

Doctrine, Symfony & the Database



With <3 from SymfonyCasts

Chapter 1: Installing Doctrine

Well hey friends! And bienvenidos to our tutorial about learning Spanish! What? That's next week? Doctrine?

Ah: welcome to our tutorial *all* about making Symfony talk to a database... in English.

We learned a *ton* in the first two courses of this series, especially the last tutorial where we demystified services, autowiring and configuration. That hard work is about to pay off as we take our app to the next level by adding a database. That's going to make things way, *way* more interesting.

[Who is Doctrine Exactly?](#)

In truth, Symfony has *no* database layer at all. Instead, it leverages *another* library called Doctrine, which has been around for a long time and is *incredible*. Symfony and Doctrine are, sort of, the BFF's of programming, the Batman and Robin of web development, the Bert & Ernie of HTTP! They're both powerful, but they have *such* a strong integration that it *feels* like you're using one library.

And not only is Doctrine powerful, but it's also easy to use. I'll admit that this was *not* always the case. But Doctrine is now more accessible and fun to use than *ever* before. I think you're going to love it.

[Project Setup](#)

To learn the *most* about Doctrine - and to become the third amigo - you should *definitely* code along with me by downloading the course code from this page. After you unzip the file, you'll find a *start/* directory with the same code that you see here. Check out this [README.md](#) file for all the setup fun!

The last step will be to open a terminal and use the Symfony binary to start a local web server - you can download the binary at <https://symfony.com/download>. Run:

A terminal window with a dark background and three light gray window control buttons in the top-left corner. The command prompt is \$ symfony serve -d.

This starts a web server in the background on port 8000. I'll copy the URL, spin over to my browser and say hello to... Cauldron Overflow! Our question and answer site for witches and wizards: a place to debug *what* went wrong when you tried to make your *cat* invisible and instead made your *car* invisible.

So far, we have a homepage that lists questions and you can view each individual question and its answers. But... this is *all* hardcoded! None of this is coming from a database... yet. *That* is our job.

[Installing Doctrine](#)

Now, remember: Symfony starts small: it does not come with every feature and library that you might *ever* need. And so, Doctrine is *not* installed yet.

To get it, find your terminal and run:

A terminal window with a dark background and three light gray window control buttons in the top-left corner. The command prompt is \$ composer require orm.

[Auto-Unpacked Packs](#)

Let's... "unpack" this command!

First, *orm* is one of those Symfony Flex *aliases*. We only need to say *composer require orm* but, in reality, this is a shortcut for a library called *symfony/orm-pack*.

Also, we talked about "packs" in a previous course. A pack is a, sort of, fake package that exists simply to help

you install several *other* packages.

Let me show you: copy the name of the package, and go open it in GitHub: <https://github.com/symfony/orm-pack>. Yep! It's nothing more than a *single* `composer.json` file! The whole point of this library is that it requires a few *other* packages. That means that we can `composer require` this one package, but in reality, we will get all four of these libraries.

Now, one of the *other* packages that we have in our project is `symfony/flex`, which is what powers the alias and recipe systems. Starting in `symfony/flex` version 1.9 - which I *am* using in this project - when you install a pack, Flex does something special.

Go and look at your `composer.json` file. What you would *expect* to see is one new line for `symfony/orm-pack`: the one library that we just required. In reality, Composer *would* also download its 4 dependencies... but only the pack would show up here. But... surprise! Instead of `symfony/orm-pack`, the 4 packages *it* requires are here instead!

```
84 lines | composer.json
1  {
2  ... lines 2 - 3
4  "require": {
5  ... lines 5 - 7
8      "composer/package-versions-deprecated": "^1.8",
9      "doctrine/doctrine-bundle": "^2.1",
10     "doctrine/doctrine-migrations-bundle": "^3.0",
11     "doctrine/orm": "^2.7",
12 ... lines 12 - 26
27 },
28 ... lines 28 - 82
83 }
```

Here's the deal: before `symfony/flex` 1.9, when you required a pack, nothing special happened: Composer added the *one* new package to `composer.json`. But starting in `symfony/flex` 1.9, instead of adding the pack, it adds the *individual* libraries that the pack requires: these 4 lines. It does this because it makes it *much* easier for us to manage the versions of each package independently.

The point is: a pack is *nothing* more than a shortcut to install *several* packages. And in the latest version of Flex, it adds those "several" packages to your `composer.json` file automatically to make life easier.

[DoctrineBundle Recipe & DATABASE_URL](#)

Anyways, if we scroll down... you can ignore this `zend-framework` abandoned warning. That's a distant dependency and it won't cause us problems. And... ah! It looks like this installed *two* recipes... and one of those gives us a *nice* set of instructions at the bottom. We'll learn *all* about this.

If you're using the latest version of Symfony Flex, this installation command will ask you if you want to also include some Docker configuration. Feel free to choose whatever you want, but we *will* use Docker to help connect to the database in this tutorial.

To see what the recipes did, I'll clear my screen and say:

```
$ git status
```

Ok: in addition to the normal files that we expect to be modified, the recipe also modified `.env` and created some *new* files.

Go check out `.env`. At the bottom... here it is: it added a new `DATABASE_URL`. This is the environment variable that Doctrine uses to connect to the database.

The default `DATABASE_URL` now uses PostgreSQL, but there is a commented-out MySQL example above if you prefer that. But in both cases, if you use our Docker integration (keep watching!) then you won't need to configure `DATABASE_URL` manually.

```
30 lines | .env
... lines 1 - 22
23 ###> doctrine/doctrine-bundle ###
24 # Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#conn
25 # For an SQLite database, use: "sqlite:///kernel.project_dir%/var/data.db"
26 # For a PostgreSQL database, use: "postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=11&char
27 # IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
28 DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
29 ###
```

And... we can see this! The recipe *also* added another file called `config/packages/doctrine.yaml`

This file is responsible for configuring DoctrineBundle. And you can actually *see* that this `doctrine.dbal.url` key *points* to the environment variable! We won't need to do much work in this file, but I wanted you to see that the environment variable is *passed* to the bundle.

```
19 lines | config/packages/doctrine.yaml
1 doctrine:
2   dbal:
3     url: '%env(resolve:DATABASE_URL)%'
4
5     # IMPORTANT: You MUST configure your server version,
6     # either here or in the DATABASE_URL env var (see .env file)
7     #server_version: '5.7'
8   orm:
9     auto_generate_proxy_classes: true
10    naming_strategy: doctrine.orm.naming_strategy.underscore_number_aware
11    auto_mapping: true
12    mappings:
13      App:
14        is_bundle: false
15        type: annotation
16        dir: '%kernel.project_dir%/src/Entity'
17        prefix: 'App\Entity'
18        alias: App
```

The recipe also added a few directories `src/Entity/` , `src/Repository/` , and `migrations/` , which we'll talk about soon.

So all we need to do to start working with Doctrine is configure this `DATABASE_URL` environment variable to point to a database that we have running somewhere.

To do that, we're going to do something *special* in this tutorial. Instead of telling you to install MySQL locally, we're going to use Docker. If you already use Docker, great! But if you *haven't* used Docker... or you tried it and didn't like it, give me just a *few* minutes to convince you - I think you're going to *love* how Symfony integrates with Docker. That's next!

Chapter 2: make:docker:database

Doctrine is installed! Woo! Now we need to make sure a database is running - like MySQL or PostgreSQL - and then update the `DATABASE_URL` environment variable to point to it.

```
30 lines | .env
... lines 1 - 27
28 DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
... lines 29 - 30
```

So: you can *absolutely* start a database manually: you can download MySQL or PostgreSQL onto your machine and start it. *Or* you can use Docker, which is what *we* will do. OooOoooo.

[Using Docker?](#)

Now, hold on: if you're nervous about Docker... or you haven't used it much... or you used it and hated it, stay with me! Using Docker *is* optional for this tutorial, but we're going to use it in a very lightweight way.

The only requirement to get started is that you need to have Docker downloaded and running on your machine. I already have Docker running on my machine for Mac.

Docker is *all* about creating tiny *containers* - like a container that holds a MySQL instance and another that holds a PHP installation. Traditionally, when I think of Docker, I think of a *full* Docker setup: a container for PHP, a container for MySQL and another container for Nginx - all of which communicate to each other. In that situation, you don't have *anything* installed on your "local" machine except for Docker itself.

That "full Docker" setup is great - and, if you like it, awesome. But it also adds complexity: sharing source code with the containers can make your app *super* slow - especially on a Mac - and if you need to run a `bin/console` command, you need to execute that from *within* a Docker container.

[Our Simple Docker Plan](#)

And so, instead, we're going to do something *much* simpler. First, we *are* going to have PHP installed on our local machine - I *do* have PHP installed on my Mac. Then, we're *just* going to use Docker to launch *services* like MySQL, Redis or Elasticsearch. Finally, we'll configure our local PHP app to communicate with those containers.

For me, it's kind of the best of both worlds: it makes it super easy to launch services like MySQL... but without the complexity that often comes with Docker.

[Hello make:docker:database](#)

Ok, ready? To manage our Docker containers, we need to create a `docker-compose.yml` file that describes what we need.

If you're using the latest version of Symfony Flex, then when you ran `composer require orm`, it asked you if you wanted Docker configuration. If you answered yes, congrats! You can skip this step because you already have a `docker-compose.yml` file with a `database` service inside. Skip ahead to around 4:30 when we run `docker-compose up`.

That file is pretty simple but... let's cheat! Find your terminal and run:

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

This command comes from MakerBundle version 1.20... and I *love* it. A big thanks to community member [Jesse Rushlow](#) for contributing this!

Ok: it doesn't see a `docker-compose.yml` file, so it's going to create a new one. I'll use MySQL and, for the version - I'll use `latest` - we'll talk more about that in a few minutes.

And... done! The database service is ready!

Well, in reality, the *only* thing this command did was create a `docker-compose.yml` file: it didn't communicate with Docker or start any containers - it *just* created this new `docker-compose.yml` file.

```
13 lines | docker-compose.yml
1  version: '3.7'
2  services:
3    database:
4      image: 'mysql:latest'
5      environment:
6        MYSQL_ROOT_PASSWORD: password
7      ports:
8        # To allow the host machine to access the ports below, modify the lines below.
9        # For example, to allow the host to connect to port 3306 on the container, you would change
10       # "3306" to "3306:3306". Where the first port is exposed to the host and the second is the container port.
11       # See https://docs.docker.com/compose/compose-file/#ports for more information.
12       - '3306'
```

And... it's pretty basic: we have a service called `database` - that's just an internal name for it - which uses a `mysql` image at its `latest` version. And we're setting an environment variable in the container that makes sure the root user password is... `password` ! At the bottom, the `ports` config means that port 3306 of the container will be exposed to our *host* machine.

That last part is important: this will make it possible for our PHP code to talk *directly* to MySQL in that container. This syntax actually means that port 3306 of the container will be exposed to a *random* port on our host machine. Don't worry: I'll show you exactly what that means.

[docker-compose up](#)

So... yay! We have a `docker-compose.yml` file! To start *all* of the containers that are described in it... which is just one - run:

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

The `-d` means "run as a daemon" - it runs in the background instead of holding onto my terminal.

The first time you run this it will take a bit longer because it needs to download MySQL. But eventually.... yes! With one command, we now have a database running in the background!

So... how do we communicate with it? Next, let's learn a bit more about `docker-compose` including how to connect to the MySQL instance *and* shut down the container.

Chapter 3: docker-compose & Exposed Ports

We just started our MySQL docker container thanks to `docker-compose`. So... ah... now what? How can we *talk* to that database? Great question!

Start by running just:

```
$ docker-compose
```

This lists *all* the commands that you can use with `docker-compose`. Most of these we won't need to worry about. But one good one is `ps`, which stands for "process status". Try it:

```
$ docker-compose ps
```

This shows *all* the containers that `docker-compose` is running for this project... which is just one right now. Ah, and check this out! Port `3306` of the container is being shared to our local machine on port `32773`. This is a *random* port number that will be different each time we restart the container.

[Connecting to the MySQL Docker Container](#)

This means that we can *talk* to the MySQL server in the container via port `32773`! Let me show you. I actually *do* have `mysql` installed on my local machine, so I can say `mysql -u root --password=password` because, in our `docker-compose.yml` file, that's what we set the root user password to. Then `--host=127.0.0.1` - to talk to my local computer - and `--port=` set to this one right here: `32773`. Try it!

```
$ mysql -u root --password=password --host=127.0.0.1 --port=32773
```

Boom! We are *inside* of the container talking to MySQL! By the way, if you *don't* have MySQL installed locally, you can also do this by running:

```
$ docker-compose exec database mysql -u root --password=password
```

That will "execute" the `mysql` command inside the container that's called `database`.

Anyways, now that we're here, we can do normal stuff like:

```
$ SHOW DATABASES
```

... or even create a new database called `docker_coolness`:

```
$ CREATE DATABASE docker_coolness
```

There it is! I'll type `exit` to get out.

[Stopping and Destroying the Containers](#)

When you're done with the containers and want to turn them off, you can do that with:

```
$ docker-compose stop
```

Or the more common:

```
$ docker-compose down
```

This loops through all of the services in `docker-compose.yml` and, not only *stops* each container, but also *removes* its image. It's like completely deleting that "mini server" - including its data.

Thanks to that, the next time we run:

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

It will create the whole container from scratch: any data from before will be gone.

[Booting isn't Instant](#)

Let's see the whole process from the start. First, run:

```
$ docker-compose down
```

to stop and destroy the container. If we try to connect to MySQL now it, of course, fails. Now run:

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

To start the container. Let's check on the process:

```
$ docker-compose ps
```

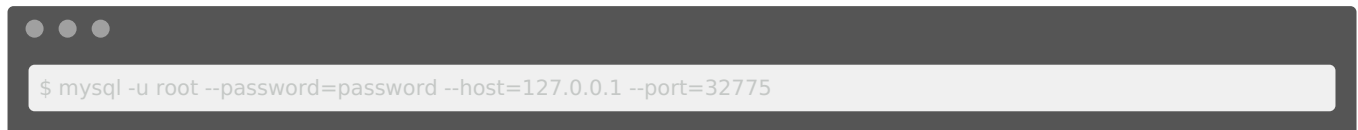
Ah! Look at that port! It was `32773` the first time we ran it. *Now* the container is exposed on port `32775`. Let's try connecting:

```
$ mysql -u root --password=password --host=127.0.0.1 --port=32775
```

And... oh! It didn't work!

```
Lost connection to MySQL server
```


Ah. The truth is that, even though it *looks* like `docker-compose up` is instant, in reality, it takes a few seconds for MySQL to *truly* start. Eventually if we try again...

A terminal window with a dark background and three light gray window control buttons (minimize, maximize, close) in the top-left corner. A light gray text input field contains the command: `$ mysql -u root --password=password --host=127.0.0.1 --port=32775`.

```
$ mysql -u root --password=password --host=127.0.0.1 --port=32775
```

Yes! We are in! But you won't see the `docker_coolness` database that we created earlier because `docker-compose down` destroyed our data.

At this point, we've created a `docker-compose.yml` file and used `docker-compose` to launch a MySQL container that we can talk to. Awesome!

To *connect* to this from our Symfony app, all *we* need to do is update the `DATABASE_URL` environment variable to use the right password and port.

But... we're *not* going to do that. It *would* work... but it turns out that our app is *already* aware of the correct `DATABASE_URL` value... even though we haven't configured anything. Let's talk about *how* next.

Chapter 4: docker-compose Env Vars & Symfony

Thanks to our `docker-compose.yml` file and the `docker-compose up` command, we started a MySQL container in Docker. You can prove it by running:



```
$ docker-compose ps
```

Yep! Port 3306 of the container is being exposed to my host machine on port 32776, which is a random port that will change each time we run `docker-compose up`.

[Configure DATABASE_URL Manually?](#)

Whether you followed me through the Docker setup... or decided to install MySQL on your own, we're now at the same point: we need to update the `DATABASE_URL` environment variable to point to the database.

Normally I would copy `DATABASE_URL`, go into `.env.local`, paste, and update it to whatever my local settings are, like `root` user, no password and a creative database.

[DATABASE_URL From Docker](#)

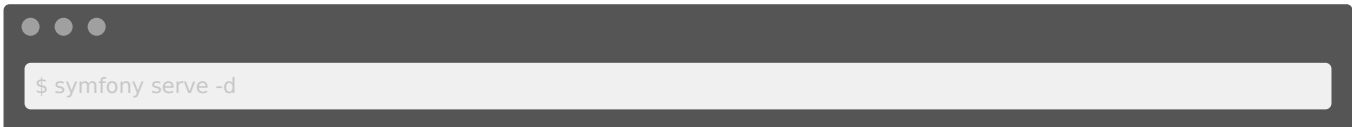
But... I'm *not* going to do that. Why? Because the `DATABASE_URL` environment variable is *already* configured correctly!

Let me show you. When we started our app, we used the Symfony binary to create a local web server. I'll run:



```
$ symfony server:stop
```

to stop it... just so I can show you the command we used. It was:



```
$ symfony serve -d
```

If you're using Ubuntu and running Docker in "root" mode, then you will need to run `sudo symfony serve -d` for Symfony to see the Docker environment variables. Later, when we use things like `symfony console`, you will also need to use `sudo`. Note that this may cause some cache file ownership issues in your Symfony app while developing. Ok, have fun!

That started a web server at `localhost:8000`. So: what we're seeing in the browser is being *served* by the `symfony` binary.

Well... surprise! The Symfony binary has *special* integration with Docker! It *detects* that we have a `docker-compose.yml` file in this project, loops over all of the running services, reads their config and exposes *real* environment variables to our app with the connection details for each one.

For example, because this service is called `database` - we technically could have called it anything - the Symfony binary is *already* exposing an environment variable called `DATABASE_URL`: the *exact* environment variable that Doctrine is looking for.

[Dumping the Environment Variable](#)

I'll show you *exactly* what I mean. First, go back to your browser, watch the bottom right of the web debug toolbar, and refresh. Ah! The little "Server" icon turned green! This is info about the Symfony web server... and *now* it says "Env Vars from Docker".

Back at your editor, open up `public/index.php` : the front controller for our project. We normally don't need to mess with this file... but let's temporarily hack in some code. After the autoload line, add `dd($_SERVER)` .

```
33 lines | public/index.php
... lines 1 - 7
8  require dirname(__DIR__).'/vendor/autoload.php';
9
10 dd($_SERVER);
11
12 (new Dotenv())->bootEnv(dirname(__DIR__).'/.env');
... lines 13 - 33
```

The `$_SERVER` global variable holds a lot of things *including* any real *environment* variables that are passed to PHP. Back at the browser, refresh and search for "database". Check it out! A `DATABASE_URL` environment variable!

That is being set by the Symfony binary, which is reading the info dynamically from Docker. It has all the correct info including port 32776.

When a *real* environment variable exists, it *overrides* the value that you have in `.env` or `.env.local` . In other words, as *soon* as we run `docker-compose up` , our app has access to a `DATABASE_URL` environment variable that points to the Docker container. We don't need to configure *anything*!

[Seeing the Environment Variables](#)

Back in `index.php` , remove the `dd()` line.

```
31 lines | public/index.php
... lines 1 - 7
8  require dirname(__DIR__).'/vendor/autoload.php';
9
10 (new Dotenv())->bootEnv(dirname(__DIR__).'/.env');
... lines 11 - 31
```

Another way to see what environment variables the Symfony binary is exporting to our app is by running:

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

And... yes! This has `DATABASE_URL` and some other `DATABASE` variables that you can use for each part... if you need to. If we added a *second* service to `docker-compose` - like a Redis container - then *that* would show up here too.

The big picture is this: all we need to do is run `docker-compose up -d` and our Symfony app is immediately setup to talk to the database. I *love* that.

But... we can't really *do* anything yet... because the MySQL instance is empty! Next, let's create our database and make sure that Doctrine knows *exactly* which version of MySQL we're using.

Chapter 5: doctrine:database:create & server_version

We have a Docker database container running *and* our app is *instantly* configured to talk to it thanks to the Symfony web server. But... we can't really *do* anything yet... because that MySQL instance is empty! In `var:export`, you can see that the database *name* is apparently "main". But that does *not* exist yet.

No problem! When we installed Doctrine, it added a *bunch* of new `bin/console` commands to our app. Run:



```
$ php bin/console
```

and scroll up to find a *huge* list that start with `doctrine:`. The *vast* majority of these are not very important - and we'll talk about the ones that *are*.

