

Go Pro with Doctrine Queries



With <3 from SymfonyCasts

Chapter 1: Doctrine DQL

Hey there friends! And thanks for joining me for to a tutorial that's all about the nerdery around running queries in Doctrine. It sounds simple... and it is for a while. But then you start adding joins, grouping, grabbing only specific data instead of full objects, counts... and... well... it gets interesting! This tutorial is about deep-diving into all that good stuff -including running native SQL queries, the Doctrine Query Language, filtering collections, fixing the "N + 1" problem, and a *ton* more.

Woh, I'm pumped. So let's get rolling!

Project Setup

To INSERT the most query knowledge into your brain I *highly* recommend coding along with me. You can download the course code from this page. After you unzip it, you'll have a `start/` directory with the same code that you see here. There's also a nifty `README.md` file with all the setup instructions. The *final* step will be to spin over to your terminal, move into the project, and run:

```
symfony serve -d
```

to start a built-in web server at <https://127.0.0.1:8000>. I'll cheat, click that, and... say "hello" to our latest initiative -*Fortune Queries*. You see, we have this side business running a multi-national fortune cookie distribution business... and this fancy app helps us track all the fortunes we've *bestowed* onto our customers.

It's exactly 2 pages: these are the categories, and you can click *into* one to see its fortunes... including how many have been printed. This is a Symfony 6.2 project, and at this point, it couldn't be simpler. We have a `Category` entity, a `FortuneCookie` entity, exactly *one* controller and no fancy queries.

Side note: this project uses MySQL... but almost everything we're going to talk about will work on Postgres or anything else.

Creating our First Custom Repository Method

Speaking of that one controller, here on the home page, you can see that we're autowiring `CategoryRepository` and using the *easiest* way to query for something in Doctrine: `findAll()`.

31 lines | [src/Controller/FortuneController.php](#)

```
... lines 1 - 5
6 use App\Repository\CategoryRepository;
... lines 7 - 10
11 class FortuneController extends AbstractController
12 {
13     #[Route('/', name: 'app_homepage')]
14     public function index(CategoryRepository $categoryRepository): Response
15     {
16         $categories = $categoryRepository->findAll();
17
18         return $this->render('fortune/homepage.html.twig', [
19             'categories' => $categories
20         ]);
21     }
... lines 22 - 29
30 }
```

Our first trick will be super simple, but interesting. I want to re-order these categories alphabetically by name. One *simple* way to do this is by changing `findAll()` to `findBy()`. This is normally used to find items WHERE they match a criteria -something like `['name' => 'foo']`.

But... you can also just leave this empty and take advantage of the second argument: an order by array. So we could say something like `['name' => 'DESC']`.

But... when I need a custom query, I like to create custom repository methods to centralize everything. Head over to the `src/Repository/` directory and open up `CategoryRepository.php`. Inside, we can add whatever methods we want. Let's create a new one called `public function findAllOrdered()`. This will return an `array` ... and I'll even advertise that this is an array of `Category` objects.

```
75 lines | src/Repository/CategoryRepository.php

... lines 1 - 4
5  use App\Entity\Category;
... lines 6 - 16
17 class CategoryRepository extends ServiceEntityRepository
18 {
... lines 19 - 23
24  /**
25   * @return Category[]
26   */
27  public function findAllOrdered(): array
28  {
29
30  }
... lines 31 - 73
74 }
```

Before we fill this in, back here... call it: `->findAllOrdered()`.

```
31 lines | src/Controller/FortuneController.php

... lines 1 - 10
11 class FortuneController extends AbstractController
12 {
... line 13
14  public function index(CategoryRepository $categoryRepository): Response
15  {
16      $categories = $categoryRepository->findAllOrdered();
... lines 17 - 20
21  }
... lines 22 - 29
30 }
```

Delightful!

[Hello DQL \(Doctrine Query Language\)](#)

If you've worked with Doctrine before, you're probably expecting me to use the Query Builder. We *will* talk about that in a minute. But I want to start even *simpler*. Doctrine works with a lot of database systems like MySQL, Postgres, MSSQL, and others. Each of these has an SQL language, but they're not all the same. So Doctrine had to invent its *own* SQL-like language called "DQL", or "Doctrine Query Language". It's fun! It looks a *lot* like SQL. The biggest difference is probably that we refer to classes and properties instead of tables and columns.

Let's write a DQL query by hand. Say `$dql` equals `SELECT category FROM App\Entity\Category as category`. We're aliasing the `App\Entity\Category` class to the string `category` in much the same way we might alias a table name to something in SQL. And over here, by just selecting `category`, we're selecting *everything*, which means it will return `Category` objects.

And that's it! To execute this, create a `Query` object with `$query = $this->getEntityManager()->createQuery($dql);`. Then run it with `return $query->getResult()`.

78 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 16
17 class CategoryRepository extends ServiceEntityRepository
18 {
... lines 19 - 26
27 public function findAllOrdered(): array
28 {
29     $dql = 'SELECT category FROM App\Entity\Category as category';
30     $query = $this->getEntityManager()->createQuery($dql);
31
32     return $query->getResult();
33 }
... lines 34 - 76
77 }
```

There's also a `$query->execute()` , and while it doesn't really matter, I prefer `getResult()` .

When we go over and try that... nothing changes! It *is* working! We just used DQL *directly* to make that query!

[Adding the DQL ORDER BY](#)

So... what does it look like to add the `ORDER BY` ? You can probably guess how it starts `ORDER BY` !

The interesting thing is, to order by `name` , we're *not* going to refer to the `name` *column* in the database. Nope, our `Category` entity has a `$name` *property*, and *that's* what we're going to refer to. The column is *probably* also called `name` ... but it *could* be called `unnecessarily_long_column_name` and we would still order by the `name` *property*.

The point is, because we have a `$name` *property*, over here, we can say `ORDER BY category.name` .

Oh, and in SQL, using the alias is *optional* - you can say `ORDER BY name` . But in *DQL*, it's required, so we *must* say `category.name` . Finally, add `DESC` .

79 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 26
27 public function findAllOrdered(): array
28 {
... line 29
30     $query = $this->getEntityManager()->createQuery($dql);
31     dd($query->getSQL());
... lines 32 - 33
34 }
... lines 35 - 79
```

If we reload the page now...it's alphabetical!

[The DQL -> SQL Transformation](#)

When we write DQL, behind the scenes, Doctrine converts that to SQL and then executes it! It looks to see which database system we're using and translates it into the SQL language *for* that system. We can see the SQL with `dd()` (for "dump and die") `$query->getSQL()` .

79 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 26
27 public function findAllOrdered(): array
28 {
... line 29
30     $query = $this->getEntityManager()->createQuery($dql);
31     dd($query->getSQL());
... lines 32 - 33
34 }
... lines 35 - 79
```

And... there it is! That's the *actual* SQL query being executed! It has this ugly `c0_` alias, but it's what we expect: it grabs every

column from that table and returns it. Pretty cool!

By the way, you can also see the query inside our profiler. If we remove that debug and refresh...down here, we can see that we're making *seven* queries. We'll talk about *why* there's seven in a bit. But if we click that little icon...boom! There's the first query! You can also see a pretty version of it, as well as a version you can *run*. If you have any variables inside **WHERE** clauses, the runnable version will fill those in for you.

Next: We normally *don't* write DQL by hand. Instead, we *build* it with the Query Builder. Let's see what that looks like.

Chapter 2: The QueryBuilder

It's really powerful to understand that DQL is *ultimately* what's being used behind the scenes in Doctrine. But most of the time, we're not going to build this DQL string by hand. Nope, we're going to use something called the "QueryBuilder". Ooooh.

Creating the QueryBuilder

Comment out the DQL. Let's *rebuild* this with the QueryBuilder. Start with `$qb` (for "QueryBuilder") = `$this->createQueryBuilder()`. Inside, say `category`.

```
81 lines | src/Repository/CategoryRepository.php
... lines 1 - 17
18 class CategoryRepository extends ServiceEntityRepository
19 {
... lines 20 - 27
28 public function findAllOrdered(): array
29 {
... line 30
31     $qb = $this->createQueryBuilder('category')
... lines 32 - 35
36 }
... lines 37 - 79
80 }
```

Because we're inside `CategoryRepository`, when we say `createQueryBuilder()`, that automatically adds `FROM App\Entity\Category` and aliases it to `category`, since that's what we passed as the argument. This also selects *everything* by default. So... with *just* this, we've already recreated *most* of this query!

To add the next spot, you can *chain* off of this: `->addOrderBy()` with `category.name`. Then I'll use this `Criteria` class (hit "tab" to autocomplete that) followed by `DESC`. Or you could just put the string `'DESC'`: it's the same thing.

```
81 lines | src/Repository/CategoryRepository.php
... lines 1 - 27
28 public function findAllOrdered(): array
29 {
... line 30
31     $qb = $this->createQueryBuilder('category')
32         ->addOrderBy('category.name', Criteria::DESC);
... lines 33 - 35
36 }
... lines 37 - 81
```

Executing the QueryBuilder

QueryBuilder done! To execute it, we still need that `Query` object. Now we can get it with `$qb->getQuery()`. Internally, this *should* generate the *exact* same DQL as before, and I can prove it! Add a `dd()` with `$query` and, instead of saying `->getSQL()`, say `->getDQL()`.

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

```
... lines 1 - 27
28     public function findAllOrdered(): array
29     {
... lines 30 - 32
33         $query = $qb->getQuery();
34         dd($query->getDQL());
... lines 35 - 36
37     }
... lines 38 - 82
```

When we try that... yeah! That is exactly what we wrote before! So, no surprise, if we remove that `dd()` and refresh... we're back to working! It's just that easy.

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

```
... lines 1 - 27
28     public function findAllOrdered(): array
29     {
... line 30
31         $qb = $this->createQueryBuilder('category')
32             ->addOrderBy('category.name', Criteria::DESC);
33         $query = $qb->getQuery();
34
35         return $query->getResult();
36     }
... lines 37 - 81
```

Okay, we have the QueryBuilder basics down. Let's get more complex by adding `andWhere()` and `orWhere()` next.

Chapter 3: andWhere() and orWhere()

Our site has this nifty search box, which...*doesn't* work. If I hit "enter" to search for "lunch", it *does* add `?q=lunch` to the end of the URL... but the results don't change. Let's hook this thing up!

Grabbing the Search Query Parameter

Spin over and find our controller: `FortuneController`. To read the query parameter, we need Symfony's `Request` object. Add a new argument - it doesn't matter if it's first or last - type-hinted with `Request` - the one from Symfony - hit "tab" to add that `use` statement, and say `$request`. We can get the search term down here with `$searchTerm = $request->query->get('q')`.

```
37 lines | src/Controller/FortuneController.php
... lines 1 - 7
8  use Symfony\Component\HttpFoundation\Request;
... lines 9 - 11
12 class FortuneController extends AbstractController
13 {
... line 14
15     public function index(Request $request, CategoryRepository $categoryRepository): Response
16     {
17         $searchTerm = $request->query->get('q');
... lines 18 - 26
27     }
... lines 28 - 35
36 }
```

We're using `q` ... just because that's what I chose in my template...you can see it down here in `templates/base.html.twig`. This is built with a very simple form that includes `<input type="text" , name="q"`. So we're reading the `q` query parameter and setting it on `$searchTerm`.

Below, if we have a `$searchTerm`, set `$categories` to `$categoryRepository->search()` (a method we're about to create) and pass `$searchTerm`. If we *don't* have a `$searchTerm`, reuse the query logic that we had before.

```
37 lines | src/Controller/FortuneController.php
... lines 1 - 14
15     public function index(Request $request, CategoryRepository $categoryRepository): Response
16     {
... line 17
18         if ($searchTerm) {
19             $categories = $categoryRepository->search($searchTerm);
20         } else {
21             $categories = $categoryRepository->findAllOrdered();
22         }
... lines 23 - 26
27     }
... lines 28 - 37
```

Adding a WHERE Clause

Awesome! Let's go create that `search()` method!

Over in our repository, say `public function search()`. This will take a `string $term` argument and return an `array`. Like last time, I'll add some PHPDoc that says this returns an array of `Category[]` objects. Remove the `@param ...` because that doesn't add anything.

89 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 17
18 class CategoryRepository extends ServiceEntityRepository
19 {
... lines 20 - 37
38 /**
39  * @return Category[]
40  */
41 public function search(string $term): array
42 {
43
44 }
... lines 45 - 87
88 }
```

Ok: our query will start like before...though we can get fancier and *return immediately*. Say `$this->createQueryBuilder()` and use the same `category` alias. It's a good idea to always use the same alias for an entity: it'll help us later to *reuse* parts of a query builder.

