Server-side
Web Development

Unit 12. Symfony. Databases with Doctrine.

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2024-25

Index

1 Doctrine **3**

2 Installation and configuration **3**

3 Creating the database **4**

4 Creating the entities/tables **5**
4.1 Creating the entity 5
4.2 Creating the table 10
4.3 Troubleshooting 11
4.4 How to establish another primary key 11

5 Operating with our database **12**
5.1 Inserting objects to the database 12
5.2 Inserting objects using parameters 15
5.3 Retrieving information from the database 17
5.3.1 Finding an object by its id 17
5.3.2 Finding an object by a property different from the primary key 21
5.3.3 Finding a list of objects by one or more conditions 22
5.3.4 Retrieving all the objects in the table 24
5.3.5 Automatically Fetching Objects 25
5.4 Advanced queries 25
5.5 Deleting objects from the database 27
5.5.1 Deleting an array of objects from the controller 28
5.5.2 Deleting an array of objects from the repository class 28
5.6 Updating objects in the database 29
5.6.1 Updating one object 29
5.6.2 Updating a list of objects 30
5.6.3 Updating a list of objects using the repository class 32

6 Relationships **33**
6.1 1:M relationships 34
6.1.1 Creating the new entity 35
6.1.2 Creating the relationship 38
6.1.3 Managing the relationships from Symfony 40
6.1.4 Adding a new object with a foreign key 42
6.1.5 Retrieving data using the primary key - foreign key link. 44

6.1.6 Adding a new entity in both classes. 47
6.2 Other association types (1:1, M:M) 48

1 Doctrine

In order to work with databases with Symfony, and to make its management easier, we will work with a library called **Doctrine**. Doctrine is an **ORM** (**Object Relational Mapping**), a framework that helps the developer to work with relational databases. The idea is to make a correspondence between our model classes and the database tables. The ORM makes the whole process transparent to the developer. In fact, we will use objects and methods to operate with the database.

2 Installation and configuration

To work with Doctrine and MySQL/MariaDB databases, we first need to install the ORM for our project. In the project folder we type:

The doctrine support should be installed if you have created your project as a full webapp. If not, you can install Doctrine with:

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

Now we need to check if the php drivers for mysql are installed. Try:

```
php -m
```

Check that `mysqli`, `mysqlnd` and `pdo_mysql` appear. If not, go to the **php.ini** file and uncomment the lines corresponding to these drivers.

Just in case the drivers are not installed, install them with:

```
sudo apt-get install php-mysql
sudo apt-get install pdo-mysql
```

Now we go to the **.env.local** file (create it as a copy from **.env** if it doesn't exist) in our project to uncomment the line where the database connection is defined. We have to define the user, the password, our server address, the port, the database name and the version. For instance:

- driver: mysql
- username: root
- password: 1234

- address and port: 127.0.0.1:3306
- database: contacts
- version: MySql 8

So the line will be like this (for MySQL):

```
DATABASE_URL="mysql://root:1234@127.0.0.1:3306/contacts?serverVersion=8&charset=
```

And for MariaDB:

```
DATABASE_URL="mysql://root:1234@127.0.0.1:3306/contacts?serverVersion=10.11.2-
MariaDB&charset=utf8mb4"
```

3 Creating the database

If the database doesn't exist, we can create it from doctrine by doing:

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

There's no need to include the name of the database we are going to create, because it's already included in our DATABASE_URL environment variable. In this case, the database **contacts** will be created.

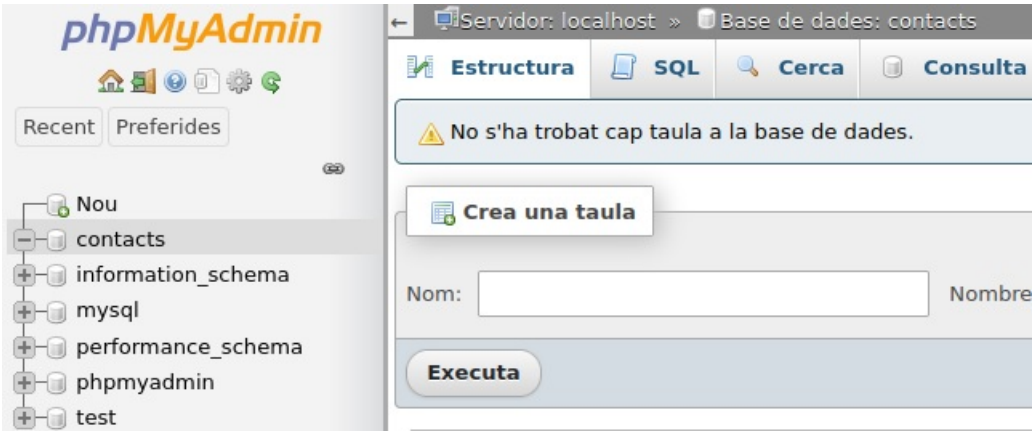

Figure 1: Database created by Doctrine

At this moment, it's recommended that you create a new user instead of root to work with the database.

Another way is to create the database and the user directly from phpMyAdmin or the console (see the guided example of this unit).

4 Creating the entities/tables

The **M** in the MVC design pattern is for the **Model**. Symfony's model component is based on an object/relational mapping layer. What does it mean? It means that PHP and Symfony work with objects, but MySQL/MariaDB work with the relational model, so they manage tables. Doctrine creates the link between the classes and the tables (***object-relational mapping***), providing classes and objects that give access to the database.

Of course, once the class is created, we can append methods that don't match a column in the table. We will go back to this in the next chapters.

4.1 Creating the entity

To create an entity, we use the instruction:

```
symfony console make:entity EntityName
```

Where *TableName* is the name of the entity we want to create.

The command is interactive, so it will guide us through the process. We will add the fields and properties of our entity/table just by answering the questions. Sometimes we will just accept the default value.

Let's create our Contacts entity / table with the name, phone and email attributes (we don't need to create an ***id*** as a primary key: Doctrine will do it automatically):

```
symfony console make:entity Contact
```

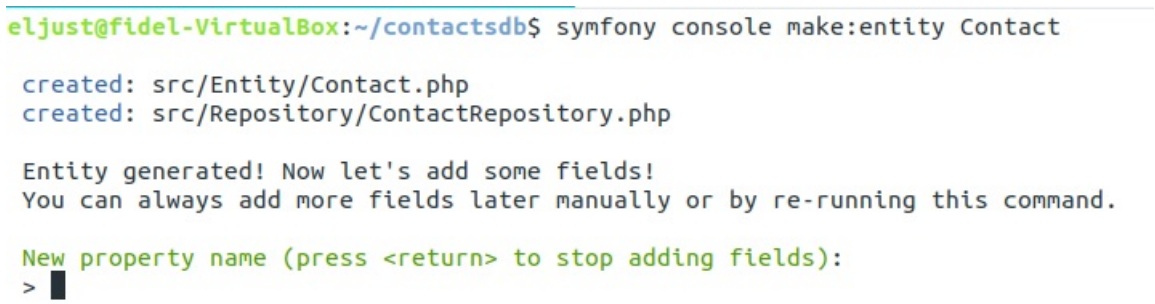

Figure 2: Creating our entity

Doctrine has created two classes:

- Entity/Contact.php (our entity class)

- Repository/ContactRepository.php (our repository class)

We will talk later about the repository utility.

Let's continue with the process of creating the entity. As you can see, Doctrine is asking us to add some fields. If we don't want to add more fields, just press <return> when asked to introduce a new property.

Now we are creating the Contact entity with the name, phone and email.

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

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

Field length [255]:
> 30

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

updated: src/Entity/Contact.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

Figure 3: Creating the property name

Let's proceed the same way with the phone and email attributes:

```
Add another property? Enter the property name (or press <return> to stop adding
fields):
> phone

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

Field length [255]:
> 15

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

updated: src/Entity/Contact.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
> email

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

Field length [255]:
> 30

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

updated: src/Entity/Contact.php

Add another property? Enter the property name (or press <return> to stop adding
fields):
>
```

Figure 4: Adding the rest of the properties

We don't want to add more properties, so we press <return>.