[The "symfony console" Command](#)

One of the handy ones is `doctrine:database:create`, which reads the database config and creates the database. So, in our case, it should create a database called `main`.

Ok! Copy the command name and run:

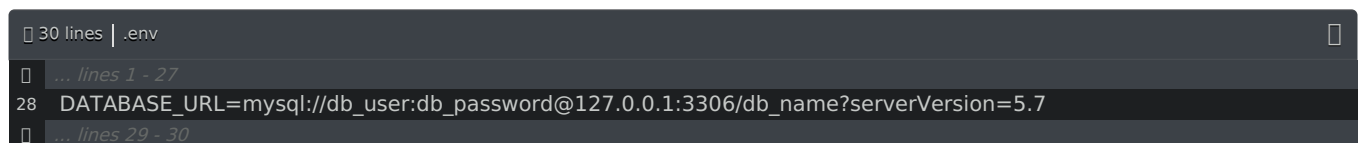


```
$ php bin/console doctrine:database:create
```

And... yikes!

```
Access denied for db_user at localhost.
```

Huh. For some reason, it's using this `DATABASE_URL` from `.env` *instead* of the one that's set by the Symfony binary.



```
30 lines | .env
... lines 1 - 27
28 DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
... lines 29 - 30
```

The problem is that, when you load your site in the browser, this is processed through the Symfony web server. That allows the Symfony binary to inject all of the environment variables.

But when you just run a random `bin/console` command, that does *not* use the symfony binary. And so, it does *not* have an opportunity to add the environment variables.

No worries! There is, of course, a solution. Instead of running:



```
$ php bin/console
```

We'll run:



```
$ symfony console
```

`symfony console` literally means `bin/console` ... but because we're running it through the Symfony executable, it *will* inject the environment variables that are coming from Docker.

The latest version of MakerBundle generates a `MYSQL_DATABASE: main` config into your `docker-compose.yml` file. If you have this, then the database will already be created! Feel free to run the command just in case ;).

So:

A terminal window with a dark background and three window control buttons in the top-left corner. The command prompt shows the command `$ symfony console doctrine:database:create` entered.

And... boom! We have a database!

Docker Image Versions

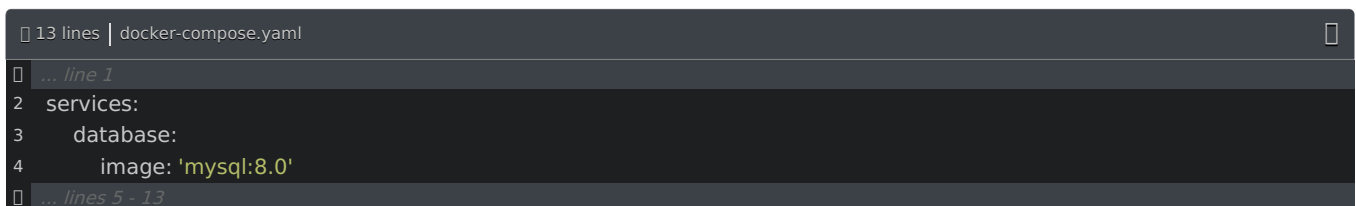
Before we jump into fun stuff like generating entities and database tables, I want to tighten up one more thing. Open up `docker-compose.yml`. The `:latest` next to the image means that we want to use the *latest* version of MySQL. Where does that image come from?

A code editor window showing the `docker-compose.yml` file. The file has 13 lines. The visible content is:

```
1 ... line 1
2 services:
3   database:
4     image: 'mysql:latest'
```

Google for Docker hub to find <https://hub.docker.com>. When you say that you want a `mysql` image at version `latest`, Docker communicates back to Docker Hub to get the details. Search for MySQL for *all* the info about that image including the *tags* that are currently available. Right now, the `latest` tag is equal to 8.0.

Head back over to `docker-compose.yml`. You don't have to do this, but I'm going to change `latest` to `8.0` so that I'm locked at a specific version that won't suddenly change.

A code editor window showing the `docker-compose.yml` file. The file has 13 lines. The visible content is:

```
1 ... line 1
2 services:
3   database:
4     image: 'mysql:8.0'
```

Over at the terminal, even though `latest` and `8.0` are *technically* the same image, let's restart `docker-compose` anyways to update the image. Run:

A terminal window with a dark background and three window control buttons in the top-left corner. The command prompt shows the command `$ docker-compose down` entered.

And then:

A terminal window with a dark background and three window control buttons in the top-left corner. The command prompt shows the command `$ docker-compose up -d` entered.

It quickly downloaded the new image... which was probably just a "pointer" to the same image we used before.

Setting server_version

In newer versions, the `server_version` config may not be in your `doctrine.yaml` file, but you can add it manually.

Now that we've set the MySQL version in Docker, we should *also* do the same thing with Doctrine. Open up `config/packages/doctrine.yaml`. See that `server_version` key?

```
19 lines | config/packages/doctrine.yaml
1 doctrine:
2   dbal:
3   ... lines 3 - 4
5     # IMPORTANT: You MUST configure your server version,
6     # either here or in the DATABASE_URL env var (see .env file)
7     #server_version: '5.7'
8   ... lines 8 - 19
```

Set this to 8.0. If you're using mariadb, you can use a format like `mariadb-10.5.4`.

```
19 lines | config/packages/doctrine.yaml
1 doctrine:
2   dbal:
3   ... lines 3 - 6
7     server_version: '8.0'
8   ... lines 8 - 19
```

This is... kind of an annoying thing to set, but it *is* important. It tells Doctrine what *version* of MySQL we're running so that it knows what features are supported. It uses that to adjust the exact SQL it generates. Make sure that your production database uses this version or *higher*.

Next, let's create our *first* database table by generating an entity.

Chapter 6: Entity Class

Doctrine is an ORM: an object relational mapper. That's a fancy way of saying that, for each table in the database, we will have a corresponding class in PHP. And for each column on that table, there will be a property in that class. When you query for a row in a table, Doctrine will give you an *object* with that row's data set on the properties.

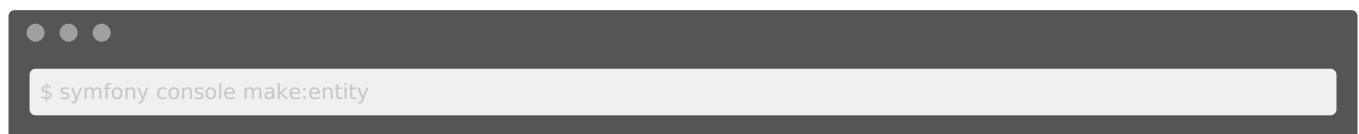
So if you want to create a database table in Doctrine, the way you do that is actually by creating the class it will map to. These "classes" that map to tables have a special name: entity classes.

[make:entity](#)

You *can* create an entity class by hand - there's nothing very special about them. But... come on! There's a *much* easier way. Find your terminal and run one of my *favorite* `bin/console` commands:

A terminal window with a dark background and three window control buttons in the top left. The command prompt shows '\$ php bin/console make:entity'.

You can also run:

A terminal window with a dark background and three window control buttons in the top left. The command prompt shows '\$ symfony console make:entity'.

It doesn't matter in this case, because this command won't talk directly to the database, and so, it doesn't need the environment variables. This command *just* generates code.

Now remember: the *whole* point of our site is for witches and wizards to ask questions and then post answers. So the *very* first thing we need is a `Question` entity. So, a `question` table in the database.

Enter `Question`. The command immediately starts asking what properties we want, which really, also means what *columns* we want in the table. Let's add a few. Call the first one `name` - that will be the short name or "title" of the question like "Reversing a Spell". The command then asks what "type" we want. Doctrine has its *own* type system: enter "?" to see the full list.

These aren't MySQL column types, but each maps to one. For example, a `string` maps to a `VARCHAR` in MySQL. And there are a bunch more.

In our case, choose `string`: that's good for any text 255 characters or less. For the column length, I'll use the default 255. The next question says:

Can this field be nullable in the database?

Say "No". This means that the column will be *required* in the database.

And... congrats! We just added our first field! Let's add a few more. Call the next `slug`: this will be the URL-safe version of the name that shows up in the URL. It will also be a string, let's set its length to 100 and, once again, I'll say "no" to "nullable".

For the actual body of the question, let's call it `question`. This will be *long*, so we can't use the `string` type. Instead, use `text`... and make this *also* required in the database.

Add *one* more field: `askedAt`. This will be a *date* field - kind of like a "published at" field. For the type, ooh! It recommends `datetime` - that is *exactly* what we want! Hit enter and this time say "yes" to nullable. The idea is that users will be able to *start* writing a question and save it to the database with a `null` `askedAt` because it's

not finished. When the user *is* finished and ready to post it, *then* we would set that value.

And... we're done! Hit enter one more time to exit `make:entity`.

Hello Entity Class

So... what did this do? Well, first, I'll tell you that this made absolutely *no* changes to our database: we do *not* have a `question` database table yet.

If you scroll back up to the top of the command, you can see that it created 2 files: `src/Entity/Question.php` and `src/Repository/QuestionRepository.php`.

For now, *completely* ignore the repository class: it's not important yet and we'll talk about it later.

This entity class, however, is *super* important. Go open it up: `src/Entity/Question.php`.

```
93 lines | src/Entity/Question.php
... lines 1 - 2
3 namespace App\Entity;
4
5 use App\Repository\QuestionRepository;
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity(repositoryClass=QuestionRepository::class)
10 */
11 class Question
12 {
13     /**
14      * @ORM\Id()
15      * @ORM\GeneratedValue()
16      * @ORM\Column(type="integer")
17      */
18     private $id;
19
20     /**
21      * @ORM\Column(type="string", length=255)
22      */
23     private $name;
24
25     /**
26      * @ORM\Column(type="string", length=100)
27      */
28     private $slug;
29
30     /**
31      * @ORM\Column(type="text")
32      */
33     private $question;
34
35     /**
36      * @ORM\Column(type="datetime", nullable=true)
37      */
38     private $askedAt;
39 ... lines 39 - 91
92 }
```

As we talked about, we're going to have *one* class per table.

The first thing I want you to notice is that... there's nothing special about this class. It's a boring, normal class... it doesn't even extend a base class! It has several private properties and, to access those, the command generated getter and setter methods, like `getName()`, `getSlug()` and `setSlug()`.


```

93 lines | src/Entity/Question.php
... lines 1 - 10
11 class Question
12 {
... lines 13 - 39
40 public function getId(): ?int
41 {
42     return $this->id;
43 }
44
45 public function getName(): ?string
46 {
47     return $this->name;
48 }
49
50 public function setName(string $name): self
51 {
52     $this->name = $name;
53
54     return $this;
55 }
56
57 public function getSlug(): ?string
58 {
59     return $this->slug;
60 }
61
62 public function setSlug(string $slug): self
63 {
64     $this->slug = $slug;
65
66     return $this;
67 }
68
69 public function getQuestion(): ?string
70 {
71     return $this->question;
72 }
73
74 public function setQuestion(string $question): self
75 {
76     $this->question = $question;
77
78     return $this;
79 }
80
81 public function getAskedAt(): ?\DateTimeInterface
82 {
83     return $this->askedAt;
84 }
85
86 public function setAskedAt(?\DateTimeInterface $askedAt): self
87 {
88     $this->askedAt = $askedAt;
89
90     return $this;
91 }
92 }

```

It's just about the most *boring* class you'll ever see.

But of course, if Doctrine is going to map this class and its properties to a database table, it needs to know a few things. For example, it needs to know that the `name` property should map to a `name` column and that its

`type` is a `string`.

```
93 lines | src/Entity/Question.php
... lines 1 - 10
11 class Question
12 {
... lines 13 - 19
20 /**
21  * @ORM\Column(type="string", length=255)
22  */
23 private $name;
... lines 24 - 91
92 }
```

The way that Doctrine does this is by reading annotations. Well, you can also use XML, but I *love* annotations.

For example, the `@ORM\Entity()` above the `class` is what actually tells Doctrine:

```
93 lines | src/Entity/Question.php
... lines 1 - 7
8 /**
9  * @ORM\Entity(repositoryClass=QuestionRepository::class)
10 */
11 class Question
12 {
... lines 13 - 91
92 }
```

Hey! This is not just a normal, boring PHP class. This is an "entity": a class that I want to be able to store in the database.

Then, the `@ORM\Column()` above the properties is how Doctrine knows which properties should be stored in the table and their types.

[Annotations Reference](#)

Now, there are a *bunch* of different annotations and options you can use to configure Doctrine. Most are pretty simple - but let me show you an *awesome* reference. Search for [doctrine annotations reference](#) to find a *really* nice spot on their site where you can see a list of *all* the different possible annotations and their options.

For example, `@Column()` tells you all the different options that you can pass to it, like a `name` option. So if you wanted to control the `name` of the slug column - instead of letting Doctrine determine it automatically - you would do that by adding a `name` option.

One of the other annotations that you'll see here is `@Table`. Oh, and notice, in the docs, the annotation will be called `@Table`. But inside Symfony, we always use `@ORM\Table`. Those are both referring to the same thing.

Anyways, if you wanted to control the name of the table, we could say `@ORM\Table()` and pass it `name=""` with some cool table name.

But I won't bother doing that because Doctrine will guess a good table name from the class. Oh, and by the way, the auto-completion that you're seeing on the annotations comes from the "PHP Annotations" plugin in PhpStorm.

Ok: status check! The `make:entity` command helped us create this new entity class, but it did *not* talk to the database. There is *still* no `question` table.

How *do* we create the table? By generating a *migration*. Doctrine's migrations system is *amazing*. It will even allow us to perform table *changes* with basically zero work. Let's find out how next.

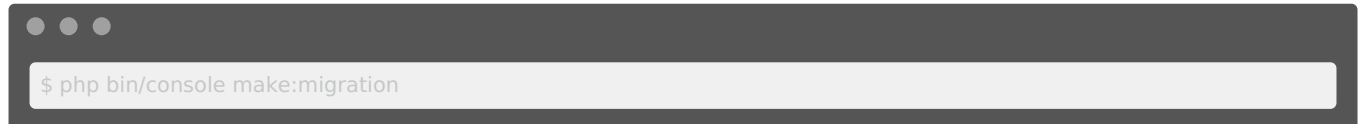
Chapter 7: Migrations

We have a beautiful new `Question` entity class that is *supposed* to map to a `question` table in the database. But... that table does *not* exist yet. How can we create it?

Well, because Doctrine has all of this configuration about the entity, like the fields and field types, it should - in theory - be able to create the table *for* us. And... it absolutely *can*!

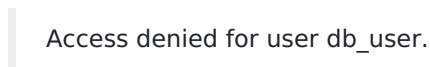
[Hello make:migration](#)

The mechanism we use to make database *structure* changes is called migrations. At your terminal, run:

A terminal window with a dark background and three light gray window control buttons in the top-left corner. The terminal text is light gray on a white background. It shows the command `$ php bin/console make:migration` entered at the prompt.

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

And... that fails:

A light gray rectangular box with a thin vertical light gray border on the left side. It contains the text "Access denied for user db_user." in a dark gray font.

```
Access denied for user db_user.
```

Of course: the command doesn't have access to the Docker environment variables. I *meant* to run:

A terminal window with a dark background and three light gray window control buttons in the top-left corner. The terminal text is light gray on a white background. It shows the command `$ symfony console make:migration` entered at the prompt.

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

This time... cool! It generated a new *file* inside of a `migrations/` directory. Let's go check it out! In `migrations/` open the one new file and... awesome! It has an `up()` method with the *exact* SQL we need!

```

32 lines | migrations/Version20200707173854.php
... lines 1 - 4
5 namespace DoctrineMigrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
9
10 /**
11  * Auto-generated Migration: Please modify to your needs!
12  */
13 final class Version20200707173854 extends AbstractMigration
14 {
15     public function getDescription() : string
16     {
17         return '';
18     }
19
20     public function up(Schema $schema) : void
21     {
22         // this up() migration is auto-generated, please modify it to your needs
23         $this->addSql('CREATE TABLE question (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255) NOT NULL, slug VA
24     }
25
26     public function down(Schema $schema) : void
27     {
28         // this down() migration is auto-generated, please modify it to your needs
29         $this->addSql('DROP TABLE question');
30     }
31 }

```

CREATE TABLE question...

and then all of the columns.

The `make:migration` command is *smart*: it compares the *actual* database - which has zero tables at the moment - with all of our entity classes - just one right now - and then generates the SQL needed to make the database *match* those entities.

It saw the one `Question` entity... but no `question` table, and so, it generated the `CREATE TABLE` statement.

Executing Migrations

But this query has *not* been *executed* yet. To do that, run:

```

$ php bin/console doctrine:migrations:migrate

```

Ah, but be careful: we can't use `bin/console` directly. Instead run:

```

$ symfony console doctrine:migrations:migrate

```

And... congratulations! We have a new `question` table in the database!

How Executed Migrations are Tracked

The way the migration system works is really cool. Run another command:

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

This shows all the migrations in your app, which is just one right now. Next to that migration is says "Status Migrated". How does it know that?

Behind the scenes, the migration system created a table in the database called `doctrine_migration_versions`. Each time it executes a migration file, it adds a new row to that table that *records* that it was executed.

That means that later, if you run

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

again... it's smart enough to *not* execute the same migration twice. It looks at the table, sees that it already ran this, and skips it.

When you deploy to production, you'll *also* run `doctrine:migrations:migrate`. When you do that, it will check the `doctrine_migration_versions` table in the *production* database and execute *any* new migrations.

Making a Column Unique

Before we keep going, you know what? When we created the `Question` entity, I forgot to do something. The `slug` column should *really* be unique in the database because we will eventually use that part of the URL to query for the *one* `Question` that matches.

One of the options you can pass to `@ORM\Column()` is `unique=true`.

93 lines | src/Entity/Question.php

```
... lines 1 - 10
11 class Question
12 {
... lines 13 - 24
25 /**
26  * @ORM\Column(type="string", length=100, unique=true)
27  */
28  private $slug;
... lines 29 - 91
92 }
```

That won't change how our *PHP* code behaves - this doesn't relate to form validation or anything like that. This *simply* tells Doctrine:

Hey! I want this column to have a unique constraint in the database

Of course... just making this change did *not* somehow magically add the unique constraint to the database. To do that, we need to generate another migration.

Cool! At your terminal, once again run:

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

to generate a *second* migration file. Let's go check it out.

```

32 lines | migrations/Version20200707174149.php
... lines 1 - 4
5 namespace DoctrineMigrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
9
10 /**
11  * Auto-generated Migration: Please modify to your needs!
12  */
13 final class Version20200707174149 extends AbstractMigration
14 {
15     public function getDescription() : string
16     {
17         return '';
18     }
19
20     public function up(Schema $schema) : void
21     {
22         // this up() migration is auto-generated, please modify it to your needs
23         $this->addSql('CREATE UNIQUE INDEX UNIQ_B6F7494E989D9B62 ON question (slug)');
24     }
25
26     public function down(Schema $schema) : void
27     {
28         // this down() migration is auto-generated, please modify it to your needs
29         $this->addSql('DROP INDEX UNIQ_B6F7494E989D9B62 ON question');
30     }
31 }

```

And... woh! It's a **CREATE UNIQUE INDEX** statement for the **slug** column! The migrations system compared the **question** table in the database to the **Question** entity, determined that the only difference was a missing unique index and then generated the SQL to add it. Honestly, that's amazing.

Let's go run it:

```

$ symfony console doctrine:migrations:migrate

```

This sees *both* migrations, but only runs the *one* that hasn't been executed yet. The **slug** column is now unique in the database.

So this is the workflow: create a new entity or change an existing entity, run **make:migration**, and then execute it with **doctrine:migrations:migrate**. This keeps your database in sync with your entity classes *and* give you a set of migrations that you can run when you deploy to production.

Next: it's time to create some **Question** objects in PHP and see how we can save those to the question table.

Chapter 8: Persisting to the Database

We have a beautiful entity class and, thanks to the migrations that we just executed, we have a corresponding `question` table in the database. Time to insert some data!

Think "Classes", not "Tables"

One of the *key* philosophies of Doctrine is that it doesn't want you to think about tables and columns. Doctrine wants you to think about classes and properties... and then leave all the details of saving and querying to a database table up to *it* to worry about.

So instead of asking:

How can I insert a new row in the `question` table?

We need to think:

Let's create a `Question` object, populate it with data and then ask Doctrine to save it.