93 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 40
41 public function search(string $term): array
42 {
43     return $this->createQueryBuilder('category')
... lines 44 - 47
48 }
... lines 49 - 93
```

For the `WHERE` clause, use `->andWhere()`. There *is* also a `where()` method... but I don't think I've ever used it! And... you shouldn't either. Using `andWhere()` is always ok - even if this is the *first* `WHERE` clause... and we don't really need the "and" part. Doctrine is smart enough to figure that out.

[andWhere\(\) vs where\(\)](#)

What's wrong with `->where()`? Well, if you added a `WHERE` clause to your `QueryBuilder` earlier, calling `->where()` would *remove* that and *replace* it with the new stuff...which probably isn't what you want. `->andWhere()` always *adds* to the query.

Inside say `category`, and since I want to search on the `name` property of the `Category` entity, say `category.name =`. This next part is *very* important. Never ever, *ever* add the dynamic part directly to your query string. This opens you up for SQL injection attacks. Yikes. *Instead*, any time you need to put a dynamic part in a query, put a placeholder instead: like `:searchTerm`. The word `searchTerm` could be anything... and you fill it in by saying `->setParameter('searchTerm', $term)`.

93 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 40
41 public function search(string $term): array
42 {
43     return $this->createQueryBuilder('category')
44         ->andWhere('category.name = :searchTerm')
45         ->setParameter('searchTerm', $term)
... lines 46 - 47
48 }
... lines 49 - 93
```

Perfecto! The ending is easy: `->getQuery()` to turn that into a `Query` object and then `->getResult()` to *execute* that query and return the array of `Category` objects.

93 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 40
41 public function search(string $term): array
42 {
43     return $this->createQueryBuilder('category')
44         ->andWhere('category.name = :searchTerm')
45         ->setParameter('searchTerm', $term)
46         ->getQuery()
47         ->getResult();
48 }
... lines 49 - 93
```

Sweet! If we head over and try this...got it!

[Making the Query Fuzzy](#)

But if we take off a few letters and search again...we get *nothing*! Ideally, we want the search to be fuzzy: matching *any* part of the name.

And that's easy to do. Change our `->andWhere()` from `=` to `LIKE` ... and down here, for `searchTerm` ... this looks a bit weird, but add a percent before and after to make it fuzzy on both sides.

93 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 40
41 public function search(string $term): array
42 {
43     return $this->createQueryBuilder('category')
44         ->andWhere('category.name LIKE :searchTerm')
45         ->setParameter('searchTerm', '%'.$term.'%')
... lines 46 - 47
48 }
... lines 49 - 93
```

If we try it now... eureka!

[Being Careful with orWhere](#)

But let's get tougher! Every category has its own icon - like `fa-quote-left` or the one below it has `fa-utensils` . This is *also* a string that's stored in the database!

Could we make our search *also* search on that property? Sure! We just need to add an `OR` to our query.

Down here, you might be tempted to use this nice `->orWhere()` passing `category.` with the name of that property...which... if we look in `Category` real quick... is `iconKey` . So `category.iconKey LIKE :searchTerm` .

And yes, we *could* do that. But don't! I recommend *never* using `orWhere()` . Why? Because... things can get weird. Imagine we had a query like this: `->andWhere('category.name LIKE :searchTerm') , ->orWhere('category.iconKey LIKE :searchTerm')`
`->andWhere('category.active = true')` .

Do you see the problem? What I'm *probably* trying to do is search for categories...but only every match *active* categories. In reality, if the `searchTerm` matches `iconKey` , a `Category` will be returned whether it's active or not. If we wrote this in SQL, we would include parenthesis around the first two parts to make it behave. But when you use `->orWhere()` , that doesn't happen.

So what's the solution? Always use `andWhere()` ... and if you need an `OR` , put it right inside that! Yup, what you pass to `andWhere()` is DQL, so we can say `OR category.iconKey LIKE :searchTerm` .

93 lines | [src/Repository/CategoryRepository.php](#)

```
... lines 1 - 40
41     public function search(string $term): array
42     {
43         return $this->createQueryBuilder('category')
44             ->andWhere('category.name LIKE :searchTerm OR category.iconKey LIKE :searchTerm')
... lines 45 - 47
48     }
... lines 49 - 93
```

That's it! In the final SQL, Doctrine will put parentheses around this **WHERE** .

Let's try it! Spin over and try searching for "utensils". I'll type part of the word and...got it! We're matching on the **iconKey** !

Oh, and to keep this consistent with the normal homepage, let's include **->addOrderBy('category.name', 'DESC')** .

94 lines | [src/Repository/CategoryRepository.php](#)

```
... lines 1 - 40
41     public function search(string $term): array
42     {
43         return $this->createQueryBuilder('category')
... lines 44 - 45
46             ->addOrderBy('category.name', Criteria::DESC)
... lines 47 - 48
49     }
... lines 50 - 94
```

Now, if we go to the homepage and just type the letter "p" in the search bar, yupIt's sorting alphabetically.

And if you have any doubts about your query, you can always head into the Doctrine profiler to see the formatted version. That's exactly what we expected.

Next: Let's extend our query, so we can search on the *fortune cookies* that are *inside* each category. To do that, we'll need a **JOIN** .

Chapter 4: JOINS

We've got this cool `->andWhere()` method that searches on the `name` or `iconKey` properties of the `Category` entity. But could we *also* search on the fortune cookie data *inside* each category? Sure!

Let's see how that relation is set up. In `Category`, we have a `OneToMany` relationship on a property called `$fortuneCookies` over to the `FortuneCookie` entity.

Thinking about JOINS in Doctrine

If we think about the problem from a database perspective, in order to update our `WHERE` clause to include `WHERE fortune_cookie.fortune = :searchTerm`, we first need to `JOIN` to the `fortune_cookie` table.

And that *is* what we're going to do in Doctrine...except with a twist. Instead of thinking about joining across *tables*, we're going to think about joining across *entity classes*. This might feel weird at first, but it's super cool. In this case, we want to `JOIN` across this `fortuneCookies` property over to the `FortuneCookie` entity.

Using `leftJoin()`

Let's do it! Back over in `CategoryRepository` ... we can add the join anywhere in the query. Unlike SQL, the QueryBuilder doesn't care what order you do things. Add `->leftJoin()` because we're joining from one category to *many* fortune cookies. Pass this `category.fortuneCookies` then `fortuneCookie`, which will be the *alias* for the joined entity.

```
95 lines | src/Repository/CategoryRepository.php
... lines 1 - 17
18 class CategoryRepository extends ServiceEntityRepository
19 {
... lines 20 - 40
41 public function search(string $term): array
42 {
43     return $this->createQueryBuilder('category')
44         ->leftJoin('category.fortuneCookies', 'fortuneCookie')
... lines 45 - 49
50 }
... lines 51 - 93
94 }
```

When we say `category.fortuneCookies`, we're referring to the `fortuneCookies` *property*. The *cool* thing is that... this is all we need! We don't need to tell Doctrine which entity or table we're joining to...and we don't need the `ON fortune_cookie.category_id = category.id` that we would normally see in SQL. We don't need *any* of this because Doctrine already has that info on the `OneToMany` mapping. We just say "join across this property" and it does the rest!

One thing to keep in mind, which we'll talk more about in a minute, is that by joining over to something, we're not *selecting* more data. We're just making the properties on `FortuneCookie` *available* inside our query. This means we can make the `->andWhere()` *even longer*. Add `OR fortuneCookie` (using the new alias from the join) `.fortune` (because `fortune` is the name of the property on `FortuneCookie` that stores the text) `LIKE :searchTerm`.

95 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 40
41     public function search(string $term): array
42     {
43         return $this->createQueryBuilder('category')
... line 44
45         ->andWhere('category.name LIKE :searchTerm OR category.iconKey LIKE :searchTerm OR fortuneCookie.fortune LIKE :searchTerm')
... lines 46 - 49
50     }
... lines 51 - 95
```

Done! Head back to the site. One of my fortunes has the word "conclusion". Spin over to the homepage, search for "conclusion" and... got it! It looks like we have at least one match in our "Proverbs" category. Missing accomplished!

But if you click on the database icon of the web debug toolbar... this page has *two* queries. The first is for the category - it has `FROM category` and includes the `LEFT JOIN` we just added. The second is `FROM fortune_cookie`.

And if we go to the homepage without searching, there are *seven* queries in total: one to fetch all the categories... and then an *additional* 6 to find the fortune cookies for each of the six categories. This is called the N+1 query problem. Let's talk about it next and fix it with joins.

Chapter 5: JOINS and addSelect Reduce Queries

When we're on the homepage, we see *seven* queries. We have one to get all the categories...then additional queries to get all the fortune cookies *for* each category. We can see this in the profiler. This is the main query `FROM category ...` then each of these down here is selecting fortune cookie data for a specific category: 3, 4, 2, 6, and so on.

Lazy-Loading Relationships

If you've used Doctrine, you probably recognize what's happening. Doctrine loads its relationships *lazily*. Let's follow the logic. In `FortuneController`, we start by querying for an array of `$categories`.

```
37 lines | src/Controller/FortuneController.php
... lines 1 - 11
12 class FortuneController extends AbstractController
13 {
... line 14
15 public function index(Request $request, CategoryRepository $categoryRepository): Response
16 {
17     $searchTerm = $request->query->get('q');
18     if ($searchTerm) {
19         $categories = $categoryRepository->search($searchTerm);
20     } else {
21         $categories = $categoryRepository->findAllOrdered();
22     }
23
24     return $this->render('fortune/homepage.html.twig',[
25         'categories' => $categories
26     ]);
27 }
... lines 28 - 35
36 }
```

In that query, if we look at it, it's *only* selecting *category* data: *not* fortune cookie data. But if we go into the template - `templates/fortune/homepage.html.twig` - we loop over the categories and eventually call `category.fortuneCookies|length`.

```
17 lines | templates/fortune/homepage.html.twig
... lines 1 - 7
8 {% for category in categories %}
9     <a class="bg-orange-400 hover:bg-orange-500 text-white text-center rounded-full p-4" href="{{ path('app_category_show', {'id': category.id}) }}">
10         <span class="fa {{ category.iconKey }}"></span> <span class="font-bold text-lg">{{ category.name }}</span> ({{ category.fortuneCookies|length }})
11     </a>
... lines 12 - 13
14 {% endfor %}
... lines 15 - 17
```

The N+1 Problem

In PHP land, we're calling the `getFortuneCookies()` method on `Category`. But until now, Doctrine has not *yet* queried for the `FortuneCookie` data for this `Category`. However, as *soon* as we access the `$this->fortuneCookies` property, it magically makes that query, basically saying:

Give me all the `FortuneCookie` data for this category

Which... it then *sets* onto the property and returns back to us. So it's at *this moment* inside of Twig when that second, third, fourth, fifth, sixth, and seventh query is executed.

This is called the "N+1 Problem", where you have "N" number of queries for the related items on your page "plus one" for the main query. In our case, it's 1 main query for the categories plus 6 more queries to get the fortune cookie data for those 6 categories.

This isn't *necessarily* a problem. It *might* hurt performance on your page... or be no big deal. But if it *is* slowing things down, we *can* fix it with a `JOIN`. After all, when we query for the categories, we're *already* joining over to the fortune cookie table. So... if we just grab the fortune cookie data in the first query, couldn't we build this whole page *with* that *one* query? The answer is... totally!

Selecting the Joined Fields

To see this in action, search for something first. I'm doing this because it will trigger the `search()` method in our repository, which already has the `JOIN`. Over here, since we have five results, it made *six* queries.

Okay, we're already *joining* over to `fortuneCookie`. So how can we select its data? It's delightfully simple. And again, order doesn't matter: `->addSelect('fortuneCookie')`.

