# Go Pro with Doctrine Queries

With <3 from SymfonyCasts

# Chapter 1: Doctrine DQL

Look, I know you already understand how to do queries in SQL - maybe you dream of JOINs, orders and sub-queries. That's really dorky, but I get it. But when you look at Doctrine, it's totally different - with its DQL and its query builder, with their *own* ways of doing joins, and this hydration of objects thing.

But did you know you can write native SQL queries in Doctrine? Yep! And you can opt in or out of *any* of its features. Use more of them in one place, where life is easier or when you're feeling like a Doctrine pro. Then go simpler and use less when things get tough or you need to squeeze out ever ounce of performance.

We'll learn about all of that. And don't worry - if you're good at SQL, you're going to be great at writing queries in Doctrine.

## Query for Fortune Cookies

Our app is a Fortune Cookie inventory system. Yep, we've *finally* hit the big time: working for a company that can tell you your future, wrapped up inside a cheap cookie shell.

There are six different fortune categories that are loaded from the database. And if you click on any of these, we see all of the fortunes for the category and how many have been printed.

The project is a small Symfony app - but all the Doctrine stuff translates to any app using the Doctrine ORM. We have 2 entities: Category and FortuneCookie:

113 lines | src/AppBundle/Entity/Category.php

```
... lines 1 - 7
/**
 * Category
 *
 * @ORM\Table(name="category")
 * @ORM\Entity(repositoryClass="AppBundle\Entity\CategoryRepository")
 */
class Category
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

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

    /**
     * @var string
     *
     * @ORM\Column(name="iconKey", type="string", length=20)
     */
    private $iconKey;

    /**
     * @ORM\OneToMany(targetEntity="FortuneCookie", mappedBy="category")
     */
    private $fortuneCookies;
... lines 43 - 111
}
```

```
176 lines | src/AppBundle/Entity/FortuneCookie.php
... lines 1 - 12
class FortuneCookie
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var Category
     *
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="fortuneCookies")
     * @ORM\JoinColumn(nullable=false)
     */
    private $category;

    /**
     * @var string
     *
     * @ORM\Column(name="fortune", type="string", length=255)
     */
    private $fortune;
... lines 38 - 174
}
```

With a ManyToOne relation from FortuneCookie to the Category:

```
176 lines | src/AppBundle/Entity/FortuneCookie.php
... lines 1 - 12
class FortuneCookie
{
... lines 15 - 23
    /**
     * @var Category
     *
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="fortuneCookies")
     * @ORM\JoinColumn(nullable=false)
     */
    private $category;
... lines 31 - 174
}
```

For our homepage, we're using the entity manager to fetch the Category's repository and call the built-in findAll function:

47 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 10
    /**
     * @Route("/", name="homepage")
     */
    public function homepageAction()
    {
        $categoryRepository = $this->getDoctrine()
            ->getManager()
            ->getRepository('AppBundle:Category');

        $categories = $categoryRepository->findAll();

        return $this->render('fortune/homepage.html.twig',[
            'categories' => $categories
        ]);
    }
... lines 26 - 47
```

This returns *every* Category, and so far, it lets us be lazy and avoid writing a custom query. The template loops over these and prints them out. AMAZING.

## Doctrine Query Language (DQL)

Time to write a query! One that will order the categories alphabetically. Call a new method called findAllOrdered():

47 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 13
    public function homepageAction()
    {
        $categoryRepository = $this->getDoctrine()
            ->getManager()
            ->getRepository('AppBundle:Category');

        $categories = $categoryRepository->findAllOrdered();
... lines 21 - 24
    }
... lines 26 - 47
```

This needs to live inside the CategoryRepository class. So create a public function findAllOrdered(). To prove things are wired up, put a die statement:

20 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 12
class CategoryRepository extends EntityRepository
{
    public function findAllOrdered()
    {
        die('this query will blow your mind...');
    }
}
```

Refresh! Sweet, ugly black text - we're hooked up!

Ok, so *you're* used to writing SQL, maybe MySQL queries. Well, Doctrine speaks a different language: DQL, or Doctrine Query Language. Don't worry though, it's so close to SQL, most of the time you won't notice the difference.

Let's see some DQL. So: $dql = 'SELECT cat FROM AppBundle\Entity\Category cat';:

24 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 12
class CategoryRepository extends EntityRepository
{
    public function findAllOrdered()
    {
        $dql = 'SELECT cat FROM AppBundle\Entity\Category cat';
... lines 18 - 21
    }
}
```

The big DQL difference is that instead of working with tables, you're working with PHP classes. And that's why we're selecting from the full class name of our entity. Symfony users are used to saying AppBundle:Category, but that's just a shortcut alias - internally it always turns into the full class name.

The cat part is an alias, just like SQL. And instead of SELECT *, you write the alias - SELECT cat. This will query for every column. Later, I'll show you how to query for only *some* fields.

## Executing DQL

To run this, we'll create a Query object. Get the EntityManager, call createQuery() and pass it in the DQL. And once we have the Query object, we can call execute() on it:

24 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 12
class CategoryRepository extends EntityRepository
{
    public function findAllOrdered()
    {
        $dql = 'SELECT cat FROM AppBundle\Entity\Category cat';

        $query = $this->getEntityManager()->createQuery($dql);

        return $query->execute();
    }
}
```

This will return an array of Category *objects*. Doctrine's normal mode is to always return *objects*, not an array of data. But we'll change that later.

Let's query for some fortunes! Refresh the page. Nice - we see the exact same results - this is what findAll() was doing in the background.

## Adding the ORDER BY

To add the ORDER BY, it looks just like SQL. Add ORDER BY, then cat.name DESC:

```
24 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 14
    public function findAllOrdered()
    {
        $dql = 'SELECT cat FROM AppBundle\Entity\Category cat ORDER BY cat.name DESC';

        $query = $this->getEntityManager()->createQuery($dql);

        return $query->execute();
    }
... lines 23 - 24
```

Refresh! Alphabetical categories! So that's DQL: SQL where you mention class names instead of table names. If you Google for "Doctrine DQL", you can find a lot more in the Doctrine docs, including stuff like joins.

## Show me the SQL!

Of course ultimately, Doctrine takes that DQL and turns it into a *real* MySQL query, or PostgreSQL of whatever your engine is. Hmm, so could we *see* this SQL? Well sure! And it might be useful for debugging. Just var_dump $query->getSQL():

```
25 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 14
    public function findAllOrdered()
    {
        $dql = 'SELECT cat FROM AppBundle\Entity\Category cat ORDER BY cat.name DESC';

        $query = $this->getEntityManager()->createQuery($dql);
        var_dump($query->getSQL());die;

        return $query->execute();
    }
... lines 24 - 25
```

Refresh! It's not terribly pretty, but there it is. For all the coolness, tried-and-true SQL lives behind the scenes. Remove that debug code.

# Chapter 2: The QueryBuilder

Doctrine speaks DQL, even though it converts it eventually to SQL. But actually, I don't write a lot of DQL. Instead, I use the QueryBuilder: an object that helps you build a DQL string. The QueryBuilder is one of my favorite parts of Doctrine.

## Creating the Query Builder

Let's comment out the $dql stuff. To create a QueryBuilder, create a $qb variable and call $this->createQueryBuilder() from inside a repository. Pass cat as the argument - this will be the alias to Category:

25 lines | src/AppBundle/Entity/CategoryRepository.php

