# Symfony Framework Form Recipes


Joshua Thijssen

# Symfony Framework Recipes - Forms

## Practical Symfony2 form recipes for real-life use-cases.

Joshua Thijssen

This book is for sale at http://leanpub.com/symfonyframeworkrecipes-forms

This version was published on 2015-11-26

# Also By Joshua Thijssen

Symfony Framework Deepdive - Security

Symfony Framework Deepdive - Console

# Contents

# Introduction

# Sample book

This is a sample of the Symfony Rainbow Form Recipes book. It will contain all the recipe descriptions, but not the explanation or the code (except for one recipe). You can always purchase the complete book through Leanpub and get a 45 day money-back guarantee. But hopefully, this sample book will be enough to convince you :)

# About this book

This book contains a set of recipes that can be used to solve real life use-cases. Many of these recipes are collected through years of experience with using Symfony forms in practice and others have been requested by others.

Each recipe starts with an introduction for the given use-case. What problem does it solve and how do we solve it. We will on occasion dive a bit deeper into the Symfony form component in order to understand what is going on, but in general, the recipes can be used by even the most novice Symfony2 developer.

## About the Symfony Rainbow series

The Symfony Rainbow series is a collection of books based around the different components that make up the Symfony framework. Every book will concentrate on a single component or a set of smaller components. The books will describe their inner workings in detail and give you recipes to solve common problems quickly and efficiently. Many of the recipes found in this book, will also available in the Symfony Rainbow edition on Forms.



[1] https://pixabay.com/en/business-office-professional-papers-988179/

[2] http://creativecommons.org/publicdomain/zero/1.0/deed.en

# A quick introduction on forms

Forms are playing an essential role in many web applications. They act as the bridge between the user and the application or website. But forms are playing a much bigger role than just serving the interaction between program and users. Even API systems where applications and servers communicate with each other automatically, forms play a key role when it come to accepting and validating the incoming data. For instance, many RESTful and HTTP APIs are using forms under the hood to exchange and validate data in an automated fashion.

Because data comes in many shapes and sizes, forms must do too. Sometimes they can be very basic in setup, like a simple login form where we just need a username and password field. And sometimes they are very complex, like a multi-page form wizard, where pages and fields depend on selections in previous fields or pages.

Creating a form component that can deal with all these different scenarios, means that it must be very generic in setup: it cannot assume too much, as we simply do not know what and how a developer wants to setup their forms. The downside however, is that such a generic form component is also likely to be more complex in usage. In general the following seems true: the more an application can do, the more complex it is to use it. And this is even true in the case of the Symfony2 Form component: it contains a lot of power, but it means that it isn't always easy to make it do what you want.

I personally do not think this is solely to blame on the component itself. In fact, I think in general the Form component uses the correct architecture for its purpose. Beginners can setup forms with relative easy, but before setting up a more complex forms, you'll need to understand the basic mechanics of the Form component itself. Once you understand these basics, - and that doesn't need to take a lot of time - writing even the most complex forms are just as easy as writing the simple ones, they will only take a bit more time to write.

The Symfony Rainbow Deepdive Form Book[3] explains in great detail the Form component internals and usage.

Note that this recipe book isn't about explaining how the Form component works. The Symfony Rainbow Deepdive Form Book[4] already takes care of explaining in great detail the Form component's usage, its internal workings and even the surrounding components like the `OptionResolver` and `propertyAccess` components. This recipe book however, is about solving very specific problems when dealing with the Form component. It's meant for Symfony2 beginners, experts and everyone in between. You can simply copy-paste the code in your project, you can read why the recipes work

[3] https://leanpub.com/symfonyframeworkdeepdive-forms

[4] https://leanpub.com/symfonyframeworkdeepdive-forms

the way they work, or you can ever dive deeper into the recipes by trying to enhance them by following some of the exercises I've added here and there. Hopefully, the recipes gives you an idea on how to solve your next form problem, or maybe it gives you some reflection in how it solves a problem you already solved in your own projects (and maybe in a different manner).

Not all recipes are about solving a problem per se. Some of the recipes are about common misconceptions surrounding the Form component (for instance, how and when to use data transformers, how form themes work etc). They provide a bit more background to make you understand the basic concepts, so you can actually use it in the correct ways.

Keep in mind one thing: no matter how complex a form is, or how simple a form seems, they all work exactly the same way. How forms are generated, how forms are handled, how data is transported from a HTTP request back onto your model or entities etc. If you understand this, writing forms will not be a hassle or burden, but will actually start becoming fun again!

# Forms in Symfony 2.3

Since the Form component's initial version in Symfony v2.0, it has seen many changes, including (many) BC breaks. Form code that was usable in Symfony v2.1 isn't guaranteed to work in v2.3 or newer. From Symfony v2.3, however, the Form component matured, and settled down a bit in functionality. Also, the BC promise that the Symfony framework added, which is a strict coding policy to make sure no backward compatibility breaks will occur, makes sure that forms written for v2.3 will be usable in any of the next v2.x versions. It will not be until version 3.0 of Symfony before these forms might not function. This allows us to use code written for lower Symfony 2.x versions, but as a downside you might not be able to take advantage of new functionality that might have been introduced in later versions of Symfony.