Creating a Question Endpoint

To play with all of this, let's add a new, sort of, fake page - `/questions/new`. When we go there, I want a new question to be added to the database.

Open up `src/Controller/QuestionController.php`, which already holds the homepage and show page. At the bottom, add `public function` and... let's call it `new()`. Above, say `@Route()` with `/questions/new`.

```
64 lines | src/Controller/QuestionController.php
... lines 1 - 10
11 class QuestionController extends AbstractController
12 {
... lines 13 - 55
56 /**
57  * @Route("/questions/new")
58  */
59 public function new()
60 {
... line 61
62 }
63 }
```

To keep things simple, return a `new Response()` - the one from `HttpFoundation` - with
Time for some Doctrine magic!

```

64 lines | src/Controller/QuestionController.php
... lines 1 - 7
8 use Symfony\Component\HttpFoundation\Response;
... lines 9 - 10
11 class QuestionController extends AbstractController
12 {
... lines 13 - 55
56 /**
57  * @Route("/questions/new")
58  */
59 public function new()
60 {
61     return new Response('Time for some Doctrine magic!');
62 }
63 }

```

There's no Doctrine logic yet, but this *should* work. At the browser, hit enter and... woh! It *doesn't* work! There's no error, but this is *not* the page we expected. It looks like the question *show* page. And, in fact, if you look down on the web debug toolbar... yea! The route is `app_question_show` !

The problem is that the url `/questions/new` *does* match this route! It look like "new" is the `slug` . Routes match from top to bottom and Symfony stops as soon as it finds the *first* matching route. So the easiest fix is to just move the *more* specific route above this one.

```

64 lines | src/Controller/QuestionController.php
... lines 1 - 10
11 class QuestionController extends AbstractController
12 {
... lines 13 - 25
26 public function homepage()
27 {
... line 28
29 }
... line 30
31 /**
32  * @Route("/questions/new")
33  */
34 public function new()
35 {
36     return new Response('Time for some Doctrine magic!');
37 }
38
39 /**
40  * @Route("/questions/{slug}", name="app_question_show")
41  */
42 public function show($slug, MarkdownHelper $markdownHelper)
43 {
... lines 44 - 61
62 }
63 }

```

This doesn't happen too often, but this is how I handle it.

Now when we go refresh... got it!

[Creating the Question Object](#)

Ok: time to work! Eventually - in a future tutorial - this page will render a form where the user can fill out all the information about their question. When they submit, we will save that question to the database.

But we're not going to talk about Symfony forms yet. Instead, let's "fake it" inside the controller. Let's create a `Question` object, set some hardcoded data on it and ask Doctrine to save it.

And because there is *nothing* special about our entity class, instantiating it looks *exactly* like you would expect: `$question = new Question()` and I'll auto-complete this so that PhpStorm adds the `Question` use statement.

```
87 lines | src/Controller/QuestionController.php
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 11
12 class QuestionController extends AbstractController
13 {
... lines 14 - 34
35 public function new()
36 {
37     $question = new Question();
... lines 38 - 59
60 }
... lines 61 - 85
86 }
```

Next, call `$question->setName('Missing pants')` - an unfortunate magical side effect of an incorrect spell.

```
87 lines | src/Controller/QuestionController.php
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 11
12 class QuestionController extends AbstractController
13 {
... lines 14 - 34
35 public function new()
36 {
37     $question = new Question();
38     $question->setName('Missing pants')
... lines 39 - 59
60 }
... lines 61 - 85
86 }
```

And `->setSlug('missing-pants')` with a random number at the end so that each one is unique.

```
87 lines | src/Controller/QuestionController.php
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 11
12 class QuestionController extends AbstractController
13 {
... lines 14 - 34
35 public function new()
36 {
37     $question = new Question();
38     $question->setName('Missing pants')
39     ->setSlug('missing-pants-'.rand(0, 1000))
... lines 40 - 59
60 }
... lines 61 - 85
86 }
```

For the *main* part of the question, call `->setQuestion()` and, because this is long, I'll use the multiline syntax - `<<<EOF` - and paste in some content. You can copy this from the code block on this page or use any text.

87 lines | src/Controller/QuestionController.php

```
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 11
12 class QuestionController extends AbstractController
13 {
... lines 14 - 34
35 public function new()
36 {
37     $question = new Question();
38     $question->setName('Missing pants')
39     ->setSlug('missing-pants-' . rand(0, 1000))
40     ->setQuestion(<<<EOF
41 Hi! So... I'm having a *weird* day. Yesterday, I cast a spell
42 to make my dishes wash themselves. But while I was casting it,
43 I slipped a little and I think `I also hit my pants with the spell`.
44
45 When I woke up this morning, I caught a quick glimpse of my pants
46 opening the front door and walking out! I've been out all afternoon
47 (with no pants mind you) searching for them.
48
49 Does anyone have a spell to call your pants back?
50 EOF
51     );
... lines 52 - 59
60 }
... lines 61 - 85
86 }
```

The *last* field is `$askedAt`. Let's add some randomness to this: if a random number between 1 and 10 is greater than 2, then call `$question->setAskedAt()`. Remember: `askedAt` is allowed to be `null` in the database... and if it *is*, we want that to *mean* that the user hasn't *published* the question yet. This if statement will give us a nice mixture of published and unpublished questions.

87 lines | src/Controller/QuestionController.php

```
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 11
12 class QuestionController extends AbstractController
13 {
... lines 14 - 34
35 public function new()
36 {
37     $question = new Question();
38     $question->setName('Missing pants')
39     ->setSlug('missing-pants-' . rand(0, 1000))
40     ->setQuestion(<<<EOF
... lines 41 - 49
50 EOF
51     );
52
53     if (rand(1, 10) > 2) {
... line 54
55     }
... lines 56 - 59
60 }
... lines 61 - 85
86 }
```

Also remember that the `$askedAt` property is a `datetime` field. This means that it will be a `DATETIME` type in MySQL: a field that is ultimately set via a date *string*. But in PHP, instead of dealing with *strings*, *thankfully* we get to deal with `DateTime` *objects*. Let's say `new \DateTime()` and add some randomness here too:

`sprintf('-%d days')` and pass a random number from 1 to 100.

```
87 lines | src/Controller/QuestionController.php
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 11
12 class QuestionController extends AbstractController
13 {
... lines 14 - 34
35 public function new()
36 {
37     $question = new Question();
38     $question->setName('Missing pants')
39     ->setSlug('missing-pants-' . rand(0, 1000))
40     ->setQuestion(<<<EOF
... lines 41 - 49
50 EOF
51     );
52
53     if (rand(1, 10) > 2) {
54         $question->setAskedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
55     }
... lines 56 - 59
60 }
... lines 61 - 85
86 }
```

So, the `askedAt` will be anywhere from 1 to 100 days ago.

Ok! Our `Question` object is done! Add a `dd($question)` at the bottom:

```
87 lines | src/Controller/QuestionController.php
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 11
12 class QuestionController extends AbstractController
13 {
... lines 14 - 34
35 public function new()
36 {
37     $question = new Question();
38     $question->setName('Missing pants')
39     ->setSlug('missing-pants-' . rand(0, 1000))
40     ->setQuestion(<<<EOF
... lines 41 - 49
50 EOF
51     );
52
53     if (rand(1, 10) > 2) {
54         $question->setAskedAt(new \DateTime(sprintf('-%d days', rand(1, 100))));
55     }
56
57     dd($question);
... lines 58 - 59
60 }
... lines 61 - 85
86 }
```

then move over, refresh and... hello nice, boring `Question` object! Notice that the `id` property is still `null` because we haven't saved it to the database yet.

[The EntityManagerInterface Service](#)

So... how *do* we ask Doctrine to save this? When we installed Doctrine, one of the packages we downloaded

was DoctrineBundle. From the Symfony Fundamentals course, you might remember that the *main* thing that a bundle gives us is new *services* in the container. And even though Doctrine is *super* powerful, it turns out that there is just *one* Doctrine service that we'll use 99% of the time. This *one* service is capable of both saving and fetching... which... is really all Doctrine does.

To find the service, head to your terminal and run:

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

This returns several services, but most are lower level. The one we want - which is the *most* important service *by far* in Doctrine - is `EntityManagerInterface`.

Let's go use it! Back in the controller, add a new argument to autowire this:

`EntityManagerInterface $entityManager`.

```
93 lines | src/Controller/QuestionController.php
... lines 1 - 6
7 use Doctrine\ORM\EntityManagerInterface;
... lines 8 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 35
36 public function new(EntityManagerInterface $entityManager)
37 {
... lines 38 - 65
66 }
... lines 67 - 91
92 }
```

[persist\(\) and flush\(\)](#)

Below, remove the `dd()`. How do we save? Call `$entityManager->persist()` and pass the object to save. And then `$entityManager->flush()`.

```
93 lines | src/Controller/QuestionController.php
... lines 1 - 6
7 use Doctrine\ORM\EntityManagerInterface;
... lines 8 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 35
36 public function new(EntityManagerInterface $entityManager)
37 {
... lines 38 - 57
58     $entityManager->persist($question);
59     $entityManager->flush();
... lines 60 - 65
66 }
... lines 67 - 91
92 }
```

Yes, you need *both* lines. The `persist()` call *simply* says:

Hey Doctrine! Please be "aware" of this `Question` object.

The `persist` line does *not* make *any* queries. The `INSERT` query happens when we call `flush()`. The `flush()` method says:

Yo Doctrine! Please look at all of the objects that you are "aware" of and make all the queries you need to

save those.

So *this* is how saving looks: a `persist()` and `flush()` right next to each other. If you ever needed to, you could call `persist()` on 5 different objects and *then* call `flush()` once at the end to make *all* of those queries at the same time.

Anyways, now that we have a `Question` object, let's make the `Response` more interesting. I'll say `sprintf` with:

Well hallo! The shiny new question is id `##d`, slug: `%s`

Passing `$question->getId()` for the first placeholder and `$question->getSlug()` for the second.

```
93 lines | src/Controller/QuestionController.php
... lines 1 - 6
7 use Doctrine\ORM\EntityManagerInterface;
... lines 8 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 35
36 public function new(EntityManagerInterface $entityManager)
37 {
... lines 38 - 57
58     $entityManager->persist($question);
59     $entityManager->flush();
60
61     return new Response(sprintf(
62         'Well hallo! The shiny new question is id ##d, slug: %s',
63         $question->getId(),
64         $question->getSlug()
65     ));
66 }
... lines 67 - 91
92 }
```

Ok, back at the browser, *before* saving, the `Question` object had *no* `id` value. But now when we refresh... yes! It has an `id`! After saving, Doctrine automatically sets the new `id` on the object. We can refresh over and over again to add more and more question rows to the table.

Let's go see them! If you ever want to make a query to see something, Doctrine has a handy `bin/console` command for that:

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

And... yes! Here is a dump of the 8 rows in the table.

Next: we know how to save. So how can we query to *fetch* data?

Chapter 9: Fetching Data & The Repository

Our `question` table has data! And each time we refresh, we got more data! You get a question! You get a question!

Copy the slug from the latest one and then go to `/questions/that-slug` to see it. Except... this is not *actually* that question. The name is kinda right... but that's it. Over in the `show()` action, this is because *nothing* is being loaded from the database. Lame!

Here's our next mission: use the `$slug` to query for a row of `Question` data and *use* that to make this page *truly* dynamic. How? The entity manager that we use to *save* data can *also* be used to *fetch* data.

The Repository

Start by adding a third argument: `EntityManagerInterface $entityManager`. This interface has a *bunch* of methods on it. But... most of the time, you'll only use three: `persist()` and `flush()` to save, and `getRepository()` when you want to *get* data.

```
97 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 70
71 public function show($slug, MarkdownHelper $markdownHelper, EntityManagerInterface $entityManager)
72 {
... lines 73 - 94
95 }
96 }
```

Say `$repository = $entityManager->getRepository()` and pass the entity *class* that we want to query. So `Question::class`.

```
97 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 70
71 public function show($slug, MarkdownHelper $markdownHelper, EntityManagerInterface $entityManager)
72 {
73     if ($this->isDebug) {
74         $this->logger->info('We are in debug mode!');
75     }
76
77     $repository = $entityManager->getRepository(Question::class);
... lines 78 - 94
95 }
96 }
```

Whenever you need to get data, you'll *first* get the *repository* for an entity. This repository object is really really good at querying from the `question` table. And it has several methods to help us.

For example, we want to query **WHERE** the `slug` column equals the `$slug` variable. Do that with `$question = $repository->` and... this auto completes a bunch of methods. We want `findOneBy()`. Pass this an array of the **WHERE** statements we need: `'slug' => $slug`. After, `dd($question)`.

```

97 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 70
71 public function show($slug, MarkdownHelper $markdownHelper, EntityManagerInterface $entityManager)
72 {
... lines 73 - 76
77     $repository = $entityManager->getRepository(Question::class);
78     $question = $repository->findOneBy(['slug' => $slug]);
79     dd($question);
... lines 80 - 94
95 }
96 }

```

Ok, let's see what this returns! Refresh and... woohoo! This gives us a **Question** *object*. Doctrine finds the matching row of data and uses that to populate an object, which is beautiful.

The repository has a number of other methods on it. For example, **findOneBy()** returns a single object and **findBy()** returns an *array* of objects that match whatever criteria you pass. The **findAll()** method returns an array of *all* **Question** objects and there are a few others. So without doing *any* work, we can easily execute the most basic queries. Now, eventually we *will* need to do more *complex* stuff - and for that, we'll write custom queries. We'll see that later.

404 On Not Found

So when Doctrine finds a matching row, we get back a **Question** object. But if we change the slug in the URL to something that does *not* exist, we get *null*. So: a **Question** object or null.

Let's think: what *do* we want to do when someone goes to a URL that doesn't match a real question? The answer is: trigger a 404 page! Great! Um... how do we trigger a 404 page in Symfony?

First, this is optional - I'm going to say `/**` space and then type **Question|null** .

```

101 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 70
71 public function show($slug, MarkdownHelper $markdownHelper, EntityManagerInterface $entityManager)
72 {
... lines 73 - 77
78     /** @var Question|null $question */
79     $question = $repository->findOneBy(['slug' => $slug]);
... lines 80 - 98
99 }
100 }

```

This simply helps my editor know that this is a **Question** object or null, which will assist auto-completion. And, to be honest, PhpStorm is so smart that... I think it already knew this.

Below, if *not* **\$question** , trigger a 404 page by saying `throw $this->createNotFoundException()` , which is a method on the parent **AbstractController** class. Pass this any message you want:

```
No question found for slug %s
```

And pass the **\$slug** variable.

```

101 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 70
71     public function show($slug, MarkdownHelper $markdownHelper, EntityManagerInterface $entityManager)
72     {
... lines 73 - 77
78         /** @var Question|null $question */
79         $question = $repository->findOneBy(['slug' => $slug]);
80         if (!$question) {
81             throw $this->createNotFoundException(sprintf('no question found for slug "%s"', $slug));
82         }
... lines 83 - 98
99     }
100 }

```

That's it! But notice the `throw . createNotFoundException()` instantiates an exception object - a very *special* exception object that triggers a 404 page. Most of the time in Symfony, if you throw an exception, it will cause a 500 page. But this special exception maps to a 404.

Let's try it: refresh and... yes! You can see it up here: "404 Not found" with our message.

Two things about this. First: this is the *development* error page. If we changed the environment to `prod`, we would see a much more boring 404 page with *no* error or stack trace details. We won't talk about it, but the Symfony docs have details about how you can customize the look and feel of your error pages on production.

The second thing I want to say is that the message - no question found for slug - is something that only *developers* will see. Feel free to make this as *descriptive* as you want: you don't need to worry about a real user seeing it.

Now that we have a `Question` object in our controller, let's use it in our template to render *real*, dynamic info. That's next.

Chapter 10: Entity objects in Twig

We *now* have a `Question` object inside our controller. And at the bottom, we render a template. What we need to do is *pass* that `Question` object *into* the template and use it on the page to print the name and other info.

Remove the `dd()`, leave the `$answers` - we'll keep those hardcoded for now because we don't have an `Answer` entity yet - and get rid of the hardcoded `$question`, and `$questionText`.

```
96 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 70
71 public function show($slug, MarkdownHelper $markdownHelper, EntityManagerInterface $entityManager)
72 {
... lines 73 - 79
80     if (!$question) {
81         throw $this->createNotFoundException(sprintf('no question found for slug "%s"', $slug));
82     }
83
84     $answers = [
85         'Make sure your cat is sitting `purrrfectly` still ?',
86         'Honestly, I like furry shoes better than MY cat',
87         'Maybe... try saying the spell backwards?',
88     ];
... lines 89 - 93
94 }
95 }
```

Instead pass a `question` variable to Twig set to the `Question` object.

```
96 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 70
71 public function show($slug, MarkdownHelper $markdownHelper, EntityManagerInterface $entityManager)
72 {
... lines 73 - 79
80     if (!$question) {
81         throw $this->createNotFoundException(sprintf('no question found for slug "%s"', $slug));
82     }
83
84     $answers = [
85         'Make sure your cat is sitting `purrrfectly` still ?',
86         'Honestly, I like furry shoes better than MY cat',
87         'Maybe... try saying the spell backwards?',
88     ];
89
90     return $this->render('question/show.html.twig', [
91         'question' => $question,
92         'answers' => $answers,
93     ]);
94 }
95 }
```

[Twig's Smart . Syntax](#)

Let's go find the template: `templates/question/show.html.twig` . The `question` variable is *no longer* a string: it's now an *object*. So... how do we render an object? Because the `Question` class has a `name` property, we can say `question.name` . It even auto-completes it for me! That doesn't always work in Twig, but it's nice when it does.

```
71 lines | templates/question/show.html.twig
... lines 1 - 2
3  {% block title %}Question: {{ question.name }}{% endblock %}
4
5  {% block body %}
6    <div class="container">
7      <div class="row">
8        <div class="col-12">
9          <h2 class="my-4">Question:</h2>
10         <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11           <div class="q-container-show p-4">
12             <div class="row">
... lines 13 - 27
28               <div class="col">
29                 <h1 class="q-title-show">{{ question.name }}</h1>
30                 <div class="q-display p-3">
31                   <i class="fa fa-quote-left mr-3"></i>
32                   <p class="d-inline">{{ question.question }}</p>
33                   <p class="pt-4"><strong>--Tisha</strong></p>
34                 </div>
35               </div>
36             </div>
37           </div>
38         </div>
39       </div>
40     </div>
... lines 41 - 68
69   </div>
70 {% endblock %}
```

Below... here's another one - `question.name` and `questionText` is now `question.question` .

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

```
... lines 1 - 2
3  {% block title %}Question: {{ question.name }}{% endblock %}
4
5  {% block body %}
6    <div class="container">
7      <div class="row">
8        <div class="col-12">
9          <h2 class="my-4">Question:</h2>
10         <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11           <div class="q-container-show p-4">
12             <div class="row">
... lines 13 - 27
28               <div class="col">
29                 <h1 class="q-title-show">{{ question.name }}</h1>
30                 <div class="q-display p-3">
31                   <i class="fa fa-quote-left mr-3"></i>
32                   <p class="d-inline">{{ question.question }}</p>
33                   <p class="pt-4"><strong>--Tisha</strong></p>
34                 </div>
35               </div>
36             </div>
37           </div>
38         </div>
39       </div>
40     </div>
... lines 41 - 68
69   </div>
70 {% endblock %}
```

I think that's it! Testing time! Move over, go back to the *real* question slug and... there it is! We have a real name and real question text. This date is still hard coded, but we'll fix that soon.

Now, some of you *might* be thinking:

Um... how the heck did that work?

We said `question.name` ... which makes it *look* like it's reading the *name* property. But... if you look at the *name* property inside of the *Question* entity... it's private! That means we *can't* access the *name* property directly. What's going on?

93 lines | src/Entity/Question.php

```
... lines 1 - 10
11 class Question
12 {
... lines 13 - 22
23   private $name;
... lines 24 - 91
92 }
```

We're witnessing some Twig magic. In reality, when we say `question.name`, Twig first *does* look to see if the *name* property exists and is public. If it *were* public, Twig would use it. But since it's not, Twig *then* tries to call a `getName()` method. Yep, we write `question.name`, but, behind the scenes, Twig is smart enough to call `getName()`.