```
// ... lines 1 - 12
class CategoryRepository extends EntityRepository
{
    public function findAllOrdered()
    {
        $qb = $this->createQueryBuilder('cat')
// ... lines 18 - 22
    }
}
```

## Building the Query

Now, let's chain some awesomeness! The QueryBuilder has methods on it like andWhere, leftJoin and addOrderBy. Let's use that - pass cat.name as the first argument and DESC as the second:

25 lines | src/AppBundle/Entity/CategoryRepository.php

```
// ... lines 1 - 12
class CategoryRepository extends EntityRepository
{
    public function findAllOrdered()
    {
        $qb = $this->createQueryBuilder('cat')
            ->addOrderBy('cat.name', 'ASC');
// ... lines 19 - 22
    }
// ... lines 24 - 25
```

This builds the exact same DQL query we had before. Because we're inside of the CategoryRepository, the createQueryBuilder() function automatically configures itself to select from the Category entity, using cat as the alias.

To get a Query object from this, say $qb->getQuery():

```
25 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 12
class CategoryRepository extends EntityRepository
{
    public function findAllOrdered()
    {
        $qb = $this->createQueryBuilder('cat')
            ->addOrderBy('cat.name', 'ASC');
        $query = $qb->getQuery();
... lines 20 - 22
    }
}
```

Wow.

Remember how we printed the SQL of a query? We can also print the DQL. So let's see how our hard work translates into DQL:

```
25 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 14
    public function findAllOrdered()
    {
        $qb = $this->createQueryBuilder('cat')
            ->addOrderBy('cat.name', 'ASC');
        $query = $qb->getQuery();
        var_dump($query->getDQL());die;

        return $query->execute();
    }
... lines 24 - 25
```

Refresh! Look closely:

```
SELECT cat FROM AppBundle\Entity\Category cat ORDER BY cat.name DESC
```

That's character-by-character the *exact* same DQL that we wrote before. So the query builder is just a *nice* way to help write DQL, and I prefer it because I get method auto-completion and it can help you re-use pieces of a query, like a complex JOIN, across multiple queries. I'll show you that later.

Remove the die statement and refresh to make sure it's working:

```
24 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 14
    public function findAllOrdered()
    {
        $qb = $this->createQueryBuilder('cat')
            ->addOrderBy('cat.name', 'ASC');
        $query = $qb->getQuery();

        return $query->execute();
    }
... lines 23 - 24
```

It looks perfect. To know more about the QueryBuilder, you can either keep watching (that's recommended) or use your IDE to see all the different methods the class has. But you should just keep watching.

# Chapter 3: And WHERE Or WHERE

The most common thing to do in a query is to add a WHERE clause. Unfortunately, Doctrine doesn't support that. I'm kidding!

I have a search box - let's search for "Lucky Number". This isn't hooked up yet, but it adds a query parameter ?q=lucky+number. Let's use that to only return categories matching that.

Back in FortuneController, add a Request $request argument to the controller. Below, let's look to see if there is a q query parameter on the URL or not. If there is, we'll search for it, otherwise, we'll keep finding all the categories. For the search, call a new method on the repository called search(), and pass in the term.

53 lines | src/AppBundle/Controller/FortuneController.php

```
// ... lines 1 - 7
use Symfony\Component\HttpFoundation\Request;
// ... lines 9 - 14
    public function homepageAction(Request $request)
    {
// ... lines 17 - 20
        $search = $request->query->get('q');
        if ($search) {
            $categories = $categoryRepository->search($search);
        } else {
            $categories = $categoryRepository->findAllOrdered();
        }
// ... lines 27 - 30
    }
// ... lines 32 - 53
```

Back in CategoryRepository, let's make that function:

33 lines | src/AppBundle/Entity/CategoryRepository.php

```
// ... lines 1 - 12
class CategoryRepository extends EntityRepository
{
// ... lines 15 - 23
    public function search($term)
    {
// ... lines 26 - 30
    }
}
```

We'll make a QueryBuilder just like before, but do the entire query in just one statement. Start by calling createQueryBuilder() and passing it cat:

```
33 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 12
class CategoryRepository extends EntityRepository
{
... lines 15 - 23
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
... lines 27 - 30
    }
}
```

The only thing our query needs is a WHERE clause to match the term to the name of the Category. Let's chain!

## Building AND WHERE into a Query

Use a function called andWhere(). Don't worry - Doctrine won't add an AND to the query, unless it's needed. Inside, write cat.name = . But instead of passing the variable directly into the string like this, use a placeholder. Type colon, then make up a term. On the next line, use setParameter to tell Doctrine what I want to fill in for that term. So type searchTerm, should be replaced with $term. This avoids SQL injection attacks, so don't muck it up! Finally, we call getQuery() - like before - and execute():

```
33 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 23
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
            ->andWhere('cat.name = :searchTerm')
            ->setParameter('searchTerm', $term)
            ->getQuery()
            ->execute();
    }
... lines 32 - 33
```

And just like that, we should be able to go back, refresh, and there's our "Lucky Number" category match. And on the homepage, we still see everything.

## Query with LIKE

But if we just search for "Lucky", we get nothing back because we're doing an exact match. But just like with normal SQL, we know that's easy to fix. And you already know how: just change = to LIKE - *just* like SQL!

It's just like writing SQL people! For the parameter value, surround it by percent signs to complete things. Refresh! We've got a match!

```
33 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 23
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
            ->andWhere('cat.name LIKE :searchTerm')
            ->setParameter('searchTerm', '%'.$term.'%')
            ->getQuery()
            ->execute();
    }
... lines 32 - 33
```

## OR WHERE

What about adding an OR WHERE to the query? The Category entity has an iconKey property, which is where we get this little bug icon. For "Lucky Number", it's set to fa-bug from Font Awesome. Search for that. No results of course!

Let's update our query to match on the name OR iconKey property. If you're guessing that there's an orWhere() method, you're right! If you're guessing that I'm going to use it, you're wrong!

The string inside of the andWhere is a mini-DQL expression. So you can add OR cat.iconKey LIKE :searchTerm:

33 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 23
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
            ->andWhere('cat.name LIKE :searchTerm OR cat.iconKey LIKE :searchTerm')
            ->setParameter('searchTerm', '%'.$term.'%')
            ->getQuery()
            ->execute();
    }
... lines 32 - 33
```

And the searchTerm placeholder is already being filled in:

Refresh! Another match!

## Avoid orWhere() and where()

So even though there is an orWhere() function, don't use it - it can cause WTF moments. Imagine if Category had an enabled property, and we built a query like this:

```
$this->createQueryBuilder('cat')
    ->andWhere('cat.name LIKE :searchTerm')
    ->orWhere('cat.iconKey LIKE :searchTerm')
    ->andWhere('cat.enabled = :enabled')
    ->setParameter('searchTerm', '%'.$term.'%')
    ->setParameter('enabled', true)
    ->getQuery()
    ->execute();
```

What would the SQL look like for this? Would it have the three WHERE clauses in a row, or would it correctly surround the first two with parentheses?

```
SELECT * FROM category WHERE
    name LIKE '%lucky%' OR iconKey LIKE '%lucky%' AND enabled = 1;

SELECT * FROM category WHERE
    (name LIKE '%lucky%' OR iconKey LIKE '%lucky%') AND enabled = 1;
```

Doctrine does the second and the query works as expected. But it's a lot less clear to read. Instead, think of each andWhere() as being surrounded by its *own* parentheses, and put OR statements in there.

Oh, and there's also a where() function. Don't use it either - it removes any previous WHERE clauses on the query, which you might be doing accidentally.

In other words, always use andWhere(), it keeps life simple.

