# Forms

2.7 version

edit this page

Dealing with HTML forms is one of the most common – and challenging – tasks for a web developer. Symfony integrates a Form component that makes dealing with forms easy. In this chapter, you'll build a complex form from the ground up, learning the most important features of the form library along the way.

The Symfony Form component is a standalone library that can be used outside of Symfony projects. For more information, see the Form component documentation on GitHub.

## Creating a Simple Form ¶

Suppose you're building a simple todo list application that will need to display "tasks". Because your users will need to edit and create tasks, you're going to need to build a form. But before you begin, first focus on the generic `Task` class that represents and stores the data for a single task:

```
1  // src/AppBundle/Entity/Task.php
2  namespace AppBundle\Entity;
3
4  class Task
5  {
6      protected $task;
7      protected $dueDate;
8
9      public function getTask()
10     {
11         return $this->task;
12     }
13
14     public function setTask($task)
15     {
16         $this->task = $task;
17     }
18
19     public function getDueDate()
20     {
21         return $this->dueDate;
22     }
23
```

```
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

This class is a "plain-old-PHP-object" because, so far, it has nothing to do with Symfony or any other library. It's quite simply a normal PHP object that directly solves a problem inside your application (i.e. the need to represent a task in your application). Of course, by the end of this chapter, you'll be able to submit data to a `Task` instance (via an HTML form), validate its data, and persist it to the database.

## Building the Form ¶

Now that you've created a `Task` class, the next step is to create and render the actual HTML form. In Symfony, this is done by building a form object and then rendering it in a template. For now, this can all be done from inside a controller:

```
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use AppBundle\Entity\Task;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
    {
        // create a task and give it some dummy data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->add('save', 'submit', array('label' => 'Create Task'))
            ->getForm();

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

> This example shows you how to build your form directly in the controller. Later, in the "Creating Form Classes" section, you'll learn how to build your form in a standalone class, which is recommended as your form becomes reusable.

Creating a form requires relatively little code because Symfony form objects are built with a "form builder". The form builder's purpose is to allow you to write simple form "recipes", and have it do all the heavy-lifting of actually building the form.

In this example, you've added two fields to your form - `task` and `dueDate` - corresponding to the `task` and `dueDate` properties of the `Task` class. You've also assigned each a "type" (e.g. `text`, `date`), which, among other things, determines which HTML form tag(s) is rendered for that field.

Finally, you added a submit button with a custom label for submitting the form to the server.

> 2.3 Support for submit buttons was introduced in Symfony 2.3. Before that, you had to add buttons to the form's HTML manually.

Symfony comes with many built-in types that will be discussed shortly (see Built-in Field Types).

## Rendering the Form ¶

Now that the form has been created, the next step is to render it. This is done by passing a special form "view" object to your template (notice the `$form->createView()` in the controller above) and using a set of form helper functions:

Twig PHP

```twig
{# app/Resources/views/default/new.html.twig #}
{{ form_start(form) }}
{{ form_widget(form) }}
{{ form_end(form) }}
```

is example assumes that you submit the form in a "POST" request and to the same URL that it was displayed in. You will learn later how to change the request method and the target URL of the form.

That's it! Just three lines are needed to render the complete form:

`form_start(form)`
Renders the start tag of the form, including the correct enctype attribute when using file uploads.

`form_widget(form)`
Renders all the fields, which includes the field element itself, a label and any validation error messages for the field.

`form_end(form)`
Renders the end tag of the form and any fields that have not yet been rendered, in case you rendered each field yourself. This is useful for rendering hidden fields and taking advantage of the automatic CSRF Protection.

easy as this is, it's not very flexible (yet). Usually, you'll want to render each form field individually so you can control how the form looks. You'll learn how to do that in the "Rendering a Form in a Template" section.

Before moving on, notice how the rendered `task` input field has the value of the `task` property from the `$task` object (i.e. "Write a blog post"). This is the first job of a form: to take data from an object and translate it into a format that's suitable for being rendered in an HTML form.

e form system is smart enough to access the value of the protected `task` property via the `getTask()` and `setTask()` methods on the `Task` class. Unless a property is public, it must have a "getter" and "setter" method so that the Form component can get and put data onto the property. For a boolean property, you can use an "isser" or "hasser" method (e.g. `isPublished()` or `hasReminder()`) instead of a getter (e.g. `getPublished()` or `getReminder()`).

## Handling Form Submissions ¶

The second job of a form is to translate user-submitted data back to the properties of an object.

To make this happen, the submitted data from the user must be written into the form. Add the following functionality to your controller:

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

public function newAction(Request $request)
{
    // just setup a fresh $task object (remove the dummy data)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', 'text')
        ->add('dueDate', 'date')
        ->add('save', 'submit', array('label' => 'Create Task'))
        ->getForm();

    $form->handleRequest($request);

    if ($form->isValid()) {
        // ... perform some action, such as saving the task to the dat

        return $this->redirectToRoute('task_success');
    }

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

aware that the `createView()` method should be called after `handleRequest` is called. Otherwise, changes done in the `*_SUBMIT` events aren't applied to the view (like validation errors).

