* legal: it will create a new unpublished `Question`, save it to the database and then that `Question` will be used as the `question` key when creating the `Answer`.

Well, that's what would *normally* happen. But since *we* are overriding the `question` key, this change should make absolutely *no* difference in our situation.

Famous last words. Reload the fixtures:

```
$ symfony console doctrine:fixtures:load
```

No errors... but check out how many questions there are in the database:

```
SELECT * from question
```

```
$ symfony console doctrine:query:sql 'SELECT * from question'
```

We *should* have 20+5: 25 questions. Instead... we have 125!

The problem is subtle... but maybe you spotted it! We're creating 100 answers... and the `getDefaults()` method is called for *every* one. That's.... good! But the moment that this `question` line is executed, it creates a new unpublished `Question` and saves it to the database. Then... a moment later, the `question` is overridden. This means that the 100 answers *were* all, in the end, correctly related to one of the 20 published questions. But it also means that, along the way, 100 extra questions were created, saved to the database... then never used.

What's the fix? Simple: remove `->create()`.

```
 62 lines | src/Factory/AnswerFactory.php
 ... lines 1 - 28
final class AnswerFactory extends ModelFactory
{
 ... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
 ... lines 41 - 44
            'question' => QuestionFactory::new()->unpublished(),
        ];
    }
 ... lines 48 - 60
}
```

This means that the `question` key is now set to a `QuestionFactory` object. The `new()` method returns a new `QuestionFactory` instance... and then the `unpublished()` method return `self`: so it returns that same `QuestionFactory` object.

Setting a relation property to a *factory* instance is totally allowed. In fact, you should *always* set a relation property to a factory instance if you can. Why?

Because this allows Foundry to *delay* creating the `Question` object until later. And in this case, it realizes that the `question` has been overridden, and so it *avoids* creating the extra object entirely... which is perfect.

Reload the fixtures one more time:

```
$ symfony console doctrine:fixtures:load
```

And check the `question` table:

```
$ symfony console doctrine:query:sql 'SELECT * from question'
```

We're back to 25 rows.

Next: let's use the new relationship to render answers on the frontend.

Chapter 6: Fetching Relations

Each *published* Question in the database will now be related to approximately 5 answers. Head to the homepage and click into a question. Time to replace this hardcoded craziness with *real*, dynamic answers.

Querying for Answers with findBy()

This means that we need to find all the answers for this specific Question . How can we do that? When we ran the make:entity command to create the Answer entity, it *also* generated an AnswerRepository class. And you might remember from the *last* tutorial that these repository classes have some nice, built-in methods for querying, like findBy() where we can find all the answers in the database that match some criteria, like WHERE votes = 5 *or* WHERE question_id = the id of some question.

51 lines | src/Repository/AnswerRepository.php

```php
// ... lines 1 - 2
namespace App\Repository;

use App\Entity\Answer;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

/**
 * @method Answer|null find($id, $lockMode = null, $lockVersion = null)
 * @method Answer|null findOneBy(array $criteria, array $orderBy = null)
 * @method Answer[]    findAll()
 * @method Answer[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class AnswerRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Answer::class);
    }
// ... lines 21 - 49
}
```

Open the controller for this page: src/Controller/QuestionController.php ... it's the show() action. Autowire the AnswerRepository service as an argument.

92 lines | src/Controller/QuestionController.php

```php
// ... lines 1 - 5
use App\Repository\AnswerRepository;
// ... lines 7 - 15
class QuestionController extends AbstractController
{
// ... lines 18 - 50
    public function show(Question $question, AnswerRepository $answerRepository)
    {
// ... lines 53 - 69
    }
// ... lines 71 - 90
}
```

Then, below, say $answers = $answerRepository->findBy() and pass this an array that should be used to build the WHERE statement in the query. To find all the answers WHERE the question_id matches this question, pass question set to the $question *object*. Remember: by this point, Doctrine has *already* used the slug in the URL to query for the Question object.

```
92 lines | src/Controller/QuestionController.php
... lines 1 - 15
class QuestionController extends AbstractController
{
... lines 18 - 50
    public function show(Question $question, AnswerRepository $answerRepository)
    {
... lines 53 - 56
        $answers = $answerRepository->findBy(['question' => $question]);
... lines 58 - 69
    }
... lines 71 - 90
}
```

The important thing here is that, when we call `findBy()`, we *don't* say `'question_id' => $question`... or `'question' => $question->getId()`. No! With Doctrine, we need to stop thinking about the database: we need to think only about the *objects*. We want to find all the `Answer` objects whose `question` property equals this `$question` *object*.

Behind the scenes, Doctrine will be smart enough to query `WHERE` the `question_id` column matches the `id` from this object.

Let's dump & die the `$answers` variable... and go see what it looks like. Refresh.

```
92 lines | src/Controller/QuestionController.php
... lines 1 - 15
class QuestionController extends AbstractController
{
... lines 18 - 50
    public function show(Question $question, AnswerRepository $answerRepository)
    {
... lines 53 - 56
        $answers = $answerRepository->findBy(['question' => $question]);
        dd($answers);
... lines 59 - 69
    }
... lines 71 - 90
}
```

Yes! This dumps an array of answers! Apparently this question is only related to *two* answers. Let's go pick a different one with more answers... cool! This question is related to *four* answers. That's, checks math, twice as interesting.

So... yay! Want to fetch all the `Answer` objects related to a `Question`? We just saw that you can do that by querying for the `Answer` entity and treating the `question` property like any *normal* property... except that you pass an entire `Question` *object* into the query.

Using the $question->getAnswers()

Now that we've done that... let's do something easier! Remove the `AnswerRepository` argument entirely... and instead say `$answers = $question->getAnswers()`. I'll put the `dd($answers)` back.

```
92 lines | src/Controller/QuestionController.php
... lines 1 - 15
class QuestionController extends AbstractController
{
... lines 18 - 50
    public function show(Question $question)
    {
... lines 53 - 56
        $answers = $question->getAnswers();
        dd($answers);
... lines 59 - 69
    }
... lines 71 - 90
}
```

When we ran the `make:entity` command, it asked us if we wanted to *also* add an `$answers` property to the `Question` class. We said yes, which generated some code that allows us to use this handy shortcut.

PersistentCollection & ArrayCollection

Over at the browser, when we refresh, we *should* see the same list of answers. And... we don't!? We get some `PersistentCollection` object. And, even stranger, I don't see the `Answer` objects anywhere *inside* of this collection. Dude, where's my answers?

Excellent question! Two important things here. First, remember that, inside the `Question` entity, the `$answers` property will *not* be a true *array* of `Answer` objects. Nope, it will be some sort of Doctrine collection object. It may be an `ArrayCollection` object *or* this `PersistentCollection` object... just depending on the situation. It doesn't *really* matter because both of these classes implement the same `Collection` interface... and both look and act like a normal array. The point is: that `PersistentCollection` is just an array-like wrapper around the answers... and not something we'll think about much.

Relations are Lazy-Loaded

The second thing to know is that when we query for a `Question`, Doctrine basically executes a `SELECT * FROM question` query. It grabs *all* the data from the `question` table and puts it onto the properties of the `Question` object. *But*, it does *not* immediately query the *answer* table for the related answers data. Nope, Doctrine doesn't query for the answers until - and unless - we actually *use* the `$answers` property. So, at this moment, it has *not* yet made the query for the answers data... which is why you don't see them inside this collection object. This feature is called "lazy loading".

Check this out: back in `QuestionController`, remove the `dd()`... and `foreach` over the `$answers` collection. Inside, do a normal `dump()` of the `$answer` variable.

```
94 lines | src/Controller/QuestionController.php
... lines 1 - 15
class QuestionController extends AbstractController
{
... lines 18 - 50
    public function show(Question $question)
    {
... lines 53 - 56
        $answers = $question->getAnswers();
        foreach ($answers as $answer) {
            dump($answer);
        }
... lines 61 - 71
    }
... lines 73 - 92
}
```

It's pretty crazy, but the moment that we `foreach` over the `$answers` collection - so the moment that we actually start *using* the answers data - Doctrine will query for that data.

We can see this! Refresh. Because we don't have a `die()` statement, the `dump()` shows up down in the web debug toolbar. And... yes! It found the same 4 answers!

Click the Doctrine icon on the toolbar to jump into its profiler... and look at the queries. There are two. First Doctrine queries for the `question` data. Then a moment later - at the moment the `foreach` line is executed - it queries `FROM answer` `WHERE question_id =` the `id` of this specific question. So, Doctrine *lazily* loads the answers data: it only makes the query once we *force* it to.

Anyways, we have answers! So next, let's pass these into the template, render their data, find an even *easier* way to do this *and* finally bring our answer-voting system to life by saving *real* vote totals to the database.

Chapter 7: Rendering Answer Data & Saving Votes

So let's render some answer data! Delete the old, hardcoded `$answers` and the `foreach`. Perfect: we're now passing this collection of `Answer` objects into the template:

```
85 lines | src/Controller/QuestionController.php

... lines 1 - 15
class QuestionController extends AbstractController
{
... lines 18 - 50
    public function show(Question $question)
    {
... lines 53 - 56
        $answers = $question->getAnswers();

        return $this->render('question/show.html.twig', [
            'question' => $question,
            'answers' => $answers,
        ]);
    }
... lines 64 - 83
}
```

Let's go open this template... because it'll probably need a few tweaks: `templates/question/show.html.twig`.

If you scroll down a bit - here it is - we loop over the `answers` variable. That *will* still work: the Doctrine collection *is* something that we can loop over. But the `answer` variable will now be an `Answer` *object*. So, to get the content, use `answer.content`:

```
96 lines | templates/question/show.html.twig

... lines 1 - 4
{% block body %}
... lines 6 - 54
    <ul class="list-unstyled">
        {% for answer in answers %}
            <li class="mb-4">
                <div class="row">
... lines 59 - 61
                    <div class="col-9">
                        {{ answer.content|parse_markdown }}
... line 64
                    </div>
... lines 66 - 89
                </div>
            </li>
        {% endfor %}
    </ul>
... line 94
{% endblock %}
```

We can also remove the hardcoded username and replace it with `answer.username`:

```
96 lines | templates/question/show.html.twig
... lines 1 - 4
{% block body %}
... lines 6 - 54
    <ul class="list-unstyled">
        {% for answer in answers %}
            <li class="mb-4">
                <div class="row">
... lines 59 - 61
                    <div class="col-9">
                        {{ answer.content|parse_markdown }}
                        <p>-- {{ answer.username }}</p>
                    </div>
... lines 66 - 89
                </div>
            </li>
        {% endfor %}
    </ul>
... line 94
{% endblock %}
```

And there's... one more spot. The vote count is hardcoded. Change that to `answer.votes`:

```
96 lines | templates/question/show.html.twig
... lines 1 - 4
{% block body %}
... lines 6 - 54
    <ul class="list-unstyled">
        {% for answer in answers %}
            <li class="mb-4">
                <div class="row">
... lines 59 - 61
                    <div class="col-9">
                        {{ answer.content|parse_markdown }}
                        <p>-- {{ answer.username }}</p>
                    </div>
                    <div class="col-2 text-end">
                        <div
... lines 68 - 73
                        >
... lines 75 - 86
                            <span><span {{ stimulus_target('answer-vote', 'voteTotal') }}>{{ answer.votes }}</span></span>
                        </div>
                    </div>
                </div>
            </li>
        {% endfor %}
    </ul>
... line 94
{% endblock %}
```

Ok! Let's see how it looks. Refresh and... alright! We have dynamic answers!

Fetching the Answers Directly in Twig

But... we're *still* doing too much work! Head back to the controller and completely remove the `$answers` variable:

```
 82 lines | src/Controller/QuestionController.php
 ... lines 1 - 15
class QuestionController extends AbstractController
{
 ... lines 18 - 50
    public function show(Question $question)
    {
        if ($this->isDebug) {
            $this->logger->info('We are in debug mode!');
        }

        return $this->render('question/show.html.twig', [
            'question' => $question,
        ]);
    }
 ... lines 61 - 80
}
```

Why are we doing this? Well, we know that we can say `$question->getAnswers()` to get all the answers for a question. And since we're passing a `$question` object into the template... we can call that method directly from Twig!

In `show.html.twig`, we don't have an `answers` variable anymore. That's ok because we can say `question.answers`:

```
 96 lines | templates/question/show.html.twig
 ... lines 1 - 4
{% block body %}
 ... lines 6 - 54
    <ul class="list-unstyled">
        {% for answer in question.answers %}
 ... lines 57 - 91
        {% endfor %}
    </ul>
 ... line 94
{% endblock %}
```

As reminder, when we say `question.answers`, Twig will first try to access the `$answers` property directly. But because it's private, it will *then* call the `getAnswers()` method. In other words, this is calling the *same* code that we were using a few minutes ago in our controller.

Back in the template, we need to update one more spot: the `answer|length` that renders the *number* of answers. Change this to `question.answers`:

```
 96 lines | templates/question/show.html.twig
 ... lines 1 - 4
{% block body %}
 ... lines 6 - 47
    <div class="d-flex justify-content-between my-4">
        <h2 class="">Answers <span style="font-size:1.2rem;">({{ question.answers|length }})</span></h2>
 ... line 50
    </div>
 ... lines 52 - 94
{% endblock %}
```

Refresh now and... we're still good! If you open the Doctrine profiler, we have the same 2 queries. But now this second query is literally being made from *inside* of the Twig template.

Saving Answer Votes

While we're here, in the first Symfony 5 tutorial, we wrote some JavaScript to support this answer voting feature. When we click, it... well... *sort of* works? It makes an Ajax call: we can see that down on the toolbar. But since there were no answers in the database when we built this, we... just "faked" it and returned a new random vote count from the Ajax call. *Now* we can make this actually work!

Before I recorded this tutorial, I refactored the JavaScript logic for this into Stimulus. If you want to check that out, it lives in `assets/controllers/answer-vote_controller.js`:

```
 32 lines | assets/controllers/answer-vote_controller.js
import { Controller } from 'stimulus';
import axios from 'axios';
... lines 3 - 12
export default class extends Controller {
    static targets = ['voteTotal'];
    static values = {
        url: String,
    }

    clickVote(event) {
        event.preventDefault();
        const button = event.currentTarget;

        axios.post(this.urlValue, {
            data: JSON.stringify({ direction: button.value })
        })
            .then((response) => {
                this.voteTotalTarget.innerHTML = response.data.votes;
            })
        ;
    }
}
```

The important thing for *us* is that, when we click the vote button, it makes an Ajax call to `src/Controller/AnswerController.php`: to the `answerVote` method. Inside, yup! We're grabbing a random number, doing nothing with it, and returning it.

To make the voting system *truly* work, start in `show.html.twig`. The way that our Stimulus JavaScript knows what URL to send the Ajax call to is via this `url` *variable* that we pass into that controller. It's generating a URL to the `answer_vote` route... which is the route above the target controller. Right now, for the `id` wildcard... we're passing in a hardcoded 10. Change that to `answer.id`:

96 lines | templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 54
    <ul class="list-unstyled">
        {% for answer in question.answers %}
            <li class="mb-4">
                <div class="row">
... lines 59 - 65
                    <div class="col-2 text-end">
                        <div
                            class="vote-arrows"
                            {{ stimulus_controller('answer-vote', {
                                url: path('answer_vote', {
                                    id: answer.id
                                })
                            }) }}
                        >
... lines 75 - 87
                        </div>
                    </div>
                </div>
            </li>
        {% endfor %}
    </ul>
... line 94
{% endblock %}
```

Back in the controller, we need to take this `id` and query for the `Answer` object. The *laziest* way to do that is by adding an `Answer $answer` argument. Doctrine will see that entity type-hint and automatically query for an `Answer` where `id` equals the `id` in the URL.

Remove this TODO stuff... and for the "up" direction, say `$answer->setVotes($answer->getVotes() + 1)`. Use the same thing for the down direction with *minus* one.

37 lines | src/Controller/AnswerController.php

```
... lines 1 - 11
class AnswerController extends AbstractController
{
... lines 14 - 16
    public function answerVote(Answer $answer, LoggerInterface $logger, Request $request, EntityManagerInterface $entity
    {
... lines 19 - 21
        // use real logic here to save this to the database
        if ($direction === 'up') {
            $logger->info('Voting up!');
            $answer->setVotes($answer->getVotes() + 1);
            $currentVoteCount = rand(7, 100);
        } else {
            $logger->info('Voting down!');
            $answer->setVotes($answer->getVotes() - 1);
        }
... lines 31 - 34
    }
}
```

If you want to create fancier methods inside `Answer` so that you can say things like `$answer->upVote()`, you *totally* should. We did that in the `Question` entity in the last tutorial.

At the bottom, return the *real* vote count: `$answer->getVotes()`. The only thing left to do *now* is save the new vote count to the database. To do that, we need the entity manager. Autowire that as a new argument -

EntityManagerInterface $entityManager - and, before the return, call $entityManager->flush().

 37 lines | src/Controller/AnswerController.php

```
... lines 1 - 11
class AnswerController extends AbstractController
{
... lines 14 - 16
    public function answerVote(Answer $answer, LoggerInterface $logger, Request $request, EntityManagerInterface $entity
    {
... lines 19 - 21
        // use real logic here to save this to the database
        if ($direction === 'up') {
            $logger->info('Voting up!');
            $answer->setVotes($answer->getVotes() + 1);
            $currentVoteCount = rand(7, 100);
        } else {
            $logger->info('Voting down!');
            $answer->setVotes($answer->getVotes() - 1);
        }

        $entityManager->flush();

        return $this->json(['votes' => $answer->getVotes()]);
    }
}
```

Ok team! Test drive time! Refresh. Everything still looks good so... let's vote! Yes! That made a successful Ajax call and the vote increased by 1. More importantly, when we refresh... the new vote count stays! It *did* save to the database!

Next: we've already learned that any *one* relationship can have *two* sides, like the Question is a OneToMany to Answer ... but also Answer is ManyToOne to Question. It turns out, in Doctrine, each side is given a special name *and* has an important distinction.

Chapter 8: Owning Vs Inverse Sides of a Relation

There's a, kind of, complex topic in Doctrine relations that we need to talk about. It's the "owning versus inverse side" of a relationship.

We already know that any relation can be seen from two different sides: `Question` is a `OneToMany` to `Answer`...

178 lines | src/Entity/Question.php

```
... lines 1 - 14
class Question
{
... lines 17 - 51
    /**
     * @ORM\OneToMany(targetEntity=Answer::class, mappedBy="question")
     */
    private $answers;
... lines 56 - 176
}
```

and that same relation can be seen as an `Answer` that is `ManyToOne` to `Question`.

97 lines | src/Entity/Answer.php

```
... lines 1 - 11
class Answer
{
... lines 14 - 37
    /**
     * @ORM\ManyToOne(targetEntity=Question::class, inversedBy="answers")
     * @ORM\JoinColumn(nullable=false)
     */
    private $question;
... lines 43 - 95
}
```

So... what's the big deal? We already know that we can *read* data from both sides: we can say `$answer->getQuestion()` and we can also say `$question->getAnswers()`.

Setting the Other Side of the Relation

But can you *set* data from both sides? In `AnswerFactory`, when we originally started playing with this relationship, we proved that you can say `$answer->setQuestion()` and Doctrine *does* correctly save that to the database.

Now let's try the *other* direction. I'm going to paste in some plain PHP code to play with. This uses the `QuestionFactory` to create one `Question` - I'm using it because I'm kinda lazy - and then creates two `Answer` objects by hand and persists them. We don't need to persist the `Question` because the `QuestionFactory` saves it entirely.

44 lines | src/DataFixtures/AppFixtures.php

```php
// ... lines 1 - 4
use App\Entity\Answer;
use App\Entity\Question;
// ... lines 7 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 16 - 29
        $question = QuestionFactory::createOne();
        $answer1 = new Answer();
        $answer1->setContent('answer 1');
        $answer1->setUsername('weaverryan');
        $answer2 = new Answer();
        $answer2->setContent('answer 1');
        $answer2->setUsername('weaverryan');

        $manager->persist($answer1);
        $manager->persist($answer2);
// ... lines 40 - 41
    }
}
```

At this point, the `Question` and these two answers are *not* related to each other. So, not surprisingly, if we run:

```
$ symfony console doctrine:fixtures:load
```

we get our favorite error: the `question_id` column cannot be null on the `answer` table. Cool! Let's relate them! But this time, instead of saying, `$answer1->setQuestion()`, do it with `$question->addAnswer($answer1)`... and `$question->addAnswer($answer2)`.

47 lines | src/DataFixtures/AppFixtures.php

```php
// ... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 16 - 29
        $question = QuestionFactory::createOne();
        $answer1 = new Answer();
// ... lines 32 - 33
        $answer2 = new Answer();
// ... lines 35 - 36

        $question->addAnswer($answer1);
        $question->addAnswer($answer2);

// ... lines 41 - 44
    }
}
```

If you think about it... this is *really* saying the same thing as when we set the relationship from the other direction: this `Question` *has* these two answers.

Let's see if it saves! Run the fixtures:

```
$ symfony console doctrine:fixtures:load
```

And... no errors! I think it worked! Double-check with:

> SELECT * FROM answer

```
$ symfony console doctrine:query:sql 'SELECT * FROM answer'
```

Let's see... yea! Here are the new answers. Oh, apparently I called them *both* "answer 1" - silly Ryan. But more importantly, each answer *is* correctly related to a `Question` .

Ok! so it turns out you *can* set data from both sides. The two sides of the relationship apparently behave identically.

Now, at this point, you might be saying to yourself:

> Why is this guy taking so much time to show me that something works exactly like I expect it too?

The "setters" Synchronize the Other Side of the Relation

Great question! Because... this *doesn't* really work like we just saw. Let me show you.

Open the `Question` class and find the `addAnswer()` method.

