**Server-side
Web Development**

Unit 13b. Symfony. Forms.

Fidel Oltra, Ricardo Sánchez

Index

1 Creating forms

A form type describes the form fields related to a model. It does the data conversion between the form fields and the model class properties.

To create the form, we have two options. One is using the **createFormBuilder()** method of the **AbstractController** class. As our controller inherits the AbstractController class, we can use the method in the controller. Thus, we need to create an object of the model class prior to create the form.

The other method is creating a specific form class. That's the option that Symfony recommends, because we don't want to put too much logic in our controllers.

1.1 Creating a form in the controller

Run this command to install the form feature before using it:

```
composer require symfony/form
```

Before create the form we need a to instantiate an object used to store and retrieve the data:

```
$contact = new Contact();
```

And then, we can create the form. Let's do it in a new method that we call **contactForm**, associated with the route /contactform.

```
#[Route('/contactform', name: 'app_contactform')]
public function contactForm() {
    $contact=new Contact();
    $form = $this->createFormBuilder($contact)
        ->add('name', TextType::class)
        ->add('phone', TextType::class)
        ->add('email', EmailType::class)
        ->add('save', SubmitType::class, [
            'label' => 'Submit',
            ])
        ->getForm();
}
```

As you can see, the `createFormBuilder` method receives an object as a parameter, and then we can add the fields and the type of every field. These are some of the available types:

Class	Content
TextType	An input of type text
TextAreaType	An input of type textarea (multiple lines)
IntegerType	An input of type integer
NumberType	An input of type number
PasswordType	An input of type password
DateType	An input of type date
CheckBoxType	An input of type checkbox
RadioType	An input of type radiobutton
ChoiceType	A multi-purpose field used to allow the user to "choose" one or more options. It can be rendered as a select tag, radio buttons, or checkboxes.
EmailType	An input of type email
HiddenType	An input of type hidden
EntityType	An input to select a value from other entity
SubmitType	A submit button
ButtonType	A generic button
FormType	A form with another inputs inside

An array with a value for the property **label** can be included in every form item. If not, the default label value will be the name of the model attribute specified in the **add** method.

We need to import the used form types in our controller class. So, in this case, we will add these `use` lines:

```php
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
```

It's important to say that, in Symfony, all of these are "form types":

- a single `<input type="text">` form field is a "form type" (`TextType`)
- a group of HTML fields is a "form type" (e.g. `DateTimeType`)
- an entire `<form>` with multiple fields is a "form type".

This may be confusing at first, but it simplifies code and makes "composing" and "embedding" form fields much easier to implement. For instance, you can embed a form in another form as a **_subform_**.

More about form types in Symfony following this link:

Form Types Reference

1.2 Displaying a form

As we know, we can't display anything in a controller. The user interface is always displayed in a view. So a form has to be rendered to a view in order to be displayed.

First, let's prepare the view. Is very easy to display a form. In a template named `contact-form.html.twig` we write this:

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

{% block title %}New Contact{% endblock %}
{% block body %}
    <h1>New Contact</h1>
    {{ form(form) }}
{% endblock %}
```
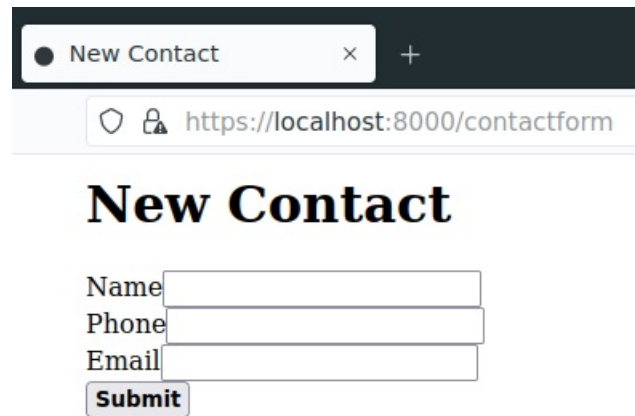
That's enough to show the form, but we need to render the from to the view from the controller. So we add these line to the controller method:

```
return $this->render('contact/contactform.html.twig', [
    'form' => $form->createView()
]);
```

We render the template sending the form as a parameter to the view.

Now, let's go to the route `/contactform` to see if it works:

The `EmailType` avoids entering a string that doesn't meet the email format.
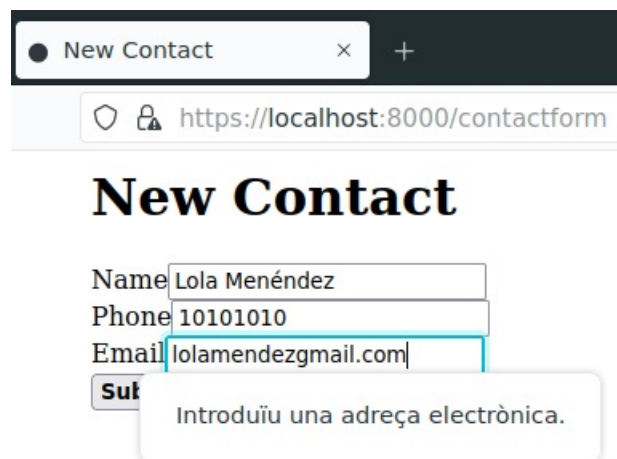
Figure 1: Our form on screen

Figure 2: Invalid email

1.3 Submitting the form

By default, the form sends the data to the same controller method where has been called from. So when we click on the **Submit** button, the form appears again.

If we want to do something with the data sent by the form, we need to handle the received request. So, in the controller, we need to import the Request class:

```
use Symfony\Component\HttpFoundation\Request;
```

Then we will pass a Request object as a parameter to the method:

```
public function functionName(Request $request) {
```

Now we can handle the request in our method:

```
$form->handleRequest($request);
```

And then we will ask if the form has been submitted and validated:

```
if($form->isSubmitted() && $form->isValid()) {
    ...
}
```

Now we can read the data to an object of the associated class:

```
$contact=$form->getData();
```

Finally, we can do whatever we want with the submitted contact: insert, remove, update, show the contact card…

1.4 Inserting a new contact using the form

Let's go back to our `/contact/new` route. At the moment we have this code:

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

    $entityManager=$doctrine->getManager();
    $entityManager->persist($contact);
    $entityManager->flush();

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

Let's change the method to insert the contact submitted by a form. First, let's remove the parameters from the route.

```
#[Route('/contact/new', name: 'app_newcontact')]
public function newContact(ManagerRegistry $doctrine): Response
```

Now, we create a new contact and a form associated with the object, and then we render the previously created view:

```
#[Route('/contact/new', name: 'app_newcontact')]
public function newContact(ManagerRegistry $doctrine): Response {
    $contact=new Contact();
    $form = $this->createFormBuilder($contact)
        ->add('name', TextType::class)
        ->add('phone', TextType::class)
        ->add('email', EmailType::class)
        ->add('save', SubmitType::class, ['label' => 'Submit'])
        ->getForm();

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

But now we need to read the data submitted by the form, so we ask for the request and check if the form has been submitted and is valid. If so, we read the contact submitted by the form. Note that we have added the parameter **Request $request** to the function. Previously, we added the **use** clause for the import of the `Request` class.

```php
#[Route('/contact/new', name: 'app_newcontact')]
public function newContact(ManagerRegistry $doctrine, Request $request):
↪   Response {
    $contact=new Contact();
    $form = $this->createFormBuilder($contact)
        ->add('name', TextType::class)
        ->add('phone', TextType::class)
        ->add('email', EmailType::class)
        ->add('save', SubmitType::class, ['label' => 'Submit'])
        ->getForm();

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $contact=$form->getData();
    }

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

And finally, if we want to insert the contact into the table, we proceed as usual. This is how the method looks at the end of the whole process.