# Chapter 4: JOINs!

I love JOINs. I do! I mean, a query isn't *truly* interesting unless you're joining across tables to do some query Kung fu. Doctrine makes JOINs *really* easy - it's one of my favorite features! Heck, they're *so* easy that I think it confuses people. Let me show you.

Right now, our search matches fields on the Category, but it *doesn't* match any of the fortunes in that Category. So if we search for cat, we get the no-results frowny face. Time to fix it!

## LEFT JOIN in SQL

The query for this page is built in the search() function. Let's think about what we need in SQL first. That query would select FROM category, but with a LEFT JOIN over to the fortune_cookie table ON fortune_cookie.categoryId = category.id. Once we have the LEFT JOIN in normal SQL land, we can add a WHERE statement to search on any column in the fortune_cookie table.

```
SELECT cat.* FROM category cat
    LEFT JOIN fortune_cookie fc ON fc.categoryId = cat.id
    WHERE fc.fortune LIKE '%cat%';
```

## ManyToOne and OneToMany Mapping

In Doctrine-entity-land, all the relationships are setup. The FortuneCookie has a ManyToOne relationship on a category property:

176 lines | src/AppBundle/Entity/FortuneCookie.php

```
... lines 1 - 12
class FortuneCookie
{
... lines 15 - 23
    /**
     * @var Category
     *
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="fortuneCookies")
     * @ORM\JoinColumn(nullable=false)
     */
    private $category;
... lines 31 - 174
}
```

And inside Category, we have the inverse side of the relationship: a OneToMany on a property called fortuneCookies:

113 lines | src/AppBundle/Entity/Category.php

```
... lines 1 - 13
class Category
{
... lines 16 - 38
    /**
     * @ORM\OneToMany(targetEntity="FortuneCookie", mappedBy="category")
     */
    private $fortuneCookies;
... lines 43 - 111
}
```

Mapping this side of the relationship is optional, but *we'll* need it to do our query.

## Adding the leftJoin Query

Let's go add our LEFT JOIN to the query builder! If you're thinking there's a leftJoin method, winner! And this time, we *are* going to use it. Join on cat.fortuneCookies. Why fortuneCookies? Because this is the *name* of the property on Category for this relationship.

The second argument to leftJoin() is the alias we want to give to FortuneCookie, fc::

36 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 23
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
... lines 27 - 29
            ->leftJoin('cat.fortuneCookies', 'fc')
... lines 31 - 33
    }
... lines 35 - 36
```

And right away, we can see why Doctrine JOINs are so easy, I mean confusing, I mean easy. This is *all* we need for a JOIN - we don't have any of the LEFT JOIN ON fortune_cookie.categoryId = category.id kind of stuff. Sure, this will be in the final query, but *we* don't need to worry about that stuff because Doctrine already knows how to join across this relationship: all the details it needs are in the relationship's annotations.

The cat.fortuneCookies thing *only* works because we have the fortuneCookies OneToMany side of the relationship. Adding this mapping for the inverse side is optional, but if we didn't have it, we'd need to add it right now: our query depends on it.

LEFT JOIN, check! And just like normal SQL, we can use the fc alias from the joined table to update the WHERE clause. I'll break this onto multiple lines for my personal sanity and then add OR fc.fortune LIKE :searchTerm because fortune is the name of the property on FortuneCookie that holds the message:

36 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 23
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
            ->andWhere('cat.name LIKE :searchTerm
                OR cat.iconKey LIKE :searchTerm
                OR fc.fortune LIKE :searchTerm')
            ->leftJoin('cat.fortuneCookies', 'fc')
            ->setParameter('searchTerm', '%'.$term.'%')
            ->getQuery()
            ->execute();
    }
... lines 35 - 36
```

Moment of truth! We've got a match! Our fortunes are being searched.

## JOINing, but Querying for the same Data

Even though we now have a LEFT JOIN, the result of the query is no different: it still returns an array of Category objects. We *can* and *will* do some JOINs in the future that actually *select* data from the joined table. But if all you do is JOIN like we're doing here, it doesn't change the data that's returned.

# Chapter 5: Joins and addSelect Reduce Queries

There's a problem with our page! Sorry, I'll stop panicking - it's not a *huge* deal, but if you look at the bottom, two queries are being executed. That's strange: the only query I remember making is inside my FortuneController when we call search().

Click the web debug toolbar to see what the queries are. Ah, the first I recognize: that's our search query. But the second one is something different. Look closely: it's querying for all the fortune_cookie rows that are related to this category:

```
SELECT t0.* FROM fortune_cookie t0
   WHERE t0.category_id = 4;
```

If you've heard about "lazy loading" of relationships, you probably know what this comes from. The query is actually coming from our template. We loop over the array of Category object, and then print category.fortuneCookies|length:

```
19 lines | app/Resources/views/fortune/homepage.html.twig
... lines 1 - 5
    {% for category in categories %}
... lines 7 - 9
                <span class="fa {{ category.iconKey }}"></span> {{ category.name }}  ({{ category.fortuneCookies|length }})
... lines 11 - 15
    {% endfor %}
... lines 17 - 19
```

The Category object has *all* of the data for itself, but at this point, it hasn't yet fetched the data of the FortuneCookie's that it's related to in the database. So at the moment we call category.fortuneCookies and try to count the results, it goes out and does a query for all of the fortune_cookie rows for this category. That's called a lazy query, because it was lazy and waited to be executed until we actually needed that data.

This "extra query" isn't the end of the world. In fact, I don't usually fix it until I'm working on my page's performance. On the homepage without a search, it's even *more* noticeable. We have *7* queries here: one for the categories, and one extra query to get the fortune cookies for *each* category in the list. That makes 2, 3, 4, 5, 6 and 7 queries. This is a classic problem called the *n+1 problem*.

And again, it's not the end of the world - so don't over-optimize. But let's fix it here.

## Reducing Queries with addSelect

Back in CategoryRepository, once we've joined over to our fortuneCookies, we can say ->addSelect('fc'):

```
37 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 23
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
... lines 27 - 29
            ->leftJoin('cat.fortuneCookies', 'fc')
            ->addSelect('fc')
... lines 32 - 34
    }
... lines 36 - 37
```

And just by doing that, our second query is gone! It's black magic - don't worry about how it works! You know I'm kidding, here's the deal. Remember that when we call $this->createQueryBuilder() from inside a repository class, that automatically selects everything from the Category. So it's equivalent to calling ->select('cat'). Calling addSelect() means that we're going to select all the Category information *and* all the FortuneCookie information.

## addSelect and the Return Value

There's one super-important thing to keep in mind: even though we're selecting more data, this function returns the *exact* same thing it did before: an array of Category objects. That's different from SQL, where selecting all the fields from a joined table will give you more fields in your result. Here, addSelect() just tells Doctrine to fetch the FortuneCookie data, but store it internally. Later, when we access the FortuneCookies for a Category, it's smart enough to know that it doesn't need that second query. So we can reduce the number of queries used without needing to go update any other code: this function still returns an array of Categories.

## Adding addSelect to findAllOrdered()

Go back to the homepage without a search. Dang, we still have those 7 ugly queries. And that's because this uses a different method: findAllOrdered. Let's to the same thing here. ->leftJoin('cat.fortuneCookies', 'fc') and then an addSelect('fc'):

```
39 lines | src/AppBundle/Entity/CategoryRepository.php
   ... lines 1 - 39
```

Our two queries start to have some duplication. That's an issue we'll fix later. We're hoping to see our 7 queries drop *all* the way to 1 - the one query for all of the Categories. Perfect!