```
178 lines | src/Entity/Question.php
... lines 1 - 14
class Question
{
... lines 17 - 155
    public function addAnswer(Answer $answer): self
    {
        if (!$this->answers->contains($answer)) {
            $this->answers[] = $answer;
            $answer->setQuestion($this);
        }

        return $this;
    }
... lines 165 - 176
}
```

This was generated for us by the `make:entity` command. It first checks to see if the `$answers` property *already* contains this answer.... just to avoid a duplication. If it does *not*, it, of course, *adds* it to that property. But it *also* does something else, something very important: `$answer->setQuestion($this)` . Yup, it sets the *other* side of the relation.

So if an `Answer` is added to a `Question` , that `Question` is *also* set *onto* that `Answer` . Now, watch what happens if we comment-out this line...

178 lines | src/Entity/Question.php

```
// ... lines 1 - 14
class Question
{
// ... lines 17 - 155
    public function addAnswer(Answer $answer): self
    {
        if (!$this->answers->contains($answer)) {
            $this->answers[] = $answer;
            //$answer->setQuestion($this);
        }

        return $this;
    }
// ... lines 165 - 176
}
```

and then go reload the fixtures:

```
$ symfony console doctrine:fixtures:load
```

An error! The `question_id` column cannot be null on the `answer` table! It did *not* relate the `Question` to the `Answer` properly!

Owning vs Inverse

This is what I wanted to talk about. Each relation has two different sides and these sides have a name: the owning side and the inverse side. For a `ManyToOne` and `OneToMany` relationship, the owning side is always the `ManyToOne` side. And it's easy to remember: the owning side is where the foreign key column lives in the database. In this case, the `answer` table will have a `question_id` column so *this* is the "owning" side.

The `OneToMany` side is called the inverse side.

Why is this important? It's important because, when Doctrine saves an entity, it *only* looks at the data on the *owning* side of a relationship. Yup, it looks at the `$question` property on the `Answer` entity to figure out what to save to the database. It completely *ignores* the data on the inverse side. Really, the inverse side exists *solely* for the convenience of us reading that data: the convenience of being able to say `$question->getAnswers()`.

So right now, we are *only* setting the inverse side of the relationship. And so, when it saves the `Answer`, it does *not* link the `Answer` to this `Question`.

Inverse Side is Optional

And actually, the inverse side of a relationship is entirely *optional*. The `make:entity` command asked us if we wanted to map this side of the relationship. We could delete *everything* inside of `Question` that's related to answers, and the relationship would *still* be set up in the database and we could *still* use it. We just wouldn't be able to say `$question->getAnswers()`.

I'm telling you all this so that you can avoid potential WTF moments if you relate two objects... but they mysteriously don't save. Fortunately, the `make:entity` command takes care of all this ugliness *for* us by generating really smart `addAnswer()` and `removeAnswer()` methods that synchronize the owning side of the relationship. So unless you don't use `make:entity` or start deleting code, you won't need to think about this problem on a day-to-day basis.

Put back the `$answer->setQuestion()` code so that we can, once again, safely set the data from either side.

```
178 lines | src/Entity/Question.php
... lines 1 - 14
class Question
{
... lines 17 - 155
    public function addAnswer(Answer $answer): self
    {
        if (!$this->answers->contains($answer)) {
            $this->answers[] = $answer;
            $answer->setQuestion($this);
        }

        return $this;
    }
... lines 165 - 176
}
```

Back in the fixtures, now that we've learned all of this, delete the custom code.

```
33 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $questions = QuestionFactory::createMany(20);

        QuestionFactory::new()
            ->unpublished()
            ->many(5)
            ->create()
        ;

        AnswerFactory::createMany(100, function() use ($questions) {
            return [
                'question' => $questions[array_rand($questions)]
            ];
        });

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

And then, let's reload our fixtures:

```
$ symfony console doctrine:fixtures:load
```

Next: when we call `$question->getAnswers()` ... which we're currently doing inside of our template, what *order* is it returning those answers? And can we *control* that order? Plus we'll learn a config trick to optimize the query that's made when all we need to do is *count* the number of items in a relationship.

Chapter 9: Relation OrderBy & fetch=EXTRA_LAZY

You know what? On this page, we're missing the "created at" for each answer: I want to be able to see *when* each answer was posted. Let's fix that.

Head over to the template - `show.html.twig` - and, down here... right before the vote arrows, add a `<small>` tag and then `{{ answer.createdAt }}`. Of course, that will give us a `DateTime` object... and you can't just print a `DateTime`. But you *can* pipe it to the `date()` filter. Or in the last tutorial, we installed a library that allows us to say `|ago`.

```
97 lines | templates/question/show.html.twig
... lines 1 - 4
{% block body %}
<div class="container">
... lines 7 - 54
    <ul class="list-unstyled">
        {% for answer in question.answers %}
            <li class="mb-4">
                <div class="row">
... lines 59 - 65
                    <div class="col-2 text-end">
                        <small>{{ answer.createdAt|ago }}</small>
... lines 68 - 89
                    </div>
                </div>
            </li>
        {% endfor %}
    </ul>
</div>
{% endblock %}
```

When we refresh now... oh! We get an error:

> The `Question` object cannot be found by the `@ParamConverter` annotation.

That's a fancy way of saying that no `Question` for the `slug` in the URL could be found in the database. And *that's* because I reloaded my fixtures. Go to the homepage, refresh... and click into a fresh question. Actually, let me try a different one... I want something with several answers. Perfect. And each answer *does* display how long ago it was added.

Ordering $question->getAnswers() with ORM\OrderBy

But this highlights a small problem... or question: what *order* are these answers being returned from the database? Right now... there's *no* specific order. You can see that in the query for the answers: it just queries for all the answers where `question_id = ?` this question... but there's no `ORDER BY`.

At first, it seems like this is one of the downsides of using the convenience methods for a relationship like `$question->getAnswers()`: you don't have a lot of control over the results. But... that's not entirely true.

The easiest thing that you *can* control is how the answers are ordered. Go into the `Question` class and scroll up to the `$answers` property. To control the order add `@ORM\OrderBy()` and pass this an array with `{"createdAt" = "DESC"}`.

```
179 lines | src/Entity/Question.php
... lines 1 - 14
class Question
{
... lines 17 - 51
    /**
     * @ORM\OneToMany(targetEntity=Answer::class, mappedBy="question")
     * @ORM\OrderBy({"createdAt" = "DESC"})
     */
    private $answers;
... lines 57 - 177
}
```

That's it! Go back, refresh and... perfect! These are now ordered with the newest first!

Optimizing The Query to Count a Relation: EXTRA_LAZY

Let's learn another trick. On the homepage, we show the number of answers for each question. Well... kind of: they all say 6 because that number is still hardcoded. Let's fix that.

Open the template for this: `templates/question/homepage.html.twig` ... and I'll search for "6". Here it is. Replace this with `{{ question.answers|length }}`

```
50 lines | templates/question/homepage.html.twig
... lines 1 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in questions %}
        <div class="col-12 mb-3">
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
... lines 20 - 37
                <a class="answer-link" href="{{ path('app_question_show', { slug: question.slug }) }}" style="color: #fff;">
                    <p class="q-display-response text-center p-3">
                        <i class="fa fa-magic magic-wand"></i> {{ question.answers|length}} answers
                    </p>
                </a>
            </div>
        </div>
        {% endfor %}
    </div>
</div>
... lines 48 - 50
```

So we get the collection of answers and then count them. Simple enough! And if we try it... this works: two answers, six answers, eight answers.

But check out the web debug toolbar. Woh! We suddenly have a *lot* of queries. Click to open Doctrine's profiler. The first query is still for all of the question objects. But then, one-by-one it selects `FROM answer WHERE question_id = ?` a specific question. It does this for the first question, then it selects the answers for the next question... and the next and the next.

This is called the N+1 problem: We have 1 query that gives us all of the questions. Then, for each of the the N questions, when we ask for its answers, it makes *another* query. The total query count is the number of questions - N - plus 1 for the original.

We're going to talk more about the N+1 problem later and how to fix it. But there's kind of a *bigger* problem right now: we're querying for *all* of the answer data.... simply to count them! That's *total* overkill!

As soon as we access this `answers` property, Doctrine queries for all the data so that it can return all of the `Answer` objects. Normally, that's great - because we *do* want to use those `Answer` objects. But in this case... all we want to do is count them!

If you find yourself in this situation, there *is* a solution. In the `Question` class, at the end of the `OneToMany()`, pass a new option called `fetch=""` set to `EXTRA_LAZY`.

```
179 lines | src/Entity/Question.php
... lines 1 - 14
class Question
{
... lines 17 - 51
    /**
     * @ORM\OneToMany(targetEntity=Answer::class, mappedBy="question", fetch="EXTRA_LAZY")
     * @ORM\OrderBy({"createdAt" = "DESC"})
     */
    private $answers;
... lines 57 - 177
}
```

Watch what happens. Right now we have 21 queries. When we refresh, we *still* have 21 queries. But open up the profiler. The first query is still the same. But every query *after* just selects `COUNT() FROM answer`! Instead of querying for all of the `answer` data, it only counts them!

This is what `fetch="EXTRA_LAZY"` gets you. If Doctrine determines that you're accessing a relation... but you're only *counting* that relation - not *actually* trying to use its data - then it will create a "count" query instead of grabbing all the data.

That's awesome! *So* awesome that you might be wondering: why isn't this the *default* behavior? If I'm counting the relation, why would we *ever* want Doctrine to query for *all* of the data?

Well... `EXTRA_LAZY` isn't *always* a good thing. Go to a question show page. Having the `EXTRA_LAZY` actually causes an *extra* query here. Before that change, this page required 2 queries. Now it has *3*. Check them out. First, it selects the question data. Then it counts the answers. And *then* it re-does that query to grab all the data for the answers. That second `COUNT` query is new... and, in theory, shouldn't be needed.

The problem is the order of the code in the template. You can see this in `show.html.twig`: *before* we loop over the answers and use their data, we *first* count them. So at this moment Doctrine says:

> Hey! You want to count the answers! I'll make a quick COUNT query for that.

Then, a millisecond later, we loop over all the answers... and so we need their data anyways. This causes Doctrine to make the full query.

If we reversed the order of this code - where we loop and use the data *first* - Doctrine would *avoid* the extra COUNT query because it would already know how many answers it has because it just queried for their data.

All of this is probably not *too* important and I'm going to leave it. In general, don't overly worry about optimizing. In the real world, I use Blackfire on production to find what my *real* performance issues are.

Next: in addition to changing the order of the answers when we call `$question->getAnswers()`, we can also *filter* this collection to, for example, only return *approved* answers. Let's get that set up next.

Chapter 10: Filtering to Return only Approved Answers

As wonderful as our users are, sometimes we need to mark an answer as spam. Or, maybe in the future, we might add a system that notices too many links in an answer and marks it as "needs approval". So each answer will be one of three statuses: needs approval, spam, or approved. And only answers with the *approved* status should be visible on the site.

Adding the Answer status Property

Right now, inside of our `Answer` entity, we don't have any way to track the `status`. So let's add a new property for it. At your console run:

```
$ symfony console make:entity
```

We're going to update the `Answer` entity. Add a new field called `status` and make it a `string` type. This property will be a, kind of, `ENUM` field: it'll hold one of three possible short status strings. Set the length to 15, which will be more than enough to hold the status string. Make this required in the database and... done!

Generate the migration immediately:

```
$ symfony console make:migration
```

Let's go double check that *just* to make sure it doesn't contain any surprises

34 lines | migrations/Version20210902182514.php

```
... lines 1 - 12
final class Version20210902182514 extends AbstractMigration
{
... lines 15 - 19
    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE answer ADD status VARCHAR(15) NOT NULL');
... line 24
    }
... lines 26 - 32
}
```

It looks good:

> ALTER TABLE answer ADD status.

Close that, spin back to your terminal and execute it:

```
$ symfony console doctrine:migrations:migrate
```

Because we have exactly three possible statuses, I'm going to add a constant for each one. Now, if you're using PHP 8.1, you could use the new `enum` type to help with this - and you totally should. But either way, you'll ultimately store a string in the database.

Add `public const STATUS_NEEDS_APPROVAL = 'needs_approval'`. I just made up that `needs_approval` part - that's what will be stored in the database. Copy that, paste it twice, and create the other two statuses: `spam` and `approved`, setting each to a simple string.

```
122 lines | src/Entity/Answer.php
... lines 1 - 11
class Answer
{
    public const STATUS_NEEDS_APPROVAL = 'needs_approval';
    public const STATUS_SPAM = 'spam';
    public const STATUS_APPROVED = 'approved';
... lines 17 - 120
}
```

Awesome. Now default the `status` property down here to `self::STATUS_NEEDS_APPROVAL`: comments will "need approval" unless we say otherwise.

```
122 lines | src/Entity/Answer.php
... lines 1 - 11
class Answer
{
    public const STATUS_NEEDS_APPROVAL = 'needs_approval';
    public const STATUS_SPAM = 'spam';
    public const STATUS_APPROVED = 'approved';
... lines 17 - 50
    private $status = self::STATUS_NEEDS_APPROVAL;
... lines 52 - 120
}
```

Finally, down on `setStatus()`, let's add a sanity check: if someone passes a status that is *not* one of those three, we should throw an exception. So if not `in_array($status, [])`... and then I'll create an array with the three constants: `self::STATUS_NEEDS_APPROVAL`, `self::STATUS_SPAM` and `self::STATUS_APPROVED`. So if it's *not* inside that array, then throw a new `InvalidArgumentException()` with a nice message.

```
122 lines | src/Entity/Answer.php
... lines 1 - 11
class Answer
{
    public const STATUS_NEEDS_APPROVAL = 'needs_approval';
    public const STATUS_SPAM = 'spam';
    public const STATUS_APPROVED = 'approved';
... lines 17 - 50
    private $status = self::STATUS_NEEDS_APPROVAL;
... lines 52 - 110
    public function setStatus(string $status): self
    {
        if (!in_array($status, [self::STATUS_NEEDS_APPROVAL, self::STATUS_SPAM, self::STATUS_APPROVED])) {
            throw new \InvalidArgumentException(sprintf('Invalid status "%s"', $status));
        }

        $this->status = $status;

        return $this;
    }
}
```

A little gatekeeping to make sure that we always have a valid status.

Creating Approved and Non-Approved Answer Fixtures

Now that the new `status` property is done, open `src/Factory/AnswerFactory.php`. Down in `getDefaults()`, set `status` to `Answer::STATUS_APPROVED`.

63 lines | src/Factory/AnswerFactory.php

```
// ... lines 1 - 28
final class AnswerFactory extends ModelFactory
{
// ... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
// ... lines 41 - 45
            'status' => Answer::STATUS_APPROVED,
        ];
    }
// ... lines 49 - 61
}
```

So when we create answers via the factory, let's make them approved by default so they show up on the site.

But I actually *do* want a mixture of approved and not approved answers in my fixtures to make sure things are working. To allow that, add a new method: `public function`, how about, `needsApproval()`, that will return `self`. Inside, return `$this->addState()` and pass this an array with `status` set to `Answer::STATUS_NEEDS_APPROVAL`.

68 lines | src/Factory/AnswerFactory.php

```
// ... lines 1 - 28
final class AnswerFactory extends ModelFactory
{
// ... lines 31 - 37
    public function needsApproval(): self
    {
        return $this->addState(['status' => Answer::STATUS_NEEDS_APPROVAL]);
    }
// ... lines 42 - 66
}
```

Now go open the fixtures class: `src/DataFixtures/AppFixtures.php`. These 100 answers, thanks to `getDefaults()`, will all be approved. Let's *also* save some "needs approval" answers. Do that with `AnswerFactory::new()` - to get a new instance of `AnswerFactory`, `->needsApproval()`, `->many()` to say that we want 20, and finally `->create()` to actually do the work.

Thanks to the `getDefaults()` method, for each `Answer`, this will create a new, unpublished question to relate to... which is actually not what we want: we want to relate this to one of the questions we've already created. Let's use the same trick we used before. Inside the `new()` method, we can pass a callable. Use the `$questions` variable to get it into scope... and then paste.

38 lines | src/DataFixtures/AppFixtures.php

```
// ... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 16 - 28
        AnswerFactory::new(function() use ($questions) {
            return [
                'question' => $questions[array_rand($questions)]
            ];
        })->needsApproval()->many(20)->create();
// ... lines 34 - 35
    }
}
```

So this will create 20 new, "needs approval" answers that are set to a random published `Question`. Phew! Let's get these loaded. At your terminal, run:

```
$ symfony console doctrine:fixtures:load
```

No errors!

