Doctrine, Symfony 6 & the Database

With <3 from SymfonyCasts

Chapter 1: Installing Doctrine

Welcome back team to episode *three* of our Symfony 6 series!The first two courses were *super* important: taking us from the basics up through the *core* of how *everything* works in Symfony: all that good "services" & configuration stuff. You are now ready to use *any* other part of Symfony and *really* start building out a site.

And... what better way to do that than to add a database?Because... so far, for all the cool things we've done, the site we've been building is 100% static. Boring! Time to change that.

Hello Doctrine

So we know that Symfony is a collection of a *lot* of libraries for solving a *ton* of different problems. So... does Symfony have some tools to help us talk to the database? The answer is... no! Because... it doesn't have to!

Why? Enter Doctrine: *the* most powerful library in the PHP world for working with databases.And Symfony and Doctrine work *great* together: they're the Frodo and Sam Gamgeeof PHP middle earth: the Han Solo and Chewbacca of the PHP Rebel Alliance. Symfony & Doctrine are like two Disney characters that finish each other's sandwiches!

Project Setup

To see this dynamic duo in action, let's get our project set up.Playing with databases is fun, so code along with me!Do that by downloading the course code from this page. After unzipping it, you'll find a `start/` directory with the same code that you see here. Pop open this `README.MD` file for all the setup instructions.

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

```
symfony serve -d
```

This uses the Symfony binary to start a local web serverwhich lives at https://127.0.0.1:8000. I'll take the lazy way out and click that to see... Mixed Vinyl! Our latest startup idea - and I swear, this one is going to be*huge* - combines the nostalgia for the "mix tapes" of the 80's and 90's with the audio experience of vinyl records. You craft your sweet mix tapes, then we press them onto a vinyl record for a *full* hipster audio experience.

So far, our site has a homepage *and* a page to browse mixes that *other* people created. Though, that page isn't *really* dynamic: it pulls from a GitHub repository... and unless you've configured an API key like we did in the last episode, this page is broken! That's the *first* thing we'll fix: by querying a databasse for the mixes.

Installing Doctrine

So let's get Doctrine installed! Find your terminal and run:

```
composer require "doctrine:^2.2" "doctrine/annotations:^1.14"
```

This is, of course, a Flex alias for a library called `symfony/orm-pack`. And remember: a "pack" is a, sort of, "fake library" that serves as a shortcut to install *several* packages at once. In this case, we're installing Doctrine itself, but also a few other relataed libraries, like the excellent Doctrine Migrations system.

Docker Configuration

Oh, and check this out! The command is asking:

> Do you want to include Docker configuration from recipes?

So, occasionally when you install a package, that package's recipe will contain Docker configuration that can, for example, start a database container. This is totally optional, but I'm going to say `p` for yes permanently. We'll talk more about the Docker configuration in a few minutes.

The Doctrine Recipes

But right now, let's check out what the recipe did. Run:

```
git status
```

Okay cool: this modified the normal files like `composer.json`, `composer.lock` and `symfony.lock`... and it *also* modified `config/bundles.php`. If you check that out... no surprise: our app now has *two* new bundles: DoctrineBundle and DoctrineMigrationsBundle.

17 lines | config/bundles.php

```
... lines 1 - 2
return [
... lines 4 - 13
    Doctrine\Bundle\DoctrineBundle\DoctrineBundle::class => ['all' => true],
    Doctrine\Bundle\MigrationsBundle\DoctrineMigrationsBundle::class => ['all' => true],
];
```

But probably the most important part of the recipe is the change it made to our `.env` file. Remember: this is where we can configure environment variables... and the recipe gave us a *new* one called `DATABASE_URL`. This, as you can see, holds all the connection details, like the username and password.

30 lines | .env

```
... lines 1 - 27
DATABASE_URL="postgresql://symfony:ChangeMe@127.0.0.1:5432/app?serverVersion=13&charset=utf8"
... lines 29 - 30
```

What *uses* this environment variable? Excellent question! Check out a new file the recipe gave us: `config/packages/doctrine.yaml`. Most of this config you won't need to think about or change. But notice this `url` key: it reads that `DATABASE_URL` environment variable!

43 lines | config/packages/doctrine.yaml

```
doctrine:
    dbal:
        url: '%env(resolve:DATABASE_URL)%'
... lines 4 - 43
```

The point is: the `DATABASE_URL` env var is the *key* to setting up your app to talk to a database... and we'll play with it in a few minutes.

The recipe also added a few new directories: `migrations/` `src/Entity/` and `src/Repository/`. Right now, other than a meaningless `.gitignore` file, these are all empty. We'll start filling them up real soon.

Ok: Doctrine *is* now installed. But to talk to a database... we need to make sure we have a database running *and* that the `DATABASE_URL` environment variable is pointing to it. Let's do that next, but with an optional & delightful twist: we're going to use Docker to start the database.

Chapter 2: docker-compose & Exposed Ports

We need to get a database running: MySQL, Postgresql, whatever. If you already have one running, awesome! All you need to do is copy your `DATABASE_URL` environment variable, open or create a `.env.local` file, paste, then change it to match whatever your local setup is using. If you decide to do this, feel free to skip ahead to the end of chapter 4 where we configure the `server_version`.

Docker Just for the Database

For me, I do *not* have a database running locally on my system...and I'm *not* going to install one. Instead, I want to use Docker. And, we're going to use Docker in an interesting way. I *do* have PHP installed locally:

```
php -v
```

So I *won't* use Docker to create a container specifically for PHP. Instead I'm going to use Docker simply to help boot up any *services* my app needs locally. And right now, I need a database service. Thanks to some magic between Docker and the Symfony binary, this is going to be *super* easy.

To start, remember when the Doctrine recipe asked us if we wanted Docker configuration? Because we said yes, the recipe gave us `docker-compose.yml` and `docker-compose.override.yml` files. When Docker boots, it will read *both* of these... and they're split into two pieces just in case you want to *also* use Docker to deploy to production. But we're not going to worry about that: we just want to use Docker to make life easier for local development.

22 lines | docker-compose.yml

```
version: '3'

services:
###> doctrine/doctrine-bundle ###
  database:
    image: postgres:${POSTGRES_VERSION:-13}-alpine
    environment:
      POSTGRES_DB: ${POSTGRES_DB:-app}
      # You should definitely change the password in production
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-ChangeMe}
      POSTGRES_USER: ${POSTGRES_USER:-symfony}
    volumes:
      - db-data:/var/lib/postgresql/data:rw
... lines 14 - 22
```

9 lines | docker-compose.override.yml

```
version: '3'

services:
###> doctrine/doctrine-bundle ###
  database:
    ports:
      - "5432"
... lines 8 - 9
```

These files say that they will boot a single Postgres database container with a user called `symfony` and password `ChangeMe`:

The username changed from `symfony` to `app` in the newest recipe version.

It will also expose port 5432 of the container - that's Postgres's normal port - to our *host* machine on a *random* port. This means that we're going to be able to talk to the Postgresql Docker container as *if* it were running on our local machine... as long as we know the random port that Docker chose. We'll see how that works in a minute.

By the way, if you want to use MySQL instead of Postgres, you absolutely can. Feel free to update these files... or delete both of them and run:

```
php bin/console make:docker:database
```

to generate a new compose file for MySQL or MariaDB. I'm going to stick with Postgres because it's awesome.

At this point, we're going to start Docker and learn a bit about how to communicate with the database that lives inside. If you're pretty comfortable with Docker, feel free to skip to the next chapter.

Starting the Container

Anyways, let's get our container running. First, make sure you have Docker actually installed on your machine: I won't show that because it varies by operating system. Then, find your terminal and run:

```
docker-compose up -d
```

The `-d` means "run in the background as a daemon". The first time you run this, it'll probably download a bunch of stuff. But eventually, our container should start!

Communicating with the Container

Cool! But now what? How can we *talk* to the container? Run a command called:

```
docker-compose ps
```

This shows info about all the containers currently running... just one for us. The really important thing is that port 5432 in the container is connected to port 50700 on my host machine. This means that if we talk to this port, we will actually be talking to that Postgres database. Oh, and this port is random: it'll be different on your machine... and it'll even change each time we stop and start our container. More on that soon.

But now that we know about port 50700, we can *use* that to connect to the database. For example, because I'm using Postgres, I could run:

```
psql --user=symfony --port=50700 --host=127.0.0.1 --password app
```

That means: connect to Postgres at 127.0.0.1 port 50700 using user `symfony` and talking to the `app` database. All of this is configured in the `docker-compose.yml` file. Copy the `ChangeMe` password because that last flag tells Postgres to ask for that password. Paste and... we're in!

If you're using MySQL, we can do this same thing with a `mysql` command.

But, this only works if we have that `psql` command installed on our *local* machine. So let's try a different command. Run:

```
docker-compose ps
```

again. The container is called `database`, which comes from our `docker-compose.yml` file. So we can change the previous command to:

```
docker-compose exec database psql --username symfony --password app
```

This time, we're executing the `psql` command *inside* the container, so we don't need to install it locally. Type `ChangeMe` for the password and... we're back in!

The point is: just by running `docker-compose up`, we have a Postgres database container that we can talk to!

Stopping the Container

Btw, when you're ready to stop the container later, you can run:

```
docker-compose stop
```

That basically turns the container off. Or you can run the more common:

```
docker-compose down
```

which turns off the containers and removes them. To start back up, it's the same:

```
docker-compose up -d
```

But notice that when we run `docker-compose ps` again, the port on my host machine is a *different* random port! So, in theory, we could configure the `DATABASE_URL` variable to point to our Postgres database, including using the correct port. But that random port that keeps changing is going to be annoying!

Fortunately, there's a trick for this! It turns our, our app is *already* configured, without us doing anything! That's next.

Chapter 3: Docker & Environment Variables

We now have a Postgres database running inside of a Docker container.We can see it by running:

```
docker-compose ps
```

This also tells us that if we want to*talk* to this database, we can connect to port `50739` on our local machine. That will be a different port for you, because it's randomly chosen when we start Docker.

We also learned that we can talk to the database directly via:

```
docker-compose exec database psql --user symfony --password app
```

To get our actual *application* to point to the database that's running on this port, we could go into `.env` or `.env.local` and customize `DATABASE_URL` accordingly: with user `symfony` password `ChangeMe` ... and with whatever your port currently is. Though... we *would* need to *update* that port each time we start and stop Docker.

Symfony Binary & Docker Env Vars

Thankfully, we don't need to do *any* of that because, surprise, the `DATABASE_URL` environment variable is *already* being correctly set! When we set up our project, we started a local dev server using the Symfony binary.

Just as a reminder, I'm going to run:

```
symfony server:stop
```

to stop that server. And then restart it with:

```
symfony serve -d
```

I'm mentioning this because the `symfony` binary has a *pretty* awesome Docker superpower.

Watch: when you refresh now... and hover over the bottom right corner of the web debug toolbar, it says "Env Vars: From Docker".

In short, the Symfony binary *noticed* that Docker was running and exposed some new environment variables pointing to the database! I'll show you. Open up `public/index.php` .

10 lines | public/index.php

```php
... lines 1 - 2
use App\Kernel;

require_once dirname(__DIR__).'/vendor/autoload_runtime.php';

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

We don't normally care about this file...but it's a great spot to dump some info *right* when our app starts booting. Inside the callback, `dd()` the `$_SERVER` superglobal. That variable contains a *lot* of information, *including* any environment variables.

Ok, spin over and refresh. Big list! Search for `DATABASE_URL` and... there it is! But that is *not* the value that we have in our `.env` file: the port is *not* what we have there. Nope, it's the *correct* port needed to talk to the Docker container!

Yup, the Symfony binary detects that Docker is running and sets a *real* `DATABASE_URL` environment variable that *points* to that container. And remember, since this is a *real* environment variable, it will win over any value placed in the `.env` or `.env.local` files.

The point is: *just* by starting Docker, everything is already set up: we didn't need to touch *any* config files. That's pretty cool.

By the way, if you want to see all the environment variables the Symfony binary is setting, you can run:

```
symfony var:export --multiline
```

But the most important one by far is `DATABASE_URL`.

Ok: Doctrine is configured! Next, let's create the database itself via a `bin/console` command. When we do that, we'll learn a trick for doing this *with* the environment variables from the Symfony binary.

Chapter 4: The "symfony console" Command & server_version

Doctrine is now configured to talk to our database, which lives inside a Docker container.That's thanks to the fact that the Symfony dev server exposes this `DATABASE_URL` environment variable, which *points* to the container. For me, the container is accessible on port 50739.

Now let's make sure the actual database has been created. But first, in `index.php` , remove the `dd()` ... then close that file.

Spin over to your terminal and run:

```
php bin/console
```

This prints *every* `bin/console` command that's available *including* a bunch of *new* ones that start with the word `doctrine` . Ooh. Most of these aren't very important and we'll walk through the ones that *are* along the way.

bin/console doctrine:database:create

For example, one is called `doctrine:database:create` . Cool, let's try it:

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

And... error! Look closely: it's trying to connect to port 5432.But our environment variable is pointing to port 50739! It's as if it's using the `DATABASE_URL` value from our `.env` file instead of the *real* one that's set by the Symfony binary.

And, in fact, that's *exactly* what's happening. And, it makes sense! When we refresh the page in our browser, that's processed *through* the `symfony` binary, which gives it the opportunity to add the environment variable.

But when we run a `bin/console` command - where `console` is just a PHP file that lives in a `bin/` directory, the `symfony` binary is *never* used as part of that process. This means it never has the opportunity to add the environment variable.And so, Symfony falls back to using the value from `.env` .

To fix this, whenever we run a `bin/console` command that needs the Docker environment variables, instead of running `bin/console` , run `symfony console` :

```
symfony console doctrine:database:create
```

That's literally a shortcut to running `bin/console` : it's no different. But the fact that we're executing it *through* the `symfony` binary gives it the opportunity to add the environment variables.

When we try this... yes! We *do* get an error because apparently the database already exists,but it *did* successfully connect and talk to the database.

Configuring the server_version

Ok, there's one last bit of configuration that we need to set.Open `config/packages/doctrine.yaml` . This file came from the recipe. Find `server_version` and un-comment it.

43 lines | config/packages/doctrine.yaml