## Adding addSelect to find()

We're on a roll! Click into a category - like Proverbs. Here, we have *two* queries. This is the same problem - query #1 is the one we're doing in our controller. Query #2 comes lazily from the template, where we're looping over all the fortune cookies for the category.

We're using the built-in find() method in the controller:

```
53 lines | src/AppBundle/Controller/FortuneController.php
   ... lines 1 - 35
    public function showCategoryAction($id)
    {
   ... lines 38 - 41
        $category = $categoryRepository->find($id);
   ... lines 43 - 50
    }
   ... lines 52 - 53
```

But since it doesn't let us do any joins, we need to do something more custom. Call a new method findWithFortunesJoin. You know the drill: we'll go into CategoryRepository and then add that method. And at this point, this should be a really easy query. I'll copy the search() function, then simplify things in the andWhere: cat.id = :id. We want to keep the leftJoin() and the addSelect to remove the extra query. Update setParameter to set the id placeholder:

```
50 lines | src/AppBundle/Entity/CategoryRepository.php
   ... lines 1 - 12
class CategoryRepository extends EntityRepository
{
   ... lines 15 - 38
    public function findWithFortunesJoin($id)
    {
        return $this->createQueryBuilder('cat')
            ->andWhere('cat.id = :id')
            ->leftJoin('cat.fortuneCookies', 'fc')
            ->addSelect('fc')
            ->setParameter('id', $id)
            ->getQuery()
            ->getOneOrNullResult();
    }
}
```

The execute() function returns an array of results, but in this case, we want just *one* Category object, or null if there isn't one. So we'll use the other function I talked about to finish the query: getOneOrNullResult().

Refresh! Two queries is now 1.

Exactly like in SQL, JOINs have two purposes. Sometimes you JOIN because you want to add a WHERE clause or an ORDER BY on the data on that JOIN'ed table. The second reason to JOIN is that you *actually* want to SELECT data from the table. In Doctrine, this second reason feels a little different because even though we're SELECTing from the fortune_cookie table, the query still returns the same array of Category objects as before. But Doctrine has that extra data in the background.

But this doesn't *have* to be the case. Over the next two chapters, we'll start SELECT'ing individual fields, instead of entire objects.

# Chapter 6: SELECT the SUM (or COUNT)

In every query so far, Doctrine gives us objects. That's its default mode, but we can also easily use it to select specific fields.

On our category page, you can see how many of each fortune has been printed over time. At the top, let's total those numbers with a SUM() query and print it out.

In showCategoryAction(), create a new variable - $fortunesPrinted, that'll be a number. And of course, we'll write a new query to get this. But instead of shoving this into CategoryRepository, this queries the FortuneCookie entity, so we'll use its repository instead. So, AppBundle:FortuneCookie, and we'll call a new countNumberPrintedForCategory method. Pass the $category object as an argument:

59 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 9
class FortuneController extends Controller
{
... lines 12 - 35
    public function showCategoryAction($id)
    {
... lines 38 - 47
        $fortunesPrinted = $this->getDoctrine()
            ->getRepository('AppBundle:FortuneCookie')
            ->countNumberPrintedForCategory($category);
... lines 51 - 56
    }
}
```

This will return the raw, summed number. To actually use this, pass this into the template:

59 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 35
    public function showCategoryAction($id)
    {
... lines 38 - 47
        $fortunesPrinted = $this->getDoctrine()
            ->getRepository('AppBundle:FortuneCookie')
            ->countNumberPrintedForCategory($category);
... lines 51 - 52
        return $this->render('fortune/showCategory.html.twig',[
            'category' => $category,
            'fortunesPrinted' => $fortunesPrinted,
        ]);
    }
... lines 58 - 59
```

And now print it out: {{ fortunesPrinted }}, put that through Twig's number_format filter and add the word "total":

36 lines | app/Resources/views/fortune/showCategory.html.twig

```
... lines 1 - 12
                    <th>
                        Printed History ({{ fortunesPrinted|number_format }} total)
                    </th>
... lines 16 - 36
```

Amazing. You already know the next step: we need to create a new countNumberPrintedForCategory method inside of FortuneCookieRepository and make it query not for an object, but just a single number: the sum of how many times each fortune has been printed. That means we'll be totaling the numberPrinted property on FortuneCookie:

176 lines | src/AppBundle/Entity/FortuneCookie.php

```
    ... lines 1 - 12
class FortuneCookie
{
    ... lines 15 - 45
    /**
     * @var integer
     *
     * @ORM\Column(name="numberPrinted", type="integer")
     */
    private $numberPrinted;
    ... lines 52 - 174
}
```

Open FortuneCookieRepository and add the new public function. We're expecting a Category object, so I'll type-hint the argument like a nice, respectable programmer. Every query starts the same: $this->createQueryBuilder() and we'll use fc as the alias. Keep the alias consistent for an entity, it'll save you heartache later.

Next, we need an andWhere() because we need to only find FortuneCookie results for this Category. So, fc.category - because category is the name of the property on FortuneCookie for the relationship. Now, equals :category. And next, we'll set that parameter:

26 lines | src/AppBundle/Entity/FortuneCookieRepository.php

```
    ... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
            ->andWhere('fc.category = :category')
            ->setParameter('category', $category)
    ... lines 20 - 22
    }
    ... lines 24 - 26
```

This looks like every query we've made before, and if we finish now, it'll return FortuneCookie objects. That's lame - I want just the sum number. To do this, call select(). Nope, this is *not* addSelect like we used before. When we call createQueryBuilder from inside FortuneCookieRepository, the query builder has a ->select('fc') built into it. In other words, it's selecting everything from FortuneCookie. Calling select() clears out anything that's being selected and replaces it with our SUM(fc.numberPrinted) as fortunesPrinted:

26 lines | src/AppBundle/Entity/FortuneCookieRepository.php

```
    ... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
            ->andWhere('fc.category = :category')
            ->setParameter('category', $category)
            ->select('SUM(fc.numberPrinted) as fortunesPrinted')
    ... lines 21 - 22
    }
    ... lines 24 - 26
```

We're giving the value an alias, just like you can do in SQL. Now, instead of an object, we're getting back a single field. Let's finish it! Add getQuery().

Last, should we call execute() or getOneOrNullResult()? If you think about the query in SQL, this will return a single row that has the fortunesPrinted value. So we want to return just one result - use getOneOrNullResult():

26 lines | src/AppBundle/Entity/FortuneCookieRepository.php

```
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
            ->andWhere('fc.category = :category')
            ->setParameter('category', $category)
            ->select('SUM(fc.numberPrinted) as fortunesPrinted')
            ->getQuery()
            ->getOneOrNullResult();
    }
... lines 24 - 26
```

Love it! I'm curious to see what this query returns, so let's var_dump $fortunesPrinted inside our controller:

59 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 35
    public function showCategoryAction($id)
    {
... lines 38 - 47
        $fortunesPrinted = $this->getDoctrine()
            ->getRepository('AppBundle:FortuneCookie')
            ->countNumberPrintedForCategory($category);
        var_dump($fortunesPrinted);die;
... lines 52 - 56
    }
... lines 58 - 59
```

Refresh! It's just what you'd expect: an array with a single key called fortunesPrinted. So the $fortunesPrinted variable isn't *quite* a number - it's this array with a key on it. But let me show you a trick. I told you about execute() and getOneOrNullResult(): the first returns many results, the second returns a single result or null. But if you're returning a single row that has only a single column, instead of getOneOrNullResult(), you can say getSingleScalarResult():

26 lines | src/AppBundle/Entity/FortuneCookieRepository.php