```
96 lines | src/Repository/CategoryRepository.php
... lines 1 - 17
18 class CategoryRepository extends ServiceEntityRepository
19 {
... lines 20 - 40
41 public function search(string $term): array
42 {
43     return $this->createQueryBuilder('category')
44         ->addSelect('fortuneCookie')
45         ->leftJoin('category.fortuneCookies', 'fortuneCookie')
... lines 46 - 50
51 }
... lines 52 - 94
95 }
```

That's it! Try this thing! The queries went down to one and the page still works!

You might notice that the fortune cookie count for each category *also* change. Before, Doctrine executed separate queries to count the related fortune cookies without considering our search term. But after adding `addSelect('fortuneCookie')`, the ORM uses *that* data to count instead of making new queries... which includes our search term!

If you open the profiler... and view the formatted query... yes! It's joining over to `fortune_cookie` and *grabbing* the `fortune_cookie` data at the same time. The "N+1" problem is *solved*!

Where does the Join Data Hide?

But I want to point out one key thing. Because we're inside of `CategoryRepository`, when we call `$this->createQueryBuilder('category')`, that automatically adds a `->select('category')` to the query. We know that.

However *now* we're selecting all of the `category` *and* `fortuneCookie` data. But... our page still works... which must mean that even though we're selecting data from *two* tables, our query is still *returning* the same thing it did before an array of `Category` objects. It's not returning some mixture of `category` and `fortuneCookie` data.

This point can be a bit confusing, so let me break it down. When we call `createQueryBuilder()`, that actually adds 2 things to our query: `FROM App\Entity\Category as category` and `SELECT category`. Thanks to the `FROM`, `Category` is our "root entity" and, unless we start doing something more complex, Doctrine will try to return `Category` objects. When we `->addSelect('fortuneCookie')`, instead of returning a mixture of categories and fortune cookies, Doctrine basically grabs the `fortuneCookie` data and stores it for later. Then, if we ever call `$category->getFortuneCookies()`, it realizes that it *already* has that data, so instead of making a query, it uses it.

The really important thing is that when we use `->addSelect()` to grab the data from a `JOIN`, it does *not* change what our method returns. Though later, we *will* see times when using `select()` or `addSelect()` *does* change what our query returns.

Ok, so we just used a JOIN to reduce our queries from 7 to 1. However, because we're only *counting* the number of fortune cookies for each category, there *is* another solution. Let's talk about EXTRA_LAZY relationships next.

Chapter 6: EXTRA_LAZY Relationships

Click back to the homepage with no search query. We still have seven queries because we're still using our very simple `findAllOrdered()` method... which doesn't have the `JOIN`. So... we should add the `JOIN` here too, right? Yep! Well... *probably*. But I want to show you an *alternative* solution.

Our homepage is unique because we don't *really* need all the `FortuneCookie` data for each `Category` ... the only thing we need is the `COUNT`.

Check out the template: we're not looping over `category.fortuneCookies` and rendering the actual `FortuneCookie` data. Nope, we're simply *counting* them. If you think about it, having a giant query that grabs *all* of the `FortuneCookie` data.... just to count them... isn't the *greatest* thing for efficiency.

```
17 lines | templates/fortune/homepage.html.twig
... lines 1 - 7
8     {% for category in categories %}
9         <a class="bg-orange-400 hover:bg-orange-500 text-white text-center rounded-full p-4" href="{{ path('app_category_show', {'id': category.id}) }}">
10             <span class="fa {{ category.iconKey }}"></span> <span class="font-bold text-lg">{{ category.name }}</span> <span class="font-size-0.8em">{{ category.fortuneCookies|count }}</span>
11         </a>
... lines 12 - 13
14     {% endfor %}
... lines 15 - 17
```

[Adding fetch: EXTRA_LAZY](#)

If you find yourself in this situation, you can tell Doctrine to be *clever* with how it loads the relation. Go into the `Category` entity and find the `OneToMany` relationship for `$fortuneCookies`. At the end, add `fetch: set to EXTRA_LAZY`.

```
91 lines | src/Entity/Category.php
... lines 1 - 10
11 class Category
12 {
... lines 13 - 23
24     #[ORM\OneToMany(mappedBy: 'category', targetEntity: FortuneCookie::class, fetch: 'EXTRA_LAZY')]
25     private Collection $fortuneCookies;
... lines 26 - 89
90 }
```

Let's go see what that does. When you refresh, watch the query count. It *stays* at seven! But if we open up the profiler, the queries *themselves* have changed. The first one is the same: it queries from `category`. But check out the others! We have `SELECT COUNT(*) FROM fortune_cookie` over and over! So we *do* have seven queries, but now each is only selecting the `COUNT`!

When you have `fetch: 'EXTRA_LAZY'` and you simply *count* a collection relation, Doctrine is smart enough to select *just* the `COUNT` instead of querying for all the data. If we *were* to loop over this collection and start printing out `FortuneCookie` data, then it *would* still make a *full* query for the data. But if all we need is to count them, then `fetch: 'EXTRA_LAZY'` is a great solution.

[Custom Query on the Category Show Page](#)

Ok: click into one of the categories. The profiler says that we have two queries. This is a, sort of, "miniature" N+1 problem. The first query selects a single `Category` ... and the second selects all the fortune cookies *for* this one category. Let's flex our `JOIN` skills to get this down to *one* query.

Open up `FortuneController` and find the `showCategory()` action. By type-hinting `Category` on this argument, we're telling *Symfony* to query for the `Category` for us, by using the `{id}`. Normally, I love this! *However*, in this case, because we want to add a `JOIN` from `Category` to `fortuneCookies`, we need to take *control* of that query.

37 lines | src/Controller/FortuneController.php

```
... lines 1 - 11
12 class FortuneController extends AbstractController
13 {
... lines 14 - 29
30 public function showCategory(Category $category): Response
31 {
32     return $this->render('fortune/showCategory.html.twig',[
33         'category' => $category
34     ]);
35 }
36 }
```

Change this so that Symfony passes us the `int $id` directly. Then, autowire `CategoryRepository $categoryRepository`.

39 lines | src/Controller/FortuneController.php

```
... lines 1 - 11
12 class FortuneController extends AbstractController
13 {
... lines 14 - 29
30 public function showCategory(int $id, CategoryRepository $categoryRepository): Response
31 {
... lines 32 - 36
37 }
38 }
```

Below, do the query manually with `$category = $categoryRepository-> ...` calling a new method: `findWithFortunesJoin($id)`. Before we create that, we also need to add `if (!$category)`, then `throw $this->createNotFoundException()`. You can give that a message if you want.

Ok, copy the method name, hop over to `CategoryRepository` and say `public function findWithFortunesJoin(int $id)`, which will return a `Category` if one is found, else `null`. I'll fix that typo in a minute.

42 lines | src/Controller/FortuneController.php

```
... lines 1 - 29
30 public function showCategory(int $id, CategoryRepository $categoryRepository): Response
31 {
32     $category = $categoryRepository->findWithFortunesJoin($id);
33     if (!$category) {
34         throw $this->createNotFoundException('Category not found!');
35     }
... lines 36 - 39
40 }
... lines 41 - 42
```

The query starts like the other....and we *could* steal some code... but since we're practicing, let's write it by hand.

`return $this->createQueryBuilder()` and pass our normal `category` alias. Then `->andWhere('category.id = :id')` - I'll fix that typo in a minutes as well - filling in the wildcard with `->setParameter() $id`, `$id` ... ideally spelled correctly. Then `->getQuery()`.

107 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 17
18 class CategoryRepository extends ServiceEntityRepository
19 {
... lines 20 - 52
53 public function findWithFortunesJoin(int $id): ?Category
54 {
55     return $this->createQueryBuilder('category')
... lines 56 - 57
58         ->andWhere('category.id = :id')
59         ->setParameter('id', $id)
60         ->getQuery()
... line 61
62     }
... lines 63 - 105
106 }
```

Until now, we've been fetching *multiple* rows... and so we've used `->getResult()` . But this time, we want either the *one* result or null if it can't be found. To do *that*, use `->getOneOrNullResult()` .

107 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 52
53 public function findWithFortunesJoin(int $id): ?Category
54 {
55     return $this->createQueryBuilder('category')
... lines 56 - 60
61         ->getOneOrNullResult();
62     }
... lines 63 - 107
```

And that's it! That *should* get things working. I'll do a little sanity check over here, and...*oh*... it would probably help if I typed things correctly. But this is cool! It recognized that it didn't know what that alias was and gave us a clear error. And now... it *works*, and we still have two queries.

Adding a Join

Time for the **JOIN** ! We're going from one `Category` to many fortune cookies, so let's say `->leftJoin()` on `category`. and the property name, which is `fortuneCookies` . Once again, the order doesn't matter, but above I'll say `->addSelect('fortuneCookie')` . Oh, and I also need to add `fortuneCookie` as a second argument inside the `->leftJoin()` : that's the alias.

107 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 52
53 public function findWithFortunesJoin(int $id): ?Category
54 {
55     return $this->createQueryBuilder('category')
56         ->addSelect('fortuneCookie')
57         ->leftJoin('category.fortuneCookies', 'fortuneCookie')
... lines 58 - 61
62     }
... lines 63 - 107
```

So we're aliasing that joined entity to `fortuneCookie` then *selecting* `fortuneCookie` . Now, we *should* see this query number go from two to one. And... it did!

Here's the takeaway: while there's no need to over-optimize, if you have the N+1 problem you can solve it by JOINing to the related table *and* selecting its data.

Ok, until now, Doctrine has returned a collection of `Category` objects or a single `Category` object. That's cool, but what if, instead of entire objects, we just need some data - like a few columns, a `COUNT` , or a `SUM` ? Let's dig into that next.

Chapter 7: SELECT the SUM (or COUNT)

New goal team! Look over at the `FortuneCookie` entity. One of its properties is `$numberPrinted`, which is the number of times that we've ever printed that fortune. On the category page, up here, I want to print the *total* number printed for *all* fortunes in this category.

We *could* solve this by looping over `$category->getFortuneCookies()` ... calling `->getNumberPrinted()` and adding it to some `$count` variable. That would work as long as we always have a small number of fortune cookies. But the cookie business is *booming*... and soon we'll have *hundreds* of cookies in each category. It would be a *huge* slowdown if we queried for 500 fortune cookies *just* to calculate the sum. Actually, we'd probably run out of memory first!

Surely there's a better way, right? You bet! Do all that work in the *database* with a *sum query*.

Overriding the Selected Fields

Let's think: the data we're querying for will ultimately come from the `FortuneCookie` entity... so open up `FortuneCookieRepository` so we can add a new method there. How about: `public function countNumberPrintedForCategory(Category $category): int`.

```
73 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 17
18 class FortuneCookieRepository extends ServiceEntityRepository
19 {
... lines 20 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27
28 }
... lines 29 - 71
72 }
```

The query starts pretty much like they all do. Say `$result = $this->createQueryBuilder('fortuneCookie')`. By the way, the alias can be anything. Personally, I try to make them long enough to be unique in my project..but short enough to not be annoying. More importantly, as soon as you choose an alias for an entity, stick with it.

```
80 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
... lines 28 - 34
35 }
... lines 36 - 80
```

Ok, we know that when we create a QueryBuilder, it will select *all* the data from `FortuneCookie`. But in this case, we *don't* want that! So, below, say `->select()` to override that.

Earlier, in `CategoryRepository`, we used `->addSelect()`, which basically says:

Take whatever we're selecting and *also* select this other stuff.

But this time, I'm purposely using `->select()` so that it *overrides* that and *only* selects what we put next. Inside, write DQL: `SUM()` a function that you're probably familiar with followed by `fortuneCookie`. and the name of the property we want to use - `numberPrinted`. And you don't *have* to do this, but I'm going to add `AS fortunesPrinted`, which will *name* that result when it's returned. We'll see that in a minute.

80 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
28         ->select('SUM(fortuneCookie.numberPrinted) AS fortunesPrinted')
... lines 29 - 34
35 }
... lines 36 - 80
```

[andWhere\(\) with an Entire Entity](#)

Ok, that takes care of the `->select()` . Now we need an `->andWhere()` with `fortuneCookie.category = :category` ... calling `->setParameter()` to fill in the dynamic `category` with the `$category` object.

80 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
... line 28
29     ->andWhere('fortuneCookie.category = :category')
30     ->setParameter('category', $category)
... lines 31 - 34
35 }
... lines 36 - 80
```

This is interesting too! In SQL, we would normally say something like `WHERE fortuneCookie.categoryId =` and then the *integer* ID. But in Doctrine, we don't think about the tables or columns: we focus on the entities. And, there *is* no `categoryId` property on `FortuneCookie` . Instead, when we say `fortuneCookie.category` we're referencing the `$category` property in `FortuneCookie` . And instead of passing *just* the integer ID, we pass the entire `Category` object. It *actually is* possible to pass the ID, but most of the time you'll pass the entire object like this.