93 lines | src/Entity/Question.php

```
... lines 1 - 10
11 class Question
12 {
... lines 13 - 22
23     private $name;
... lines 24 - 44
45     public function getName(): ?string
46     {
47         return $this->name;
48     }
... lines 49 - 91
92 }
```

I *love* this: it means you can run around saying `question.name` in your template and not really worry about whether there's a getter method or not. It's especially friendly to non-PHP frontend devs.

If you wanted to actually *call* a method - like `getName()` - that *is* allowed, but it's usually not necessary.

The one thing that we *did* lose is that, originally, the question text was being parsed through markdown. We can fix that really easily by using the `parse_markdown` filter that we created in the last tutorial.

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

```
... lines 1 - 4
5 {% block body %}
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                 <div class="q-container-show p-4">
12                     <div class="row">
... lines 13 - 27
28                         <div class="col">
29                             <h1 class="q-title-show">{{ question.name }}</h1>
30                             <div class="q-display p-3">
31                                 <i class="fa fa-quote-left mr-3"></i>
32                                 <p class="d-inline">{{ question.question|parse_markdown }}</p>
33                                 <p class="pt-4"><strong>--Tisha</strong></p>
34                             </div>
35                         </div>
36                     </div>
37                 </div>
38             </div>
39         </div>
40     </div>
... lines 41 - 68
69 </div>
70 {% endblock %}
```

Refresh and... it works.

[The Doctrine Web Debug Toolbar](#)

You may not have noticed, but near the middle of the web debug toolbar, there's a little database icon that says 1 database query. And we can click the icon to jump into the profiler and... see the *exact* query! If this page made multiple queries, you would see *all* of them here.

If you ever want to debug a query directly, click "View runnable query" to get a version that you can copy.

[Seeing the Profiler for AJAX Requests](#)

Now, here's a challenge: how could we see the `INSERT` query that's made when we go to `/questions/new`? This

did just make that query... but because we're not rendering HTML, this doesn't have a web debug toolbar. The same problem happens whenever you make an AJAX call.

So... are we out of luck? Nah - we can use a trick. Go to [/_profiler](#) to find a list of the most recent requests we've made. Here's the one we *just* made to [/questions/new](#). Click the little token string on the right to jump into the *full* profiler for that request! Go to the "Doctrine" tab and... bam! Cool! It even wraps the INSERT in a transaction.

Remember this trick the next time you want to see database queries, a rendered version of an error, or something else for an AJAX request.

Go back a few times to the question show page. The last piece of question data that's hardcoded is this "asked 10 minutes ago" text. Search for it in the template... there it is, line 18.

Let's make this dynamic... but, not just by printing some boring date like "July 10th at 10:30 EST". Yuck. Let's print a much-friendlier "10 minutes ago" type of message next.

Chapter 11: "5 Minutes Ago" Strings

Let's make this date dynamic! The field on `Question` that we're going to use is `askedAt`, which - remember - *might* be `null`. If a `Question` hasn't been published yet, then it won't have an `askedAt`.

Let's plan for this. In the template, add `{% if question.askedAt %}` with an `{% else %}` and `{% endif %}`

```
75 lines | templates/question/show.html.twig
... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
14              
15              <div class="mt-3">
16                <small>
17                  {% if question.askedAt %}
... lines 18 - 19
20                    {% else %}
... line 21
22                      {% endif %}
23                </small>
... lines 24 - 29
30            </div>
31          </div>
... lines 32 - 39
40        </div>
41      </div>
42    </div>
43  </div>
44 </div>
... lines 45 - 72
73 </div>
... lines 74 - 75
```

If the question is *not* published, say `(unpublished)`.

```

... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
14              
15              <div class="mt-3">
16                <small>
17                  {% if question.askedAt %}
... lines 18 - 19
20                    {% else %}
21                      (unpublished)
22                    {% endif %}
23                </small>
... lines 24 - 29
30            </div>
31          </div>
... lines 32 - 39
40        </div>
41      </div>
42    </div>
43  </div>
44 </div>
... lines 45 - 72
73 </div>
... lines 74 - 75

```

In a real app, we would probably *not* allow users to see *unpublished* questions... we could do that in our controller by checking for this field and saying `throw $this->createNotFoundException()` if it's null. But... maybe a user will be able to *preview* their *own* unpublished questions. If they did, we'll show *unpublished*.

[The Twig date Filter](#)

The easiest way to *try* to print the date would be to say `{{ question.askedAt }}`.

```

... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
14              
15              <div class="mt-3">
16                <small>
17                  {% if question.askedAt %}
... line 18
19                    {{ question.askedAt }}
20                    {% else %}
21                      (unpublished)
22                    {% endif %}
23                </small>
... lines 24 - 29
30            </div>
31          </div>
... lines 32 - 39
40        </div>
41      </div>
42    </div>
43  </div>
44 </div>
... lines 45 - 72
73 </div>
... lines 74 - 75

```

But... you might be shouting: "Hey Ryan! That's not going to work!".

And... you're right:

Object of class `DateTime` could not be converted to string

We know that when we have a `datetime` type in Doctrine, it's stored in PHP as a `DateTime` object. That's nice because `DateTime` objects are easy to work with... but we can't simply print them.

To fix this, pass the `DateTime` object through a `|date()` filter. This takes a format argument - something like `Y-m-d H:i:s`.


```

... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
14              
15              <div class="mt-3">
16                <small>
17                  {% if question.askedAt %}
... line 18
19                    {{ question.askedAt|date('Y-m-d H:i:s') }}
20                    {% else %}
21                      (unpublished)
22                    {% endif %}
23                  </small>
... lines 24 - 29
30                </div>
31              </div>
... lines 32 - 39
40            </div>
41          </div>
42        </div>
43      </div>
44    </div>
... lines 45 - 72
73 </div>
... lines 74 - 75

```

When we try the page now... it's technically *correct*... but yikes! This... well... how can I put this politely: it looks like a backend developer designed this.

[KnpTimeBundle](#)

Whenever I render dates, I like to make them relative. Instead of printing an exact date, I prefer something like "10 minutes ago". It also avoids timezone problems... because 10 minutes ago makes sense to everyone! But this exact date would *really* need a timezone to make sense.

So let's do this. Start by adding the word "Asked" back before the date. Cool.

```
75 lines | templates/question/show.html.twig
... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
14              
15              <div class="mt-3">
16                <small>
17                  {% if question.askedAt %}
18                    Asked <br>
19                    {{ question.askedAt|date('Y-m-d H:i:s') }}
20                  {% else %}
21                    (unpublished)
22                  {% endif %}
23                </small>
24              </div>
25            </div>
26          </div>
27        </div>
28      </div>
29    </div>
30  </div>
31 </div>
... lines 32 - 39
40 </div>
41 </div>
42 </div>
43 </div>
44 </div>
... lines 45 - 72
73 </div>
... lines 74 - 75
```

To convert the `DateTime` into a friendly string, we can install a nice bundle. At your terminal, run:

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

You could find this bundle if you googled for "Symfony ago". As we know, the *main* thing that a bundle gives us is more *services*. In this case, the bundle gives us one main service that provides a Twig filter called `ago`.

It's pretty awesome. Back in the template, add `|ago`.

```

... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
14              
15              <div class="mt-3">
16                <small>
17                  {% if question.askedAt %}
18                    Asked <br>
19                    {{ question.askedAt|ago }}
20                  {% else %}
21                    (unpublished)
22                  {% endif %}
23                </small>
24              </div>
25            </div>
26          </div>
27        </div>
28      </div>
29    </div>
30  </div>
31 </div>
... lines 32 - 39
40 </div>
41 </div>
42 </div>
43 </div>
44 </div>
... lines 45 - 72
73 </div>
... lines 74 - 75

```

We're done! When we refresh now... woohoo!

Asked 1 month ago

Next: let's make the homepage dynamic by querying for *all* of the questions in the database and rendering them. Along the way, we're going to learn a *secret* about the repository object.

Chapter 12: Custom Repository Class

Now that the show page is working, let's bring the homepage to life! This time, instead of querying for one **Question** object, we want to query for *all* of them.

[findAll\(\) for All Data](#)

Head over to **QuestionController** and scroll up to **homepage()**. Ok, to fetch data, we need to autowire the entity manager with **EntityManagerInterface \$entityManager**.

```
100 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 27
28 public function homepage(EntityManagerInterface $entityManager)
29 {
... lines 30 - 34
35 }
... lines 36 - 98
99 }
```

Now add **\$repository = \$entityManager->getRepository(Question::class)**.

```
100 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 27
28 public function homepage(EntityManagerInterface $entityManager)
29 {
30     $repository = $entityManager->getRepository(Question::class);
... lines 31 - 34
35 }
... lines 36 - 98
99 }
```

And finally, **\$questions = \$repository->findAll()**. Let's **dd(\$questions)** to see what these look like.

```
100 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 27
28 public function homepage(EntityManagerInterface $entityManager)
29 {
30     $repository = $entityManager->getRepository(Question::class);
31     $questions = $repository->findAll();
32     dd($questions);
... lines 33 - 34
35 }
... lines 36 - 98
99 }
```

[Rendering all the Questions](#)

Ok, refresh the homepage. There we go! 12 **Question** objects for the 12 rows in my table. *Now* we're dangerous because we can pass these into our template. Add a second argument to **render()** - an array - to pass a

`questions` variable set to our array of `Question` objects.

```
101 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 27
28 public function homepage(EntityManagerInterface $entityManager)
29 {
30     $repository = $entityManager->getRepository(Question::class);
31     $questions = $repository->findAll();
32
33     return $this->render('question/homepage.html.twig', [
34         'questions' => $questions,
35     ]);
36 }
... lines 37 - 99
100 }
```

Pop open the template: `templates/question/homepage.html.twig`. Let's see: the homepage currently has two hard coded questions. I want to loop right inside the `row`: `{% for question in questions %}`. Trace the markup down to see where this ends and... add `{% endfor %}`. Delete the 2nd hard-coded question completely.

```
50 lines | templates/question/homepage.html.twig
... lines 1 - 9
10 <div class="container">
... lines 11 - 15
16 <div class="row">
17     {% for question in questions %}
18     <div class="col-12 mb-3">
... lines 19 - 43
44     </div>
45     {% endfor %}
46 </div>
47 </div>
... lines 48 - 50
```

Perfect. *Now* it's just like the show page because we have a `question` variable. The first thing to update is the question name - `{{ question.name }}` and the slug also needs to be dynamic: `question.slug`.

```

50 lines | templates/question/homepage.html.twig
... lines 1 - 9
10 <div class="container">
... lines 11 - 15
16 <div class="row">
17     {% for question in questions %}
18     <div class="col-12 mb-3">
19         <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
20             <div class="q-container p-4">
21                 <div class="row">
... lines 22 - 27
28                     <div class="col">
29                         <a class="q-title" href="{{ path('app_question_show', { slug: question.slug }) }}"><h2>{{ question.na
... lines 30 - 34
35                     </div>
36                 </div>
37             </div>
... lines 38 - 42
43         </div>
44     </div>
45     {% endfor %}
46 </div>
47 </div>
... lines 48 - 50

```

Below, for the question text, use `{{ question.question|parse_markdown }}`. We might also want to only show *some* of the question on the page - we could do that by adding a new method - like `getQuestionPreview()` to the entity - and using it here. We'll see this idea of custom entity methods later.

```

50 lines | templates/question/homepage.html.twig
... lines 1 - 9
10 <div class="container">
... lines 11 - 15
16 <div class="row">
17     {% for question in questions %}
18     <div class="col-12 mb-3">
19         <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
20             <div class="q-container p-4">
21                 <div class="row">
... lines 22 - 27
28                     <div class="col">
29                         <a class="q-title" href="{{ path('app_question_show', { slug: question.slug }) }}"><h2>{{ question.na
30                         <div class="q-display p-3">
31                             <i class="fa fa-quote-left mr-3"></i>
32                             <p class="d-inline">{{ question.question|parse_markdown }}</p>
33                             <p class="pt-4"><strong>--Tisha</strong></p>
34                         </div>
35                     </div>
36                 </div>
37             </div>
... lines 38 - 42
43         </div>
44     </div>
45     {% endfor %}
46 </div>
47 </div>
... lines 48 - 50

```

At the bottom, there's one more link: `question.slug`.

```

50 lines | templates/question/homepage.html.twig
... lines 1 - 9
10 <div class="container">
... lines 11 - 15
16 <div class="row">
17     {% for question in questions %}
18     <div class="col-12 mb-3">
19         <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
20             <div class="q-container p-4">
21                 <div class="row">
... lines 22 - 27
28                 <div class="col">
29                     <a class="q-title" href="{{ path('app_question_show', { slug: question.slug }) }}"><h2>{{ question.na
30                     <div class="q-display p-3">
31                         <i class="fa fa-quote-left mr-3"></i>
32                         <p class="d-inline">{{ question.question|parse_markdown }}</p>
33                         <p class="pt-4"><strong>--Tisha</strong></p>
34                     </div>
35                 </div>
36             </div>
37         </div>
38         <a class="answer-link" href="{{ path('app_question_show', { slug: question.slug }) }}" style="color: #fff;">
... lines 39 - 41
42     </a>
43 </div>
44 </div>
45 {% endfor %}
46 </div>
47 </div>
... lines 48 - 50

```

Done! Doctrine makes it easy to query for data and Twig makes it easy to render. Go team! At the browser, refresh and... *cool!*

Ordering the Data

Each question has a random **askedAt** date - you can see it by clicking into each one. What we *probably* want to do is put the *newest* questions on top. In other words, we want to do the *same* query but with **ORDER BY askedAt DESC**.

If you click the database icon on the web debug toolbar, you can see that the query doesn't have an **ORDER BY** yet. When you're working with the built-in methods on the repository class, you're a bit limited - there are many custom things that these methods simply *can't* do. For example, **findAll()** doesn't have *any* arguments: there's no way to customize the order or anything else. Soon we'll learn how to write *custom* queries so we can do whatever we want.

But, in this case, there *is* another method that can help: **findBy()**. Pass this an empty array - we don't need *any* WHERE statements - and then another array with **'askedAt' => 'DESC'**.

```

101 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 27
28     public function homepage(EntityManagerInterface $entityManager)
29     {
... line 30
31         $questions = $repository->findBy([], ['askedAt' => 'DESC']);
... lines 32 - 35
36     }
... lines 37 - 99
100 }

```

Let's try it! Refresh! And... click the first: 10 days ago. Click the second: 1 month ago! I think we got it! If we jump into the profiler... yes! It has `ORDER BY asked_at DESC`.

We've now pushed the built-in repository methods *about* as far as they can go.

EntityRepository

Question time: when we call `getRepository()`, what does that *actually* return? It's an object of course, but what *type* of object? The answer is: `EntityRepository`.

In PhpStorm, I'll press Shift+Shift and type `EntityRepository.php`. I want to see what this looks like. Make sure to include all "non project items". Here it is!

`EntityRepository` lives deep down inside of Doctrine and *it* is where the methods we've been using live, like `find()`, `findAll()`, `findBy()`, `findOneBy()` and some more.

Our Custom Repository Class

But check this out: in the controller, `dd($repository)`.

```
102 lines | src/Controller/QuestionController.php
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 27
28 public function homepage(EntityManagerInterface $entityManager)
29 {
30     $repository = $entityManager->getRepository(Question::class);
31     dd($repository);
... lines 32 - 36
37 }
... lines 38 - 100
101 }
```

When we refresh... surprise! I lied! Sort of...

Instead of being an instance of `EntityRepository` - like I *promised* - this is an instance of `App\Repository\QuestionRepository`. Hey! That's a class that lives in our project! Open it up: `src/Repository/QuestionRepository.php`.

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


When we originally ran `make:entity` to generate `Question`, it actually generated *two* classes: `Question` and `QuestionRepository`. This class extends another called `ServiceEntityRepository`. And if you hold Command or Ctrl and click into it, *that* class extends `EntityRepository`! The class we were just looking at.

When we ask for the repository for the `Question` entity, Doctrine *actually* returns a `QuestionRepository` object. But since that ultimately extends `EntityRepository`, we have access to all the helper methods like `findAll()` and `findBy()`.

But... how does Doctrine know to give us an instance of this class? How does it connect the `Question` entity to the `QuestionRepository` class? Is it relying on a naming convention?

Nope! The answer lives at the top of the `Question` class: we have `@ORM\Entity()` with `repositoryClass=QuestionRepository::class`. This was generated for us by `make:entity`.

```
93 lines | src/Entity/Question.php
... lines 1 - 7
8  /**
9   * @ORM\Entity(repositoryClass=QuestionRepository::class)
10  */
11  class Question
12  {
... lines 13 - 91
92 }
```

Here's the big picture: when we call `getRepository()` and pass it `Question::class`, Doctrine will give us an instance of `QuestionRepository`. And because that extends `EntityRepository`, we get access to the shortcut methods!

[Custom Repository Methods](#)

The reason this is *cool* is that anytime we need to write a custom query for the `Question` entity, we can add a new *method* inside of `QuestionRepository`.

The class already has an example: uncomment the `findByExampleField()` method. If I have a `findByExampleField()` method in the repository, it means that we can *call* this from the controller.

```
49 lines | src/Repository/QuestionRepository.php
... lines 1 - 14
15 class QuestionRepository extends ServiceEntityRepository
16 {
... lines 17 - 21
22  /**
23   * @return Question[] Returns an array of Question objects
24   */
25  public function findByExampleField($value)
26  {
27      return $this->createQueryBuilder('q')
28         ->andWhere('q.exampleField = :val')
29         ->setParameter('val', $value)
30         ->orderBy('q.id', 'ASC')
31         ->setMaxResults(10)
32         ->getQuery()
33         ->getResult()
34      ;
35  }
... lines 36 - 47
48 }
```

In a few minutes, we're going to write a custom query that finds all questions *WHERE askedAt IS NOT NULL*. In `QuestionRepository`, let's create a method to hold this. How about: `findAllAskedOrderedByNewest()` and this won't need any arguments.

49 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 14
15 class QuestionRepository extends ServiceEntityRepository
16 {
... lines 17 - 24
25     public function findAllAskedOrderedByNewest()
26     {
... lines 27 - 34
35     }
... lines 36 - 47
48 }
```

In the controller, remove the `dd()` and say `$questions = $repository->findAllAskedOrderedByNewest()` .

101 lines | src/Controller/QuestionController.php

```
... lines 1 - 12
13 class QuestionController extends AbstractController
14 {
... lines 15 - 27
28     public function homepage(EntityManagerInterface $entityManager)
29     {
30         $repository = $entityManager->getRepository(Question::class);
31         $questions = $repository->findAllAskedOrderedByNewest();
... lines 32 - 35
36     }
... lines 37 - 99
100 }
```

Of course, that won't work yet because the logic is all wrong, but it *will* call the new method.

Next, let's learn about DQL and the query builder. Then, we'll create a custom query that will return the *exact* results we want.

Chapter 13: DQL & The Query Builder

We just learned that when you ask for a repository, what you *actually* get back is a custom class. Well, technically you don't *have* to have a custom repository class - and if you don't, Doctrine will just give you an instance of `EntityRepository`. But in practice, I *always* have custom repository classes.

Anyways, when we ask for the repository for the `Question` entity, we get back an instance of this `QuestionRepository`. The *cool* thing is that we can add custom methods to hold custom queries. In fact, *every* time I write a custom query, I'll put it in a repository class.

Here's the new goal: I want to change the query on the homepage so that it *hides* any questions `WHERE askedAt IS NULL`. This will hide "unpublished" questions.

DQL

We know that we use SQL queries to talk to databases. Internally, Doctrine has a slightly different language called DQL: Doctrine Query Language. But don't worry, it's almost *identical* to SQL. The main difference is that, with DQL, you reference class and property names instead of table and column names. Otherwise, it basically looks the same.

The QueryBuilder

Now, you can *absolutely* write DQL strings by hand and execute them. Or you can use a *super* handy object called the `QueryBuilder`, which allows you to *build* that DQL string using a convenient object. *That* is what you see here.

The `$this->createQueryBuilder()` line creates the `QueryBuilder` object. And because we're inside of the `QuestionRepository`, the `QueryBuilder` will *already* know to query `FROM` the `question` table. The `q` is basically the table alias, like `SELECT * FROM question as q`. We'll use that everywhere to refer to properties on `Question`.

Then, most of the methods on `QueryBuilder` are pretty intuitive, like, `andWhere()` and `orderBy()`. `setMaxResults()` is probably one of the *least* intuitive and it's still pretty simple: this adds a `LIMIT`.

Prepared Statements

Check out the `andWhere()`: `q.exampleField = :value`. Doctrine uses prepared statements... which is a fancy way of saying that you should *never* concatenate a dynamic value into a string. *This* allows for SQL injections.

Instead, whenever you have something dynamic, set it to a placeholder - like `:value` and then *set* that placeholder with `setParameter()`. This is how prepared statements work. It's not unique at *all* to Doctrine, but I wanted to point it out.

Writing our Custom Query

Ok: let's clear out these four lines and make our *own* query. Start with `->andWhere('q.askedAt IS NOT NULL')`.

47 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 14
15 class QuestionRepository extends ServiceEntityRepository
16 {
... lines 17 - 24
25 public function findAllAskedOrderedByNewest()
26 {
27     return $this->createQueryBuilder('q')
28         ->andWhere('q.askedAt IS NOT NULL')
... line 29
30     ->getQuery()
31     ->getResult()
32 ;
33 }
... lines 34 - 45
46 }
```

I'm using `askedAt` because that's the name of the *property*... even though the column in the table is `asked_at`. Now add `->orderBy()` with `q.askedAt` and `DESC`.