```
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
            ->andWhere('fc.category = :category')
            ->setParameter('category', $category)
            ->select('SUM(fc.numberPrinted) as fortunesPrinted')
            ->getQuery()
            ->getSingleScalarResult();
    }
... lines 24 - 26
```

This says: ok, you're only returning one row with one column, let me just give you that value directly. This is really handy for SUMs and COUNTs.

Refresh! Hey, we have *just* the number! Time to celebrate - take out the var_dump, refresh and... great success! So not only can we select specific fields instead of getting back objects, if you're selecting just one field on one row, getSingleScalarResult() is your new friend.

# Chapter 7: Selecting Specific Fields

We're on a roll! Let's select more fields - like an average of the numberPrinted *and* the name of the category, all in one query. Yea yea, we *already* have the Category's name over here - we're querying for the entire Category object. But just stick with me - it makes for a good example.

Head back to FortuneCookieRepository. As I hope you're guessing, SELECTing more fields is just like SQL: add a comma and get them. You could also use the addSelect() function if you want to get fancy.

Add, AVG(fc.numberPrinted) and give that an alias - fortunesAverage. I'm just making that up. Let's *also* grab cat.name - the name of the category that we're using here:

26 lines | src/AppBundle/Entity/FortuneCookieRepository.php

```
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
            ->andWhere('fc.category = :category')
            ->setParameter('category', $category)
            ->select('SUM(fc.numberPrinted) as fortunesPrinted, AVG(fc.numberPrinted) as fortunesAverage, cat.name')
            ->getQuery()
            ->getSingleScalarResult();
    }
... lines 24 - 26
```

I don't trust myself. So, var_dump($fortunesPrinted) in the controller with our trusty die statement:

59 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 35
    public function showCategoryAction($id)
    {
... lines 38 - 47
        $fortunesPrinted = $this->getDoctrine()
            ->getRepository('AppBundle:FortuneCookie')
            ->countNumberPrintedForCategory($category);
        var_dump($fortunesPrinted);die;
... lines 52 - 56
    }
... lines 58 - 59
```

Refresh! Uh oh, that's an awesome error:

```
[Semantical Error] line 0, col 88 near 'cat.name FROM': Error 'cat'
is not defined.
```

## Debugging Bad DQL Queries

This is what it looks like when you mess up your DQL. Doctrine does a really good job of lexing and parsing the DQL you give it, so when you make a mistake, it'll give you a pretty detailed error. Here, cat is not defined is because our query references cat with cat.name, but I haven't made any JOINs to create a cat alias. cat is not defined.

But real quick - go back to the error. If you scroll down the stack trace a little, you'll eventually see the full query:

```
SELECT SUM(fc.numberPrinted) as fortunesPrinted,
   AVG(fc.numberPrinted) fortunesAverage,
   cat.name
   FROM AppBundle\Entity\FortuneCookie fc
   WHERE fc.category = :category
```

For me, sometimes the top error is so small, it doesn't make sense. But if I look at it in context of the full query, it's a lot easier to figure out what mistake I made.

Fixing our error is easy: we need to add a JOIN - this time an innerJoin(). So, innerJoin('fc.category', 'cat'):

```
27 lines | src/AppBundle/Entity/FortuneCookieRepository.php
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
... lines 18 - 19
            ->innerJoin('fc.category', 'cat')
            ->select('SUM(fc.numberPrinted) as fortunesPrinted, AVG(fc.numberPrinted) as fortunesAverage, cat.name')
... lines 22 - 23
    }
... lines 25 - 27
```

Why fc.category? Because in the FortuneCookie entity, we have a category property. That's how it knows which relationship we're talking about. So cat is now aliased! Let's try again.

Ooook, another error: NonUniqueResultException. We're still finishing the query with getSingleScalarResult(). But now that we're returning multiple columns of data, it doesn't make sense anymore. The NonUniqueResultException means that you either have this situation, or, more commonly, you're using getOneOrNullResult(), but your query is returning mulitple rows. Watch out for that.

Change the query to getOneOrNullResult(): the query still returns only one row, but multiple columns:

```
27 lines | src/AppBundle/Entity/FortuneCookieRepository.php
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
... lines 18 - 21
            ->getQuery()
            ->getOneOrNullResult();
    }
... lines 25 - 27
```

Refresh! Beautiful! The result is an associative array with fortunesPrinted, fortunesAverage and name keys. And notice, we didn't give the category name an alias in the query - we didn't say as something, so it just used name by default:

```
27 lines | src/AppBundle/Entity/FortuneCookieRepository.php
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        return $this->createQueryBuilder('fc')
... lines 18 - 20
            ->select('SUM(fc.numberPrinted) as fortunesPrinted, AVG(fc.numberPrinted) as fortunesAverage, cat.name')
... lines 22 - 23
    }
... lines 25 - 27
```

And hey, I was even a bit messy: for the sum I said as fortunesPrinted but for the average, I just said fortunesAverage with the as. The as is optional - I didn't leave it out on purpose, but hey, good learning moment.

The query is beautiful, so let's actually use our data. In the controller, change the result from $fortunesPrinted to $fortunesData - it's really an array. And below, set $fortunesPrinted to $fortunesData['...']. I'll check my query to remember the alias - it's fortunesPrinted, so I'll use that. I'll do the same thing for the other two fields:

63 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 35
    public function showCategoryAction($id)
    {
... lines 38 - 47
        $fortunesData = $this->getDoctrine()
            ->getRepository('AppBundle:FortuneCookie')
            ->countNumberPrintedForCategory($category);
        $fortunesPrinted = $fortunesData['fortunesPrinted'];
        $averagePrinted = $fortunesData['fortunesAverage'];
        $categoryName = $fortunesData['name'];
... lines 54 - 60
    }
... lines 62 - 63
```

The alias for the average is fortunesAverage. And the last one just uses name. Let's pass these into the template:

63 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 35
    public function showCategoryAction($id)
    {
... lines 38 - 54
        return $this->render('fortune/showCategory.html.twig',[
            'category' => $category,
            'fortunesPrinted' => $fortunesPrinted,
            'averagePrinted' => $averagePrinted,
            'categoryName' => $categoryName
        ]);
    }
... lines 62 - 63
```

And again, I know, the categoryName is redundant - we already have the whole category object. But to prove things, use categoryName in the template. And below, add an extra line after the total and print averagePrinted:

37 lines | app/Resources/views/fortune/showCategory.html.twig

```
... lines 1 - 37
```

Moment of truth! Woot! 244,829 total, 81,610 average, and the category name still prints out. Doctrine normally queries for objects, and that's great! But remember, nothing stops you from using that select() function to say: no no no: I don't want to select objects anymore, I want to select specific fields.

# Chapter 8: Raw SQL Queries

All this Doctrine entity object stuff and DQL queries are really great. But if you ever feel overwhelmed with all of this or need write a *really* complex query, you can always fall back to using raw SQL. Seriously, this is *huge*. When Doctrine scares you, you are *totally* free to run away and use plain SQL. I won't judge you - just get that feature finished and launch it already. If a tool isn't helping you, don't use it.

Put on your DBA hat and let's write some SQL!

Open up FortuneCookieRepository. This is where we built a query that selected a sum, an average and the category name.