Okay, let's finish this! Convert this to a query with `->getQuery()` . Below, if you think about it, we really only want *one* row of results. So let's say `->getOneOrNullResult()` . Finally, `return $result` .

80 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
28         ->select('SUM(fortuneCookie.numberPrinted) AS fortunesPrinted')
29         ->andWhere('fortuneCookie.category = :category')
30         ->setParameter('category', $category)
31         ->getQuery()
32         ->getOneOrNullResult();
33
34     return $result;
35 }
... lines 36 - 80
```

Until now, all of our queries have returned *objects*. Since we're selecting just *one* thing... does that finally change? Let's find out! Add `dd($result)` and then head to `FortuneController` to use this. For the show page controller, add an argument `FortuneCookieRepository $fortuneCookieRepository` . Then below, say `$fortunesPrinted` equals `$fortuneCookieRepository->countNumberPrintedForCategory()` passing `$category` .

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

```
... lines 1 - 24
25     public function countNumberPrintedForCategory(Category $category): int
26     {
    ... lines 27 - 32
33         dd($result);
    ... lines 34 - 35
36     }
    ... lines 37 - 81
```

Beautiful! Take that `$fortunesPrinted` variable and pass it into Twig as `fortunesPrinted`.

45 lines | [src/Controller/FortuneController.php](#)

```
... lines 1 - 6
7     use App\Repository\FortuneCookieRepository;
    ... lines 8 - 12
13     class FortuneController extends AbstractController
    ... lines 14 - 30
31     public function showCategory(int $id, CategoryRepository $categoryRepository, FortuneCookieRepository $fortuneCookieRepository): Response
32     {
    ... lines 33 - 36
37         $fortunesPrinted = $fortuneCookieRepository->countNumberPrintedForCategory($category);
    ... line 38
39         return $this->render('fortune/showCategory.html.twig',[
    ... line 40
41             'fortunesPrinted' => $fortunesPrinted,
42         ]);
43     }
44 }
```

Finally, find the template - `showCategory.html.twig` - and... there's a table header that says "Print History". Add some parentheses with `{{ fortunesPrinted }}`. Add `|number_format` to make this prettier than the word `total`.

40 lines | [templates/fortune/showCategory.html.twig](#)

```
... lines 1 - 8
9     <table class="table-auto border mb-6">
10         <thead class="bg-slate-500 text-white">
    ... lines 11 - 14
15             <th class="border p-4">
16                 Print History ({{ fortunesPrinted|number_format }} total)
17             </th>
    ... line 18
19         </thead>
    ... lines 20 - 31
32     </table>
    ... lines 33 - 40
```

Awesome! Since we have that `dd()`, let's refresh and... look at that! We get an array back with 1 key called `fortunesPrinted`! Yup, as soon as we start selecting specific data, we *just* get back that specific data. It's exactly like you'd expect with a normal SQL query.

If we had said `->select('fortuneCookie')` (which is redundant because that's what `createQueryBuilder()` already does), that would have given us a `FortuneCookie` object. But as soon as we're selecting one specific thing, it gets rid of the object and returns an associative array.

[Using getSingleScalarResult\(\)](#)

Because our method should return an `int`, we *could* complete this by saying `return $result['fortunesPrinted']`. But if you have a situation where you're selecting one row of data... and only one *column* of data, there's a shortcut to *get* that one column:

`->getSingleScalarResult()`. We can return *that* directly.

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

```
... lines 1 - 17
18 class FortuneCookieRepository extends ServiceEntityRepository
19 {
... lines 20 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
... lines 28 - 31
32     ->getSingleScalarResult();
... lines 33 - 35
36 }
... lines 37 - 79
80 }
```

I'll keep the `dd()` so we can see it. And... awesome! We get *just* the number! Well, technically it's a string. If you want to be strict, you can add `(int)`. And now... got it! We have a nicely formatted total number!

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

```
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
... lines 27 - 34
35     return (int) $result;
36 }
... lines 37 - 81
```

Next: Let's select even *more* data and see how that complicates things.

Chapter 8: Selecting Specific Fields

Let's add more stuff to this page! How about the *average* number of fortune cookies printed for this category? To do that, head back to our query: it lives in `countNumberPrintedForCategory()`.

```
80 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 17
18 class FortuneCookieRepository extends ServiceEntityRepository
19 {
... lines 20 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
28         ->select('SUM(fortuneCookie.numberPrinted) AS fortunesPrinted')
29         ->andWhere('fortuneCookie.category = :category')
30         ->setParameter('category', $category)
31         ->getQuery()
32         ->getSingleScalarResult();
33
34     return (int) $result;
35 }
... lines 36 - 78
79 }
```

[SELECTing the AVG](#)

To get the average, we *could* add a comma then use the `AVG()` function. Or we can use `addSelect()` ... which looks a bit better to me. We want the `AVG()` of `fortuneCookie.numberPrinted` aliased to `fortunesAverage`.

This time, I did *not* use the word `AS` ... just to demonstrate that the word `AS` is optional. In fact, the *entire* `fortunesAverage` or `AS fortunesPrinted` part is optional. But by giving each a name, we can control the keys in the final result array, which we'll see in a minute.

```
81 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
... line 28
29     ->addSelect('AVG(fortuneCookie.numberPrinted) fortunesAverage')
... lines 30 - 35
36 }
... lines 37 - 81
```

While we're here, instead of printing out the name from the `$category` object, let's see if we can grab the category name right inside this query. I'll say `->addSelect('category.name')`.

If you see a problem with this, you're right! But let's ignore that and forge ahead blindly! `dd($result)` at the bottom.

83 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
... lines 28 - 29
30     ->addSelect('category.name')
... lines 31 - 34
35     dd($result);
... lines 36 - 37
38 }
... lines 39 - 83
```

Previously, this returned *only* the integer `fortunesPrinted` . But *now*, we're selecting *three* things. So what will it return now?

The answer is... a gigantic error!

'category' is not defined.

Yup - I referenced `category` ... but we never *joined* over to it. Let's add that. We're querying from the `FortuneCookie` entity, and it has a `category` property, which is a `ManyToOne` . So we're joining over to *one* object. Do that with `->innerJoin()` passing `fortuneCookie.category` and giving it the alias `category` .

84 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): int
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
... lines 28 - 30
31     ->innerJoin('fortuneCookie.category', 'category')
... lines 32 - 38
39 }
... lines 40 - 84
```

Returning Multiple Columns of Results

If we go refresh the page now... *this* is the error I was expecting:

The query returned a row containing multiple columns.

This `->getSingleScalarResult()` is perfect when you're returning a single row *and* a single column. As soon as you return *multiple* columns, `->getSingleScalarResult()` won't work. To fix that, change to `->getSingleResult()` .

84 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): array
26 {
27     $result = $this->createQueryBuilder('fortuneCookie')
... lines 28 - 34
35     ->getSingleResult();
... lines 36 - 38
39 }
... lines 40 - 84
```

This basically says:

Give me the one row of data from the database.

Try this again. *That's* what we want! It returns the exact three columns we selected!

And now... we need to change this method a bit. Update the `int` return to an `array` ... and, down here, take off the `(int)` entirely and return `$result` . We can also remove the `dd()` ... and you *could* put the `return` up here if you wanted to.

```
83 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 24
25 public function countNumberPrintedForCategory(Category $category): array
26 {
... lines 27 - 36
37 return $result;
38 }
... lines 39 - 83
```

Updating our Project to use the Results

Our method is good to go! Now let's fix the controller. This `$fortunesPrinted` isn't right anymore. Change it to `$result` instead. Then... read that out below with - `$result['fortunesPrinted']` . Copy that, paste, and send a `fortunesAverage` variable to the template set to the `fortunesAverage` key. Also pass `categoryName` set to `$result['name']` .

```
47 lines | src/Controller/FortuneController.php
... lines 1 - 12
13 class FortuneController extends AbstractController
14 {
... lines 15 - 30
31 public function showCategory(int $id, CategoryRepository $categoryRepository, FortuneCookieRepository $fortuneCookieRepository): Response
32 {
... lines 33 - 38
39 return $this->render('fortune/showCategory.html.twig',[
40     'category' => $category,
41     'fortunesPrinted' => $result['fortunesPrinted'],
42     'fortunesAverage' => $result['fortunesAverage'],
43     'categoryName' => $result['name'],
44 ]);
45 }
46 }
```

Template time! Over in `showCategory.html.twig` , we have access to the *entire* `$category` object... which is how we're printing `category.name` . But *now*, we also have a `categoryName` variable. Replace `category.name` with `categoryName` .

```
40 lines | templates/fortune/showCategory.html.twig
... lines 1 - 2
3 {% block body %}
... lines 4 - 5
6 <h1 class="text-3xl p-5 text-center my-4 font-semibold"><span class="fa {{ category.iconKey }}"></span> {{ categoryName }} Fortunes</h1>
... lines 7 - 38
39 {% endblock %}
```

There's... no *actual* reason to do that - I'm just proving that we *are* able to grab extra data in our new query. Though, if we had *also* selected `iconKey` , then we *could* potentially avoid querying for the `Category` object entirely. However, while that *might* make our page a *tiny* bit faster, it's almost definitely overkill and makes our code more confusing. Using objects is best!

Ok, below, for the "Print History", hit "enter" and add `{{ fortunesAverage|number_format }}` then `average` .

```
... lines 1 - 2
3  {% block body %}
... lines 4 - 9
10    <thead class="bg-slate-500 text-white">
... lines 11 - 14
15        <th class="border p-4">
16            Print History ({{ fortunesPrinted|number_format }} total, {{ fortunesAverage|number_format }} average)
17        </th>
... line 18
19    </thead>
... lines 20 - 38
39 {% endblock %}
```

Awesome. Try this again! If I didn't make any mistakes...got it! Everything *works*! We have two queries: one for the *category* that's joined over to *fortune_cookies* and the one that we just made that grabs the *SUM* , *AVG* , and the *name* *also* with a *JOIN* . Love it!

Getting full entity objects back from Doctrine is the *ideal* situation because... objects are just really nice to work with. But at the end of the day, if you need to query for specific data or columns, you can *totally* do that. And as we just saw, Doctrine will return an associative array.

However, we *can* go one step further and ask Doctrine to *return* this specific data *inside* an object. Let's talk about that *next*.

Chapter 9: SELECTing into a New DTO Object

Having the flexibility to select any data we want is *awesome*. Dealing with the associative array that we get back is... *less* awesome! I like to work with objects whenever possible. Fortunately, Doctrine gives us a simple way to *improve* this situation: we query for the data we want... but tell it to give us an *object*.

[Creating the DTO](#)

First, we need to create a new class that will hold the data from our query! I'll make a new directory called `src/Model/` ... but it could be called anything. Call the class... how about `CategoryFortuneStats` .

The *entire* purpose of this class is to hold the data from this specific query! So add a `public function __construct()` with a few `public` properties for simplicity: `public int $fortunesPrinted` , `public float $fortunesAverage` , and `public string $categoryName` .

```
15 lines | src/Model/CategoryFortuneStats.php
... lines 1 - 4
5 class CategoryFortuneStats
6 {
7     public function __construct(
8         public int $fortunesPrinted,
9         public float $fortunesAverage,
10        public string $categoryName,
11    )
12    {
13    }
14 }
```

Beautiful!

Back in the repository, we actually *don't* need any Doctrine magic to use this new class. We could query for the associative array, then return `new CategoryFortuneStats()` and pass each key into it.

That's a *great* option, dead simple and then this repository method would return an object instead of an array. *But...* Doctrine makes this even easier thanks to a little-known feature.

Add a new `->select()` that will contain *all* of these selects in one. Also add a `sprintf()` : you'll see why in a minute. Inside, check this out! Say `NEW %s()` then pass `CategoryFortuneStats::class` for that placeholder. Basically, we're saying `NEW App\Model\CategoryFortuneStats()` ... I just wanted to avoid typing that long class name.

Inside of `NEW` , grab each of the 3 things that we want to select and paste them, as *if* we're passing them directly as the first, second and third arguments to our new class's constructor.