```
doctrine:
    dbal:
... lines 3 - 6
        server_version: '13'
... lines 8 - 43
```

This value "13" is referring to the version of my database engine. Since I'm using Postgres version 13, I need 13 here. If you're using MySQL, you might need 8 or 5.7.

This helps Doctrine determine which features your database does or doesn't support...since a newer version of a database might support features that an older version doesn't. It's not a particularly interesting piece of configuration, we just need to make sure it's set.

Ok team: all the boring setup is *done*. Next: let's create our first entity class! Entities are the most *foundational* concept in Doctrine and the *key* to talking to our first database table.

Chapter 5: Entity Class

One of the coolest, but maybe most *surprising* things about Doctrine, is that it wants you to pretend like the database doesn't exist! Yea, instead of thinking about tables and columns, Doctrine wants us to think about objects and properties.

For example, let's say that we want to save some product data. The way we do that with Doctrine is by creating a `Product` class with *properties* that hold the data. Then you instantiate a `Product` object, set data onto it and politely ask Doctrine to save it for you. *We* don't have to worry about *how* Doctrine does that.

But, of course, behind the scenes Doctrine *is* talking to a database. It will INSERT the data from the `Product` object into a `product` table where each property is mapped to a column. This is called an Object Relational Mapper, or *ORM*.

Later, when we want to get that data back, we don't think about "querying" that table and its columns. Nope, we simply ask Doctrine to find the object that we had earlier. Of course, it *will* query the table... then recreate the object with the data. But that's not a detail *we* think about: we ask for the `Product` object, and it gives it to us. Doctrine handles all of the saving and querying *automatically*.

Generating the Entity with make:entity

Anyways, when we use an ORM like Doctrine, if we want to save something to the database, we need to create a class that *models* the thing we want to save, like a `Product` class. In Doctrine, these classes are given a special name: *entities*. Though, they're really just normal PHP classes. And while you *can* create these entity classes by hand, there's a MakerBundle command that makes life *much* nicer.

Spin over to your terminal and run:

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

In this case, we don't have to run `symfony console make:entity` because this command will *not* talk to the database: it *just* generates code. But, if you're ever not sure, using `symfony console` is always safe.

Okay, we want to create a class to store all of the vinyl mixes in our system. So let's create a new class called `VinylMix`. Then answer `no` for broadcasting entity updates: that's an extra feature related to Symfony Turbo.

Ok, here's the important part: it asks which properties we want. We're going to add *several*. Start with one called `title`. Next it asks which *type* this field is. Hit `?` to see the full list.

These are *Doctrine* types... and each one will map to a different column type in your database, depending on which database you're using, like MySQL or Postgres. The basic types are on top like `string`, `text` - which can hold *more* than a string) - `boolean`, `integer` and `float`. Then relationship fields - we'll talk about those in the next tutorial - some special fields, like storing JSON and date fields.

For `title`, use `string`, which can hold up to 255 characters. I'll keep the default length... then it asks us if the field can be null in the database. I'll answer `no`. This means that the column *cannot* be null. In other words, the column will be *required* in the database.

And... one field done! Let's add a few more. We need a `description`, and make this a `text` type. `string` maxes out at 255 characters, `text` can hold a ton more. This time, I'll say `yes` to making it nullable. So this will be an *optional* column in the database. Another one down!

For the next property, call it `trackCount`. It will be an `integer` and will be *not* null. Then add `genre`, as a `string`, length 255... and also not null so that it's required in the database.

Finally, add a `createdAt` field so we can know when each vinyl mix was originally created.This time, because the field name ends in "At", the command suggests a `datetime_immutable` type. Hit "enter" to use that, and also make this *not* null in the database.

We don't need to add any more properties right now so hit"enter" one more time to exit the command.

Done! What did this do? Well first, I can tell you that this did *not* talk to or change our database at *all*. Nope, it simply generated two classes. The first is `src/Entity/VinylMix.php`. The *second* is `src/Repository/VinylMixRepository.php`. Ignore the `Repository` one for now... we'll talk about its purpose in a few minutes.

97 lines | src/Entity/VinylMix.php

```
// ... lines 1 - 8
#[ORM\Entity(repositoryClass: VinylMixRepository::class)]
class VinylMix
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column()]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $title = null;

    #[ORM\Column(type: Types::TEXT, nullable: true)]
    private ?string $description = null;
// ... lines 22 - 31
    public function getId(): ?int
    {
        return $this->id;
    }

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

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

        return $this;
    }
// ... lines 48 - 95
}
```

Checking out the Entity Class & Attributes

Go open up the `VinylMix.php` entity. Say hello to... a... wow, pretty normal, boring PHP class!It generated a `private` property for each field we added, plus an extra `id` property. The command also added a getter and setter method for each of these.So... this is basically just a class that holds data... and we can access and set that data via the getter and setter methods

The *only* thing that makes this class special are the attributes.The `ORM\Entity` above the class tells Doctrine:

> Hey! I want to be able to save objects of this class to the database.This is an *entity*.

Then, above each property, we use `ORM\Column` to tell Doctrine that we want to save this property as a *column* in the table. This also communicates other options like the *length* of the column and whether or not it should be *nullable*. `nullable: false` is the default... so the command only generated `nullable: true` on the *one* property that needs it.

The other thing `ORM\Column` controls is the field *type*. That's set via this `type` option. As I mentioned, this doesn't refer directly to a MySQL or Postgres type... its a *Doctrine* type that will then *map* to something specific based on our database.

Field Type Guessing

But, interesting: the `type` option only shows up on the `$description` field. The reason for that is *really* cool... and new! Doctrine is smart. It looks at the type on your *property* and *guesses* the field type from that. So when you have a `string` property type, Doctrine assumes that you want that to be *its* `string` type. You *could* write `Types::STRING` inside `ORM\Column`... but that would be totally redundant.

We *do* need it for the `description` field, however... because we want to use the `TEXT` type, *not* the `STRING` type. But in every *other* situation, it works. Doctrine guesses the correct type from the `?int` property type... and the same thing happens down here for the `?\DateTimeImmutable` type.

Table and Column Naming

In addition to controlling things about each column, we can *also* control the *name* of the table by adding an `ORM\Table` above the class with name set to, for example, `vinyl_mix`. But, *surprise*! We don't need to do that! Why? Because Doctrine is really good at generating great names. It generates the table name by transforming the class into snake case. So even *without* `ORM\Table`, this will be the name of the table. The same applies to properties. `$trackCount` will map to a `track_count` column. Doctrine handles all of this for us: we don't need to think about our table or column names at all.

At this point, we've run `make:entity` and it generated an entity class for us. Yay! But... we don't actually *have* a `vinyl_mix` table in our database yet. How do we create one? With the magic of *database migrations*. That's next.

Chapter 6: Migrations

We created an entity class! But... *that's it*. The corresponding table does not *yet* exist in our database.

Let's think. In theory, Doctrine knows about our entity, all of its properties and their `ORM\Column` attributes. So... shouldn't Doctrine be able to make that table *for* us automatically? Yes! It *can*.

The make:migration Command

When we installed Doctrine earlier, it came with a migrations library that's *amazing*. Check it out! Whenever you make a change to your database structure - like adding a new entity class, or even adding a new property to an *existing* entity, you should spin over to your terminal and run:

```
symfony console make:migration
```

In this case, I'm running `symfony console` because this *is* going to talk to our database. Run that and... perfect! It created one new file in a `migrations/` directory with a timestamp for today's date. Let's go check it out! Find `migrations/` and open the new file.

36 lines | migrations/Version20220718170654.php

```
... lines 1 - 12
final class Version20220718170654 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 SEQUENCE vinyl_mix_id_seq INCREMENT BY 1 MINVALUE 1 START 1');
        $this->addSql('CREATE TABLE vinyl_mix (id INT NOT NULL, title VARCHAR(255) NOT NULL, description TEXT DEFAULT NULL, tra
        $this->addSql('COMMENT ON COLUMN vinyl_mix.created_at IS \'(DC2Type:datetime_immutable)\'');
    }

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

This holds a class with `up()` and `down()` methods... though I never run migrations in the "down" direction, so we'll focus only on `up()`. And... this is great! The migrations command saw our `VinylMix` entity, *realized* that its table was missing in the database, and generated the SQL needed in Postgres to create it, including all of the columns. That was *so* easy.

Executing the Migration

Ok... so how do we *execute* this migration? Back at your terminal, run:

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

Say `y` to confirm and... beautiful! It tells us that it's `Migrating up to` that specific version. It seems... like that worked! To make sure, you can try another `bin/console` command: `symfony console doctrine:query:sql` with `SELECT * FROM vinyl_mix`.

```
symfony console doctrine:query:sql 'SELECT * FROM vinyl_mix'
```

When we try that... whoops! Pardon my typo... nothing to see here. Try that again and... perfect! We didn't get an error! It just says that `The query yielded an empty result set`. If that table did *not* exist, like `vinyl_foo`, Doctrine would have *screamed* at us.

So, the migration *did* run!

How Migrations Work

This beautiful system deserves some explanation. Run

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

again. Check it out! It's smart enough to *avoid* executing that migration a second time! It *knows* that it already did that. But... how? Try running a different command:

```
symfony console doctrine:migrations:status
```

This gives some general info about the migration system. The most important part is in `Storage` where it says `Table Name` and `doctrine_migration_versions`.

Here's the deal: the first time we executed the migration, Doctrine *created* this special table, which literally stores a list of all of the migration classes that *have* been executed. Then, each time we run `doctrine:migrations:migrate`, it looks in our `migrations/` directory, finds all the classes, checks the database to see which have *not* already been executed, and only calls those. Once the new migrations finish, it adds them as rows to the `doctrine_migration_versions` table.

You can visualize this table by running:

```
symfony console doctrine:migrations:list
```

It sees our *one* migration and knows it already ran it. It even has the date!

This is cool... but let's push it further. Next, let's add a new property to our entity and generate a *second* migration to add the column.

Chapter 7: Adding new Properties

In our VinylMix entity, I forgot to add a property earlier: votes . We're going to keep track of the number of up votes or down votes that a particular mix has.

Modifying with make:entity

Ok... so how can we add a *new* property to an entity? Well, we can *absolutely* do it by hand: all we need to do is create the property and the getter and setter methods. *But*, a much easier way is to head *back* to our favorite make:entity command:

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

This is used to *create* entities, but we can also use it to *update* them. Type VinylMix as the class name and... it sees that it exists! Add a new property: votes ... make it an integer , say "no" to nullable.. then hit "enter" to finish.

The end result? Our class has a new property... and getter and setter methods below.

112 lines | src/Entity/VinylMix.php

```
... lines 1 - 9
class VinylMix
{
... lines 12 - 31
    #[ORM\Column]
    private ?int $votes = null;
... lines 34 - 99
    public function getVotes(): ?int
    {
        return $this->votes;
    }

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

        return $this;
    }
}
```

Generating a Second Migration

Ok, let's think. We have a vinyl_mix table in the database... but it does *not* yet have the new votes column. We need to *alter* the table to add it. How can we do that? The exact same way as before: with a migration! At your terminal, run:

```
symfony console make:migration
```

Then go check out the new class.

33 lines | migrations/Version20220718170741.php

```
... lines 1 - 12
final class Version20220718170741 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 vinyl_mix ADD votes INT NOT NULL');
    }
... lines 25 - 31
}
```

This is amazing! Inside the `up()` method, it says

> ALTER TABLE vinyl_mix ADD votes INT NOT NULL

So it saw our `VinylMix` entity, checked out the `vinyl_mix` table in the database, and generated a *diff* between them. It realized that, in order to make the database look like our entity, it needed to alter the table and add that `votes` column. That's simply *amazing*.

Back over at the terminal, if you run

```
symfony console doctrine:migrations:list
```

you'll see that it recognizes *both* migrations and it knows that it has *not* executed the second one. To do that, run:

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

Doctrine is smart enough to *skip* the first and execute the *second*. Nice!

When you deploy to production, all you need to do is run `doctrine:migrations:migrate` each time. It will handle executing any and all migrations that the *production* database hasn't yet executed.

Giving Properties Default Values

Ok, one more quick thing while we're here. Inside of `VinylMix`, the new `votes` property defaults to `null`. But when we create a new `VinylMix`, it would make a lot of sense to default the votes to *zero*. So let's change this to `= 0`.

Cool! And if we do that, the property in PHP no longer needs to allow `null`... so remove the `?`. Because we're initializing to an integer, this property will *always* be an `int`: it will never be null.

112 lines | src/Entity/VinylMix.php

```
... lines 1 - 9
class VinylMix
{
... lines 12 - 32
    private int $votes = 0;
... lines 34 - 110
}
```

But... I wonder... because I made this change, do I need to alter anything in my database? The answer is *no*. I can prove it by running a helpful command:

```
symfony console doctrine:schema:update --dump-sql
```

This is very similar to the `make:migration` command... but instead of generating a file with the SQL, it just prints out the SQL needed to bring your database up to date. In this case, it shows that our database is *already* in sync with our entity.

The point is: if we initialize the value of a property in PHP...that's *just* a PHP change. It doesn't change the column in the database or give the *column* a default value, which is totally fine.

Auto-Setting createdAt

Let's initialize one other field: `$createdAt`. It would be *amazing* if something automatically set this property whenever we created a new `VinylMix` object... instead of us needing to set it manually.

Whelp, we can do that by creating a good, old-fashioned PHP `__construct()` method. Inside, say `$this->createdAt = new \DateTimeImmutable()`, which will default to *right now*.

117 lines | src/Entity/VinylMix.php

```php
... lines 1 - 9
class VinylMix
{
... lines 12 - 34
    public function __construct()
    {
        $this->createdAt = new \DateTimeImmutable();
    }
... lines 39 - 115
}
```

That's it! And... we don't need the `= null` anymore since it will be initialized down here...and we also don't need the `?`, because it will *always* be a `DateTimeImmutable` object.

117 lines | src/Entity/VinylMix.php

```php
... lines 1 - 9
class VinylMix
{
... lines 12 - 29
    private \DateTimeImmutable $createdAt;
... lines 31 - 115
}
```

Nice! Thanks to this, the `$createdAt` property will automatically be set *every time* we instantiate our object. And that's just a PHP change: it doesn't change the column in the database.

All right, we have a `VinylMix` entity *and* the corresponding table. Next, let's instantiate a `VinylMix` object and *save* it to the database.

Chapter 8: Persisting to the Database

Now that we have an entity class and corresponding table, we're ready to save some stuff! So... how *do* we insert rows into the table? Wrong question! We're *only* going to focus on *creating objects* and *saving* them. Doctrine will handle the insert queries *for* us.

To help do this in the simplest way possible, let's make a fake "new Vinyl Mix" page.

In the `src/Controller/` directory, create a new `MixController` class and make this extend the normal `AbstractController`. Perfect! Inside, add a `public function` called `new()` that will return a `Response` from HttpFoundation. To make this a page, above, use the `#[Route]` attribute, hit "tab" to autocomplete that and let's call the URL `/mix/new`. Finally, to see if this is working, `dd('new mix')`.

17 lines | src/Controller/MixController.php

```
... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class MixController extends AbstractController
{
    #[Route('/mix/new')]
    public function new(): Response
    {
        dd('new mix');
    }
}
```

In the real world, this page might render a form. Then, when we submit that form, we would take its data, create a `VinylMix()` object and save it. We'll work on stuff like that in a future tutorial. For now, let's just see if this page works. Head over to `/mix/new` and... got it!

Ok, let's go create a `VinylMix()` object! Do that with `$mix = new VinylMix()` ... and then we can start setting data on it! Let's create a mix of one of my absolute *favorite* artists as a kid. I'll quickly set some other properties... we need to set, at the very least, all of the properties that have *required* columns in the database. For `trackCount`, how about some randomness for fun. And, for `votes`, the same thing... including *negative* votes... though the Internet would *never* be so cruel as to downvote any of my mixes *that* much. Finally, `dd($mix)`.

25 lines | src/Controller/MixController.php

```
... lines 1 - 12
    public function new(): Response
    {
        $mix = new VinylMix();
        $mix->setTitle('Do you Remember... Phil Collins?!');
        $mix->setDescription('A pure mix of drummers turned singers!');
        $mix->setGenre('pop');
        $mix->setTrackCount(rand(5, 20));
        $mix->setVotes(rand(-50, 50));

        dd($mix);
    }
... lines 24 - 25
```

So far, this has *nothing* to do with Doctrine. We're just creating an object and setting data onto it. This data is hard-coded, but you can imagine replacing this with whatever the user just submitted via a form. Regardless of where we get the data, when we refresh... we have an object with data on it. Cool!

Services vs Entities

By the way, our entity class, `VinylMix`, is the *first* class we've created that is *not* a service. There are generally *two* types of classes. First, there are *service* objects, like `TalkToMeCommand` or the `MixRepository` we created in the last tutorial. These objects do *work*... but they don't hold any data besides *maybe* some basic config. And we always fetch services from the container, usually via autowiring. *We* never instantiate them directly.

The *second* type of classes are *data* classes like `VinylMix`. The primary job of these classes is to hold *data*. They don't usually *do* any work except maybe some basic data manipulation. And unlike services, we *don't* fetch these objects from the container. Instead, we create them manually wherever and whenever we need them, like we just did!

Hello Entity Manager!

Anyway, now that we have an object, how can we *save* it? Well, saving something to the database is *work*. And so, no surprise, that work is done by a *service*! Add an argument to the method, type-hinted with `EntityManagerInterface`. Let's call it `$entityManager`.

`EntityManagerInterface` is, by far, *the* most important service for Doctrine. We're going to use it to *save*, *and* indirectly when we *query*. To save, call `$entityManager->persist()` and pass it the object that we want to save (in this case, `$mix`). Then we also need to call `$entityManager->flush()` with *no* arguments.

33 lines | src/Controller/MixController.php

```
... lines 1 - 5
use Doctrine\ORM\EntityManagerInterface;
... lines 7 - 10
class MixController extends AbstractController
{
... line 13
    public function new(EntityManagerInterface $entityManager): Response
    {
... lines 16 - 22
        $entityManager->persist($mix);
        $entityManager->flush();
... lines 25 - 30
    }
}
```

But... wait. Why do we have to call *two* methods?

