Server-side Web Development

Unit 12. Databases with Doctrine. Guided example.

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

In this example we're going to do the model part and database connection of the Contacts application. It follows the last guided practice.

1 Overview

The diagram entity-relationship of our database is:

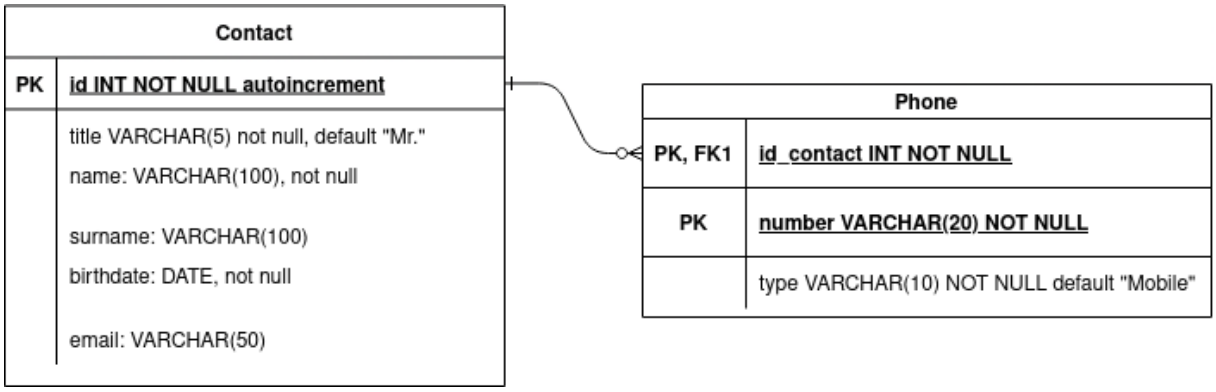

Figure 1: Contacts database

Each contact can have 0 or more phones and each phone must belong to a contact.

As we need the **Doctrine ORM** packet on our project, install it using Composer:

```
composer require symfony/orm-pack
```

2 Database connection configuration

First of all, we need to create a `.env.local` file in our project root folder. The purpose of this file is to write the configurations needed to connect to the local database, including passwords.

This file **must be listed in the `.gitignore` file** because we don't want to upload it to the version control. Please, check it carefully.

The content of the .env.local file should have the line:

```
DATABASE_URL="mysql://contactsymfony:contacts@127.0.0.1:3306/contactsymfony?serverVersio
```

Check your server version. You can change your user and password if you want.

As you should know, the user used to operate with the database must be a user with permissions only to that database. So it's better if you create the database and the user needed before. You can do that operation with PhpMyAdmin or in the MySql console:

```
CREATE USER 'contactsymfony'@'%' IDENTIFIED BY 'contacts';
CREATE DATABASE `contactsymfony`;
GRANT ALL PRIVILEGES ON `contactsymfony`.* TO 'contactsymfony'@'%';
```

3 Models

To create the Contact model, run the next command in the project folder:

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

And answer the questions:

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

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/Contact.php

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

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

Field length [255]:
```

```
> 100

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

updated: src/Entity/Contact.php

Add another property? Enter the property name (or press <return> to stop adding
> surname

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

Field length [255]:
> 100

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
> birthdate

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

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
> email

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

```
Field length [255]:
> 50

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

updated: src/Entity/Contact.php
```

And the Phone model:

```
symfony console make:entity Phone
```

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

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

 What class should this entity be related to?:
 > Contact

What type of relationship is this?
 ------------ ------------------------------------------------
  Type         Description
 ------------ ------------------------------------------------
  ManyToOne    Each Phone relates to (has) one Contact.
               Each Contact can relate to (can have) many Phone objects.

  OneToMany    Each Phone can relate to (can have) many Contact objects.
               Each Contact relates to (has) one Phone.

  ManyToMany   Each Phone can relate to (can have) many Contact objects.
               Each Contact can also relate to (can also have) many Phone object

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

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

 Is the Phone.id_contact property allowed to be null (nullable)? (yes/no) [yes]:
 > no

 Do you want to add a new property to Contact so that you can access/update Phon
e.g. $contact->getPhones()? (yes/no) [yes]:
 > yes

 A new property will also be added to the Contact class so that you can access t

 New field name inside Contact [phones]:
 > phones

 Do you want to activate orphanRemoval on your relationship?
 A Phone is "orphaned" when it is removed from its related Contact.
 e.g. $contact->removePhone($phone)

 NOTE: If a Phone may *change* from one Contact to another, answer "no".