Now we can see our new Contact class created automatically by Doctrine in our `src\Entity` folder:

```php
<?php

namespace App\Entity;

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

#[ORM\Entity(repositoryClass: ContactRepository::class)]
class Contact
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 30)]
    private ?string $name = null;

    #[ORM\Column(length: 15)]
    private ?string $phone = null;

    #[ORM\Column(length: 30, nullable: true)]
    private ?string $email = null;

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getName(): ?string
    {
        return $this->name;
    }

    public function setName(string $name): static
    {
        $this->name = $name;

        return $this;
```

```
    }

    public function getPhone(): ?string
    {
        return $this->phone;
    }

    public function setPhone(string $phone): static
    {
        $this->phone = $phone;

        return $this;
    }

    public function getEmail(): ?string
    {
        return $this->email;
    }

    public function setEmail(?string $email): static
    {
        $this->email = $email;

        return $this;
    }
}
```

As you can see, the class includes the attributes, the getters and setters, along with some annotations. These annotations represent some metadata that Doctrine will use to map the class to its related database table.

But the table doesn't exist yet, so we are going to create it.

> **Note**: In this class you can see how the setters methods return the same static object. This is because they are fluent methods. The fluent methods allow a more fluent way to assign the properties. For example:
>
> `$c = new Contact();`
>
> `$c->setName('John')->setPhone('666554433')->setEmail('john@mail.com');`
>
> More examples about fluent methods.

4.2 Creating the table

Once the entity has been created, we just need to make a migration to the database. Doctrine provides us with a specific tool to initiate the process. First, it creates a *migration* class that describes the changes needed to update the database with the new changes. Second, the migration itself will be made.

```
symfony console make:migration
```

This command creates a file in the `migrations` folder with all the data needed to do the migration. If you need to create the tables on a new server you can use this file.

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

```
symfony 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.

Let's take a look now to our database:

Figure 5: The table has been created

As has been told before, note that the primary key ***id*** has been created automatically as an AUTO_INCREMENT column.

We can modify our table structure at any moment just by executing again the *make:entity* command with the name of the class that we want to change. Once the changes have been done, we have to generate a new migration and migrate it.

4.3 Troubleshooting

Just in case you get an error with the migration procedure like *'The metadata storage is not up to date, please run the sync-metadata-storage command to fix this issue'*, try this:

```
symfony console doctrine:migrations:sync-metadata-storage
```

If you see a message like `[OK] Metadata storage synchronized` but the problem still persists, add this code to your `config\packages\doctrine.yaml` file:

```yaml
doctrine_migrations:
    migrations_paths:
        # namespace is arbitrary but should be different from
↪   App\Migrations
        # as migrations classes should NOT be autoloaded
        'DoctrineMigrations': '%kernel.project_dir%/src/Migrations'
    storage:
        # Default (SQL table) metadata storage configuration
        table_storage:
            table_name: 'migration_versions'
            version_column_name: 'version'
            version_column_length: 1024
            executed_at_column_name: 'executed_at'
            execution_time_column_name: 'execution_time'
```

4.4 How to establish another primary key

If we want our table to have another primary key different than the automatically generated ***id***, we can change it by doing this:

- Delete the ***id*** attribute in the class, along with its getter and setter methods.
- Define the new primary key by adding the annotation `#[ORM\Id]` to the attribute or attributes that will form the new primary key.

```php
#[ORM\Id]
private $newId;
```

5 Operating with our database

To operate with our database, we need to use the ***EntityManagerInterface*** of Doctrine. This is a class that includes the methods needed to perform tasks like insert, delete, update or select.

Let's see first how to add rows to our tables.

5.1 Inserting objects to the database

To add a contact to our database, we need first to create a Contact object with some data. Create a new method in the `ContactController` class (create the controller if it doesn't exist), let's say "newContact". We link the method to the route **`/contact/new`**.

First, add the **`use`** for the `Contact` class in the controller class. We must use the **Entity Manager Interface** as well.

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Attribute\Route;
use App\Entity\Contact;
use Doctrine\ORM\EntityManagerInterface;
```

Now we can create the new method. The route will be `/contact/new`. The method receives an object of the **ManagerRegistryInterface** class, and returns a **Response**.

```
class ContactController extends AbstractController
{
    #[Route('/contact/new', name: 'app_newcontact')]
    public function newContact(EntityManagerInterface $entityManager):
    ↪  Response
    {

    }
}
```

Inside the method we create the object of the *Contact* class with the data that we want to insert into the table. We don't assign an id to the contact because the primary key is an autoincrement column in the database, so the id will be automatically assigned.

```
class ContactController extends AbstractController
{
    #[Route('/contact/new', name: 'app_newcontact')]
    public function newContact(EntityManagerInterface $entityManager):
     ↪  Response
    {
        $contact=new Contact();
        $contact->setName("Eduardo Gómez");
        $contact->setPhone("23454321");
        $contact->setEmail("eduardogomez@gmail.com");
    }
}
```

Then, we have to use the **`EntityManager`** object which will provide us with the methods to make the insert. We get the `ManagerRegistry` object as a parameter in our function (this is a *dependency injection*). We will need two methods:

- first, `persist()` to generate the information for the manager, before doing the query,
- second, `flush()` to effectively save the object to our table in the database and to do the query,

```
class ContactController extends AbstractController
{
    #[Route('/contact/new', name: 'app_newcontact')]
    public function newContact(EntityManagerInterface $entityManager):
     ↪  Response
    {
        $contact=new Contact();
        $contact->setName("Eduardo Gómez");
        $contact->setPhone("23454321");
        $contact->setEmail("eduardogomez@gmail.com");

        $entityManager->persist($contact);
        $entityManager->flush();
    }
}
```

Let's go to the URL `http://localhost:8000/contact/new` and see what happens.

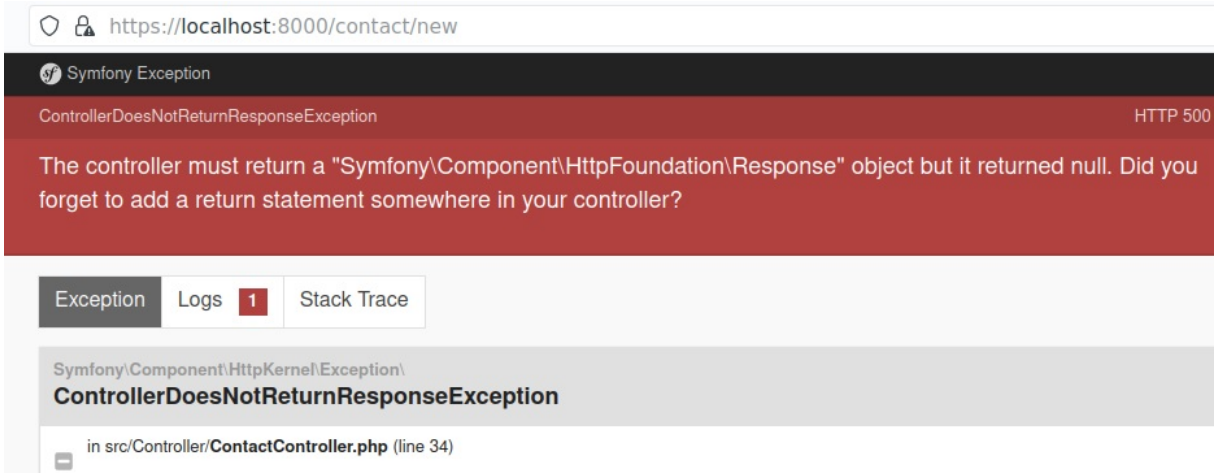

Figure 6: It works but with a warning

The warning tells us that the method must return a response object. We solve this by creating a view with a message "The contact has been added" and rendering the view from the method.