Here's the deal. When we call `persist()`, that doesn't actually *save* the object or talk to the database at *all*. It just tells Doctrine:

> Hey! I want you to be "aware" of this object, so that later when we call `flush()`, you'll know to save it.

Most of the time, you'll see these two lines together - `persist()` and then `flush()`. The reason it's split into two methods is to help with batch data loading... where you could persist *a hundred* `$mix` objects and then *flush* them to the database *all at once*, which is more efficient. But most of the time, you'll call `persist()` and *then* `flush()`.

Okay, to make this a valid page, let's `return new Response()` from HttpFoundation and I'll use `sprintf` to return a message: `mix %d is %d tracks of pure 80\'s heaven`... and for those two wildcards, pass `$mix->getId()` and `$mix->getTrackCount()`.

33 lines | src/Controller/MixController.php

```
... lines 1 - 13
    public function new(EntityManagerInterface $entityManager): Response
    {
... lines 16 - 25
        return new Response(sprintf(
            'Mix %d is %d tracks of pure 80\'s heaven',
            $mix->getId(),
            $mix->getTrackCount()
        ));
    }
... lines 32 - 33
```

Let's try it! Move over, refresh and... yes! We see "Mix 1". That's so cool! *We* never actually *set* the ID (which makes sense). But when we *saved*, Doctrine grabbed the new ID and put that onto the `id` property.

If we refresh a few more times, we get mixes 2, 3, 4, 5, and 6. That's super fun. All *we* had to do is persist and flush the object. Doctrine handles all of the querying stuff *for* us.

Another way we can prove this is working is by running:

```
symfony console doctrine:query:sql 'SELECT * FROM vinyl_mix'
```

This time, we *do* see the results. *Awesome*!

Okay, now that we have stuff in the database, how do we *query* for it? Let's tackle that *next*.

Chapter 9: Querying the Database

Now that we've saved some stuff to the database, how can we read or query for it? Once again, at least for simple stuff, Doctrine doesn't want you to worry about querying. Instead, we just *ask* Doctrine for the *objects* we want.

Head over to `src/Controller/VinylController.php` and find the `browse()` action.

50 lines | src/Controller/VinylController.php

```
... lines 1 - 10
class VinylController extends AbstractController
{
... lines 13 - 37
    public function browse(string $slug = null): Response
    {
        $genre = $slug ? u(str_replace('-', ' ', $slug))->title(true) : null;

        $mixes = $this->mixRepository->findAll();

        return $this->render('vinyl/browse.html.twig', [
            'genre' => $genre,
            'mixes' => $mixes,
        ]);
    }
}
```

Here, we're loading all of the `$mixes` in our project... and we're currently doing it via this `MixRepository` service class that we created in the last episode. This class talks to a GitHub repository and reads from a hard-coded text file.

We're going to *stop* using this `MixRepository` and instead load these `$mixes` from the database.

Querying through the Entity Manager

Ok: to *save* objects, we leveraged the `EntityManagerInterface` service, which is *the* most important service *by far* in Doctrine. Whelp, this service can also *query* for objects. Let's take advantage of that. Add a new argument to `browse()`, type-hinted with `EntityManagerInterface`... and call it `$entityManager`.

54 lines | src/Controller/VinylController.php

```
... lines 1 - 6
use Doctrine\ORM\EntityManagerInterface;
... lines 8 - 12
class VinylController extends AbstractController
{
... lines 15 - 39
    public function browse(EntityManagerInterface $entityManager, string $slug = null): Response
    {
... lines 42 - 51
    }
}
```

Then, below, replace the `$mixes` line with *two* lines. Start with `$mixRepository = $entityManager->getRepository()` passing this the name of the *class* that we want to query from. Yes, we think about querying from an entity *class*, not a table. In this case, we want to query from `VinylMix::class`.

We'll talk more about this repository concept in a minute. Then, to get the mixes *themselves*, say `$mixes = $mixRepository->` and call one of the methods on it: `findAll()`.

To see what this gives us, let's `dd($mixes)`.

54 lines | src/Controller/VinylController.php

```
... lines 1 - 39
    public function browse(EntityManagerInterface $entityManager, string $slug = null): Response
    {
... lines 42 - 43
        $mixRepository = $entityManager->getRepository(VinylMix::class);
        $mixes = $mixRepository->findAll();
        dd($mixes);
... lines 47 - 51
    }
... lines 53 - 54
```

Ok, testing time! Spin over, head back to the homepage, click "Browse mixes" to hit that action, and...voila! We get six results! And *each* of them, *most importantly*, is a `VinylMix` *object*.

Behind the scenes, Doctrine *did* query the table and the columns. But instead of giving us that raw data, it put it onto *objects* and gave us *those*, which is *so* much nicer.

Working with Objects in Twig

If we remove the `dd()` ... this array of `VinylMix` object will be passed into the template, instead of the array of array data that we had before. But... the page *still* works. Though, these images are broken because *apparently* the service I'm using to load them is down right now. Ah... the joys of video recording. But that won't stop us!

The fact that all the data still renders without any errors is...actually kind of by *luck*. When we render the template - `templates/vinyl/browse.html.twig` - we loop over all of the `mixes`. The template works because the old GitHub repository text file had the same *keys* (like `title`, `trackCount`, and `genre`) as our `VinylMix` class.

46 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 28
            {% for mix in mixes %}
... line 30
                <div class="mixed-vinyl-container p-3 text-center">
... line 32
                    <p class="mt-2"><strong>{{ mix.title }}</strong></p>
                    <span>{{ mix.trackCount }} Tracks</span>
                    |
                    <span>{{ mix.genre }}</span>
                    |
                    <span>{{ mix.createdAt|ago }}</span>
... lines 39 - 40
            {% endfor %}
... lines 42 - 46
```

There *is* one cool thing happening here, though. When we say `mix.genre`, `mix` is now an *object*... and this `genre` property is *private*. That means we *cannot* access it directly. *But* Twig is smart. It realizes that this is private and looks for a `getGenre()` method. So in our template, we say `mix.genre`, but in reality, it calls the `getGenre()` method. That's pretty awesome.

Visualizing the Queries for the Page

Know what else is awesome? We can *see* the queries any page made! Down in the web debug toolbar, Doctrine gives us a fancy new icon. Oooo. And if we click into that...tah dah! There's one database query... and we can even see what it is. You can also see a *formatted* version of it... though I need to refresh the page for this to work...because the Turbo JavaScript library we installed in the first tutorial doesn't always play nice with this profiler area. Anyways, we can also see a *runnable* version of the query or run "Explain" on it.

The "Repository"

All right, back in the controller, even though we *can* query through the `EntityManagerInterface`, we normally query through something called the *repository*. `dd()` this `$mixRepository` object to get more info about it.

54 lines | src/Controller/VinylController.php

```
... lines 1 - 12
class VinylController extends AbstractController
{
... lines 15 - 39
    public function browse(EntityManagerInterface $entityManager, string $slug = null): Response
    {
... lines 42 - 44
        dd($mixRepository);
... lines 46 - 51
    }
}
```

Then go back to the `/browse` page and... it's an `App\Repository\VinylMixRepository` object. Hey! We *know* that class! It lives in *our* code, in the `src/Repository/` directory. It was generated by MakerBundle.

Inside the `ORM\Entity` attribute above our entity class, MakerBundle generated a `repositoryClass` option that *points* to this. Thanks to this config, our entity, `VinylMix` , is tied to `VinylMixRepository` . So when you ask Doctrine to give us the *repository* for the `VinylMix` class, it knows to return the `VinylMixRepository` object.

The repository for an entity knows *everything* about how to query for its data. And, without us doing *anything*, it already has a bunch of useful methods on it for basic queries, like `findAll()` , `findOneBy()` and several others. In a bit, we'll learn how to add *new* methods to the repository to make custom queries.

Anyway, `VinylMixRepository` is actually a service in the container...so we can get it more easily by autowiring it *directly*. Add a `VinylMixRepository $mixRepository` argument... and then we don't need this line at all. That is simpler... and it still works!

51 lines | src/Controller/VinylController.php

```
... lines 1 - 38
    public function browse(VinylMixRepository $mixRepository, string $slug = null): Response
    {
... lines 41 - 42
        $mixes = $mixRepository->findAll();
... lines 44 - 48
    }
... lines 50 - 51
```

The takeaway is this: if you want to query from a table, you'll do that through the *repository* of the entity whose data you need.

Next: The fact that we changed our code to load from the database and didn't need to update our Twig template *at all* was kind of awesome! And courtesy of some Twig magic. Let's talk *more* about that magic and create a virtual property that we can print in the template.

Chapter 10: Custom Entity Methods & Twig Magic

Our `VinylMix` entity has a `$votes` integer property... but we're not *printing* that on the page... just yet. Let's do that. Over in `templates/vinyl/browse.html.twig`, after `createdAt`, add a line break and print `mix.votes`... (which even autocompleted for us)! If we float over and refresh... nice! We see the votes, which can be positive *or* negative because, alas, the Internet *can* apparently be an unfriendly place!

The Built-in Repository Methods

Right now, we're querying the database and the results are coming backin whatever order the database wants. Could we order these by the highest votes first? Sure! One option is to write a custom query insideof `VinylMixRepository`, which we'll learn about soon. But these repository classes have several methodsthat allow us to, at least, do some basic stuff!

For example, we can call `findAll()`... or we could call `find()` and pass it an ID to find a*single* `VinylMix`. And there are others, like `findOneBy()` or `findBy()`, where you pass it an arrayof criteria to use in a WHERE clause. For example, we could find all mixes WHERE name equals some value.

But for this situation, leave that criteria *empty* so it returns *everything*. Why? Because I want to leverage the second argument: the "order". Pass an array with `'votes' => 'DESC'`.

51 lines | src/Controller/VinylController.php

```
... lines 1 - 11
class VinylController extends AbstractController
{
... lines 14 - 38
    public function browse(VinylMixRepository $mixRepository, string $slug = null): Response
    {
... lines 41 - 42
        $mixes = $mixRepository->findBy([], ['votes' => 'DESC']);
... lines 44 - 48
    }
}
```

And now... nice! The highest votes are first!

Adding a Custom Entity Method

Ok, so votes can be positive or negative. To make that *super* obvious, I want to print a plus sign in front of the positive votes. We *could* do that by adding some logic in Twig. But remember, we have this nice entity class! Sure, right *now* it only has getter and setter methods. But we *are* allowed to add our own custom methods. And that's a *great* way to organize your code.

Check it out: create a new public function called, how about `getVotesString()`, which will return a . I'm kidding, it'll return a `string` of course. Then calculate the "+" or "-" prefix with some fancy logic that says:

> If the votes are equal to zero, we want *no prefix*. If the votes are greater than zero, we want a plus symbol. Else we want a minus symbol.

And... let me surround this entire second statement in parenthesis. This is probably the fanciest line of code I've ever written... which also means it's the most confusing! Feel free to break this onto multiple lines.

124 lines | src/Entity/VinylMix.php

```
... lines 1 - 9
class VinylMix
{
... lines 12 - 116
    public function getVotesString(): string
    {
        $prefix = ($this->votes === 0) ? '' : (($this->votes >= 0) ? '+' : '-');
... lines 120 - 121
    }
}
```

At the bottom, `return sprintf()` with `%s`, which will be the prefix, and `%d`, which will be the vote count. Pass these in: `$prefix` then the absolute value of `$this->votes`... since we're adding the negative sign in manually.

124 lines | src/Entity/VinylMix.php

```
... lines 1 - 9
class VinylMix
{
... lines 12 - 116
    public function getVotesString(): string
    {
... lines 119 - 120
        return sprintf('%s %d', $prefix, abs($this->votes));
    }
}
```

We can now use this nice method anywhere in our app...like from inside a template with `mix.getVotesString()`. *Or* shorten this to `mix.votesString`.

48 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 2
{% block body %}
... lines 4 - 28
        {% for mix in mixes %}
... line 30
            <div class="mixed-vinyl-container p-3 text-center">
... lines 32 - 39
                {{ mix.votesString }} votes
            </div>
... line 42
        {% endfor %}
... lines 44 - 46
{% endblock %}
```

Twig is smart enough to realize that `votesString` is *not* a real property... but that there *is* a `getVotesString()` method. And so, it will call *that*. Think of this as a virtual property inside of Twig.

If we fly back over and refresh...awesome! We get the minus *and* plus signs.

A Second Custom Entity Method!

While we're here, the broken images - caused by the placeholder site I'm using being down - are...kind of annoying! Time to fix those!

In a real app, we'll probably let our users upload real images...though for now, we'll stick with dummy images. But either way, we'll probably need the ability to get the URL to a vinyl mix's image from multiple places in our code. To make that easy *and* keep the code centralized, let's add another entity method!

How about `public function getImageUrl()`. Give this a `$width` argument so we can ask for different sizes. Inside I'll paste in some code that uses a *different* service for dummy images. This looks a bit fancy - but I'm just trying to use the id to get a predictable, but random image... skipping the first 50, which are all nearly identical on this site.

133 lines | src/Entity/VinylMix.php

```
... lines 1 - 9
class VinylMix
{
... lines 12 - 123
    public function getImageUrl(int $width): string
    {
        return sprintf(
            'https://picsum.photos/id/%d/%d',
            ($this->getId() + 50) % 1000, // number between 0 and 1000, based on the id
            $width
        );
    }
}
```

Anyways, now we have this nice reusable method!

Back in the template... up here is where I have the hardcoded image URL. Replace this with `mix.imageUrl()` , but this time, we *do* need to pass an argument. Pass `300` ... and let's update the `alt` attribute as well to `Mix album cover` .

48 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 2
{% block body %}
... lines 4 - 28
            {% for mix in mixes %}
... lines 30 - 31
                    <img src="{{ mix.getImageUrl(300) }}" alt="Mix album cover">
... lines 33 - 42
            {% endfor %}
... lines 44 - 46
{% endblock %}
```

If we go over and refresh... *lovely*. Our mixes have images!

Cleanup: Deleting the Old Repository

Ok one last *tiny* cleanup thing. We no longer need this `MixRepository` service, which loads mixes from GitHub. Let's delete it so I don't get confused... since its name is *so* similar to the new `VinylMixRepository` . Right click on `MixRepository.php` , go to "Refactor", and click on "Safe Delete".

Easy! But... we *might* still be using that somewhere, right? If you go to your terminal and run:

```
git grep MixRepository
```

that'll show you where it's still being mentioned.

Though, Symfony's service container is so smart, it will often *tell* us if we've messed something up, like if we're still using a service that doesn't exist. Watch. Try refreshing any page. Yup!

> Cannot autowire service `App\Command\TalkToMeCommand` : argument `$mixRepository` of method `__construct()` has type `App\Service\MixRepository` .

Even though this page doesn't even *use* the `TalkToMeCommand` class, it figured out that there's a problem with it. Open it up: `src/Command/TalkToMeCommand.php` . Yep! We were using `MixRepository` ... so that we could call its `findAll()` method. Change that to use `VinylMixRepository` ... and then we can remove the `use` statement on top. The `VinylMixRepository` *still* has a `findAll()` method, so this will *still* work. This isn't a very efficient way to find a random mix, but it's good enough for now.

```
57 lines | src/Command/TalkToMeCommand.php
... lines 1 - 4
use App\Repository\VinylMixRepository;
... lines 6 - 17
class TalkToMeCommand extends Command
{
    public function __construct(
        private VinylMixRepository $mixRepository
    )
... lines 23 - 55
}
```

Ok, close that class and go refresh again. The service container found *another* problem spot in `VinylController`! Head over there and... up in the constructor... yep! We're autowiring it here too. But... we're not even using the property anymore, so remove it. Also delete its `use` statement and a couple of other `use` statements that are not being...uh... used anymore more.