 Do you want to automatically delete orphaned App\Entity\Phone objects (orphanRe
 > no

 updated: src/Entity/Phone.php
 updated: src/Entity/Contact.php

 Add another property? Enter the property name (or press <return> to stop adding
 > number

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

 Field length [255]:
 > 20

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

```
updated: src/Entity/Phone.php

Add another property? Enter the property name (or press <return> to stop adding
> type

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

Field length [255]:
> 10

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

updated: src/Entity/Phone.php
```

Before doing the migration, let's change the models so that they match our requirements.

In the `Contact` class file, we need to add the default option to the `title` property:

```php
#[ORM\Column(length: 5, options: ["default"=> "Mr."])]
private ?string $title = null;
```

Do the same in the Phone class with the `type` property:

```php
#[ORM\Column(length: 10, options: ["default"=> "Mobile"])]
private ?string $type = null;
```

In Phone, to make the primary key the combination of the properties `id_contact` and `number`, first we delete the `id` property that has been created by the maker. Then, add the `#[ORM\Id]` attribute to the properties:

```php
#[ORM\Id]
#[ORM\ManyToOne(inversedBy: 'phones')]
#[ORM\JoinColumn(nullable: false)]
private ?Contact $id_contact = null;

#[ORM\Id]
```

```
#[ORM\Column(length: 20)]
private ?string $number = null;
```

Now, we are ready to do the migration:

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

We can check that the tables in our database have been created according to our schema:

```
mysql> describe contact;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| id        | int          | NO   | PRI | NULL    | auto_increment |
| title     | varchar(5)   | NO   |     | Mr.     |                |
| name      | varchar(100) | NO   |     | NULL    |                |
| surname   | varchar(100) | YES  |     | NULL    |                |
| birthdate | date         | NO   |     | NULL    |                |
| email     | varchar(50)  | YES  |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+

mysql> describe phone;
+---------------+-------------+------+-----+---------+-------+
| Field         | Type        | Null | Key | Default | Extra |
+---------------+-------------+------+-----+---------+-------+
| number        | varchar(20) | NO   | PRI | NULL    |       |
| id_contact_id | int         | NO   | PRI | NULL    |       |
| type          | varchar(10) | NO   |     | Mobile  |       |
+---------------+-------------+------+-----+---------+-------+
```

4 Queries

Before doing the select queries, you can import the sample data to the database:

```
INSERT INTO `contact` (`id`, `title`, `name`, `surname`, `birthdate`,
↪  `email`) VALUES
(9, 'Miss', 'Lory', 'Grimes', '1967-02-08', 'logrimes@mail.com'),
(11, 'Mr.', 'Mike', 'Molina', '1975-10-21', 'molina@mail.com'),
(12, 'Mrs.', 'Mary Jane', 'Smith', '1986-05-22', 'mj@mail.com'),
(13, 'Mr.', 'Arthur', 'McFly', '1990-08-14', 'mcfly@mail.com');
```

```
INSERT INTO `phone` (`number`, `id_contact_id`, `type`) VALUES
('664444666', 9, 'Work'),
('667889888', 9, 'Mobile'),
('295667788', 11, 'Landline'),
('666557744', 11, 'Mobile'),
('667889900', 13, 'Work');
```

4.1 Select queries

First, we'll modify the controller to select a contact from the database by its `Id`. Go to the `ContactsController` class and remove the `App\Service\ContactData` use and add the classes `Contact` and Phone. We also need the `ManagerRegistry` interface from Doctrine:

```
use Doctrine\Persistence\ManagerRegistry;
use App\Entity\Contact;
use App\Entity\Phone;
```

In the `contact` function we create a persistent object named `$entityManager` via *dependency injection* (we pass a `ManagerRegistryInterface` object as argument to the function), get a `Contact` class repository from it (with the method `getRepository`) and find the contact by its `Id`:

```
#[Route('/contact/{id<\d+>}', name: 'single_contact')]
public function contact(EntityManagerInterface $entityManager, $id=''):
↪  Response
{
    $contact = $entityManager->getRepository(Contact::class)->find($id);
```

```
    return $this->render('contacts/contact.html.twig', [
        'contact' => $contact,
        'page_title' => 'My Contacts App - Contact'
    ]);
}
```

We need to modify the Twig template so that the birth date could be shown properly using a date filter:

```
<li><strong>Birthdate:</strong> {{ contact.birthdate|date('Y-m-d') }}</li>
```

These are the only changes we need to do in the template.