## The DBAL Connection

The most important object in Doctrine is.... the entity manager! But that's just a puppet for the *real* guy in charge: the connection! Grab it by getting the entity manager and calling getConnection(). Let's var_dump() this:

32 lines | src/AppBundle/Entity/FortuneCookieRepository.php

```
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        $conn = $this->getEntityManager()
            ->getConnection();

        var_dump($conn);die;
... lines 21 - 28
    }
... lines 30 - 32
```

Head to the browser and click into one of the category pages. There's our beautiful var_dump(). Hey! Look at the class name:

```
Doctrine\DBAL\Connection
```

Fun fact! The Doctrine library is actually *multiple* libraries put together. The two parts we care about are the ORM and the DBAL. The ORM is what does all the magic mapping of data onto objects. The DBAL - or database abstraction layer - can be used *completely* independent of the ORM. It's basically a wrapper around PDO. Said in a less boring way, it's a library for executing database queries.

So this DBAL Connection objects is *our* key to running raw database queries. Google for "Doctrine DBAL Query" so we can follow its docs. Find the Data Retrieval And Manipulaton section. Scroll down a little to a good example:

```
$sql = "SELECT * FROM articles WHERE id = ?";
$stmt = $conn->prepare($sql);
$stmt->bindValue(1, $id);
$stmt->execute();
```

This DBAL library is a really light wrapper around PHP's PDO. So if you've used that before, you'll like this. But if not, it's like 3 steps, so stick with me.

## Making a Raw SQL Query

Back in FortuneCookieRepository, let's write some simple SQL to test with:

```
SELECT * FROM fortune_cookie;
```

When you use the DBAL, there are *no* entities and it doesn't know about any of our Doctrine annotations. Yep, we're talking to the raw tables and columns. So I used fortune_cookies because that's the name of the actual table in the database.

Next, we'll use the SQL to get a statement. So:

```
$stmt = $conn->prepare($sql);
```

And then we can execute() that, which runs the query but doesn't give you the result. To get *that*, call $stmt->fetchAll() and var_dump() that:

```
35 lines | src/AppBundle/Entity/FortuneCookieRepository.php
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        $conn = $this->getEntityManager()
            ->getConnection();

        $sql = 'SELECT * FROM fortune_cookie';
        $stmt = $conn->prepare($sql);
        $stmt->execute();
        var_dump($stmt->fetchAll());die;
... lines 24 - 31
    }
... lines 33 - 35
```

Try it! And there it is: exactly what you'd expect with no effort at all. It's literally the results - in array format - from the raw SQL query. Doctrine isn't trying to hide this feature from you - just grab the Connection object and you're dangerous.

## Prepared Statements

The query we made with the query builder is a bit more complex. Could we replacle that with raw SQL? Sure! Well there's not really a good reason to do this, since it's built and working. But let's prove we can do it!

Let's grab the "select" part of the query and stick that in our query. I hate long lines, so let's use multiple. Piece by piece, add the other query parts. The FROM is fortune_cookie fc. Add the INNER JOIN to category ON cat.id = fc.category_id. And since we're in DBAL land, we don't have any of our annotation mapping configuration, so we have to tell it exactly how to join - it's just raw SQL. And for the same reason, we're using the *real* column names, like category_id.

Add a single WHERE of fc.category_id = :category. That's some good-old-fashioned boring SQL. I love it! The only thing we *still* need to do is fill in the :category placeholder. Even though we're using the DBAL, we still don't concatenate strings in our queries, unless you love SQL attacks or prefer to live dangerously. Are you feeling lucky, punk?

Ahem. To give :category a value, just pass an array to execute() and pass it a category key assigned to the id. Ok, done! Let's dump this!

```
40 lines | src/AppBundle/Entity/FortuneCookieRepository.php
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        $conn = $this->getEntityManager()
            ->getConnection();

        $sql = '
            SELECT SUM(fc.numberPrinted) as fortunesPrinted, AVG(fc.numberPrinted) as fortunesAverage, cat.name
            FROM fortune_cookie fc
            INNER JOIN category cat ON cat.id = fc.category_id
            WHERE fc.category_id = :category
            ';
        $stmt = $conn->prepare($sql);
        $stmt->execute(array('category' => $category->getId()));
        var_dump($stmt->fetchAll());die;
... lines 29 - 36
    }
... lines 38 - 40
```

Boom! That's *exactly* what I was hoping for.

## Using fetch() to get back the First Row

Since our SQL gives us just *one* row, it'd be awesome to get just *its* columns, instead of an array with one result. Just use fetch()!

```
40 lines | src/AppBundle/Entity/FortuneCookieRepository.php
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        $conn = $this->getEntityManager()
            ->getConnection();

        $sql = '
            SELECT SUM(fc.numberPrinted) as fortunesPrinted, AVG(fc.numberPrinted) as fortunesAverage, cat.name
            FROM fortune_cookie fc
            INNER JOIN category cat ON cat.id = fc.category_id
            WHERE fc.category_id = :category
            ';
        $stmt = $conn->prepare($sql);
        $stmt->execute(array('category' => $category->getId()));
        var_dump($stmt->fetch());die;
... lines 29 - 36
    }
... lines 38 - 40
```

And now, this is exactly what our query builder gave us before. So get rid of the die() statement and return the fetch() line:

```
41 lines | src/AppBundle/Entity/FortuneCookieRepository.php
... lines 1 - 14
    public function countNumberPrintedForCategory(Category $category)
    {
        $conn = $this->getEntityManager()
            ->getConnection();

        $sql = '
            SELECT SUM(fc.numberPrinted) as fortunesPrinted, AVG(fc.numberPrinted) as fortunesAverage, cat.name
            FROM fortune_cookie fc
            INNER JOIN category cat ON cat.id = fc.category_id
            WHERE fc.category_id = :category
            ';
        $stmt = $conn->prepare($sql);
        $stmt->execute(array('category' => $category->getId()));

        return $stmt->fetch();
... lines 30 - 37
    }
... lines 39 - 41
```

Just let the old code sit down there. Refresh! And we're prefectly back to normal. Man, that was kinda easy. So if Doctrine ever looks hard or you're still learning it, totally use SQL. It's no big deal.

## Native Queries?

One slightly confusing thing is that if you google for "doctrine raw sql", you'll find a different solution - something called NativeQuery. It sort of looks the same, just with some different function names. But there's this ResultSetMapping thing. Huh. This NativeQuery thing allows you to run a raw SQL query and then map that *back* to an object. That's pretty neat I guess. But for me, if I'm writing some custom SQL, I'm fine just getting back an array of data. I can deal with that. The ResultSetMapping confuses me, and probably isn't worth the effort. But it's there if you want to geek out on it.

# Chapter 9: Reusing Queries with the Query Builder

Enough with all this SQL stuff. Remember the query builder I was raving about earlier? I promised that one of its benefits is that, with a query builder, you can re-use parts of a query. But we don't have any of that right now.

