Databases and the Doctrine ORM

Version: current Edit this page

Screencast

Do you prefer video tutorials? Check out the Doctrine screencast series.

Symfony provides all the tools you need to use databases in your applications thanks to Doctrine, the best set of PHP libraries to work with databases. These tools support relational databases like MySQL and PostgreSQL and also NoSQL databases like MongoDB.

Databases are a broad topic, so the documentation is divided in three articles:

- This article explains the recommended way to work with **relational databases** in Symfony applications;
- Read this other article if you need **low-level access** to perform raw SQL queries to relational databases (similar to PHP's PDO);
- Read DoctrineMongoDBBundle docs if you are working with **MongoDB databases**.

Installing Doctrine

First, install Doctrine support via the `orm` Symfony pack, as well as the MakerBundle, which will help generate some code:

```
$ composer require symfony/orm-pack
$ composer require --dev symfony/maker-bundle
```

Configuring the Database

The database connection information is stored as an environment variable called `DATABASE_URL`. For development, you can find and customize this inside `.env`:

```
# .env (or override DATABASE_URL in .env.local to avoid committing your changes)

# customize this line!
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"

# to use mariadb:
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=mariadb-10.5.8"

# to use sqlite:
# DATABASE_URL="sqlite:///%kernel.project_dir%/var/app.db"

# to use postgresql:
# DATABASE_URL="postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=11&charset=utf8"

# to use oracle:
# DATABASE_URL="oci8://db_user:db_password@127.0.0.1:1521/db_name"
```

Caution

If the username, password, host or database name contain any character considered special in a URI (such as `+`, `@`, `$`, `#`, `/`, `:`, `*`, `!`), you must encode them. See RFC 3986 for the full list of reserved characters or use the urlencode function to encode them. In this case you need to remove the `resolve:` prefix in `config/packages/doctrine.yaml` to avoid errors: `url: '%env(resolve:DATABASE_URL)%'`

Now that your connection parameters are setup, Doctrine can create the `db_name` database for you:

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

There are more options in `config/packages/doctrine.yaml` that you can configure, including your `server_version` (e.g. 5.7 if you're using MySQL 5.7), which may affect how Doctrine functions.

Tip

There are many other Doctrine commands. Run `php bin/console list doctrine` to see a full list.

Creating an Entity Class

Suppose you're building an application where products need to be displayed. Without even thinking about Doctrine or databases, you already know that you need a `Product` object to represent those products.

You can use the `make:entity` command to create this class and any fields you need. The command will ask you some questions - answer them like done below:

```
$ php bin/console make:entity

Class name of the entity to create or update:
> Product

New property name (press <return> to stop adding fields):
> name

Field type (enter ? to see all types) [string]:
> string

Field length [255]:
> 255

Can this field be null in the database (nullable) (yes/no) [no]:
> no

New property name (press <return> to stop adding fields):
> price

Field type (enter ? to see all types) [string]:
> integer

Can this field be null in the database (nullable) (yes/no) [no]:
> no

New property name (press <return> to stop adding fields):
>
```

1.3 The interactive behavior of the `make:entity` command was introduced in MakerBundle 1.3.

Woh! You now have a new `src/Entity/Product.php` file:

```
// src/Entity/Product.php
namespace App\Entity;

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

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

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

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

    // ... getter and setter methods
}
```

Note

Confused why the price is an integer? Don't worry: this is just an example. But, storing prices as integers (e.g. 100 = $1 USD) can avoid rounding issues.