```php
#[Route('/contact/new', name: 'app_newcontact')]
public function newcontact(ManagerRegistry $doctrine, Request $request):
↪   Response {
    $contact=new Contact();
    $form = $this->createFormBuilder($contact)
        ->add('name', TextType::class)
        ->add('phone', TextType::class)
        ->add('email', EmailType::class)
        ->add('save', SubmitType::class, ['label' => 'Submit'])
        ->getForm();

    $form->handleRequest($request);
```
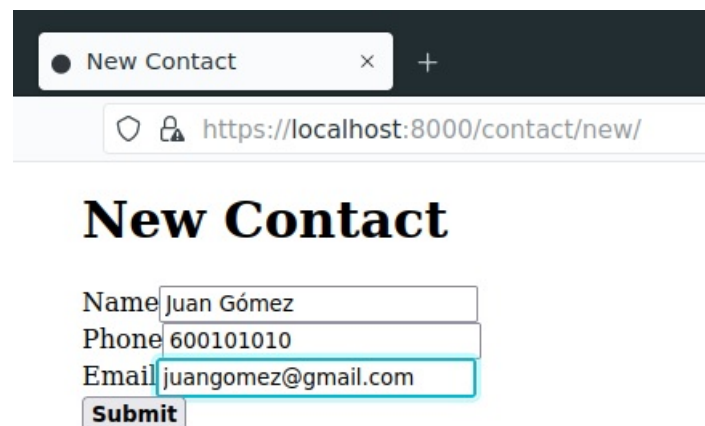
```php
    if($form->isSubmitted() && $form->isValid()) {
        $contact=$form->getData();
        $entityManager=$doctrine->getManager();
        $entityManager->persist($contact);
        $entityManager->flush();
        return $this->redirectToRoute('app_contacts');
    }

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

With the line

```php
return $this->redirectToRoute('app_contacts');
```

We are calling the method that show the contact list. We can't render the twig directly because the view needs the $contact array that the method sends. So we use the function **redirectToRoute** to call a controller using the attribute **name** of the associated route.

Finally, let's go to the /contact/new route. The form appears, and we enter the new contact:

Figure 3: Entering a new contact
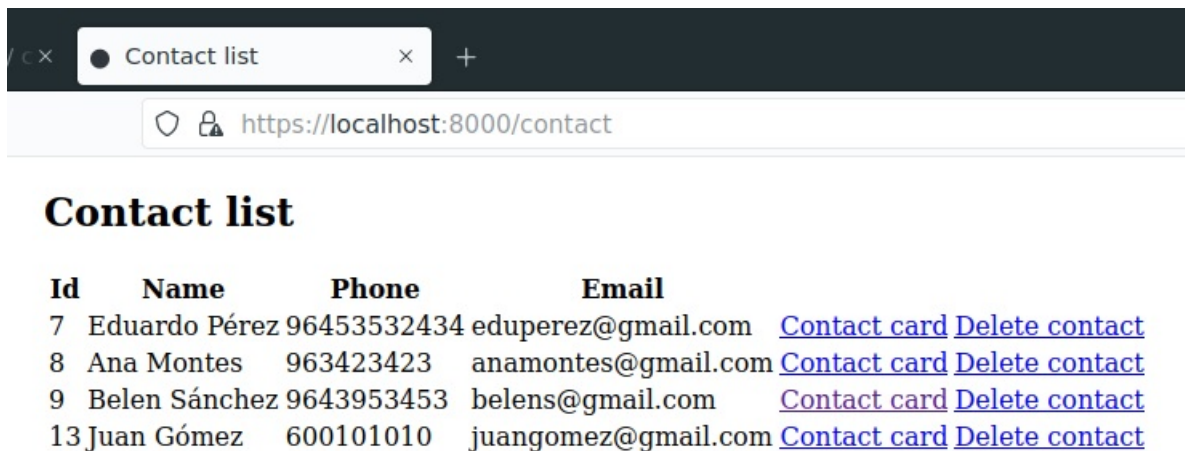
Now we press the Submit button and…

Figure 4: Contact added to the list

1.5 Updating a contact using the form

To update a contact, as we know, we just need to recover the contact from the database, then we change the required data, and finally with the **flush()** method the changes are saved to the database.

The only thing we need to change in our updating method is that the new data will be read from the form instead of manually written.