```
49 lines | src/Controller/VinylController.php
... lines 1 - 4
use App\Repository\VinylMixRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use function Symfony\Component\String\u;

class VinylController extends AbstractController
{
    public function __construct(
        private bool $isDebug
    )
... lines 16 - 47
}
```

And now... the site works again!

Next, let's learn how to build custom queries via the query builder!

Chapter 11: The Query Builder

The `/browse` page is working... but what if we click on one of these genres? Well... that *kind of* works. It shows the *name* of the genre... but we get a list of *all* the mixes. What we *really* want is to filter these to only show mixes for *that* specific genre.

Right now, every mix in the database is in the "Pop" genre. Head back to `MixController` and find the fake method that creates new mixes so that we can make some more interesting dummy data. Add a `$genres` variable with "Pop" *and* "Rock" included... Then select a random one with `$genres[array_rand($genres)]` .

34 lines | src/Controller/MixController.php

```
... lines 1 - 10
class MixController extends AbstractController
{
... line 13
    public function new(EntityManagerInterface $entityManager): Response
    {
... lines 16 - 18
        $genres = ['pop', 'rock'];
        $mix->setGenre($genres[array_rand($genres)]);
... lines 21 - 31
    }
}
```

Cool! Now go to `/mix/new` and refresh a few times... until we have about 15 mixes. Back on `/browse` ... yup! We have a mix of "Rock" and "Pop" genres... they just don't *filter* yet.

So our mission is clear: customize the database query to *only* return the results for a specific genre. Ok, we can actually do that super easily in `VinylController` via the `findBy()` method. The genre is in the URL as the `$slug` wildcard.

So we *could* add an "if" statement where, if there *is* a genre, we return all the results where `genre` matches `$slug` . *But* this is a *great* opportunity to learn how to create a custom query. So let's undo that.

Custom Repository Method

The best way to make a custom query, is to create a new method in the *repository* for whatever entity you're fetching data for. In this case, that means `VinylMixRepository` . This holds a few example methods. Un-comment the first... and then start *simple*.

67 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 16
class VinylMixRepository extends ServiceEntityRepository
{
... lines 19 - 41
    /**
     * @return VinylMix[] Returns an array of VinylMix objects
     */
    public function findByExampleField($value): array
    {
        return $this->createQueryBuilder('v')
            ->andWhere('v.exampleField = :val')
            ->setParameter('val', $value)
            ->orderBy('v.id', 'ASC')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult()
        ;
    }
... lines 56 - 65
}
```

Call it `findAllOrderedByVotes()` . We won't worry about the genre quite yet: I just want to make a querythat returns all of the mixes ordered by votes. Remove the argument, this will return an array and the PHPdoc above helps my editor knowthat this will be an array of `VinylMix` objects

64 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 41
    /**
     * @return VinylMix[] Returns an array of VinylMix objects
     */
    public function findAllOrderedByVotes(): array
    {
... lines 47 - 51
    }
... lines 53 - 64
```

DQL and the QueryBuilder

There are a few different ways to execute a custom query in Doctrine. Doctrine, of course, eventually makes SQL queries. But Doctrine works with MySQL, Postgres and other database engines... and the SQL needed for each of those looks slightly different.

To handle this, internally, Doctrine has its *own* query language called Doctrine Query Language or "DQL", It looks something like:

> SELECT v FROM App\Entity\VinylMix v WHERE v.genre = 'pop';

You *can* write these strings by hand, but I leverage Doctrine's "QueryBuilder": a nice object that helps... ya know... build that query!

Creating the QueryBuilder

To use it, start with `$this->createQueryBuilder()` and pass an *alias* that will be used to identify this class within the query. Make this short, but unique among your entities - something like `mix` .

64 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 44
    public function findAllOrderedByVotes(): array
    {
        return $this->createQueryBuilder('mix')
... lines 48 - 51
    }
... lines 53 - 64
```

Because we're calling this from inside of `VinylMixRepository` , the QueryBuilder already knows to query from the `VinylMix` entity... and will use `mix` as the alias. If we executed this query builder right now, it would basically be:

> SELECT * FROM vinyl_mix AS mix

The query builder is *loaded* with methods to control the query.For example, call `->orderBy()` and pass `mix` - since that's our alias - `.votes` then `DESC` .

64 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 44
    public function findAllOrderedByVotes(): array
    {
        return $this->createQueryBuilder('mix')
            ->orderBy('mix.votes', 'DESC')
... lines 49 - 51
    }
... lines 53 - 64
```

Done! Now that our query is built, to execute call `->getQuery()` (that turns it into a `Query` object) and then `->getResult()` .

64 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 44
    public function findAllOrderedByVotes(): array
    {
        return $this->createQueryBuilder('mix')
... line 48
            ->getQuery()
            ->getResult()
        ;
    }
... lines 53 - 64
```

Well actually, there are a number of methods you can call to get the results.The main two are `getResult()` - which returns an *array* of the matching objects - or `getOneOrNullResult()` , which is what you would use if you were querying for one *specific* `VinylMix` or null. Because we want to return an array of matching mixes, use `getResult()` .

Now we can use this method. Over in `VinylController` (let me close `MixController` ...), instead of `findBy()` , call `findAllOrderedByVotes()` .

49 lines | src/Controller/VinylController.php

```
... lines 1 - 10
class VinylController extends AbstractController
{
... lines 13 - 36
    public function browse(VinylMixRepository $mixRepository, string $slug = null): Response
    {
... lines 39 - 40
        $mixes = $mixRepository->findAllOrderedByVotes();
... lines 42 - 46
    }
}
```

I *love* how clear that method is: it makes it super obvious exactly *what* we're querying for. And when we try it...it still works! It's not filtering yet, but the order *is* correct.

Adding the WHERE Statement

Okay, back to our new method. Add an optional `string $genre = null` argument. *If* a genre is passed, we need to add a "where" statement. To make space for that, break this onto multiple lines...and replace `return` with `$queryBuilder =` . Below, `return $queryBuilder` with `->getQuery()` , and `->getResult()` .

66 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 16
class VinylMixRepository extends ServiceEntityRepository
{
... lines 19 - 44
    public function findAllOrderedByVotes(string $genre = null): array
    {
        $queryBuilder = $this->createQueryBuilder('mix')
            ->orderBy('mix.votes', 'DESC');

        return $queryBuilder
            ->getQuery()
            ->getResult()
        ;
    }
... lines 55 - 64
}
```

Now we can say `if ($genre)` , and add the "where" statement. How? I bet you could guess: `$queryBuilder->andWhere()` .

But a word of warning. There is *also* a `where()` method... but I *never* use it. When you call `where()` , it will *clear* any existing "where" statements that the query builder might have... so you might accidentally *remove* something you added earlier. So, *always* use `andWhere()` . Doctrine is smart enough to figure out that, because this is the *first* WHERE, it doesn't *actually* need to add the `AND` .

Inside of `andWhere()` , pass `mix.genre =` ... but *don't* put the dynamic genre right in the string. That is a *huge* no-no: *never* do that. That opens you up for SQL injection attacks. Instead, whenever you need to put a dynamic value into a query, use a "prepared statement"... which is a fancy way of saying that you put a placeholder here, like `:genre` . The name of this could be *anything*... like "dinosaur" if you want. But *whatever* you call it, you'll then fill *in* the placeholder by saying `->setParameter()` with the *name* of the parameter - so `genre` - and then the value: `$genre` .

71 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 44
    public function findAllOrderedByVotes(string $genre = null): array
    {
... lines 47 - 49
        if ($genre) {
            $queryBuilder->andWhere('mix.genre = :genre')
                ->setParameter('genre', $genre);
        }
... lines 54 - 58
    }
... lines 60 - 71
```

Beautiful! Back over in `VinylController` , pass `$slug` as the genre.

Let's try this! Click back to the browse page first. Awesome! We get all the results. Now click "Rock" and... nice! Less results and all genres show "Rock"! If I filter by "Pop"... got it! We can even see the query for this...here it is. It has the "where" statement for genre equaling "Pop". Woo!

Reusing Query Builder Logic

As your project gets bigger and bigger, you're going to create more and more methods in your repository for custom queries.

And you may start repeating the same query logic over and over again.For example, we might order by the votes in a*bunch* of different methods in this class.

To avoid duplication, we can isolate that logic into a private method.Check it out! Add `private function addOrderByVotesQueryBuilder()` . This will accept a `QueryBuilder` argument (we want the one from `Doctrine\ORM`), but let's make it *optional*. And we will also *return* a `QueryBuilder` .

78 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 17
class VinylMixRepository extends ServiceEntityRepository
{
... lines 20 - 60
    private function addOrderByVotesQueryBuilder(QueryBuilder $queryBuilder = null): QueryBuilder
    {
... lines 63 - 65
    }
... lines 67 - 76
}
```

The job of this method is to add this `->orderBy()` line. And for convenience, if we don't pass in a `$queryBuilder` , we'll create a new one.

To allow that, start with `$queryBuilder = $queryBuilder ?? $this->createQueryBuilder('mix')` . I'm purposely using `mix` again for the alias. To keep life simple, choose an alias for an entity and *consistently* use it everywhere.

78 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 60
    private function addOrderByVotesQueryBuilder(QueryBuilder $queryBuilder = null): QueryBuilder
    {
        $queryBuilder = $queryBuilder ?? $this->createQueryBuilder('mix');
... lines 64 - 65
    }
... lines 67 - 78
```

Anyways, this line itself may look weird, but it basically says:

> If there *is* a QueryBuilder, then use it. Else, create a new one.

Below `return $queryBuilder` ... go steal the `->orderBy()` logic from up here and... paste. Awesome!

78 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 60
    private function addOrderByVotesQueryBuilder(QueryBuilder $queryBuilder = null): QueryBuilder
    {
... lines 63 - 64
        return $queryBuilder->orderBy('mix.votes', 'DESC');
    }
... lines 67 - 78
```

PhpStorm is a little angry with me... but that's just because it's having a rough morning and needs a restart:our code is, hopefully, just fine.

Back up in the original method, simplify to `$queryBuilder = $this->addOrderByVotesQueryBuilder()` and pass it *nothing*.

78 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 45
    public function findAllOrderedByVotes(string $genre = null): array
    {
        $queryBuilder = $this->addOrderByVotesQueryBuilder();
... lines 49 - 58
    }
... lines 60 - 78
```

Isn't that nice? When we refresh... it's not broken! Take *that* PhpStorm!

Next, let's add a "mix show" page where we can view a *single* vinyl mix. For the first time, we'll query for a single object from the database and deal with what happens if *no* matching mix is found.

Chapter 12: Querying for a Single Entity for a "Show" Page

Our users *really* need to be able to click on a mix and navigate to a page with more information about it...like eventually its track list! So let's make that possible! Let's create a page to display just *one* mix's details.

Creating the new Route & Controller

Head over to `src/Controller/MixController.php`. After the `new` action, add `public function show()` with the `[#Route()]` attribute above. The URL for this will be... how about `/mix/{id}`, where `id` will be the ID of that mix in the database. Below, add the corresponding `$id` argument. And... just to see if this is working, `dd($id)`.

40 lines | src/Controller/MixController.php

```
... lines 1 - 10
class MixController extends AbstractController
{
... lines 13 - 33
    #[Route('/mix/{id}')]
    public function show($id): Response
    {
        dd($id);
    }
}
```

Coolio! Spin over and go to, how about, `/mix/7`. Awesome! Our route and controller are hooked up!

Querying for a Single Object

Ok, now that we have the ID, we need to query for the *one* `VinylMix` in the database matching that. And we know how to query: via the *repository*. Add a second argument to the method type-hinted with `VinylMixRepository` and call it `$mixRepository`. Now replace the `dd()` with `$mix = $mixRepository->` and, for the first time, we're going to use the `find()` method. It's dead simple: it finds a single object using the primary key. So pass it `$id`. To make sure *this* is working, `dd($mix)`.

42 lines | src/Controller/MixController.php

```
... lines 1 - 5
use App\Repository\VinylMixRepository;
... lines 7 - 11
class MixController extends AbstractController
{
... lines 14 - 35
    public function show($id, VinylMixRepository $mixRepository): Response
    {
        $mix = $mixRepository->find($id);
        dd($mix);
    }
}
```

We don't know which IDs we actually *have* in our database right now, so as a workaround, go to `/mix/new` to create a *new* mix. In my case, it has ID 16. Cool: go to `/mix/16` and... hello `VinylMix` `id: 16`! The important thing to notice is that this returns a `VinylMix` *object*. Unless you do something custom, Doctrine *always* gives us back either a *single* object or an *array* of objects, depending on which method you call.

Rendering the Template

Now that we have the `VinylMix` object, let's render a template and pass that in. Do that with `return $this->render()` and call the template `mix/show.html.twig`. The template path *could* be anything, but since we're inside `MixController`, the directory `mix` makes sense. And since we're in the `show` action, `show.html.twig` *also* makes sense. Consistency is a great way to make friends with your fellow teammates!

Pass in a variable called `mix` set to the `VinylMix` object `$mix` .

45 lines | src/Controller/MixController.php

```
    ... lines 1 - 35
    public function show($id, VinylMixRepository $mixRepository): Response
    {
    ... lines 38 - 39
        return $this->render('mix/show.html.twig', [
            'mix' => $mix,
        ]);
    }
    ... lines 44 - 45
```

All right, let's go create that template. In `templates/` , add a new directory called `mix/` ... and inside of *that*, a new file called `show.html.twig` . Pretty much every template is going to start the same way.Begin by saying `{% extends 'base.html.twig' %}` .

8 lines | templates/mix/show.html.twig

```
{% extends 'base.html.twig' %}
... lines 2 - 8
```

As a reminder, `base.html.twig` has several blocks in it.The most important one down here is `block body` . *That's* what we'll override with our content. At the top, there's also a `block title` , which allows us to control the title of the page.Let's override *both*.

Say `{% block title %}{% endblock %}` and, in between, `{{ mix.title }} Mix` . Then override `{% block body %}` with `{% endblock %}` below. Inside, just to get started, add an `<h1>` with `{{ mix.title }}` .

8 lines | templates/mix/show.html.twig

```
... lines 1 - 2
{% block title %}{{ mix.title }} Mix{% endblock %}

{% block body %}
    <h1>{{ mix.title }}</h1>
{% endblock %}
```

When we try that... hello page! This is *super* simple - the `<h1>` isn't even in the right place - but it's*working*. Now we can add some *pizzazz*.

Making the Page All Fancy Looking

I'm going to head back to my template and paste in a bunch of new content.You can copy this from the code block on this page. The top of this is *exactly* the same: it extends `base.html.twig` and the `block title` looks like it did before. But then, in the body, we have a bunch of new markup, we print the mix title... and down here, I have a few `TODO` s for us where we'll print out more details.

41 lines | templates/mix/show.html.twig

```
... lines 1 - 4
{% block body %}
    <div class="container">
        <h1 class="d-inline me-3">{{ mix.title }}</h1>
        <div class="row mt-5">
            <div class="col-12 col-md-4">

                <svg width="100%" height="100%" viewBox="0 0 496 496" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1
... lines 12 - 32
                </svg>
            </div>
            <div class="col-12 col-md-8 ps-5">
                TODO: print track count, genre and description
            </div>
        </div>
    </div>
{% endblock %}
```

If you refresh now... nice! We even have the cute little record SVG...which you probably recognize from the homepage. That's awesome... except that duplicating this entire SVG in both templates is...*not* so awesome. Let's fix that duplication.

Avoiding Duplication with a Template Partial

Select all of this `<svg>` content, copy it, and over in the `mix/` directory, create a new file called `_recordSvg.html.twig`. Paste that here!