90 lines | [src/Repository/FortuneCookieRepository.php](#)

```
... lines 1 - 18
19 class FortuneCookieRepository extends ServiceEntityRepository
20 {
... lines 21 - 25
26 public function countNumberPrintedForCategory(Category $category): array
27 {
28     $result = $this->createQueryBuilder('fortuneCookie')
29         ->select(sprintf(
30             'NEW %s(
31                 SUM(fortuneCookie.numberPrinted) AS fortunesPrinted,
32                 AVG(fortuneCookie.numberPrinted) fortunesAverage,
33                 category.name
34             )',
35             CategoryFortuneStats::class
36         ))
... lines 37 - 44
45 }
... lines 46 - 88
89 }
```

Isn't that cool? Let's `dd($result)` so we can see what it looks like!

[No Aliasing with NEW](#)

If we head over and refresh... oh... I get an error: `T_CLOSE_PARENTHESIS, got 'AS'`. When we select data into an object, aliasing is no longer needed... or allowed. And it makes sense: Doctrine will pass whatever this is to the first argument of our constructor, this to the second argument, and this to the third. Since aliases don't make sense anymore... remove them.

90 lines | [src/Repository/FortuneCookieRepository.php](#)

```
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): array
27 {
28     $result = $this->createQueryBuilder('fortuneCookie')
29         ->select(sprintf(
30             'NEW %s(
31                 SUM(fortuneCookie.numberPrinted),
32                 AVG(fortuneCookie.numberPrinted),
33                 category.name
34             )',
35             CategoryFortuneStats::class
36         ))
... lines 37 - 44
45 }
... lines 46 - 90
```

If we check it now... got it! I love it! We have an object with our data inside!

Let's celebrate by cleaning up our method. Instead of an `array`, we're returning a `CategoryFortuneStats`. Also remove the `dd($result)` down here.

89 lines | [src/Repository/FortuneCookieRepository.php](#)

```
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 43
44 }
... lines 45 - 89
```

Back in the controller, to show off how nice this is, change `$result` to... how about `$stats`. Then we can use `$stats->fortunesPrinted`, `$stats->fortunesAverage`, and `$stats->categoryName`.

47 lines | src/Controller/FortuneController.php

```
... lines 1 - 12
13 class FortuneController extends AbstractController
14 {
... lines 15 - 30
31 public function showCategory(int $id, CategoryRepository $categoryRepository, FortuneCookieRepository $fortuneCookieRepository): Response
32 {
... lines 33 - 36
37     $stats = $fortuneCookieRepository->countNumberPrintedForCategory($category);
... line 38
39     return $this->render('fortune/showCategory.html.twig', [
40         'category' => $category,
41         'fortunesPrinted' => $stats->fortunesPrinted,
42         'fortunesAverage' => $stats->fortunesAverage,
43         'categoryName' => $stats->categoryName,
44     ]);
45 }
46 }
```

Now that we've tidied up a bit, let's check to see if this still works. And... it *does*.

Next: Sometimes queries are so complex... the best option is just to write the darn thing in raw, native SQL. Let's talk about how to do that.

Chapter 10: Raw SQL Queries

The QueryBuilder is fun to use *and* powerful. But if you're writing a *super* complex query... it might be tough to figure out how to transform it into the QueryBuilder format. If you find yourself in this situation, you can always resort to just...writing *raw* SQL! I wouldn't make this my *first* choice - but there's no *huge* benefit to spending hours adapting a well-written SQL query into a query builder.

The Connection Object

Let's see how raw SQL queries work. To start, comment out the `->createQueryBuilder()` query. Then, we need to fetch the low-level Doctrine `Connection` object. We can get that with `$conn = $this->getEntityManager()->getConnection()`. Toss `dd($conn)` onto the end so we can see it.

```
92 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 18
19 class FortuneCookieRepository extends ServiceEntityRepository
20 {
... lines 21 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
28 // $result = $this->createQueryBuilder('fortuneCookie')
... lines 29 - 40
41 // ->getSingleResult();
... line 42
43 $conn = $this->getEntityManager()->getConnection();
44 dd($conn);
... lines 45 - 46
47 }
... lines 48 - 90
91 }
```

Head over, refresh and... awesome! We get a `Doctrine\DBAL\Connection` object.

The Doctrine library is actually *two* main parts. First there's a lower-level part called "DBAL", which stands for "Database Abstraction Library". This acts as a wrapper around PHP's native PDO and adds some features on top of it.

The *second* part of Doctrine is what we've been dealing with so far: it's the higher-level part called the "ORM" or "Object Relational Mapper". That's when you query by selecting classes and properties...and get back objects.

For this raw SQL query, we're going to deal with the lower-level `Connection` object directly.

Writing & Executing the Query

Say `$sql = 'SELECT * FROM fortune_cookie'`. That's as *boring* as SQL queries can get. I used `fortune_cookie` for the table name because I know that, by default, Doctrine *underscores* my entities to make table names.

```
92 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 43
44 $sql = 'SELECT * FROM fortune_cookie';
... lines 45 - 46
47 }
... lines 48 - 92
```

Now that we have the query string, we need to create a `Statement` with `$stmt = $conn->prepare()` and pass `$sql`.

93 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 44
45 $stmt = $conn->prepare($sql);
... line 46
47 return $result;
... lines 48 - 93
```

This creates a `Statement` object... which is kind of like the `Query` object we would create with the `QueryBuilder` by saying `->getQuery()` at the end. It's... just an object that we'll use to execute this. Do that with `$result = $stmt->executeQuery()` .

94 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 44
45 $stmt = $conn->prepare($sql);
46 $result = $stmt->executeQuery();
... lines 47 - 48
49 }
... lines 50 - 94
```

Finally, to get the actual *data* off of the result, say `dd($result->)` ... and there are a number of methods to choose from. Use `fetchAllAssociative()` .

95 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 46
47 dd($result->fetchAllAssociative());
... lines 48 - 49
50 }
... lines 51 - 95
```

This will fetch all the rows and give each one to us as an *associative* array.

Watch: head back over and... perfect! We get 20 rows for each of the 20 fortune cookies in the system! This is the raw data coming from the database.

A More Complex Query

Okay, let's rewrite this entire `QueryBuilder` query up here in raw SQL. To save time, I'll paste in the final product: a *long* string... with nothing particularly special. We're selecting `SUM` , `AS fortunesPrinted` , the `AVG` , `category.name` , `FROM fortune_cookie` , and then we do our `INNER JOIN` over to `category` .

95 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 43
44 $sql = 'SELECT SUM(fortune_cookie.number_printed) AS fortunesPrinted, AVG(fortune_cookie.number_printed) fortunesAverage, cat
... lines 45 - 49
50 }
... lines 51 - 95
```

The big difference is that, when we do a `JOIN` with the `QueryBuilder`, we can just join across the relationship...and that's all we need to say. In raw SQL, of course, we need to help it by *specifying* that we're joining over to `category` and describe that we're joining on `category.id = fortune_cookie.category_id` .

The rest is pretty normal... except for `fortune_cookie.category_id = :category` . Even though we're running raw SQL, we're *still not* going to concatenate dynamic stuff directly into our query. That's a *huge* no-no, and, as we know, opens us up to SQL injection attacks. Instead, stick with these nice placeholders like `:category` . To fill that in, down where we execute the query, pass `'category' =>` . But this time, instead of passing the entire `$category` object like we did before, this is raw SQL, so we need to pass `$category->getId()` .

```
97 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 45
46     $result = $stmt->executeQuery([
47         'category' => $category->getId(),
48     ]);
... lines 49 - 51
52 }
... lines 53 - 97
```

Ok! Spin over and check this out. Got it! So writing raw SQL doesn't look as awesome...but if your query is complex enough, don't hesitate to try this.

Using `bindValue()`

By the way, instead of using `executeQuery()` to pass the `category` , we *could*, replace that with `$stmt->bindValue()` to bind `category` to `$category->getId()` . That's going to give us the same results as before, so your call.

```
96 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 45
46     $stmt->bindValue('category', $category->getId());
... lines 47 - 50
51 }
... lines 52 - 96
```

But, hmm, I'm realizing now that the result is an array inside another array. What we *really* want to do is return *only* the associative array for the *one* result. No problem: instead of `fetchAllAssociative()` , use `fetchAssociative()` .

```
96 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 25
26 public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27 {
... lines 28 - 47
48     dd($result->fetchAssociative());
... lines 49 - 50
51 }
... lines 52 - 96
```

And now... beautiful! We get just that first row.

Hydrating into an Object

Now, you *may* remember that our method is *supposed* to return a `CategoryFortuneStats` object that we created earlier. Can we convert our array result into that object? Sure! It's not fancy, but easy enough.

We could return a `new CategoryFortuneStats()` ... and then grab the array keys from `$result->fetchAssociative()` ... and pass them as the correct arguments.

Or, you can be even *lazier* and use the spread operator along with named arguments. Check it out: the arguments are called `fortunesPrinted` , `fortunesAverage` , and `categoryName` . Over *here*, they are `fortunesPrinted` , `fortunesAverage` , and `name` ... not `categoryName` . Let's fix that. Down here, add `as categoryName` . And then... yep! It's called `categoryName` .

Now we can use named arguments. Remove the `dd()` and the other return. To `CategoryFortuneStats` , pass `...$result->fetchAssociative()` .

```
95 lines | src/Repository/FortuneCookieRepository.php
... lines 1 - 25
26     public function countNumberPrintedForCategory(Category $category): CategoryFortuneStats
27     {
... lines 28 - 48
49         return new CategoryFortuneStats(...$result->fetchAssociative());
50     }
... lines 51 - 95
```

This will grab that array and spread it out across those arguments so that we have three *correctly* named arguments... which is just kind of fun.

And now... our page works!

Next: Let's talk about organizing our repository so we can *reuse* parts of our queries in *multiple* methods.

Chapter 11: Reusing Queries in the Query Builder

Open up `CategoryRepository` . We have a few places in here where we `->leftJoin()` over to `fortuneCookies` and select fortune cookies. In the future, we may need to do that in *even more* methods... so it would be super-duper if we could *reuse* that logic instead of repeating it over and over again. Let's do that!

```
107 lines | src/Repository/CategoryRepository.php
... lines 1 - 17
18 class CategoryRepository extends ServiceEntityRepository
19 {
... lines 20 - 52
53 public function findWithFortunesJoin(int $id): ?Category
54 {
55     return $this->createQueryBuilder('category')
56         ->addSelect('fortuneCookie')
57         ->leftJoin('category.fortuneCookies', 'fortuneCookie')
... lines 58 - 61
62 }
... lines 63 - 105
106 }
```

Anywhere inside here, add a new `private function` called `addFortuneCookieJoinAndSelect()` . This will accept a `QueryBuilder` object (make sure you get the one from `Doctrine\ORM` - the "Object Relational Mapper"), and let's call it `$qb` . This will also *return* a `QueryBuilder` .

```
113 lines | src/Repository/CategoryRepository.php
... lines 1 - 7
8 use Doctrine\ORM\QueryBuilder;
... lines 9 - 18
19 class CategoryRepository extends ServiceEntityRepository
20 {
... lines 21 - 82
83 private function addFortuneCookieJoinAndSelect(QueryBuilder $qb): QueryBuilder
84 {
85
86 }
... lines 87 - 111
112 }
```

The next step is pretty simple. Go steal the `JOIN` logic from above... and, down here, say `return $qb ...` and paste that... being sure to clean up any spacing mess that may have occurred.

```
115 lines | src/Repository/CategoryRepository.php
... lines 1 - 82
83 private function addFortuneCookieJoinAndSelect(QueryBuilder $qb): QueryBuilder
84 {
85     return $qb
86         ->addSelect('fortuneCookie')
87         ->leftJoin('category.fortuneCookies', 'fortuneCookie');
88 }
... lines 89 - 115
```

And... done! We can now call this method, pass in the `QueryBuilder` , and *it* will add the `JOIN` and `SELECT` for us.

The result is *pretty* nice. Up here, we can say `$qb = $this->createQueryBuilder('category')` ... then below, `return $this->addFortuneCookieJoinAndSelect()` passing `$qb` .

```
115 lines | src/Repository/CategoryRepository.php
... lines 1 - 41
42 public function search(string $term): array
43 {
44     $qb = $this->createQueryBuilder('category');
45
46     return $this->addFortuneCookieJoinAndSelect($qb)
... lines 47 - 51
52 }
... lines 53 - 115
```

We create the `$qb` , pass it to the method, it *modifies* it... then also *returns* the `QueryBuilder` , so we can just chain off of it like normal.