Since Symfony v2.3 is a Long Term Support (LTS) version, it will be supported and updated until to May 2016. The writing of this book took place in October/November 2015, so we assume that most users will be or are in the process of upgrading to at least v2.7 (and maybe even to v3.0!). Still, we tried to keep the recipes compatible with v2.3 where possible.

When writing a form type which you like to distribute, try and make it compatible for v2.3 as much if possible. This allows both v2.3 and v2.7 users to be able to use your form type. Otherwise, only v2.7 users might do so. Don't forget: there are still many users running Symfony v2.3.

# Differences between Symfony 2.3 and 2.7

Since the BC promise of Symfony from v2.3, backward compatibility is guaranteed in the Symfony2 components. But even then, some smaller BC breaks are present between v2.3 and v2.7 of the Form component.

- The `FormInterface::getTransformationFailure()` was added in v2.5.
- Two optional parameters to FormInterface::getErrors() are added and changed the method to return a `Symfony\Component\Form\FormErrorIterator` instance instead of an array, in v2.5.

There was another BC break in v2.6.0 concerning timezones and the date form types, but these changes have been reverted back in v2.6.2.

Upgrading your forms from Symfony v2.3 to v2.7 should be relatively easy, especially when you already upgraded to the non-LTS releases v2.4, v2.5, v2.6 in between.

In v2.7, Symfony implemented warning notifications on deprecated function calls. This is to let developers know which deprecated functionality they are using that will be removed in Symfony v3.0. It's quite possible that - especially in older form code - deprecated functionality is still being used. This is not a problem however, but the deprecation notifications gives you a good idea on what needs to be changed in order to get the forms compatible for Symfony v3.0.

Symfony v2.7 also provides some new features, for instance, a complete new choice system for creating drop down lists. But the downside is that using these features, your code would not be compatibly anymore with Symfony v2.3. If you need them, use them, as it especially makes the upgrade path to Symfony v3.0 easier. However, if you create (open source) form types for instance, you might want to stay compatible with v2.3 and not include these features. It's up to you to decide what is important.

## The big options resolver change

Another major change that took place in Symfony v2.6, is the deprecation of some features of the option resolver, namely the `OptionsResolverInterface` and the `setDefaultOptions()` method. When using the option resolver component, something that each form type by default does, a form type uses the method `setDefaultOptions()`, which will be called in order to easily override your form type options. Since Symfony v2.6, this method has been changed to `configureOptions()`. It still does the same thing, but now it accepts the `OptionsResolver` class instead of the `OptionsResolver-Interface`. In order to stay backward compatibly, the `AbstractType` class found in the Symfony v2.6 Form component will actually use both the `setDefaultOptions()` and `configureOptions()`. To stay compatible with lower versions, `setDefaultOptions()` must be used though in your own form type.

In the recipes I will be using mostly the `setDefaultOptions()` method to keep compatiblity with v2.3. However, sometimes I will be talking about, or using the `configureOptions()` method instead. Both functions will do the same thing, and in the newer versions of Symfony, the `setDefaultOptions()` is nothing more than a redirect to `configureOptions()`.

# Differences between Symfony 2.7 and 3.0

Even though Symfony 3.0 seems like "the next big thing", it does not provide a lot of new features or rewrites. In fact, moving from Symfony 2.7 towards Symfony 3.0 is quite easy to do so, and automated compatibility checkers even exists to identify and convert deprecated function calls.

One of the biggest differences between v2.7 and v3.0 however, is the removal of all the deprecated functionality that needed to be supported in the v2.x releases. From v2.7 onwards, all deprecated code will trigger deprecation messages so it will be easy enough to spot any code that still uses deprecated code. This makes the upgrade path even easier.

Another big change in the Form component itself, is the fact that form types are not referenced anymore by their form type name (like `text`, `email`, `textarea` etc), but by fully qualified class name (FQCN). This seems like a major change in how to create forms, but actually PHP v5.5 makes it easy for us with the help of the `::class` keyword. Because Symfony v3.0 needs at least PHP v5.5.9, this is not a problem. This change has been added in Symfony v2.8 in a backward compatible way, so in version v2.8 you can use either the textual name, or the FQCN. From Symfony 3.0 onwards, only the FQCN is supported.

```
// Usable in every Symfony 2.x version
$form = $this->createFormBuilder()
    ->add('name', 'text')
    ->add('age', 'integer')
    ->getForm();
```

```
// Usable in Symfony 2.8 and Symfony 3.x.
use Symfony\Component\Form\Extension\Core\Type\IntegerType;
use Symfony\Component\Form\Extension\Core\Type\TextType;

$form = $this->createFormBuilder()
    ->add('name', TextType::class)
    ->add('age', IntegerType::class)
    ->getForm();
```

This change should make user experience better, even though it might take a bit of time to get used to this new syntax.