Creating Question::getApprovedAnswers()

Cool. *But* how do we actually *hide* the non-approved answers from the frontend?

Go back to the homepage... and find a question with a lot of answers. This one has 10, so there's a *pretty* good chance that one of these is *not* approved and should be hidden. But how *can* we hide those answers?

Inside of `show.html.twig`, we get the answers by saying `question.answers`.

```
97 lines | templates/question/show.html.twig
... lines 1 - 4
{% block body %}
... lines 6 - 54
    <ul class="list-unstyled">
        {% for answer in question.answers %}
... lines 57 - 92
        {% endfor %}
    </ul>
... line 95
{% endblock %}
```

So this is calling `$question->getAnswers()`, which, of course, returns *all* of the related answers.

```
179 lines | src/Entity/Question.php
... lines 1 - 14
class Question
{
... lines 17 - 148
    /**
     * @return Collection|Answer[]
     */
    public function getAnswers(): Collection
    {
        return $this->answers;
    }
... lines 156 - 177
}
```

We *could* solve this by going back to `QuestionController` and, in the `show()` action, executing a custom query through the `AnswerRepository` where question equals this question *and* status = approved... and *then* passing that array into the template.

But... ugggh, I don't want to do that! I *still* want to be able to use a nice shortcut method in my template! It makes my life so much easier! So... let's do that!

In the `Question` class... anywhere, but right after `getAnswers()` makes sense, create a new function called `getApprovedAnswers()`. This will return a `Collection`, just like `getAnswers()`: `Collection` is the common interface that `ArrayCollection` and `PersistentCollection` both implement.

```
186 lines | src/Entity/Question.php
... lines 1 - 14
class Question
{
... lines 17 - 156
    public function getApprovedAnswers(): Collection
    {
... lines 159 - 161
    }
... lines 163 - 184
}
```

Inside, we're going to loop over the answers and *remove* any that are *not* approved. We could do this with a `foreach` loop... but there's a helper method on `Collection` for exactly this.

Return `$this->answers->filter()` and pass this a callback with an `$answer` argument. This callback will be executed one time for *each* `Answer` object inside the answers collection. If we return true, it will be included in the final collection that's returned. And if we return false, it won't. So we're taking the answers collection and filtering it.

```
186 lines | src/Entity/Question.php
... lines 1 - 14
class Question
{
... lines 17 - 156
    public function getApprovedAnswers(): Collection
    {
        return $this->answers->filter(function(Answer $answer) {
... line 160
        });
    }
... lines 163 - 184
}
```

Inside the callback, we need to check if this answer's status is "approved". Instead of doing that here, let's add a helper method inside of `Answer`.

Down here, add `public function isApproved()` that will return a boolean. Inside, we need return `$this->status === self::STATUS_APPROVED`.

```
127 lines | src/Entity/Answer.php
... lines 1 - 11
class Answer
{
... lines 14 - 121
    public function isApproved(): bool
    {
        return $this->status === self::STATUS_APPROVED;
    }
}
```

Back over in `Question`, it's easy: include this answer if `$answer->isApproved()`.

186 lines | src/Entity/Question.php

```
... lines 1 - 14
class Question
{
... lines 17 - 156
    public function getApprovedAnswers(): Collection
    {
        return $this->answers->filter(function(Answer $answer) {
            return $answer->isApproved();
        });
    }
... lines 163 - 184
}
```

Sweet! We now have a new method inside of `Question` that will only return *approved* answers. All we need to do *now* is use this our template. In `show.html.twig`, use it in both spots: `question.approvedAnswers` ... and `question.approvedAnswers`.

97 lines | templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 47
    <div class="d-flex justify-content-between my-4">
        <h2 class="">Answers <span style="font-size:1.2rem;">({{ question.approvedAnswers|length }})</span></h2>
        <button class="btn btn-sm btn-secondary">Submit an Answer</button>
    </div>
... lines 52 - 54
    <ul class="list-unstyled">
        {% for answer in question.approvedAnswers %}
... lines 57 - 92
        {% endfor %}
    </ul>
... line 95
{% endblock %}
```

There's also a spot on the homepage where we show the count... make sure to use `question.approvedAnswers` here too.

50 lines | templates/question/homepage.html.twig

```
... lines 1 - 2
{% block body %}
... lines 4 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in questions %}
        <div class="col-12 mb-3">
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
... lines 20 - 37
                <a class="answer-link" href="{{ path('app_question_show', { slug: question.slug }) }}" style="color: #fff;">
                    <p class="q-display-response text-center p-3">
                        <i class="fa fa-magic magic-wand"></i> {{ question.approvedAnswers|length}} answers
                    </p>
                </a>
            </div>
        </div>
        {% endfor %}
    </div>
</div>
{% endblock %}
... lines 49 - 50
```

Ok! Moment of truth. Right now we have 10 answers on this question. When I refresh... oh, it's still 10! Boo. We either have a bug... or that was bad luck and this question has only *approved* answers. Click back. Find another question that has a lot of answers. Let's see... try this one. We got it! This question originally had 11 answers, but now that we're only showing *approved* answers, we see 6.

So... this works! *But*.... there's a performance problem... and you may have spotted it. Open up the profiler to see the queries. We're still querying for all of the answers `WHERE question_id = 457`. But then... we're only rendering the six *approved* ones. That's wasteful! What we *really* want is some way to have this nice `getApprovedAnswers()` method... but make it query *only* for the approved answers... instead of querying for *all* of them and filtering them in PHP.

Is that possible? Yes! Via an amazing "criteria" system.

Chapter 11: Collection Criteria for Custom Relation Queries

When we render the answers for a question, we only want to render the *approved* answers. I got clever and did this by adding a `getApprovedAnswers()` method. It loops over *all* of the answers... but then returns only the *approved* ones.

186 lines | src/Entity/Question.php

```
... lines 1 - 14
class Question
{
... lines 17 - 156
    public function getApprovedAnswers(): Collection
    {
        return $this->answers->filter(function(Answer $answer) {
            return $answer->isApproved();
        });
    }
... lines 163 - 184
}
```

The *problem* with this approach is... performance. It's pretty silly to query for *every* `Answer` related to this question... and then only render *some* of them.

Realistically, if there are only ever a *few* non-approved answers, this is no big deal. But if it's possible that a question could have *many* non-approved answers, this page could *really* slow down. Imagine querying for 200 answers because some SPAM bot hit our site... only to render 6 of them.

Unfortunately, because we're in an entity, we can't simply grab the `AnswerRepository` service from inside of this method and create a custom query. So... are we stuck? Do we need to back up to our controller and do a custom query for the answers there?

Fortunately, no! These Doctrine Collection objects have a few tricks up their sleeves, including a special "criteria" system for *just* this situation. It allows us to *describe* how we want to filter the answers and then it *uses* that when it queries!

Creating the Criteria Expression

Remove the filter stuff entirely... and instead say `$criteria = Criteria` - the one from `Doctrine\Collections` - `::create()`.

188 lines | src/Entity/Question.php

```
... lines 1 - 7
use Doctrine\Common\Collections\Criteria;
... lines 9 - 15
class Question
{
... lines 18 - 157
    public function getApprovedAnswers(): Collection
    {
        $criteria = Criteria::create()
... lines 161 - 163
    }
... lines 165 - 186
}
```

This object "kind of" looks like a query builder. For example, it has an `->andWhere()` method. The big difference is what we put inside of this. Instead of a simple string, we need to use a criteria "expression": `Criteria::expr()`, `->eq()` - for equals - and pass this `status`: the property we want to use in the WHERE. For the second arg, use

Answer::STATUS_APPROVED .

188 lines | src/Entity/Question.php

```
... lines 1 - 7
use Doctrine\Common\Collections\Criteria;
... lines 9 - 15
class Question
{
... lines 18 - 157
    public function getApprovedAnswers(): Collection
    {
        $criteria = Criteria::create()
            ->andWhere(Criteria::expr()->eq('status', Answer::STATUS_APPROVED));
... lines 162 - 163
    }
... lines 165 - 186
}
```

This Criteria object now "describes" how we want to filter the answers: "where status = approved". To *use* this say $this->answers->matching($criteria) .

188 lines | src/Entity/Question.php

```
... lines 1 - 7
use Doctrine\Common\Collections\Criteria;
... lines 9 - 15
class Question
{
... lines 18 - 157
    public function getApprovedAnswers(): Collection
    {
        $criteria = Criteria::create()
            ->andWhere(Criteria::expr()->eq('status', Answer::STATUS_APPROVED));

        return $this->answers->matching($criteria);
    }
... lines 165 - 186
}
```

For me, the criteria syntax - especially the expression stuff - is a little cryptic. But it's still usually pretty easy to create whatever logic I need. And, most importantly, it gets the job done.

Check it out: we have 6 approved answers now. After we refresh... good: this *still* renders 6 answers. Open the Doctrine profiler to look at the queries. And... amazing! When we call getApprovedAnswers() , it now queries from answer where question_id equals this question *and* status = 'approved ! Even the COUNT() query *above* this is smart!

So... *that's* the criteria system! And other than getting a little bit fancier with the expressions you create... it's just that simple and powerful. It's one of my favorite secrets inside Doctrine.

Moving Criteria Logic into the Repository

By the way, if you don't like having the query logic in your entity, I don't blame you! But no worries: we can move it into our repository. Copy the criteria code and then open up src/Repository/AnswerRepository.php ... since this criteria relates to answers. Delete the example code and replace it with a new public *static* function called createApprovedCriteria() . This will, of course, return a Criteria object. Paste the logic and return.

```
 29 lines | src/Repository/AnswerRepository.php
... lines 1 - 6
use Doctrine\Common\Collections\Criteria;
... lines 8 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 22
    public static function createApprovedCriteria(): Criteria
    {
        return Criteria::create()
            ->andWhere(Criteria::expr()->eq('status', Answer::STATUS_APPROVED));
    }
}
```

There are two reasons I'm making this *static*. First, because I want to be able to call this from my `Question` entity. And since I can't inject service *objects* into an entity, the only way to do that is by making this method static. And second... this method *can* be static! It doesn't need to use the `$this` variable to call any methods on the repository.

Anyways, now that we have this, back in `Question` we can simplify to return `$this->answers->matching()`, `AnswerRepository::createApprovedCriteria()`.

```
 186 lines | src/Entity/Question.php
... lines 1 - 4
use App\Repository\AnswerRepository;
... lines 6 - 16
class Question
{
... lines 19 - 158
    public function getApprovedAnswers(): Collection
    {
        return $this->answers->matching(AnswerRepository::createApprovedCriteria());
    }
... lines 163 - 184
}
```

Cool! If you go back to the site and try it now... still 6 questions: it still works.

Using Criteria in a QueryBuilder

One other cool thing about these `Criteria` objects is that you can reuse them with the query builder. For example, suppose we need to create a custom query that returns 10 approved answers. Add a new method for this: `public function findAllApproved()` with an `int $max = 10` argument... and this will return and array. Though, more specifically, I'll advertise in PHPDoc that this will return an array of `Answer` objects.

```
 41 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 28
    /**
     * @return Answer[]
     */
    public function findAllApproved(int $max = 10): array
    {
... lines 34 - 38
    }
}
```

Inside, create the query builder like normal: return `$this->createQueryBuilder()` and pass it `answer` for the alias. To filter for only approved answers, we would normally say `->andWhere('answer.status = :status')`. But... I want to do this *without* duplicating the approved logic that we already wrote in the criteria method. Fortunately, we can put a criteria *into* a `QueryBuilder` by saying `->addCriteria()` and then `self::createApprovedCriteria()` .

```
41 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 28
    /**
     * @return Answer[]
     */
    public function findAllApproved(int $max = 10): array
    {
        return $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
... lines 36 - 38
    }
}
```

Cool huh? From here, we can finish the query like normal: `->setMaxResults($max)` and then `->getQuery()->getResult()` .

```
41 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 28
    /**
     * @return Answer[]
     */
    public function findAllApproved(int $max = 10): array
    {
        return $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->setMaxResults($max)
            ->getQuery()
            ->getResult();
    }
}
```

I won't *use* this method right now, but hopefully you get the idea.

Next: let's add a new page to our site that shows the most popular answers. This will give us a great example to learn more about - then solve - the N+1 problem.

Chapter 12: Most Popular Answers Page

Let's build a "top answers" page where we list the answers with the most votes for *all* questions on our site.

Creating the Route, Controller & Template

Open `AnswerController` and create a new public function called `popularAnswers()`.

45 lines | src/Controller/AnswerController.php

```php
// ... lines 1 - 11
class AnswerController extends AbstractController
{
// ... lines 14 - 16
    public function popularAnswers()
    {
// ... line 19
    }
// ... lines 21 - 43
}
```

Add an `@Route()` above this - or use the `Route` attribute if you're on PHP 8 - with the URL `/answers/popular`. Immediately give this a name so we can link to it: `app_popular_answers`.

45 lines | src/Controller/AnswerController.php

```php
// ... lines 1 - 11
class AnswerController extends AbstractController
{
    /**
     * @Route("/answers/popular", name="app_popular_answers")
     */
    public function popularAnswers()
    {
// ... line 19
    }
// ... lines 21 - 43
}
```

Inside, render a template: `answer/popularAnswers.html.twig`.

45 lines | src/Controller/AnswerController.php

```php
// ... lines 1 - 11
class AnswerController extends AbstractController
{
    /**
     * @Route("/answers/popular", name="app_popular_answers")
     */
    public function popularAnswers()
    {
        return $this->render('answer/popularAnswers.html.twig');
    }
// ... lines 21 - 43
}
```

Now, copy that template name and, down in the `templates/` directory, create the new `answer/` folder... and inside, the new file: `popularAnswers.html.twig`. I'll paste in a little structure to get us started.

```
12 lines | templates/answer/popularAnswers.html.twig
{% extends 'base.html.twig' %}

{% block title %}Popular Answers{% endblock %}

{% block body %}
... lines 6 - 10
{% endblock %}
```

This extends `base.html.twig`, overrides the `title` block to customize the title... and in the `body` block, adds some basic structure. Let's put an `<h1>` that says "Most Popular Answers".

```
12 lines | templates/answer/popularAnswers.html.twig
{% extends 'base.html.twig' %}

{% block title %}Popular Answers{% endblock %}

{% block body %}
<div class="container my-md-4">
    <div class="row">
        <h1>Most Popular Answers</h1>
    </div>
</div>
{% endblock %}
```

Before we try this, open up `base.html.twig` so we can link to this. Scroll down a little. Inside of the `navbar`, we have an empty `<ul>` that's just *waiting* for a link. Add an `<li>` with `class="nav-item"` ... and an `a` tag inside with `href` set to our new page: `path('app_popular_answers')`. Say "Answers" for the link text... and this needs `class="nav-link"`.

```
38 lines | templates/base.html.twig
... line 1
<html>
... lines 3 - 14
    <body>
        <nav class="navbar navbar-expand-lg navbar-light bg-light px-1" style="height: 60px;">
... lines 17 - 20
            <div class="collapse navbar-collapse">
                <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                    <li class="nav-item">
                        <a class="nav-link" href="{{ path('app_popular_answers') }}">Answers</a>
                    </li>
                </ul>
            </div>
... lines 28 - 29
        </nav>
... lines 31 - 35
    </body>
</html>
```

Now let's try this thing. Refresh... and click the link. Hello normal, boring, but *functional* page.

Querying for the Most Popular Answers

To get the most popular answers, we need a custom query. Well, technically we could use the `findBy()` method on `AnswerRepository` and use its "order by" argument. But let's add a full custom repository method instead: that will be nice and descriptive.

Open up `AnswerRepository`. At the bottom, add the method. Let's call it `findMostPopular()` and set the return type to an array. Like normal, I'll use PHPDoc to advertise that, more specifically, this will return an array of `Answer` objects.

```
54 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 40
    /**
     * @return Answer[] Returns an array of Answer objects
     */
    public function findMostPopular(): array
    {
... lines 46 - 51
    }
}
```

Inside, it's a simple query: return `$this->createQueryBuilder('answer')`, `->addCriteria()` and reuse `self::createApprovedCriteria()` so that this only returns *approved* answers.

```
54 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 40
    /**
     * @return Answer[] Returns an array of Answer objects
     */
    public function findMostPopular(): array
    {
        return $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
... lines 48 - 51
    }
}
```

Then `->orderBy('answer.votes', 'DESC')`, `->setMaxResults(10)` to only return the top 10 answers, `->getQuery()`, `->getResult()`.

```
54 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 40
    /**
     * @return Answer[] Returns an array of Answer objects
     */
    public function findMostPopular(): array
    {
        return $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->orderBy('answer.votes', 'DESC')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult();
    }
}
```

Beautiful! Back in the controller, autowire `AnswerRepository $answerRepository`, and then we can say `$answers = $answerRepository->findMostPopular()`.

```
 50 lines | src/Controller/AnswerController.php
... lines 1 - 5
use App\Repository\AnswerRepository;
... lines 7 - 12
class AnswerController extends AbstractController
{
... lines 15 - 17
    public function popularAnswers(AnswerRepository $answerRepository)
    {
        $answers = $answerRepository->findMostPopular();
... lines 21 - 24
    }
... lines 26 - 48
}
```

Add a second argument to `render()` so that we can pass an `answers` variable to Twig set to this array of answers.

```
 50 lines | src/Controller/AnswerController.php
... lines 1 - 5
use App\Repository\AnswerRepository;
... lines 7 - 12
class AnswerController extends AbstractController
{
... lines 15 - 17
    public function popularAnswers(AnswerRepository $answerRepository)
    {
        $answers = $answerRepository->findMostPopular();

        return $this->render('answer/popularAnswers.html.twig', [
            'answers' => $answers
        ]);
    }
... lines 26 - 48
}
```

In the template, add a `ul` and loop over `answers` with `{% for answer in answers %}`. Let's start real simple: render `answer.votes` so we can at *least* make sure that we have the most popular on top.

```
 18 lines | templates/answer/popularAnswers.html.twig
... lines 1 - 4
{% block body %}
<div class="container my-md-4">
    <div class="row">
... lines 8 - 9
        <ul class="list-unstyled">
            {% for answer in answers %}
                <li>{{ answer.votes }}</li>
            {% endfor %}
        </ul>
    </div>
</div>
{% endblock %}
```

Spin over to your browser, refresh and... got it! 10 answers with the most highly voted on top.

Reusing the Answer Templates

So on the question show page, we already have a nice structure for rendering answers. I want to reuse this on our new popular answers page. Open `question/show.html.twig`. Select everything inside the `for` loop - the entire `<li>` that renders a single answer - and copy it. Then, in the `templates/answer/` directory, create a new file called `_answer.html.twig`... and paste!