Modify the contactList function is even easier:

```
#[Route('/contact_list', name: 'contact_list')]
public function contactList(EntityManagerInterface $entityManager):
↪   Response
{
    return $this->render('contacts/list.html.twig', [
        'contacts' =>
        ↪   $entityManager->getRepository(Contact::class)->findAll(),
        'page_title' => 'My Contacts App - Contact List'
    ]);
}
```

4.1.1 Searching contacts

We can search a contact in the database by its name or surname. In the `ContactsController` class we need a new route and a function to handle it. Then, we pass to the `list` template a list of contacts obtained by a new method, `findByNameOrSurname`:

```
#[Route('/contact/search/{search_string}', name: 'search_contact')]
public function searchContact(EntityManagerInterface $entityManager,
↪   $search_string=''): Response
{
    return $this->render('contacts/list.html.twig', [
        'contacts' => $entityManager->getRepository(Contact::class)
```

```
            ->findByNameOrSurname($search_string),
        'page_title' => 'My Contacts App - Search results'
    ]);
}
```

Now, in the `ContactRepository` class, we can make the new method:

```
public function findByNameOrSurname($value): ?array
{
    return $this->createQueryBuilder('contact')
        ->andWhere('contact.name LIKE :val OR contact.surname LIKE :val')
        ->setParameter('val', '%'.$value.'%')
        ->getQuery()
        ->getArrayResult()
    ;
}
```

Try to search by a complete name or by letters.

4.2 Inserts

4.2.1 Adding a new contact

For adding a new contact we create a new method in the `ContactsController` class. The route will be `/contact/test/new` (we use the `test` route for testing, the actual inserts, update and deletes will be done in the next unit). At this moment we will add a contact manually, but in the next Unit we will learn how to retrieve the data to be added via API.

```
#[Route('/contact/test/new', name: 'new_contact')]
public function newContact(EntityManagerInterface $entityManager) :
↪   Response {
    $contact = new Contact();
    $contact->setTitle("Mrs.");
    $contact->setName("Carla");
    $contact->setSurname("Fontana");
    $contact->setBirthdate(date_create("1980-01-30"));
    $contact->setEmail("carlafon@mail.com");

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

```
    $entityManager->flush();

    $action = ($contact ? 'New contact added' : 'Failed to add contact');

    return $this->render('contacts/new_edit_contact.html.twig', [
        'contact' => $contact,
        'page_title' => 'My Contacts App - New contact',
        'action' => $action
    ]);
}
```

Once the contact has been inserted, we show a message with the data added. To do that, we can modify a copy of the contact template or we can modify the contact template for working with the 2 routes, adding more parameters. In this case we've tried to do the first option, but adding a new parameter, `action`, for showing a message with the action done. You can see the template code in the GitHub repository.

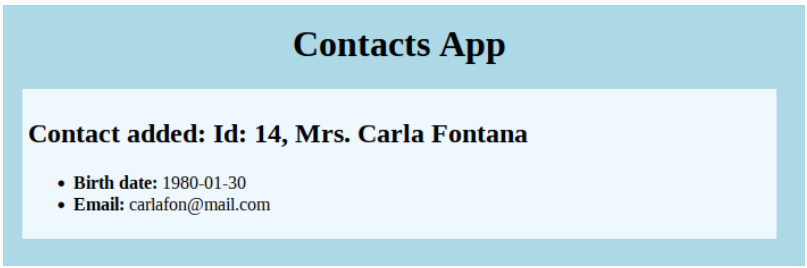

Figure 2: New contact view with the data inserted to the database

4.2.2 Adding a new phone

For adding a new phone to an existing contact, we need to make a new controller:

```
symfony console make:controller PhoneController
```

In the controller, make sure you import the `App\Entity\Phone` and `Contact` classes.

Then, modify the route, the route name and the method name. The code is similar to the previous one, but in this case we want to check if the id belongs to a valid contact:

```php
#[Route('/phone/test/new/{id<\d+>}', name: 'new_phone')]
public function newPhone(EntityManagerInterface $entityManager, $id=''):
    Response
{
    $contact = $entityManager->getRepository(Contact::class)->find($id);
    if($contact == null) {
        return $this->render('phone/new_edit_phone.html.twig', [
            'contact'=> $contact,
            'phone' => null,
            'action' => 'Contact not found',
            'page_title' => 'My Contacts App - New phone',
        ]);
    } else {
        $phone = new Phone();
        $phone->setIdContact($contact);
        $phone->setNumber("656565652");
        $phone->setType("Mobile");