2.3 The `handleRequest()` method was introduced in Symfony 2.3. Previously, the `$request` was passed to the `submit` method - a strategy which is deprecated and will be removed in Symfony 3.0. For details on that method, see Passing a Request to Form::submit() (Deprecated).

This controller follows a common pattern for handling forms, and has three possible paths:

1. When initially loading the page in a browser, the form is simply created and

rendered. `handleRequest()` recognizes that the form was not submitted and does nothing. `isValid()` returns `false` if the form was not submitted.

2. When the user submits the form, `handleRequest()` recognizes this and immediately writes the submitted data back into the `task` and `dueDate`properties of the `$task` object. Then this object is validated. If it is invalid (validation is covered in the next section), `isValid()` returns`false` again, so the form is rendered together with all validation errors;

   u can use the method `isSubmitted()` to check whether a form was submitted, regardless of whether or not the submitted data is actually valid.

3. When the user submits the form with valid data, the submitted data is again written into the form, but this time `isValid()` returns `true`. Now you have the opportunity to perform some actions using the `$task`object (e.g. persisting it to the database) before redirecting the user to some other page (e.g. a "thank you" or "success" page).

   directing a user after a successful form submission prevents the user from being able to hit the "Refresh" button of their browser and re-post the data.

you need more control over exactly when your form is submitted or which data is passed to it, you can use the `submit()` for this. Read more about it in the cookbook.

## Submitting Forms with Multiple Buttons ¶

2.3 Support for buttons in forms was introduced in Symfony 2.3.

When your form contains more than one submit button, you will want to check which of the buttons was clicked to adapt the program flow in your controller. To do this, add a second button with the caption "Save and add" to your form:

```
1  $form = $this->createFormBuilder($task)
2      ->add('task', 'text')
3      ->add('dueDate', 'date')
4      ->add('save', 'submit', array('label' => 'Create Task'))
```

```
    ->add('saveAndAdd', 'submit', array('label' => 'Save and Add'))
    ->getForm();
```

In your controller, use the button's `isClicked()` method for querying if the "Save and add" button was clicked:

```
if ($form->isValid()) {
    // ... perform some action, such as saving the task to the databas

    $nextAction = $form->get('saveAndAdd')->isClicked()
        ? 'task_new'
        : 'task_success';

    return $this->redirectToRoute($nextAction);
}
```

# Form Validation ¶

In the previous section, you learned how a form can be submitted with valid or invalid data. In Symfony, validation is applied to the underlying object (e.g. `Task`). In other words, the question isn't whether the "form" is valid, but whether or not the `$task` object is valid after the form has applied the submitted data to it. Calling `$form->isValid()` is a shortcut that asks the `$task` object whether or not it has valid data.

Validation is done by adding a set of rules (called constraints) to a class. To see this in action, add validation constraints so that the `task` field cannot be empty and the `dueDate` field cannot be empty and must be a valid DateTime object.

Annotations YAML XML PHP

```
// src/AppBundle/Entity/Task.php
namespace AppBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Task
{
    /**
     * @Assert\NotBlank()
     */
    public $task;

    /**
```

```
14          * @Assert\NotBlank()
15          * @Assert\Type("\DateTime")
16          */
17         protected $dueDate;
18     }
```

That's it! If you re-submit the form with invalid data, you'll see the corresponding errors printed out with the form.

### TML5 Validation

As of HTML5, many browsers can natively enforce certain validation constraints on the client side. The most common validation is activated by rendering a `required` attribute on fields that are required. For browsers that support HTML5, this will result in a native browser message being displayed if the user tries to submit the form with that field blank.

Generated forms take full advantage of this new feature by adding sensible HTML attributes that trigger the validation. The client-side validation, however, can be disabled by adding the `novalidate`attribute to the `form` tag or `formnovalidate` to the submit tag. This is especially useful when you want to test your server-side validation constraints, but are being prevented by your browser from, for example, submitting blank fields.

Twig PHP

```
1  {# app/Resources/views/default/new.html.twig #}
2  {{ form(form, {'attr': {'novalidate': 'novalidate'}}) }}
```

Validation is a very powerful feature of Symfony and has its own dedicated chapter.

## Validation Groups ¶

If your object takes advantage of validation groups, you'll need to specify which validation group(s) your form should use:

```
1  $form = $this->createFormBuilder($users, array(
2      'validation_groups' => array('registration'),
3  ))->add(...);
```

2.7 The `configureOptions()` method was introduced in Symfony 2.7. Previously, the method was called `setDefaultOptions()`.

If you're creating form classes (a good practice), then you'll need to add the following to the `configureOptions()` method:

```
use Symfony\Component\OptionsResolver\OptionsResolver;

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array('registration'),
    ));
}
```

In both of these cases, only the `registration` validation group will be used to validate the underlying object.