```
37 lines | templates/answer/_answer.html.twig
<li class="mb-4">
    <div class="row">
        <div class="col-1">
            <img src="{{ asset('images/tisha.png') }}" width="50" height="50" alt="Tisha avatar">
        </div>
        <div class="col-9">
            {{ answer.content|parse_markdown }}
            <p>-- {{ answer.username }}</p>
        </div>
        <div class="col-2 text-end">
            <small>{{ answer.createdAt|ago }}</small>
            <div
                class="vote-arrows"
                {{ stimulus_controller('answer-vote', {
                    url: path('answer_vote', {
                        id: answer.id
                    })
                }) }}
            >
                <button
                    class="vote-up btn btn-link"
                    name="direction"
                    value="up"
                    {{ stimulus_action('answer-vote', 'clickVote') }}
                ><i class="far fa-arrow-alt-circle-up"></i></button>
                <button
                    class="vote-down btn btn-link"
                    name="direction"
                    value="down"
                    {{ stimulus_action('answer-vote', 'clickVote') }}
                ><i class="far fa-arrow-alt-circle-down"></i></button>
                <span><span {{ stimulus_target('answer-vote', 'voteTotal') }}>{{ answer.votes }}</span></span>
            </div>
        </div>
    </div>
</li>
```

Back in `show.html.twig`, delete all of this and replace it with `{{ include('answer/_answer.html.twig') }}`.

```
62 lines | templates/question/show.html.twig
... lines 1 - 4
{% block body %}
<div class="container">
... lines 7 - 54
    <ul class="list-unstyled">
        {% for answer in question.approvedAnswers %}
            {{ include('answer/_answer.html.twig') }}
        {% endfor %}
    </ul>
</div>
{% endblock %}
```

Now copy *that* line and, in the popular answers template, repeat this! The new template *includes* the `<li>` element... so this will fit perfectly inside of our `ul`.

18 lines | templates/answer/popularAnswers.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container my-md-4">
    <div class="row">
... lines 8 - 9
        <ul class="list-unstyled">
            {% for answer in answers %}
                {{ include('answer/_answer.html.twig') }}
            {% endfor %}
        </ul>
    </div>
</div>
{% endblock %}
```

Conditionally Rendering the Answer's Question

Phew! Let's check it! Refresh and... very nice! But hmm, in *this* context, we really need to render which *question* this answer is answering. We *don't* want to do that on the question show page - that would be redundant - but we *do* want it here.

To allow that, in `popularAnswers.html.twig` , add a second argument to `include()` and pass in a new variable called `showQuestion` set to `true` .

20 lines | templates/answer/popularAnswers.html.twig

```
... lines 1 - 4
{% block body %}
<div class="container my-md-4">
    <div class="row">
... lines 8 - 9
        <ul class="list-unstyled">
            {% for answer in answers %}
                {{ include('answer/_answer.html.twig', {
                    showQuestion: true
                }) }}
            {% endfor %}
        </ul>
    </div>
</div>
{% endblock %}
```

In `_answer.html.twig` , we can use that: if `showQuestion|default(false)` and `endif` . Thanks to the `default` filter, if this variable is *not* passed, instead of an error, it'll default to false.

48 lines | templates/answer/_answer.html.twig

```
<li class="mb-4">
    {% if showQuestion|default(false) %}
... lines 3 - 11
    {% endif %}
    <div class="row">
... lines 14 - 45
    </div>
</li>
```

Inside, add an `<a>` tag with `href=""` set to `{{ path('app_question_show') }}` : the route to the question show page. This route needs a `slug` parameter set to `answer.question.slug` . Also give this some classes: `mb-1` and `link-secondary"` . For the text, say `<strong>` "Question" and then print the question text: `answer.question.question` .

 48 lines | templates/answer/_answer.html.twig

```
<li class="mb-4">
    {% if showQuestion|default(false) %}
        <a
            href="{{ path('app_question_show', {
                slug: answer.question.slug
            }) }}"
            class="mb-1 link-secondary"
        >
            <strong>Question:</strong>
            {{ answer.question.question }}
        </a>
    {% endif %}
    <div class="row">
 ... lines 14 - 45
    </div>
</li>
```

That *does* look funny, but... it's correct: `answer.question` gives us the `Question` object... then the last part reads *its* `question` property.

Back at our browser, refresh and... yikes! That *technically* works but these questions are *way* too long! We need to shorten them!

Next, let's learn about Twig's powerful `u` filter *and* add a method to our `Answer` class that will make our code a *whole* lot more readable.

Chapter 13: The |u Filter & String Component

The questions on this page are *way* too long. We need to shorten them!

But before we do, this `answer.question.question` thing is bothering me: it looks... kind of confusing. Let's make this more clear by adding a custom method to our `Answer` class.

48 lines | templates/answer/_answer.html.twig

```
<li class="mb-4">
    {% if showQuestion|default(false) %}
        <a
... lines 4 - 7
        >
... line 9
            {{ answer.question.question }}
        </a>
    {% endif %}
... lines 13 - 46
</li>
```

Adding Answer::getQuestionText() for Clarity

Open `src/Entity/Answer.php`. It doesn't matter where... but right by `getQuestion()` makes sense, add a new method: public function `getQuestionText()`, which will return a `string`.

136 lines | src/Entity/Answer.php

```
... lines 1 - 11
class Answer
{
... lines 14 - 98
    public function getQuestionText(): string
    {
... lines 101 - 105
    }
... lines 107 - 134
}
```

On a high level, this method makes me happy! If I have an `Answer` object, there's a good chance that I might want to easily get the question text *related* to this answer. Inside, I'll start by coding defensively: if not `$this->getQuestion()` - so if there is *no* related `Question` object, return empty quotes.

136 lines | src/Entity/Answer.php

```
... lines 1 - 11
class Answer
{
... lines 14 - 98
    public function getQuestionText(): string
    {
        if (!$this->getQuestion()) {
            return '';
        }
... lines 104 - 105
    }
... lines 107 - 134
}
```

Now, you might be screaming:

> Hey Ryan! I thought the question property was required in the database! How could we *not* have a question?

And... that's mostly right! We can't *save* an `Answer` to the database without a `Question`. But, in theory, we *could* create a new `Answer` object and call `getQuestionText()` on it *before* even *trying* to save it. To avoid an error if we did that, I'm coding defensively.

At the bottom, return `$this->getQuestion->getQuestion()`... but cast that to a `string`, just in case it's `null`... which, again, isn't likely since that property is required in the database, but it *is* technically possible.

136 lines | src/Entity/Answer.php

```
// ... lines 1 - 11
class Answer
{
// ... lines 14 - 98
    public function getQuestionText(): string
    {
        if (!$this->getQuestion()) {
            return '';
        }

        return (string) $this->getQuestion()->getQuestion();
    }
// ... lines 107 - 134
}
```

Thanks to the new method, over in `_answer.html.twig`, we can change this to `{{ answer.questionText }}`.

48 lines | templates/answer/_answer.html.twig

```
<li class="mb-4">
    {% if showQuestion|default(false) %}
        <a
// ... lines 4 - 7
        >
            <strong>Question:</strong>
            {{ answer.questionText }}
        </a>
    {% endif %}
// ... lines 13 - 46
</li>
```

So much nicer. But... the front-end still looks weird. So let's shorten the question string!

Twig's "u" Filter & the String Component

In Twig, we have a special filter called `|u`. This filter leverages Symfony's `string` component to give you what's called a `UnicodeString` object. It's basically an object that *wraps* this string... and gives you access to a bunch of useful methods. One of those methods is called `truncate()`. This means we can say `.truncate()`. Pass this 80 and `'...`.

48 lines | templates/answer/_answer.html.twig

```
<li class="mb-4">
    {% if showQuestion|default(false) %}
        <a
... lines 4 - 7
        >
            <strong>Question:</strong>
            {{ answer.questionText|u.truncate(80, '...') }}
        </a>
    {% endif %}
... lines 13 - 46
</li>
```

So *if* the string is longer than 80 characters, truncate it and add a `...` to the end. I love it!

Before we try this, search for "Symfony string component" to find its documentation. If you scroll down... you'll see a bunch of examples of what you can do - in PHP - with the string component. This `u()` function in PHP creates the same thing as our `|u` filter. Down here, you can see a ton of examples of what you can do - like lower-casing, title-casing, camel-casing... and a lot more... *including* a `truncate()` method. So if you *ever* need to mess around with strings - in Twig or PHP - don't forget about this component!

But... if we try this... it doesn't actually work! It says:

> the `u` filter is part of the `StringExtension`... try running `composer require twig/string-extra`.

No problem! Find your terminal and run that:

```
$ composer require twig/string-extra
```

When it finishes... we can now refresh and see... awesome! We have truncated questions!

But look down at the web debug toolbar. This page made 8 queries... which seems like a lot just to render 10 answers. This is because we're suffering from the the N+1 query problem.

Next, let's learn more about this and see how we can join across a relationship to solve it.

Chapter 14: Joining Across a Relationship & The N + 1 Problem

Look at the queries on this page: there are 8... or for you there might be 11, 10 or 9: it depends on how many *unique* questions these 10 answers are related to.

The N+1 Problem

Whatever your number is, that's a lot of queries for such a simple page! The *cause* of all of this is the N+1 problem.

Look at the queries in the profiler. The first is for the answers: where status is approved, ordered by the most votes DESC, limit 10. Simple enough. Then, each time we render an answer, we *also* render that answer's question text. The moment we do that, Doctrine makes a second query from the `question` table to get that answer's question data: so in this case `WHERE id = 463`. Then we render the second answer... and make another query to get *its* question data... which is this third query.

Ultimately, we end up with 1 query to get the 10 answers plus 10 *more* queries: one for each answer's question. That's the N + 1 problem. Well, if two answers share the same question, you might have *less* than 11 queries - but it's still not great.

This is a classic problem that's *really* easy to trigger when using a nice system like Doctrine. In `AnswerController`, we simply query for the answers.

```
50 lines | src/Controller/AnswerController.php
... lines 1 - 12
class AnswerController extends AbstractController
{
... lines 15 - 17
    public function popularAnswers(AnswerRepository $answerRepository)
    {
        $answers = $answerRepository->findMostPopular();
... lines 21 - 24
    }
... lines 26 - 48
}
```

Then, as we loop over them and render `_answer.html.twig`, we innocently render `answer.questionText` and `answer.question.slug`.

```
48 lines | templates/answer/_answer.html.twig
<li class="mb-4">
    {% if showQuestion|default(false) %}
        <a
            href="{{ path('app_question_show', {
                slug: answer.question.slug
            }) }}"
            class="mb-1 link-secondary"
        >
            <strong>Question:</strong>
            {{ answer.questionText|u.truncate(80, '...') }}
        </a>
    {% endif %}
... lines 13 - 46
</li>
```

It doesn't look like much, but *those* lines trigger an extra query.

The point is: we end up with a lot of queries on this page and, in theory, we shouldn't need so many! Let's think: in a normal database, how would we solve this? Thinking about the query, we could select the most popular answers and then `INNER JOIN` over to the `question` table to grab *that* data all at once. Yup, one query to return both the answer *and* question data.

Joining in a QueryBuilder

Can we add a join with Doctrine? Of course! Head over to `AnswerRepository`, to the `findMostPopular()` method. It's this simple: `->innerJoin()` passing this `answer.question` and then `question`.

```
55 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 43
    public function findMostPopular(): array
    {
        return $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->orderBy('answer.votes', 'DESC')
            ->innerJoin('answer.question', 'question')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult();
    }
}
```

Remember: `answer` is the alias we're using for our `Answer` entity. So the `answer.question` part refers to the `question` *property* on the `Answer` class. This basically tells Doctrine:

> Hey! I want you to do an inner join across the `answer.question` relationship.

We don't need to tell Doctrine *how* to join like you would in a normal query... we don't need to say "JOIN question ON answer.question_id = question.id". Nope! Doctrine looks at the `$question` property in `Answer`, sees that it's a relationship over to the `question` table and then generates the SQL needed automatically. It's awesome!

The second argument isn't important yet, but this becomes the "alias" to the `Question` entity, just like how `answer` is the alias to the `Answer` entity.

The 2 Reasons to Join

Ok, so let's try this! Close the profiler, refresh and... hmm. We have the same number of queries! So... that didn't work.

Open up the profiler. If you look at the first query... cool! There's the inner join! And it's perfect: Doctrine generated the exact SQL needed. So then... why do we still have all these extra queries? Shouldn't Doctrine be able to get all the question data from the first?

Yes... but the problem is that, while we *did* join over to the `question` table... we didn't actually *select* any question *data*. It's still only selecting from `answer`. This is more obvious if we look at the formatted query. It joins to `question`, but only selects from `answer`.

This leads us to an important point! There are two reasons that you might use a JOIN in a query. The first is when you want to select more data, and that's our situation: we want to select all the answer *and* question data.

The *second* situation is when you want to join across a relationship... not to select more data, but to filter or order the results based on something in the joined table. We'll see that in a minute.

Selecting Data on a Joined Table

The point is: if you want to select more data, then you need to actually *say* that in the query. You do that with `->addSelect()` and then the alias to the entity: `question`.

```
56 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 43
    public function findMostPopular(): array
    {
        return $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->orderBy('answer.votes', 'DESC')
            ->innerJoin('answer.question', 'question')
            ->addSelect('question')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult();
    }
}
```

Two important things here. First, notice that I'm not saying `question.id`, `question.slug` or even `question.*`: I'm just saying `question`. This tells Doctrine to grab everything from `question`.

Second, even though we're now selecting more data, this does *not* change what this method returns: it will *still* return an array of `Answer` objects. But now, each `Answer` object will *already* have the `Question` data preloaded into it.

I'll prove it. Refresh the page. Yup! It still works *exactly* like before, because that method *still* returns an array of `Answer` objects! *But* our query count is down to 1!

Because we're now grabbing the `question` data in the first query, when we try to render the question for each answer, Doctrine realizes that it *already* has that data and avoids the query. That's the fix for the N+1 problem.

What about the *other* reason for joining... where you want to join across a relationship in order to filter the results... like to only return answers whose question is *published*.

Let's talk about that next by adding a search to our most popular answers page.

Chapter 15: Search, the Request Object & OR Query Logic

New mission: let's add a search box to this answers page. Head over to `popularAnswers.html.twig`. We don't actually need a row here... so I'm going to simplify my markup: move this `<ul>` to the bottom. Cool. Now we can give this `div` on top a `d-flex` class and also `justify-content-between`. This will let us have this `<h1>` on the left and a search form on the right.

```
 30 lines | templates/answer/popularAnswers.html.twig
... lines 1 - 5
<div class="container my-md-4">
    <div class="d-flex justify-content-between">
        <h1>Most Popular Answers</h1>
... lines 9 - 18
    </div>

    <ul class="list-unstyled">
        {% for answer in answers %}
            {{ include('answer/_answer.html.twig', {
                showQuestion: true
            }) }}
        {% endfor %}
    </ul>
</div>
... lines 29 - 30
```

Adding the Search Form

Add the `form` tag. This will submit right to this `AnswerController` route. So set the action to `{{ path('app_popular_answers') }}`. I'm going to *not* add a `method=""` attribute, because that defaults to `GET`, which is what you want for a search form.

```
 30 lines | templates/answer/popularAnswers.html.twig
... lines 1 - 5
<div class="container my-md-4">
    <div class="d-flex justify-content-between">
        <h1>Most Popular Answers</h1>

        <form action="{{ path('app_popular_answers') }}">
... lines 11 - 17
        </form>
    </div>
... lines 20 - 27
</div>
... lines 29 - 30
```

Inside, add the search field: `<input type="search">`. I'll break this on multiple lines. Add `name="q"` - that `q` could be anything, but we'll read that from our controller - a `class`, a `placeholder` and an `aria-label=""` for accessibility since we don't have a *real* label for this field.

```
30 lines | templates/answer/popularAnswers.html.twig
... lines 1 - 5
<div class="container my-md-4">
    <div class="d-flex justify-content-between">
        <h1>Most Popular Answers</h1>

        <form action="{{ path('app_popular_answers') }}">
            <input
                type="search"
                name="q"
                class="form-control"
                placeholder="Search..."
                aria-label="Search"
            >
        </form>
    </div>
... lines 20 - 27
</div>
... lines 29 - 30
```

By the way, I'm not using the Symfony's form component because we haven't talked about it yet... but also because this form is *so* simple that it's overkill anyways.

Refresh now. Looks awesome! And if we fill in the box and hit enter... we come *right* back to this page, but now with `?q=bananas` on the URL. The results don't change because we're not *reading* that query parameter in our code yet. So let's do that.

Symfony's Request Object

Head into `AnswerController`. Here's the plan: we're going to read that `?q=` from the URL, pass that string into `findMostPopular()` as an argument, and then use that inside of the query to add a where `answer.content LIKE` that search term. So, a fuzzy search.

But inside of the controller, how *can* we read the `?q=` from the URL in Symfony? Whenever you need to read *anything* from the request - like query parameters, post data, headers or cookies - you need Symfony's `Request` object: it holds *all* of these goodies.

And if you're in a controller, it's easy to get! Add a new argument type-hinted with `Request` - the one from `HttpFoundation`. You can *call* the argument anything, but I'll use `$request` to avoid being crazy.

```
52 lines | src/Controller/AnswerController.php
... lines 1 - 12
class AnswerController extends AbstractController
{
... lines 15 - 17
    public function popularAnswers(AnswerRepository $answerRepository, Request $request)
    {
... lines 20 - 26
    }
... lines 28 - 50
}
```

Here's how this works, it's pretty simple: *if* you have an argument to your controller that's *type-hinted* with Symfony's `Request` class, Symfony will pass you the `Request` object. This class has a *bunch* of methods on it to get *anything* you need from the request. To fetch a query parameter, use `$request->query->get()` and then the name: `q`. If that query parameter isn't there, this will return null.

```
 52 lines | src/Controller/AnswerController.php
... lines 1 - 12
class AnswerController extends AbstractController
{
... lines 15 - 17
    public function popularAnswers(AnswerRepository $answerRepository, Request $request)
    {
        $answers = $answerRepository->findMostPopular(
            $request->query->get('q')
        );
... lines 23 - 26
    }
... lines 28 - 50
}
```

Adding The Fuzzy LIKE Search

Over in the repository, add a new `string $search` argument... I'll let it be optional, in part, so that it accepts a `null` value.

```
 63 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 43
    public function findMostPopular(string $search = null): array
    {
... lines 46 - 60
    }
}
```

For the query, let's do it in pieces. Add `$queryBuilder =` the first part... and stop after the `addSelect()`.

```
 63 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 43
    public function findMostPopular(string $search = null): array
    {
        $queryBuilder = $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->orderBy('answer.votes', 'DESC')
            ->innerJoin('answer.question', 'question')
            ->addSelect('question');
... lines 51 - 60
    }
}
```

At the bottom `return $queryBuilder` and then the rest. I'll... fix my typo.

```
 63 lines | src/Repository/AnswerRepository.php
 ... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
 ... lines 18 - 43
    public function findMostPopular(string $search = null): array
    {
        $queryBuilder = $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->orderBy('answer.votes', 'DESC')
            ->innerJoin('answer.question', 'question')
            ->addSelect('question');
 ... lines 51 - 60
    }
}
```

The reason we're splitting this into two pieces is that we only want to apply the search logic *if* a search term was actually passed. Splitting it lets us say if `$search`, then, `$queryBuilder->andWhere()` with `answer.content` - that's the field we're going to search inside of - `LIKE :searchTerm`. That `searchTerm` could be anything: it's just a placeholder that we fill in by saying `->setParameter('searchTerm', $search)`. Except... to be a fuzzy search, we need to put `%` on each side. I know, it looks funny, but that's exactly what we want.

```
 63 lines | src/Repository/AnswerRepository.php
 ... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
 ... lines 18 - 43
    public function findMostPopular(string $search = null): array
    {
        $queryBuilder = $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->orderBy('answer.votes', 'DESC')
            ->innerJoin('answer.question', 'question')
            ->addSelect('question');

        if ($search) {
            $queryBuilder->andWhere('answer.content LIKE :searchTerm')
                ->setParameter('searchTerm', '%'.$search.'%');
        }

        return $queryBuilder
            ->setMaxResults(10)
            ->getQuery()
            ->getResult();
    }
}
```

Let's try it! Clear the `?q=` from the URL first. Cool: we have our normal, non-filtered results. Copy a word from an answer to search for. And... got it! The top item became the *second* result... but this *third* result is definitely new. But let's search a different word to make it even more obvious. Yup! That's working.