47 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 14
15 class QuestionRepository extends ServiceEntityRepository
16 {
... lines 17 - 24
25 public function findAllAskedOrderedByNewest()
26 {
27     return $this->createQueryBuilder('q')
28         ->andWhere('q.askedAt IS NOT NULL')
29         ->orderBy('q.askedAt', 'DESC')
30         ->getQuery()
31         ->getResult()
32 ;
33 }
... lines 34 - 45
46 }
```

Oh, and notice that I'm using `andWhere()` ... even though there are no WHERE clauses before this! I'm doing this for 2 reasons. First... because it's allowed! Doctrine is smart enough to figure out if it needs an `AND` statement or not. And second, there *is* a `where()` method... but it's kind of dangerous because it will *override* any `where()` or `andWhere()` calls that you had earlier. So, I *never* use it.

Once we're done building our query, we always finish with `getQuery()` to transform it into a finished `Query` object. Then, the `getResult()` method will return an *array* of `Question` objects. My `@return` already says this! Woo!

The other common *final* method is `getOneOrNullResult()` which I use when I want to find a *single* record.

Ok: with any luck, this will return the array of `Question` objects we need! Let's try it! Find your browser, refresh and... no errors! But I can't exactly tell if it's hiding the right stuff. Let's click on the web debug toolbar to see the query. I think that's right! Click "View formatted query". That's *definitely* right!

[More Complex Queries? SQL?](#)

We're not going to talk too much more about creating custom queries, but we *do* have an entire tutorial about [Doctrine queries](#). It's built on an old version of Symfony, but all of the info about Doctrine queries hasn't changed.

And yes, if you ever have a super duper custom complex query and you *just* want to write it in normal SQL, you can *absolutely* do that. The Doctrine queries tutorial will show you how.

[Autowiring the Repository Directly](#)

Anyways, whenever we need to query for something, we're going to get the repository for that entity and either call a *custom* method that we created or a built-in method. And actually... I've been making us do too much work! There's an *easier* way to get the repository. Instead of autowiring the entity manager and calling `getRepository()`, the `QuestionRepository` *itself* is a service in the container. That means we can autowire it directly!

Check it out: remove the `EntityManagerInterface` argument and replace it with `QuestionRepository $repository`. Celebrate by deleting the `getRepository()` call.

```
101 lines | src/Controller/QuestionController.php
... lines 1 - 5
6 use App\Repository\QuestionRepository;
... lines 7 - 13
14 class QuestionController extends AbstractController
15 {
... lines 16 - 28
29 public function homepage(QuestionRepository $repository)
30 {
31     $questions = $repository->findAllAskedOrderedByNewest();
32
33     return $this->render('question/homepage.html.twig', [
34         'questions' => $questions,
35     ]);
36 }
... lines 37 - 99
100 }
```

If we move over and refresh... it *still* works! In practice, when I need to query for something, this is what I do: I autowire the specific repository I need. The only time that I work with the entity manager directly is when I need to *save* something - like we're doing in the `new()` method.

Thanks to the `QueryBuilder` object, we can leverage a pattern inside our repository that will allow us to *reuse* pieces of query logic for multiple queries. Let me show you how next.

Chapter 14: Reusing Query Logic & Param Converters

Maybe my favorite thing about the `QueryBuilder` is that if you have multiple methods inside a repository, you can *reuse* query logic between them. For example, a lot of queries might need this `andWhere('q.AskedAt IS NOT NULL')` logic. That's *not* complex, but I would still *love* to not repeat this line over and over again in every method and query. Instead, let's centralize this logic.

[Private Method to Mutate a QueryBuilder](#)

Create a new private function at the bottom. Let's call it `addIsAskedQueryBuilder()` with a `QueryBuilder` argument - the one from ORM. Make this also *return* a `QueryBuilder`.

```
54 lines | src/Repository/QuestionRepository.php
... lines 1 - 6
7 use Doctrine\ORM\QueryBuilder;
... lines 8 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 36
37 private function addIsAskedQueryBuilder(QueryBuilder $qb): QueryBuilder
38 {
... line 39
40 }
... lines 41 - 52
53 }
```

Inside, we're going to *modify* the `QueryBuilder` that's passed to us to *add* the custom logic. So, `$qb->` and then copy the `andWhere('q.AskedAt IS NOT NULL')`. Oh, and *return* this.

```
54 lines | src/Repository/QuestionRepository.php
... lines 1 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 36
37 private function addIsAskedQueryBuilder(QueryBuilder $qb): QueryBuilder
38 {
39     return $qb->andWhere('q.AskedAt IS NOT NULL');
40 }
... lines 41 - 52
53 }
```

Pretty much every `QueryBuilder` method returns *itself*, which is nice because it allows us to do method chaining. By returning the `QueryBuilder` from *our* method, we will *also* be able to chain off of it.

Ok, back in the original method, *first* create a `QueryBuilder` and set it to a variable. So, `$qb = $this->createQueryBuilder()`.

54 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 25
26 public function findAllAskedOrderByNewest()
27 {
28     $qb = $this->createQueryBuilder('q');
... lines 29 - 34
35 }
... lines 36 - 52
53 }
```

Then we can say `return $this->addIsAskedQueryBuilder($qb)` and then the rest of the query.

54 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 25
26 public function findAllAskedOrderByNewest()
27 {
28     $qb = $this->createQueryBuilder('q');
29
30     return $this->addIsAskedQueryBuilder($qb)
31         ->orderBy('q.AskedAt', 'DESC')
32         ->getQuery()
33         ->getResult()
34     ;
35 }
... lines 36 - 52
53 }
```

How cool is that? We now have a private method that we can call whenever we have a query that should *only* return published questions. And as a bonus... when we refresh... it doesn't break!

[Making the QueryBuilder Argument Option](#)

But it *is* kind of a bummer that we needed to *first* create this empty `QueryBuilder`. It broke our cool-looking method chaining. Let's see if we can improve this.

Create another private method at the bottom called `getOrCreateQueryBuilder()`. This will accept an *optional* `QueryBuilder` argument - so `QueryBuilder $qb = null`. And, it will *return* a `QueryBuilder`.

58 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 40
41 private function getOrCreateQueryBuilder(QueryBuilder $qb = null): QueryBuilder
42 {
... line 43
44 }
... lines 45 - 56
57 }
```

This is *totally* a convenience method. If the `QueryBuilder` is passed, return it, else, return `$this->createQueryBuilder()` using the same `q` alias.

58 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 40
41 private function getOrCreateQueryBuilder(QueryBuilder $qb = null): QueryBuilder
42 {
43     return $qb ? $this->createQueryBuilder('q');
44 }
... lines 45 - 56
57 }
```

This is useful because, in `addIsAskedQueryBuilder()`, we can add `= null` to make *its* `QueryBuilder` argument optional. Make this work by saying `return $this->getOrCreateQueryBuilder()` passing `$qb`. Then `->andWhere('q.AskedAt IS NOT NULL')`

58 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 34
35 private function addIsAskedQueryBuilder(QueryBuilder $qb = null): QueryBuilder
36 {
37     return $this->getOrCreateQueryBuilder($qb)
38         ->andWhere('q.AskedAt IS NOT NULL');
39 }
... lines 40 - 56
57 }
```

So, if somebody passes us an existing `QueryBuilder`, we use it! But if not, we'll create an empty `QueryBuilder` automatically. That's customer service!

All of this basically just makes the helper method easier to use above. Now we can just `return $this->addIsAskedQueryBuilder()` with *no* `$qb` argument.

58 lines | src/Repository/QuestionRepository.php

```
... lines 1 - 15
16 class QuestionRepository extends ServiceEntityRepository
17 {
... lines 18 - 25
26 public function findAllAskedOrderByNewest()
27 {
28     return $this->addIsAskedQueryBuilder()
29         ->orderBy('q.AskedAt', 'DESC')
30         ->getQuery()
31         ->getResult()
32     ;
33 }
... lines 34 - 56
57 }
```

Before we celebrate and throw a well-deserved taco party, let's make sure it works. Refresh and... it does! Sweet! Tacos!

Next, I've got another shortcut to show you! This time it's about letting Symfony query for an object automatically in the controller... a feature I *love*.

Chapter 15: Automatic Controller Queries: Param Converter

Once again, I have a confession: I've *still* be making us do too much work. Dang!

Head over to `QuestionController` and find the `show()` action. Instead of manually querying for the `Question` object via `findOneBy()`, Symfony can make that query *for* us automatically.

```
101 lines | src/Controller/QuestionController.php
... lines 1 - 13
14 class QuestionController extends AbstractController
15 {
... lines 16 - 72
73 /**
74  * @Route("/questions/{slug}", name="app_question_show")
75  */
76 public function show($slug, EntityManagerInterface $entityManager)
77 {
... lines 78 - 81
82     $repository = $entityManager->getRepository(Question::class);
83     /** @var Question|null $question */
84     $question = $repository->findOneBy(['slug' => $slug]);
... lines 85 - 98
99 }
100 }
```

Automatic Queries

Here's how: replace the `$slug` argument with `Question $question`. The important thing here is *not* the *name* of the argument, but the type-hint: we're type-hinting the argument with an *entity* class.

And... we're done! Symfony will *see* the type-hint and automatically query for a `Question` object *WHERE* `slug =` the `{slug}` route wildcard value.

This means that we *don't* need any of the repository logic down here... or even the 404 stuff. I explain why in a minute. We can also delete my `EntityManagerInterface` argument... and, actually, we haven't needed this `MarkdownHelper` argument for awhile.

```

94 lines | src/Controller/QuestionController.php
... lines 1 - 13
14 class QuestionController extends AbstractController
15 {
... lines 16 - 72
73 /**
74  * @Route("/questions/{slug}", name="app_question_show")
75  */
76 public function show(Question $question)
77 {
78     if ($this->isDebug) {
79         $this->logger->info('We are in debug mode!');
80     }
81
82     $answers = [
83         'Make sure your cat is sitting `purrrfectly` still ?',
84         'Honestly, I like furry shoes better than MY cat',
85         'Maybe... try saying the spell backwards?',
86     ];
87
88     return $this->render('question/show.html.twig', [
89         'question' => $question,
90         'answers' => $answers,
91     ]);
92 }
93 }

```

Before we chat about *what's* going on, let's try it. Refresh the homepage, then click into one of the questions. Yes! It works! You can even see the query in the web debug toolbar. It's exactly what we expect: **WHERE slug =** that slug.

[How... Does this Work?](#)

This magic is *actually* provided by a bundle that we already have installed called SensioFrameworkExtraBundle. When that bundle sees a controller argument that's type-hinted with an entity class, it tries to query for that entity *automatically* by using *all* of the wildcard values.

```

94 lines | src/Controller/QuestionController.php
... lines 1 - 13
14 class QuestionController extends AbstractController
15 {
... lines 16 - 72
73 /**
74  * @Route("/questions/{slug}", name="app_question_show")
75  */
76 public function show(Question $question)
77 {
... lines 78 - 91
92 }
93 }

```

So this works because our wildcard is called **slug**, which *exactly* matches the property name. Quite literally this makes a query where **slug** equals the **{slug}** part of the URL. If we also had an **{id}** wildcard in the URL, then the query would be **WHERE slug = {slug} AND id = {id}**.

It even handles the 404 for us! If we add **foo** to the slug in the URL... we *still* get a 404!

This feature is called a param converter and I freakin' *love* it. But it doesn't always work. If you have a situation where you need a more complex query... or maybe for some reason the wildcard can't match your property name... or you have an extra wildcard that is *not* meant to be in the query, then this won't work. Well, there *is* a way to get it to work - but I don't think it's worth the trouble.

And... that's fine! In those cases, just use your repository object to make the query like you normally would. The

param converter is an *awesome* shortcut for the most common cases.

Next: let's add some *voting* to our question. When we do that, we're going to look closer at the *methods* inside of the **Question** entity, which right now, are just getter and setter methods. Are we allowed to add our own custom methods here? And if so, when should we?

Chapter 16: Smarter Entity Methods

We are on an epic quest to make everything on the question page truly dynamic. In the design, each question can get up and down voted... but this doesn't work yet and the vote count - + 6 - is hardcoded in the template.

To get this working, let's add a new `votes` property to the `Question` entity. When a user clicks the up button, we will increase the votes. When they click down, we'll decrease it. In the future, when we have a true user authentication system, we could make this smarter by recording *who* is voting and preventing someone from voting multiple times. But our simpler plan will work *great* for now.

Adding the votes Property

Step one: add a new field to the entity. We *could* do this by hand by copying an existing property, adjusting the options and then adding getter and setter methods for it. But... it's easier just to run `make:entity`. At your terminal, run:

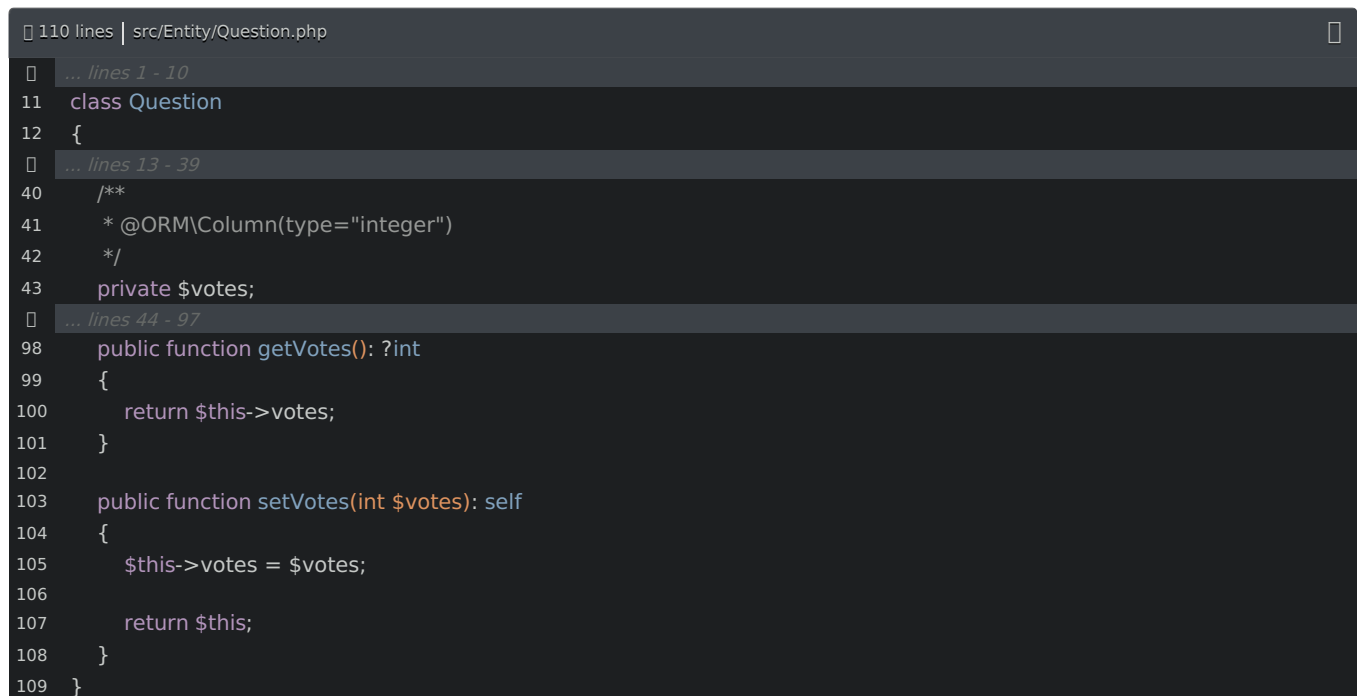


```
$ php bin/console make:entity
```

Once again, I *could* use `symfony console` ... and I probably should. But since this command doesn't need the database environment variables, `bin/console` also works.

This time, enter `Question` so that we can *update* the entity. Yea! `make:entity` can also be used to *modify* an entity! Add a new field called `votes`, make it an `integer` type and set it to *not* nullable in the database. Hit enter to finish.

Ok! Let's go check out the `Question` entity. It looks *exactly* like we expected: a `$votes` property and, at the bottom, `getVotes()` and `setVotes()` methods.



```
110 lines | src/Entity/Question.php
... lines 1 - 10
11 class Question
12 {
... lines 13 - 39
40 /**
41  * @ORM\Column(type="integer")
42  */
43 private $votes;
... lines 44 - 97
98 public function getVotes(): ?int
99 {
100     return $this->votes;
101 }
102
103 public function setVotes(int $votes): self
104 {
105     $this->votes = $votes;
106
107     return $this;
108 }
109 }
```

Let's generate the migration for this. Run:

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

so that the Symfony binary can inject the environment variables. When this finishes, I like to double check the migration to make sure it doesn't contain any surprises.

```
32 lines | migrations/Version20200708195925.php
... lines 1 - 4
5 namespace DoctrineMigrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
9
... lines 10 - 12
13 final class Version20200708195925 extends AbstractMigration
14 {
... lines 15 - 19
20 public function up(Schema $schema) : void
21 {
22     // this up() migration is auto-generated, please modify it to your needs
23     $this->addSql('ALTER TABLE question ADD votes INT NOT NULL');
24 }
25
26 public function down(Schema $schema) : void
27 {
28     // this down() migration is auto-generated, please modify it to your needs
29     $this->addSql('ALTER TABLE question DROP votes');
30 }
31 }
```

This looks perfect. Execute it with:

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

Beautiful!

[Default Values with Doctrine](#)

But... this *did* break one little thing. Go to `/questions/new` - our endpoint to create a new `Question`. And... woh! There's an exception coming from the database:

```
Integrity constraint violation: Column 'votes' cannot be null
```

Hmm, yea: that makes sense. We didn't set the `votes` property, so it's trying to create a new row with `null` for that column. What we probably want to do is default `votes` to be `zero`. How can we set a default value for a column in Doctrine?

Actually, that's not really the right question to ask. A better question would be: how can we default the value of a *property* in *PHP*?

And the answer to that is simple. In `Question`, just say `private $votes = 0`

110 lines | src/Entity/Question.php

```
... lines 1 - 10
11 class Question
12 {
... lines 13 - 39
40 /**
41  * @ORM\Column(type="integer")
42  */
43 private $votes = 0;
... lines 44 - 108
109 }
```

It's that easy. Now, when we instantiate a `Question` object, `votes` will be zero. And when it saves the database... the `votes` column will be zero instead of null. There *is* actually a way inside the `@ORM\Column` annotation to *specifically* set the default value of the column in the *database*, but I've never used it. Setting the default value on the property works beautifully.

Hit the URL again and... it works!

[Giving getVotes\(\) a Non-Nullable Return Type](#)

Back in the entity, scroll down to `getVotes()`. The return type of this method is a *nullable* integer. It was generated that way because there was no guarantee that the `votes` property would ever be set: it *was* possible for `votes` to be `null` in PHP. But thanks to the change we just made, we can now *remove* the question mark: we know that this will *always* be an integer.

110 lines | src/Entity/Question.php

```
... lines 1 - 10
11 class Question
12 {
... lines 13 - 97
98 public function getVotes(): int
99 {
100     return $this->votes;
101 }
... lines 102 - 108
109 }
```

[Rendering the Vote](#)

Before we hook up the voting functionality, let's *render* the vote count. To make this more interesting - because all of the questions in the database right now have *zero* votes - let's set a random vote number for new questions. In `QuestionController`, scroll up to the `new()` action. Near the bottom, add `$question->setVotes()` and pass a random number from negative 20 to 50.