## Disabling Validation ¶

2.3 The ability to set `validation_groups` to false was introduced in Symfony 2.3.

Sometimes it is useful to suppress the validation of a form altogether. For these cases you can set the `validation_groups` option to `false`:

```
use Symfony\Component\OptionsResolver\OptionsResolver;

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => false,
    ));
}
```

Note that when you do that, the form will still run basic integrity checks, for example whether an uploaded file was too large or whether non-existing fields were submitted. If you want to suppress validation, you can use the POST_SUBMIT event.

## Groups based on the Submitted Data ¶

If you need some advanced logic to determine the validation groups (e.g. based on submitted data), you can set the `validation_groups` option to an array callback:

```
use Symfony\Component\OptionsResolver\OptionsResolver;
```

```
2
3   // ...
4   public function configureOptions(OptionsResolver $resolver)
5   {
6       $resolver->setDefaults(array(
7           'validation_groups' => array(
8               'AppBundle\Entity\Client',
9               'determineValidationGroups',
10          ),
11      ));
12  }
```

This will call the static method `determineValidationGroups()` on the `Client`class after the form is submitted, but before validation is executed. The Form object is passed as an argument to that method (see next example). You can also define whole logic inline by using a `Closure`:

```
1   use AppBundle\Entity\Client;
2   use Symfony\Component\Form\FormInterface;
3   use Symfony\Component\OptionsResolver\OptionsResolver;
4
5   // ...
6   public function configureOptions(OptionsResolver $resolver)
7   {
8       $resolver->setDefaults(array(
9           'validation_groups' => function (FormInterface $form) {
10              $data = $form->getData();
11
12              if (Client::TYPE_PERSON == $data->getType()) {
13                  return array('person');
14              }
15
16              return array('company');
17          },
18      ));
19  }
```

Using the `validation_groups` option overrides the default validation group which is being used. If you want to validate the default constraints of the entity as well you have to adjust the option as follows:

```
1   use AppBundle\Entity\Client;
2   use Symfony\Component\Form\FormInterface;
3   use Symfony\Component\OptionsResolver\OptionsResolver;
4
5   // ...
6   public function configureOptions(OptionsResolver $resolver)
```

```
{
    $resolver->setDefaults(array(
        'validation_groups' => function (FormInterface $form) {
            $data = $form->getData();

            if (Client::TYPE_PERSON == $data->getType()) {
                return array('Default', 'person');
            }

            return array('Default', 'company');
        },
    ));
}
```

You can find more information about how the validation groups and the default constraints work in the book section about validation groups.

## Groups based on the Clicked Button ¶

2.3 Support for buttons in forms was introduced in Symfony 2.3.

When your form contains multiple submit buttons, you can change the validation group depending on which button is used to submit the form. For example, consider a form in a wizard that lets you advance to the next step or go back to the previous step. Also assume that when returning to the previous step, the data of the form should be saved, but not validated.

First, we need to add the two buttons to the form:

```
$form = $this->createFormBuilder($task)
    // ...
    ->add('nextStep', 'submit')
    ->add('previousStep', 'submit')
    ->getForm();
```

Then, we configure the button for returning to the previous step to run specific validation groups. In this example, we want it to suppress validation, so we set its `validation_groups` option to false:

```
$form = $this->createFormBuilder($task)
    // ...
    ->add('previousStep', 'submit', array(
        'validation_groups' => false,
    ))
    ->getForm();
```

Now the form will skip your validation constraints. It will still validate basic integrity constraints, such as checking whether an uploaded file was too large or whether you tried to submit text in a number field.

# Built-in Field Types ¶

Symfony comes standard with a large group of field types that cover all of the common form fields and data types you'll encounter:

## Text Fields ¶

- text
- textarea
- email
- integer
- money
- number
- password
- percent
- search
- url

## Choice Fields ¶

- choice
- entity
- country
- language
- locale
- timezone
- currency

## Date and Time Fields ¶

- date

- datetime
- time
- birthday

## Other Fields ¶

- checkbox
- file
- radio

## Field Groups ¶

- collection
- repeated

## Hidden Fields ¶

- hidden

## Buttons ¶

- button
- reset
- submit

## Base Fields ¶

- form

You can also create your own custom field types. This topic is covered in the "How to Create a Custom Form Field Type" article of the cookbook.

## Field Type Options ¶

Each field type has a number of options that can be used to configure it. For example, the `dueDate` field is currently being rendered as 3 select boxes. However, the date field can be configured to be rendered as a single text box (where the user would enter the date as a string in the box):

```
->add('dueDate', 'date', array('widget' => 'single_text'))
```


Each field type has a number of different options that can be passed to it. Many of these are specific to the field type and details can be found in the documentation for each type.

### The required Option

The most common option is the `required` option, which can be applied to any field. By default, the `required` option is set to `true`, meaning that HTML5-ready browsers will apply client-side validation if the field is left blank. If you don't want this behavior, either set the `required` option on your field to `false` or disable HTML5 validation.

Also note that setting the `required` option to `true` will not result in server-side validation to be applied. In other words, if a user submits a blank value for the field (either with an old browser or web service, for example), it will be accepted as a valid value unless you use Symfony's `NotBlank` or `NotNull` validation constraint.

