Symfony lab6

$ php bin/console make:entity Category

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):

>

(press enter again to finish)

This will generate your new entity class:

<?php

// src/Entity/Category.php

namespace App\Entity;

use App\Repository\CategoryRepository;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity(repositoryClass=CategoryRepository::class)

\*/

class Category

{

/\*\*

\* @ORM\Id

\* @ORM\GeneratedValue

\* @ORM\Column(type="integer")

\*/

private $id;

/\*\*

\* @ORM\Column(type="string", length=255)

\*/

private $name;

public function getId(): ?int

{

return $this->id;

}

public function getName(): ?string

{

return $this->name;

}

public function setName(string $name): self

{

$this->name = $name;

return $this;

}

}

[**Mapping the ManyToOne Relationship**](https://symfony.com/doc/current/doctrine/associations.html#mapping-the-manytoone-relationship)

In this example, each category can be associated with *many* products. But, each product can be associated with only *one* category. This relationship can be summarized as: *many* products to *one* category (or equivalently, *one* category to *many* products).

From the perspective of the Product entity, this is a many-to-one relationship. From the perspective of the Category entity, this is a one-to-many relationship.

To map this, first create a category property on the Product class with the ManyToOne annotation. You can do this by hand, or by using the make:entity command, which will ask you several questions about your relationship. If you're not sure of the answer, don't worry! You can always change the settings later:

$ php bin/console make:entity

Class name of the entity to create or update (e.g. BraveChef):

> Product

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

> category

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

> relation

What class should this entity be related to?:

> Category

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:

> ManyToOne

Is the Product.category property allowed to be null (nullable)? (yes/no) [yes]:

> no

Do you want to add a new property to Category so that you can access/update

Product objects from it - e.g. $category->getProducts()? (yes/no) [yes]:

> yes

New field name inside Category [products]:

> products

Do you want to automatically delete orphaned App\Entity\Product objects

(orphanRemoval)? (yes/no) [no]:

> no

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

>

(press enter again to finish)

This made changes to *two* entities. First, it added a new category property to the Product entity (and getter & setter methods):

<?php

// 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;

/\*\*

\* @ORM\ManyToOne(targetEntity=Category::class, inversedBy="products")

\* @ORM\JoinColumn(nullable=false)

\*/

private $category;

public function getId(): ?int

{

return $this->id;

}

public function getName(): ?string

{

return $this->name;

}

public function setName(string $name): self

{

$this->name = $name;

return $this;

}

public function getPrice(): ?int

{

return $this->price;

}

public function setPrice(int $price): self

{

$this->price = $price;

return $this;

}

public function getCategory(): ?Category

{

return $this->category;

}

public function setCategory(?Category $category): self

{

$this->category = $category;

return $this;

}

}

This ManyToOne mapping is required. It tells Doctrine to use the category\_id column on the product table to relate each record in that table with a record in the category table.

Next, since one Category object will relate to many Product objects, the make:entity command also added a products property to the Category class that will hold these objects:

<?php

// src/Entity/Category.php

namespace App\Entity;

use App\Repository\CategoryRepository;

use Doctrine\ORM\Mapping as ORM;

use Doctrine\Common\Collections\ArrayCollection;

use Doctrine\Common\Collections\Collection;

/\*\*

\* @ORM\Entity(repositoryClass=CategoryRepository::class)

\*/

class Category

{

/\*\*

\* @ORM\Id

\* @ORM\GeneratedValue

\* @ORM\Column(type="integer")

\*/

private $id;

/\*\*

\* @ORM\Column(type="string", length=255)

\*/

private $name;

public function getId(): ?int

{

return $this->id;

}

public function getName(): ?string

{

return $this->name;

}

public function setName(string $name): self

{

$this->name = $name;

return $this;

}

/\*\*

\* @ORM\OneToMany(targetEntity="App\Entity\Product", mappedBy="category")

\*/

private $products;

public function \_\_construct()

{

$this->products = new ArrayCollection();

}

/\*\*

\* @return Collection|Product[]

\*/

public function getProducts(): Collection

{

return $this->products;

}

public function addProduct(Product $product): self

{

if (!$this->products->contains($product)) {

$this->products[] = $product;

$product->setCategory($this);

}

return $this;

}

public function removeProduct(Product $product): self

{

if ($this->products->removeElement($product)) {

// set the owning side to null (unless already changed)

if ($product->getCategory() === $this) {

$product->setCategory(null);

}

}

return $this;

}

}

$ php bin/console doctrine:migrations:diff

$ php bin/console doctrine:migrations:migrate

## [Saving Related Entities](https://symfony.com/doc/current/doctrine/associations.html#saving-related-entities)

Now you can see this new code in action! Imagine you're inside a controller:

<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\Routing\Annotation\Route;

use App\Entity\Category;

use App\Entity\Product;

use Doctrine\ORM\EntityManagerInterface;

class ProductController extends AbstractController

{

/\*\*

\* @Route("/product", name="create\_product")