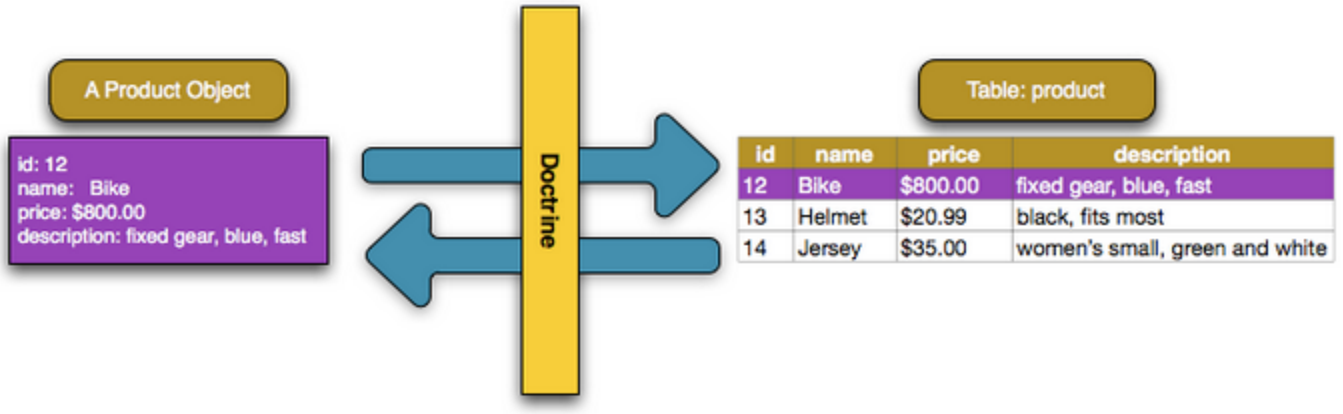
Note

If you are using an SQLite database, you'll see the following error: *PDOException: SQLSTATE[HY000]: General error: 1 Cannot add a NOT NULL column with default value NULL*. Add a `nullable=true` option to the `description` property to fix the problem.

Caution

There is a limit of 767 bytes for the index key prefix when using InnoDB tables in MySQL 5.6 and earlier versions. String columns with 255 character length and `utf8mb4` encoding surpass that limit. This means that any column of type `string` and `unique=true` must set its maximum `length` to `190`. Otherwise, you'll see this error: "*[PDOException] SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 767 bytes*".

This class is called an "entity". And soon, you'll be able to save and query Product objects to a `product` table in your database. Each property in the `Product` entity can be mapped to a column in that table. This is usually done with annotations: the `@ORM\...` comments that you see above each property:



id	name	price	description
12	Bike	$800.00	fixed gear, blue, fast
13	Helmet	$20.99	black, fits most
14	Jersey	$35.00	women's small, green and white

The `make:entity` command is a tool to make life easier. But this is *your* code: add/remove fields, add/remove methods or update configuration.

Doctrine supports a wide variety of field types, each with their own options. To see a full list, check out Doctrine's Mapping Types documentation. If you want to use XML instead of annotations, add `type: xml` and `dir: '%kernel.project_dir%/config/doctrine'` to the entity mappings in your `config/packages/doctrine.yaml` file.

Caution

Be careful not to use reserved SQL keywords as your table or column names (e.g. `GROUP` or `USER`). See Doctrine's Reserved SQL keywords documentation for details on how to escape these. Or, change the table name with `@ORM\Table(name="groups")` above the class or configure the column name with the `name="group_name"` option.

Migrations: Creating the Database Tables/Schema

The `Product` class is fully-configured and ready to save to a `product` table. If you just defined this class, your database doesn't actually have the `product` table yet. To add it, you can leverage the DoctrineMigrationsBundle, which is already installed:

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

If everything worked, you should see something like this:

```
SUCCESS!

Next: Review the new migration "migrations/Version20211116204726.php"
Then: Run the migration with php bin/console doctrine:migrations:migrate
```

If you open this file, it contains the SQL needed to update your database! To run that SQL, execute your migrations:

```
$ php bin/console doctrine:migrations:migrate
```

This command executes all migration files that have not already been run against your database. You should run this command on production when you deploy to keep your production database up-to-date.