In other words, the `required` option is "nice", but true server-side validation should always be used.

### The label Option

The label for the form field can be set using the `label` option, which can be applied to any field:

```
->add('dueDate', 'date', array(
    'widget' => 'single_text',
    'label'  => 'Due Date',
))
```

The label for a field can also be set in the template rendering the form, see below. If you don't need a label associated to your input, you can disable it by setting its value to `false`.

# Field Type Guessing ¶

Now that you've added validation metadata to the `Task` class, Symfony already knows a bit about your fields. If you allow it, Symfony can "guess" the type of your field and set it up for you. In this example, Symfony can guess from the validation rules that both the `task` field is a normal `text`field and the `dueDate` field is a `date` field:

```
1  public function newAction()
2  {
3      $task = new Task();
4
5      $form = $this->createFormBuilder($task)
6          ->add('task')
7          ->add('dueDate', null, array('widget' => 'single_text'))
8          ->add('save', 'submit')
9          ->getForm();
10 }
```

The "guessing" is activated when you omit the second argument to the`add()` method (or if you pass `null` to it). If you pass an options array as the third argument (done for `dueDate` above), these options are applied to the guessed field.

> your form uses a specific validation group, the field type guesser will still consider all validation constraints when guessing your field types (including constraints that are not part of the validation group(s) being used).

# Field Type Options Guessing ¶

In addition to guessing the "type" for a field, Symfony can also try to guess the correct values of a number of field options.

> hen these options are set, the field will be rendered with special HTML attributes that provide for HTML5 client-side validation. However, it doesn't generate the equivalent server-side constraints (e.g. `Assert\Length`). And though you'll need to manually add your server-side validation, these field type options can then be guessed from that information.

`required`

The `required` option can be guessed based on the validation rules (i.e. is the

field `NotBlank` or `NotNull`) or the Doctrine metadata (i.e. is the field`nullable`). This is very useful, as your client-side validation will automatically match your validation rules.

`max_length`
If the field is some sort of text field, then the `max_length` option can be guessed from the validation constraints (if `Length` or `Range` is used) or from the Doctrine metadata (via the field's length).

ese field options are only guessed if you're using Symfony to guess the field type (i.e. omit or pass `null` as the second argument to `add()`).

If you'd like to change one of the guessed values, you can override it by passing the option in the options field array:

```
->add('task', null, array('attr' => array('maxlength' => 4)))
```

# Rendering a Form in a Template ¶

So far, you've seen how an entire form can be rendered with just one line of code. Of course, you'll usually need much more flexibility when rendering:

Twig PHP

```
{# app/Resources/views/default/new.html.twig #}
{{ form_start(form) }}
    {{ form_errors(form) }}

    {{ form_row(form.task) }}
    {{ form_row(form.dueDate) }}
{{ form_end(form) }}
```

You already know the `form_start()` and `form_end()` functions, but what do the other functions do?

`form_errors(form)`
Renders any errors global to the whole form (field-specific errors are displayed next to each field).

`form_row(form.dueDate)`
Renders the label, any errors, and the HTML form widget for the given field (e.g. `dueDate`) inside, by default, a `div` element.

The majority of the work is done by the `form_row` helper, which renders the label, errors and HTML form widget of each field inside a `div` tag by default. In the Form Theming section, you'll learn how the `form_row` output can be customized on many different levels.

ou can access the current data of your form via `form.vars.value`:

Twig PHP

```
1  {{ form.vars.value.task }}
```

## Rendering each Field by Hand ¶

The `form_row` helper is great because you can very quickly render each field of your form (and the markup used for the "row" can be customized as well). But since life isn't always so simple, you can also render each field entirely by hand. The end-product of the following is the same as when you used the `form_row` helper:

Twig PHP

```
1  {{ form_start(form) }}
2      {{ form_errors(form) }}
3
4      <div>
5          {{ form_label(form.task) }}
6          {{ form_errors(form.task) }}
7          {{ form_widget(form.task) }}
8      </div>
9
10     <div>
11         {{ form_label(form.dueDate) }}
12         {{ form_errors(form.dueDate) }}
13         {{ form_widget(form.dueDate) }}
14     </div>
15
16     <div>
17         {{ form_widget(form.save) }}
18     </div>
19
20 {{ form_end(form) }}
```

If the auto-generated label for a field isn't quite right, you can explicitly specify it:

Twig PHP

```
{{ form_label(form.task, 'Task Description') }}
```

Some field types have additional rendering options that can be passed to the widget. These options are documented with each type, but one common options is `attr`, which allows you to modify attributes on the form element. The following would add the `task_field` class to the rendered input text field:

Twig PHP

```
{{ form_widget(form.task, {'attr': {'class': 'task_field'}}) }}
```

If you need to render form fields "by hand" then you can access individual values for fields such as the `id`, `name` and `label`. For example to get the `id`:

Twig PHP

```
{{ form.task.vars.id }}
```

To get the value used for the form field's name attribute you need to use the `full_name` value:

Twig PHP

```
{{ form.task.vars.full_name }}
```

## Twig Template Function Reference ¶

If you're using Twig, a full reference of the form rendering functions is available in the reference manual. Read this to know everything about the helpers available and the options that can be used with each.