24 lines | templates/mix/_recordSvg.html.twig

```
<svg width="100%" height="100%" viewBox="0 0 496 496" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
    <defs>
        <linearGradient x1="50%" y1="0%" x2="50%" y2="100%" id="linearGradient-1">
            <stop stop-color="#C380F3" offset="0%"></stop>
            <stop stop-color="#4A90E2" offset="100%"></stop>
        </linearGradient>
    </defs>
    <g id="Mixed-Vinyl" stroke="none" stroke-width="1" fill="none" fill-rule="evenodd">
        <g id="Group">
            <g id="record-vinyl" fill="#000000" fill-rule="nonzero">
                <path d="M248,144 C190.562386,144 144,190.562386 144,248 C144,305.437614 190.562386,352 248,352 C305.437614,352 35
            </g>
            <g id="record-vinyl" transform="translate(144.000000, 144.000000)" fill="url(#linearGradient-1)" fill-rule="nonzero">
                <path d="M104,0 C46.562386,0 0,46.562386 0,104 C0,161.437614 46.562386,208 104,208 C161.437614,208 208,161.437614 2
            </g>
            <circle id="Oval" stroke="#979797" cx="248" cy="248" r="235"></circle>
            <circle id="Oval" stroke="#979797" cx="248" cy="248" r="215"></circle>
            <circle id="Oval" stroke="#979797" cx="248" cy="248" r="195"></circle>
            <circle id="Oval" stroke="#979797" cx="248" cy="248" r="175"></circle>
            <circle id="Oval" stroke="#979797" cx="248" cy="248" r="155"></circle>
        </g>
    </g>
</svg>
```

The reason I prefixed the name with `_` is to indicate that this is a template *partial*. That means it's a template that doesn't include a whole page - just *part* of a page. The `_` is optional... and just something that's done as a common convention: it doesn't change any behavior.

Thanks to this, we can go into `show.html.twig` and `{{ include('mix/_recordSvg.html.twig) }}`.

```
19 lines | templates/mix/show.html.twig
... lines 1 - 4
{% block body %}
    <div class="container">
... lines 7 - 8
            <div class="col-12 col-md-4">
                {{ include('mix/_recordSvg.html.twig') }}
            </div>
... lines 12 - 16
    </div>
{% endblock %}
```

Let's go do the same thing in the homepage template: `templates/vinyl/homepage.html.twig` . This is the *same* SVG here, so we'll include that same template.

```
37 lines | templates/vinyl/homepage.html.twig
... lines 1 - 4
{% block body %}
<div class="container">
... lines 7 - 8
        <div class="col-12 col-md-4">
            {{ include('mix/_recordSvg.html.twig') }}
        </div>
... lines 12 - 34
</div>
{% endblock %}
```

Nice! If we go check the homepage...it *still* looks great! And if we head back to the mix page and refresh...that looks great too!

To finish the template, let's fill in the missing details.Add an `<h2>` with `class="mb-4"` , and inside, say `{{ mix.trackCount }} songs` , followed by a `<small>` tag with `(genre: {{ mix.genre }})` ... and below this, a `<p>` tag with `{{ mix.description }}` .

```
19 lines | templates/mix/show.html.twig
... lines 1 - 4
{% block body %}
... lines 6 - 7
        <div class="row mt-5">
... lines 9 - 11
            <div class="col-12 col-md-8 ps-5">
                <h2 class="mb-4">{{ mix.trackCount }} songs <small>(genre: {{ mix.genre }})</small></h2>
                <p>{{ mix.description }}</p>
            </div>
        </div>
... line 17
{% endblock %}
```

And now... this is starting to come to life!We don't have a track list yet... because that's another database table we'll create in a future tutorial. But it's a nice start.

Linking to the Show Page

To complete the new feature, when we're on the `/browse` page, we need to *link* each mix to its show page.Open `templates/vinyl/browse.html.twig` and scroll down to where we loop.Ok: change the `<div>` that surrounds everything to an `<a>` tag. Then... break this onto multiple lines and add `href=""` . As you can see, PhpStorm was clever enough to update the closing tag to an `a` *automatically*.

To link to a page in Twig, we use the `path()` function and pass the name of the route.What... *is* the name of the route to our show page? The answer is... it doesn't *have* one! Ok, Symfony auto-generates a name...but we don't want to rely on that.As soon as we want to link to a route, we should give that route a proper name. How about `app_mix_show` .

49 lines | src/Controller/MixController.php

```
... lines 1 - 11
class MixController extends AbstractController
{
... lines 14 - 34
    #[Route('/mix/{id}', name: 'app_mix_show')]
    public function show($id, VinylMixRepository $mixRepository): Response
... lines 37 - 47
}
```

Copy that, head back to `browse.html.twig` and *paste*.

But this time, just pasting the route name isn't going to be enough!Check out this sweet error:

> Some mandatory parameters are missing ("id") to generate a URL for route "app_mix_show".

That makes sense! Symfony is trying to generate the URL to this route,but we need to tell it what *wildcard* value to use for `{id}` . We do that by passing a second array argument with `{}` . Inside set `id` to `mix.id` .

50 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 2
{% block body %}
... lines 4 - 28
            {% for mix in mixes %}
            <div class="col col-md-4">
                <a href="{{ path('app_mix_show', {
                    id: mix.id
                }) }}" class="mixed-vinyl-container p-3 text-center">
... lines 34 - 42
                </a>
... line 44
            {% endfor %}
... lines 46 - 48
{% endblock %}
```

And now... the page works! And we can click any of these to hop in and see more details.

Okay, we've got the happy path working! But what if *no* mix can be found for a given ID? Next: let's talk 404 pages *and* learn how we can be even *lazier* by getting Symfony to query for the `VinylMix` object *for* us.

Chapter 13: Param Converter & 404's

We've programmed the happy path. When I go to `/mix/13`, my database *does* find a mix with that id and...life is *good*. But what if I change this to `/99`? *Yikes*. That's a 500 error: *not* something we want our site to *ever* do. This really *should* be a 404 error. So, how do we trigger a 404?

Triggering a 404 Page

Over in the method, this `$mix` variable will either be a `VinylMix` object *or* null if one isn't found. So we can say `if (!$mix)`, and then, to trigger a 404, `throw $this->createNotFoundException()`. You can give this a message if you want, but it'll only be seen by developers.

49 lines | src/Controller/MixController.php

```
... lines 1 - 11
class MixController extends AbstractController
{
... lines 14 - 35
    public function show($id, VinylMixRepository $mixRepository): Response
    {
... lines 38 - 39
        if (!$mix) {
            throw $this->createNotFoundException('Mix not found');
        }
... lines 43 - 46
    }
}
```

This `createNotFoundException()`, as the name suggests, creates an exception object. So we're actually *throwing* an exception here... which is nice, because it means that code *after* this won't be executed.

Now, *normally* if you or something in your code throws an exception, it will trigger a 500 error. But this method creates a *special* type of exception that maps to a 404. Watch! Over here, in the upper right, when I refresh...404!

By the way, this is *not* what the 404 or 500 pages would look like on production. If we switched to the `prod` environment, we'd see a pretty generic error page with no details. Then you *customize* how those look, even making separate styles for 404 errors, 403 Access Denied errors, or even... *gasp* ... 500 errors if something goes *really* wrong. Check out the Symfony docs for how to customize error pages.

Param Converter: Automatic Query

Okay! We've queried for a single `VinylMix` object and even handled the 404 path. But we can do this with *way* less work. Check it out! Replace the `$id` argument with a *new* argument, type-hinted with our entity class `VinylMix`. Call it, how about, `$mix` to match the variable below. Then... delete the query... and also the 404. And now, we don't even need the `$mixRepository` argument at all.

43 lines | src/Controller/MixController.php

```
... lines 1 - 35
    public function show(VinylMix $mix): Response
    {
        return $this->render('mix/show.html.twig', [
            'mix' => $mix,
        ]);
    }
... lines 42 - 43
```

This... deserves some explanation. So far, the "things" that we are "allowed" to have as arguments to our controllers are (1) route wildcards like `$id` or (2) services. Now we have a *third* thing. When you type-hint an *entity* class, Symfony will query for the object *automatically*. Because we have have a wildcard called `{id}`, it will take this value (so "99" or "16") and query for a `VinylMix` whose `id` is *equal* to that. The name of the wildcard - `id` in this case - needs to match the property name it should use for the query.

But if I go back and refresh... it *doesn't* work!?

> Cannot autowire argument `$mix` of `MixController::show()`: it references `VinylMix` but no such service exists.

We *know* this isn't a service... so that make sense. But... why isn't it querying for the object like I just said it would?

Because... to get this feature to work, we need to install another bundle! Well, if you're using Symfony 6.2 and a new enough DoctrineBundle - probably version 2.8 - then this *should* work without needing *anything* else. But since we're using Symfony 6.1, we need one extra library.

Find your terminal and say:

```
composer require sensio/framework-extra-bundle
```

This is a bundle full of nice little shortcuts that, by Symfony 6.2, will all have been moved into Symfony itself. So eventually, you won't need this.

And now... without doing anything else... it works! It automatically queried for the `VinylMix` object and the page renders! And if you go to a bad ID, like `/99`... yes! Check it out! We get a 404! This feature is called a "ParamConverter"... which is mentioned in the error:

> `VinylMix` object not found by the `@ParamConverter` annotation.

Anyways, I *love* this feature. If I need to query for *multiple* objects, like in the `browse()` action, I'll use the correct repository service. But if I need to query for a *single* object in a controller, I use this trick.

Next, let's make it possible to up vote and down vote our mixes by leveraging a simple form. To do this, for the first time, we will *update* an entity in the database.

Chapter 14: The Request Object

New goal team: to allow users to upvote and downvote a mix.To accomplish this, in the VinylMix entity, when a user votes, we need to send an UPDATE query to change the $votes integer property in the database.

Adding a Simple Form

Let's *first* focus on the user interface. Open templates/mix/show.html.twig . To start, print {{ mix.votesString }} votes so we can see that here.

27 lines | templates/mix/show.html.twig

```
... lines 1 - 4
{% block body %}
    <div class="container">
... lines 7 - 11
            <div class="col-12 col-md-8 ps-5">
... lines 13 - 15
                {{ mix.votesString }} votes
... lines 17 - 22
            </div>
... line 24
    </div>
{% endblock %}
```

And... perfect! To add the upvote and downvote functionality, we *could* use some fancy JavaScript. But we're going to keep it simple by adding a button that posts a form. Well this will actually be fancier than it sounds.In the first tutorial, we installed the Turbo JavaScript library. So even though we'll use a normal <form> tag and button, Turbo will *automatically* submit it via AJAX for a smooth experience.

By the way, Symfony *does* have a form component and we'll talk about that in a future tutorial.But this form is going to be *so* simple that we don't really need it anyway. Add a beautifully boring <form> tag with action set to the path() function.

The form will submit to a new controller that...we still need to create!

Head over to MixController and add a new public function called vote() . Give this the #[Route()] attribute with the URL /mix/{id}/vote . And because we need to link to this, add a name: app_mix_vote .

49 lines | src/Controller/MixController.php

```
... lines 1 - 11
class MixController extends AbstractController
{
... lines 14 - 42
    #[Route('/mix/{id}/vote', name: 'app_mix_vote', methods: ['POST'])]
    public function vote(VinylMix $mix): Response
... lines 45 - 47
}
```

The {id} route wildcard will hold the id of the specific VinylMix that the user is voting on. To query for that, use the trick we learned earlier: add an argument type-hinted with VinylMix and call it $mix . Oh, and while we don't *need* to, I'll add the Response return type. Adding this is just a good practice.

Inside, to make sure things are working, dd($mix) .

49 lines | src/Controller/MixController.php

```
... lines 1 - 43
    public function vote(VinylMix $mix): Response
    {
        dd($mix);
    }
... lines 48 - 49
```

Cool! Copy the name of the route, go back to the template -`show.html.twig` - and inside `path()`, paste. And because this route has an `{id}` wildcard, pass `id` set to `mix.id`. Also give the form `method="POST"`... because anytime that submitting a form will *change* data on your server, it should submit with `POST`.

27 lines | templates/mix/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 11
            <div class="col-12 col-md-8 ps-5">
... lines 13 - 16
                <form action="{{ path('app_mix_vote', {id: mix.id }) }}" method="POST">
... lines 18 - 21
                </form>
            </div>
... lines 24 - 25
{% endblock %}
```

Heck, we can even *enforce* this requirement on our route by adding `methods: ['POST']`. That's optional, but now, if someone, for some reason, goes directly to this URL, which is a GET request, it won't match the route. Handy!

49 lines | src/Controller/MixController.php

```
... lines 1 - 11
class MixController extends AbstractController
{
... lines 14 - 42
    #[Route('/mix/{id}/vote', name: 'app_mix_vote', methods: ['POST'])]
    public function vote(VinylMix $mix): Response
... lines 45 - 47
}
```

Head back over to the form. This form... will be kind of strange. Instead of having fields the user can type into, all we need is a button. Add `<button>` with `type="submit"`... and then some classes for styling. For the text, use a Font Awesome icon: a `<span>` with `class="fa fa-thumbs-up"`.

27 lines | templates/mix/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 16
                <form action="{{ path('app_mix_vote', {id: mix.id }) }}" method="POST">
                    <button
                        type="submit"
                        class="btn btn-outline-primary"
                    ><span class="fa fa-thumbs-up"></span></button>
                </form>
... lines 23 - 25
{% endblock %}
```

Perfecto! Let's go check it out. Refresh and... thumbs up! And when we click it...beautiful! It hits the endpoint! Notice that the URL didn't change... that's just because Turbo submitted the form via Ajax...and then our `dd()` stopped everything.

Ok, in a minute, we're going to add another button with a thumbs down. So, somehow, in our controller, we're going to need to figure out which button - up or down - was just pushed.

To do that, on the button, add `name="direction"` and `value="up"`. Now, if we click this button, it will send one pieceof POST data called `direction` set to the value `up` ... almost as if the user typed the word `up` into a text field.

```
29 lines | templates/mix/show.html.twig
... lines 1 - 16
            <form action="{{ path('app_mix_vote', {id: mix.id }) }}" method="POST">
                <button
... lines 19 - 20
                    name="direction"
                    value="up"
                ><span class="fa fa-thumbs-up"></span></button>
            </form>
... lines 25 - 29
```

Fetching the Request DAta

Ok... but how do we *read* POST data in Symfony? Whenever you need to read *anything* from the request - like POST data, query parameters, uploaded files, or headers - you'll need Symfony's `Request` object. And there are two ways to get it.

The first is by autowiring a service called `RequestStack`. Then you can get the current request by saying `$requestStack->getCurrentRequest()`.

This works anywhere that you can autowire a service. But in a controller, there's an easier way. Undo that... and instead, add an argument that is type-hinted with `Request`. Get the one from Symfony's HttpFoundation. Let's call it `$request`.

```
50 lines | src/Controller/MixController.php
... lines 1 - 8
use Symfony\Component\HttpFoundation\Request;
... lines 10 - 12
class MixController extends AbstractController
{
... lines 15 - 44
    public function vote(VinylMix $mix, Request $request): Response
    {
... line 47
    }
}
```

At first, this looks like autowiring, right? It looks like `Request` is a service and we're autowiring that as an argument. *But*... surprise! `Request` is *not* a service. Nope, this is yet *another* "thing" that you're allowed to have as an argument to your controller.

Let's review. We now know *four* different types of arguments that you can have on a controller method. One: you can have route wildcards like `$id`. Two: You can autowire services. Three: You can type-hint entities. And four: You can type-hint the `Request` class. Yup, the `Request` object is *so* important that Symfony created a special case *just* for it.

And... it's kind of beautiful. Our *whole* job as developers is to "read the incoming request" and use it to "create a response". So it's... almost poetic that we can have a method that takes the `Request` as an argument and returns a `Response`. Input `Request`, *output* `Response`.

Fetching POST Data

But I digress. There are a lot of different methods and properties on the Request to fetch whatever you need. To read POST data, say `$request->request->get()` and then the name of the field. In this case, `direction`.

```
50 lines | src/Controller/MixController.php
... lines 1 - 44
    public function vote(VinylMix $mix, Request $request): Response
    {
        dd($request->request->get('direction'));
    }
... lines 49 - 50
```

We're not going to talk a lot about the `Request` object... because it's... just a simple object that holds data. If you need to read something from it, just look at the docs and it'll tell you how to do it.

All right, back over here, refresh the page... upvote and... got it! Okay, remove the `dd()` and set this to a direction variable with `$direction =` .

If, for some reason, the `direction` POST data is missing (this shouldn't happen unless someone ismessing with our site), default it to `up` .

50 lines | src/Controller/MixController.php

```
... lines 1 - 44
    public function vote(VinylMix $mix, Request $request): Response
    {
        $direction = $request->request->get('direction', 'up');
    }
... lines 49 - 50
```

Now let's add the downvote. Copy the entire button... paste... change the value to `down` and update the icon class to `fa fa-thumbs-down` .

35 lines | templates/mix/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 16
            <form action="{{ path('app_mix_vote', {id: mix.id }) }}" method="POST">
... lines 18 - 23
                <button
                    type="submit"
                    class="btn btn-outline-primary"
                    name="direction"
                    value="down"
                ><span class="fa fa-thumbs-down"></span></button>
            </form>
... lines 31 - 33
{% endblock %}
```

Okay, we know that the value will either be `up` or `down` . In our controller, let's use this. `if ($direction === 'up')` , then `$mix->setVotes($mix->getVotes() + 1)` . Else, do the same thing... except it will be `- 1` . Below, `dd($mix)` .