96 lines | src/Controller/QuestionController.php

```
... lines 1 - 13
14 class QuestionController extends AbstractController
15 {
... lines 16 - 40
41 public function new(EntityManagerInterface $entityManager)
42 {
... lines 43 - 62
63     $question->setVotes(rand(-20, 50));
64
65     $entityManager->persist($question);
66     $entityManager->flush();
... lines 67 - 72
73 }
... lines 74 - 94
95 }
```

Back on the browser, I'll refresh `/questions/new` a few times to get some fresh data. Copy the new `slug` and put

that into the address bar to *view* the new Question.

Rendering the true vote count should be easy. Open up `templates/question/show.html.twig` . Find the vote number... `+ 6` and replace it with `{{ question.votes }}`

```
75 lines | templates/question/show.html.twig
... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
... line 14
15              <div class="mt-3">
... lines 16 - 24
25                <div class="vote-arrows vote-arrows-alt flex-fill pt-2" style="min-width: 90px;">
... lines 26 - 27
28                  <span>{{ question.votes }}</span>
29                </div>
30              </div>
31            </div>
... lines 32 - 39
40          </div>
41        </div>
42      </div>
43    </div>
44  </div>
... lines 45 - 72
73 </div>
... lines 74 - 75
```

That's good boring code. Back at the browser, when we refresh... nice! This has minus 10 votes... it must not be a great question.

[Adding the + / - Sign](#)

Because the vote is negative, it naturally has a "minus" sign next to it. But that *won't* be there for a *positive* number. Let me create another `Question` that will hopefully have a positive vote number. Yes! When it's positive, it's just 10, not + 10.

But... our designer actually *does* want positive vote numbers to have a plus sign. No problem. We could add some extra Twig logic: if the number is positive, then add a plus sign before printing the votes.

There's nothing wrong with having simple logic like this in Twig. But if there is *another* place that we could put that logic, that's usually better. In this case, we could add a new method to the `Question` entity itself: a method that returns the *string* representation of the vote count - complete with the + and - signs. That would keep the logic out of Twig and even make that code *reusable*. Heck! We could also unit test it!

Check it out: inside the `Question` entity - it doesn't matter where, but I'll put it right after `getVotes()` so that it's next to related methods - add `public function getVotesString()` with a `string` return type. Inside, I'll paste some logic.

117 lines | src/Entity/Question.php

```
... lines 1 - 10
11 class Question
12 {
... lines 13 - 102
103     public function getVotesString(): string
104     {
... lines 105 - 107
108     }
... lines 109 - 115
116 }
```

This first determines the "prefix" - the plus or minus sign - and then adds that before the number - using the `abs()` function to avoid two minus signs for negative numbers. In other words, this returns the *exact* string we want. How nice is that? Easy to read & reusable.

117 lines | src/Entity/Question.php

```
... lines 1 - 10
11 class Question
12 {
... lines 13 - 102
103     public function getVotesString(): string
104     {
105         $prefix = $this->getVotes() >= 0 ? '+' : '-';
106
107         return sprintf('%s %d', $prefix, abs($this->getVotes()));
108     }
... lines 109 - 115
116 }
```

To use it in Twig, we can say `question.votesString`.

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

```
... lines 1 - 5
6 <div class="container">
7     <div class="row">
8         <div class="col-12">
9             <h2 class="my-4">Question:</h2>
10             <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11                 <div class="q-container-show p-4">
12                     <div class="row">
13                         <div class="col-2 text-center">
... line 14
15                             <div class="mt-3">
... lines 16 - 24
25                             <div class="vote-arrows vote-arrows-alt flex-fill pt-2" style="min-width: 90px;">
... lines 26 - 27
28                                 <span>{{ question.votesString }}</span>
29                             </div>
30                         </div>
31                     </div>
... lines 32 - 39
40                 </div>
41             </div>
42         </div>
43     </div>
44 </div>
... lines 45 - 72
73 </div>
... lines 74 - 75
```

That's it. Let's try it! Over on the browser, refresh and... there it is! + 10!

The *cool* thing about this is that we said `question.votesString` . But... there is *no* `$votesString` property inside of `Question` ! And... that's fine! When we say `question.votesString` , Twig is smart enough to call the `getVotesString()` method.

Now that we're printing the vote number, let's make it possible to *click* these up and down vote buttons. This will be the first time we execute an *update* query *and* we'll get to talk more about "smart" entity methods. That's all next.

Chapter 17: Request Object & POST Data

Time to hook up the vote functionality. Here's the plan: these up and down vote icons are actually *buttons*. I'll show you: in `show.html.twig` ... it's a `<button>` with `name="direction"` and `value="up"` .

```
75 lines | templates/question/show.html.twig
... lines 1 - 5
6 <div class="container">
7   <div class="row">
8     <div class="col-12">
9       <h2 class="my-4">Question:</h2>
10      <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11        <div class="q-container-show p-4">
12          <div class="row">
13            <div class="col-2 text-center">
14              
15              <div class="mt-3">
16                ... lines 16 - 24
25                <div class="vote-arrows vote-arrows-alt flex-fill pt-2" style="min-width: 90px;">
26                  <button class="vote-up btn btn-link" name="direction" value="up"><i class="far fa-arrow-alt-circle-up">
27                  <button class="vote-down btn btn-link" name="direction" value="down"><i class="far fa-arrow-alt-circle-down">
28                  <span>{{ question.votesString }}</span>
29                </div>
30              </div>
31            </div>
32          </div>
33          <div class="col">
34            ... lines 33 - 38
35            <div>
36              <div>
37                <div>
38                  <div>
39                    <div>
40                      <div>
41                        <div>
42                          <div>
43                        </div>
44                      </div>
45                    </div>
46                  </div>
47                </div>
48              </div>
49            </div>
50          </div>
51        </div>
52      </div>
53    </div>
54  </div>
55  </div>
56  </div>
57  </div>
58  </div>
59  </div>
60  </div>
61  </div>
62  </div>
63  </div>
64  </div>
65  </div>
66  </div>
67  </div>
68  </div>
69  </div>
70  </div>
71  </div>
72  </div>
73  </div>
74  </div>
75  </div>
```

Thanks to the `name` and `value` attributes, if we wrapped this in a `<form>` and then click one of these buttons, the form would submit and send a POST parameter called `direction` that's equal to either `up` or `down` , based on which button was clicked. It's like having an extra input in your form.

So that's *exactly* what we're going to do: wrap this in a form, make it submit to a new endpoint, read the `direction` value and increase or decrease the vote count. We *could* do this with an AJAX call instead of a form submit. From a Doctrine and Symfony perspective, it really makes no difference. So I'll keep it simple and leave JavaScript out of this.

Creating a POST-Only Endpoint

Let's start by creating that endpoint. In `src/Controller/QuestionController` - because this is still related to questions - at the bottom, create a new method called `questionVote()` . Above, add the normal `@Route()` . For the URL, how about `/questions/{slug}` - that's equal to the show page above - then `/vote` . And because I *know* we'll need to generate a URL to this route for the form, give it a name: `name=""` , how about, `app_question_vote` . Finally, add `methods="POST"` .

104 lines | src/Controller/QuestionController.php

```
... lines 1 - 13
14 class QuestionController extends AbstractController
15 {
... lines 16 - 95
96 /**
97  * @Route("/questions/{slug}/vote", name="app_question_vote", methods="POST")
98  */
99 public function questionVote(Question $question)
100 {
... line 101
102 }
103 }
```

This means that I can *only* make a **POST** request to this endpoint. If we try to make a **GET** request, the route won't match. That's nice for 2 reasons. First, it's a best-practice: if an endpoint *changes* data on the server, it should *not* allow **GET** requests. The second reason is... really an example of *why* this best practice exists. If we allowed **GET** requests, then it would make it *too* easy to vote: someone could post the voting URL somewhere and unknowing users would vote *just* by clicking it. Worse, bots might *follow* that link and start voting themselves.

Anyways, like before with the show page, we have a **{slug}** route wildcard that we need to use to query for the **Question** object. Let's do that the same way: add an argument with a **Question** type-hint. And, for now, just `dd($question)` .

104 lines | src/Controller/QuestionController.php

```
... lines 1 - 13
14 class QuestionController extends AbstractController
15 {
... lines 16 - 95
96 /**
97  * @Route("/questions/{slug}/vote", name="app_question_vote", methods="POST")
98  */
99 public function questionVote(Question $question)
100 {
101     dd($question);
102 }
103 }
```

Adding the Form

Time for the form. In `show.html.twig` , add a `<form>` element above the vote buttons... and a closing `</form>` after them. Inside the `form` tag, we need a few things, like `action=""` set to `{{ path() }}` to generate a URL to the `app_question_vote` route. Set the `slug` wildcard to `question.slug` . The form tag also needs `method="POST"` .

```

77 lines | templates/question/show.html.twig
... lines 1 - 5
6 <div class="container">
7 <div class="row">
8 <div class="col-12">
9 <h2 class="my-4">Question:</h2>
10 <div style="box-shadow: 2px 3px 9px 4px rgba(0,0,0,0.04);">
11 <div class="q-container-show p-4">
12 <div class="row">
13 <div class="col-2 text-center">
14 
15 <div class="mt-3">
... lines 16 - 24
25 <form action="{{ path('app_question_vote', { slug: question.slug }) }}" method="POST">
26 <div class="vote-arrows vote-arrows-alt flex-fill pt-2" style="min-width: 90px;">
27 <button class="vote-up btn btn-link" name="direction" value="up"><i class="far fa-arrow-alt-circle-up"></i>
28 <button class="vote-down btn btn-link" name="direction" value="down"><i class="far fa-arrow-alt-circle-down"></i>
29 <span>{{ question.votesString }}</span>
30 </div>
31 </form>
32 </div>
33 </div>
... lines 34 - 41
42 </div>
43 </div>
44 </div>
45 </div>
46 </div>
... lines 47 - 74
75 </div>
... lines 76 - 77

```

Cool! With any luck, when we refresh the page, we should be able to click either button to submit to the endpoint. And... yes! Symfony queried for the `Question` object and we dumped it.

Getting the Request Object

This form doesn't *look* much like a traditional HTML form: it doesn't have any inputs or other real fields. But because it *does* have these two buttons and each has a `name="direction"` attribute, when we click a vote button, it will send a `direction` POST field... exactly like if we had typed the word "up" in a text box and submitted.

So the question now is: how can we read POST data from inside of Symfony? Well, whenever you need to read POST data or query parameters or headers, what you're *really* doing is reading information from the `Request`. And, in Symfony, there is a `Request` object that holds *all* of this data. To read POST data, we need to get the `Request` object!

And because needing the request is so common, you can get it in a controller by using its type-hint. Check this out: add `Request` - make sure you get the one from `HttpFoundation` - and then `$request`.

105 lines | src/Controller/QuestionController.php

```
... lines 1 - 10
11 use Symfony\Component\HttpFoundation\Request;
... lines 12 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request)
101 {
... line 102
103 }
104 }
```

This *looks* like service autowiring. It looks *just* like how we can type-hint `EntityManagerInterface` to get that service. But... the truth is that the `Request` is *not* a service in the container.

[All the Arguments Allowed to a Controller Method](#)

What we're seeing here is one of the *final* cases of "things that you can have as controller arguments". Let's review by listing *all* of the things that we're allowed to have as arguments to a controller.

First, we can have an argument whose *name* matches one of the *wildcards* in the route. Second, we can autowire services with their type-hint. Third, we can type-hint an *entity* class to tell Symfony to automatically query for it. And *finally*, we can type-hint the `Request` class to get the request. Yep, this *specific* class has its own special case.

There *are* a few other possible types of arguments that you can have in your controllers, but these are the main ones.

[Fetching POST Data](#)

Now that we have the `Request` object, we're in luck! This is a simple class: it has a bunch of methods & properties to help us read *anything* from the request, like POST parameters, headers, cookies or the IP address. If you need to read some info from the request, it's usually a matter of just looking at the class or Googling:

Symfony request ip address

to find the right method. Let's dump one part of the request: `$request->request->all()`.

105 lines | src/Controller/QuestionController.php

```
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request)
101 {
102     dd($question, $request->request->all());
103 }
104 }
```

Yeah, I know: it looks a little funny: `$request->request`? Technically, POST parameters are known as "request" parameters. So this `$request->request` is a small object that holds *all* of the POST parameters. The `->all()` method returns them as an array.

So when we go over now and refresh... yes! We see `'direction' => 'up'`!

[The UPDATE Query](#)

Now, we're dangerous. In the controller, add `$direction = $request->`. Oh, and here you can see some other ways to get data - like `$request->query` is how you get query parameters and `$request->headers->get()` can be used to read a header. In this case, use `$request->request->get('direction')`.

112 lines | src/Controller/QuestionController.php

```
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request)
101 {
102     $direction = $request->request->get('direction');
... lines 103 - 108
109     dd($question);
110 }
111 }
```

Now, if `$direction === 'up'`, then `$question->setVotes($question->getVotes() + 1)`. Else if `$direction === 'down'`, do the same thing, but `- 1`.

112 lines | src/Controller/QuestionController.php

```
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request)
101 {
102     $direction = $request->request->get('direction');
103
104     if ($direction === 'up') {
105         $question->setVotes($question->getVotes() + 1);
106     } elseif ($direction === 'down') {
107         $question->setVotes($question->getVotes() - 1);
108     }
109     dd($question);
110 }
111 }
```

If the direction is some other value, let's just ignore it. That probably means that someone is messing with our form and ignoring it is safe. At the bottom, `dd($question)` to see what it looks like.

Ok, right now this question has 10 votes. When we refresh... yes! 11! Go back to the show page and hit down. 9!

But... this did *not* save to the database yet: it's just updating the value on our PHP object. And also, I think we can accomplish this `+ 1` and `- 1` logic in a cleaner way.

Next, let's talk about anemic versus rich models. Then we'll learn how to make an `UPDATE` query to update the vote count. Hint: we already know how to do this.

Chapter 18: Update Query & Rich vs Anemic Models

On the show page, we can now up vote or down vote the question...mostly. In the controller, we read the **direction** POST parameter to know which button was clicked and change the vote count. This doesn't save to the database yet, but we'll do that in a few minutes.

[Adding upVote and downVote Methods](#)

Before we do, we have another opportunity to improve our code. The logic inside the controller to increase or decrease the vote isn't complex, but it *could* be simpler and more descriptive.

In **Question**, at the bottom, add a new **public function** called **upVote()**. I'm going to make this return **self**.

```
131 lines | src/Entity/Question.php
... lines 1 - 10
11 class Question
12 {
... lines 13 - 116
117     public function upVote(): self
118     {
... lines 119 - 121
122     }
... lines 123 - 129
130 }
```

Inside, say **\$this->votes++**. Then, **return \$this...** just because that allows method chaining. All of the setter methods return **\$this**.

```
131 lines | src/Entity/Question.php
... lines 1 - 10
11 class Question
12 {
... lines 13 - 116
117     public function upVote(): self
118     {
119         $this->votes++;
120
121         return $this;
122     }
... lines 123 - 129
130 }
```

Copy this, paste, and create another called **downVote()** that will do **\$this->votes--**.

```
131 lines | src/Entity/Question.php
... lines 1 - 10
11 class Question
12 {
... lines 13 - 123
124     public function downVote(): self
125     {
126         $this->votes--;
127
128         return $this;
129     }
130 }
```

I'm not going to bother adding any PHP documentation above these, because... their names are already so

descriptive: `upVote()` and `downVote()` !

I love doing this because it makes the code in our controller so nice. If the direction is `up` , `$question->upVote()` . If it's `down` , `$question->downVote()` .

```
112 lines | src/Controller/QuestionController.php
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request)
101 {
102     $direction = $request->request->get('direction');
103
104     if ($direction === 'up') {
105         $question->upVote();
106     } elseif ($direction === 'down') {
107         $question->downVote();
108     }
109     dd($question);
110 }
111 }
```

How beautiful is that? And when we move over to try it... we're still good!

[Rich vs Anemic Models](#)

We've now added *three* custom methods to `Question` : `upVote()` , `downVote()` and `getVotesString()` . And this touches on a somewhat controversial topic related to entities. Notice that every property in our entity has a getter and setter method. This makes the entity super flexible: you can get or set any field you want.

But sometimes you might not need - or even *want* - a getter or setter method. For example, do we really want a `setVotes()` method? Should anything in our app be able to set the vote directly to *any* number? Probably not. Probably we will always want to use `upVote()` or `downVote()` .

Now, I *will* keep this method... but only because we're using it in `QuestionController` . In the `new()` method... we're using it to set the fake data.

But this touches on a really interesting idea: by removing any unnecessary getter or setter methods in your entity and *replacing* them with more descriptive methods that fit your business logic - like `upVote()` and `downVote()` - you can, little by little, give your entities more clarity. `upVote()` , and `downVote()` are *very* clear & descriptive. Someone calling these doesn't even need to know or care how they work internally.

Generally-speaking, an "anemic" model is a class where you can directly modify and access its properties (e.g. via getter/setter methods). A "rich" model is where you, instead, create methods specific to your business logic - like `upVote()` .

Some people take this to an extreme and have almost *zero* getter and setter methods on their entities. Here at Symfonycasts, we tend to be more pragmatic. We usually have getters and setters method, but we always look for ways to be more descriptive - like `upVote()` and `downVote()` .

[Updating an Entity in the Database](#)

Okay, let's finish this! In our controller, back down in `questionVote()` , how can we execute an *update* query to save the new vote count to the database? Well, no surprise, whenever we need to *save* something in Doctrine, we need the entity manager.

Add another argument: `EntityManagerInterface $entityManager` .


```

117 lines | src/Controller/QuestionController.php
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request, EntityManagerInterface $entityManager)
101 {
... lines 102 - 114
115 }
116 }

```

Then, below, replace the `dd($question)` with `$entityManager->flush()` .

```

117 lines | src/Controller/QuestionController.php
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request, EntityManagerInterface $entityManager)
101 {
102     $direction = $request->request->get('direction');
103
104     if ($direction === 'up') {
105         $question->upVote();
106     } elseif ($direction === 'down') {
107         $question->downVote();
108     }
109
110     $entityManager->flush();
... lines 111 - 114
115 }
116 }

```

Done! Seriously! Doctrine is smart enough to *realize* that the `Question` object already exists in the database and make an *update* query instead of an insert. *We* don't need to worry about "is this an insert or an update" at all? Doctrine has that covered.

[No persist\(\) on Update?](#)

But wait, didn't I forget the `persist()` call? Up in the `new()` action, we learned that to insert something, we need to get the entity manager and then call `persist()` and `flush()` .

This time, we *could* have added `persist()` , but we don't need to. Scroll back up to `new()` . Remember: the point of `persist()` is to make Doctrine *aware* of your object so that when you call `flush()` , it knows to *check* that object and execute whatever query it needs to save that into the database, whether that is an INSERT or UPDATE query.

Down in `questionVote()` , because Doctrine was used to *query* for this `Question` object... it's *already* aware of it! When we call `flush()` , it already knows to check the `Question` object for changes and performs an UPDATE query. Doctrine is smart.

[Redirecting](#)

Ok, now that this is saving... what should our controller return? Well, usually after a form submit, we will redirect somewhere. Let's do that. How? `return $this->redirectToRoute()` and then pass the name of the route that we want to redirect to. Let's use `app_question_show` to redirect to the show page and then pass any wildcard values as the second argument: `slug` set to `$question->getSlug()` .

```

117 lines | src/Controller/QuestionController.php
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
... lines 17 - 99
100 public function questionVote(Question $question, Request $request, EntityManagerInterface $entityManager)
101 {
102     $direction = $request->request->get('direction');
103
104     if ($direction === 'up') {
105         $question->upVote();
106     } elseif ($direction === 'down') {
107         $question->downVote();
108     }
109
110     $entityManager->flush();
111
112     return $this->redirectToRoute('app_question_show', [
113         'slug' => $question->getSlug()
114     ]);
115 }
116 }

```

Two things about this. First, until now, we've *only* generated URLs from inside of Twig, by using the `{{ path() }}` function. We pass the same arguments to `redirectToRoute()` because, internally, it generates a URL just like `path()` does.

And second... more of a question. On a high level... what *is* a redirect? When a server wants to redirect you to another page, how does it do that?

A redirect is *nothing* more than a special type of *response*. It's a response that has a 301 or 302 status code and a `Location` header that tells your browser where to go.