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

        $action = ($phone ? 'New phone added' : 'Failed to add phone');

        return $this->render('phone/new_edit_phone.html.twig', [
            'phone' => $phone,
            'contact' => $contact,
            'action' => $action,
            'page_title' => 'My Contacts App - New phone',
        ]);
    }
}
```

We pass the contact `id` as a parameter to the route and retrieve the contact from the database. If it exists, insert the phone and show the data. Otherwise, show an error message:

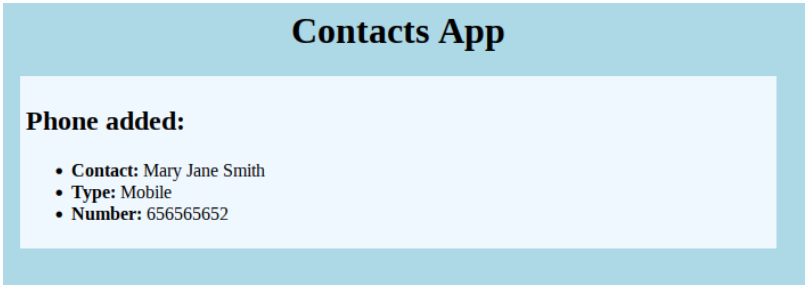

Figure 3: New phone added

4.3 Updates

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

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

We create a new route and method to update a single contact:

```
#[Route('/contact/test/edit/{id<\d+>}', name: 'contact_edit')]
public function updateContact(EntityManagerInterface $entityManager,
↪   $id='') : Response {
    $contact = $entityManager->getRepository(Contact::class)->find($id);
    if($contact) {
        $action = "Contact updated";
        $contact->setName("New Name");
        $entityManager->flush();
    } else {
        $action = "Failed to modify contact";
    }

    return $this->render('contacts/new_edit_contact.html.twig', [
        'contact' => $contact,
        'page_title' => 'My Contacts App - Update contact',
        'action' => $action
    ]);
}
```

We reuse the same template but with a different action message.

Updating a phone is the same procedure, but we need the contact id and the phone number to find the required phone:

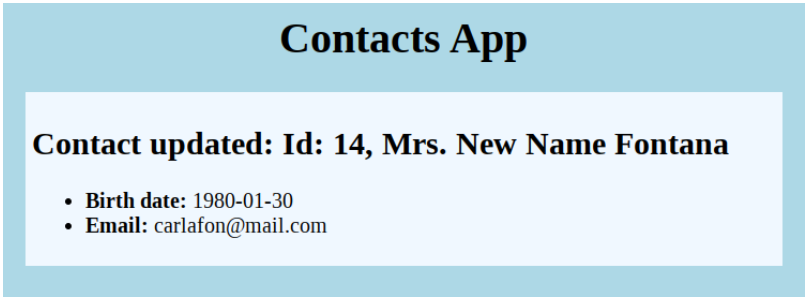

Figure 4: Update contact

```php
#[Route('/phone/test/edit/{id<\d+>}/{number}', name: 'phone_edit')]
public function updatePhone(EntityManagerInterface $entityManager, $id='',
    $number=''): Response
{
    $contact = $entityManager->getRepository(Contact::class)->find($id);
    if($contact == null) {
        return $this->render('phone/new_edit_phone.html.twig', [
            'contact'=> $contact,
            'phone' => null,
            'page_title' => 'My Contacts App - New phone',
            'action' => 'Failed to modify phone: no contact found'
        ]);
    } else {
        $phone = $entityManager->getRepository(Phone::class)
            ->findOneBy(['number'=>$number, 'id_contact'=>$id]);
        if($phone) {
            $phone->setNumber("686868680");
            $entityManager->flush();
            $action = "Phone updated";
        } else {
            $action = "Failed to modify phone";
        }

        return $this->render('phone/new_edit_phone.html.twig', [
            'phone' => $phone,
            'contact' => $contact,
            'page_title' => 'My Contacts App - Update phone',
            'action' => $action
        ]);
    }
}
```

Note that we are fetching the phone with `findOneBy` instead of with `find`.

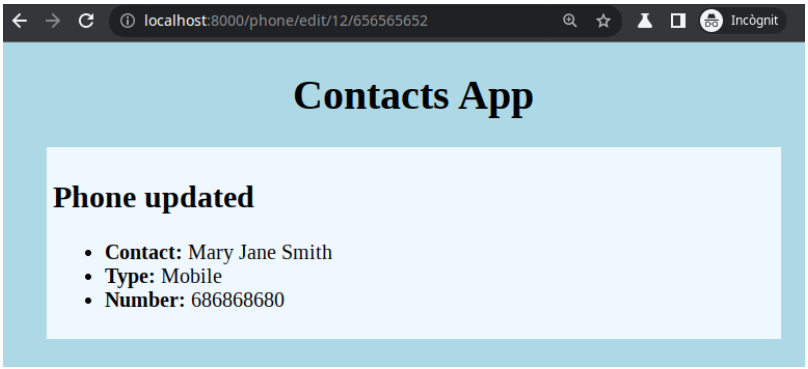

Figure 5: Update phone

4.4 Deletions

Deletions are similar to updates, but they require a call to the method `remove`.

When we delete a contact we delete the associated phones too.