Using the Request Object in Twig

Though... it's not very obvious that we're filtering because we're not rendering the search term in the search box. Open up `popularAnswers.html.twig` and add a `value=""`. To render the current search term, we *could* read the query parameter in the controller and pass it into our template as a variable. But in this case, we can cheat because the request object is available in every template via `app.request`. So we can say `app.request.query.get('q')`.

```
 31 lines | templates/answer/popularAnswers.html.twig
... lines 1 - 5
<div class="container my-md-4">
    <div class="d-flex justify-content-between">
... lines 8 - 9
        <form action="{{ path('app_popular_answers') }}">
            <input
... lines 12 - 13
                value="{{ app.request.query.get('q') }}"
... lines 15 - 17
            >
        </form>
    </div>
... lines 21 - 28
</div>
... lines 30 - 31
```

Now... much better.

Filtering Across a Join

But, our search could be smarter! Well, if we wanted to make our search *really* smart, we should probably use something like Elasticsearch. But to make our search a *little* bit cooler, let's *also* return results that match the *question's* text.

For example, clear out the search term... and let's search for something that's in the first *question*. Hit enter. That result disappears because we're *not* searching the question text yet.

Over in `AnswerRepository`, let's think. We want to query where `answer.content LIKE :searchTerm` *or* the question's text is `LIKE :searchTerm`.

The `QueryBuilder` *does* have an `orWhere()` method. Big win, right!

Actually... no! I *never* use that method. The reason is that it gets tricky to get the parentheses correct in a query when using `orWhere()`. I'll show you what I mean when we see the final query. The point is that if you need an `OR` in a WHERE statement, you should *still* use `andWhere()`. Yup, we can say: `answer.content LIKE :searchTerm OR` and then pass another expression. We want to search on the `$question` property of the `Question` entity. And since we joined over to the `Question` entity and aliased it to `question`, we can say `question.question LIKE` and use that same `:searchTerm` placeholder.

```
 63 lines | src/Repository/AnswerRepository.php
... lines 1 - 15
class AnswerRepository extends ServiceEntityRepository
{
... lines 18 - 43
    public function findMostPopular(string $search = null): array
    {
        $queryBuilder = $this->createQueryBuilder('answer')
            ->addCriteria(self::createApprovedCriteria())
            ->orderBy('answer.votes', 'DESC')
            ->innerJoin('answer.question', 'question')
            ->addSelect('question');

        if ($search) {
            $queryBuilder->andWhere('answer.content LIKE :searchTerm OR question.question LIKE :searchTerm')
                ->setParameter('searchTerm', '%'.$search.'%');
        }
... lines 56 - 60
    }
}
```

That's it! When we refresh now... yes! That first result showed back up! And check out the query for this page, it's pretty sweet.... and easier to see in the formatted version. Check out the WHERE clause. I totally forgot that we were *already* filtering WHERE `status = approved`. But because we put the `OR` statement *inside* of the `andWhere()`, Doctrine surrounded the entire fuzzy search part with parentheses. If we had used `orWhere()`, that wouldn't have happened... and our query logic would have been wrong: it would have allowed non-approved answers to be returned as long the search term matched the question text.

Ok! We've mastered the `ManyToOne` relationship, which is actually the same as the `OneToMany` relationship. We got two for one! That means that there are only two more relationships to learn about: `OneToOne` and `ManyToMany`. Except... that's not true: we really only have *one* more relationship to learn about. Next: we'll discover that there are really only *two* types of relationships, not four.

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

Officially, there are *four* types of relations in Doctrine: `ManyToOne`, `OneToMany`, `OneToOne` and `ManyToMany`. But... that's kind of a lie! In reality, there are only *two* types.

Let me explain. We already know that a `ManyToOne` relationship and a `OneToMany` relationship are really just the *same* relationship seen from the two different sides. So that means that instead of four different types of relations, there are really only three.

OneToOne is ManyToOne in Disguise

But... the `OneToOne` relationship is... kind of *not* a different relationship.

For example, you decide to add a `OneToOne` relationship from a `User` entity to a `Profile` entity... which would hold *more* data about that user. If you did this, in the database, your `user` table would have a `profile_id` foreign key column. But wait: isn't that *exactly* what a `ManyToOne` relationship looks like?

Yup! In reality, a `OneToOne` relationship is the same as a `ManyToOne`, except that Doctrine puts a unique key on the `profile_id` column to prevent a single profile from being being linked to multiple users. But... that's really the only difference!

And, by the way, I try to avoid `OneToOne` relationships. Instead of splitting user data across two different entities, I tend to put it all in one class to reduce complexity. Splitting into two different entities *could* help performance, but I think it's almost always more of a bother than a help. Wait until you have *real* performance problems and *then* debug it.

Generating the Tag Entity

Anyways, this means that `ManyToOne`, `OneToMany` *and* `OneToOne` are all... really just the same relationship! That leaves only `ManyToMany`, which *is* a bit different. So let's build one!

Imagine that every `Question` can get be tagged with text descriptors.

In order to store tags in the database, let's make a `Tag` entity. Spin over to your console and run:

```
$ symfony console make:entity
```

Call the new entity `Tag`... and it's going to be *real* simple: a single field called `name` that will be a `string` type, `255` length, not nullable. Hit enter again to finish up.

Before I generate that migration, open up the new `Tag` class...

```
42 lines | src/Entity/Tag.php
... lines 1 - 2
namespace App\Entity;

use App\Repository\TagRepository;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass=TagRepository::class)
 */
class Tag
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

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

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

    public function getName(): ?string
    {
        return $this->name;
    }

    public function setName(string $name): self
    {
        $this->name = $name;

        return $this;
    }
}
```

because you *know* that I love to use `TimestampableEntity`.

```
45 lines | src/Entity/Tag.php
... lines 1 - 6
use Gedmo\Timestampable\Traits\TimestampableEntity;
... lines 8 - 11
class Tag
{
    use TimestampableEntity;

... lines 16 - 43
}
```

We could also add a `slug` column if we wanted to be able to go to a nice url like `/tags/{slug}` to show all the questions related a slug. I *won't* do that... mostly because we showed how to do that in the last tutorial: how to generate a `slug` automatically from some other property.

Ok: we now have a functional `Tag` entity. So let's generate a migration for it:

```
$ symfony console make:migration
```

Beautiful! Go give it a quick peek to make sure nothing funny snuck in. Nope! That looks boring: `CREATE TABLE tag` with `id`, `name` and the date fields.

45 lines | src/Entity/Tag.php

```
// ... lines 1 - 6
use Gedmo\Timestampable\Traits\TimestampableEntity;
// ... lines 8 - 11
class Tag
{
    use TimestampableEntity;

// ... lines 16 - 43
}
```

Go run it:

```
$ symfony console doctrine:migrations:migrate
```

Awesomesauce. So let's think about our goal: each `Question` could have many tags... and each `Tag` could be related to many questions. In other words, this is a *many* to *many* relationship. Next: let's generate that and see what it looks like!

Chapter 17: ManyToMany Relation

Each `Question` is going to be able to have many tags: we're going to render the list of tags below each question. But *then*, each tag could *also* be related to many different questions. OMG! We need a `ManyToMany` relationship! But don't take my word for it, let's pretend that we haven't figured *which* relationship we need yet: we just know that we want to be able to set multiple `Tag` objects onto a `Question` object. In other words, we want our `Question` class to have a *tags* property. Let's add that! Find your terminal and run:

```
$ symfony console make:entity
```

For which entity to edit, we could actually choose `Question` or `Tag`... it won't make much difference. But in my mind, I want to edit the `Question` entity in order to add a new property called `tags` to it. Once again, use the fake type called `relation` to activate the relationship wizard.

Okay: what class should this entity be related to? We want to relate to the `Tag` entity. And just like before, we see a nice table describing the different relationship types. If you focus on `ManyToMany`, it says:

> Each question can have many `Tag` objects and each `Tag` can also relate to many `Question` objects.

That describes our situation perfectly. Answer `ManyToMany`. Next, it asks a familiar question:

> Do we want to add a new property to `Tag` so that we can access or update `Question` objects from it?

It's basically saying:

> Hey! Would it be useful to have a `$tag->getQuestions()` method?

I'm not *so* sure that it *would* be useful... but let's say yes: it doesn't hurt anything. This will cause it to generate the *other* side of the relationship: we'll see that code in a minute. What should the property be called inside `Tag`? `questions` sounds perfect.

And... we're done! Hit enter to exit the wizard... and let's go check out the entities! Start in `Question`. Awesome. No surprise: it added a new `$tags` property, which will hold a *collection* of `Tag` objects. And as we mentioned before, whenever you have a relationship that holds a "collection" of things - whether that's a collection of answers or a collection of tags - in the `__construct` method, you need to initialize it to an `ArrayCollection`. That's taken care of for us.

```
 216 lines | src/Entity/Question.php
... lines 1 - 16
class Question
{
... lines 19 - 59
    /**
     * @ORM\ManyToMany(targetEntity=Tag::class, inversedBy="questions")
     */
    private $tags;

    public function __construct()
    {
... line 67
        $this->tags = new ArrayCollection();
    }
... lines 70 - 214
}
```

Above the property, we have a `ManyToMany` to tags... and if you scroll to the bottom of the class, we have `getTags()`, `addTag()` and `removeTag()` methods.

```
 216 lines | src/Entity/Question.php
... lines 1 - 16
class Question
{
... lines 19 - 59
    /**
     * @ORM\ManyToMany(targetEntity=Tag::class, inversedBy="questions")
     */
    private $tags;

    public function __construct()
    {
... line 67
        $this->tags = new ArrayCollection();
    }
... lines 70 - 191
    /**
     * @return Collection|Tag[]
     */
    public function getTags(): Collection
    {
        return $this->tags;
    }

    public function addTag(Tag $tag): self
    {
        if (!$this->tags->contains($tag)) {
            $this->tags[] = $tag;
        }

        return $this;
    }

    public function removeTag(Tag $tag): self
    {
        $this->tags->removeElement($tag);

        return $this;
    }
}
```

If you're thinking that this looks a *lot* like the code generated for a `OneToMany` relationship, you're right!

Now let's check out the `Tag` class. Things here... well... they look pretty much the same! We have a `$questions` property... which is initialized to an `ArrayCollection`. It is *also* a `ManyToMany` and points to the `Question` class.

```
84 lines | src/Entity/Tag.php
... lines 1 - 13
class Tag
{
... lines 16 - 29
    /**
     * @ORM\ManyToMany(targetEntity=Question::class, mappedBy="tags")
     */
    private $questions;

    public function __construct()
    {
        $this->questions = new ArrayCollection();
    }
... lines 39 - 82
}
```

And below, it has `getQuestions()`, `addQuestion()` and `removeQuestion()`.

84 lines | src/Entity/Tag.php

```
... lines 1 - 13
class Tag
{
... lines 16 - 29
    /**
     * @ORM\ManyToMany(targetEntity=Question::class, mappedBy="tags")
     */
    private $questions;

    public function __construct()
    {
        $this->questions = new ArrayCollection();
    }
... lines 39 - 56
    /**
     * @return Collection|Question[]
     */
    public function getQuestions(): Collection
    {
        return $this->questions;
    }

    public function addQuestion(Question $question): self
    {
        if (!$this->questions->contains($question)) {
            $this->questions[] = $question;
            $question->addTag($this);
        }

        return $this;
    }

    public function removeQuestion(Question $question): self
    {
        if ($this->questions->removeElement($question)) {
            $question->removeTag($this);
        }

        return $this;
    }
}
```

Now that we've seen what this look like in PHP, let's generate the migration:

```
$ symfony console make:migration
```

Once it finishes... spin over and open that new file. And... woh! It creates a brand new *table*? It's called `question_tag` ... and it has only *two* columns: a `question_id` foreign key column and a `tag_id` foreign key column. That's it.

```
 34 lines | migrations/Version20210907185958.php
... lines 1 - 4
namespace DoctrineMigrations;

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

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20210907185958 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE TABLE question_tag (question_id INT NOT NULL, tag_id INT NOT NULL, INDEX IDX_339D56FB1
        $this->addSql('ALTER TABLE question_tag ADD CONSTRAINT FK_339D56FB1E27F6BF FOREIGN KEY (question_id) REF
        $this->addSql('ALTER TABLE question_tag ADD CONSTRAINT FK_339D56FBBAD26311 FOREIGN KEY (tag_id) REFEREN
    }

    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('DROP TABLE question_tag');
    }
}
```

And... this makes sense! Even outside of Doctrine, *this* is how you build a `ManyToMany` relationship: you create a "join table" that keeps track of which tags are related to which questions.

With Doctrine, it's no different... except that Doctrine is going to handle the heavy lifting of inserting and removing records to and from this table *for* us. We'll see that in a minute.

But before I forget, head back to your terminal and run this migration:

```
$ symfony console doctrine:migrations:migrate
```

Next: let's see our relationship in action, by relating questions and tags in PHP and watching Doctrine automatically inserts rows into the join table.

Chapter 18: Saving Items in a ManyToMany Relation

We just successfully generated a ManyToMany relationship between `Question` and `Tag`... and we even made and executed the migration.

Now let's see how we can relate these objects in PHP. Open up `src/DataFixtures/AppFixtures.php`. We're going to create a couple of objects by hand. Start with `$question = QuestionFactory::createOne()` to create a question - the lazy way - using our factory. Then I'll paste in some code that creates two `Tag` objects for some very important topics to my 4 year old son.

49 lines | src/DataFixtures/AppFixtures.php

```
// ... lines 1 - 6
use App\Entity\Tag;
// ... lines 8 - 12
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 17 - 35
        $question = QuestionFactory::createOne();

        $tag1 = new Tag();
        $tag1->setName('dinosaurs');
        $tag2 = new Tag();
        $tag2->setName('monster trucks');
// ... lines 42 - 45
        $manager->flush();
    }
}
```

To actually *save* these, we need to call `$manager->persist($tag1)` and `$manager->persist($tag2)`.

49 lines | src/DataFixtures/AppFixtures.php

```
// ... lines 1 - 6
use App\Entity\Tag;
// ... lines 8 - 12
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 17 - 35
        $question = QuestionFactory::createOne();

        $tag1 = new Tag();
        $tag1->setName('dinosaurs');
        $tag2 = new Tag();
        $tag2->setName('monster trucks');

        $manager->persist($tag1);
        $manager->persist($tag2);

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

Relating the Objects

Awesome! Right now, this will create one new `Question` and two new tags... but they won't be related in the database. So how *do* we relate them? Well, don't think at *all* about the join table that was created... you really want to pretend like that doesn't even exist. Instead, like we've done with the other relationship, *just* think:

> I want to relate these two `Tag` objects to this `Question` object.

Doing *that* is pretty simple: `$question->addTag($tag1)` and `$question->addTag($tag2)`.

52 lines | src/DataFixtures/AppFixtures.php

```
// ... lines 1 - 12
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 17 - 35
        $question = QuestionFactory::createOne();

        $tag1 = new Tag();
        $tag1->setName('dinosaurs');
        $tag2 = new Tag();
        $tag2->setName('monster trucks');

        $question->addTag($tag1);
        $question->addTag($tag2);
// ... lines 45 - 49
    }
}
```

That's it! Let's try this thing! Reload the fixtures:

```
$ symfony console doctrine:fixtures:load
```

And... no errors! Check the database:

```
$ symfony console doctrine:query:sql 'SELECT * FROM tag'
```

No surprise: we have two tags in this table. Now `SELECT * FROM question_tag` - the *join* table.

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

And... yes! This has two rows! The first relates the first tag to the question and the second relates the *second* tag to that same question. How cool is that? We simply relate the objects in PHP and *Doctrine* handles inserting the rows into the join table.

If we saved all of this stuff and then, down here, said `$question->removeTag($tag1)` and saved again, this would cause Doctrine to *delete* the first row in that table. All of the inserting and deleting happens automatically.

Owning vs Inverse on a ManyToMany

By the way, like with *any* relationship, a `ManyToMany` has an *owning* side and an inverse side. Because we originally modified the `Question` entity and added a `$tags` property, *this* is the owning side.

In a `ManyToOne` and `OneToMany` relationship, the owning side is *always* the `ManyToOne`... because that's the entity where the foreign key column exists, like `question_id` on the `answer` table.

But a `ManyToMany` is a bit different: you get to *choose* which side is the owning side. Because we decided to update the `Question` entity when we ran `make:entity`, that command set up *this* class to be the owning side. The way you know is that it points to the *other* side by saying `inversedBy=""`. So it's pointing to the *other* side of the relationship as the *inverse* side.

```
216 lines | src/Entity/Question.php
... lines 1 - 16
class Question
{
... lines 19 - 59
    /**
     * @ORM\ManyToMany(targetEntity=Tag::class, inversedBy="questions")
     */
    private $tags;
... lines 64 - 214
}
```

Then, over in `Tag`, this is the inverse side. And you can see that it says `mappedBy="tags"`. This says:

> The owning side - or "mapped side" - is the `tags` property over in the `Question` entity.

```
84 lines | src/Entity/Tag.php
... lines 1 - 13
class Tag
{
... lines 16 - 29
    /**
     * @ORM\ManyToMany(targetEntity=Question::class, mappedBy="tags")
     */
    private $questions;
... lines 34 - 82
}
```

But... remember: this distinction isn't *that* important. *Technically* speaking, when we want to relate a `Tag` and `Question`, the only way to do that is by setting the owning side: setting the `$tags` property on `Question`.

So let's do an experiment: change the code to be `$tag1->addQuestion($question)` and `$tag2->addQuestion($question)`.

```
52 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 12
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 17 - 42
        $tag1->addQuestion($question);
        $tag2->addQuestion($question);
... lines 45 - 49
    }
}
```

So we're now setting the *inverse* side of the relationship *only*. In theory, this should *not* save correctly. But let's try it: reload the fixtures.