The `templates\contact\new.html.twig` file:

```
{% extends 'base.html.twig' %}

{% block title %}It works!{% endblock %}

{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%;
    ↪  font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h2>The contact has been added!</h2>
</div>
{% endblock %}
```

And then the method (change the data not to repeat the same contact):

```
class ContactController extends AbstractController
{
    #[Route('/contact/new', name: 'app_newcontact')]
```

```
    public function newContact(EntityManagerInterface $entityManager):
    ↪   Response
    {
        $contact=new Contact();
        $contact->setName("Eduardo Gómez");
        $contact->setPhone("23454321");
        $contact->setEmail("eduardogomez@gmail.com");

        $entityManager->persist($contact);
        $entityManager->flush();

        return $this->render('contact/new.html.twig', []);
    }
}
```

Now, when we run the controller by typing the url, we get the message:

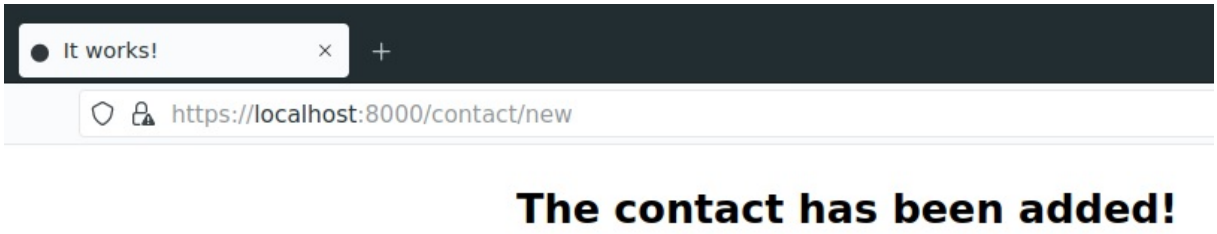

Figure 7: It totally works!

5.2 Inserting objects using parameters

Of course we don't want to change our code every time we need to add a contact to the database. We will see later how to use an API to introduce data. By now, we can specify that information by adding parameters to our URL.

```
#[Route('/contact/new/{name}/{phone}/{email}', name:
↪   'app_newcontact_param')]
public function newContactParam(EntityManagerInterface $entityManager,
↪   $name='', $phone='', $email=''): Response
{
    $contact=new Contact();
    $contact->setName($name);
    $contact->setPhone($phone);
```

```
    $contact->setEmail($email);

    $entityManager->persist($contact);
    $entityManager->flush();

    return $this->render('contact/new.html.twig', []);
}
```

Now we can add a new contact with different data without having to modify the controller code.

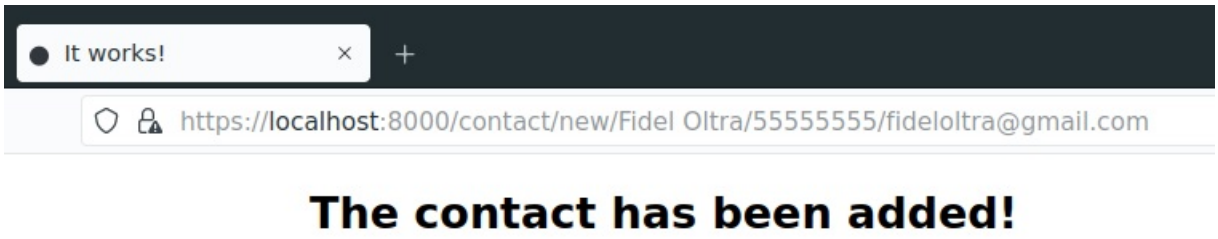

Figure 8: Adding a new contact using the URL

Let's check the database:

Figure 9: Checking if the database has been updated

5.3 Retrieving information from the database

To retrieve information from the database, we will need to instantiate a new repository object. As you remember, when we created the *Contact* model Symfony added two classes to the application: the class *Contact* itself and the class *ContactRepository*. This class is in the *Repository* folder and has some methods, inherited from *ServiceEntityRepository*, to recover information from our database:

- **find(value)** finds an object using the primary key. Returns a single object with its primary key being equal to the value.
- **findOneBy(condition)** finds an object that accomplishes the condition. Returns a single object.
- **findBy(condition)** finds all the objects that accomplish the condition. Return an array of objects.
- **findAll()** finds all the objects in the table. Return an array of objects.

Let's see an example of each one.

5.3.1 Finding an object by its id

We want to implement a method that receives the id of a contact and then shows the contact card in a view.

First, let's prepare the view. We call it **contact.html.twig**

```
{% extends 'base.html.twig' %}

{% block title %}Contact card{% endblock %}

{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%;
    ↪  font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h2>Contact card</h2>
    <h3>Id: {{ contact.id }}</h3>
    <h3>Name: {{ contact.name }}</h3>
    <h3>Phone: {{ contact.phone }}</h3>
    <h3>Email: {{ contact.email }}</h3>
</div>
{% endblock %}
```

And now we create the method **`getContactById()`** that receives an id as a parameter, finds the contact in the database, and then sends the retrieved contact to the view.

```
#[Route('/contact/{id}', name: 'app_getcontactbyid')]
public function getContactById(EntityManagerInterface $entityManager, $id =
↪   ''): Response
{
    $rep = $entityManager->getRepository(Contact::class);
    $contact = $rep->find($id);
    return $this->render('contact/contact.html.twig', [
        'contact' => $contact,
    ]);
}
```

With the line:

```
$rep = $entityManager->getRepository(Contact::class)
```

we create a repository for our class *`Contact`*. And then with the line:

```
$contact = $rep->find($id);
```

We search in the associated table for a contact with the given **id**, then the result is saved into the object. Finally, we send the object to the view. The result is:

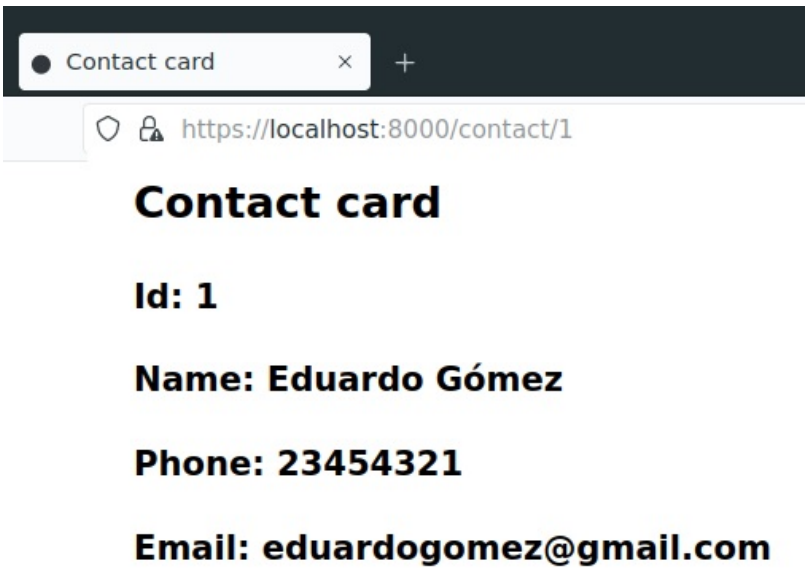

Figure 10: Contact found

If the supplied **id** doesn't exist, we will get:

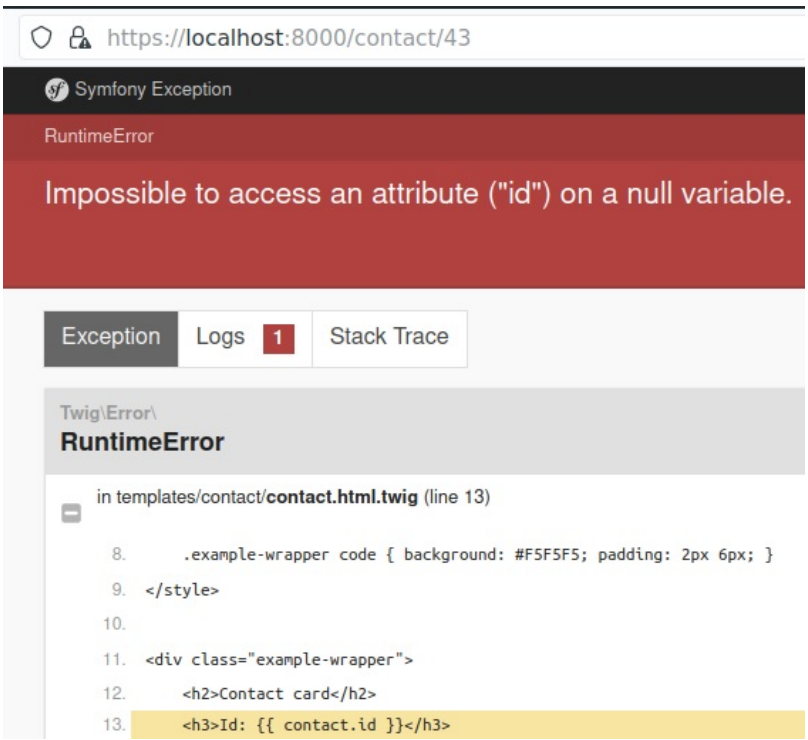

Figure 11: Contact not found error

As we already know, to solve the problem we have to check the existence of the object in our template.

```
<div class="example-wrapper">
    <h2>Contact card</h2>
    {% if contact %}
        <h3>Id: {{ contact.id }}</h3>
        <h3>Name: {{ contact.name }}</h3>
        <h3>Phone: {{ contact.phone }}</h3>
        <h3>Email: {{ contact.email }}</h3>
    {% else %}
        <h3>The contact doesn't exist</h3>
    {% endif %}
</div>
```

And now, if we search for an nonexistent id, we get:

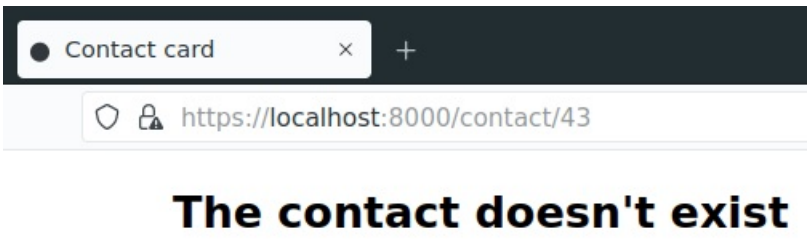

Figure 12: Contact not found message

5.3.2 Finding an object by a property different from the primary key

We can search for an object in our database using any condition different from the primary key value. If we know that only an object will accomplish the condition we can use `findOneBy`. Let's see how to find a contact by its name:

```
#[Route('/contact/{name}', name: 'app_getcontactbyname')]
public function getcontactbyname(EntityManagerInterface $entityManager,
↪   $name='') {
    $rep=$entityManager->getRepository(Contact::class);
    $contact=$rep->findOneBy(["name"=>$name]);
    return $this->render('contact/contact.html.twig', [
      'contact' => $contact,
    ]);
}
```

We can search a contact by its name from the URL:

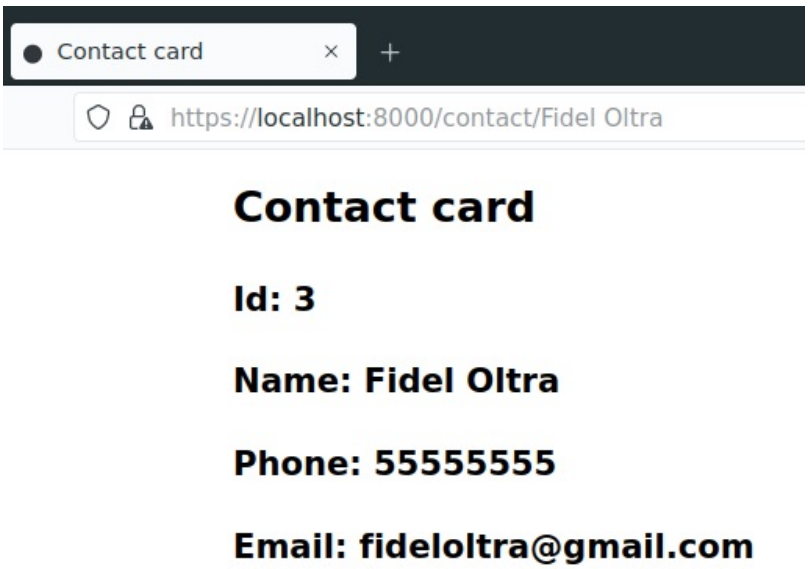

Figure 13: Contact found by name

As we are searching for a contact, we render the same view that in the search by id.

> **ATTENTION!** To avoid conflict between the two routes `contact/{id}` and `contact/{name}` we need to discriminate when the parameter is a number. Thus, the route for searching by the id must be changed in order to add the requirement.

```
#[Route('/contact/{id<\d+>}', name: 'app_getcontactbyid')]
```

Of course, we can also rename both routes to avoid duplicates.

```
#[Route('/contact/byid/{id}', name: 'app_getcontactbyid')]
#[Route('/contact/byname/{name}', name: 'app_getcontactbyname')]
```

The `findOneBy` method accepts more than one condition. The conditions are separated by commas.

```
$contact=$rep->findOneBy(["name"=>$name, "phone"=>$phone]);
```

5.3.3 Finding a list of objects by one or more conditions

Sometimes our condition can be accomplished by more than one object. In this case, we will use the **findBy** method. This method returns an array of objects of the given class. Let's suppose that we have

some contacts with the same name, and we want to retrieve them all. We should use **`findBy`** instead of **`findOneBy`**.

```
$contacts=$rep->findBy(["name"=>$name]);
```

Then we send the **$contacts** array to a new template and, using a loop, we can show every single object. For instance, let's create a new template named **`contactlist.html.twig`** with this content:

```
{% extends 'base.html.twig' %}

{% block title %}Contact card{% endblock %}

{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%;
    ↪  font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h2>Contact list</h2>
    {% if contacts %}
        {% for contact in contacts %}
            <strong>Id:</strong> {{ contact.id }}<br/>
            <strong>Name:</strong> {{ contact.name }}<br/>
            <strong>Phone:</strong> {{ contact.phone }}<br/>
            <strong>Email:</strong> {{ contact.email }}<br/>
            <hr/>
        {% endfor %}
    {% else %}
        <h3>No matching contacts</h3>
    {% endif %}
</div>
{% endblock %}
```

Then the controller gets the contact list and renders the view.

```
#[Route('/contact/{name}', name: 'app_getcontactbyname')]
public function getcontactbyname(EntityManagerInterface $entityManager,
↪   $name) {
    $rep=$entityManager->getRepository(Contact::class);
```

```
    $contacts=$rep->findBy(["name"=>$name]);
    return $this->render('contact/contactlist.html.twig', [
        'contacts' => $contacts,
    ]);
}
```

5.3.4 Retrieving all the objects in the table

To retrieve all the objects in the table, we will use the **findAll()** method. This method will return an array with every single object in our table.

Let's create a method **contacts()** to retrieve all the contacts in the table and then render our **contactlist.html.twig** template to show them.

```
#[Route('/contact', name: 'app_contacts')]
public function contacts(EntityManagerInterface $entityManager): Response
{
    $rep=$entityManager->getRepository(Contact::class);
    $contacts=$rep->findAll();
    return $this->render('contact/contactlist.html.twig', [
        'contacts' => $contacts,
    ]);
}
```

Then, by calling the **/contact** route we will get the whole contact list.

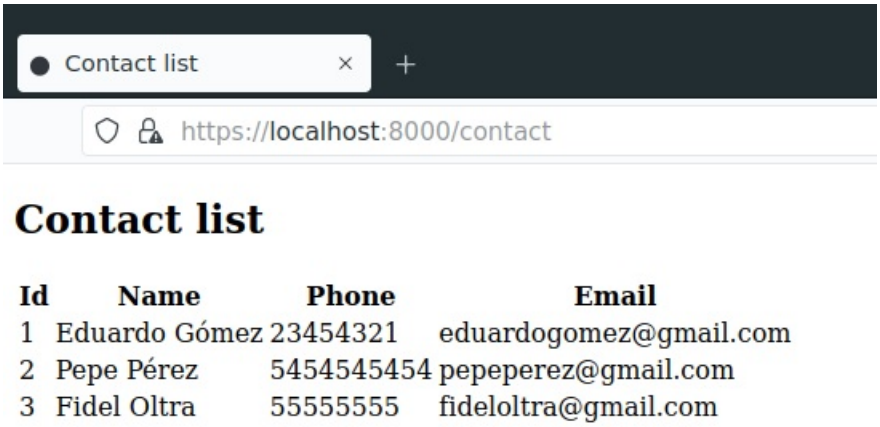

Figure 14: Contact list

> Remember to use `symfony console debug:router` console command to view the routes order.