Spin over and try the "Search" feature. And... oh... of course that breaks! We need to remove this excess code. If we try it now... great success!

To celebrate, repeat that same thing down here. Replace `return` with `$qb = ...` below that, say `return $this->addFortuneCookieJoinAndSelect()` passing in `$qb` , and then remove `->addSelect()` and `->leftJoin()` .

```
115 lines | src/Repository/CategoryRepository.php
... lines 1 - 53
54 public function findWithFortunesJoin(int $id): ?Category
55 {
56     $qb = $this->createQueryBuilder('category');
57
58     return $this->addFortuneCookieJoinAndSelect($qb)
59         ->andWhere('category.id = :id')
60         ->setParameter('id', $id)
61         ->getQuery()
62         ->getOneOrNullResult();
63 }
... lines 64 - 115
```

This is for the Category page, so if we click any category...perfect! It's still rocking.

[Making the QueryBuilder Argument Optional](#)

But... we can even make this even nicer! Instead of requiring the `QueryBuilder` object as an argument, make it *optional*.

Watch: down here, tweak this so that *if* we have a `$qb` , use it, otherwise, `$this->createQueryBuilder('category')` . So *if* a `QueryBuilder` was passed in, use this and call `->addSelect()` , *else* create a fresh `QueryBuilder` and call `->addSelect()` on *that*.

```
111 lines | src/Repository/CategoryRepository.php
... lines 1 - 78
79 private function addFortuneCookieJoinAndSelect(QueryBuilder $qb = null): QueryBuilder
80 {
81     return ($qb ?? $this->createQueryBuilder('category'))
... lines 82 - 83
84 }
... lines 85 - 111
```

The advantage is that we don't need to initialize our `QueryBuilder` at all up here...and the same thing goes for the method above.

```
111 lines | src/Repository/CategoryRepository.php
... lines 1 - 41
42 public function search(string $term): array
43 {
44     return $this->addFortuneCookieJoinAndSelect()
... lines 45 - 49
50 }
... line 51
52 public function findWithFortunesJoin(int $id): ?Category
53 {
54     return $this->addFortuneCookieJoinAndSelect()
... lines 55 - 58
59 }
... lines 60 - 111
```

But you *can* see how important it is that we're using a *consistent* alias everywhere. We're referencing `category.name` , `category.iconKey` , and `category.id` ... so we need to make sure that we always create a `QueryBuilder` using that *exact* alias. Else... things would get explodey.

Let's add one more reusable method: `private function addOrderByCategoryName()` ... because we're probably going to want to *always* order our data in the same way. Give this the usual `QueryBuilder $qb = null` argument, return a `QueryBuilder` , and the inside is pretty simple. I'll steal the code above... let me hit "enter" so it looks a bit better...and then start the same way. Create a `QueryBuilder` if we need to, and then say `->addOrderBy('category.name')` , followed by `Criteria::DESC` , which we used earlier in our `search()` method. And yes, we *are* sorting in *reverse* alphabetical order because, well, honestly I have no idea *what* I was thinking when I coded that part.

```
117 lines | src/Repository/CategoryRepository.php
... lines 1 - 85
86 private function addOrderByCategoryName(QueryBuilder $qb = null): QueryBuilder
87 {
88     return ($qb ?? $this->createQueryBuilder('category'))
89         ->addOrderBy('category.name', Criteria::DESC);
90 }
... lines 91 - 117
```

To use this, we need to break things up a bit. Start with `$qb = $this->addOrderByCategoryName()` and pass nothing. Then pass *that* `$qb` to the second part.

```
118 lines | src/Repository/CategoryRepository.php
... lines 1 - 41
42 public function search(string $term): array
43 {
44     $qb = $this->addOrderByCategoryName();
45
46     return $this->addFortuneCookieJoinAndSelect($qb)
... lines 47 - 50
51 }
... lines 52 - 118
```

As soon as you have multiple shortcut methods, you can't chain them *all*... which is a small bummer. But this *does* still allow us to remove the `->addOrderBy()` down here.

If we try it now... the page still works! And if we try searching for something on the homepage...that's looking good too!

Next: let's learn about the `Criteria` system: a *really* cool way to efficiently filter *collection* relationships inside the database, while keeping your code dead-simple.

Chapter 12: Criteria: Filter Relation Collections

On the category show page, we're looping over all the fortune cookies in that category. Let's check out the template: `templates/fortune/showCategory.html.twig`. Here it is: we loop over `category.fortuneCookies` and render some stuff.

```
40 lines | templates/fortune/showCategory.html.twig
... lines 1 - 20
21     {% for fortuneCookie in category.fortuneCookies %}
22         <tr class="hover:bg-slate-200">
23             <td class="border p-4">
24                 {{ fortuneCookie.fortune }}
25             </td>
26             <td class="border p-4">
27                 {{ fortuneCookie.numberPrinted }} printed since {{ fortuneCookie.createdAt|date('M jS Y') }}
28             </td>
29         </tr>
30     {% endfor %}
... lines 31 - 40
```

But... there's a problem. Open up the `FortuneCookie` entity. It has a `bool $discontinued` flag. Occasionally, we have to stop producing a specific fortune cookie... for one reason or another. Like the time we had a fortune cookie that said "You will be happy... until you realize reality is an illusion". That one slipped past quality control. When this happens, we set `discontinued` to true.

At the moment, we're looping over *all* the fortune cookies for a category: including both current *and* discontinued cookies! But management is really only interested in *current* fortune cookies. We need a way to *hide* the discontinued ones. How can we do that?

Over in the controller for this page - `FortuneController` - we *could* create a separate query from the `$fortuneCookieRepository` with

```
WHERE category = :category and discontinued = false.
```

But... that's lame! Looping over `category.fortuneCookies` is so easy! Do we really need to back up to the controller, create a custom query and pass in the results as a new Twig variable? Couldn't we somehow use the `category` object... but filter *out* the discontinued cookies! Absolutely! And if we do it correctly, we can do it *really* efficiently.

The first step is optional, but in the controller, change `->findWithFortunesJoin()` back to just `->find()`. I'm doing this - which removes the join - *just* so that it's easier to see the end result of what we're about to do.

```
47 lines | src/Controller/FortuneController.php
... lines 1 - 12
13 class FortuneController extends AbstractController
14 {
... lines 15 - 30
31 public function showCategory(int $id, CategoryRepository $categoryRepository, FortuneCookieRepository $fortuneCookieRepository): Response
32 {
33     $category = $categoryRepository->find($id);
... lines 34 - 44
45 }
46 }
```

Doing this doesn't change the page... except that our queries go up to three. That's one query for the `Category`, our custom query that we're making, and then one query for all the fortunes *inside* of this `Category`.

[Adding a Custom Entity Method for Discontinued Cookies](#)

Remember the goal: we want to be able to call *something* on the `Category` object to get back the related fortune cookies...but hiding the discontinued ones.

Open up the `Category` entity and find `getFortuneCookies()`. There it is. Below, add a new method called `getFortuneCookiesStillInProduction()`. This, like the normal method, will return a Doctrine `Collection`. And... just to help my editor, copy the `@return` doc above to say that this is a `Collection` of `FortuneCookie` objects.

```
99 lines | src/Entity/Category.php
... lines 1 - 10
11 class Category
12 {
    ... lines 13 - 60
61 /**
62  * @return Collection<int, FortuneCookie>
63  */
64 public function getFortuneCookiesStillInProduction(): Collection
65 {
66
67 }
    ... lines 68 - 97
98 }
```

So... what do we do inside? We *could* loop over `$this->fortuneCookies` as `$fortuneCookie` and create an array of objects that are *not* discontinued. Easy!

But... as soon as we start working with `$this->getFortuneCookies()`, that will cause Doctrine to query for *every* related fortune cookie. Do you see the problem? We might be asking Doctrine to query and prepare 100 `FortuneCookie` objects... even though this final `$inProduction` collection may only contain 10 of them. What a waste!

What we *really* want to do is tell Doctrine that *when* it makes the query for the related fortune cookies, it should add an extra `WHERE discontinued = false` to that query.

Hello Criteria

But... how the heck do we do that? Doctrine makes that query automatically and... magically somewhere in the background. Whelp, this is where the *criteria system* comes in handy.

It works like this: `$criteria = Criteria::` - the one from `Doctrine\Common\Collections` - `create()`.

```
103 lines | src/Entity/Category.php
... lines 1 - 7
8 use Doctrine\Common\Collections\Criteria;
    ... lines 9 - 11
12 class Category
13 {
    ... lines 14 - 64
65 public function getFortuneCookiesStillInProduction(): Collection
66 {
67     $criteria = Criteria::create()
    ... lines 68 - 70
71 }
    ... lines 72 - 101
102 }
```

This object is a bit like the `QueryBuilder`, but not *exactly* the same. We can say `->andWhere()` and then use `Criteria::` again with `expr()->`. This `expr()` or "expression" lets us, sort of, *build* the WHERE clause. It has methods like `in`, `contains` or `gt` for "greater than". We want `eq()` for "equals". Inside, say `discontinued`, `false`.

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

```
... lines 1 - 64
65     public function getFortuneCookiesStillInProduction(): Collection
66     {
67         $criteria = Criteria::create()
68             ->andWhere(Criteria::expr()->eq('discontinued', false));
... lines 69 - 70
71     }
... lines 72 - 103
```

Ok, this, by itself, just creates an object that "describes" a **WHERE** clause that could be added to some *other* query. To use it, return `$this->fortuneCookies->matching($criteria)` .

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

```
... lines 1 - 64
65     public function getFortuneCookiesStillInProduction(): Collection
66     {
... lines 67 - 69
70         return $this->fortuneCookies->matching($criteria);
71     }
... lines 72 - 103
```

Cool, huh? We're saying:

Hey Doctrine! Take this collection, but only return the ones that match this criteria.

And as we'll see in a minute, this will *modify* the query to get those fortune cookies!

To use this method, over in `showCategory.html.twig` , replace the `category.fortuneCookies` loop with `category.fortuneCookiesStillInProduction` .

40 lines | [templates/fortune/showCategory.html.twig](#)

```
... lines 1 - 2
3     {% block body %}
... lines 4 - 20
21         {% for fortuneCookie in category.fortuneCookiesStillInProduction %}
... lines 22 - 29
30         {% endfor %}
... lines 31 - 38
39     {% endblock %}
```

Let's do this! Refresh, and... I don't actually know if any of these are discontinued, but it *did* go from three to two! And the best part? Check out that query! Here's the first one for the category, here's our custom one...but take a look at this last query. When we ask for the "fortune cookies still in production", it queries from `fortune_cookie` , where the `category` = our category *and* where `t0.discontinued` is false! So it made the most *efficient* query to fetch *just* the fortune cookies that we need. That's *amazing*.

[Organizing your Criteria Code in the Repository](#)

Now, one minor downside is that... I normally like to keep my query logic inside a repository...not in the middle of an entity. Fortunately, we *can* move it there.

Because this deals with fortune cookies, open `FortuneCookieRepository` and, anywhere, add a new **public static function** called... how about `createFortuneCookiesStillInProductionCriteria()` . This will return a `Criteria` object.

Now, grab the `$criteria` statement from the entity... and return that.

102 lines | src/Repository/FortuneCookieRepository.php

```
... lines 1 - 8
9  use Doctrine\Common\Collections\Criteria;
... lines 10 - 19
20 class FortuneCookieRepository extends ServiceEntityRepository
21 {
... lines 22 - 26
27     public static function createFortuneCookiesStillInProductionCriteria(): Criteria
28     {
29         return Criteria::create()
30             ->andWhere(Criteria::expr()->eq('discontinued', false));
31     }
... lines 32 - 100
101 }
```

The Method is Static?

And yes, this *is* a **static** method... which I don't use *too* often. There are two reasons for this. First, these **Criteria** objects aren't actually making queries... and they don't rely on any data or services. And so, this method *can* be static. Second, and more importantly, we don't have access to the repository object from inside **Category**. So... if we want to call a method on a repository, it needs to be **static**. This is a special thing I typically do in my repositories *only* for this criteria situation.

Back in the entity, say `$criteria` equals `FortuneCookieRepository::createFortuneCookiesStillInProductionCriteria()`.