Open up CategoryRepository. We have three methods, and *all* of them repeat the same leftJoin() to cat.fortuneCookies and the addSelect():

50 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 14
    public function findAllOrdered()
    {
        $qb = $this->createQueryBuilder('cat')
... line 18
            ->leftJoin('cat.fortuneCookies', 'fc')
            ->addSelect('fc');
... lines 21 - 23
    }
... line 25
    public function search($term)
    {
        return $this->createQueryBuilder('cat')
... lines 29 - 31
            ->leftJoin('cat.fortuneCookies', 'fc')
            ->addSelect('fc')
... lines 34 - 36
    }
... line 38
    public function findWithFortunesJoin($id)
    {
        return $this->createQueryBuilder('cat')
... line 42
            ->leftJoin('cat.fortuneCookies', 'fc')
            ->addSelect('fc')
... lines 45 - 47
    }
... lines 49 - 50
```

Ah, duplication! When you see duplication like this - whether it's a WHERE, an ORDER BY or a JOIN - there's a simple solution. Just add a new private function and have *it* add this stuff to the query builder.

## Query-modifying Functions

Create a private function called addFortuneCookieJoinAndSelect(), because that's what it's going to do! It'll accept a QueryBuilder as an argument. Our goal is to, well, add the join to that. So I'll copy the 2 pieces that we want, add a $qb, then paste it there. And just for convenience, let's return this too:

```
66 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 52
    /**
     * Joins over to cat.fortuneCookies AND selects its fields
     *
     * @param QueryBuilder $qb
     * @return QueryBuilder
     */
    private function addFortuneCookieJoinAndSelect(QueryBuilder $qb)
    {
        return $qb->leftJoin('cat.fortuneCookies', 'fc')
            ->addSelect('fc');
    }
... lines 64 - 66
```

So, our function takes in a QueryBuilder, it modifies it, then it returns it so we can make any more changes. I'll be a *nice* programmer and add some PHPDoc.

## Calling those Functions

The findAllOrdered() function is the one that fuels the homepage. So let's start here! Get rid of that duplicated leftJoin and addSelect. Instead, just call $this->addFortuneCookieJoinAndSelect() and pass it the $qb. So *we* create the query builder, do some things with it, but let our new function take care of the join stuff.

```
66 lines | src/AppBundle/Entity/CategoryRepository.php
... lines 1 - 15
    public function findAllOrdered()
    {
        $qb = $this->createQueryBuilder('cat')
            ->addOrderBy('cat.name', 'ASC');
        $this->addFortuneCookieJoinAndSelect($qb);

        $query = $qb->getQuery();

        return $query->execute();
    }
... lines 26 - 66
```

This *should* give us the exact same results. But you should never believe me, so let's go back and refresh the homepage. Yep, nice!

Now we get to celebrate by removing the rest of the duplication. So, addSelect and leftJoin should be gone. Instead of returning the result directly, we need to get a QueryBuilder first. So put $qb = in front and move the getQuery() stuff down and put the return in front of it. In the middle, call addFortuneCookieJoinAndSelect() like before:

66 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 26
    public function search($term)
    {
        $qb = $this->createQueryBuilder('cat')
            ->andWhere('cat.name LIKE :searchTerm
                OR cat.iconKey LIKE :searchTerm
                OR fc.fortune LIKE :searchTerm');
        $this->addFortuneCookieJoinAndSelect($qb);

        return $qb
            ->setParameter('searchTerm', '%'.$term.'%')
            ->getQuery()
            ->execute();
    }
... lines 40 - 66
```

And one more time in findWithFortunesJoin(). Remove the duplication, create a $qb variable, return the last part of the query, and stick our magic line in the middle::

66 lines | src/AppBundle/Entity/CategoryRepository.php

```
... lines 1 - 39

    public function findWithFortunesJoin($id)
    {
        $qb = $this->createQueryBuilder('cat')
            ->andWhere('cat.id = :id');
        $this->addFortuneCookieJoinAndSelect($qb);

        return $qb
            ->setParameter('id', $id)
            ->getQuery()
            ->getOneOrNullResult();
    }
... lines 52 - 66
```

Try it! Refresh and click into a category. It all works. And you know, I feel a lot better. If there's one things I don't want to duplicate, it's query logic. I hope this looks really obvious to you - it's just a simple coding technique. But it's kind of amazing, because it's not something you can do easily with string queries. And it can *really* save you if once you've got complex WHERE clauses that need to be re-used. You don't want to screw that stuff up.

# Chapter 10: Filters

I setup my fixtures so that about half of my FortuneCookies have a "discontinued" value of true. For our startup fortune cookie company, that means we don't make them anymore. But we're not showing this information anywhere on the frontend yet.

But what if we wanted to *only* show fortune cookies on the site that we're still making? In other words, where discontinued is false. Yes yes, I know. This is easy. We could just go into CategoryRepository and add some andWhere() calls in here for fc.discontinued = true.

But what if we wanted this WHERE clause to be added automatically, and everywhere across the site? That's possible, and it's called a Doctrine Filter.

## Creating the Filter Class

Let's start by creating the filter class itself. Create a new directory called Doctrine. There's no real reason for that, just keeping organized. In there, create a new class called DiscontinuedFilter, and make sure we put it in the right namespace:

8 lines | src/AppBundle/Doctrine/DiscontinuedFilter.php

```php
<?php

namespace AppBundle\Doctrine;

class DiscontinuedFilter
{
}
```

That's a nice blank class. To find out what goes inside, Google for "Doctrine Filters" to get into their documentation. These filter classes are simple: just extend the SQLFilter class, and that'll force us to have one method. So let's do that - extends SQLFilter. My IDE is *angry* because SQLFilter has an abstract method we need to add. I'll use PHPStorm's Code->Generate shortcut and choose "Implement Methods". It does the work of adding that addFilterConstraint method for me. And for some reason, it's extra generous and gives me an extra ClassMetadata use statement, so I'll take that out.

23 lines | src/AppBundle/Doctrine/DiscontinuedFilter.php

```php
... lines 1 - 2
namespace AppBundle\Doctrine;

use Doctrine\ORM\Mapping\ClassMetadata;
use Doctrine\ORM\Query\Filter\SQLFilter;

class DiscontinuedFilter extends SQLFilter
{
    /**
     * Gets the SQL query part to add to a query.
     *
     * @param ClassMetaData $targetEntity
     * @param string $targetTableAlias
     *
     * @return string The constraint SQL if there is available, empty string otherwise.
     */
    public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias)
    {
        // ...
    }
}
```

Ok, here's how this works. *If* this filter is enabled - and we'll talk about that - the addFilterConstraint() method will be called on every query. And this is *our* chance to add a WHERE clause to it. The $targetEntity argument is information about which entity we're querying for. Let's dump that to test that the method is called, and to see what that looks like:

23 lines | src/AppBundle/Doctrine/DiscontinuedFilter.php

```
... lines 1 - 7
class DiscontinuedFilter extends SQLFilter
{
... lines 10 - 17
    public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias)
    {
        var_dump($targetEntity);die;
    }
}
```

## Adding the Filter

Next, Doctrine somehow has to know about this class. If you're using Doctrine outside of Symfony, you'll use its Configuration object and call addFilter on it:

```
// from http://doctrine-orm.readthedocs.org/en/latest/reference/filters.html#configuration
$config->addFilter('locale', '\Doctrine\Tests\ORM\Functional\MyLocaleFilter');
```

You pass it the class name and some "key" - locale in their example. This becomes its nickname, and we'll refer to the filter later by this key.

In Symfony, we need the same, but it's done with configuration. Open up app/config/config.yml and find the doctrine spot, and under orm, add filters:. On the next line, go out four spaces, make up a key for the filter - I'll say fortune_cookie_discontinued and set that to the class name: AppBundle\Doctrine\DiscontinuedFilter:

76 lines | app/config/config.yml

```
... lines 1 - 46
doctrine:
... lines 48 - 62
    orm:
... lines 64 - 65
        filters:
            fortune_cookie_discontinued: AppBundle\Doctrine\DiscontinuedFilter
... lines 68 - 76
```

Awesome - now Doctrine knows about our filter.

## Enabling a Filter

But if you refresh the homepage, nothing! We do *not* hit our die statement. Ok, so adding a filter to Doctrine is 2 steps. First, you say "Hey Doctrine, this filter exists!" We just did that. Second, you need to *enable* the filter. That ends up being nice, because it means you can enable or disable a filter on different parts of your site.

Open up FortuneController. Let's enable the filter on our homepage. Yes yes, we *are* going to enable this filter globally for the site later. Just stay tuned.