# Changing the Action and Method of a Form ¶

So far, the `form_start()` helper has been used to render the form's start tag and we assumed that each form is submitted to the same URL in a POST request. Sometimes you want to change these parameters. You can do so in a few different ways. If you build your form in the controller, you can use `setAction()` and `setMethod()`:

```
$form = $this->createFormBuilder($task)
    ->setAction($this->generateUrl('target_route'))
    ->setMethod('GET')
    ->add('task', 'text')
    ->add('dueDate', 'date')
    ->add('save', 'submit')
    ->getForm();
```

This example assumes that you've created a route called `target_route` that points to the controller that processes the form.

In Creating Form Classes you will learn how to move the form building code into separate classes. When using an external form class in the controller, you can pass the action and method as form options:

```
$form = $this->createForm(new TaskType(), $task, array(
    'action' => $this->generateUrl('target_route'),
    'method' => 'GET',
));
```

Finally, you can override the action and method in the template by passing them to the `form()` or the `form_start()` helper:

Twig PHP

```
{# app/Resources/views/default/new.html.twig #}
{{ form_start(form, {'action': path('target_route'), 'method': 'GET'}) }}
```

When the form's method is not GET or POST, but PUT, PATCH or DELETE, Symfony will insert a hidden field with the name `_method` that stores this method. The form will be submitted in a normal POST request, but Symfony's router is capable of detecting the `_method` parameter and will interpret it as a PUT, PATCH or DELETE request. Read the cookbook chapter "How to Use HTTP Methods beyond GET and POST in Routes" for more information.

# Creating Form Classes ¶

As you've seen, a form can be created and used directly in a controller. However, a better practice is to build the form in a separate, standalone PHP class, which can then be reused anywhere in your application. Create a new class that will house the logic for building the task form:

```
1  // src/AppBundle/Form/Type/TaskType.php
2  namespace AppBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6
7  class TaskType extends AbstractType
8  {
9      public function buildForm(FormBuilderInterface $builder, array $op
10     {
11         $builder
12             ->add('task')
13             ->add('dueDate', null, array('widget' => 'single_text'))
14             ->add('save', 'submit')
15         ;
16     }
17
18     public function getName()
19     {
20         return 'task';
21     }
22 }
```

The `getName()` method returns the identifier of this form "type". These identifiers must be unique in the application. Unless you want to override a built-in type, they should be different from the default Symfony types and from any type defined by a third-party bundle installed in your application. Consider prefixing your types with `app_` to avoid identifier collisions.

This new class contains all the directions needed to create the task form. It can be used to quickly build a form object in the controller:

```
1  // src/AppBundle/Controller/DefaultController.php
2
3  // add this new use statement at the top of the class
4  use AppBundle\Form\Type\TaskType;
5
6  public function newAction()
```

```
{
    $task = ...;
    $form = $this->createForm(new TaskType(), $task);

    // ...
}
```

Placing the form logic into its own class means that the form can be easily reused elsewhere in your project. This is the best way to create forms, but the choice is ultimately up to you.

### ɛtting the data_class

Every form needs to know the name of the class that holds the underlying data (e.g. `AppBundle\Entity\Task`). Usually, this is just guessed based off of the object passed to the second argument to`createForm` (i.e. `$task`). Later, when you begin embedding forms, this will no longer be sufficient. So, while not always necessary, it's generally a good idea to explicitly specify the `data_class` option by adding the following to your form type class:

```
use Symfony\Component\OptionsResolver\OptionsResolver;

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'AppBundle\Entity\Task',
    ));
}
```

ıen mapping forms to objects, all fields are mapped. Any fields on the form that do not exist on the mapped object will cause an exception to be thrown.

In cases where you need extra fields in the form (for example: a "do you agree with these terms" checkbox) that will not be mapped to the underlying object, you need to set the `mapped` option to`false`:

```
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array
{
    $builder
        ->add('task')
```

```
7           ->add('dueDate', null, array('mapped' => false))
8           ->add('save', 'submit')
9       ;
10  }
```

Additionally, if there are any fields on the form that aren't included in the submitted data, those fields will be explicitly set to `null`.

The field data can be accessed in a controller with:

```
1  $form->get('dueDate')->getData();
```

In addition, the data of an unmapped field can also be modified directly:

```
1  $form->get('dueDate')->setData(new \DateTime());
```

## Defining your Forms as Services ¶

Defining your form type as a service is a good practice and makes it really easy to use in your application.

rvices and the service container will be handled later on in this book. Things will be more clear after reading that chapter.