```
$ symfony console doctrine:fixtures:load
```

[Foundry Proxy Objects]()

Ah! This error is unrelated: it's from Foundry: it says that `$tag->addQuestion()` argument `one` should be a `Question` object, but it received a `Proxy` object.

When you create an object with Foundry, like up here, it actually returns a `Proxy` object that *wraps* the true `Question` object. It doesn't normally matter, but if you start mixing Foundry code with non-Foundry code, sometimes you can get this error. To fix it, add `->object()`.

52 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 12
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 17 - 35
        $question = QuestionFactory::createOne()->object();
... lines 37 - 49
    }
}
```

This will now be a *pure* `Question` object.

Anyways, reload the fixtures again:

```
$ symfony console doctrine:fixtures:load
```

And... it works. More importantly, if we query the join table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

We still have two rows! That means that we *were* able to relate `Tag` and `Question` object by setting only the *inverse* side of the relation... which is exactly the opposite of what I said.

But... this only works because our entity code is smart. Look at the `Tag` class... and go down to the `addQuestion()` method.

84 lines | src/Entity/Tag.php

```
... lines 1 - 13
class Tag
{
... lines 16 - 64
    public function addQuestion(Question $question): self
    {
        if (!$this->questions->contains($question)) {
            $this->questions[] = $question;
            $question->addTag($this);
        }

        return $this;
    }
... lines 74 - 82
}
```

Yep, it calls `$question->addTag($this)`. We saw this *exact* same thing with the `Question` `Answer` relationship. When we call, `addQuestion()`, *it* handles setting the owning side of the relationship. *That* is why this saved. Watch: if we comment this line out...

```
 84 lines | src/Entity/Tag.php
... lines 1 - 13
class Tag
{
... lines 16 - 64
    public function addQuestion(Question $question): self
    {
        if (!$this->questions->contains($question)) {
            $this->questions[] = $question;
            //$question->addTag($this);
        }

        return $this;
    }
... lines 74 - 82
}
```

reload the fixtures...

```
$ symfony console doctrine:fixtures:load
```

... and query the join table, it's empty! We *do* have 2 `Tag` objects... but they are not related to any questions in the database because we never set the owning side of the relationship. So... let's put that code back.

```
 84 lines | src/Entity/Tag.php
... lines 1 - 13
class Tag
{
... lines 16 - 64
    public function addQuestion(Question $question): self
    {
        if (!$this->questions->contains($question)) {
            $this->questions[] = $question;
            $question->addTag($this);
        }

        return $this;
    }
... lines 74 - 82
}
```

Next: let's use Foundry to create a bunch of `Tag` objects and randomly relate them to questions.

Chapter 19: Handling ManyToMany in Foundry

Now that we've seen how we can relate `Tag` objects and `Question` objects, let's use Foundry to properly create some fresh `Tag` fixture data. Start by generating the `Tag` factory

```
$ symfony console make:factory
```

And... we want to generate the one for `Tag`. Beautiful!

61 lines | src/Factory/TagFactory.php

```php
// ... lines 1 - 2
namespace App\Factory;

use App\Entity\Tag;
use App\Repository\TagRepository;
use Zenstruck\Foundry\RepositoryProxy;
use Zenstruck\Foundry\ModelFactory;
use Zenstruck\Foundry\Proxy;

// ... lines 11 - 28
final class TagFactory extends ModelFactory
{
    public function __construct()
    {
        parent::__construct();

        // TODO inject services if required (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#factori
    }

    protected function getDefaults(): array
    {
        return [
            // TODO add your default values here (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#r
            'name' => self::faker()->text(),
            'createdAt' => null, // TODO add DATETIME ORM type manually
            'updatedAt' => null, // TODO add DATETIME ORM type manually
        ];
    }

    protected function initialize(): self
    {
        // see https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#initialization
        return $this
            // ->afterInstantiate(function(Tag $tag) {})
        ;
    }

    protected static function getClass(): string
    {
        return Tag::class;
    }
}
```

Go check out that class: `src/Factory/TagFactory.php`. Remember: our only job is to make sure that we have good default values for all of the required properties. For `name`, instead of using `text()`, we can use `->word()`. And like I've done before, I'm going to remove `updatedAt`... but set `createdAt` to `self::faker->dateTimeBetween('-1 year')`.

```
 59 lines | src/Factory/TagFactory.php
// ... lines 1 - 28
final class TagFactory extends ModelFactory
{
// ... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
            'name' => self::faker()->word(),
            'createdAt' => self::faker()->dateTimeBetween('-1 year'),
        ];
    }
// ... lines 45 - 57
}
```

Now that we have this, at the top of the fixtures class, we can create 100 random tags with `TagFactory::createMany(100)`. I *love* doing that!

```
 44 lines | src/DataFixtures/AppFixtures.php
// ... lines 1 - 9
use App\Factory\TagFactory;
// ... lines 11 - 13
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        TagFactory::createMany(100);
// ... lines 19 - 41
    }
}
```

Below, for these 20 published questions, I want to relate *each* one to a random number of tags. To do that, pass a second argument: this is an array of attribute *overrides*. Let's think: the property we want to set on each `Question` object is called `tags`. So pass `tags => ` some collection of tags. To get that collection, let's pass this a *new* function: `TagFactory::randomRange(0, 5)`.

```
 44 lines | src/DataFixtures/AppFixtures.php
// ... lines 1 - 13
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        TagFactory::createMany(100);

        $questions = QuestionFactory::createMany(20, [
            'tags' => TagFactory::randomRange(0, 5),
        ]);
// ... lines 23 - 41
    }
}
```

This is pretty cool: it will return 0 to 5 random tags from the database, giving each question a different *number* of random tags. There *is* a small problem with this code... and maybe you see it... but let's try it anyways.

Spin over and reload the fixtures:

```
$ symfony console doctrine:fixtures:load
```

Awesome. And now check the database. I'll first say:

```
$ symfony console doctrine:query:sql 'SELECT * FROM tag'
```

Yep! We *do* have 100 tags. Actually, we have 102 tags. Go the bottom of the fixtures class and delete our code from earlier: we don't need that anymore.

Anyways, this created 100 tags. Now check the join table: `SELECT * FROM question_tag`

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

And... it *did* work... though if we're assigning 0 to 5 tags to *each* of the 20 questions... 20 *total* seems a little low. And... it *is*! Look closely: every row is related to the *same* tag!

Of course! I keep making this mistake! Because we're passing an array of attributes, the `TagFactory::randomRange()` method is only called *once*. So in my situation, this returned *one* random `Tag` ... and then assigned that one `Tag` to all 20 questions... which is why we ended up with 20 rows.

We know the fix: change this to a callback... that *returns* that array.

46 lines | src/DataFixtures/AppFixtures.php

```php
// ... lines 1 - 13
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 18 - 19
        $questions = QuestionFactory::createMany(20, function() {
            return [
                'tags' => TagFactory::randomRange(0, 5),
            ];
        });
// ... lines 25 - 43
    }
}
```

Try it again:

```
$ symfony console doctrine:fixtures:load
```

And then query the join table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

Sweet! 41 results seems right! And we can see that each question is related to different tags... and a different *number* of tags: some only have one, this one has 4. So, it's perfect.

Next: each published question is now related to 0 to 5 tags. Time to render the ManyToMany relationship on the frontend *and* learn how to join across it in a query.

Chapter 20: Joining Across a ManyToMany

Each published question now relates to 0 to 5 random tags. Over on the homepage, let's render the list of tags for each question under its vote count.

Rendering the ManyToMany Relation

And I'm happy to report that *using* a ManyToMany relationship... isn't anything special. Open up the template for this page: `templates/question/homepage.html.twig`. Down here... right after the vote string, add `{% for tag in question.tags %}`.

```
53 lines | templates/question/homepage.html.twig
... lines 1 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in questions %}
        <div class="col-12 mb-3">
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
                <div class="q-container p-4">
                    <div class="row">
                        <div class="col-2 text-center">
... lines 23 - 26
                            {% for tag in question.tags %}
... line 28
                            {% endfor %}
                        </div>
... lines 31 - 38
                    </div>
                </div>
... lines 41 - 45
            </div>
        </div>
        {% endfor %}
    </div>
</div>
... lines 51 - 53
```

It's that easy: our `Question` object has a `tags` property that will return a collection of all the related `Tag` objects. Behind the scenes, to *get* this data, Doctrine will need to query across the join table *and* the `tag` table. But... we don't really care about that! We just get to say `question.tags` and that returns all the `Tag` objects for this `Question`. It's really no different than how we could say `question.answers` to get all of the answers for a question.

So inside the loop, we're dealing with a `Tag` object. Add a span, print `{{ tag.name }}`... and then I'll give this a couple of classes to make it look cool.

```
53 lines | templates/question/homepage.html.twig
... lines 1 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in questions %}
        <div class="col-12 mb-3">
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
                <div class="q-container p-4">
                    <div class="row">
                        <div class="col-2 text-center">
... lines 23 - 26
                            {% for tag in question.tags %}
                                <span class="badge rounded-pill bg-light text-dark">{{ tag.name }}</span>
                            {% endfor %}
                        </div>
... lines 31 - 38
                    </div>
                </div>
... lines 41 - 45
            </div>
        </div>
        {% endfor %}
    </div>
</div>
... lines 51 - 53
```

Let's try this thing! Refresh and... done! We're *awesome*.

Joining in a Query with a ManyToMany

But check out the queries on this page: there are 41! Yikes! If you open this up, we have another N+1 problem. This first query is from the `question` table: it returns all of the questions. This second query selects the `tag` data for a *specific* `question` ... this is triggered when the `question.tags` line is executed. Then... if you keep looking down - skip this one - we have that same query for the next question... and the same query for the next... and the next. We *also* have extra queries for counting the answers for each question, but ignore those right now.

So... when we render the tags for each question, we have the N+1 query problem! When we had this problem before on the answers page, we fixed it inside of `AnswerRepository` ... by joining across the `question` relationship and then *selecting* the `question` data. We can do the *exact* same thing again.

The controller for this page is `src/Controller/QuestionController.php` ... it's the `homepage()` method.

```
82 lines | src/Controller/QuestionController.php
... lines 1 - 15
class QuestionController extends AbstractController
{
... lines 18 - 30
    public function homepage(QuestionRepository $repository)
    {
        $questions = $repository->findAllAskedOrderedByNewest();

        return $this->render('question/homepage.html.twig', [
            'questions' => $questions,
        ]);
    }
... lines 39 - 80
}
```

To fetch the questions, we're already calling a custom repository method called `findAllAskedOrderedByNewest()` .

Let's go find that: open up `QuestionRepository` . Here it is. So far, it's pretty simple: it makes sure that the askedAt is not null - that's this addIsAskedQueryBuilder() part - and then orders the newest first.

```
58 lines | src/Repository/QuestionRepository.php
... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
... lines 18 - 25
    public function findAllAskedOrderedByNewest()
    {
        return $this->addIsAskedQueryBuilder()
            ->orderBy('q.askedAt', 'DESC')
            ->getQuery()
            ->getResult()
        ;
    }

    private function addIsAskedQueryBuilder(QueryBuilder $qb = null): QueryBuilder
    {
        return $this->getOrCreateQueryBuilder($qb)
            ->andWhere('q.askedAt IS NOT NULL');
    }
... lines 40 - 56
}
```

To fix the N+1 problem, we need to add a join. And *this* is where things get interesting. In the database, we need to join from question to question_tag ... and then join from question_tag over to tag. So we actually need *two* joins.

But in Doctrine, we get to *pretend* like that join table doesn't exist: Doctrine wants us to pretend that there is a *direct* relationship from question to tag. What I mean is, to do the join, all we need is ->leftJoin() - because we want to get the *many* tags for this question - q.tags, tag.

```
60 lines | src/Repository/QuestionRepository.php
... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
... lines 18 - 25
    public function findAllAskedOrderedByNewest()
    {
        return $this->addIsAskedQueryBuilder()
            ->orderBy('q.askedAt', 'DESC')
            ->leftJoin('q.tags', 'tag')
... lines 31 - 33
        ;
    }
... lines 36 - 58
}
```

That's it. We reference the tags property on question ... and let *Doctrine* figure out how to join over to that. The second argument - tag - becomes the alias to the data on the tag table. We need that to select its data: addSelect('tag').

60 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
... lines 18 - 25
    public function findAllAskedOrderedByNewest()
    {
        return $this->addIsAskedQueryBuilder()
            ->orderBy('q.askedAt', 'DESC')
            ->leftJoin('q.tags', 'tag')
            ->addSelect('tag')
... lines 32 - 33
        ;
    }
... lines 36 - 58
}
```

So... yup! Joining across a `ManyToMany` relationship is *no* different than joining across a `ManyToOne` relationship: you reference the relation property and Doctrine does the heavy lifting.

Try it now. We have 41 queries and... when we refresh... yes! Down to 21! Open up the profiler and look at that first query... it's pretty awesome. It selects all of the `question` data... and then took care of left joining over to `question_tag`, left joining *again* over to `tag` and *then* selecting the tag data. *So* cool!

Next: the `question_tag` table - the join table - only has 2 columns: `question_id` and `tag_id`. What if we wanted to add more columns to this? Like a `taggedAt` date column? There's no entity class for this table... so is adding a 3rd or 4th column even possible? The answer is yes: but it *does* require some changes.

Chapter 21: ManyToMany... with Extra Fields on the Join Table?

The `ManyToMany` relationship is unique in Doctrine because *Doctrine* actually creates & manages a table - the join table - for us. This is the *only* time in Doctrine where we have a table *without* a corresponding entity class.

But what if we needed to add more *columns* to this table? Like a `tagged_at` `DateTime` column? Excellent question! And the answer is... that's not possible! I'm serious! But, it's by design. As soon as you need even *one* extra column on this join table, you need to *stop* using a `ManyToMany` relationship. Instead, you need to create an *entity* for the join table and manually relate that entity to `Question` and `Tag`.

Let's see what this looks like. But, it's actually easier to do this from the beginning than to try to refactor an existing `ManyToMany` relationship. So before you create a `ManyToMany`, try to think if you might need extra columns in the future. And if you *will* need them, *start* with the solution that we're about to see.

Undoing the ManyToMany

Ok, step 1: I'm actually going to hit the rewind button on our code and *remove* the `ManyToMany`. In `Question`, delete everything related to tags. So, the property, the constructor and the getter and setter methods.

Inside of `Tag`, do the same thing for questions: delete the methods and, on top, the property and the entire constructor.

So we *still* have a `Question` entity and a `Tag` entity... but they're no longer related.

Generating the Join Entity

Now we're going to put this relationship *back*, but with a new entity that represents the join table. Find your terminal and run:

```
$ symfony console make:entity
```

Let's call this entity `QuestionTag`, but if there's a more descriptive name for your situation, use that. This entity will have at *least* two properties: one for the relation to `Question` and another for the relation to `Tag`.

Start with the `question` property... and use the `relation` type to trigger the wizard. This will relate to the `Question` entity... and it's going to be a `ManyToOne`: each `QuestionTag` relates to one `Question` and each `Question` could have many `QuestionTag` objects.

Is the property allowed to be nullable? No... and then do we want to add a new property to `Question` so we can say `$question->getQuestionTags()`? That *probably* will be handy, so say yes. Call that property `$questionTags`. Finally, say "no" to orphan removal.

Cool! The other property - the `tag` property - will be exactly the same: a `ManyToOne`, related to `Tag`, say "no" for nullable and, in this case, I'm going to say "no" to generating the *other* side of the relationship. I'm doing this mostly so we can see an example of a relationship where only *one* side is mapped. But we also aren't going to need this shortcut method for what we're building. So say "no".

And... perfect! That is the *minimum* needed in the new `QuestionTag` entity: a `ManyToOne` relationship to `Question` and a `ManyToOne` relationship to `Tag`. So *now* we can start adding whatever *other* fields we want. I'll add `taggedAt`... and make this a `datetime_immutable` property that is not nullable in the database. Hit enter a one more time to finish the command.

Ok! Let's go check out the new class: `src/Entity/QuestionTag.php`. It looks... beautifully boring! It has a `question` property that's a `ManyToOne` to `Question`, a `tag` property that's a `ManyToOne` to `Tag` and a `taggedAt` date property.

78 lines | src/Entity/QuestionTag.php

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

use App\Repository\QuestionTagRepository;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass=QuestionTagRepository::class)
 */
class QuestionTag
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\ManyToOne(targetEntity=Question::class, inversedBy="questionTags")
     * @ORM\JoinColumn(nullable=false)
     */
    private $question;

    /**
     * @ORM\ManyToOne(targetEntity=Tag::class)
     * @ORM\JoinColumn(nullable=false)
     */
    private $tag;

    /**
     * @ORM\Column(type="datetime_immutable")
     */
    private $taggedAt;

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

    public function getQuestion(): ?Question
    {
        return $this->question;
    }

    public function setQuestion(?Question $question): self
    {
        $this->question = $question;

        return $this;
    }

    public function getTag(): ?Tag
    {
        return $this->tag;
    }

    public function setTag(?Tag $tag): self
    {
        $this->tag = $tag;

        return $this;
    }

    public function getTaggedAt(): ?\DateTimeImmutable
    {
        return $this->taggedAt;
    }

    public function setTaggedAt(\DateTimeImmutable $taggedAt): self
    {
        $this->taggedAt = $taggedAt;

        return $this;
    }
}
```

Inside `Question`... scroll all the way up. Because we also decided to map *this* side of the relationships, this has a `OneToMany` relationship to the join entity.

 222 lines | src/Entity/Question.php

```
... lines 1 - 16
class Question
{
... lines 19 - 59
    /**
     * @ORM\OneToMany(targetEntity=QuestionTag::class, mappedBy="question")
     */
    private $questionTags;

    public function __construct()
    {
... line 67
        $this->questionTags = new ArrayCollection();
    }
... lines 70 - 191
    /**
     * @return Collection|QuestionTag[]
     */
    public function getQuestionTags(): Collection
    {
        return $this->questionTags;
    }

    public function addQuestionTag(QuestionTag $questionTag): self
    {
        if (!$this->questionTags->contains($questionTag)) {
            $this->questionTags[] = $questionTag;
            $questionTag->setQuestion($this);
        }

        return $this;
    }

    public function removeQuestionTag(QuestionTag $questionTag): self
    {
        if ($this->questionTags->removeElement($questionTag)) {
            // set the owning side to null (unless already changed)
            if ($questionTag->getQuestion() === $this) {
                $questionTag->setQuestion(null);
            }
        }

        return $this;
    }
}
```

But there were *no* changes to the `Tag` entity, since we decided *not* to map the other side of *that* relationship.