56 lines | src/Controller/MixController.php

```
... lines 1 - 12
class MixController extends AbstractController
{
... lines 15 - 44
    public function vote(VinylMix $mix, Request $request): Response
    {
        $direction = $request->request->get('direction', 'up');
        if ($direction === 'up') {
            $mix->setVotes($mix->getVotes() + 1);
        } else {
            $mix->setVotes($mix->getVotes() - 1);
        }
        dd($mix);
    }
}
```

On a real site, we'll probably also store *which* user is voting so that they can't vote over and over again. We'll learn how to do that in a future tutorial. But this will work just fine for now.

All right, head back and refresh. We have 49 votes. If we click the upvote button... 50! If we refresh and click downvote... 48!

Yay! *But*, we still haven't *saved* this value to the database. When we refresh, it always goes back to the original "49".

So... next, let's do that! We'll make an UPDATE query to the databaseand *also* finish the endpoint by redirecting to another page.

Chapter 15: Updating an Entity

We *are* successfully changing the value of the `votes` property. *Now* we need to make an update query to*save* that to the database.

To insert a `VinylMix`, we used the `EntityManagerInterface` service, and then called `persist()` and `flush()`. To update, we'll use that *exact* same service.

Updating an Entity with the Entity Manager

Add a new argument to the `vote()` method type-hinted with `EntityManagerInterface`. I'll call it `$entityManager`. Then, very simply, after we've set the `votes` property to the new value, call `$entityManager->flush()`.

58 lines | src/Controller/MixController.php

```
... lines 1 - 12
class MixController extends AbstractController
{
... lines 15 - 44
    public function vote(VinylMix $mix, Request $request, EntityManagerInterface $entityManager): Response
    {
... lines 47 - 53
        $entityManager->flush();
... line 55
    }
}
```

That's it people! Before I explain this, let's make sure it works.Refresh. We have 49 votes right now. I'll hit up. It says 50. But the *real* proof is that when we refresh...it *still* shows 50! It *did* save!

Persisting and Flushing: The Details

Ok, so when we created a new `VinylMix` earlier, we had to call `persist()` - passing the `VinylMix` object - *and* then `flush()`. But now, all we need is `flush()`. Why?

Here's the full story. When you call `flush()`, Doctrine loops over all of the entity objectsthat it "knows about" and "saves" them. And that "save" is smart. If Doctrine determines that an entity has *not* been saved yet, it will execute an INSERT query. But if it's an object that *does* already exist in the database, Doctrine will figureout *what* has changed on the object - if anything - and execute an `UPDATE` query. Yep! We just call `flush()` and *Doctrine* figures out what to do. It's... the best thing since Starburst Jellybeans.

But... why don't we need to call `persist()` when we're updating? Well, you *can* say `$entityManager->persist($mix)` if you want to. It's just... totally redundant!

When you call `persist()`, it tells Doctrine:

> Hey! I want you to be *aware* of this object so that, next time I call `flush()`, you'll know to save it.

When you create a new entity object, Doctrine doesn't really*know* about that object until you call `persist()`. But when you're *updating* an entity, it means that you've already *asked* Doctrine to query for that object. So Doctrine is *already* aware of it... and when we call `flush()`, Doctrine *will* - automatically - check that object to seeif any changes have been made to it.

Redirecting to Another Page

So... we are successfully saving the new vote count to the databaseNow what? Because... I don't think this `die` statement is going to look good on production.

Well, *anytime* you submit a form successfully, you always do the same thing: redirect to another page. How do we redirect in

Symfony? With `return $this->redirect()` passing whatever URL you want to redirect to. Though, usually we're redirecting to another page on our site... so we use a similar shortcut called `redirectToRoute()` and then pass a route name.

Let's redirect back to the show page. Copy the `app_mix_show` route name, paste... and just like with the Twig `path()` function, this accepts a second argument: an array of the route wildcards that we need to fill in. In this case, we have an `{id}` wildcard... so pass `id` set to `$mix->getId()`.

61 lines | src/Controller/MixController.php

```
... lines 1 - 44
    public function vote(VinylMix $mix, Request $request, EntityManagerInterface $entityManager): Response
    {
... lines 47 - 55
        return $this->redirectToRoute('app_mix_show', [
            'id' => $mix->getId(),
        ]);
    }
... lines 60 - 61
```

Now, remember: controllers *always* return a `Response` object. And, whelp it turns out that a redirect *is* a response. It's a response that, instead of containing HTML, basically says:

> Please send the user to this other URL

The `redirectToRoute()` method is a shortcut that returns this special response object, called a `RedirectResponse`.

Anyways, let's test the whole flow! Refresh, and... got it! After voting, we end up right back on this page. And, thanks to Turbo, this is all happening via Ajax calls... which is a nice bonus.

The only problem is that... it's so smooth that it's not *super* obvious that my vote *was* actually saved - other than seeing the vote number change. It might be better if we showed a success message. Let's do that next by learning about flash messages. We're also going to make our `VinylMix` entity trendier by exploring the concept of smart versus anemic models.

Chapter 16: Flash Message & Rich vs Anemic Models

After we submit a form successfully, we *always* redirect. Often times, we'll *also* want to show the user a success message so they *know* everything worked. Symfony has a special way to handle this: *flash messages*.

To *set* a flash message, before redirecting, call `$this->addFlash()` and pass, in this situation, `success`. For the second argument, put the message that you want to show to the user, like `Vote counted!`.

62 lines | src/Controller/MixController.php

```
... lines 1 - 12
class MixController extends AbstractController
{
... lines 15 - 44
    public function vote(VinylMix $mix, Request $request, EntityManagerInterface $entityManager): Response
    {
... lines 47 - 53
        $entityManager->flush();
        $this->addFlash('success', 'Vote counted!');
... lines 56 - 59
    }
}
```

The `success` key could be anything... it's kind of like a "category" for the flash message...and you'll see how we use that in a minute.

Flash messages have a fancy name, but they're a simple idea; Symfony stores flash messages in the user's *session*. What makes them special is that Symfony will *remove* the message *automatically* as soon as we *read* it. They're like self-destructing messages. Pretty cool.

Reading Flash Messages

So... how *do* we read them? The way I like to do it is by opening up my base template - `base.html.twig` - and reading and rendering them here. Let's put it right after the navigation but before the `{% block body %}`. Say `{% for message in %}`. Then, we want to read out any `success` category flash messages we might have. To do this, we can leverage the *one* global Twig variable in Symfony: `app`. This has several methods on it, like `environment`, `request`, `session`, the current `user`, or one called `app.flashes`. Pass *this* the *category* (in our case, `success`). As I mentioned, this could be *anything*. If you put `dinosaur` as the key in a controller, then you'd read the `dinosaur` messages out *here*. Finish with `{% endfor %}`.

87 lines | templates/base.html.twig

```
... lines 1 - 19
    <body>
        <div class="mb-5">
... lines 22 - 57
            {% for message in app.flashes('success') %}
... lines 59 - 61
            {% endfor %}
        </div>
... lines 64 - 84
    </body>
... lines 86 - 87
```

Typically, you'll only have *one* success message in your flash at a time, but *technically* you can have multiple. That's why we're looping over them.

Inside of this, render a `<div>` with `class="alert alert-success"` so it looks like a *happy* message. Then, print out `message`.

87 lines | templates/base.html.twig

```
... lines 1 - 57
            {% for message in app.flashes('success') %}
                <div class="alert alert-success">
                    {{ message }}
                </div>
            {% endfor %}
... lines 63 - 87
```

So if this works correctly, it will read all of our `success` flash messages and render them. And once they've been read, Symfony will *remove* them so that they won't render again on the *next* page load. By putting this in the base template, we can now set flash messages from *anywhere* in our app and they'll be rendered on the page. Pretty cool.

Watch. Head back to our page, upvote and...beautiful! We'll probably want to remove this extra margin in a real project, but we'll leave it for now.

Making our Entity Class Smarter

All right, look back at `MixController`. The logic for doing our "up" and "down" voting is pretty simple... but I think it can be better. Try this! Open up `VinylMix` ... and scroll down to `setVotes()`. Right after this, just to keep things organized, create a new `public function` called `upVote()` and return `void`. Inside, say `$this->votes++`. Copy that, and create a *second* method which we'll call - you guessed it - `downVote()` ... with `$this->votes--`.

143 lines | src/Entity/VinylMix.php

```
... lines 1 - 9
class VinylMix
{
... lines 12 - 116
    public function upVote(): void
    {
        $this->votes++;
    }

    public function downVote(): void
    {
        $this->votes--;
    }
... lines 126 - 141
}
```

Thanks to these methods, in `MixController`, instead of having `$mix->setVotes()` set to `$mix->getVotes() + 1`, we can just say `$mix->upVote()` ... and `$mix->downVote()`.

62 lines | src/Controller/MixController.php

```
... lines 1 - 12
class MixController extends AbstractController
{
... lines 15 - 44
    public function vote(VinylMix $mix, Request $request, EntityManagerInterface $entityManager): Response
    {
... line 47
        if ($direction === 'up') {
            $mix->upVote();
        } else {
            $mix->downVote();
... lines 52 - 59
    }
}
```

Now *that's* nice. Our controller reads much more clearly, and we've encapsulated the `upVote()` and `downVote()` logic *into* our entity. If we head over and refresh, it *still* works.

Smart vs Anemic Models

This highlights an interesting topic. We've now added *four* custom methods to our entity: two that help read the datain a special way, and two that help *set* data. When we run `make:entity` , it creates getter and setter methods for every single property. That's nice, because it makes our entity *infinitely* flexible. *Anyone* from *anywhere* can read or set any property. But sometimes, you might *not* want or need that. For example, do we really want a `setVotes()` method? Is there really a use case in our code for something to set the vote count to *any* number it wants? Probably not. We'll likely only need `upVote()` and `downVote()` . I *will* keep the `setVotes()` method... though, because we use it when we generate our dummy `VinylMix` object.

But, in general, by removing unnecessary getter and setter methods in your entityand replacing them with more descriptive methods like `upVote()` , `downVote()` , `getVoteString()` , or `getImageUrl()` - methods that fit your business logic - you can,little by little, give your entities more clarity. Our `upVote()` and `downVote()` methods are *super* clear and descriptive. Someone calling these doesn't even need to know or *care* how they work internally.

Entities that *only* have getter and setter methods are sometimes called "anemic models". Entities that *remove* these and replace them with specific methods for your business logic are sometimes called "rich models". Some people take this to an extreme and have almost *no* getter or setter methods. Here at SymfonyCasts, we tend to be pragmatic. We usually *do* have getter and setter methods, but we always look for ways to be more descriptive, like by adding `upVote()` and `downVote()` .

Next, let's install an *awesome* library called DoctrineExtensions. This is a magic library full of superpowers, like automatic timestampable, and slug creation behaviors.

Chapter 17: Doctrine Extensions: Timestampable

I *really* like adding timestampable behavior to my entities. That's where you have `$createdAt` and `$updatedAt` properties that are set automatically. It just... helps keep track of when things happened. We added `$createdAt` and cleverly set it by hand in the constructor. But what about `$updatedAt`? Doctrine *does* have an awesome event system, and we *could* hook into that to run code on "update" that sets that property. *But* there's a library that *already* does that. So let's get it installed.

Installing stof/doctrine-extensions-bundle

At your terminal, run:

```
composer require stof/doctrine-extensions-bundle
```

This installs a small bundle, which is a wrapper around a library called DoctrineExtensions. Like a lot of packages, this includes a recipe. But this is the first recipe that comes from the "contrib" repository. Remember: Symfony actually has *two* repositories for recipes. There's the main one, which is closely guarded by the Symfony core team. Then another called `recipes-contrib`. There *are* some quality checks on that repository, but it's maintained by the community. The first time that Symfony installs a recipe from the "contrib" repository, it asks you if that's okay. I'm going to say `p` for "yes permanently". Then run:

```
git status
```

Awesome! It enabled a bundle and added a new configuration file that we'll look at in a second.

Enabling Timestampable

So this bundle obviously has its own documentation. You can search for `stof/doctrine-extensions-bundle` and find it on Symfony.com. But the *majority* of the docs live on the underlying DoctrineExtensions *library*... which contains a bunch of really cool behaviors, including "sluggable" and "timestampable". Let's add "timestampable" first.

Step one: go into `config/packages/` and open the configuration file it just added. Here, add `orm` because we're using Doctrine ORM, then `default`, and lastly `timestampable: true`.

8 lines | config/packages/stof_doctrine_extensions.yaml

```yaml
// ... lines 1 - 2
stof_doctrine_extensions:
    default_locale: en_US
    orm:
        default:
            timestampable: true
```

This won't really *do* anything yet. It just activates a Doctrine listener that will be *looking* for entities that support timestampable each time an entity is inserted or updated. How do we make our `VinylMix` support timestampable? The easiest way (and the way I like to do it) is via a trait.

At the top of the class, say `use TimestampableEntity`.

126 lines | src/Entity/VinylMix.php

```
... lines 1 - 7
use Gedmo\Timestampable\Traits\TimestampableEntity;
... lines 9 - 10
class VinylMix
{
    use TimestampableEntity;
... lines 14 - 124
}
```

That's *it*. We're done! Lunch break!

To understand this black magic, hold "cmd" or "ctrl" and click into `TimestampableEntity`. This adds two properties: `createdAt` and `updatedAt`. And these are just normal fields, like the `createdAt` that we had before. It also has getter and setter methods down here, just like *we* have in our entity.

The magic is this `#[Gedmo\Timestampable()]` attribute. This says that:

> this property should be set `on: 'update'`

and

> this property should be set `on: 'create'`.

Thanks to this trait, we get all of this for free!And... we no longer need *our* `createdAt` property... because it already lives in the trait. So delete the property... and the constructor... and down here, remove the getter and setter methods.Cleansing!

Adding the Migration

The trait has a `createdAt` property like we had before, but it also adds an `updatedAt` field. And so, we need to create a new migration for that. You know the drill.At your terminal, run:

```
symfony console make:migration
```

Then... let's go check that file... just to make sure it looks like we expect.Let's see here... yup! We've got `ALTER TABLE vinyl_mix ADD updated_at`. And apparently the `created_at` column will be a *little* bit different than we had before.

39 lines | migrations/Version20220718170826.php

```
... lines 1 - 12
final class Version20220718170826 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 vinyl_mix ADD updated_at TIMESTAMP(0) WITHOUT TIME ZONE NOT NULL');
        $this->addSql('ALTER TABLE vinyl_mix ALTER created_at TYPE TIMESTAMP(0) WITHOUT TIME ZONE');
        $this->addSql('ALTER TABLE vinyl_mix ALTER created_at DROP DEFAULT');
        $this->addSql('COMMENT ON COLUMN vinyl_mix.created_at IS NULL');
    }
... lines 28 - 37
}
```

When Migrations Fail

Okay, let's go run that:

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

And... it *fails*!

> [...] column "updated_at" of relation "vinyl_mix" contains null values .

This is a Not null violation ... which makes sense. Our database already has a bunch of records in it...so when we try to add a new updated_at column that doesn't allow null values... it freaks out.

If the current state of our database were already on production, we would need to tweak this migration to give the new column a default value for those existing records. *Then* we could change it back to not allowing null. To learn more about handling failed migrations, check out a chapter on our Symfony 5 Doctrine tutorial.

But since we do *not* have a production database yet that contains viny_mix rows, we can take a shortcut: drop the database and start over with zero rows. To do that, run

```
symfony console doctrine:database:drop --force
```

to completely drop our database. And recreate it with

```
symfony console doctrine:database:create
```

At this point, we have an empty database with no tables - even the migrations table is gone. So we can re-run *all* of our migrations from the very beginning. Do it:

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

Sweet! Three migrations were executed: all successfully.