YAML XML PHP

```
1  # src/AppBundle/Resources/config/services.yml
2  services:
3      app.form.type.task:
4          class: AppBundle\Form\Type\TaskType
5          tags:
6              - { name: form.type, alias: task }
```

That's it! Now you can use your form type directly in a controller:

```php
// src/AppBundle/Controller/DefaultController.php
// ...

public function newAction()
{
    $task = ...;
    $form = $this->createForm('task', $task);

    // ...
}
```

or even use from within the form type of another form:

```php
// src/AppBundle/Form/Type/ListType.php
// ...

class ListType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $op
    {
        // ...

        $builder->add('someTask', 'task');
    }
}
```

Read Creating your Field Type as a Service for more information.

# Forms and Doctrine ¶

The goal of a form is to translate data from an object (e.g. `Task`) to an HTML form and then translate user-submitted data back to the original object. As such, the topic of persisting the `Task` object to the database is entirely unrelated to the topic of forms. But, if you've configured the `Task`class to be persisted via Doctrine (i.e. you've added mapping metadata for it), then persisting it after a form submission can be done when the form is valid:

```php
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($task);
    $em->flush();

    return $this->redirectToRoute('task_success');
}
```

If, for some reason, you don't have access to your original `$task` object, you can fetch it from the form:

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

For more information, see the Doctrine ORM chapter.

The key thing to understand is that when the form is submitted, the submitted data is transferred to the underlying object immediately. If you want to persist that data, you simply need to persist the object itself (which already contains the submitted data).

# Embedded Forms ¶

Often, you'll want to build a form that will include fields from many different objects. For example, a registration form may contain data belonging to a `User` object as well as many `Address` objects. Fortunately, this is easy and natural with the Form component.

## Embedding a Single Object ¶

Suppose that each `Task` belongs to a simple `Category` object. Start, of course, by creating the `Category` object:

```
// src/AppBundle/Entity/Category.php
namespace AppBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Category
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Next, add a new `category` property to the `Task` class:

```
// ...

class Task
{
```

```
    // ...

    /**
     * @Assert\Type(type="AppBundle\Entity\Category")
     * @Assert\Valid()
     */
    protected $category;

    // ...

    public function getCategory()
    {
        return $this->category;
    }

    public function setCategory(Category $category = null)
    {
        $this->category = $category;
    }
}
```

The `Valid` Constraint has been added to the property `category`. This cascades the validation to the corresponding entity. If you omit this constraint the child entity would not be validated.

Now that your application has been updated to reflect the new requirements, create a form class so that a `Category` object can be modified by the user:

```
// src/AppBundle/Form/Type/CategoryType.php
namespace AppBundle\Form\Type;

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

class CategoryType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $op
    {
        $builder->add('name');
    }

    public function configureOptions(OptionsResolver $resolver)
    {
```

```
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Category',
        ));
    }

    public function getName()
    {
        return 'category';
    }
}
```

The end goal is to allow the `Category` of a `Task` to be modified right inside the task form itself. To accomplish this, add a `category` field to the`TaskType` object whose type is an instance of the new `CategoryType` class:

```
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $option
{
    // ...

    $builder->add('category', new CategoryType());
}
```

The fields from `CategoryType` can now be rendered alongside those from the `TaskType` class.

Render the `Category` fields in the same way as the original `Task` fields:

Twig PHP

```
{# ... #}

<h3>Category</h3>
<div class="category">
    {{ form_row(form.category.name) }}
</div>

{# ... #}
```

When the user submits the form, the submitted data for the `Category`fields are used to construct an instance of `Category`, which is then set on the `category` field of the `Task` instance.

The `Category` instance is accessible naturally via `$task->getCategory()` and can be persisted to the

database or used however you need.

## Embedding a Collection of Forms ¶

You can also embed a collection of forms into one form (imagine a`Category` form with many `Product` sub-forms). This is done by using the`collection` field type.

For more information see the "How to Embed a Collection of Forms" cookbook entry and the collection field type reference.

# Form Theming ¶

Every part of how a form is rendered can be customized. You're free to change how each form "row" renders, change the markup used to render errors, or even customize how a `textarea` tag should be rendered. Nothing is off-limits, and different customizations can be used in different places.

Symfony uses templates to render each and every part of a form, such as`label` tags, `input` tags, error messages and everything else.

In Twig, each form "fragment" is represented by a Twig block. To customize any part of how a form renders, you just need to override the appropriate block.

In PHP, each form "fragment" is rendered via an individual template file. To customize any part of how a form renders, you just need to override the existing template by creating a new one.

To understand how this works, customize the `form_row` fragment and add a class attribute to the `div` element that surrounds each row. To do this, create a new template file that will store the new markup:

Twig PHP

```
1  {# app/Resources/views/form/fields.html.twig #}
2  {% block form_row %}
3  {% spaceless %}
4      <div class="form_row">
5          {{ form_label(form) }}
6          {{ form_errors(form) }}
7          {{ form_widget(form) }}
8      </div>
9  {% endspaceless %}
10 {% endblock form_row %}
```

The `form_row` form fragment is used when rendering most fields via the`form_row` function. To tell the Form component to use your new `form_row`fragment defined above, add the following to the top of the template that renders the form:

Twig PHP