Back in `QuestionTag`, before we generate the migration, let's give our `$taggedAt` a default value. Create a `public function __construct()` and, inside, say `$this->taggedAt = new \DateTimeImmutable()` which will default to "now".

```
 83 lines | src/Entity/QuestionTag.php
... lines 1 - 10
class QuestionTag
{
... lines 13 - 36
    public function __construct()
    {
        $this->taggedAt = new \DateTimeImmutable();
    }
... lines 41 - 81
}
```

How this Looks Different / the Same in the Database

Ok - migration time! At your terminal, make it:

```
$ symfony console make:migration
```

And then go open up the new file... cause this is really cool! It *looks* like there are a lot of queries to change from the old `ManyToMany` structure to our *new* structure.

```
 43 lines | migrations/Version20210907192236.php
... lines 1 - 9
/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20210907192236 extends AbstractMigration
{
... lines 15 - 19
    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE question_tag DROP FOREIGN KEY FK_339D56FB1E27F6BF');
        $this->addSql('ALTER TABLE question_tag DROP FOREIGN KEY FK_339D56FBBAD26311');
        $this->addSql('ALTER TABLE question_tag ADD id INT AUTO_INCREMENT NOT NULL, ADD tagged_at DATETIME NOT N
        $this->addSql('ALTER TABLE question_tag ADD CONSTRAINT FK_339D56FB1E27F6BF FOREIGN KEY (question_id) REF
        $this->addSql('ALTER TABLE question_tag ADD CONSTRAINT FK_339D56FBBAD26311 FOREIGN KEY (tag_id) REFEREN
    }
... lines 29 - 41
}
```

But look closer. We *already* had a `question_tag` table thanks to the `ManyToMany` relationship. So we don't need to drop that table and create a new one: all the migration needs to do is *tweak* it. It drops the `question_id` and `tag_id` foreign key constraint from the table... but then adds them back down here. So the first two lines and last two lines cancel each other out.

This means that the only *real* change is `ALTER TABLE question_tag` to add a true `id` auto-increment column and the `tagged_at` column. Yup, we just did a *massive* refactoring of our entity code - replacing the `ManyToMany` with a new entity and two new relationships - but in the database... we have almost the exact same structure! In reality, a `ManyToMany` relationship is just a shortcut that allows you to have the join table *without* needing to create an *entity* for it.

So now that we understand that, from the database's perspective not much is changing, let's run the migration to make those tweaks:

```
$ symfony console doctrine:migrations:migrate
```

And... it fails! Rut roo. Next: let's find out why this migration failed. And, more importantly, how we can fix it and safely *test* it so that we confidently know that it will *not* fail when we deploy to production.

Chapter 22: When a Migration Falls Apart

Our new migration file is pretty cool! We created an *entirely* new entity - `QuestionTag` - with a relationship to `Question` and a relationship to `Tag`. But this massive change in PHP didn't translate into much in the database: the migration basically adds new `id` and `taggedAt` columns to the `question_tag` table.

Unfortunately... when we *executed* that migration, it blew up in our face! The reason is that the `question_tag` table already has data in it! And so when we told the table to add a new `tagged_at` `DATETIME` column that can't be `NULL`... it didn't know what value to *use* for the existing rows in the table! And so... explosion!

If you haven't actually deployed your `question_tag` table to production, then this isn't a real problem... because, when you *do* finally deploy, this table won't have any data in it the moment that this executes. In that case, all you need to do is fix your local setup. You can do this dropping your database, recreating it... and running all of your migrations from the beginning. We'll see how to do that in a minute.

But... I want to pretend like our `question_tag` table *has* been deployed to the production and it *does* have data in it.. and I want to fix this migration so that it does *not* explode when we run it on production.

Fixing the Migration

The fix for the migration is fairly simple. When we add the `tagged_at` column, instead of saying `DATETIME NOT NULL`, say `DATETIME DEFAULT NOW()`.

```
43 lines | migrations/Version20210907192236.php
... lines 1 - 12
final class Version20210907192236 extends AbstractMigration
{
... lines 15 - 19
    public function up(Schema $schema): void
    {
... lines 22 - 24
        $this->addSql('ALTER TABLE question_tag ADD id INT AUTO_INCREMENT NOT NULL, ADD tagged_at DATETIME DEFAU
... lines 26 - 27
    }
... lines 29 - 41
}
```

This is a *temporary* change: it will allow us to add that new column and give the existing rows a default value. Then, in a separate migration that we'll create in a few minutes, we can *then* safely make that column `NOT NULL`.

How to Test a Half-Executed Migration?

Cool! But... now that we've fixed the migration, how can we run it again? Let's try the obvious:

```
$ symfony console doctrine:migrations:migrate
```

It fails again! But for a different reason: dropping the foreign key failed because that's already *gone*.

Here's the problem. When we *first* ran this migration, the top two lines *did* successfully execute. And then the *third* line failed. This means that our migration is in a strange state... a, sort of, "half" executed state.

If you're using PostgreSQL, this is *not* a problem. Each migration is wrapped in a transaction. This means that, if *any* of the queries fail, they will *all* be reverted. Unfortunately, while this works *great* in PostgreSQL, MySQL does *not* support that coolness. So if you *are* using PostgreSQL, you rock and the migration command we ran a minute ago *did* work for you.

But if you're using MySQL, then you're in *our* reality. To test this migration, we need to do a, sort of, "manual" rollback: we need to put our database back into the state it was *before* we ran this migration. Once we've done that, *then* we'll run this migration again to make sure it works.

At your terminal, run;

```
$ git status
```

Before we created the `QuestionTag` entity, I committed everything to `git`. Add the new changes we've done since then:

```
$ git add src migrations/
```

And then run

```
$ git status
```

again. Yup: everything is ready to be committed. But instead of committing, *remove* these changes with:

```
$ git stash
```

If you're not familiar with the `git stash` command, it removes all of the changes from your project and "stashes" them somewhere so we can put them back later. If we check the code... the migration is gone... and so is the new entity. Now that our code is back to its "old" state, we can reset the database.

Start by dropping it entirely:

```
$ symfony console doctrine:database:drop --force
```

And then re-recreate it:

```
$ symfony console doctrine:database:create
```

And *then* migrate:

```
$ symfony console doctrine:migrations:migrate
```

This executes all the migrations up to this point, which is back when we had the `ManyToMany` relationship. Finally, to mimic production - where we have data in the join table - run:

```
$ symfony console doctrine:fixtures:load
```

Perfect! *Now* bring back all of our changes by saying:

```
$ git stash pop
```

Awesome! Finally, we can *now* test the new migration:

```
$ symfony console doctrine:migrations:migrate
```

Migrating Again to add NOT NULL

This time... it works! It added that new column *without* failing. The only small problem is that, right now, in the database, the `tagged_at` column is *not* required: it *does allow* null... which is not what we want. But fixing this is easy: ask Doctrine to generate one more migration:

```
$ symfony console make:migration
```

This is really cool: it looked at the new `QuestionTag` entity, *realized* that the `tagged_at` column isn't set up correctly, and generated a new migration with `ALTER TABLE question_tag CHANGE tagged_at` to `NOT NULL`.

32 lines | migrations/Version20210907192620.php

```
// ... lines 1 - 12
final class Version20210907192620 extends AbstractMigration
{
// ... lines 15 - 19
    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE question_tag CHANGE tagged_at tagged_at DATETIME NOT NULL COMMENT \'(DC2Type:
    }
// ... lines 25 - 30
}
```

Run this:

```
$ symfony console doctrine:migrations:migrate
```

And... it works!

So refactoring the relationship between `Question` and `Tag` to include a new `QuestionTag` entity didn't *really* change the structure of the database... though this migration *did* cause a headache. However, in PHP, how we save and *use* this relationship *did* just change substantially. So next, let's update our fixtures to work with the new structure.

Chapter 23: QuestionTag Fixtures & DateTimeImmutable with Faker

We no longer have a `ManyToMany` relationship between `Question` and `Tag`. Instead, each `Question` has many `QuestionTag` objects and each `QuestionTag` object is related to one `Tag`. This means that setting and *using* this relation - the relationship between `Question` and `Tag` - just changed. Let's update the fixtures to *reflect* this.

Generating & Configuring the QuestionTag Factory

First, since we now have a `QuestionTag` entity, we are going to be creating and persisting `QuestionTag` objects directly. So let's generate a Foundry factory for it. At your terminal, run:

```
$ symfony console make:factory
```

And choose `QuestionTag`. Go open that up: `src/Factory/QuestionTagFactory.php`.

59 lines | src/Factory/QuestionTagFactory.php

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

use App\Entity\QuestionTag;
use App\Repository\QuestionTagRepository;
use Zenstruck\Foundry\RepositoryProxy;
use Zenstruck\Foundry\ModelFactory;
use Zenstruck\Foundry\Proxy;

... lines 11 - 28
final class QuestionTagFactory extends ModelFactory
{
    public function __construct()
    {
        parent::__construct();

        // TODO inject services if required (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#factorie
    }

    protected function getDefaults(): array
    {
        return [
            // TODO add your default values here (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#r
            'taggedAt' => self::faker()->datetime(),
        ];
    }

    protected function initialize(): self
    {
        // see https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#initialization
        return $this
            // ->afterInstantiate(function(QuestionTag $questionTag) {})
        ;
    }

    protected static function getClass(): string
    {
        return QuestionTag::class;
    }
}
```

In `getDefaults()`, our job, as usual, is to add all the required fields. Set `question` to `QuestionFactory::new()` and do the same thing for `tag`, setting that to `TagFactory::new()`.

60 lines | src/Factory/QuestionTagFactory.php

```
... lines 1 - 28
final class QuestionTagFactory extends ModelFactory
{
... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
            'question' => QuestionFactory::new(),
            'tag' => TagFactory::new(),
            'taggedAt' => self::faker()->datetime(),
        ];
    }
... lines 46 - 58
}
```

As a reminder, the `new()` method returns a `QuestionFactory` instance. So we're assigning the `question` attribute to a QuestionFactory object. We talked earlier about how this is better than calling createOne() because, when you set a relationship property to a *factory* instance, Foundry will *use* that to create the Question object... but only if it *needs* to.

Anyways with this setup, when we use this factory, it will create a brand new Question and a brand new Tag each time it makes a QuestionTag .

We can see this. Open up the fixtures class and say QuestionTagFactory::createMany(10) . I'm going to put a return statement here because some of the code below is currently broken.

51 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 9
use App\Factory\QuestionTagFactory;
... lines 11 - 14
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        TagFactory::createMany(100);

        QuestionTagFactory::createMany(10);

        return;
... lines 24 - 48
    }
}
```

Let's try this:

```
$ symfony console doctrine:fixtures:load
```

Handling DateTimeImmutable & Faker

And... it fails! But... for an unrelated reason. It says:

> QuestionTag::setTaggedAt() argument 1 must be a DateTimeImmutable instance, DateTime given.

This is subtle... and related to Faker. In Faker, when you say self::faker()->datetime() , that returns a DateTime object. No surprise!

But if you look at the QuestionTag entity, the taggedAt field is set to a datetime_immutable Doctrine type. This means that, instead of that property being a DateTime object, it will be a DateTimeImmutable object. Really... the same thing... except that DateTimeImmutable objects can't be changed.

The point is, the type-hint on the setter is DateTimeImmutable ... but we're trying to pass a DateTime instance... which isn't the same. The easiest way to fix this is to update the fixtures. Wrap the value with DateTimeImmutable::createFromMutable() ... which is a method that exists *just* for this situation.

60 lines | src/Factory/QuestionTagFactory.php

```php
// ... lines 1 - 28
final class QuestionTagFactory extends ModelFactory
{
// ... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
// ... lines 41 - 42
            'taggedAt' => \DateTimeImmutable::createFromMutable(self::faker()->datetime()),
        ];
    }
// ... lines 46 - 58
}
```

And if we reload the fixtures now...

```
$ symfony console doctrine:fixtures:load
```

No errors! Run:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

And... cool! We have 10 rows. Now query the question table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question'
```

And this *also* has 10 rows. That proves that, each time the factory creates a `QuestionTag`, it creates a brand new `Question` to relate to it.

So... this works... but it's not really what we want. Instead of creating *new* questions, we want to relate each `QuestionTag` to one of the *published* questions that we're creating in our fixtures.

Let's do that next, by doing some *seriously* cool stuff with Foundry.

Chapter 24: Doing Crazy things with Foundry & Fixtures

We *are* able to create new `QuestionTag` objects with its factory... but when we do that, it creates a brand *new* `Question` object for each new `QuestionTag`. That's... not what we want! I want what we had before... where we create our 20 published questions and relate *those* to a random set of tags.

Delete the `return` statement and the `QuestionTagFactory` line. Right now, this says:

> Create 20 questions. And, for each one, set the `tags` property to 5 random `Tag` objects.

Setting the questionTags Property on Question

The problem is that our `Question` entity doesn't *have* a `tags` property anymore: it now has a `questionTags` property. Okay. So let's change this to `questionTags`. We *could* set this to `QuestionTagFactory::randomRange()`. But that would require us to create those `QuestionTag` objects up here... which we *can't* do because we need the *question* object to exist first. Well, we *could* do that, but we would end up with extra questions that we don't really want.

By the way, we're about to see some *really* cool, really advanced stuff in Foundry. But at the end, I'm also going to show a simpler solution to creating the objects we need.

Foundry Passes the Outer Object to the Inner Factory

Anyways, set `questionTags` to `QuestionTagFactory::new()`. So, to an *instance* of this factory.

47 lines | src/DataFixtures/AppFixtures.php

```php
// ... lines 1 - 14
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        TagFactory::createMany(100);

        $questions = QuestionFactory::createMany(20, function() {
            return [
                'questionTags' => QuestionTagFactory::new(),
            ];
        });
// ... lines 26 - 44
    }
}
```

There *is* a problem with this... but it's *mostly* correct. And... it's kind of crazy! This tells Foundry to use this `QuestionTagFactory` instance to create a new `QuestionTag` object. *Normally* when we use `QuestionFactory`, it creates a *new* `Question` object. But in this case, that *won't* happen. Because we're calling this from *inside* the `QuestionFactory` logic, the `question` attribute that's passed to `QuestionTagFactory` will be overridden and set to the `Question` object that is currently being created by its factory.

In other words, this will *not* cause a new, extra `Question` to be created in the database. Instead, the new `QuestionTag` object will be related to whatever Question is currently being created. Foundry does this by *reading* the Doctrine relationship and smartly overriding the `question` attribute on `QuestionTagFactory`.

But... I *did* say that there was a problem with this. And... we'll see it right now:

```
$ symfony console doctrine:fixtures:load
```

This gives us a weird error from `PropertyAccessor` about how the `questionTags` attribute cannot be set on `Question`. The `PropertyAccessor` is what's used by Foundry to *set* each attribute onto the object. And while it's true that we don't have a `setQuestionTags()` method, we *do* have `addQuestionTag()` and `removeQuestionTag()`, which the accessor is smart enough to use.

So, the *real* problem here is simpler: `QuestionTagFactory::new()` says that we want to create a *single* `QuestionTag` and set it onto `questionTags`. But we need an *array*. *That* confused the property accessor. To fix this, add `->many()`.

This "basically" returns a factory instance that's now configured to create *multiple* objects. Pass 1, 5 to create anywhere from 1 to 5 `QuestionTag` objects.

47 lines | src/DataFixtures/AppFixtures.php

```php
// ... lines 1 - 14
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        TagFactory::createMany(100);

        $questions = QuestionFactory::createMany(20, function() {
            return [
                'questionTags' => QuestionTagFactory::new()->many(1, 5),
            ];
        });
// ... lines 26 - 44
    }
}
```

Try the fixtures again:

```
$ symfony console doctrine:fixtures:load
```

No errors! And if we `SELECT * FROM question`:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question'
```

We only have 25 rows: the correct amount! That's the 20 published... and the 5 unpublished. This proves that the `QuestionTagFactory` did *not* create new question objects like it did before: all the new question tags are related to these 20 questions. We can see that by querying: `SELECT * FROM question_tag`

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

60 rows seems about right. This is related to question 57, 57, 57, 57... then 56, 56 and then 55. So each question has a random number of question tags.

Overriding the tag Attribute

Unfortunately this line *is* still creating a new random `Tag` each time. Check the `tag` table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM tag'
```

We *want* there to be 100 rows... from the 100 in our fixtures. We don't want *extra* tags to be created down here. But... we get 160: 100 plus 1 more for each `QuestionTag` .

And... this make sense... thanks to the `getDefaults()` method.

The fix... is both nuts and simple: pass an array to `new()` to override the `tag` attribute. Set it to `TagFactory::random()` to grab `one` existing random `Tag` .

49 lines | src/DataFixtures/AppFixtures.php

```php
// ... lines 1 - 14
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 19 - 20
        $questions = QuestionFactory::createMany(20, function() {
            return [
                'questionTags' => QuestionTagFactory::new([
                    'tag' => TagFactory::random()
                ])->many(1, 5),
            ];
        });
// ... lines 28 - 46
    }
}
```

Reload the fixtures again:

```
$ symfony console doctrine:fixtures:load
```

And query the tag table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM tag'
```

We're back to 100 tags! But... I made a mistake... and maybe you saw it. Check out the `question_tag` table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

These last two are both related to question id 82... actually the last 3. And that's fine: each `Question` will be related to 1 to 5 question tags. The problem is that all of these are *also* related to the same `Tag` !

In the fixtures, each time a `Question` is created, it executes this callback. So it's executed 20 times. But then, when the 1 to 5 `QuestionTag` object are created, `TagFactory::random()` is only called *once*... meaning that the *same* `Tag` is used for each of the 1 to 5 question tags.

Yup, this is the *same* problem we've seen multiple times before... I'm trying to make this mistake a *ton* of times in this tutorial, so that you *never* experience it.

Refactor this to use a callback.

```
51 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 14
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
... lines 19 - 20
        $questions = QuestionFactory::createMany(20, function() {
            return [
                'questionTags' => QuestionTagFactory::new(function() {
                    return [
                        'tag' => TagFactory::random(),
                    ];
                })->many(1, 5),
            ];
        });
... lines 30 - 48
    }
}
```

Then, reload the fixtures:

```
$ symfony console doctrine:fixtures:load
```

And check the `question_tag` table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

Yes! These last 2 have the same question id... but they have *different* tag ids. Mission accomplished! And... this is probably the most *insane* thing that you'll ever do with Foundry. This says:

> Create 20 questions. For each question, the `questionTags` property should be set to 1 to 5 new `QuestionTag` objects... except where the `question` attribute is overridden and set to the new `Question` object. Then, for each `QuestionTag` , select a random `Tag` .

Congratulations, you now have a PhD in Foundry!

The Simpler Solutions

But... you do *not* need to make it this complicated! I did this mostly for the *pursuit of learning*! To show off some advanced stuff you can do with Foundry.

An easier way to do this would be to create 100 tags, 20 published questions and *then*, down here, use the `QuestionTagFactory` to create, for example, 100 `QuestionTag` objects where each one is related to a random `Tag` and also a random `Question` .

```
50 lines | src/DataFixtures/AppFixtures.php
// ... lines 1 - 14
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
// ... lines 19 - 22
        QuestionTagFactory::createMany(100, function() {
            return [
                'tag' => TagFactory::random(),
                'question' => QuestionFactory::random(),
            ];
        });
// ... lines 29 - 47
    }
}
```

Then, above, when we create the Questions... we can just create normal, boring `Question` objects... because the `QuestionTag` stuff is handled below.

```
50 lines | src/DataFixtures/AppFixtures.php
// ... lines 1 - 14
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        TagFactory::createMany(100);

        $questions = QuestionFactory::createMany(20);

        QuestionTagFactory::createMany(100, function() {
            return [
                'tag' => TagFactory::random(),
                'question' => QuestionFactory::random(),
            ];
        });
// ... lines 29 - 47
    }
}
```

If we try this:

```
$ symfony console doctrine:fixtures:load
```

No errors. And if you look inside the `question_tag` table:

```
$ symfony console doctrine:query:sql 'SELECT * FROM question_tag'
```

We get 100 question tags, each related to a random `Question` and a random `Tag`. It's not *exactly* the same as we had before, but it's probably close enough, and *much* simpler.