Migrations & Adding more Fields

But what if you need to add a new field property to `Product`, like a `description`? You can edit the class to add the new property. But, you can also use `make:entity` again:

```
$ php bin/console make:entity

Class name of the entity to create or update
> Product

New property name (press <return> to stop adding fields):
> description

Field type (enter ? to see all types) [string]:
> text

Can this field be null in the database (nullable) (yes/no) [no]:
> no

New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

This adds the new `description` property and `getDescription()` and `setDescription()` methods:

```
// src/Entity/Product.php
  // ...

  class Product
  {
      // ...

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

      // getDescription() & setDescription() were also added
  }
```

The new property is mapped, but it doesn't exist yet in the `product` table. No problem! Generate a new migration:

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

This time, the SQL in the generated file will look like this:

```
ALTER TABLE product ADD description LONGTEXT NOT NULL
```

The migration system is *smart*. It compares all of your entities with the current state of the database and generates the SQL needed to synchronize them! Like before, execute your migrations:

```
$ php bin/console doctrine:migrations:migrate
```

This will only execute the *one* new migration file, because DoctrineMigrationsBundle knows that the first migration was already executed earlier. Behind the scenes, it manages a `migration_versions` table to track this.

Each time you make a change to your schema, run these two commands to generate the migration and then execute it. Be sure to commit the migration files and execute them when you deploy.

Tip

If you prefer to add new properties manually, the `make:entity` command can generate the getter & setter methods for you:

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

If you make some changes and want to regenerate *all* getter/setter methods, also pass `--overwrite`.

Persisting Objects to the Database

It's time to save a `Product` object to the database! Let's create a new controller to experiment:

```
$ php bin/console make:controller ProductController
```

Inside the controller, you can create a new `Product` object, set data on it, and save it:

```php
// src/Controller/ProductController.php
namespace App\Controller;

// ...
use App\Entity\Product;
use Doctrine\Persistence\ManagerRegistry;
use Symfony\Component\HttpFoundation\Response;

class ProductController extends AbstractController
{
    /**
     * @Route("/product", name="create_product")
     */
    public function createProduct(ManagerRegistry $doctrine): Response
    {
        $entityManager = $doctrine->getManager();

        $product = new Product();
        $product->setName('Keyboard');
        $product->setPrice(1999);
        $product->setDescription('Ergonomic and stylish!');

        // tell Doctrine you want to (eventually) save the Product (no queries yet)
        $entityManager->persist($product);

        // actually executes the queries (i.e. the INSERT query)
        $entityManager->flush();

```

Try it out!

http://localhost:8000/product

Congratulations! You just created your first row in the `product` table. To prove it, you can query the database directly:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM product'

# on Windows systems not using Powershell, run this command instead:
# php bin/console doctrine:query:sql "SELECT * FROM product"
```

Take a look at the previous example in more detail:

- **line 14** The `ManagerRegistry $doctrine` argument tells Symfony to inject the Doctrine service into the controller method.
- **line 16** The `$doctrine->getManager()` method gets Doctrine's *entity manager* object, which is the most important object in Doctrine. It's responsible for saving objects to, and fetching objects from, the database.
- **lines 20-23** In this section, you instantiate and work with the `$product` object like any other normal PHP object.
- **line 26** The `persist($product)` call tells Doctrine to "manage" the `$product` object. This does **not** cause a query to be made to the database.

- **line 29** When the `flush()` method is called, Doctrine looks through all of the objects that it's managing to see if they need to be persisted to the database. In this example, the `$product` object's data doesn't exist in the database, so the entity manager executes an `INSERT` query, creating a new row in the `product` table.

Note

If the `flush()` call fails, a `Doctrine\ORM\ORMException` exception is thrown. See Transactions and Concurrency.

Whether you're creating or updating objects, the workflow is always the same: Doctrine is smart enough to know if it should INSERT or UPDATE your entity.