Let's do some digging and find out how `redirectToRoute()` does this. Hold Command or Ctrl and click `redirectToRoute()` to jump to that method inside of `AbstractController`. This apparently calls another method: `redirect()`. Hold Command or Ctrl again to jump to that.

Ah, *here's* the answer: this returns a `RedirectResponse` object. Hold Command or Ctrl *one* more time to jump into this class.

`RedirectResponse` live deep in the core of Symfony and it *extends* `Response`! Yes this is just a special subclass of `Response` that's really good at creating *redirect* responses.

Let's close all of these core classes. The point is: the `redirectToRoute()` method doesn't do anything magical: it simply returns a `Response` object that's really good at redirecting.

Ok: testing time! Spin over to your browser and go back to the show page. Right now this has 10 votes. Hit "up vote" and... 11! Do it again: 12! Then... 13! Downvote... 12. We got it!

Like I said earlier, in a real app, when we have user authentication, we might prevent someone from voting multiple times. But, we can worry about that later.

Next: we *have* created a way to load dummy data into our database via the `/questions/new` page. But... it's pretty hacky.

Let's replace this with a proper *fixtures* system.

Chapter 19: Data Fixtures

Our `/questions/new` page is nice... it gave us a simple way to create and save some dummy data, so that we could have enough to work on the homepage & show page.

Having a rich set of data to work with while you're developing is *pretty* important. Without it, you're going to spend a lot of time constantly setting up your database before you work on something. It's a *big* waste in the long-run.

Installing DoctrineFixturesBundle

This "dummy data" has a special name: data fixtures. And instead of creating them in a random controller like `QuestionController`, we can install a bundle to do it properly. Find your terminal and run:



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

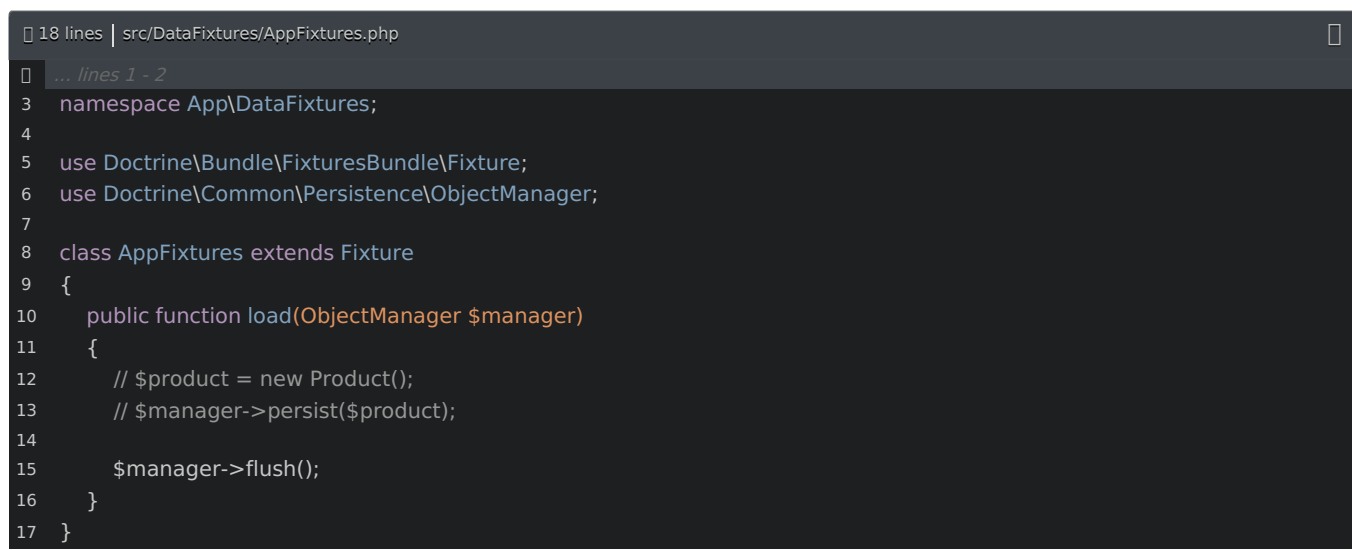
This is another flex alias: `orm-fixtures` installs `doctrine/doctrine-fixtures-bundle`

When this finishes... it installed a recipe! I committed all of my changes before recording, so I'll run:



```
$ git status
```

to see what it did. Ok: it updated the normal `composer.json`, `composed.lock` and `symfony.lock` files, it enabled the bundle *and* ooh: it created a new `src/DataFixtures/` directory! Let's go see what's inside `src/DataFixtures/` - a shiny new `AppFixtures` class!



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

The DoctrineFixturesBundle that we just installed is a *beautifully* simple bundle. First, we create one or more of these fixture classes: classes that extend `Fixture` and have this `load()` method. Second, inside `load()`, we write normal PHP code to create as many dummy objects as we want. And third, we run a new console command that will call the `load()` method on every fixture class.

Fixing the Type-Hint

Before we get to work, PhpStorm is mad! The details aren't too important and this code *would* work... despite what PhpStorm is saying. But to remove the warning and make our code future-proof with newer versions of

Doctrine, find the `ObjectManager` type-hint and replace it with one from `Doctrine\Persistence`.

```
18 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 2
3 namespace App\DataFixtures;
4
5 use Doctrine\Bundle\FixturesBundle\Fixture;
6 use Doctrine\Persistence\ObjectManager;
7
8 class AppFixtures extends Fixture
9 {
10     public function load(ObjectManager $manager)
11     {
12         ... lines 12 - 15
13     }
14 }
15 }
```

Creating Dummy Data

Anyways: let's see this bundle in action. Find the `new()` method in the controller, copy *all* of the question-creating code and delete it. We'll *properly* create this page in a future tutorial when we talk about forms... so let's just render that: this sounds like a *great* feature for V2!

```
88 lines | src/Controller/QuestionController.php
... lines 1 - 14
15 class QuestionController extends AbstractController
16 {
17     ... lines 17 - 41
18
19     public function new()
20     {
21         return new Response('Sounds like a GREAT feature for V2!');
22     }
23
24     ... lines 46 - 86
25 }
26 }
```

Back in `AppFixtures`, paste the code and... check it out! PhpStorm was smart enough to *see* that we're using the `Question` class and ask us if we want to import its `use` statement. We definitely do!

```

40 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 8
9 class AppFixtures extends Fixture
10 {
11     public function load(ObjectManager $manager)
12     {
13         $question = new Question();
14         $question->setName('Missing pants')
15             ->setSlug('missing-pants-' . rand(0, 1000))
16             ->setQuestion(<<<EOF
17 Hi! So... I'm having a *weird* day. Yesterday, I cast a spell
18 to make my dishes wash themselves. But while I was casting it,
19 I slipped a little and I think `I also hit my pants with the spell`.
20
21 When I woke up this morning, I caught a quick glimpse of my pants
22 opening the front door and walking out! I've been out all afternoon
23 (with no pants mind you) searching for them.
24
25 Does anyone have a spell to call your pants back?
26 EOF
27     );
28
29     if (rand(1, 10) > 2) {
30         $question->setAskedAt(new \DateTime(sprintf('%d days', rand(1, 100))));
31     }
32
33     $question->setVotes(rand(-20, 50));
34
35     $entityManager->persist($question);
36
37     $manager->flush();
38 }
39 }

```

The only problem now is that we don't have an `$entityManager` variable. Hmm, but we *do* have a `$manager` variable that's passed to the `load()` method - it's an `ObjectManager` ?

This is *actually* the entity manager in disguise: `ObjectManager` is an interface that it implements. So change the `persist()` call to `$manager` ... and we only need one `flush()` .

```

40 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 4
5 use App\Entity\Question;
... lines 6 - 8
9 class AppFixtures extends Fixture
10 {
11     public function load(ObjectManager $manager)
12     {
... lines 13 - 34
35         $manager->persist($question);
36
37         $manager->flush();
38     }
39 }

```

Done! Well, this isn't a very interesting fixture class... it's only going to create *one* `Question` but it's a good start. Let's see if it works!

[Executing the Fixtures](#)

Head over to your terminal. The new bundle gave us one new command: `doctrine:fixtures:load` . Execute that

through the symfony binary:

A terminal window with a dark background and three light gray window control buttons (minimize, maximize, close) in the top-left corner. The terminal has a light gray input area containing the command `$ symfony console doctrine:fixtures:load`.

It asks us to confirm because *each* time we run this command, it will *completely* empty the database before loading the new data. And... I think it worked! Go check out the homepage. Refresh and... yes! We have the *one* question from the fixture class.

This isn't *that* useful yet, but it gave us a chance to see how the bundle works. Oh, and if you don't see *anything* on this page, it's probably because the *one* **Question** that was loaded has an **askedAt** set to null... so it's not showing up. Try re-running the command once or twice to get a fresh **Question** .

So what I *love* about DoctrineFixturesBundle is how simple it is: I have a **load()** method where I can create and save as many objects as I want. We can even create *multiple* fixtures classes to organize better and we can control the *order* in which each is called.

What I *hate* about DoctrineFixturesBundle is that... I need to do all this work by hand! If you start creating a *lot* of objects - especially once you have database relationships where objects are linked to *other* objects... these classes can get ugly fast. And they're not much fun to write.

So, next: let's use a shiny new library called Foundry to create numerous, random, rich dummy data.

Chapter 20: Foundry: Fixture Model Factories

In the `load()` method of the fixture class, we can create as much dummy data as we want. Right now... we're creating exactly *one* `Question` ... which isn't making for a very realistic experience.

If we created *more* questions... and especially in the future when we will have multiple database tables that relate to each other, this class would start to get ugly. It's... already kind of ugly.

[Hello Foundry!](#)

No, we deserve better! Let's use a super fun new library instead. Google for "Zenstruck Foundry" and find its [GitHub Page](#).

Foundry is *all* about creating Doctrine entity objects in an easy, repeatable way. It's perfect for fixtures as well as for functional tests where you want to seed your database with data at the start of each test. It even has extra features for test assertions!

The bundle was created by [Kevin Bond](#) - a *long* time Symfony contributor and friend of mine who's been creating some *really* excellent libraries lately. Foundry is Canadian for fun!

[Installing Foundry](#)

Let's get to work! Scroll down to the installation, copy the composer require line, find your terminal and paste. The `--dev` is here because we only need to load dummy data in the `dev` & `test` environments.

A terminal window with a dark background and three window control buttons in the top left. The command `$ composer require zenstruck/foundry --dev` is entered in the terminal's input field.

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

While that's running, head back to the docs. Let me show you what this bundle is all about. Suppose you have entities like `Category` or `Post`. The idea is that, for each entity, we will generate a corresponding *model factory*. So, a `Post` entity will have a `PostFactory` class, which will look something like this.

Once we have that, we can configure some default data for the entity class and then... start creating objects!

I know I explained that quickly, but that's because we're going to see this in action. Back at the terminal... let's wait for this to finish. I'm actually recording at my parents' house... where the Internet is *barely* a step up from dial-up.

After an edited break where I ate a sandwich and watched Moana, this finally finishes.

[make:factory](#)

Let's generate one of those fancy-looking model factories for `Question`. To do that, run:

A terminal window with a dark background and three window control buttons in the top left. The command `$ symfony console make:factory` is entered in the terminal's input field.

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

I also could have run `bin/console make:factory` ... because this command doesn't need the database environment variables... but it's easier to get in the habit of *always* using `symfony console`.

Select `Question` from the list and... done! Go check out the new class `src/Factory/QuestionFactory.php`.

```

42 lines | src/Factory/QuestionFactory.php
... lines 1 - 2
3 namespace App\Factory;
4
5 use App\Entity\Question;
6 use App\Repository\QuestionRepository;
7 use Zenstruck\Foundry\RepositoryProxy;
8 use Zenstruck\Foundry\ModelFactory;
9 use Zenstruck\Foundry\Proxy;
10
11 /**
12  * @method static Question|Proxy findOrCreate(array $attributes)
13  * @method static Question|Proxy random()
14  * @method static Question[]|Proxy[] randomSet(int $number)
15  * @method static Question[]|Proxy[] randomRange(int $min, int $max)
16  * @method static QuestionRepository|RepositoryProxy repository()
17  * @method Question|Proxy create($attributes = [])
18  * @method Question[]|Proxy[] createMany(int $number, $attributes = [])
19  */
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
25             // TODO add your default values here (https://github.com/zenstruck/foundry#model-factories)
26         ];
27     }
28
29     protected function initialize(): self
30     {
31         // see https://github.com/zenstruck/foundry#initialization
32         return $this
33             // ->beforeInstantiate(function(Question $question) {});
34     };
35 }
36
37 protected static function getClass(): string
38 {
39     return Question::class;
40 }
41 }

```

Adding Default Values

The only method that we need to worry about right now is `getDefaults()`. The idea is that we'll return an array of all of the data needed to create a `Question`. For example, we can set a `name` key to our dummy question name - "Missing pants".

```

57 lines | src/Factory/QuestionFactory.php
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
25             'name' => 'Missing pants',
26             ... lines 26 - 40
27         ];
28     }
29     ... lines 43 - 55
30 }

```


This works a bit like Twig. When Foundry sees the `name` key, it will call the `setName()` method on `Question`. Internally, this uses Symfony's property-access component, which I'm mentioning, because it also supports passing data through the constructor if you need that.

Copy the rest of the dummy code from our fixture class, delete it... and delete *everything* actually.

```
19 lines | src/DataFixtures/AppFixtures.php
... lines 1 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager)
13     {
14         ... lines 14 - 15
16         $manager->flush();
17     }
18 }
```

Back in `QuestionFactory`, paste!

But we need to convert all of this into array keys. As *exciting* as this is... I'll... type really fast.

```
57 lines | src/Factory/QuestionFactory.php
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
25             'name' => 'Missing pants',
26             'slug' => 'missing-pants-' . rand(0, 1000),
27             'question' => <<<EOF
28 Hi! So... I'm having a *weird* day. Yesterday, I cast a spell
29 to make my dishes wash themselves. But while I was casting it,
30 I slipped a little and I think `I also hit my pants with the spell`.
31
32 When I woke up this morning, I caught a quick glimpse of my pants
33 opening the front door and walking out! I've been out all afternoon
34 (with no pants mind you) searching for them.
35
36 Does anyone have a spell to call your pants back?
37 EOF
38 ,
39             'askedAt' => rand(1, 10) > 2 ? new \DateTime(sprintf('-%d days', rand(1, 100))) : null,
40             'votes' => rand(-20, 50),
41         ];
42     }
43     ... lines 43 - 55
56 }
```

And.... done! Phew...

[Using the Factory](#)

Ok! We now have a simple array of "default" values that are enough to create a valid `Question` object. Our `QuestionFactory` is ready! Let's use it in `AppFixtures`.

How? First, say `QuestionFactory::new()`. That will give us a new *instance* of the `QuestionFactory`. Now `->create()` to create a *single* `Question`.

19 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 5
6 use App\Factory\QuestionFactory;
... lines 7 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager)
13     {
14         QuestionFactory::new()->create();
15
16         $manager->flush();
17     }
18 }
```

Done! Ok, it's *still* not interesting - it will create just *one* Question ... but let's try it! Re-run the fixtures:

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

Answer yes and... no errors! Head over to the browser, refresh and... oh! Zero questions! Ah, my *one* question is probably unpublished. Load the fixtures again:

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

Refresh and... there it is!

[createMany\(\)](#)

At this point, you might be wondering: why is this better? Valid question. It's better because we've only just *started* to scratch the service of what Foundry can do. Want to create 20 questions instead of just one? Change `create()` to `createMany(20)`.

In the latest version of Foundry, creating many objects is easier: just `QuestionFactory::createMany()`. You can also create a single object with the shorter `QuestionFactory::createOne()`.

19 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 5
6 use App\Factory\QuestionFactory;
... lines 7 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager)
13     {
14         QuestionFactory::new()->createMany(20);
15
16         $manager->flush();
17     }
18 }
```

That's it. Reload the fixtures again:

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

Then go check out the homepage. Hello 20 questions created with one line of very readable code.

But wait there's *more* Foundry goodness! Foundry comes with built-in support for a library called faker. A handy tool for creating *truly* fake data. Let's improve the quality of our fake data *and* see a few other cool things that Foundry can do next.

Chapter 21: Foundry Tricks

In `QuestionFactory`, we're already doing a pretty good job of making some of this data random so that all of the questions aren't identical. To help with this, Foundry comes with built-in support for Faker: a library that's great at creating all *kinds* of interesting, fake data.

Using Faker

If you look at the top of the Foundry docs, you'll see a section called Faker and a link to the [Faker documentation](#). This tells you everything that Faker can do... which is... a *lot*. Let's use it to make our fixtures even better.

The Faker library now has a new home! At <https://github.com/FakerPHP/Faker>. Same great library, shiny new home.

For example, for the random -1 to -100 days, we can make it more readable by replacing the `new \DateTime()` with `self::faker()` - that's how you can get an instance of the `Faker` object - then `->dateTimeBetween()` to go from -100 days to -1 day.

```
57 lines | src/Factory/QuestionFactory.php
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
... lines 25 - 38
39         'askedAt' => rand(1, 10) > 2 ? self::faker()->dateTimeBetween('-100 days', '-1 days') : null,
... line 40
41     ];
42 }
... lines 43 - 55
56 }
```

And because this is more flexible, we can even change it from -100 days to -1 minute!

```
57 lines | src/Factory/QuestionFactory.php
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
... lines 25 - 38
39         'askedAt' => rand(1, 10) > 2 ? self::faker()->dateTimeBetween('-100 days', '-1 minute') : null,
... line 40
41     ];
42 }
... lines 43 - 55
56 }
```

Even the random true/false condition at the beginning can be generated by Faker. What we *really* want is to create published questions about 70% of the time. We can do that with `self::faker()->boolean(70)`:

```

57 lines | src/Factory/QuestionFactory.php
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
... lines 25 - 38
39         'askedAt' => self::faker()->boolean(70) ? self::faker()->dateTimeBetween('-100 days', '-1 minute') : null,
... line 40
41     ];
42 }
... lines 43 - 55
56 }

```

This is cool, but the *real* problem is that the name and question are always the same. That is *definitely* not realistic. Let's fix that: set `name` to `self::faker()->realText()` to get several words of "real looking" text:

```

57 lines | src/Factory/QuestionFactory.php
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
25             'name' => self::faker()->realText(50),
... lines 26 - 40
41     ];
42 }
... lines 43 - 55
56 }

```

For `slug`, there's a feature for that! `self::faker()->slug` :

```

57 lines | src/Factory/QuestionFactory.php
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
... line 25
26         'slug' => self::faker()->slug,
... lines 27 - 40
41     ];
42 }
... lines 43 - 55
56 }

```

Direct property access is deprecated since v1.14 of `fakerphp/faker` - use `self::faker()->slug()` instead of `self::faker()->slug`

Finally, for the question text, it can be made much more interesting by using `self::faker()->paragraphs()` .

49 lines | src/Factory/QuestionFactory.php

```
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
25             ... lines 25 - 26
26             'question' => self::faker()->paragraphs(
27                 ... lines 28 - 29
28             ),
29             ... lines 31 - 32
30         ];
31     }
32 }
... lines 35 - 47
48 }
```

Faker lets you use `paragraphs` like a property or you can call a function and pass arguments, which are the *number* of paragraphs and whether you want them returned as text - which we do - or as an array. For the number of paragraphs, we can use Faker again! `self::faker()->numberBetween(1, 4)` and then `true` to return this as a string.

49 lines | src/Factory/QuestionFactory.php

```
... lines 1 - 19
20 final class QuestionFactory extends ModelFactory
21 {
22     protected function getDefaults(): array
23     {
24         return [
25             ... lines 25 - 26
26             'question' => self::faker()->paragraphs(
27                 self::faker()->numberBetween(1, 4),
28                 true
29             ),
30             ... lines 31 - 32
31         ];
32     }
33 }
... lines 35 - 47
48 }
```

Let's take this for a test drive! Find your terminal and reload the fixtures with:

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

Go check the homepage and... yea!

Oh, but the "real text" for the name is *way* too long. What I meant to do is pass `->realText(50)`. Let's reload the fixtures again:

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

And... there we go! We now have *many* `Question` objects *and* they represent a rich set of unique data. This is why I *love* Foundry.

[Doing Things Before Saving](#)

If you click into one of the questions, you can see that the `slug` is unique... but was generated in a way that is

completely unrelated to the question's `name`. That's "maybe" ok... but it's not realistic. Could we fix that?