103 lines | src/Entity/Category.php

```
... lines 1 - 5
6  use App\Repository\FortuneCookieRepository;
... lines 7 - 12
13 class Category
14 {
... lines 15 - 65
66     public function getFortuneCookiesStillInProduction(): Collection
67     {
68         $criteria = FortuneCookieRepository::createFortuneCookiesStillInProductionCriteria();
69
70         return $this->fortuneCookies->matching($criteria);
71     }
... lines 72 - 101
102 }
```

Logic centralization, check! Oh, and we can even reuse these **Criteria** objects inside a **QueryBuilder**. Let's see... I don't have a good example... so... in this method, above, let's pretend I'm creating a **QueryBuilder** with `$this->createQueryBuilder('fortune_cookie')`. To add the criteria it's... `->addCriteria(self::createFortuneCookiesStillInProduction())`.

So, even though the criteria system is a bit different from the normal **QueryBuilder**, we *can* still reuse them everywhere. Oh, and let's check that things are still working. We're good!

Using the Criteria System in the Controller + EXTRA_LAZY Fetch

On the homepage, we have a similar problem. This says "Proverbs(3)", but if we click that, there are *two*. What's happening here? Over in `homepage.html.twig` ... let's see... ah, yes. We're looping over **categories**, and then calling `category.fortuneCookies|length` which, as we know, returns *all* the fortune cookies. Change that to `fortuneCookiesStillInProduction`.

17 lines | templates/fortune/homepage.html.twig

```
... lines 1 - 2
3  {% block body %}
... lines 4 - 7
8  {% for category in categories %}
... line 9
10     <span class="fa {{ category.iconKey }}"></span> <span class="font-bold text-lg">{{ category.name }}</span> ({{ category.fortuneCoc
... lines 11 - 13
14     {% endfor %}
... line 15
16 {% endblock %}
```

Back on the homepage, watch this "(3)". It *should* go down to 2, and...it *does*. But that's not even the best part. Open up the query for that. Remember, thanks to our fetch **EXTRA_LAZY**, because we're only counting the number of fortune cookies, it knows to make a super-fast **COUNT** query. And thanks to the criteria system, it's selecting

COUNT FROM fortune_cookies WHERE the category = our category and discontinued = false. Wow!

Next: We want to hide discontinued fortune cookies from everywhere on our site. Is there a way that we could hook into Doctrine and add that **WHERE** clause automatically... *everywhere*? There *is*. It's called *filters*.

Chapter 13: Filters: Automatically Modify Queries

Thanks to our cool new method, we can filter out discontinued fortune cookies. But what if we want to apply some criteria like this *globally* to *every* query to a table? Like, telling Doctrine that *whenever* we query for fortune cookies, we want to add a `WHERE discontinued = false` to that query.

That sounds *crazy*. And yet, it's *totally* possible. To demonstrate, let's revert our two templates back to the way they were before. And now... if we go into "Proverbs"... yep! All 3 fortunes show up again.

Hello Filters

To apply a "global" WHERE clause, we can create a Doctrine *filter*. In the `src/` directory, add a new directory called `Doctrine/` for organization. Inside that, add a new *class* called `DiscontinuedFilter`. Make this extend `SQLFilter` ... then go to Code -> Generate (or "command" + "N" on a Mac) and select "Implement Methods" to generate the one method we need `addFilterConstraint()`.

```
15 lines | src/Doctrine/DiscontinuedFilter.php
... lines 1 - 4
5 use Doctrine\ORM\Mapping\ClassMetadata;
6 use Doctrine\ORM\Query\Filter\SQLFilter;
7
8 class DiscontinuedFilter extends SQLFilter
9 {
10     public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias)
11     {
12         // TODO: Implement addFilterConstraint() method.
13     }
14 }
```

Once we have things set up, Doctrine will call `addFilterConstraint()` when it's building *any* query and pass us some info about *which* entity we're querying for: that's this `ClassMetadata` thing. It will also pass us the `$targetTableAlias`, which we'll need in a minute to modify the query.

Oh, and to avoid a deprecation notice, add a `string` return type to the method.

To better see what's happening, let's do our favorite thing and `dd($targetEntity, $targetTableAlias)`.

```
15 lines | src/Doctrine/DiscontinuedFilter.php
... lines 1 - 9
10 public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias): string
11 {
12     dd($targetEntity, $targetTableAlias);
13 }
... lines 14 - 15
```

Activating the Filter

But... when we head over and refresh the page...nothing happens! Unlike some things, filters are *not* activated automatically simply by creating the class. Activating it is a two-step process.

First, in `config/packages/doctrine.yaml`, we need to tell Doctrine that the filter exists. Anywhere directly under the `orm` key, add `filters` and then `fortuneCookie_discontinued`. That string could be anything... and you'll see how we use it in a minute. Set this to the class: `App\Doctrine\DiscontinuedFilter`.

47 lines | config/packages/doctrine.yaml

```
1 doctrine:
  ... lines 2 - 7
8 orm:
  ... lines 9 - 18
19 filters:
20     fortuneCookie_discontinued: App\Doctrine\DiscontinuedFilter
  ... lines 21 - 47
```

Easy peasy.

This *is* now *registered* with Doctrine... but as you can see over here, it's *still* not *called*. The second step is to *activate* it *where* you want it. In some cases, you might want this `DiscontinuedFilter` to be used on *one* section of your site, but not on another.

Open the controller... there we go... head up to the homepage and autowire `EntityManagerInterface $entityManager`. Then, right on top, say `$entityManager->getFilters()` followed by `->enable()`. Then pass this the *same* key we used in `doctrine.yaml` - `fortuneCookie_discontinued`. Go grab it... and paste.

50 lines | src/Controller/FortuneController.php

```
... lines 1 - 13
14 class FortuneController extends AbstractController
15 {
  ... line 16
17     public function index(Request $request, CategoryRepository $categoryRepository, EntityManagerInterface $entityManager): Response
18     {
19         $entityManager->getFilters()
20         ->enable('fortuneCookie_discontinued');
  ... lines 21 - 30
31     }
  ... lines 32 - 48
49 }
```

With any luck, every query that we make *after* this line will use that filter. Head over to the homepage and...yes! It hit it!

And woh! This `ClassMetadata` is a *big* object that knows *all* about our entity. Down here, apparently, for whatever query we're making first, the table alias - the alias being used in the query - is `c0_`. Ok! Let's get to work!

[Adding the Filter Logoc](#)

As I mentioned, this will be called for *every* query. So we need to be careful to *only* add our `WHERE` clause when we're querying for fortune cookies. To do that, say if `$targetEntity->name !== FortuneCookie::class`, then `return ""`.

20 lines | src/Doctrine/DiscontinuedFilter.php

```
... lines 1 - 8
9 class DiscontinuedFilter extends SQLFilter
10 {
11     public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias): string
12     {
13         if ($targetEntity->getReflectionClass()->name !== FortuneCookie::class) {
14             return "";
15         }
  ... lines 16 - 17
18     }
19 }
```

This method returns a `string` ... and that string is basically added to a `WHERE` clause. At the bottom, `return sprintf('"%s.discontinued = false')'`, passing `$targetTableAlias` for the wildcard.

20 lines | [src/Doctrine/DiscontinuedFilter.php](#)

```
... lines 1 - 10
11 public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias): string
12 {
... lines 13 - 16
17     return sprintf('%s.discontinued = false', $targetTableAlias);
18 }
... lines 19 - 20
```

Ready to check this out? On the homepage, the "Proverbs" count should go from 3 to 2. And... it does! Check out the query for this. Yup! It has `t0.discontinued = false` inside of every query for fortune cookies. That's *awesome*!

Passing Parameters to Filters

Now, one *tricky* thing about these filters is that they are *not* services. So you *can't* have a constructor... it's just not allowed. If we need to pass something to this - like some config - we have to do it a different way. For example, let's pretend that sometimes we want to *hide* discontinued cookies... but other times, we want to show *only* discontinued ones - the reverse. Essentially, we want to be able to toggle this value from `false` to `true`.

To do that, change this to `%s` and fill it in with `$this->getParameter()` ... passing some string I'm making up: `discontinued`. You'll see how that's used in a minute.

20 lines | [src/Doctrine/DiscontinuedFilter.php](#)

```
... lines 1 - 10
11 public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias): string
12 {
... lines 13 - 16
17     return sprintf('%s.discontinued = %s', $targetTableAlias, $this->getParameter('discontinued'));
18 }
... lines 19 - 20
```

Now, I don't *normally* add `%s` to my queries... because that can allow SQL injection attacks. In this case, it's okay, but only because the `getParameter()` method is designed to escape the value *for* us. In every other situation, avoid this.

If we head over and try it now... we get a giant error! Yay!

```
Parameter 'discontinued' does not exist.
```

That's true! As soon as you read a parameter, you need to pass that *in* when you enable the filter. Do that with `->setParameter('discontinued')` ... and let's say `false`.

51 lines | [src/Controller/FortuneController.php](#)

```
... lines 1 - 13
14 class FortuneController extends AbstractController
15 {
... line 16
17 public function index(Request $request, CategoryRepository $categoryRepository, EntityManagerInterface $entityManager): Response
18 {
19     $entityManager->getFilters()
20         ->enable('fortuneCookie_discontinued')
21         ->setParameter('discontinued', false);
... lines 22 - 31
32 }
... lines 33 - 49
50 }
```

If we reload now... it's working! What happens if we change this to `true`? Refresh again and... yep! The number changed! We rule!

Activating this Globally

Though... you're probably thinking:

Ryan, dude, yea, this is cool... but can't I enable this filter *globally*... without needing to put this code in every controller?

Absolutely! Head back to the controller and comment this out.

When we do that, the number goes back to 3. To enable it globally, head back to the configuration: we're going to make this a *little* more complicated. Bump this onto a new line, set that to `class` then set `enabled` to `true`.

And just like that, this will be enabled *everywhere*... though you could still disable it in specific controllers. Oh, but since we have the parameter, we also need `parameters`, with `discontinued: false`.

```
51 lines | config/packages/doctrine.yaml
1  doctrine:
    ... lines 2 - 7
8   orm:
    ... lines 9 - 18
19  filters:
20      fortuneCookie_discontinued:
21          class: App\Doctrine\DiscontinuedFilter
22          enabled: true
23          parameters:
24              discontinued: false
    ... lines 25 - 51
```

And... there we go! Filters are *cool*.

Next: Let's talk about how to use the handy `IN` operator with a query.

Chapter 14: WHERE IN()

We have categories for "Pets" and "Love", but if we search up here for "pets love"...no results! That makes sense. We're searching to see if this string is matching the `name` or the `iconKey`. Let's make our search smarter to see if we can match *both* of those categories by searching word by word.

The query for this lives in `CategoryRepository` ... on the `search()` method. The `$term` argument is the string we type in. Down here, let's say `$termList =` then `explode` that string into an array by splitting on empty spaces. If you want a *really* rich search, you should use a *real* search system. But we can do some pretty cool stuff just with the database.

```
119 lines | src/Repository/CategoryRepository.php
... lines 1 - 18
19 class CategoryRepository extends ServiceEntityRepository
20 {
... lines 21 - 41
42 public function search(string $term): array
43 {
44     $termList = explode(' ', $term);
... lines 45 - 51
52 }
... lines 53 - 117
118 }
```

Here's the goal: I want to *also* match results where `category.name` is *in* one of the words in the array.

Using the IN

Right after `category.name LIKE :searchTerm`, add `OR category.name IN`. The only tricky thing about this is the syntax. Add `()`. If we were writing a raw SQL query, we would write a list here, like `'foo', 'bar'`. But with the query builder, instead, put a placeholder - like `:termList`. Below pass that in: `->setParameter('termList', $termList)`.

```
120 lines | src/Repository/CategoryRepository.php
... lines 1 - 41
42 public function search(string $term): array
43 {
... lines 44 - 46
47     return $this->addFortuneCookieJoinAndSelect($qb)
48         ->andWhere('category.name LIKE :searchTerm OR category.name IN (:termList) OR category.iconKey LIKE :searchTerm OR fortun
... line 49
50         ->setParameter('termList', $termList)
... lines 51 - 52
53 }
... lines 54 - 120
```

The *key* thing is that, when you use `IN`, you *will* need the parentheses like normal...but inside of that, instead of a comma-separated list, you'll set an *array*. Doctrine will transform that *for* us.