Validating Objects

The Symfony validator reuses Doctrine metadata to perform some basic validation tasks:

```php
// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Validator\Validator\ValidatorInterface;
// ...

class ProductController extends AbstractController
{
    /**
     * @Route("/product", name="create_product")
     */
    public function createProduct(ValidatorInterface $validator): Response
    {
        $product = new Product();
        // This will trigger an error: the column isn't nullable in the database
        $product->setName(null);
        // This will trigger a type mismatch error: an integer is expected
        $product->setPrice('1999');

        // ...

        $errors = $validator->validate($product);
        if (count($errors) > 0) {
            return new Response((string) $errors, 400);
        }

        //
```

Although the `Product` entity doesn't define any explicit validation configuration, Symfony introspects the Doctrine mapping configuration to infer some validation rules. For example, given that the `name` property can't be `null` in the database, a NotNull constraint is added automatically to the property (if it doesn't contain that constraint already).

The following table summarizes the mapping between Doctrine metadata and the corresponding validation constraints added automatically by Symfony:

Doctrine attribute	Validation constraint	Notes
`nullable=false`	NotNull	Requires installing the PropertyInfo component
`type`	Type	Requires installing the PropertyInfo component
`unique=true`	UniqueEntity	
`length`	Length	

Because the Form component as well as API Platform internally use the Validator component, all your forms and web APIs will also automatically benefit from these automatic validation constraints.

This automatic validation is a nice feature to improve your productivity, but it doesn't replace the validation configuration entirely. You still need to add some validation constraints to ensure that data provided by the user is correct.

Fetching Objects from the Database

Fetching an object back out of the database is even easier. Suppose you want to be able to go to `/product/1` to see your new product:

```php
// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

class ProductController extends AbstractController
{
    /**
     * @Route("/product/{id}", name="product_show")
     */
    public function show(ManagerRegistry $doctrine, int $id): Response
    {
        $product = $doctrine->getRepository(Product::class)->find($id);

        if (!$product) {
            throw $this->createNotFoundException(
                'No product found for id '.$id
            );
        }

        return new Response('Check out this great product: '.$product->getName());

        // or render a template
        // in the template, print things with {{ product.name }}
        // return $this->render('product/show.html.twig', ['product' => $product]);
    }
```

Another possibility is to use the `ProductRepository` using Symfony's autowiring and injected by the dependency injection container:

```php
// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use App\Repository\ProductRepository;
use Symfony\Component\HttpFoundation\Response;
// ...

class ProductController extends AbstractController
{
    /**
     * @Route("/product/{id}", name="product_show")
     */
    public function show(int $id, ProductRepository $productRepository): Response
    {
        $product = $productRepository
            ->find($id);

        // ...
    }
}
```

Try it out!

http://localhost:8000/product/1

When you query for a particular type of object, you always use what's known as its "repository". You can think of a repository as a PHP class whose only job is to help you fetch entities of a certain class.