First, let's go to our update method. Now it's associated to the route `/contact/update/{id}/{newname}`, but we are going to change that as you can see below:

```
#[Route('/contact/change/{id<\d+>}', name: 'app_changecontact')]
public function changecontact(ManagerRegistry $doctrine, Request $request,
↪  $id) {
```

Then we create the repository and search the contact with the `$id` primary key.

```
$rep=$doctrine->getRepository(Contact::class);
$contact=$rep->find($id);
```

Now we create the form with the `$contact` object. In this case, as we are updating, we need to include the `$id` as a hidden field in the form.

```
$form = $this->createFormBuilder($contact)
    ->add('id', HiddenType::class)
    ->add('name', TextType::class)
    ->add('phone', TextType::class)
    ->add('email', EmailType::class)
    ->add('save', SubmitType::class, array('label' => 'Submit'))
    ->getForm();
```

We need to add a **setId** method in the entity, as well. So in the Contact.php file we add the method:

```
public function setId(int $id): self {
    $this->id = $id;
    return $this;
}
```

Finally, in the controller again, we need to read the request and, if the form has been submitted and is valid, do the update operation as usual:

```
$form->handleRequest($request);
if($form->isSubmitted() && $form->isValid()) {
    $contact=$form->getData();
    $entityManager=$doctrine->getManager();
    $entityManager->flush();
    return $this->redirectToRoute('contact');
}
```

The last line in the method is the rendering of the view with the form.

```
return $this->render('contact/contactform.html.twig', array('form' =>
↪    $form->createView()));
```

This is how the method looks at the end:

```
#[Route('/contact/change/{id<\d+>}', name: 'app_changecontact')]
public function changeContact(ManagerRegistry $doctrine, Request $request,
↪    $id) {
    $rep=$this->getDoctrine()->getRepository(Contact::class);
    $contact=$rep->find($id);
```

```php
    $form = $this->createFormBuilder($contact)
        ->add('id', HiddenType::class)
        ->add('name', TextType::class)
        ->add('phone', TextType::class)
        ->add('email', EmailType::class)
        ->add('save', SubmitType::class, array('label' => 'Submit'))
        ->getForm();

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $contact=$form->getData();
        $entityManager=$this->getDoctrine()->getManager();
        $entityManager->flush();
        return $this->redirectToRoute('contact');
    }

    return $this->render('contact/contactform.html.twig', array('form' =>
    ↪   $form->createView()));
}
```

Just a little detail: let's change the title in the view with the form to show a different message when we are adding a new contact and when we are updating an existing one. So in the `contact-form.html.twig` template we change the title block:

```twig
{% block title %}{{ title }}{% endblock %}
```
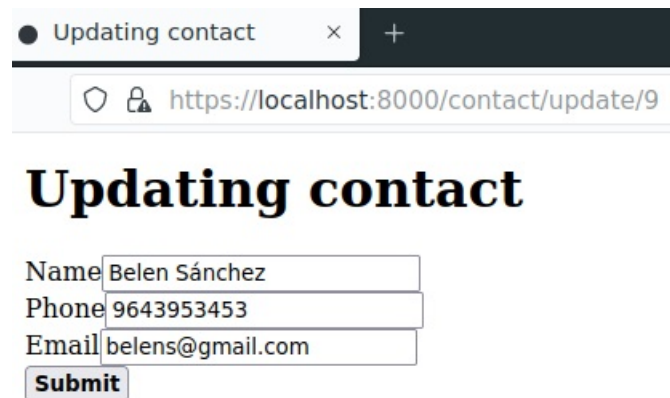
and the h1 tag:

```html
<h1>{{ title }}</h1>
```

So now we need to give a value to the title parameter when we render the view.

```php
return $this->render('contact/contactform.html.twig', array(
    'form' => $form->createView(),
    'title'=> "Updating contact"));
```

Now let's modify a contact.
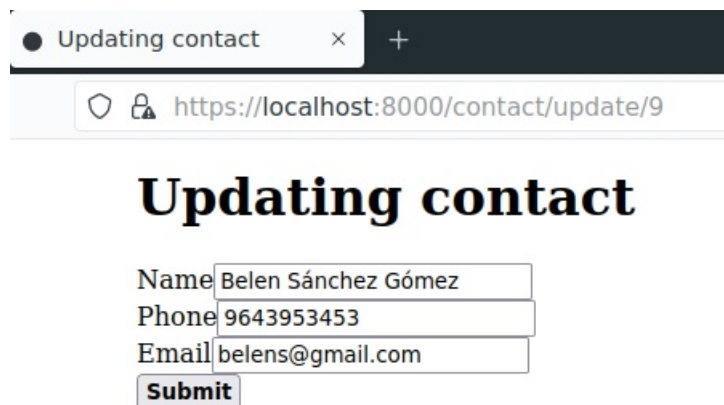
Figure 5: Updating a contact

We change the name:

Figure 6: Changing the name

And submit the changes:

Figure 7: The contact has been updated!