Next: let's fix the frontend *and* our JOIN to use the refactored `QuestionTag` relationship.

Chapter 25: JOINing Across Multiple Relationships

We decided to change the relationship between `Question` and `Tag` from a true `ManyToMany` to a relationship where we have an entity in between that allows us to add more fields to the join table.

In the database... this didn't change much: we have the same join table and foreign keys as before. But in PHP, it *did* change things. In `Question`, instead of a `$tags` property - which returned a collection of `Tag` objects - we now have a `$questionTags` property that returns a collection of `QuestionTag` objects. This change almost *certainly* broke our frontend.

We're only rendering the tags on the homepage.... so let's open up that template `templates/question/homepage.html.twig`. Here it is: for `tag in question.tags`. That's not going to work anymore because there *is* no `tags` property. Though, if you want to be clever, you *could* create a `getTags()` method that loops over the `questionTags` property and returns an array of the related `Tag` objects.

Or... you can fix it here to use `questionTag in questionTags`. Then say `questionTag.tag` to reach across that relationship.

53 lines | templates/question/homepage.html.twig

```
... lines 1 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in questions %}
        <div class="col-12 mb-3">
            <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
                <div class="q-container p-4">
                    <div class="row">
                        <div class="col-2 text-center">
... lines 23 - 26
                            {% for questionTag in question.questionTags %}
                                <span class="badge rounded-pill bg-light text-dark">{{ questionTag.tag.name }}</span>
                            {% endfor %}
                        </div>
... lines 31 - 38
                    </div>
                </div>
... lines 41 - 45
            </div>
        </div>
        {% endfor %}
    </div>
</div>
... lines 51 - 53
```

So still fairly straightforward... just a bit more code to go across both relationships.

Let's refresh and see what happens. And... whoa!

> Semantical error: near `tag WHERE q.askedAt`: Class `Question` has no association named `tags`.

So... that sounds like a query error... but let's look down the stack trace. Yup! It's coming from `QuestionRepository`.

Joining Across Two Entities

Go open that up: `src/Repository/QuestionRepository.php`... here it is. To solve the N+1 problem, we joined directly across the previous q.tags relationship. Now we're going to need *two* joins to get to the tag table.

No problem: change q.tags to q.questionTags and alias that to question_tag. Then do an inner join from QuestionTag to Tag - ->innerJoin('question_tag.tag') - and alias that to tag.

```
61 lines | src/Repository/QuestionRepository.php
... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
... lines 18 - 25
    public function findAllAskedOrderedByNewest()
    {
        return $this->addIsAskedQueryBuilder()
            ->orderBy('q.askedAt', 'DESC')
            ->leftJoin('q.questionTags', 'question_tag')
            ->innerJoin('question_tag.tag', 'tag')
            ->addSelect('tag')
            ->getQuery()
            ->getResult()
        ;
    }
... lines 37 - 59
}
```

Cool! And we're still selecting the tag data... so that looks good to me.

Refresh again and... another error! This one... is even more confusing.

> The parent object of entity result with alias tag was not found. The parent alias is question_tag.

This is trying to say that it doesn't like that we're selecting the tag data... but we're *not* selecting the question_tag data that's *between* Question and Tag. Doing that *is* legal in SQL... but it messes up how Doctrine creates the objects, so it doesn't allow it.

The solution is easy enough: select both. You can actually pass an array to addSelect() to select question_tag *and* tag.

```
61 lines | src/Repository/QuestionRepository.php
... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
... lines 18 - 25
    public function findAllAskedOrderedByNewest()
    {
        return $this->addIsAskedQueryBuilder()
            ->orderBy('q.askedAt', 'DESC')
            ->leftJoin('q.questionTags', 'question_tag')
            ->innerJoin('question_tag.tag', 'tag')
            ->addSelect('question_tag', 'tag')
            ->getQuery()
            ->getResult()
        ;
    }
... lines 37 - 59
}
```

Try it now. And... we're back! Woo! Check out what the query looks like... it's this big first one. So cool: we select from question left join to question_tag, inner join over to tag... and grab all of that data.

Okay team: there's just *one* last topic I want to cover... and, I admit, it's not *strictly* related to relations. Let's add pagination to our homepage.

Chapter 26: Pagination with Pagerfanta

I want to add *one* more Doctrine-specific feature to our site: pagination.

Right now, on the homepage, we're rendering *every* question on the site. That's... not very realistic. Instead, let's render 5 on each page with pagination links.

KnpPaginator and Pagerfanta

Doctrine *does* come with tools for pagination... but they're a little "low level". Fortunately, the Symfony ecosystem has two libraries that build on *top* of Doctrine's tools to make pagination a pleasure. They're called KnpPaginator and Pagerfanta.

Both of these are really good... and I have a hard time choosing between them. In our Symfony 4 Doctrine tutorial, we talked about KnpPaginator. So in *this* tutorial, let's explore Pagerfanta.

Installing PagerfantaBundle

Search for "pagerfanta bundle" to find a GitHub page under the "BabDev" organization. Scroll down a little and click into the documentation.

The PagerfantaBundle is a wrapper around a Pagerfanta *library* that holds most of the functionality. So the documentation is kind of split between the bundle and the library. Open the docs for the library in another tab so we have it handy... then come back and click "Installation".

Copy the "composer require" line, spin over to your terminal and get it:

```
$ composer require "babdev/pagerfanta-bundle:^3.6"
```

Let's see what that did:

```
$ git status
```

Ok: nothing too interesting... though it *did* automatically enable the new bundle.

Pagers Work with QueryBuilders

The controller for the homepage lives at `src/Controller/QuestionController.php` : the `homepage` action.

```
82 lines | src/Controller/QuestionController.php
... lines 1 - 15
class QuestionController extends AbstractController
{
... lines 18 - 27
    /**
     * @Route("/", name="app_homepage")
     */
    public function homepage(QuestionRepository $repository)
    {
        $questions = $repository->findAllAskedOrderedByNewest();

        return $this->render('question/homepage.html.twig', [
            'questions' => $questions,
        ]);
    }
... lines 39 - 80
}
```

We're calling this custom repository method, which returns an array of `Question` objects.

```
61 lines | src/Repository/QuestionRepository.php
... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
... lines 18 - 22
     /**
      * @return Question[] Returns an array of Question objects
      */
    public function findAllAskedOrderedByNewest()
    {
... lines 28 - 35
    }
... lines 37 - 59
}
```

The biggest difference when using a paginator is that *we* will no longer execute the query directly. Instead, our job will be to create a `QueryBuilder` and pass *that* to the paginator... which will then figure out which page we're on, set up the limit and offset parts of the query, and *then* execute it.

In other words, to prep for Pagerfanta, instead of returning an array of `Question` objects, we need to return a `QueryBuilder`. Rename the method to `createAskedOrderedByNewestQueryBuilder()` - good luck thinking of a longer name than that - and it will return a `QueryBuilder`.

```
56 lines | src/Repository/QuestionRepository.php
... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
... lines 18 - 21

    public function createAskedOrderedByNewestQueryBuilder(): QueryBuilder
    {
... lines 25 - 30
    }
... lines 32 - 54
}
```

Inside, all we need to do is remove `getQuery()` and `getResult()`.

```
 56 lines | src/Repository/QuestionRepository.php
 ... lines 1 - 15
class QuestionRepository extends ServiceEntityRepository
{
 ... lines 18 - 21

    public function createAskedOrderedByNewestQueryBuilder(): QueryBuilder
    {
        return $this->addIsAskedQueryBuilder()
            ->orderBy('q.askedAt', 'DESC')
            ->leftJoin('q.questionTags', 'question_tag')
            ->innerJoin('question_tag.tag', 'tag')
            ->addSelect('question_tag', 'tag')
        ;
    }
 ... lines 32 - 54
}
```

Back over in the controller, change this to `$queryBuilder` equals `$repository->createAskedOrderedByNewestQueryBuilder()` .

```
 82 lines | src/Controller/QuestionController.php
 ... lines 1 - 15
class QuestionController extends AbstractController
{
 ... lines 18 - 30
    public function homepage(QuestionRepository $repository)
    {
        $queryBuilder = $repository->createAskedOrderedByNewestQueryBuilder();
 ... lines 34 - 37
    }
 ... lines 39 - 80
}
```

We're ready!

Installing the ORM Pagerfanta Adapter

The next step is to create a `Pagerfanta` object... you can see how in the "Rendering Pagerfantas" section. This looks simple enough: create a new `Pagerfanta` and, because we're using Doctrine, create a new `QueryAdapter` and pass in our `$queryBuilder` .

Cool: `$pagerfanta = new Pagerfanta()` ... and `new QueryAdapter()` ... huh. PhpStorm isn't finding that class!

This is a... kind of weird... but also really cool thing about the Pagerfanta packages. Go back to library's documentation and click "Pagination Adapters". The Pagerfanta library can be used to paginate a *lot* of different things. Actually, click "Available Adapters".

For example, you can use Pagerfanta to paginate a relationship property - like `$question->getAnswers()` - via its `CollectionAdapter` . Or you can use it to paginate Doctrine DBAL queries... which is a lower-level way to use Doctrine. You can also paginate MongoDB or, if you're using the Doctrine ORM like we are, you can paginate with the `QueryAdapter` .

This is cool! But each adapter lives in its own Composer *package*... which is why we don't have the `QueryAdapter` class yet. So let's install it: copy the package name, spin over to your terminal, and run:

```
$ composer require pagerfanta/doctrine-orm-adapter
```

Once PhpStorm indexes the new code... try `new QueryAdapter()` again. We have it! Pass this `$queryBuilder` . We can also configure a few things, like `->setMaxPerPage(5)` . I'm using 5 per page so that pagination is *really* obvious.

```
 87 lines | src/Controller/QuestionController.php
... lines 1 - 9
use Pagerfanta\Doctrine\ORM\QueryAdapter;
use Pagerfanta\Pagerfanta;
... lines 12 - 17
class QuestionController extends AbstractController
{
... lines 20 - 32
    public function homepage(QuestionRepository $repository)
    {
        $queryBuilder = $repository->createAskedOrderedByNewestQueryBuilder();

        $pagerfanta = new Pagerfanta(new QueryAdapter($queryBuilder));
        $pagerfanta->setMaxPerPage(5);
... lines 39 - 42
    }
... lines 44 - 85
}
```

Looping Over a Pagerfanta

For the template, instead of passing a `questions` variable, we're going to pass a `pager` variable set to the `$pagerfanta` object.

```
 87 lines | src/Controller/QuestionController.php
... lines 1 - 17
class QuestionController extends AbstractController
{
... lines 20 - 32
    public function homepage(QuestionRepository $repository)
    {
        $queryBuilder = $repository->createAskedOrderedByNewestQueryBuilder();

        $pagerfanta = new Pagerfanta(new QueryAdapter($queryBuilder));
        $pagerfanta->setMaxPerPage(5);

        return $this->render('question/homepage.html.twig', [
            'pager' => $pagerfanta,
        ]);
    }
... lines 44 - 85
}
```

Now, pop into the homepage template... and scroll up. We *were* looping over the `questions` array.

```
 53 lines | templates/question/homepage.html.twig
... lines 1 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in questions %}
... lines 18 - 47
        {% endfor %}
    </div>
</div>
... lines 51 - 53
```

What do we do now? Loop over `pager`: `for question in pager`.

```
53 lines | templates/question/homepage.html.twig
... lines 1 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in pager %}
... lines 18 - 47
        {% endfor %}
    </div>
</div>
... lines 51 - 53
```

Yup, we can treat the `Pagerfanta` object like an *array*. The *moment* that we loop, Pagerfanta will execute the query it needs to get the results for the current page.

Testing time! Go back to the homepage. If we refresh now... 1, 2, 3, 4, 5. Yes! The paginator is limiting the results!

And check out the query for this page. Remember, the original query - before we added pagination - was *already* pretty complex. The pager *wrapped* that query in *another* query to get *just* the 5 question ids needed, ordered in the right way. Then, with a second query, it grabbed the *data* for those 5 questions.

The point is: the pager does some heavy lifting to make this work... and our complex query *doesn't* cause any issues.

So... cool! It returned only the first 5 results! But what about pagination links? Like a link to get to the next page... or the last page? Let's handle that next.

Chapter 27: Themed Pagination Links

Pagerfanta is now controlling the query for this page and returning only the first 5 results. So... how do we get to page 2? How can we render some pagination links?

Pagerfanta makes this delightfully easy. Scroll down. After the `endfor`, render the pagination links with `{{ pagerfanta(pager) }}`.

55 lines | templates/question/homepage.html.twig

```
... lines 1 - 9
<div class="container">
... lines 11 - 15
    <div class="row">
        {% for question in pager %}
... lines 18 - 47
        {% endfor %}

        {{ pagerfanta(pager) }}
    </div>
</div>
... lines 53 - 55
```

Let's try it! Refresh and... bah!

> Unknown `pagerfanta()` Function

Installing the Pagerfanta Twig Library

This is *another* feature that's a - sort of- "plugin" for Pagerfanta. Click back to the library's docs and go to "Installation and set up". It lists a bunch of different adapters... and also one *other* special package if you want Twig support for Pagerfanta. Copy that package name, find your terminal, and install it:

```
$ composer require pagerfanta/twig
```

Once that finishes, try the homepage again. This time... it works! Those links are pretty ugly... but we'll fix that in a minute.

Setting the Current Page

If you hover over the links, each adds a different `?page=`. There's 4 pages because we have 20 total questions. So these links are smart: they render the correct number based on how many results we have and how many we're showing per page.

Go to page 2. Hmm... I think this is actually the same results as page 1. And if we look down at the links... even though you see `?page=2` on the URL, Pagerfanta still highlights that we're on page 1. Why?

Because... we need to *help* Pagerfanta *know* which page we're on: we need to *read* the `?q=` and *pass* it *to* Pagerfanta.

Back in the controller, to read the query parameter, we need the request object. Add a `$request` argument type-hinted with the `Request` class from HttpFoundation. Then, below, add `$pagerfanta->setCurrentPage()` passing `$request->query->get('page', 1)` so that this returns `1` if there is *no* `?q=` on the URL.

88 lines | src/Controller/QuestionController.php

```php
// ... lines 1 - 17
class QuestionController extends AbstractController
{
// ... lines 20 - 32
    public function homepage(QuestionRepository $repository, Request $request)
    {
// ... lines 35 - 36
        $pagerfanta = new Pagerfanta(new QueryAdapter($queryBuilder));
        $pagerfanta->setMaxPerPage(5);
        $pagerfanta->setCurrentPage($request->query->get('page', 1));
// ... lines 40 - 43
    }
// ... lines 45 - 86
}
```

One small word of warning. At the time of recording, you can't *switch* these two lines. You need to set the max for the page and *then* the current page. If you swap them, weird things happen. This may get fixed, but to be safe, put the lines in this order.

Anyways, when we refresh now... beautiful! It sees that we're on page 2... and the results look different. If we go to page 3... that works too! Woo!

Customizing the Pagerfanta "View"

So let's talk about making these links prettier. You can *totally* customize them as much as you want, including with a custom template. But there are several built-in, sort of "themes"... including one for Bootstrap 5.

Back on the bundle documentation, click on "Default Configuration". This bundle has a `default_view` key... and one of the built-in views is called `twitter_bootstrap5`.

So... where do we make this config change? When we installed the bundle, it did *not* create a configuration file. And... that's fine! The bundle works great with the *default* config, so the author chose not to ship a config file. So now that we *do* want to configure the bundle, we'll create one ourselves.

Copy this `babdev_pagerfanta` key. Then, in `config/packages/`, create a new file called `babdev_pagerfanta.yaml`. Now technically, this file could be called *anything*: there's no significance to the filenames in this directory. But this name makes sense.

Inside, paste the root key, then set `default_view:` to `twitter_bootstrap5`. Before recording, I dug into the documentation to discover that this is one of the valid values.

If you need your pagination links to be translated, try this config in `pagerfanta.yaml`:

```yaml
# config/packages/pagerfanta.yaml
babdev_pagerfanta:
    default_view: twig
    default_twig_template: '@BabDevPagerfanta/twitter_bootstrap5.html.twig'
```

Hat-tip to Tomas in the comments for figuring that out!

3 lines | config/packages/pagerfanta.yaml

```yaml
babdev_pagerfanta:
    default_view: twitter_bootstrap5
```

Let's check it! Refresh and... huh... nothing changes: it's still rendering *exactly* like before. I wonder if Symfony didn't see my new config file. Let's manually clear the cache to be sure:

```
$ php bin/console cache:clear
```

Refresh again and... got it! You should *not* normally need to clear Symfony's cache while developing... that's super rare. But if you're ever not sure, it's safe to try. The point is, this *now* renders with Bootstrap 5 markup and it looks much better.

Putting the {page} Into the Route

Let's try one more thing. What if, instead of having `?page=2` on the URL, we wanted a URL like `/2`. So where the page is *inside* the main part of the URL.

That's... no problem. Over in `QuestionController`, add a new `{page}` to the URL. Now we need to be *very* careful because this is a wildcard. And so, if there are any other URLs on the site that are just `/something`, this route *could* break those if it matches first.

To avoid that, let's make this route *only* match if the `{page}` part of the URL is a *number*. Do that by adding a requirement - `<>` - with a regular expression inside: `\d+`.

So: only match this route if `{page}` is a *digit* of any length. If we go to `/foo`, this route won't match. Give the controller an `int $page` argument and default it to 1. This will allow the user to go to *just* `/` ... and `$page` will be 1.

```
88 lines | src/Controller/QuestionController.php
... lines 1 - 17
class QuestionController extends AbstractController
{
... lines 20 - 29
    /**
     * @Route("/{page<\d+>}", name="app_homepage")
     */
    public function homepage(QuestionRepository $repository, int $page = 1)
    {
... lines 35 - 43
    }
... lines 45 - 86
}
```

Below, pass the `$page` variable in directly. And... we don't need the request object at all anymore.

```
88 lines | src/Controller/QuestionController.php
... lines 1 - 17
class QuestionController extends AbstractController
{
... lines 20 - 29
    /**
     * @Route("/{page<\d+>}", name="app_homepage")
     */
    public function homepage(QuestionRepository $repository, int $page = 1)
    {
... lines 35 - 38
        $pagerfanta->setCurrentPage($page);
... lines 40 - 43
    }
... lines 45 - 86
}
```

Phew! Let's try it! Refresh. It jumped back to page 1 because we're not reading the page from the query parameter anymore. Click page 2. Yes! It's `/2` ... then `/3` ! So cool!

Ok team! Congratulations on finishing *both* Doctrine courses! Big team high five! Doctrine is one of *the* most important parts of Symfony and it will unlock you for almost *anything* else you do. So let us know what cool stuff you're building and, if you have any questions or ideas, we're here for you down in the comments.

Alright friends, seeya next time!