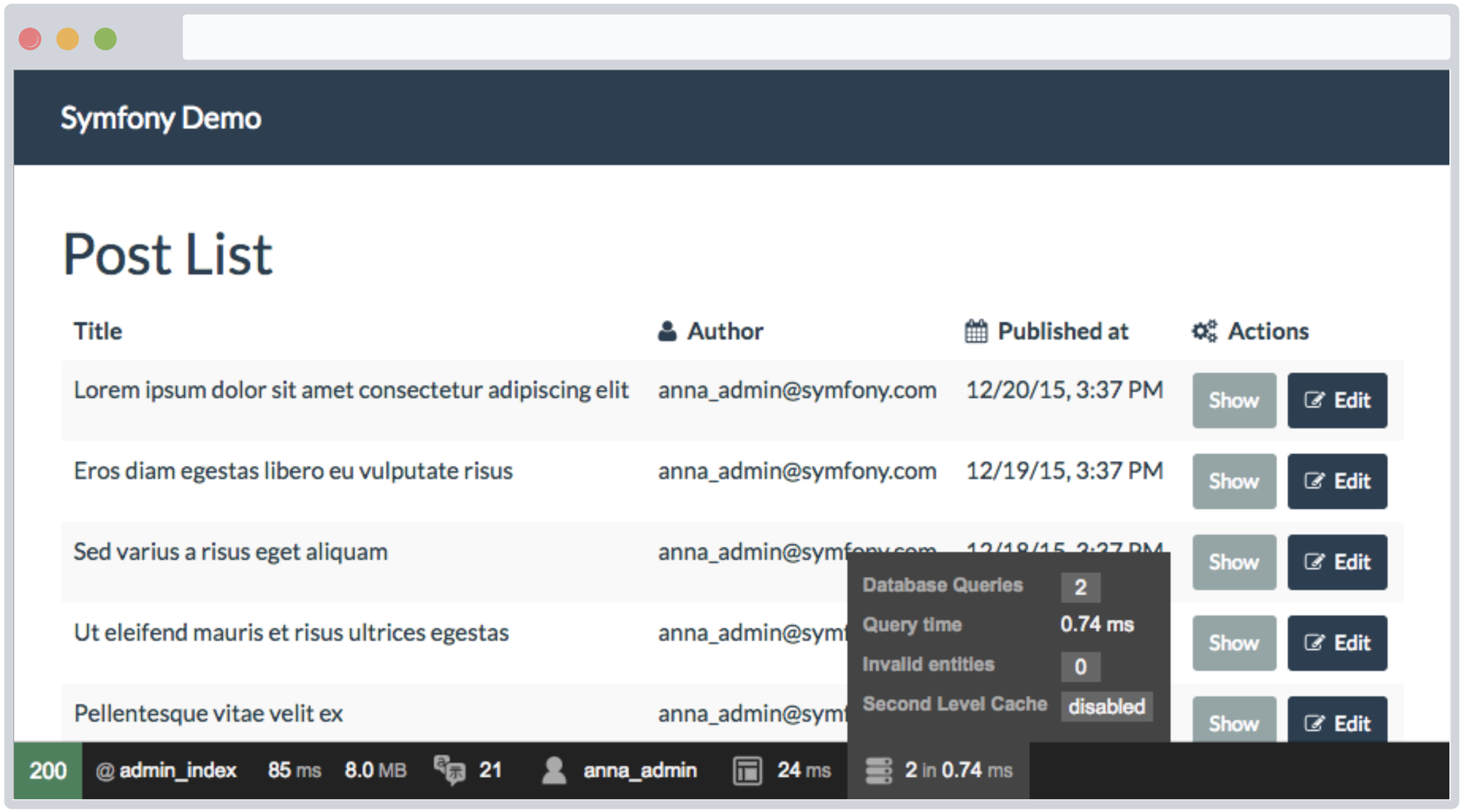
Once you have a repository object, you have many helper methods:

```
$repository = $doctrine->getRepository(Product::class);

// look for a single Product by its primary key (usually "id")
$product = $repository->find($id);

// look for a single Product by name
$product = $repository->findOneBy(['name' => 'Keyboard']);
// or find by name and price
$product = $repository->findOneBy([
    'name' => 'Keyboard',
    'price' => 1999,
]);

// look for multiple Product objects matching the name, ordered by price
$products = $repository->findBy(
    ['name' => 'Keyboard'],
    ['price' => 'ASC']
);

// look for *all* Product objects
$products = $repository->findAll();
```

You can also add *custom* methods for more complex queries! More on that later in the Databases and the Doctrine ORM section.

Tip

When rendering an HTML page, the web debug toolbar at the bottom of the page will display the number of queries and the time it took to execute them:



Symfony Demo

Post List

Title	Author	Published at	Actions
Lorem ipsum dolor sit amet consectetur adipiscing elit	anna_admin@symfony.com	12/20/15, 3:37 PM	Show Edit
Eros diam egestas libero eu vulputate risus	anna_admin@symfony.com	12/19/15, 3:37 PM	Show Edit
Sed varius a risus eget aliquam	anna_admin@symfony.com	12/18/15, 3:37 PM	Show Edit
Ut eleifend mauris et risus ultrices egestas	anna_admin@sym		Show Edit
Pellentesque vitae velit ex	anna_admin@sym		Show Edit

Database Queries	2
Query time	0.74 ms
Invalid entities	0
Second Level Cache	disabled

```
200  @ admin_index  85 ms  8.0 MB  21  anna_admin  24 ms  2 in 0.74 ms
```

If the number of database queries is too high, the icon will turn yellow to indicate that something may not be correct. Click on the icon to open the Symfony Profiler and see the exact queries that were executed. If you don't see the web debug toolbar, install the `profiler` Symfony pack by running this command: `composer` `require --dev symfony/profiler-pack`.