1.6 Creating a form in a Form class

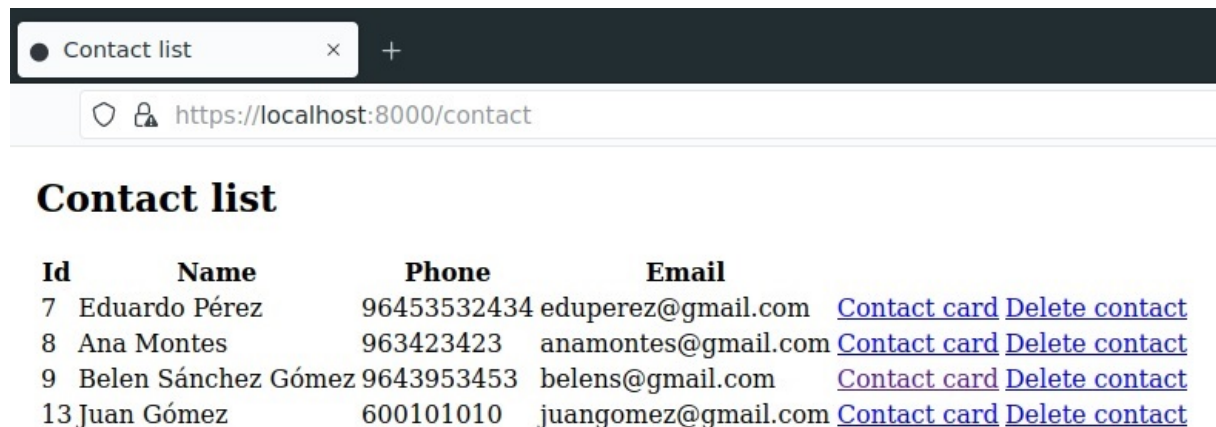Form classes are form types that implement ***FormTypeInterface***. However, it's better to extend from ***AbstractType***, which already implements the interface and provides some additional utilities.

We can create a Form class with the `make:form` and `make:registration-form` commands. If we do

```
symfony console make:form
```

We'll be asked for the form name and the name of the entity associated to the form.

```
 The name of the form class (e.g. BraveGnomeType):
 > Contact

 The name of Entity or fully qualified model class name that the new form
↪  will be bound to (empty for none):
 > Contact
```

We get a new folder named `Form` and this ***ContactType*** php file:

```
namespace App\Form;

use App\Entity\Contact;
```

```php
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ContactType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options):
    ↪    void {
        $builder
            ->add('name')
            ->add('phone')
            ->add('email')
        ;
    }

    public function configureOptions(OptionsResolver $resolver): void {
        $resolver->setDefaults([
            'data_class' => Contact::class,
        ]);
    }
}
```

We should add the field types, import them with the use clauses, and the Save button:

```php
namespace App\Form;

use App\Entity\Contact;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;

class ContactType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options):
    ↪    void {
        $builder
            ->add('name',TextType::class)
            ->add('phone',TextType::class)
            ->add('email',EmailType::class)
            ->add('Save',SubmitType::class)
```

```
        ;
    }

    public function configureOptions(OptionsResolver $resolver): void {
        $resolver->setDefaults([
            'data_class' => Contact::class,
        ]);
    }
}
```

> **Note**: If you don't write the type class for each field, Symfony can guess the type of your field.