Back over on our site, if we go to "Browse Mixes", it's empty...because we cleared our database. So let's go to /mix/new to create mix ID 1... then refresh a few more times. Now head to /mix/7 ... and upvote that, which will *update* that VinylMix .

Ok! Let's see if timestampable worked! Check the database by running:

```
symfony console doctrine:query:sql 'SELECT * FROM vinyl_mix WHERE id = 7'
```

And... awesome! The created_at is set and then the updated_at is set to just a few seconds later when we upvoted the mix. It *works*. We can now easily add timestampable to *any* new entity in the future, just by adding that trait.

Next: let's leverage another behavior: sluggable. This will let us create fancier URLs by automatically saving a URL-safe version of the title to a new property.

Chapter 18: Clean URLs with Sluggable

Using a database ID in your URL is...kind of lame. It's more common to use *slugs*. A slug is a URL-safe version of the name or title of an item. In this case, the title of our mix.

To make this possible, we only need to do one thing: give our `VinylMix` class a `slug` property that *holds* this URL-safe string. Then, it'll be super easy to query for it. The only trick is that... something needs to *look* at the mix's title and *set* that `slug` property whenever a mix is saved. And, ideally that could happen automatically... cause I'm feeling kinda lazy... and I don't really want to do that work manually *everywhere*. Whelp, *that* is the job of the sluggable behavior from Doctrine Extensions.

Activating the Sluggable Listener

Head back to `config/packages/stof_doctrine_extensions.yaml` and add `sluggable: true`.

9 lines | config/packages/stof_doctrine_extensions.yaml

```yaml
// ... lines 1 - 2
stof_doctrine_extensions:
// ... line 4
    orm:
        default:
// ... line 7
            sluggable: true
```

Once again, this enables a listener that will be *looking* at each entity, whenever one is saved, to see if the sluggable behavior is activated on it. How do we do that?

Adding the Slug Property

First, we need a `slug` property on our entity. To add it, at your terminal, run:

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

Update `VinylMix` to add a new `slug` field. This will be a string, and let's limit it to a 100 characters. Also make this *not* nullable: it should be required in the database. And that's it! Hit "enter" one more time to finish.

That, not surprisingly, added a `slug` property.. plus `getSlug()` and `setSlug()` methods at the bottom.

141 lines | src/Entity/VinylMix.php

```
... lines 1 - 10
class VinylMix
{
... lines 13 - 34
    #[ORM\Column(length: 100)]
    private ?string $slug = null;
... lines 37 - 128
    public function getSlug(): ?string
    {
        return $this->slug;
    }

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

        return $this;
    }
}
```

One thing the `make:entity` command *doesn't* ask you is whether or not you want a property to be *unique* in the database. In `slug`'s case, we *do* want it to be unique, so add `unique: true`. That will add a `unique` constraint in the database to make sure that we never get duplicates.

141 lines | src/Entity/VinylMix.php

```
... lines 1 - 10
class VinylMix
{
... lines 13 - 34
    #[ORM\Column(length: 100, unique: true)]
    private ?string $slug = null;
... lines 37 - 139
}
```

Before we think about any sluggable magic, generate a migration for the new property:

```
symfony console make:migration
```

As usual, I'll open up that new file to make sure it looks okay. And... it does! It adds `slug` including a `UNIQUE INDEX` for `slug`. And when we run it with

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

it *explodes*... for the *same* reason as last time: `Not null violation`. We're adding a new `slug` column to our table that is *not null*... which means that any existing records won't work. As I said in the previous chapter, if your database is already on production, you would need to fix this. But since ours is *not*, we can cheat and reset the database like we did before:

```
symfony console doctrine:database:drop --force
```

Then:

```
symfony console doctrine:database:create
```

Finally re-run all of the migrations from the very beginning:

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

And... yes! 4 migrations executed.

Adding the Sluggable Attribute

At this point, we've activated the sluggable listener and added a `slug` column. But we're *still* missing a step. I'll prove it by going to `/mix/new` and... *error*:

> [...] column "slug" of relation "vinyl_mix" violates not-null constraint.

Yup! Nothing is *setting* the `slug` property yet. To tell the extensions library that this is a `slug` property that it should set automatically, we need to add - *surprise* - an attribute! It's called `#[Slug]`. Hit "tab" to autocomplete that, which will add the `use` statement that you need on top. Then, say `fields`, which is set to an array, and inside, just `title`.

143 lines | src/Entity/VinylMix.php

```
// ... lines 1 - 7
use Gedmo\Mapping\Annotation\Slug;
// ... lines 9 - 11
class VinylMix
{
// ... lines 14 - 36
    #[Slug(fields: ['title'])]
    private ?string $slug = null;
// ... lines 39 - 141
}
```

This says:

> use the "title" field to generate this slug.

And now... it looks like it's working! If we check the database...

```
symfony console doctrine:query:sql 'SELECT * FROM vinyl_mix'
```

Woohoo! The `slug` is down here... and you can see the library is *also* smart enough to add a little `-1`, `-2`, `-3` to keep it unique.

Updating our Route to use {slug}

Now that we have this `slug` column, over in `MixController`, let's make our route trendier by using `{slug}`.

```
62 lines | src/Controller/MixController.php
... lines 1 - 12
class MixController extends AbstractController
{
... lines 15 - 35
    #[Route('/mix/{slug}', name: 'app_mix_show')]
    public function show(VinylMix $mix): Response
... lines 38 - 60
}
```

What else do we need to change here?Nothing! Because the route wildcard is now called `{slug}` , Doctrine will use this value to query from the `slug` property. Genius!

Updating Links to the Route

Though, we do need to update any links that we generate *to* this route. Watch: copy the route name - `app_mix_show` - and search inside this file. Yup! We use it down here to redirect after we vote.Now, instead of passing the `id` wildcard, pass `slug` set to `$mix->getSlug()` .

```
62 lines | src/Controller/MixController.php
... lines 1 - 12
class MixController extends AbstractController
{
... lines 15 - 44
    public function vote(VinylMix $mix, Request $request, EntityManagerInterface $entityManager): Response
    {
... lines 47 - 56
        return $this->redirectToRoute('app_mix_show', [
            'slug' => $mix->getSlug(),
        ]);
    }
}
```

And if you searched, there's one other place we generate a URL to this route:`templates/vinyl/browse.html.twig` . Right here, we need to change the link on the "Browse" page to `slug: mix.slug` .

```
50 lines | templates/vinyl/browse.html.twig
... lines 1 - 2
{% block body %}
... lines 4 - 28
            {% for mix in mixes %}
            <div class="col col-md-4">
                <a href="{{ path('app_mix_show', {
                    slug: mix.slug
                }) }}" class="mixed-vinyl-container p-3 text-center">
... lines 34 - 42
                </a>
            </div>
            {% endfor %}
... lines 46 - 48
{% endblock %}
```

Testing time! Let me refresh a few times... then head back to the homepage... click "Browse Mixes", and... there's our list! If we click one of these mixes... beautiful! It used the slug and it *queried* via the slug. Life is good.

Ok, right now, to add dummy data so we can use the site, we've created this `new` action. But that's a pretty poor way to handle dummy data: it's manual, requires refreshing the page and, though we have *some* randomness, it creates boring data!

So next, let's add a proper data fixture system to remedy this.

Chapter 19: Simple Doctrine Data Fixtures

"Data fixtures" is the name given to dummy data that you add to your app while developing or running tests to make life easier. It's a lot nicer to work on a new feature when you actually *have* decent data in your database. We created some data fixtures, in a sense, via this `new` action. But Doctrine has a system specifically designed for this.

Installing DoctrineFixturesBundle

Search for "doctrinefixturesbundle" to find its GitHub repository. And you can actually read its documentation over on Symfony.com. Copy the install line and, at your terminal, run it:

```
composer require --dev orm-fixtures
```

`orm-fixtures` is, of course, a Flex alias, in this case, to `doctrine/doctrine-fixtures-bundle`. And... done! Run

```
git status
```

to see that this added a bundle, as well as a new `src/DataFixtures/` directory. Go open that up. Inside, we have a single new file called `AppFixtures.php`.

18 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 7
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        // $product = new Product();
        // $manager->persist($product);

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

DoctrineFixturesBundle is a delightfully simple bundle. It gives us a new console command called `doctrine:fixtures:load`. When we run this, it will empty our database and then execute the `load()` method inside of `AppFixtures`. Well, it will actually execute the `load()` method on *any* service we have that extends this `Fixture` class. So we *could* have multiple classes in this directory if we want.

If we run it right now... with an empty `load()` method, it clears our database, calls that blank method, and... the result over on the "Browse" page is that we have nothing!

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

Filling in the load() Method

That's not very interesting, so let's go fill in that `load()` method! Start in `MixController`: steal all of the `VinylMix` code... and paste it here. Hit "Ok" to add the `use` statement.

24 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 10
    public function load(ObjectManager $manager): void
    {
        $mix = new VinylMix();
        $mix->setTitle('Do you Remember... Phil Collins?!');
        $mix->setDescription('A pure mix of drummers turned singers!');
        $genres = ['pop', 'rock'];
        $mix->setGenre($genres[array_rand($genres)]);
        $mix->setTrackCount(rand(5, 20));
        $mix->setVotes(rand(-50, 50));

        $manager->flush();
    }
... lines 23 - 24
```

Notice the `load()` method accepts some `ObjectManager` argument. That's actually the `EntityManager`, since we're using the ORM. If you look down here, it already has the `flush()` call. The only thing we're missing is the `persist()` call: `$manager->persist($mix)`.

25 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 10
    public function load(ObjectManager $manager): void
    {
... lines 13 - 19
        $manager->persist($mix);
... lines 21 - 22
    }
... lines 24 - 25
```

So the variable is *called* `$manager` here... but these two lines are *exactly* what we have our controller: `persist()` and `flush()`.

Try the command again:

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

It empties the database, executes our fixtures, and we have... *one* new mix!

Okay, this is *kind of* cool. We have a new `bin/console` command to load stuff. But for developing, I want a really *rich* set of data fixtures, like... maybe 25 mixes. We *could* add those by hand here... or even create a loop. But there's a better way, via a library called "Foundry". Let's explore it next!

Chapter 20: Foundry: Fixtures You'll Love

Building fixtures is pretty simple, but *kind of* boring. And it would be *super* boring to manually create *25* mixes inside the `load()` method. That's why we're going to install an awesome library called "Foundry".To do that, run:

```
composer require zenstruck/foundry --dev
```

We're using `--dev` because we only need this tool when we're developing or running tests.When this finishes, run

```
git status
```

to see that the recipe enabled a bundle and also created one config file... which we won't need to look at.

Factories: make:factory

In short, Foundry helps us create entity objects.It's... almost easier just to see it in action.First, for each entity in your project (right now, we only have one), you'll need a corresponding *factory* class. Create that by running

```
php bin/console make:factory
```

which is a Maker command that comes from Foundry.Then, you can select which entity you want to create a factory for...or generate a factory for *all* your entities. We'll generate one for `VinylMix` . And... that created a single file: `VinylMixFactory.php` . Let's go check it out: `src/Factory/VinylMixFactory.php` .

65 lines | src/Factory/VinylMixFactory.php

```
... lines 1 - 10
/**
 * @extends ModelFactory<VinylMix>
 *
 * @method static VinylMix|Proxy createOne(array $attributes = [])
 * @method static VinylMix[]|Proxy[] createMany(int $number, array|callable $attributes = [])
... lines 16 - 27
 */
final class VinylMixFactory extends ModelFactory
{
... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
            // TODO add your default values here (https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html#model-factories)
            'title' => self::faker()->text(),
            'trackCount' => self::faker()->randomNumber(),
            'genre' => self::faker()->text(),
            'votes' => self::faker()->randomNumber(),
            'slug' => self::faker()->text(),
            'createdAt' => null, // TODO add DATETIME ORM type manually
            'updatedAt' => null, // TODO add DATETIME ORM type manually
        ];
    }
... lines 51 - 63
}
```

Cool! Above the class, you can see a bunch of methods being described...which will help our editor know what super-powers this has. This factory is really good at creating and saving `VinylMix` objects... *or* creating *many* of them, *or* finding a *random* one, *or* a random *set*, *or* a random *range*. Phew!

getDefaults()

The only important code that we see inside this class is `getDefaults()`, which returns default data that should be used for each property when a `VinylMix` is created. We'll talk more about that in a minute.

But first... let's run blindly forward and *use* this class! In `AppFixtures`, delete *everything* and replace it with `VinylMixFactory::createOne()`.

19 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 5
use App\Factory\VinylMixFactory;
... lines 7 - 9
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        VinylMixFactory::createOne();

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

That's it! Spin over and reload the fixtures with:

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

And... it *fails*! Boo

> Expected argument type "DateTime", "null" given at property path"createdAt"

It's telling us that *something* tried to call `setCreatedAt()` on `VinylMix` ... but instead of passing a `DateTime` object, it passed `null` . Hmm. Inside of `VinylMix` , if you scroll up and open `TimestampableEntity` , yup! We have a `setCreatedAt()` method that expects a `DateTime` object. Something called this... but passed `null` .

This actually helps show off how Foundry works.When we call `VinylMixFactory::createOne()` , it creates a new `VinylMix` and then sets all of this data onto it. But remember, all of these properties are *private*. So it doesn't set the title property directly. Instead, it calls `setTitle()` and `setTrackCount()` Down here, for `createdAt` and `updatedAt` , it called `setCreatedAt()` and passed it `null` .

In reality, we don't *need* to set these two properties because they will be set automatically by the timestampable behavior.

If we try this now...

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

It *works*! And if we go check out our site...*awesome*. This mix has 928,000 tracks, a random title, and 301 votes.All of this is coming from the `getDefaults()` method.

Fake Data with Faker

To generate interesting data, Foundry leverages *another* library called "Faker", whose only job is to... create *fake* data. So if you want some fake text, you can say `self::faker()->` , followed by whatever you want to generate. There are *many* different methods you can call on `faker()` to get all *kinds* of fun fake data. Super handy!

Creating Many Objects

Our factory did a *pretty* good job... but let's customize things to make it a bit more realistic.Actually, first, having *one* `VinylMix` still isn't very useful. So instead, inside `AppFixtures` , change this to `createMany(25)` .