Automatically Fetching Objects (ParamConverter)

In many cases, you can use the SensioFrameworkExtraBundle to do the query for you automatically! First, install the bundle in case you don't have it:

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

Now, simplify your controller:

```php
// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use App\Repository\ProductRepository;
use Symfony\Component\HttpFoundation\Response;
// ...

class ProductController extends AbstractController
{
    /**
     * @Route("/product/{id}", name="product_show")
     */
    public function show(Product $product): Response
    {
        // use the Product!
        // ...
    }
}
```

That's it! The bundle uses the `{id}` from the route to query for the `Product` by the `id` column. If it's not found, a 404 page is generated.

There are many more options you can use. Read more about the ParamConverter.

Updating an Object

Once you've fetched an object from Doctrine, you interact with it the same as with any PHP model:

```php
// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use App\Repository\ProductRepository;
use Symfony\Component\HttpFoundation\Response;
// ...

class ProductController extends AbstractController
{
    /**
     * @Route("/product/edit/{id}")
     */
    public function update(ManagerRegistry $doctrine, int $id): Response
    {
        $entityManager = $doctrine->getManager();
        $product = $entityManager->getRepository(Product::class)->find($id);

        if (!$product) {
            throw $this->createNotFoundException(
                'No product found for id '.$id
            );
        }

        $product->setName('New product name!');
        $entityManager->flush();

        return $this->redirectToRoute('product_show', [
```

Using Doctrine to edit an existing product consists of three steps:

1. fetching the object from Doctrine;
2. modifying the object;
3. calling `flush()` on the entity manager.

You *can* call `$entityManager->persist($product)`, but it isn't necessary: Doctrine is already "watching" your object for changes.

Deleting an Object

Deleting an object is very similar, but requires a call to the `remove()` method of the entity manager:

```
$entityManager->remove($product);
$entityManager->flush();
```

As you might expect, the `remove()` method notifies Doctrine that you'd like to remove the given object from the database. The `DELETE` query isn't actually executed until the `flush()` method is called.

Querying for Objects: The Repository

You've already seen how the repository object allows you to run basic queries without any work:

```php
// from inside a controller
$repository = $doctrine->getRepository(Product::class);
$product = $repository->find($id);
```

But what if you need a more complex query? When you generated your entity with `make:entity`, the command *also* generated a `ProductRepository` class:

```php
// src/Repository/ProductRepository.php
namespace App\Repository;

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

class ProductRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Product::class);
    }
}
```

When you fetch your repository (i.e. `->getRepository(Product::class)`), it is *actually* an instance of *this* object! This is because of the `repositoryClass` config that was generated at the top of your `Product` entity class.

Suppose you want to query for all Product objects greater than a certain price. Add a new method for this to your repository:

```php
// src/Repository/ProductRepository.php

// ...
class ProductRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Product::class);
    }

    /**
     * @return Product[]
     */
    public function findAllGreaterThanPrice(int $price): array
    {
        $entityManager = $this->getEntityManager();

        $query = $entityManager->createQuery(
            'SELECT p
            FROM App\Entity\Product p
            WHERE p.price > :price
            ORDER BY p.price ASC'
        )->setParameter('price', $price);

        // returns an array of Product objects
        return $query->getResult();
    }
}
```

The string passed to `createQuery()` might look like SQL, but it is Doctrine Query Language. This allows you to type queries using commonly known query language, but referencing PHP objects instead (i.e. in the `FROM` statement).