And then, in the controller, instead of creating the form doing this:

```
$form = $this->createFormBuilder($contact)
    ->add('name', TextType::class)
    ->add('phone', TextType::class)
    ->add('email', EmailType::class)
    ->add('save', SubmitType::class, ['label' => 'Submit'])
    ->getForm();
```

We just create the form with this single line:

```
$form=$this->createForm(ContactType::class, $contact);
```

More information about forms with classes following this link

1.7 Relations in forms

Let's make a new form for the *city* entity. Let's make a controller method to add a new city with a form.

```
#[Route('/city/new', name: 'app_newcity')]
public function newCity(ManagerRegistry $doctrine, Request $request):
↪    Response
{
    $city=new City();
    $form = $this->createFormBuilder($city)
```

```
        ->add('postalcode', TextType::class,
        array('attr' => ['pattern' => '[0-9]{5}',
            'maxlength' => 5],
            'label'=>'Postal Code'))
        ->add('cityname', TextType::class, array('label'=>"City name"))
        ->add('save', SubmitType::class, array('label' => 'Submit'))
        ->getForm();

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $city=$form->getData();
        $entityManager=$doctrine->getManager();
        $entityManager->persist($city);
        $entityManager->flush();
        return $this->redirectToRoute('app_city');
    }

    return $this->render('city/cityform.html.twig', array(
        'form' => $form->createView(),
        'title'=> "New city"));
}
```

The template with the form:

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

{% block title %}{{ title }}{% endblock %}
{% block body %}
    <h1>{{ title }}</h1>
{{ form(form) }}
{% endblock %}
```

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 8: Adding a new city

Figure 9: Cities list

Go to the **Contact** class and add the lines:

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

Now, in the **City** class we add a new method called `__toString` that returns the value we want to show in our form when we choose a city. In our case, *cityname*.

```
public function __toString(): String {
        return $this->cityname;
}
```

And now, in our **Contact** form, we add a new property: the city.

If you are working in the controller, with `createFormBuilder`, do this:

```
$form = $this->createFormBuilder($contact)
    ->add('name', TextType::class)
    ->add('phone', TextType::class)
    ->add('email', EmailType::class)
    // this is the new line
    ->add('city', EntityType::class, array('class' => City::class))
    //
    ->add('save', SubmitType::class, array('label' => 'Submit'))
    ->getForm();
```

In case that you are using a Form class, do this in the `\src\Form\ContactType.php` file:

```
 public function buildForm(FormBuilderInterface $builder, array $options):
 ↪   void
{
    $builder
        ->add('name',TextType::class)
        ->add('phone',TextType::class)
        ->add('email',EmailType::class)
        ->add('city',EntityType::class ['class'=>City::class]) // THIS
        ->add('Save',SubmitType::class)
    ;
}
```

We have created a new form property, called 'city'. This property will be an object of the **City** class that we will be able to select with a form selector. Let's see how it works:

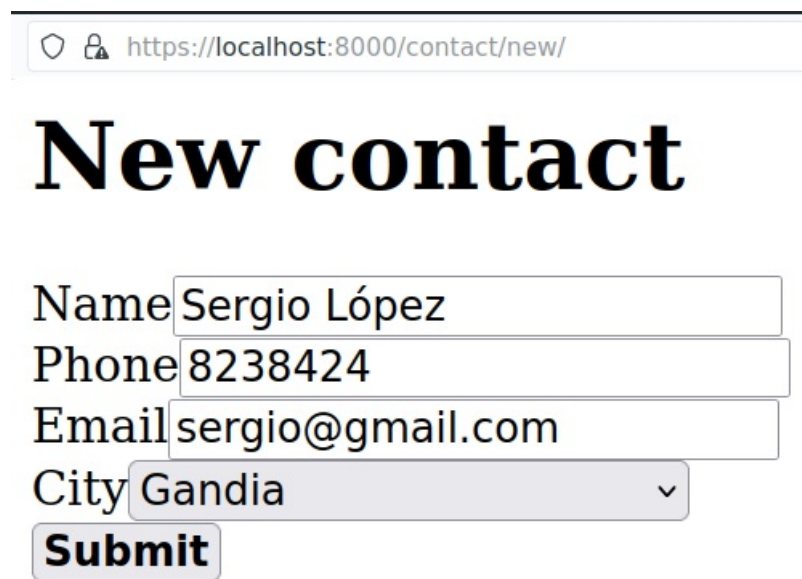**Figure 10:** Selecting the city for our new contact

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

```
$contact=$form->getData();
$entityManager=$this->getDoctrine()->getManager();
$entityManager->persist($contact);
$entityManager->flush();
```

Figure 11: New contact

Figure 12: 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 }}</td>
```

The property *city* will return the name of the city, because that is what the method **__toString()** of the **City** class is returning.