Of course! Foundry comes with a nice "hook" system where we can do actions before or after each item is saved. Inside `QuestionFactory`, the `initialize()` method is where you can *add* these hooks.

Remove the `slug` key from `getDefaults()` and, instead, down here, uncomment this `beforeInstantiate()` and change it to `afterInstantiate()`.

```
54 lines | src/Factory/QuestionFactory.php
... lines 1 - 20
21 final class QuestionFactory extends ModelFactory
22 {
23     protected function getDefaults(): array
24     {
25         return [
26             'name' => self::faker()->realText(50),
27             'question' => self::faker()->paragraphs(
28                 self::faker()->numberBetween(1, 4),
29                 true
30             ),
31             'askedAt' => self::faker()->boolean(70) ? self::faker()->dateTimeBetween('-100 days', '-1 minute') : null,
32             'votes' => rand(-20, 50),
33         ];
34     }
... lines 35 - 52
53 }
```

So `afterInstantiate()`, we want to run this function. Inside, to generate a random slug based off of the name, we can say: if *not* `$question->getSlug()` - in case we set it manually for some reason:

```
54 lines | src/Factory/QuestionFactory.php
... lines 1 - 20
21 final class QuestionFactory extends ModelFactory
22 {
... lines 23 - 35
36     protected function initialize(): self
37     {
38         // see https://github.com/zenstruck/foundry#initialization
39         return $this
40             ->afterInstantiate(function(Question $question) {
41                 if (!$question->getSlug()) {
... lines 42 - 43
44                 }
45             })
46         ;
47     }
... lines 48 - 52
53 }
```

then use Symfony's Slugger - `$slugger = new AsciiSlugger()` :

54 lines | src/Factory/QuestionFactory.php

```
... lines 1 - 6
7 use Symfony\Component\String\Slugger\AsciiSlugger;
... lines 8 - 20
21 final class QuestionFactory extends ModelFactory
22 {
... lines 23 - 35
36 protected function initialize(): self
37 {
38     // see https://github.com/zenstruck/foundry#initialization
39     return $this
40         ->afterInstantiate(function(Question $question) {
41             if (!$question->getSlug()) {
42                 $slugger = new AsciiSlugger();
... line 43
44             }
45         })
46     ;
47 }
... lines 48 - 52
53 }
```

and set it with `$question->setSlug($slugger->slug($question->getName()))` .

54 lines | src/Factory/QuestionFactory.php

```
... lines 1 - 6
7 use Symfony\Component\String\Slugger\AsciiSlugger;
... lines 8 - 20
21 final class QuestionFactory extends ModelFactory
22 {
... lines 23 - 35
36 protected function initialize(): self
37 {
38     // see https://github.com/zenstruck/foundry#initialization
39     return $this
40         ->afterInstantiate(function(Question $question) {
41             if (!$question->getSlug()) {
42                 $slugger = new AsciiSlugger();
43                 $question->setSlug($slugger->slug($question->getName()));
44             }
45         })
46     ;
47 }
... lines 48 - 52
53 }
```

Nice! Let's try it. Move over, reload the fixtures again:

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

And... go back to the homepage. Let's see: if I click the first one... yes! It works. It has some uppercase letters... which we could normalize to lowercase. But I'm not going to worry about that because, in a few minutes, we'll add an even *better* way of generating slugs across our entire system.

Foundry "State"

Let's try *one* last thing with Foundry. To have nice testing data, we need a mixture of published and unpublished questions. We're currently accomplishing that by randomly setting some `askedAt` properties to `null` . Instead let's create two different *sets* of fixtures: exactly 20 that are published and exactly 5 that are *unpublished* .

To do this, first remove the randomness from `askedAt` in `getDefaults()` : let's *always* set this.

```
59 lines | src/Factory/QuestionFactory.php
... lines 1 - 20
21 final class QuestionFactory extends ModelFactory
22 {
... lines 23 - 27
28     protected function getDefaults(): array
29     {
30         return [
... lines 31 - 35
36         'askedAt' => self::faker()->dateTimeBetween('-100 days', '-1 minute'),
... line 37
38     ];
39 }
... lines 40 - 57
58 }
```

If we stopped here, we would, of course, have 20 questions that are all published. But now, add a new public function to the factory: `public function unpublished()` that returns `self` .

```
59 lines | src/Factory/QuestionFactory.php
... lines 1 - 20
21 final class QuestionFactory extends ModelFactory
22 {
23     public function unpublished(): self
24     {
... line 25
26     }
... lines 27 - 57
58 }
```

I *totally* just made up that name. Inside, `return $this->addState()` and pass it an array with `askedAt` set to null.

```
59 lines | src/Factory/QuestionFactory.php
... lines 1 - 20
21 final class QuestionFactory extends ModelFactory
22 {
23     public function unpublished(): self
24     {
25         return $this->addState(['askedAt' => null]);
26     }
... lines 27 - 57
58 }
```

Here's the deal: when you call `addState()` , it *changes* the default data inside this *instance* of the factory. Oh, and the return statement here just helps to return `self` ... which allows method chaining.

To use this, go back to `AppFixtures` . Start with `QuestionFactory::new()` - to get a *second* instance of `QuestionFactory` :

24 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager)
13     {
14         ... lines 14 - 15
15         QuestionFactory::new()
16         ... lines 17 - 18
17         ;
18         ... lines 20 - 21
19     }
20 }
21 }
```

then `->unpublished()` to change the default `askedAt` data. You can see why I called the method `unpublished()` : it makes this super clear.

24 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager)
13     {
14         ... lines 14 - 15
15         QuestionFactory::new()
16         ->unpublished()
17         ... line 18
18         ;
19         ... lines 20 - 21
20     }
21 }
22 }
```

Finish with `->createMany(5)` .

24 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager)
13     {
14         ... lines 14 - 15
15         QuestionFactory::new()
16         ->unpublished()
17         ->createMany(5)
18         ;
19         ... lines 20 - 21
20     }
21 }
22 }
```

I *love* this! It reads like a story: create a new factory, make everything unpublished and create 5.

Let's... even make sure it works! At the terminal, reload the fixtures:

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

Then... refresh the homepage.

All good! If we dug into the database, we'd find 20 published questions and five unpublished. Foundry can do more - especially with Doctrine relations and testing - and we'll talk about Doctrine relations in the next tutorial.

But first, the `slug` property *is* being set automatically in our fixtures. That's cool... but I'd *really* love for the slug to automatically be set to a URL-safe version of the `name` no matter *where* we create a `Question` object. Basically, we shouldn't *never* need to worry about setting the slug manually.

So next let's install a bundle that will give our entity Sluggable *and* Timestampable superpowers.

Chapter 22: Sluggable: Doctrine Extensions

The whole point of the `slug` is to be a URL-safe version of the `name`. And, ideally, this wouldn't be something we need to set manually... or even think about! In a perfect world, we would be able to set the name of a `Question`, save and something *else* would automatically calculate a unique `slug` from the `name` right before the INSERT query.

We accomplished this in our fixtures, but *only* there. Let's accomplish this *everywhere*.

[Hello StofDoctrineExtensionsBundle](#)

To do that, we're going to install another bundle. Google for StofDoctrineExtensionsBundle and find its GitHub page. And then click over to its documentation, which lives on Symfony.com. This bundle gives you a *bunch* of superpowers for entities, including one called Sluggable. And actually, the bundle is just a tiny layer around *another* library called `doctrine extensions`.

This is where the majority of the documentation lives. Anyways, let's get the bundle installed. Find your terminal and run:

```
$ composer require "stof/doctrine-extensions-bundle:^1.5.0"
```

You can find this command in the bundle's documentation.

[Contrib Recipes](#)

And, oh, interesting! The install stops and says:

The recipe for this package comes from the contrib repository, which is open to community contributions. Review the recipe at this URL. Do you want to execute this recipe?

When you install a package, Symfony Flex looks for a *recipe* for that package... and recipes can live in one of *two* different repositories. The first is `symfony/recipes`, which is the main recipe repository and is closely guarded: it's hard to get recipes accepted here.

The other repository is called `symfony/recipes-contrib`. This is still guarded for quality... but it's much easier to get recipe accepted here. For that reason, the first time you install a recipe from `recipes-contrib`, Flex asks you to make sure that you want to do that. So you can say yes or I'm actually going to say P for yes, permanently.

I committed my changes before recording, so when this finishes I'll run,

```
$ git status
```

to see what the recipe did! Ok: it enabled the bundle - of course - and it also created a new config file `stof_doctrine_extensions.yaml`. Let's go check that out: `config/packages/stof_doctrine_extensions.yaml`.

```
5 lines | config/packages/stof_doctrine_extensions.yaml
1 # Read the documentation: https://symfony.com/doc/current/bundles/StofDoctrineExtensionsBundle/index.html
2 # See the official DoctrineExtensions documentation for more details: https://github.com/Atlantic18/DoctrineExtensions/tree
3 stof_doctrine_extensions:
4     default_locale: en_US
```

Ok... nothing too interesting yet.

Activating Sluggable in Config

As we saw, this bundle comes with a *bunch* of special features for entities. And each time you want to use a feature, you need to enable it in this config file. The first behavior we want is sluggable. To enable it add `orm:` - because we're using the Doctrine ORM:

```
8 lines | config/packages/stof_doctrine_extensions.yaml
... lines 1 - 2
3 stof_doctrine_extensions:
4   default_locale: en_US
5   orm:
... lines 6 - 8
```

and then `default:`, because we want to enable this on our *default* entity manager. That's... really not important except in edge cases where you have multiple database connections.

```
8 lines | config/packages/stof_doctrine_extensions.yaml
... lines 1 - 2
3 stof_doctrine_extensions:
4   default_locale: en_US
5   orm:
6     default:
... lines 7 - 8
```

Then, `sluggable: true`.

```
8 lines | config/packages/stof_doctrine_extensions.yaml
... lines 1 - 2
3 stof_doctrine_extensions:
4   default_locale: en_US
5   orm:
6     default:
7       sluggable: true
```

That's it! Well... sort of. This won't make any *real* difference in our app yet. But, internally, the sluggable feature *is* now active.

Before we start using it, in `QuestionFactory`, remove the code that sets the slug. I'll delete this logic, but keep an example function for later.

```
54 lines | src/Factory/QuestionFactory.php
... lines 1 - 20
21 final class QuestionFactory extends ModelFactory
22 {
... lines 23 - 40
41 protected function initialize(): self
42 {
43     // see https://github.com/zenstruck/foundry#initialization
44     return $this
45         //->afterInstantiate(function(Question $question) { });
46     ;
47 }
... lines 48 - 52
53 }
```

Now, temporarily, if we reload our fixtures with:

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

Yep! A huge error because `slug` is not being set.

[The @Gedmo\Slug Annotation](#)

So how do we tell the Doctrine extensions library that we want the `slug` property to be set automatically? The library works via annotations. In the `Question` entity, above the `slug` property, add `@Gedmo\Slug()` - making sure to autocomplete this so that PhpStorm adds the `use` statement for this annotation.

The `@Gedmo\Slug` annotation has one required option called `fields={}`. Set it to `name`.

```
133 lines | src/Entity/Question.php
... lines 1 - 6
7 use Gedmo\Mapping\Annotation as Gedmo;
... lines 8 - 11
12 class Question
13 {
... lines 14 - 25
26 /**
27  * @ORM\Column(type="string", length=100, unique=true)
28  * @Gedmo\Slug(fields={"name"})
29  */
30 private $slug;
... lines 31 - 131
132 }
```

Done! The `slug` will now be automatically set right before saving to a URL-safe version of the `name` property.

Back at the terminal, try the fixtures now:

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

No errors! And on the homepage... yes! The slug looks perfect. We now *never* need to worry about setting the slug manually.

[Doctrine's Event System](#)

Internally, this magic works by leveraging Doctrine's *event* or "hook" system. The event system makes it possible to run custom code at almost *any* point during the "lifecycle" of an entity. Basically, you can run custom code right before or after an entity is inserted or updated, right after an entity is queried for or other times. You do this by creating an event subscriber or entity listener. We *do* have an example of an entity listener in our [API Platform tutorial](#) if you're interested.

Next, let's add two more handy fields to our entity: `createdAt` and `updatedAt`. The trick will be to have something automatically set `createdAt` when the entity is first inserted and `updatedAt` whenever it's updated. Thanks to Doctrine extensions, you're going to love how easy this is.

Chapter 23: Timestampable & Failed Migrations

Ok team: I've got one more mission for us: to add `createdAt` and `updatedAt` fields to our `Question` entity *and* make sure that these are automatically set whenever we create or update that entity. This functionality is called timestampable, and Doctrine Extensions *totally* has a feature for it.

Activating Timestampable

Start by activating it in the config file: `stof_doctrine_extensions.yaml` . Add `timestampable: true` .

```
9 lines | config/packages/stof_doctrine_extensions.yaml
... lines 1 - 2
3 stof_doctrine_extensions:
4   default_locale: en_US
5   orm:
6     default:
7       sluggable: true
8       timestampable: true
```

Back at the browser, click into the Doctrine Extensions docs and find the Timestampable page. Scroll down to the example... Ah, this works a lot like Sluggable: add `createdAt` and `updatedAt` fields, then put an annotation above each to tell the library to set them automatically.

The TimestampableEntity Trait

Easy! But oh, this library makes it even easier! It has a trait that holds the fields *and* annotations! Check it out: at the top of `Question` , add `use TimestampableEntity` .

```
136 lines | src/Entity/Question.php
... lines 1 - 7
8 use Gedmo\Timestampable\Traits\TimestampableEntity;
... lines 9 - 12
13 class Question
14 {
15     use TimestampableEntity;
... lines 16 - 134
135 }
```

That's it. Hold command or control and click to open that trait. How beautiful is this? It holds the two properties with the `ORM` annotations *and* the `Timestampable` annotations. It even has getter and setter methods. It's everything we need.

But since this *does* mean that we just added two new fields to our entity, we need a migration! At your terminal run:

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

Then go check it out to make sure it doesn't contain any surprises. Yup! It looks good: it adds the two columns.

```

32 lines | migrations/Version20200709153558.php
... lines 1 - 2
3 declare(strict_types=1);
4
5 namespace DoctrineMigrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
... lines 9 - 12
13 final class Version20200709153558 extends AbstractMigration
14 {
... lines 15 - 19
20 public function up(Schema $schema) : void
21 {
22     // this up() migration is auto-generated, please modify it to your needs
23     $this->addSql('ALTER TABLE question ADD created_at DATETIME NOT NULL, ADD updated_at DATETIME NOT NULL');
24 }
25
26 public function down(Schema $schema) : void
27 {
28     // this down() migration is auto-generated, please modify it to your needs
29     $this->addSql('ALTER TABLE question DROP created_at, DROP updated_at');
30 }
31 }

```

Back at the terminal, run it with:

```

$ symfony console doctrine:migrations:migrate

```

And... yikes!

Invalid datetime format 0000 for column `created_at` at row one.

When Migrations Fail

The problem is that our database already has rows in the `question` table! And so, when we add a new `datetime` field that does *not* allow null, MySQL... kinda freaks out! How can we fix this?

There are two options depending on your situation. First, if you have *not* deployed your app to production yet, then you can reset your local database and start over. Why? Because when you eventually deploy, you will *not* have any questions in the database yet and so you will *not* have this error when the migration runs. I'll show you the commands to drop a database in a minute.

But if you *have* already deployed to production and your production database *does* have questions in it, then when you deploy, this *will* be a problem. To fix it, we need to be smart.

Let's see... what we need to do is first create the columns but make them *optional* in the database. Then, with a second query, we can set the `created_at` and `updated_at` of all the existing records to right now. And *finally*, once that's done, we can execute another alter table query to make the two columns required. *That* will make this migration safe.

Modifying a Migration

Ok! Let's get to work. *Usually* we don't need to modify a migration by hand, but this is *one* rare case when we *do*. Start by changing both columns to `DEFAULT NULL`.


```

33 lines | migrations/Version20200709153558.php
... lines 1 - 12
13 final class Version20200709153558 extends AbstractMigration
14 {
... lines 15 - 19
20 public function up(Schema $schema) : void
21 {
22     // this up() migration is auto-generated, please modify it to your needs
23     $this->addSql('ALTER TABLE question ADD created_at DATETIME DEFAULT NULL, ADD updated_at DATETIME DEFAULT NULL');
... line 24
25 }
... lines 26 - 31
32 }

```

Next call `$this->addSql()` with:

```
UPDATE question SET created_at = NOW(), updated_at = NOW();
```

```

33 lines | migrations/Version20200709153558.php
... lines 1 - 12
13 final class Version20200709153558 extends AbstractMigration
14 {
... lines 15 - 19
20 public function up(Schema $schema) : void
21 {
22     // this up() migration is auto-generated, please modify it to your needs
23     $this->addSql('ALTER TABLE question ADD created_at DATETIME DEFAULT NULL, ADD updated_at DATETIME DEFAULT NULL');
24     $this->addSql('UPDATE question SET created_at = NOW(), updated_at = NOW();');
25 }
... lines 26 - 31
32 }

```

Let's start here: we'll worry about making the columns required in another migration.

The *big* question now is... should we just run our migrations again? Not so fast. That *might* be safe - and would in this case - but you need to be careful. If a migration has multiple SQL statements and it fails, it's possible that *part* of the migration was executed successfully and part was *not*. This can leave us in a, sort of, invalid migration state.

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

It would *look* like a migration was *not* executed, when in fact, maybe *half* of it actually *was*! Oh, and by the way, if you use something like PostgreSQL, which supports transactional DDL statements, then this is *not* a problem. In that case, if any part of the migration fails, all the changes are rolled back.

[Safely Re-Testing the Migration](#)

Anyways, let's play it extra safe by resetting our database back to its original state and *then* testing the new migration. Start by dropping the database completely by running:

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

Then `doctrine:database:create` to re-create it:

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

Next, I'll temporarily comment out the new trait in `Question`. That will allow us to reload the fixtures using the *old* database structure - the one *before* the migration. I also need to do a little hack and take the `.php` off of the new migration file so that Doctrine won't see it. I'm doing this so that I can easily run all the migrations *except* for this one.

Let's do it:

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

Excellent: we're back to the database structure *before* the new columns. Now load some data:

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

Beautiful. Back in our editor, undo those changes: put the `.php` back on the end of the migration filename. And, in `Question`, re-add the `TimestampableEntity` trait.

Now we can properly test the new version of the migration. Do it with:

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

And this time... yes! It works perfectly. We can even run:

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

to see those beautiful new `created_at` and `updated_at` columns.

[Making the Columns Required](#)

The *final* thing we need to do is create another migration to make the two columns required in the database. And... we can just make Doctrine do this for us:

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

Go check out the new file. Doctrine: you smartie! Doctrine noticed that the columns were *not* required in the database and generated the `ALTER TABLE` statement needed to fix that.

```

32 lines | migrations/Version20200709155920.php
... lines 1 - 2
3 declare(strict_types=1);
4
5 namespace DoctrineMigrations;
6
7 use Doctrine\DBAL\Schema\Schema;
8 use Doctrine\Migrations\AbstractMigration;
... lines 9 - 12
13 final class Version20200709155920 extends AbstractMigration
14 {
... lines 15 - 19
20 public function up(Schema $schema) : void
21 {
22     // this up() migration is auto-generated, please modify it to your needs
23     $this->addSql('ALTER TABLE question CHANGE created_at created_at DATETIME NOT NULL, CHANGE updated_at update_at DATETIME NOT NULL');
24 }
25
26 public function down(Schema $schema) : void
27 {
28     // this down() migration is auto-generated, please modify it to your needs
29     $this->addSql('ALTER TABLE question CHANGE created_at created_at DATETIME DEFAULT NULL, CHANGE updated_at update_at DATETIME DEFAULT NULL');
30 }
31 }

```

Run the migrations one last time:

```

$ symfony console doctrine:migrations:migrate

```

And... got it! These are two *perfectly* safe migrations.

Okay, friends, we did it! We just unlocked *huge* potential in our app thanks to Doctrine. We know how to create entities, update entities, generate migrations, persist data, create dummy fixtures and more! The only big thing that we have *not* talked about yet is doctrine relations. That's an important enough topic that we'll save it for the next tutorial.

Until then start building and, if you have questions, thoughts, or want to show us what you're building - whether that's a Symfony app or an extravagant Lego creation, we're here for you down in the comments.

Alright friends, seeya next time.