5.3.5 Automatically Fetching Objects

In many cases, you can use the `EntityValueResolver` to do the query automatically. You can simplify the controller to:

```
#[Route('/contact/{id}', name: 'app_getcontactbyid')]
public function contact(Contact $contact): Response
{
    return $this->render('contact/contact.html.twig', [
        'contact' => $contact,
    ]);
}
```

Entity Value Resolver was introduced in Symfony 6.2. This behavior is enabled by default on all the controllers, but it can be disabled by setting the `doctrine.orm.controller_resolver.auto_mapping` config option to false.

More info about the Entity Value Resolver: https://symfony.com/doc/6.3/doctrine.html#automatically-fetching-objects-entityvalueresolver

5.4 Advanced queries

The methods that we have seen in the previous chapter (find, findOneBy, findBy, findAll) belong to a class named **ServiceEntityRepository**. This class is extended by a repository class that Symfony creates for every entity. In our case, the class for the **Contact** entity is **ContactRepository**.

This class can be found in the `src\Repository` folder of our project, and it's editable. Therefore, we can modify its content and, more important, we can add new methods. Let's see an example.

When we search for a contact by its name, we need to introduce the whole name. If we introduce only a part of the name, the `findOneBy` or `findBy` methods won't be able to find the contact. Imagine that our two contacts named "Fidel" have a surname: one of them is *Fidel Oltra* and the other one is *Fidel Landete*.

If we search for "Fidel", we will get:

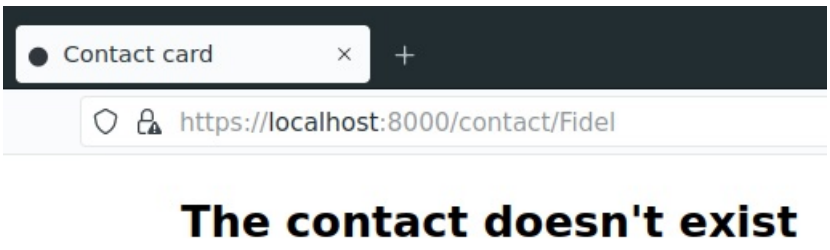

Figure 15: Contact "Fidel" not found

That's because the default *find* methods make a full search. If we want to perform a partial search (WHERE name LIKE...), we need to add a new *find* method to our repository. In this case, let's add a method **`findByName`** in our `src\Repository\ContactRepository` class:

```
public function findByName($name) {
    return $this->createQueryBuilder('contact')
    ->andWhere('contact.name LIKE :val')
    ->setParameter('val', '%'.$name.'%')
    ->getQuery()
    ->getResult();
}
```

We use the *`createQueryBuilder`* method to create a new query. The function *`andWhere`* adds a condition, and *`setParameter`* gives values to the parameters of the condition. Finally, *`getQuery()->getResult()`* returns an array with the contacts that match the query.

Now, in our **`getContactByname`** method, we can use our **`findByName`** function instead:

```
$contacts=$rep->findByName($name);
```

Now the method calls the new function **`findByName`** so if we look for `"Fidel"` we will get every contact with `"Fidel"` in its name.

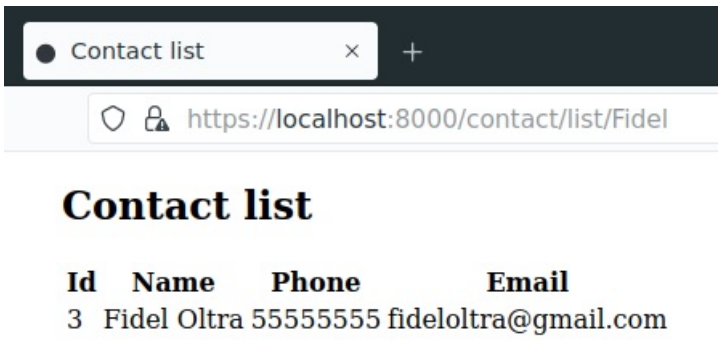

Figure 16: List with the contacts that match the partial name

More about the QueryBuilder syntax:

QueryBuilder

5.5 Deleting objects from the database

It's very easy to remove **one** object from the database. We just need to find the object and then call the method ***remove*** of our entity manager.

Let's add a new method, **deleteContact**, that receives an id from the route `/contact/remove`:

```
#[Route('/contact/remove/{id}', name: 'app_deletecontact')]
public function deleteContact(EntityManagerInterface $entityManager, $id) {
    $rep=$entityManager->getRepository(Contact::class);
    // we find the contact with the given id
    $contact=$rep->find($id);
    // if it exists, we remove it
    if($contact) {
        $entityManager->remove($contact);
        $entityManager->flush();
    }
    // now we refresh the contact list
    $contacts=$rep->findAll();
    return $this->render('contact/contactlist.html.twig', [
        'contacts' => $contacts,
    ]);
}
```

But, what if we want to delete all the objects that match a given condition?

We have two options:

5.5.1 Deleting an array of objects from the controller

If we can use a ***find*** method to retrieve the objects that we want to delete, we can solve the problem in the controller method.

```
$rep=$entityManager->getRepository(Contact::class);
// we retrieve the contacts that match the condition
$contacts=$rep->findBy(...write the condition here..);
if($contacts) {
    // and now, we delete them one by one
    foreach($contacts as $contact) {
        $entityManager->remove($contact);
        $entityManager->flush();
    }
}
```

After retrieving the contacts that match the given condition with ***findBy()***, we do a *foreach* to delete the contacts one by one.

5.5.2 Deleting an array of objects from the repository class

As we have seen before, if we can't make use of the *find* methods available from the repository, we can create our own methods by adding functions to the repository class. If we need a new *find* method to search the objects we want to delete, we can remove them in the same method.

Remember our findByName method we created in the ContactRepository class to be able to search for the contacts which name match a certain string pattern:

```
public function findByName($name) {
    return $this->createQueryBuilder('contact')
    ->andWhere('contact.name LIKE :val')
    ->setParameter('val', '%'.$name.'%')
    ->getQuery()
    ->getResult();
}
```

We can copy and paste to create a ***deleteByName*** function that deletes the contacts that meet the condition, instead of returning them. So, we first add a call to the function ***delete()*** before the

condition; then, as we are not going to return an array, we will change the call to ***getResult()*** for a call to the ***execute()*** function.

```
public function deleteByName($name) {
    return $this->createQueryBuilder('contact')
    ->delete()
    ->andWhere('contact.name LIKE :val')
    ->setParameter('val', '%'.$name.'%')
    ->getQuery()
    ->execute();
}
```

Now, in the controller, we can use the new **deleteByName** function to create a new method that receives a partial name as a parameter and then removes any contact that match the name pattern:

```
#[Route('/contact/remove/{name}', name: 'app_deletecontactbyname')]
    public function deleteContactByName(EntityManagerInterface
    ↪   $entityManager, $name) {
        $rep=$entityManager->getRepository(Contact::class);
        $deleted=$rep->deleteByName($name);
        // now we refresh the contact list
        $contacts=$rep->findAll();
        return $this->render('contact/contactlist.html.twig', [
            'contacts' => $contacts,
        ]);
    }
```

5.6 Updating objects in the database

Updating an object in the database is very easy. We just need to find the object we want to update, modify the desired attributes and then make a **`flush()`**.