To enable it here, first, get the EntityManager. And I'm going to add a comment, which will help with auto-completion on the next steps:

```
69 lines | src/AppBundle/Controller/FortuneController.php
... lines 1 - 10
class FortuneController extends Controller
{
... lines 13 - 15
    public function homepageAction(Request $request)
    {
        /** @var EntityManager $em */
        $em = $this->getDoctrine()->getManager();
... lines 20 - 36
    }
... lines 38 - 67
}
```

Once you have the entity manager, call getFilters() on it, then enable(). The argument to enable() needs to be whatever nickname you gave the filter before. Actually, I have a typo in mine - I'll fix that now. Copy the fortune_cookie_discontinued string and pass it to enable():

```
69 lines | src/AppBundle/Controller/FortuneController.php
... lines 1 - 15
    public function homepageAction(Request $request)
    {
        /** @var EntityManager $em */
        $em = $this->getDoctrine()->getManager();
        $em->getFilters()
            ->enable('fortune_cookie_discontinued');
... lines 22 - 36
    }
... lines 38 - 69
```

Filter class, check! Filter register, check! Filter enabled, check. Moment of truth. Refresh! And there's our dumped ClassMetadata.

## Adding the Filter Logic

We haven't put anything in DiscontinuedFilter yet, but most of the work is done. That ClassMetadata argument is your best friend: this is the Doctrine object that knows *everything* about the entity we're querying for. You can read your annotation mapping config, get details on associations, find out about the primary key and anything else your heart desires.

Now, this method will be called for *every* query. But we *only* want to add our filtering logic if the query is for a FortuneCookie. To do that, add: if, $targetEntity->getReflectionClass() - that's the PHP ReflectionClass object, ->name() != AppBundle\Entity\FortuneCookie, then we're going to return an empty string:

```
27 lines | src/AppBundle/Doctrine/DiscontinuedFilter.php
... lines 1 - 7
class DiscontinuedFilter extends SQLFilter
{
... lines 10 - 17
    public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias)
    {
        if ($targetEntity->getReflectionClass()->name != 'AppBundle\Entity\FortuneCookie') {
            return '';
        }
... lines 23 - 24
    }
}
```

It's *gotta* be an empty string. That tells Doctrine: hey, I don't want to add any WHERE clauses here - so just leave it alone. If you return null, it adds the WHERE but doesn't put anything in it.

Below this, it's our time to shine. We're going to return what you want in the WHERE clause. So we'll use sprintf, then %s. This will be the table alias - I'll show you in a second. Then, .discontinued = false. This is the string part of what we normally put in an andWhere() with the query builder. To fill in the %s, pass in $targetTableAlias:

27 lines | src/AppBundle/Doctrine/DiscontinuedFilter.php

```
... lines 1 - 17
    public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias)
    {
        if ($targetEntity->getReflectionClass()->name != 'AppBundle\Entity\FortuneCookie') {
            return '';
        }

        return sprintf('%s.discontinued = false', $targetTableAlias);
    }
... lines 26 - 27
```

Remember how every entity in a query has an alias? We usually call createQueryBuilder() and pass it something like fc. That's the alias. In this case, Doctrine is *telling* us what the alias is so we can use it.

Alright. Refresh! Um ok, no errors. But it's also not obvious if this is working. So look at the number of fortune cookies in each category: 1, 2, 3, 3, 3, 4. Go back to FortuneController and delete the enable() call. Refresh again. Ah hah! All the numbers went *up* a little. Our filter is working.

Put the enable() call back and refresh again. Click the database icon on the web debug toolbar. You can see in the query that when we LEFT JOIN to fortune_cookie, it added this f1_.discontinued = false.

Woh woh woh. This is more amazing than I've been promising. Even though our query is for Category's, it was smart enough to apply the filter when it joined over to FortuneCookie. Because of this, when we call Category::getFortuneCookies(), that's *only* going to have the ones that are *not* discontinued. The filter is applied if the fortune cookie shows up *anywhere* in our query.

## Passing Values to/Configuring a Filter

Sometimes, like in an admin area, we might want to show only *discontinued* fortune cookies. So can we control the value we're passing in the filter? To do this, remove false and add another %s. Add another argument to sprintf: $this->getParameter('discontinued'):

27 lines | src/AppBundle/Doctrine/DiscontinuedFilter.php

```
... lines 1 - 17
    public function addFilterConstraint(ClassMetadata $targetEntity, $targetTableAlias)
    {
... lines 20 - 23
        return sprintf('%s.discontinued = %s', $targetTableAlias, $this->getParameter('discontinued'));
    }
... lines 26 - 27
```

This is kind of like the parameters we use in the query builder, except instead of using :discontinued, we concatenate it into the string. But wait! Won't this make SQL injection attacks possible! I hope you were yelling that :). But with filters, it's ok because getParameter() automatically adds the escaping. So, it's no worry.

If we *just* did this and refreshed, we've got a great error!

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

This new approach means that when we enable the filter, we need to pass this value to it. In FortuneController, the enable() method actually returns an instance of our DiscontinuedFilter. And now we can call setParameter(), with the parameter name as the first argument and the value we want to set it to as the second:

70 lines | src/AppBundle/Controller/FortuneController.php

```
... lines 1 - 15
    public function homepageAction(Request $request)
    {
        /** @var EntityManager $em */
        $em = $this->getDoctrine()->getManager();
        $filters = $em->getFilters()
            ->enable('fortune_cookie_discontinued');
        $filters->setParameter('discontinued', false);
... lines 23 - 68
}
```

Refresh! We see the slightly-lower cookie numbers. Change that to true and we should see *really* low numbers. We do!

## Enabling a Filter Globally

Through all of this, you might be asking: "What good is a filter if I need to enable it all the time." Well first, the nice thing about filters is that you *do* have this ability to enable or disable them if you need to.

To enable a filter globally, you just need to follow these same steps in the bootstrap of your app. To hook into the beginning process of Symfony, we'll need an event listener.

I did the hard-work already and created a class called BeforeRequestListener:

19 lines | src/AppBundle/EventListener/BeforeRequestListener.php

```
... lines 1 - 7
class BeforeRequestListener
{
    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }

    public function onKernelRequest(GetResponseEvent $event)
    {
        // ...
    }
}
```

For Symfony peeps, you'll recognize the code in my services.yml:

10 lines | app/config/services.yml

```
services:
    before_request_listener:
        class: AppBundle\EventListener\BeforeRequestListener
        arguments: ["@doctrine.orm.entity_manager"]
        tags:
            -
                name: kernel.event_listener
                event: kernel.request
                method: onKernelRequest
```

It registers this as a service and the tags at the bottom says, "Hey, when Symfony boots, like right at the very beginning, call the onKernelRequest method." I'm also passing the EntityManager as the first argument to the __construct() function. Because, ya know, we need that to enable filters.

Let's go steal the enabling code from FortuneController, take it all out and paste it into onKernelRequest. Instead of simply $em, we have $this->em, since it's set on a property:

23 lines | src/AppBundle/EventListener/BeforeRequestListener.php

```
... lines 1 - 14
    public function onKernelRequest(GetResponseEvent $event)
    {
        $filter = $this->em
            ->getFilters()
            ->enable('fortune_cookie_discontinued');
        $filter->setParameter('discontinued', false);
    }
... lines 22 - 23
```

Let's try it! Even though we took the enable() code out of the controller, the numbers don't change: our filter is still working. If we click into "Proverbs", we see only 1. But if I disable the filter, we see all 3.

That's it! You're dangerous. If you've ever built a multi-tenant site where almost *every* query has a filter, life just got easy.