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 13: 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.

1.8 Adding Bootstrap styles

The Symfony rendering way is not very flexible. Usually, you'll need more control about how the entire form or some of its fields look. Luckily, **Bootstrap** has a good integration with Symfony forms. You can set this option to generate forms compatible with the Bootstrap 5 CSS framework in the `config/packages/twig.yaml` config file:

```
# config/packages/twig.yaml
twig:
    form_themes: ['bootstrap_5_layout.html.twig']
```

Then, add the bootstrap links to your base template:

```
{# templates/base.html.twig #}

{# beware that the blocks in your template may be named different #}
{% block stylesheets %}
    <!-- Copy CSS from
    https://getbootstrap.com/docs/5.0/getting-started/introduction/#css -->
    <!-- Link to your styles here -->
{% endblock %}
{% block javascripts %}
```

```
    <!-- Copy JavaScript from
↪   https://getbootstrap.com/docs/5.0/getting-started/introduction/#js -->
{% endblock %}
```

[Symfony forms documentation](#)

2 Form validation

In Symfony, the validation is done in the underlying objects ($contact and $city in this example). Symfony checks if they are valid after the form has applied the submitted data to it. Calling $form->isValid() is a shortcut that asks the object if it has valid data.

If you have created your project as a *full webapp*, you have already the validation support installed. Otherwise, add it by doing:

```
composer require symfony/validator
```

Validation is done by adding a set of rules, called **constraints**, to a class. You can add them either to the entity class or to the form class.

We are going to do so in the classes. The first thing is to import the validation constraints:

```
use Symfony\Component\Validator\Constraints as Assert;
```

Then, add the constraints as **asserts** to each property with annotations:

```
#[ORM\Column(length: 50)]
#[Assert\NotBlank]
private ?string $name = null;

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

#[ORM\Column(length: 60)]
#[Assert\Email]
private ?string $email = null;
```

```
#[ORM\Column(length: 5)]
#[Assert\NotBlank]
#[Assert\Regex('/[0-9]{5}/')]
private ?string $postalcode = null;

#[ORM\Column(length: 50)]
#[Assert\NotBlank]
private ?string $cityname = null;
```

Done! If you submit the form with invalid data, you'll see the corresponding errors printed out with the form.

If you want you can customize the error message:

```
#[Assert\Email(
    message: 'The email {{ value }} is not a valid email.',
)]
private ?string $email = null;
```

Here, a list of all the constraints: Symfony validation constraints

2.1 Disabling client-side validation

Generated forms take full advantage of the client-side validation done in the browsers. Symfony automatically adds the HTML validation attributes and types ('required', type='email'...). This feature will cause you to never see the server-side error messages. This is especially useful when you want to test your server-side validation constraints.

The client-side validation can be disabled by adding the novalidate attribute to the <form> tag or formnovalidate to the submit tag or the 'required' => false option to the form type class. Let's see one example for each one:

```
{# Disabling validation in the template #}
{{ form_start(form, {'attr': {'novalidate': 'novalidate'}}) }}
    {{ form_widget(form) }}
{{ form_end(form) }}
```

```php
// Disabling validation when building the form
$builder
        ...
        //this only works for this button
        ->add('Save',SubmitType::class, [
            'attr' => [
                'formnovalidate' => 'formnovalidate',
            ]
        ])
```

```php
// ContactType class
public function configureOptions(OptionsResolver $resolver): void
{
    $resolver->setDefaults([
        'data_class' => Contact::class,
        'required' => false,
    ]);
}
```

2.2 Troubleshooting with validation

If you get an error message like: *Expected argument of type "string", "null" given at property path...* the reason is because Symfony tries to use a getter method that not accepts null values. Try to change the type of the argument to null or string:

```php
public function setName(?string $name)
```

If you get an error of type *SQL exception*, use try-catch to manage the exception and to change the types to the correct ones.

Another problem could happen when you try to validate an email in the server but you can't because the input is of type *email*. You can solve this by changing the type in the Type class to TextType instead of EmailType:

```php
->add('email', TextType::class)
```

More information about validation in form classes:

Validating forms