5.6.1 Updating one object

To update a single object, we do:

- a `find($id)` to find and retrieve the object
- a call to the respective **set** method to make the change
- a call to `flush()`

```
$rep = $entityManager->getRepository(Contact::class);
$contact = $rep->find($id);
if ($contact) {
    $contact->setName($newname); // we change the name
    // other modifications...
    $entityManager->flush();
}
```

5.6.2 Updating a list of objects

To update a list of objects from the controller, we do:

- a ***findBy(condition)*** to find and retrieve the objects (or a ***findAll()*** if we want to apply the changes to all the objects in the table).
- a ***foreach*** to go over every object in the list.
- inside the ***foreach***, a call to the respective set methods to make the changes and then finally a ***flush()***.

Let's add a "96" area code to every phone in our contact table:

```
#[Route('/contact/add/{code}', name: 'addareacode')]
public function addAreaCode(EntityManagerInterface $entityManager,$code) {
    $rep=$entityManager->getRepository(Contact::class);
    $contacts=$rep->findAll();
    foreach($contacts as $contact) {
        $phoneNumber=$contact->getPhone();
        $phoneNumber=$code.$phoneNumber;
        $contact->setPhone($phoneNumber);
        $entityManager->flush();
    }
    $contacts=$rep->findAll();
    return $this->render('contact/contactlist.html.twig', [
        'contacts' => $contacts,
    ]);
}
```

This is our contact list:

Now let's do `http://localhost:8000/contact/add/96` and then check the list again:

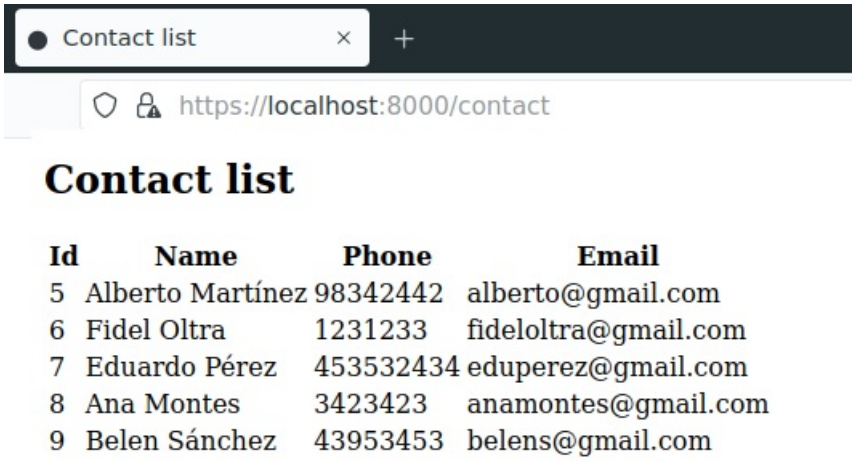

Figure 17: Contact list before adding the area code

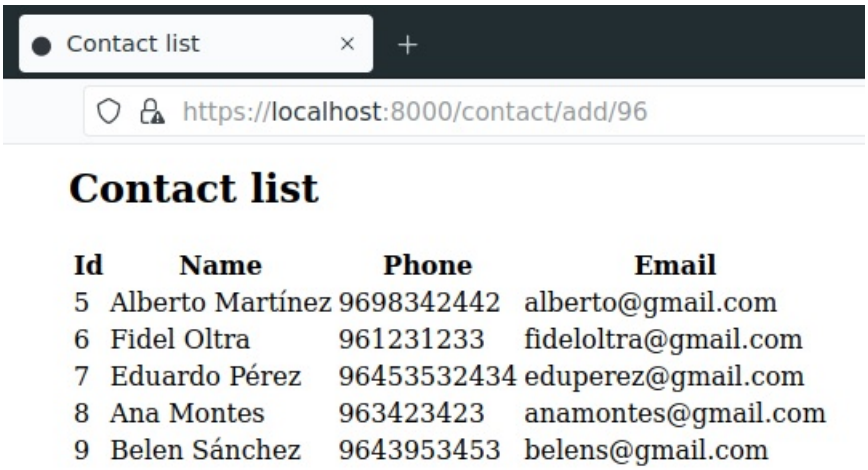

Figure 18: Contact list after adding the area code

5.6.3 Updating a list of objects using the repository class

What if we add a couple of contacts without the “96” in the phone, and then we want to add the “96” only to these contacts? Time to get back to the repository class.

First, we would need a *find* method to search only the contacts whose phone number doesn’t start with “96” (or whatever the code is).

```
public function findByAreaCodeNOT($code) {
    return $this->createQueryBuilder('contact')
    ->andWhere('contact.phone NOT LIKE :val')
    ->setParameter('val', $code.'%')
    ->getQuery()
    ->getResult();
}
```

And now let’s change the code in our controller method to:

```
#[Route('/contact/add/{code}', name: 'addareacode')]
public function addAreaCode(EntityManagerInterface $entityManager,$code) {
    $rep=entityManager->getRepository(Contact::class);
    $contactstochange=$rep->findByAreaCodeNOT($code);

    foreach($contactstochange as $contact) {
        $phoneNumber=$contact->getPhone();
        $phoneNumber="96".$phoneNumber;
        $contact->setPhone($phoneNumber);
        $entityManager->flush();
    }
    $contacts=$rep->findAll();

    return $this->render('contact/contactlist.html.twig', [
        'contacts' => $contacts,
    ]);
}
```

Now, if we have this contact list:

id	name	phone	email
5	Alberto Martínez	9698342442	alberto@gmail.com
6	Fidel Oltra	961231233	fideloltra@gmail.com
7	Eduardo Pérez	96453532434	eduperez@gmail.com
8	Ana Montes	963423423	anamontes@gmail.com
9	Belen Sánchez	9643953453	belens@gmail.com
10	Abel Martinez	894322	abelm@gmail.com
11	Lucía Fuentes	943242	lucyfuentes@gmail.com

Figure 19: Contact list before the update

After executing the method from the route `localhost:8000/contact/add/96` we will get:

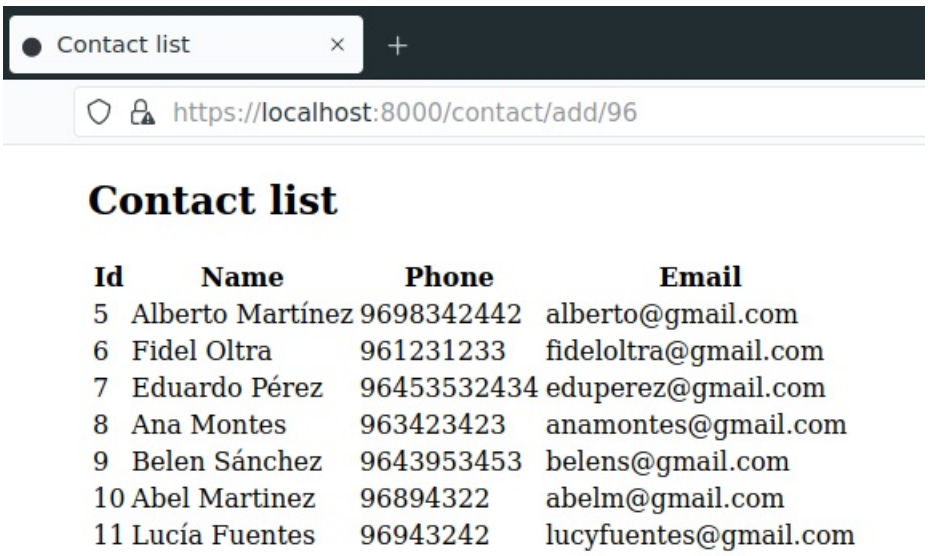

Figure 20: Contact list after the update

Pay attention to the contacts 10 and 11. They are the only two contacts that have been modified.

6 Relationships

As you know, a database usually contains more than one table, and these tables should have some kind of relationship between them. We know how to implement a relationship between tables in SQL, and how to use tools like *phpMyAdmin*. But, how to implement the same relationships in Doctrine?

The short answer is: the same way that we add any other property to an entity. We just need to pay

attention to the **type** of the new property, and be sure that the referred table already exists.