```
#[Route('/contact/test/delete/{id<\d+>}', name: 'contact_delete')]
public function deleteContact(EntityManagerInterface $entityManager,
↪   $id=''): Response {
    $contact = $entityManager->getRepository(Contact::class)->find($id);
    if($contact) {
        //Remove the phones
        $phones = $entityManager->getRepository(Phone::class)
            ->findBy(['id_contact'=>$id]);
        foreach ($phones as $phone){
            $entityManager->remove($phone);
        }
        $entityManager->remove($contact);
        $entityManager->flush();
        $action = "Contact deleted";
    } else {
        $action = "Failed to delete contact";
    }

    return $this->render('contacts/new_edit_contact.html.twig', [
        'contact' => $contact,
        'page_title' => 'My Contacts App - Delete contact',
```

```
        'action' => $action
    ]);
}
```

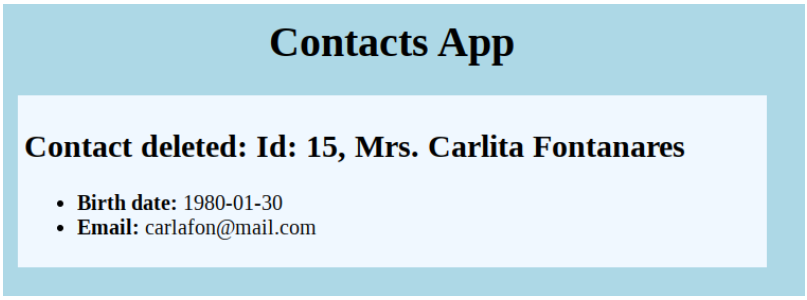

Figure 6: Delete contact

Deleting a phone is similar to updating it:

```
#[Route('/phone/test/delete/{id<\d+>}/{number}', name: 'phone_delete')]
public function deletePhone(EntityManagerInterface $entityManager, $id='',
↪   $number=''): Response
{
    $contact = $entityManager->getRepository(Contact::class)->find($id);
    if($contact == null) {
        return $this->render('phone/new_edit_phone.html.twig', [
            'contact'=> $contact,
            'phone' => null,
            'page_title' => 'My Contacts App - New phone',
            'action' => 'Failed to delete phone: no contact found'
        ]);
    } else {
        $phone = $entityManager->getRepository(Phone::class)
            ->findOneBy(['number'=>$number, 'id_contact'=>$id]);
        if($phone) {
            $entityManager->remove($phone);
            $entityManager->flush();
            $action = "Phone deleted";
        } else {
            $action = "Failed to delete phone";
        }

        return $this->render('phone/new_edit_phone.html.twig', [
            'phone' => $phone,
```

```
            'contact' => $contact,
            'page_title' => 'My Contacts App - Delete phone',
            'action' => $action
        ]);
    }
}
```

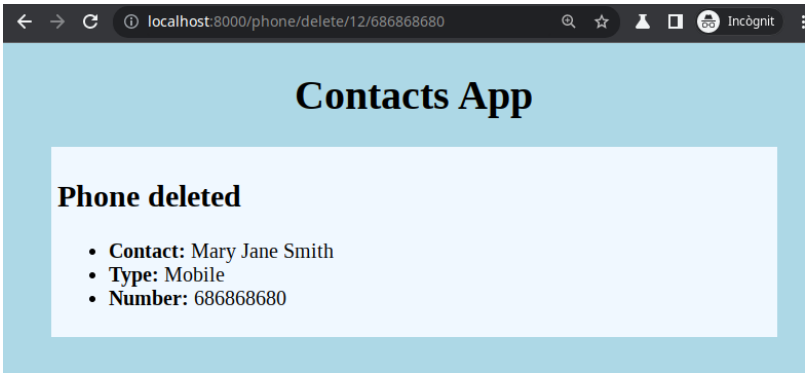

Figure 7: Delete phone

You can get all the code from the GitHub repository.