Now, you can call this method on the repository:

```
// from inside a controller
$minPrice = 1000;

$products = $doctrine->getRepository(Product::class)->findAllGreaterThanPrice($minPrice);

// ...
```

See Service Container for how to inject the repository into any service.

Querying with the Query Builder

Doctrine also provides a Query Builder, an object-oriented way to write queries. It is recommended to use this when queries are built dynamically (i.e. based on PHP conditions):

```php
// src/Repository/ProductRepository.php

// ...
class ProductRepository extends ServiceEntityRepository
{
    public function findAllGreaterThanPrice(int $price, bool $includeUnavailableProducts = false): arr
    {
        // automatically knows to select Products
        // the "p" is an alias you'll use in the rest of the query
        $qb = $this->createQueryBuilder('p')
            ->where('p.price > :price')
            ->setParameter('price', $price)
            ->orderBy('p.price', 'ASC');

        if (!$includeUnavailableProducts) {
            $qb->andWhere('p.available = TRUE');
        }

        $query = $qb->getQuery();

        return $query->execute();

        // to get just one result:
        // $product = $query->setMaxResults(1)->getOneOrNullResult();
    }
}
```

Querying with SQL

In addition, you can query directly with SQL if you need to:

```php
// src/Repository/ProductRepository.php

// ...
class ProductRepository extends ServiceEntityRepository
{
    public function findAllGreaterThanPrice(int $price): array
    {
        $conn = $this->getEntityManager()->getConnection();

        $sql = '
            SELECT * FROM product p
            WHERE p.price > :price
            ORDER BY p.price ASC
            ';
        $stmt = $conn->prepare($sql);
        $stmt->execute(['price' => $price]);

        // returns an array of arrays (i.e. a raw data set)
        return $stmt->fetchAllAssociative();
    }
}
```

With SQL, you will get back raw data, not objects (unless you use the NativeQuery functionality).

Configuration

See the Doctrine config reference.

Relationships and Associations

Doctrine provides all the functionality you need to manage database relationships (also known as associations), including ManyToOne, OneToMany, OneToOne and ManyToMany relationships.

For info, see How to Work with Doctrine Associations / Relations.

Database Testing

Read the article about testing code that interacts with the database.

Doctrine Extensions (Timestampable, Translatable, etc.)

Doctrine community has created some extensions to implement common needs such as "*set the value of the createdAt property automatically when creating an entity*". Read more about the available Doctrine extensions and use the StofDoctrineExtensionsBundle to integrate them in your application.

Learn more

# How to Work with Doctrine Associations / Relations

- # Doctrine Events
- # How to Implement a Registration Form
- # How to Register custom DQL Functions
- # How to Use Doctrine DBAL
- # How to Work with multiple Entity Managers and Connections
- # How to Define Relationships with Abstract Classes and Interfaces
- # How to Generate Entities from an Existing Database
- # Store Sessions in a Database
- # How to Test A Doctrine Repository

Symfony Conferences

SymfonyWorld Online 2022 Summer Edition
Jun 16–17, 2022

SymfonyLive Paris 2022
Apr 7–8, 2022

What is Symfony?

Symfony at a Glance
Symfony Components
Case Studies
Symfony Releases
Security Policy
Logo & Screenshots
Trademark & Licenses
symfony1 Legacy

Learn Symfony

Getting Started
Components
Best Practices
Bundles
Reference
Training
eLearning Platform
Certification

Screencasts

Learn Symfony
Learn PHP
Learn JavaScript
Learn Drupal
Learn RESTful APIs

Community

SymfonyConnect
Support
How to be Involved
Code of Conduct
Events & Meetups
Projects using Symfony
Downloads Stats
Contributors
Backers

Blog

Events & Meetups
A week of symfony
Case studies
Cloud
Community
Conferences
Diversity
Documentation
Living on the edge
Releases
Security Advisories
SymfonyInsight
Twig
SensioLabs

Services

SensioLabs services
Train developers
Manage your project quality
Improve your project performance
Host Symfony projects

Deployed on

SymfonyCloud

Follow Symfony

Casts

Dynamic (same as OS)