Remember that we can find two kind of relationships between tables:

- A relationship where more than one table has a M participation (1:1, 1:M, M:1, 1:1:M…)
- A relationship where none or only one table has a M participation (M:M, 1:M:M, M:M:M…)

The main difference is that, in the second type, a new table has to be created. In the first type there's no need to create a new table, we will implement the relationship only by using foreign keys in the existing tables.

Let's implement a couple of examples to see how it works in Doctrine / Symfony.

6.1 1:M relationships

We are going to work with our **contacts** database. At this moment, we have only one table in the database: **contact**. The structure of the table is:

id	name	phone	email
7	Eduardo Pérez	96453532434	eduperez@gmail.com
8	Ana Montes	963423423	anamontes@gmail.com
9	Belen Sánchez Gómez	9643953453	belens@gmail.com
13	Juan Gómez	96600101010	juangomez@gmail.com

Figure 21: Contact table

Now, we want to add a **city** column that references a new table called **city** as well. In a city we can have a lot of contacts, but in a given moment a contact will have only one city. So the relationship is **1:M** (1 city –> M contacts).

We know how to work in these situations. First, we create the new table **city** with, let's say, a primary key (id), the postal code and the city name. Then we will add a column in the **contact** table that will be a foreign key linked to the primary key of **city**. We create the relationship in the relation view section, and that's all.

But we don't want to work directly in the database, we want to do exactly the same process using Doctrine and Symfony.

6.1.1 Creating the new entity

First, let's create our **city** table/entity as we already know: with `symfony console make:entity`.

Now we can create the entity **city**. Remember: we won't add the **`id`** here, it's automatic. So the fields will be the postal code and the city name.

```
symfony console make:entity

 Class name of the entity to create or update (e.g. BravePopsicle):
 > City

 created: src/Entity/City.php
 created: src/Repository/CityRepository.php

 Entity generated! Now let's add some fields!
 You can always add more fields later manually or by re-running this
 ↪  command.

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

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

 Field length [255]:
 > 5

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

 updated: src/Entity/City.php

 Add another property? Enter the property name (or press <return> to stop
 ↪  adding fields):
 > cityName

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

 Field length [255]:
 > 50
```

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

 updated: src/Entity/City.php

 Add another property? Enter the property name (or press <return> to stop
 ↪  adding fields):
 >
```

Now, we must generate the migration files and then run the migration:

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

Now we should have the entity **City** in our project, and the table **city** in our database.

The entity:

```php
<?php

namespace App\Entity;

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

#[ORM\Entity(repositoryClass: CityRepository::class)]
class City
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 5)]
    private ?string $postalCode = null;

    #[ORM\Column(length: 50)]
    private ?string $cityName = null;

    public function getId(): ?int
    {
```

```
        return $this->id;
    }

    public function getPostalCode(): ?string
    {
        return $this->postalCode;
    }

    public function setPostalCode(string $postalCode): static
    {
        $this->postalCode = $postalCode;

        return $this;
    }

    public function getCityName(): ?string
    {
        return $this->cityName;
    }

    public function setCityName(string $cityName): static
    {
        $this->cityName = $cityName;

        return $this;
    }
}
```

And here's the table:

	#	Nombre	Tipo	Cotejamiento
☐	1	**id**	int	
☐	2	**postal_code**	varchar(5)	utf8mb4_unicode_ci
☐	3	**city_name**	varchar(50)	utf8mb4_unicode_ci

Figure 22: City table

6.1.2 Creating the relationship

Now, to complete the process, we need to add a property ***city*** to our entity **Contact**. That property will perform the function of foreign key, making reference to the city where our contact lives.

We add the property doing again a **make:entity**. Then, when we are asked about the property type, we must answer ***relation***. After that we must enter the related entity, and the type of the relationship.

Finally, we must say if we want to have a method in the referred class to get an array with all the occurrences of the first class. In our case, we will be asked if we want to have a `getContacts()` method in the City class that would return a list of contacts of a given city.

This is the whole process.

```
$ symfony console make:entity

 Class name of the entity to create or update (e.g. OrangePizza):
 > contact
 Your entity already exists! So let's add some new fields!
 New property name (press <return> to stop adding fields):
 > city
 Field type (enter ? to see all types) [string]:
 > relation
 What class should this entity be related to?:
 > city

What type of relationship is this?

-----------------------------
Type           Description
 ↪
-----------------------------
ManyToOne   Each Contact relates to (has) one city.
 ↪
            Each city can relate to (can have) many Contact objects
OneToMany   Each Contact can relate to (can have) many city objects.
 ↪
            Each city relates to (has) one Contact
 ↪
ManyToMany  Each Contact can relate to (can have) many city objects.
 ↪
            Each city can also relate to (can also have) many Contact
 ↪  objects
```

```
OneToOne    Each Contact relates to (has) exactly one city.
 ↪
            Each city also relates to (has) exactly one Contact.
 ↪
----------------------------

 Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
 > ManyToOne
 Is the Contact.city property allowed to be null (nullable)? (yes/no)
 ↪  [yes]:
 >
 Do you want to add a new property to city so that you can access/update
 ↪  Contact objects from it - e.g. $city->getContacts()? (yes/no) [yes]:
 >
 A new property will also be added to the city class so that you can access
 ↪  the related Contact objects from it.
 New field name inside city [contacts]:
 >
 updated: src/Entity/Contact.php
 updated: src/Entity/City.php

 Add another property? Enter the property name (or press <return> to stop
 ↪  adding fields):
 >
```

Now we do again the `symfony console make:migration` and `symfony console doctrine:migrations:migrate`. As a result, we get a new property (city) in our contact class with its methods get and set:

```php
#[ORM\ManyToOne(inversedBy: 'contacts')]
private ?City $city = null;

public function getCity(): ?City {
    return $this->city;
}

public function setCity(?City $city): static {
    $this->city = $city;
    return $this;
}
```

We get a new property (contacts) in our city class with the method `get`, a method to add a new contact to the city, and a method to remove a contact from the city:

```
#[ORM\OneToMany(mappedBy: 'city', targetEntity: Contact::class)]
private Collection $contacts;
...
/**
 * @return Collection<int, Contact>
 */
public function getContacts(): Collection {
    return $this->contacts;
}

public function addContact(Contact $contact): static {
    if (!$this->contacts->contains($contact)) {
        $this->contacts->add($contact);
        $contact->setCity($this);
    }
    return $this;
}

public function removeContact(Contact $contact): static {
    if ($this->contacts->removeElement($contact)) {
        // set the owning side to null (unless already changed)
        if ($contact->getCity() === $this) {
            $contact->setCity(null);
        }
    }
    return $this;
}
```

And finally, we get the foreign key in our **contact** table:

6.1.3 Managing the relationships from Symfony

At this point, we need to add some things to our application. First, we should create a new controller for the entity **City** with at least two methods:

- a method to list the cities (with its template)
- a method to add a new city (with its template)

First, let's create the **CityController** controller:

Estructura de tabla | Vista de relaciones

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo
☐	1	**id**	int(11)			No
☐	2	**name**	varchar(50)	utf8mb4_unicode_ci		No
☐	3	**phone**	varchar(15)	utf8mb4_unicode_ci		No
☐	4	**email**	varchar(60)	utf8mb4_unicode_ci		No
☐	5	**city_id**	int(11)			Sí

Figure 23: Foreign key added to contact table

```
symfony console make:controller CityController
```

Then we change the ***index()*** method as follows:

```
#[Route('/city', name: 'app_city')]
public function index(EntityManagerInterface $entityManager): Response {
    $rep=$entityManager->getRepository(City::class);
    $cities=$rep->findAll();
    return $this->render('city/index.html.twig', [
        'cities' => $cities,
    ]);
}
```

We are just retrieving all the cities from the database, and then rendering the view with a cities array as a parameter. The view uses a twig template:

```
{% extends 'base.html.twig' %}

{% block title %}Cities list{% endblock %}

{% block body %}

{% if cities %}
    <h2>Cities list</h2>
    <table>
    <tr><th>Id</th><th>Postal Code</th><th>Name</th></tr>
```