And now... nice! Once you know how it works, it's *just* that easy.

Next: You're probably familiar with the `RAND()` function for MySQL, or maybe the `YEAR()` function... or one of the many MySQL or PostgreSQL functions that exist. Well, you might be surprised to learn that some of those *don't* work out of the box.

Chapter 15: Using RAND() or Other Non-Supported Functions

For the heck of it, let's randomize the order of the fortunes on a page. Try this category, which has 4.

Start by opening up `FortuneController` and finding `showCategory()`. Right now, we're querying for the category in the normal way. Then, in our template, we loop over `category.fortuneCookies`.

Change the query *back* to `->findWithFortunesJoin()`, which lives over here in `CategoryRepository`. Remember: this joins over to `FortuneCookie` and selects that data, solving our N+1 problem.

```
52 lines | src/Controller/FortuneController.php
... lines 1 - 13
14 class FortuneController extends AbstractController
15 {
... lines 16 - 35
36 public function showCategory(int $id, CategoryRepository $categoryRepository, FortuneCookieRepository $fortuneCookieRepository): Response
37 {
38     $category = $categoryRepository->findWithFortunesJoin($id);
... lines 39 - 49
50 }
51 }
```

Now that we're doing this, we can *also* control the *order*. Say `->orderBy('RAND()', Criteria::ASC)`. We're only querying for *one* `Category` ... but this will control the order of the related fortune cookies as well...which we'll see when we loop over them.

```
121 lines | src/Repository/CategoryRepository.php
... lines 1 - 6
7 use Doctrine\Common\Collections\Criteria;
... lines 8 - 18
19 class CategoryRepository extends ServiceEntityRepository
20 {
... lines 21 - 54
55 public function findWithFortunesJoin(int $id): ?Category
56 {
57     return $this->addFortuneCookieJoinAndSelect()
... lines 58 - 59
60     ->orderBy('RAND()', Criteria::ASC)
... lines 61 - 62
63 }
... lines 64 - 119
120 }
```

Pretty cool! If we try this... *error*?

Expected known function, got `RAND`

Wait... `RAND` is a known MySQL function. So... why doesn't it work? Ok, Doctrine supports a *lot* of functions inside DQL, but not *everything*. Why? Because Doctrine is designed to work with many different types of databases...and if only one or some databases support a function like `RAND`, then Doctrine *can't* support it. *Fortunately*, we *can* add this function or any custom function we want *ourselves* or, really, via a library.

Search for the `beberlei/doctrineextensions` library. This is *awesome*. It allows us to add a *bunch* of different functions to multiple database types. Go down here and grab the `composer require` line... but we don't need the `dev-master` part. Run that!


```
composer require beberlei/doctrineextensions
```

Installing this doesn't change anything in our app...it just adds a bunch of code that we can *activate* for any functions that we want. To do that, back over in `config/packages/doctrine.yaml`, somewhere under `orm`, say `dql`. There are a bunch of different categories under here, which you can read more about in the documentation. In our case, we need to add `numeric_functions` along with the *name* of the function, which is `rand`. Set this to the class that will let Doctrine know what to do:

`DoctrineExtensions\Query\Mysql\Rand`.

54 lines | [config/packages/doctrine.yaml](#)

```
1 doctrine:
  ... lines 2 - 7
8   orm:
  ... lines 9 - 24
25     dql:
26         numeric_functions:
27             rand: DoctrineExtensions\Query\Mysql\Rand
  ... lines 28 - 54
```

You definitely don't have to take my word about how this should be set up. Over in the documentation... there's a "config" link down here... and if you click on `mysql.yml`, you can see that it describes *all* the different things you can do and how to activate them.

I'll close that up... refresh, and... got it! Each time we refresh, the results are coming back in a different order.

Okay, *one more* topic team! Let's finish with a complex `groupBy()` situation where we select some objects *and* some extra data all at once.

Chapter 16: Using GROUP BY to Fetch & Count in 1 Query

One last challenge. On the homepage, we have seven queries. That's one to fetch the categories... and 6 more to get the fortune cookie count for *each* of those 6 categories.

Having 7 queries is... probably not a problem... and you shouldn't worry about optimizing performance until you actually see that there *is* a problem. But let's *challenge* ourselves to turn these seven queries into *one*.

Let's think: we *could* query for all the categories, **JOIN** over to the related fortune cookies, **GROUP BY** the category, and then **COUNT** the fortune cookies. If that doesn't make sense, no worries. We'll see it in action.

Using a Group By To Select an Object + Other Data

Head over to **FortuneController**. We're on the homepage, and we're using the `findAllOrdered()` method from `$categoryRepository`. Go find that method... here it is. We're already selecting from `category`. Now *also*

`->addSelect('COUNT(fortuneCookie.id) AS fortuneCookiesTotal')`. To join and get that `fortuneCookie` alias, add `->leftJoin('category.fortuneCookies', 'fortuneCookie')`. Finally, for this **COUNT** to work correctly, say `->addGroupBy('category.id')`.

```
125 lines | src/Repository/CategoryRepository.php

... lines 1 - 18
19 class CategoryRepository extends ServiceEntityRepository
20 {
... lines 21 - 28
29 public function findAllOrdered(): array
30 {
... line 31
32     $qb = $this->createQueryBuilder('category')
33         ->addOrderBy('category.name', Criteria::DESC)
34         ->addSelect('COUNT(fortuneCookie.id) AS fortuneCookiesTotal')
35         ->leftJoin('category.fortuneCookies', 'fortuneCookie')
36         ->addGroupBy('category.id');
... lines 37 - 40
41     }
... lines 42 - 123
124 }
```

Okay, let's see what we get! Down here, `dd($query->getResult())`.

```
125 lines | src/Repository/CategoryRepository.php

... lines 1 - 28
29 public function findAllOrdered(): array
30 {
... lines 31 - 36
37     $query = $qb->getQuery();
38     dd($query->getResult());
... lines 39 - 40
41     }
... lines 42 - 125
```

Previously, this returned an `array` of `Category` objects. If we refresh... it *is* an array, but it's now an *array of arrays* where the `0` key is the `Category` object, and then we have this extra `fortuneCookiesTotal`. So... it selected *exactly* what we wanted! But... it changed the underlying structure. And it kind of *had* to, right? It needed to *somehow* give us the `Category` object *and* the extra column behind the scenes.

Remove the `dd` statement. This still returns an `array` ... but remove the `@return` because it no longer returns an array of `Category` objects. We could also update that to some fancier phpdoc that describes the new structure.

Next, to account for the new return, head to `homepage.html.twig` . We're looping over `category in categories` ... which isn't quite right now: the category is on this `0` index. Change this to say `for categoryData in categories` ... then inside add `set category = categoryData[0]` . It's ugly, but more on that in a minute.

```
18 lines | templates/fortune/homepage.html.twig
... lines 1 - 2
3  {% block body %}
... lines 4 - 7
8  {% for categoryData in categories %}
9      {% set category = categoryData[0] %}
... lines 10 - 14
15  {% endfor %}
... line 16
17  {% endblock %}
```

Scroll over to the `length` . Instead of reaching across the relationship - which *would* work, but would trigger extra queries - use `categoryData.fortuneCookiesTotal` .

```
18 lines | templates/fortune/homepage.html.twig
... lines 1 - 7
8  {% for categoryData in categories %}
... line 9
10  <a class="bg-orange-400 hover:bg-orange-500 text-white text-center rounded-full p-4" href="{{ path('app_category_show', {'id': categoryData.fortuneCookiesTotal }) }}">
11      <span class="fa {{ categoryData.iconKey }}"></span> <span class="font-bold text-lg">{{ categoryData.name }}</span> ({{ categoryData.fortuneCookiesTotal }})
12  </a>
... lines 13 - 14
15  {% endfor %}
... lines 16 - 18
```

Let's do this! Refresh and... just one query! Woo!

The Ugly Data Structure

The *worst* part about this is that the structure of our data changed...and now we have to read this ugly `0` key. I won't do it now, but a *better* solution would be to leverage a DTO object to hold this. For example, we might create a new class called `CategoryWithFortuneCount` with two properties - `$category` and `$fortuneCount` . In this repository method, we could loop over `$query->getResults()` and create a `CategoryWithFortuneCount` object for each one. Ultimately, our method would return an array of `CategoryWithFortuneCount` . Returning an array of objects is much nicer than an *array of arrays*.. with some random `0` index.

Fixing the Search Page

Speaking of that changed structure, if we search for something...we get an error:

```
Impossible to access a key "0" on an object of class Category .
```

It's... this line right here. When we search for something, we use the `search()` method and... surprise! That method doesn't have the new `addSelect()` and `groupBy()` : it still returns an array of `Category` objects.

121 lines | [src/Repository/CategoryRepository.php](#)

```
... lines 1 - 18
19 class CategoryRepository extends ServiceEntityRepository
20 {
    ... lines 21 - 38
39 /**
40  * @return Category[]
41  */
42 public function search(string $term): array
43 {
    ... lines 44 - 52
53 }
    ... lines 54 - 119
120 }
```

To fix that, create a **private function** down here that can hold the group by: `addGroupByCategory(QueryBuilder $qb)` and it'll return a **QueryBuilder**. Oh, and make the argument optional... then create a new query builder if we don't have one.

126 lines | [src/Repository/CategoryRepository.php](#)

```
... lines 1 - 79
80 private function addGroupByCategory(QueryBuilder $qb = null): QueryBuilder
81 {
82     return ($qb ?? $this->createQueryBuilder('category'))
    ... lines 83 - 85
86 }
    ... lines 87 - 126
```

Ok, head up and steal the logic - the `->addSelect()`, `->leftJoin()`, and `->addGroupBy()`. Paste that down here. Oh, and `addGroupByCategory()` isn't a great name: use `addGroupByCategoryAndCountFortunes()`.

124 lines | [src/Repository/CategoryRepository.php](#)

```
... lines 1 - 77
78 private function addGroupByCategory(QueryBuilder $qb = null): QueryBuilder
79 {
80     return ($qb ?? $this->createQueryBuilder('category'))
81         ->addSelect('COUNT(fortuneCookie.id) AS fortuneCookiesTotal')
82         ->leftJoin('category.fortuneCookies', 'fortuneCookie')
83         ->addGroupBy('category.id');
84 }
    ... lines 85 - 124
```

Awesome. Above, simplify! Change *this* to `addGroupByCategoryAndCountFortunes()` ... and then we don't need the `->addGroupBy()`, `->leftJoin()`, or `->addSelect()`.

124 lines | [src/Repository/CategoryRepository.php](#)

```
... lines 1 - 25
26 public function findAllOrdered(): array
27 {
    ... line 28
29     $qb = $this->addGroupByCategory()
30         ->addOrderBy('category.name', Criteria::DESC);
    ... lines 31 - 33
34 }
    ... lines 35 - 124
```

To make sure *that* part is working, spin over and... head back to the homepage. That looks good... but if we go forward... still broken. Down in `search()` add `$qb = $this->addGroupByCategoryAndCountFortunes($qb)`.

124 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 35
36 public function search(string $term): array
37 {
... line 38
39     $qb = $this->addOrderByCategoryName();
40     $qb = $this->addGroupByCategory($qb);
... line 41
42     return $this->addFortuneCookieJoinAndSelect($qb)
... lines 43 - 46
47     ->getResult();
48 }
... lines 49 - 124
```

And now... *another* error:

`fortuneCookie` is already defined.

Darn! But, yea, that makes sense. We're joining in our new method...and also in `addFortuneCookieJoinAndSelect()`. Fortunately, we don't *need* this second call at all anymore: we were joining and selecting to solve the N+1 problem...but now we have an even *more* advanced query to do that. Copy our new method, delete, then paste it over the old one.

123 lines | src/Repository/CategoryRepository.php

```
... lines 1 - 35
36 public function search(string $term): array
37 {
... lines 38 - 40
41     return $this->addGroupByCategory($qb)
... lines 42 - 46
47 }
... lines 48 - 123
```

And now... got it! Only 1 query!

Yo friends, we did it! Woo! Thanks for joining me on this magical ride through all things Doctrine Query. This stuff is just weird, cool and fun. I hope you enjoyed it as much as I did. If you encounter any *crazy* situation that we haven't thought about, have any questions, or pictures of your cat, we're always here for you down in the comments. Alright, see you next time!