19 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 11
    public function load(ObjectManager $manager): void
    {
        VinylMixFactory::createMany(25);
... lines 15 - 16
    }
... lines 18 - 19
```

This is where Foundry shines. If we reload our fixtures now:

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

With a *single* line of code, we have *25* random fixtures to work with! Though, the random data *could* be a bit better... so let's improve that.

Customizing getDefaults()

Inside `VinylMixFactory` , change the title. Instead of `text()` - which can sometimes be a *wall* of text, change to `words()` ... and let's use *5* words, and pass *true* so it returns this as a string. Otherwise, the `words()` method returns an array. For `trackCount` , we *do* want a random number, but... probably a number between 5 and 20.For `genre` , let's go for a `randomElement()` to randomly choose either `pop` or `rock` . Those are the two genres that we've been working with so far.And, whoops... make sure you call this like a function. There we go. Finally, for `votes`, choose a random number between -50 and 50.

62 lines | src/Factory/VinylMixFactory.php

```php
// ... lines 1 - 28
final class VinylMixFactory extends ModelFactory
{
// ... lines 31 - 37
    protected function getDefaults(): array
    {
        return [
            'title' => self::faker()->words(5, true),
            'trackCount' => self::faker()->numberBetween(5, 20),
            'genre' => self::faker()->randomElement(['pop', 'rock']),
            'votes' => self::faker()->numberBetween(-50, 50),
            'slug' => self::faker()->text(),
        ];
    }
// ... lines 48 - 60
}
```

Much better! Oh, and you can see that `make:factory` added a *bunch* of our properties here by default, but it didn't add *all* of them. One that's missing is `description`. Add it: `'description' => self::faker()->` and then use `paragraph()`. Finally, for `slug`, we don't need that *at all* because it will be set automatically.

62 lines | src/Factory/VinylMixFactory.php

```php
// ... lines 1 - 37
    protected function getDefaults(): array
    {
        return [
// ... line 41
            'description' => self::faker()->paragraph(),
// ... lines 43 - 45
        ];
    }
// ... lines 48 - 62
```

Phew! Let's try this! Reload the fixtures:

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

Then head over and refresh. That looks *so* much better. We *do* have one broken image... but that's just because the API I'm using has some "gaps" in it... nothing to worry about.

Foundry can do a *ton* of other cool things, so *definitely* check out its docs. It's especially useful when writing tests, and it works *great* with database relations. So we'll see it again in a more complex way in the next tutorial.

Next, let's add pagination! Because eventually, we won't be able to list *every* mix in our database all at once.

Chapter 21: Pagination

Eventually, this page is going to get *super* long. By the time we have a thousand mixes, it probably won't even load!We can fix this by adding *pagination*. Does Doctrine have the ability to paginate results?It does! Though, I *usually* install another library that adds more features on top of those from Doctrine.

Find your terminal and run:

```
composer require babdev/pagerfanta-bundle pagerfanta/doctrine-orm-adapter
```

This installs a Pagerfanta bundle, which is a wrapperaround a really nice library called Pagerfanta. Pagerfanta can paginate lots of things, like Doctrine results, results from Elasticsearch, and much more. We also installed its Doctrine ORM *adapter*, which will give us everything we need to paginate our Doctrine results. In this case, when we run

```
git status
```

it added a *bundle*, but the recipe didn't need to do anything else.Cool! So how does this library work?

Open up `src/Controller/VinylController` and find the `browse()` action. Instead of querying for *all* of the mixes, like we're doing now, we're going to tell the Pagerfanta library *which* page the user is currently on, how many results to show *per* page, and then *it* will query for the correct results *for* us.

Returning a QueryBuilder

To get this working, instead of calling `findAllOrderedByVotes()` and getting back *all* of the results, we need to call a method on our repository that returns a *QueryBuilder*. Open `src/Repository/VinylMixRepository` and scroll down to `findAllOrderedByVotes()`. We're only using this method right here at the moment,so rename it to `createOrderedByVotesQueryBuilder()` ... and this will now return a `QueryBuilder` - the one from Doctrine ORM. I'll remove the PHP documentation on top... and the only thing we need to do down here is remove `getQuery()` and `getResult()` so that we're *just* returning `$queryBuilder`.

72 lines | src/Repository/VinylMixRepository.php

```
... lines 1 - 6
use Doctrine\ORM\QueryBuilder;
... lines 8 - 17
class VinylMixRepository extends ServiceEntityRepository
{
... lines 20 - 42
    public function createOrderedByVotesQueryBuilder(string $genre = null): QueryBuilder
    {
... lines 45 - 51
        return $queryBuilder;
    }
... lines 54 - 70
}
```

Over in `VinylController`, change this to `$queryBuilder = $mixRepository->createOrderedByVotesQueryBuilder($slug)`

57 lines | src/Controller/VinylController.php

```
... lines 1 - 12
class VinylController extends AbstractController
{
... lines 15 - 38
    public function browse(VinylMixRepository $mixRepository, string $slug = null): Response
    {
... lines 41 - 42
        $queryBuilder = $mixRepository->createOrderedByVotesQueryBuilder($slug);
... lines 44 - 54
    }
}
```

Initializing Pagerfanta is two lines. First, create the adapter - `$adapter = new QueryAdapter()` and pass it `$queryBuilder`. Then create the `Pagerfanta` object with `$pagerfanta = Pagerfanta::createForCurrentPageWithMaxPerPage()`

That's a *mouthful*. Pass this the `$adapter`, the current page - right now, I'm going to hardcode `1` - and finally the max results per page that we want. Let's use `9` since our mixes show up in three columns.

57 lines | src/Controller/VinylController.php

```
... lines 1 - 5
use Pagerfanta\Doctrine\ORM\QueryAdapter;
use Pagerfanta\Pagerfanta;
... lines 8 - 12
class VinylController extends AbstractController
{
... lines 15 - 38
    public function browse(VinylMixRepository $mixRepository, string $slug = null): Response
    {
... lines 41 - 43
        $adapter = new QueryAdapter($queryBuilder);
        $pagerfanta = Pagerfanta::createForCurrentPageWithMaxPerPage(
            $adapter,
            1,
            9
        );
... lines 50 - 54
    }
}
```

Now that we have this Pagerfanta object, we're going to pass *that* into the *template* instead of `mixes`. Replace this with a new variable called `pager` set to `$pagerfanta`.

57 lines | src/Controller/VinylController.php

```
... lines 1 - 38
    public function browse(VinylMixRepository $mixRepository, string $slug = null): Response
    {
... lines 41 - 50
        return $this->render('vinyl/browse.html.twig', [
... line 52
            'pager' => $pagerfanta,
        ]);
    }
... lines 56 - 57
```

The cool thing about this `$pagerfanta` object is that you can *loop* over it. And as soon as you do, it will execute the correct query to get *just* this pages results. In `templates/vinyl/browse.html.twig`, instead of `{% for mix in mixes %}`, say `{% for mix in pager %}`.

```
50 lines | templates/vinyl/browse.html.twig
... lines 1 - 2
{% block body %}
... lines 4 - 27
    <div class="row">
        {% for mix in pager %}
... lines 30 - 44
        {% endfor %}
    </div>
... lines 47 - 48
{% endblock %}
```

That's *it*. Each result in the loop will *still* be a VinylMix object.

If we go over and reload... got it! It shows *nine* results: the results for Page 1!

Linking to the Next Page

What we need now are *links* to the next and previous pages...and this library can help with that too. Back at your terminal, run:

```
composer require pagerfanta/twig
```

One of the trickiest things about the Pagerfanta library is, instead of it being one *giant* library that has everything you need, it's broken down into a bunch of smaller libraries. So if you want the ORM adapter support, you need to install it like we did earlier. If you want Twig support for adding links, you need to install that too. Once you do though, it's pretty simple.

Back in our template, find the `{% endfor %}`, and right after, say `{{ pagerfanta() }}`, passing it the `pager` object.

```
52 lines | templates/vinyl/browse.html.twig
... lines 1 - 2
{% block body %}
... lines 4 - 26
    <h2 class="mt-5">Mixes</h2>
    <div class="row">
... lines 29 - 46
        {{ pagerfanta(pager) }}
    </div>
... lines 49 - 50
{% endblock %}
```

Check it out! When we refresh... we have links at the bottom! They're... ugly, but we'll fix that in a minute.

Reading the Current Page

If you click the "Next" link, up in our URL, we see `?page=2`. Though... the results don't actually *change*. We're still seeing the same results from Page 1. And... that makes sense. Remember, back in `VinylController`, I hardcoded the current page to `1`. So even though we have `?page=2` up here, Pagerfanta *still* thinks we're on Page 1.

What we need to do is *read* this query parameter and pass it as this second argument. No problem! How do we read query parameters? Well, that's information from the request, so we need the `Request` object.

Right before our optional argument, add a new `$request` argument type-hinted with `Request`: the one from HttpFoundation. Now, down here, instead of `1`, say `$request->query` (that's how you get query parameters), with `->get('page')` ... and default this to `1` if there is *no* `?page=` on the URL.

58 lines | src/Controller/VinylController.php

```
... lines 1 - 8
use Symfony\Component\HttpFoundation\Request;
... lines 10 - 13
class VinylController extends AbstractController
{
... lines 16 - 39
    public function browse(VinylMixRepository $mixRepository, Request $request, string $slug = null): Response
    {
... lines 42 - 45
        $pagerfanta = Pagerfanta::createForCurrentPageWithMaxPerPage(
... line 47
            $request->query->get('page', 1),
... line 49
        );
... lines 51 - 55
    }
}
```

By the way, if you want, you can also add `{page}` up here. This way, Pagerfanta will *automatically* put the page number inside the URL instead of setting it as a query parameter.

If we head over and refresh... right now, we have `?page=2`. Down here... it knows we're on Page 2! If we go to the next page... yes! We see a different set of results!

Styling the Pagination Links

Though, this is *still* super ugly. Fortunately, the bundle *does* give us a way to control the markup that's used for the pagination links. And it even comes with automatic support for Bootstrap CSS-friendly markup. We just need to tell the bundle to *use* that.

So... we need to configure the bundle. But... the bundle didn't give us any new config files when it was installed. That's okay! Not all new bundles give us config files. But as soon as you need one, create one! Since this bundle's called `BabdevPagerfantaBundle`, I'm going to create a new file called `babdev_pagerfanta.yaml`. As we learned in the last tutorial, the *name* of these files *aren't* important. What's important is the root key, which should be `babdev_pagerfanta`. To change how the pagination renders, add `default_view: twig` and then `default_twig_template` set to `@BabDevPagerfanta/twitter_bootstrap5.html.twig`.

4 lines | config/packages/babdev_pagerfanta.yaml

```yaml
babdev_pagerfanta:
    default_view: twig
    default_twig_template: '@BabDevPagerfanta/twitter_bootstrap5.html.twig'
```

Like any other config, there's no way you would know that this is the *correct* configuration just by guessing. You need to check out the docs.

If we go back and refresh... huh, nothing changed. This is a little bug that you sometimes run into in Symfony when you create a *new* configuration file. Symfony didn't *notice* it... and so it didn't know it needed to rebuild its cache. This is a *super* rare situation, but if you ever think it might be happening, it's easy enough to manually clear the cache by running:

```
php bin/console cache:clear
```

And... oh... it *explodes*. You probably noticed why. I love this error!

> There is no extension able to load the configuration for "baberdev_pagerfanta"

It's supposed to be `babdev_pagerfanta`. Whoops! And now... perfect! It's happy. And when we refresh... it sees it! In a real project, we'll probably want to add some extra CSS to make this "dark mode"... but we've *got it*.

Okay team, we're basically done! As a bonus, we're going to refactor this pagination into a JavaScript-powered *forever scroll*... except plot twist! We're going to do that without writing a single line of JavaScript. That's *next*.

Chapter 22: Forever Scroll with Turbo Frames

You've made it to the final chapter of the Doctrine tutorial!This chapter is... a *total* bonus. Instead of talking about Doctrine, we're going to leverage some JavaScript to turn this page into a "forever scroll". But don't worry! We'll talk more about Doctrine in the next tutorial when we cover Doctrine Relations.

Here's the goal: instead of pagination *links*, I want this page to load nine results like we see on Page 1.Then, when we scroll to the bottom, I want to make an AJAX request to show the *next* nine results, and so on. The *result* is a "forever scroll".

In the first tutorial in this series, we installed a library called Symfony UX Turbo,which enabled a JavaScript package called Turbo. Turbo turns all of our link clicks and form submits into AJAX calls,giving us a really nice single page app-like experience without doing anything special.

Whelp, as cool as that is, Turbo has two *other*, *optional* superpowers: Turbo Frames and Turbo Streams. You can learn all about these in our Turbo tutorial. But let's get a quick sample of how we could leverage Turbo Frames to add forever scroll without writing a *single* line of JavaScript.

turbo-frame Basics!

Frames work by dividing parts of your page into separate `turbo-frame` elements, which acts a lot like an `iframe` ... if you're old enough to remember those. When you surround something in a `<turbo-frame>` , any clicks inside of that frame will only navigate *that* one frame.

For example, open the template for this page - `templates/vinyl/browse.html.twig` - and scroll up to where we have our `for` loop. Add a new `turbo-frame` element right here. The only rule of a Turbo Frame is that it needs to have a unique ID.So say `id="mix-browse-list"` , and then go all the way to the endof that row and paste the closing tag. And, just for my own sanity, I'm going to indent that row.

54 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 2
{% block body %}
... lines 4 - 27
        <turbo-frame id="mix-browse-list">
            <div class="row">
                {% for mix in pager %}
... lines 31 - 45
                {% endfor %}
... lines 47 - 48
            </div>
        </turbo-frame>
... lines 51 - 52
{% endblock %}
```

Okay, so... what does that *do*? If you refresh the page now, any navigation inside of this frame *stays* inside the frame. Watch! If I click "2"... that *worked*. It made an AJAX request for Page 2, our app returned that *full* HTML page - including the header, footer and all - but then Turbo Frame found the matching `mix-browse-list` `<turbo-frame>` *inside* of that, grabbed its contents, and put it here.

And though it's not easy to see in *this* example, the *only* part of the page that's changing is this `<turbo-frame>` element. If I... say... messed with the title up here on my page, and then click down here and back to Page 2..that did *not* update that part of the page. Again, it works a lot like iframes, but without the weirdness.You could imagine using this, for example, to power an "Edit" button that adds inline editing.

But in our situation, this isn't very useful yet...because it works pretty much the same as before: we click the link, we see new results. The only difference is that clicking inside a `<turbo-frame>` didn't change the URL. So no matter what page I'm on, if I refresh, I'm transported right back to Page 1. So this was *kind of* a step backwards!

But stick with me. I *have* a solution, but it involves a few pieces. To start, I'm going to make the ID *unique* to the current page. Add a `-` , and then we can say `pager.currentPage` .

While you're here, also add `target="_top"` to the `turbo-frame` . That will make link clicks (lke to the mix show page) navigate the entire page, like normal.

54 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 27
        <turbo-frame id="mix-browse-list-{{ pager.currentPage }}">
... lines 29 - 49
        </turbo-frame>
... lines 51 - 54
```

Next, down at the bottom, remove the Pagerfanta links and replace them with *another* Turbo Frame. Say `{% if pager.hasNextPage %}` , and inside of it, add a `turbo-frame` , just like above, with that same `id="mix-browse-list-{{ }}"` . But this time, say `pager.nextPage` . Let me break this onto multiple lines here...and then we're also going to tell it what `src` to use for that. Oh, let me fix my typo... and then use another Pagerfanta helper called `pagerfanta_page_url` and pass that `pager` and then `pager.nextPage` . *Finally*, add `loading="lazy"` .

56 lines | templates/vinyl/browse.html.twig

```
... lines 1 - 27
        <turbo-frame id="mix-browse-list-{{ pager.currentPage }}">
            <div class="row">
... lines 30 - 47
                {% if pager.hasNextPage %}
                    <turbo-frame id="mix-browse-list-{{ pager.nextPage }}" src="{{ pagerfanta_page_url(pager, pager.nextPage) }}" loading="lazy">
                {% endif %}
            </div>
        </turbo-frame>
... lines 53 - 56
```

Woh! Lemme explain, because this is kind of wild. First, one of the super-powers of a `<turbo-frame>` is that you can give it a `src` attribute and then leave it empty. This tells Turbo:

> Hey! I'm going to be lazy and start this element empty...maybe because it's a little heavy to load. But as *soon* as this element becomes *visible* to the user, make an Ajax request to this URL to get its contents.

So, this `<turbo-frame>` will start empty... but as soon as we scroll down to it, Turbo will make an AJAX request for the next page of results.

For example, if this frame is loading for page 2, the Ajax response will contain a `<turbo-frame>` with `id="mix-browse-list-2"` . The Turbo Frame system will grab that from the Ajax response and put it here at the bottom of our list. And if there's a page 3, that will include yet *another* Turbo Frame down here that will point to Page 3.

This all might seem a bit crazy, so let's try this out. I'm going to scroll up to the top of the page, refresh and... perfect! Now scroll down here and *watch*. You should see an AJAX request show up in the web debug toolbar As we scroll... down here... ah! *There's* the AJAX request! Scroll down again and... there's a *second* AJAX request: one for Page 2 and one for Page 3. If we keep scrolling, we run out of results and reach the bottom of the page.

If you're new to Turbo Frames, that concept may have been a little confusing, but you can learn more on our Turbo tutorial. And a shout-out to an AppSignal blog post that introduced this cool idea.

All right, team! Congrats on finishing the Doctrine course! I hope you're feeling *powerful*. You should be! The only major missing part of Doctrine now is Doctrine Relations: being able to associate one entity to another through relationships, like many-to-one and many-to-many. We'll cover all of that in the next tutorial. Until then, if you have any questions or have a great riddle you want to ask us, we're here for you in the comments section. Thanks a lot, friends! And see you next time!