```
    {% for city in cities %}
        <tr>
        <td>{{ city.id }}</td>
        <td>{{ city.postalcode }}</td>
        <td>{{ city.cityname }}</td>
        </tr>
    {% endfor %}
    </table>
{% else %}
    <h2>No cities in the database</h2>
{% endif %}
{% endblock %}
```

Now let's make a controller method to add a new city manually. The process is the same seen in the point to add a new contact.

Now, by calling the URL `localhost:8000/city/new`, or through a link in the cities list, we add a new city and then the updated list appears.

Figure 24: Cities list

By now, all these operations are very familiar to us. We are performing the same tasks that we did with the contact table: show a list, add a new city…

But at this moment, we have a relationship between both tables, contact and city. How can we say that a contact lives in a certain city?

6.1.4 Adding a new object with a foreign key

We are going to learn how to add a new object to a class that has a foreign key. In the entities, that means that the class (in our case, **Contact**) has a property that is an instance of another class (in our

case, **City**). Let's see how to proceed in this situation.

First, in the **ContactController** class we add the lines:

```
use App\Entity\City;
```

And in the update and insert methods add the new property retrieving a city object from the database (by its `id`) and then using the new `setCity` method:

```
...
$city = $entityManager->getRepository(City::class)->find($id_city);
$contact->setCity($city);
...
```

Just by doing `persist()` and `flush()` with the contact, the property *city* will be updated in our database.

```
$entityManager->persist($contact);
$entityManager->flush();
```

Figure 25: Contact added to the database

Now, in the contact list we can show the city where the contact lives. We just need to edit the contact list template and add a new column to the table:

```
<td>{{ contact.city.cityname }}</td>
```

Contact list

Id	Name	Phone	Email	City	
7	Eduardo Pérez	96453532434	eduperez@gmail.com		Contact
8	Ana Montes	963423423	anamontes@gmail.com		Contact
9	Belen Sánchez Gómez	9643953453	belens@gmail.com		Contact
13	Juan Gómez	96600101010	juangomez@gmail.com		Contact
14	Pedro González	968324231	pedroooo@gmail.com		Contact
15	Fidel Oltra	969123123	fideloltra@gmail.com		Contact
16	Sergio López	8238424	sergio@gmail.com	Gandia	Contact

New contact

Figure 26: The city in the contact list

We should change the updating form and the contact card as well in order to modify / show the city.

6.1.5 Retrieving data using the primary key - foreign key link.

Now we know how to enter the city where a new contact lives. When we show the contact list, the city where every contact lives will appear.

Now, let's try something different. We want to show, in every city card, a list of the contacts that live there. How?

First, we need to remember that, when we created the relationship, we said that we wanted the link to work in both directions. That means that we have a ***getCity()*** method in the **Contact** class, but we have a ***getContact()*** in the **City** class as well.

```
/**
 * @return Collection<int, Contact>
 */
public function getContacts(): Collection {
    return $this->contacts;
}
```

Let's create a twig template to show the content of a city, and a controller method to render that template.

This is the method we have to recover a city given its id:

```
#[Route('/city/{id<id\d+>}', name: 'app_getcitybyid')]
public function getCityById(EntityManagerInterface $entityManager, $id):
↪   Response
{
    $city = $entityManager->getRepository(City::class)->find($id);
    return $this->render('city/city.html.twig', [
        'city' => $city,
    ]);
}
```

The twig template:

```
{% extends 'base.html.twig' %}

{% block title %}{{ title }}{% endblock %}

{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%;
    ↪   font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h2>{{ title }}</h2>
    {% if city %}
        <strong>Id:</strong> {{ city.id }}<br/>
        <strong>Postal Code:</strong> {{ city.postalcode }}<br/>
        <strong>City Name:</strong> {{ city.cityname }}<br/>
    {% else %}
        <h3>The city doesn't exist</h3>
    {% endif %}
</div>
{% endblock %}
```

As a result, we get:

Figure 27: The city card

And finally let's see how to show the contacts. It's very easy, because we have the **`getContacts()`** method in the **City** class, as we saw above. Thus, as we know, we can use `city.contacts` in the twig to get the list of the contacts that live in the city.

So, we add to the twig template a `for` loop to go over the contact list that belongs to the city we are displaying. This is the modified template:

```
{% extends 'base.html.twig' %}

{% block title %}{{ title }}{% endblock %}
{% block body %}
<h1>{{ title }}</h1>
<h4>Postal code: {{ city.postalcode }}</h4>
<h4>City name: {{ city.cityname }}</h4>
<h4>Contact list: </h4>
{% for contact in city.contacts %}
    <strong>Name:</strong> {{ contact.name }}
    <strong>Phone:</strong> {{ contact.phone }}
    <strong>Email:</strong> {{ contact.email }}
    <br/>
{% endfor %}
{% endblock %}
```

And when we go to the city card, this is the result we get:

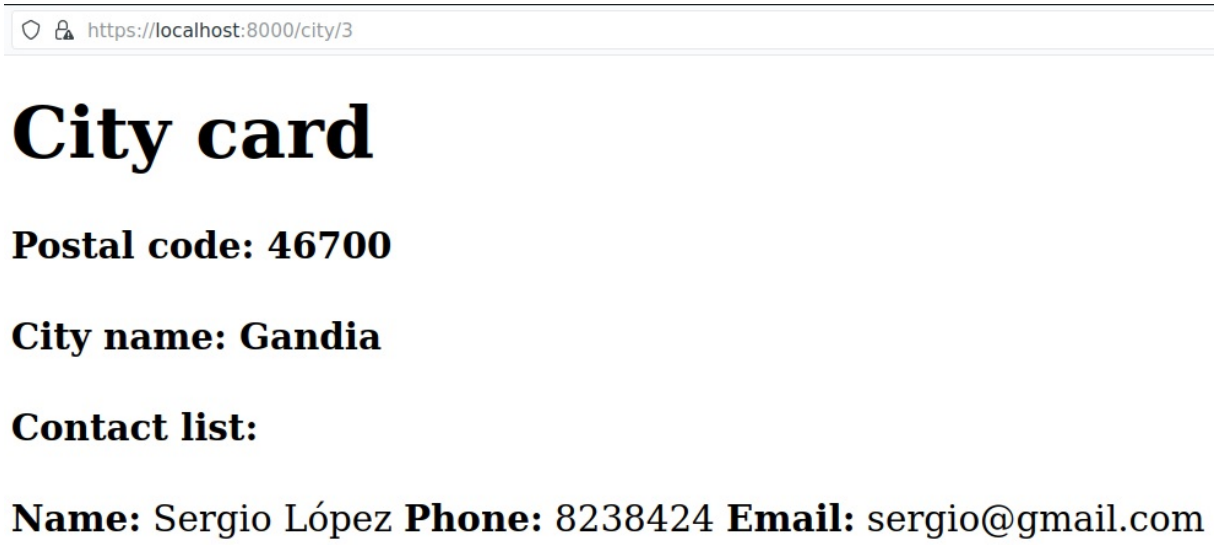

Figure 28: City with their contacts

6.1.6 Adding a new entity in both classes.

Let's suppose that we want to add a new contact, and this contact lives in a city that doesn't exist in our database.

We can add both objects to the database in the same method, but first we need to persist the referred object, and then the object with the foreign key.

In our example:

```
// we create the new city
$city = new City();
$city->setPostalcode(...);
$city->setCityname(...);
//we create the contact
$contact = new Contact();
$contact->setName(...);
$contact->setPhone(...);
$contact->setEmail(...);
// we assign the city to the contact
$contact->setCity($city);
// and now first we persist the city
$entityManager->persist($city);
// then the contact
$entityManager->persist($contact);
```

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

6.2 Other association types (1:1, M:M)

The relationships **1:1** and **M:M** work the same way that **1:M**.

If the relationship is **1:1**, we will get in both classes an object of the other class.

If the relationship is **M:M**, we will get in both classes a collection of objects of the other class.

More about mapping relationships in Doctrine.

Association mapping in Doctrine

Symfony Doctrine documentation