```twig
{# app/Resources/views/default/new.html.twig #}
{% form_theme form 'form/fields.html.twig' %}

{# or if you want to use multiple themes #}
{% form_theme form 'form/fields.html.twig' 'form/fields2.html.twig' %}

{# ... render the form #}
```

The `form_theme` tag (in Twig) "imports" the fragments defined in the given template and uses them when rendering the form. In other words, when the `form_row` function is called later in this template, it will use the`form_row` block from your custom theme (instead of the default `form_row`block that ships with Symfony).

Your custom theme does not have to override all the blocks. When rendering a block which is not overridden in your custom theme, the theming engine will fall back to the global theme (defined at the bundle level).

If several custom themes are provided they will be searched in the listed order before falling back to the global theme.

To customize any portion of a form, you just need to override the appropriate fragment. Knowing exactly which block or file to override is the subject of the next section.

For a more extensive discussion, see How to Customize Form Rendering.

## Form Fragment Naming ¶

In Symfony, every part of a form that is rendered – HTML form elements, errors, labels, etc. – is defined in a base theme, which is a collection of blocks in Twig and a collection of template files in PHP.

In Twig, every block needed is defined in a single template file (e.g.form_div_layout.html.twig) that lives inside the Twig Bridge. Inside this file, you can see every block needed to render a form and every default field type.

In PHP, the fragments are individual template files. By default they are located in the `Resources/views/Form` directory of the FrameworkBundle (view on GitHub).

Each fragment name follows the same basic pattern and is broken up into two pieces, separated by a single underscore character (_). A few examples are:

- `form_row` - used by `form_row` to render most fields;
- `textarea_widget` - used by `form_widget` to render a `textarea` field type;
- `form_errors` - used by `form_errors` to render errors for a field;

Each fragment follows the same basic pattern: `type_part`. The `type` portion corresponds to the field type being rendered (e.g. `textarea`, `checkbox`,`date`, etc) whereas the `part` portion corresponds to what is being rendered (e.g. `label`, `widget`, `errors`, etc). By default, there are 4 possible parts of a form that can be rendered:

| | | |
|---|---|---|
| `label` | (e.g. `form_label`) | renders the field's label |
| `widget` | (e.g. `form_widget`) | renders the field's HTML representation |
| `errors` | (e.g. `form_errors`) | renders the field's errors |
| `row` | (e.g. `form_row`) | renders the field's entire row (label, widget & errors) |

ere are actually 2 other parts - `rows` and `rest` - but you should rarely if ever need to worry about overriding them.

By knowing the field type (e.g. `textarea`) and which part you want to customize (e.g. `widget`), you can construct the fragment name that needs to be overridden (e.g. `textarea_widget`).

## Template Fragment Inheritance ¶

In some cases, the fragment you want to customize will appear to be missing. For example, there is no `textarea_errors` fragment in the default themes provided with Symfony. So how are the errors for a textarea field rendered?

The answer is: via the `form_errors` fragment. When Symfony renders the errors for a textarea type, it looks first for a `textarea_errors` fragment before falling back to the `form_errors` fragment. Each field type has aparent type (the parent type of `textarea` is `text`, its parent is `form`), and Symfony uses the fragment for the parent type if the base fragment doesn't exist.

So, to override the errors for only `textarea` fields, copy the `form_errors`fragment, rename it to `textarea_errors` and customize it. To override the default error rendering for all fields, copy and customize the `form_errors`fragment directly.

> e "parent" type of each field type is available in the form type reference for each field type.

## Global Form Theming ¶

In the above example, you used the `form_theme` helper (in Twig) to "import" the custom form fragments into just that form. You can also tell Symfony to import form customizations across your entire project.

### Twig ¶

To automatically include the customized blocks from the `fields.html.twig`template created earlier in all templates, modify your application configuration file:

YAML XML PHP

```yaml
# app/config/config.yml
twig:
    form_themes:
        - 'form/fields.html.twig'
    # ...
```

Any blocks inside the `fields.html.twig` template are now used globally to define form output.

> *ustomizing Form Output all in a Single File with Twig*
>
> In Twig, you can also customize a form block right inside the template where that customization is needed:

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

{# import "_self" as the form theme #}
{% form_theme form _self %}

{# make the form fragment customization #}
{% block form_row %}
```

```
    {# custom field row output #}
{% endblock form_row %}

{% block content %}
    {# ... #}

    {{ form_row(form.task) }}
{% endblock %}
```

The `{% form_theme form _self %}` tag allows form blocks to be customized directly inside the template that will use those customizations. Use this method to quickly make form output customizations that will only ever be needed in a single template.

> This `{% form_theme form _self %}` functionality will only work if your template extends another. If your template does not, you must point `form_theme` to a separate template.

### PHP ¶

To automatically include the customized templates from the `app/Resources/views/Form` directory created earlier in all templates, modify your application configuration file:

YAML XML PHP

```
# app/config/config.yml
framework:
    templating:
        form:
            resources:
                - 'Form'
# ...
```

Any fragments inside the `app/Resources/views/Form` directory are now used globally to define form output.