\*/

public function index(): Response

{

$category = new Category();

$category->setName('Computer Peripherals');

$product = new Product();

$product->setName('Keyboard');

$product->setPrice(19.99);

// relates this product to the category

$product->setCategory($category);

$entityManager = $this->getDoctrine()->getManager();

$entityManager->persist($category);

$entityManager->persist($product);

$entityManager->flush();

return new Response(

'Saved new product with id: '.$product->getId()

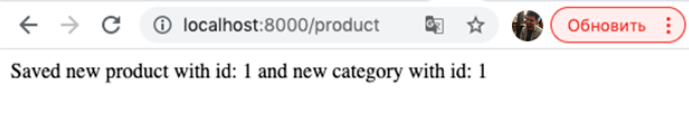
.' and new category with id: '.$category->getId()

);

}

}

When you go to /product, a single row is added to both the category and product tables. The product.category\_id column for the new product is set to whatever the id is of the new category. Doctrine manages the persistence of this relationship for you:



## [Fetching Related Objects](https://symfony.com/doc/current/doctrine/associations.html#fetching-related-objects)

When you need to fetch associated objects, your workflow looks like it did before. First, fetch a $product object and then access its related Category object:

public function show(int $id): Response

{

$product = $this->getDoctrine()

->getRepository(Product::class)

->find($id);

$categoryName = $product->getCategory()->getName();

}

What's important is the fact that you have access to the product's related category, but the category data isn't actually retrieved until you ask for the category (i.e. it's "lazily loaded").

Because we mapped the optional OneToMany side, you can also query in the other direction:

public function showProducts(int $id): Response

{

$category = $this->getDoctrine()

->getRepository(Category::class)

->find($id);

$products = $category->getProducts();

}

/\*\*

\* @Route("/product/{id}", name="product\_show")

\*/

public function show(int $id): Response

{

$product = $this->getDoctrine()

->getRepository(Product::class)

->find($id);

$categoryName = $product->getCategory()->getName();

if (!$product) {

throw $this->createNotFoundException(

'No product found for id '.$id

);

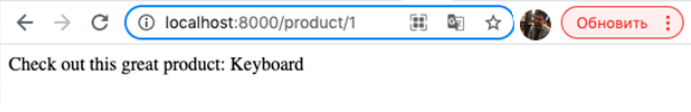
}

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

}



Название продукта по его id



## [Joining Related Records](https://symfony.com/doc/current/doctrine/associations.html#joining-related-records)

In the examples above, two queries were made - one for the original object (e.g. a Category) and one for the related object(s) (e.g. the Product objects).

If you know up front that you'll need to access both objects, you can avoid the second query by issuing a join in the original query. Add the following method to the ProductRepository class:

<?php

// src/Repository/ProductRepository.php

namespace App\Repository;

use App\Entity\Product;

use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;

use Doctrine\Persistence\ManagerRegistry;

/\*\*

\* @method Product|null find($id, $lockMode = null, $lockVersion = null)

\* @method Product|null findOneBy(array $criteria, array $orderBy = null)

\* @method Product[] findAll()

\* @method Product[] findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)

\*/

class ProductRepository extends ServiceEntityRepository

{

public function \_\_construct(ManagerRegistry $registry)

{

parent::\_\_construct($registry, Product::class);

}

// /\*\*

// \* @return Product[] Returns an array of Product objects

// \*/

/\*

public function findByExampleField($value)

{

return $this->createQueryBuilder('p')

->andWhere('p.exampleField = :val')

->setParameter('val', $value)

->orderBy('p.id', 'ASC')

->setMaxResults(10)

->getQuery()

->getResult()

;

}

\*/

/\*

public function findOneBySomeField($value): ?Product

{

return $this->createQueryBuilder('p')

->andWhere('p.exampleField = :val')

->setParameter('val', $value)

->getQuery()

->getOneOrNullResult()

;

}

\*/

public function findOneByIdJoinedToCategory(int $productId): ?Product

{

$entityManager = $this->getEntityManager();

$query = $entityManager->createQuery(

'SELECT p, c

FROM App\Entity\Product p

INNER JOIN p.category c

WHERE p.id = :id'

)->setParameter('id', $productId);

return $query->getOneOrNullResult();

}

}

Thanks to this, if you call $category->removeProduct($product), the category\_id on that Product will be set to null in the database.

But, instead of setting the category\_id to null, what if you want the Product to be deleted if it becomes "orphaned" (i.e. without a Category)? To choose that behavior, use the [orphanRemoval](https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/working-with-associations.html" \l "orphan-removal" \t "_blank) option inside Category:

/\*\*

\* @ORM\OneToMany(targetEntity="App\Entity\Product", mappedBy="category", orphanRemoval=true)

\*/

private $products;