## CSRF Protection ¶

CSRF - or Cross-site request forgery - is a method by which a malicious user attempts to make your legitimate users unknowingly submit data that they don't intend to submit. Fortunately, CSRF

attacks can be prevented by using a CSRF token inside your forms.

The good news is that, by default, Symfony embeds and validates CSRF tokens automatically for you. This means that you can take advantage of the CSRF protection without doing anything. In fact, every form in this chapter has taken advantage of the CSRF protection!

CSRF protection works by adding a hidden field to your form - called `_token` by default - that contains a value that only you and your user knows. This ensures that the user - not some other entity - is submitting the given data. Symfony automatically validates the presence and accuracy of this token.

The `_token` field is a hidden field and will be automatically rendered if you include the `form_end()` function in your template, which ensures that all un-rendered fields are output.

ıce the token is stored in the session, a session is started automatically as soon as you render a form with CSRF protection.

The CSRF token can be customized on a form-by-form basis. For example:

```
use Symfony\Component\OptionsResolver\OptionsResolver;

class TaskType extends AbstractType
{
    // ...

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class'      => 'AppBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // a unique key to help generate the secret token
            'intention'       => 'task_item',
        ));
    }

    // ...
}
```

To disable CSRF protection, set the `csrf_protection` option to false. Customizations can also be made globally in your project. For more information, see the form configuration reference section.

e `intention` option is optional but greatly enhances the security of the generated token by making it different for each form.

RF tokens are meant to be different for every user. This is why you need to be cautious if you try to cache pages with forms including this kind of protection. For more information, seeCaching Pages that Contain CSRF Protected Forms.

## Using a Form without a Class ¶

In most cases, a form is tied to an object, and the fields of the form get and store their data on the properties of that object. This is exactly what you've seen so far in this chapter with the Task class.

But sometimes, you may just want to use a form without a class, and get back an array of the submitted data. This is actually really easy:

```
1  // make sure you've imported the Request namespace above the class
2  use Symfony\Component\HttpFoundation\Request;
3  // ...
4
5  public function contactAction(Request $request)
6  {
7      $defaultData = array('message' => 'Type your message here');
8      $form = $this->createFormBuilder($defaultData)
9          ->add('name', 'text')
10         ->add('email', 'email')
11         ->add('message', 'textarea')
12         ->add('send', 'submit')
13         ->getForm();
14
15     $form->handleRequest($request);
16
17     if ($form->isValid()) {
18         // data is an array with "name", "email", and "message" keys
19         $data = $form->getData();
20     }
21
22     // ... render the form
23 }
```

By default, a form actually assumes that you want to work with arrays of data, instead of an object. There are exactly two ways that you can change this behavior and tie the form to an object instead:

1. Pass an object when creating the form (as the first argument to `createFormBuilder` or the second argument to `createForm`);
2. Declare the `data_class` option on your form.

If you don't do either of these, then the form will return the data as an array. In this example, since `$defaultData` is not an object (and no `data_class` option is set), `$form->getData()` ultimately returns an array.

> You can also access POST values (in this case "name") directly through the request object, like so:
>
> ```
> $request->request->get('name');
> ```
>
> Be advised, however, that in most cases using the `getData()` method is a better choice, since it returns the data (usually an object) after it's been transformed by the Form component.

## Adding Validation ¶

The only missing piece is validation. Usually, when you call `$form->isValid()`, the object is validated by reading the constraints that you applied to that class. If your form is mapped to an object (i.e. you're using the `data_class` option or passing an object to your form), this is almost always the approach you want to use. See Validation for more details.

But if the form is not mapped to an object and you instead want to retrieve a simple array of your submitted data, how can you add constraints to the data of your form?

The answer is to setup the constraints yourself, and attach them to the individual fields. The overall approach is covered a bit more in the validation chapter, but here's a short example:

```
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;

$builder
    ->add('firstName', 'text', array(
        'constraints' => new Length(array('min' => 3)),
    ))
    ->add('lastName', 'text', array(
```

```
        'constraints' => array(
            new NotBlank(),
            new Length(array('min' => 3)),
        ),
    ))
;
```

you are using validation groups, you need to either reference the `Default` group when creating the form, or set the correct group on the constraint you are adding.

```
new NotBlank(array('groups' => array('create', 'update'))
```

# Final Thoughts ¶

You now know all of the building blocks necessary to build complex and functional forms for your application. When building forms, keep in mind that the first goal of a form is to translate data from an object (`Task`) to an HTML form so that the user can modify that data. The second goal of a form is to take the data submitted by the user and to re-apply it to the object.

There's still much more to learn about the powerful world of forms, such as how to handle file uploads with Doctrine or how to create a form where a dynamic number of sub-forms can be added (e.g. a todo list where you can keep adding more fields via JavaScript before submitting). See the cookbook for these topics. Also, be sure to lean on the field type reference documentation, which includes examples of how to use each field type and its options.