# Recipes

Do you have a certain recipe you would like to see in this book? Send them to me at `info@symfony-rainbow.com`.

Even though I've created the recipes with great care, and checked them with multiple Symfony2 versions (v2.3, v2.7, and v3.0 where possible), and through multiple browsers. Use them at your own risk, but honestly, try and understand them first before you use them. Some recipes may not work in your specific use-cases, or maybe they simply contain bugs. I'll try and update each recipe according to feedback from readers.

The recipes are also not necessarily the one-and-only solution. Just like with anything in programming, there are more ways than one to do things, and they are not always simply good or bad solutions. Things has to be taken into context and what might be a good solution for one problem, might actually be a bad solution in another. Understanding is key to a successful implementation.

> The code will be available on GitHub: https://github.com/SymfonyRainbow/form_recipes

# Recipe 1: Creating an advanced custom form type

The difficulty level for this recipe is **hard**

Many developers don't realize that when creating forms within Symfony2, we aren't really creating a form, but rather we create form types which in turn will create the forms for us. This is why developers often get confused when dealing with dynamic forms: we simply cannot add the dynamics in the form type itself, as these are not really our forms, and as such, form types have no idea on what data will be used after the forms are actually created. Instead, we hook the dynamic part into the actual form instances through form events like the `PRE_SET_DATA` and `PRE_SUBMIT` events. These events CAN be added in the form type, but are only executed when the form type has created the actual forms.

Form types can been seen as the "DNA" of a form. They are not forms by themselves, but contains all the needed information to instantiate them.

This recipe shows you how to create a more complicated form type that can be used for your own forms. This form type allows you to add a value in either bytes up to gigabytes in different ways depending on the configuration. Ultimately, the actual values that are stored and read inside the form data are floats representing the size in bytes (so when a user enters `1kb`, the actual value will become 1024).

# Recipe 2: Creating a simple CAPTCHA form type

The difficulty level for this recipe is **easy**

It happens often that automated scripts or bots stroll the internet and automatically fill in forms they encounter. Most likely to send unsolicited emails (forms submissions are still often emailed), or to detect weaknesses in applications. This is why many public forms on the internet are somehow protected through CAPTCHAs. These "puzzles" are considered easy solvable for humans, but hard for machines. A good example is a picture with a number or word, which you have to type over.

This recipe creates a simple CAPTCHA form type, which you can use for protecting your own forms.

Not that this is just a very simple example, which can easily be solved by machines. It will probably not protect you against all spam robots.

# Recipe 3: Ship to billing address

The difficulty level for this recipe is **easy**

A common use-case with shipping forms is to have a checkbox that can be checked if your shipping address is the same as your billing address. This saves the user a lot of work because now it doesn't have to type the same address twice.

This recipe shows you how to create a simple form that handles this, while the user is still able to select a different shipping address if needed.

# Recipe 4: Using data value objects in forms

The difficulty level for this recipe is **medium**

Value objects are commonly used when working in a Domain Driven environment. The form component will by default populate entities through getters and setters, but with value objects there often are no setters and objects can only created through its constructor.

This recipe shows how you can simply make the form component use value objects with the help of data mappers.

# Recipe 5: Filtering and transforming data

The difficulty level for this recipe is **medium**

This recipe allows you to create a `textarea` where you can easily edit JSON data. This data will be converted from and to standard arrays in your model trough the help of data transformers.

This recipe is fairly easy, but the recipe focusses on explaining how data formats are used within the Form component, when you should use what kind of data transformer, and how you should sanitize and filter data.

# Recipe 6: Easily create HTML5 input elements

The difficulty level for this recipe is **easy**

The HTML5 specification adds additional types for the HTML `<input>` tag that can be used to enhance the user experience. For instance, the `"email"` type can be used whenever a user needs to input an email address. Browsers can validate if the user actually entered an email address before submitting data (this is called client-side validation), and mobile browsers can even display a different type of keyboard map (one dedicated to entering email addresses).

Symfony2 has a few HTML5 capable form types, like `email`, `url`, `date` etc. However, there are more HTML5 input types available that are not supported by Symfony2 directly.

This recipe creates a form extension that is added onto the `"text"` form type, so it will allow to add the HTML5 type you want the text element to be.

## The recipe

This recipe can easily be created by implementing a new form type, but we use a different approach here. Instead, we actually extend the `text` form type by creating a new `FormTypeExtension`. This saves us much work in maintaining a new form type, and writing Twig and PHP templates for it.

Form type extensions are a great way to extend, or even overwrite existing form type functionality.

The extension itself is fairly trivial: it first defines inside the `setDefaultOptions()` method a new option called `html5_type`, which by default is set to nothing (`""`).

In the `buildView()` method, we check if the user has actually set this `html5_type` option, and if so, we set the template variable named `type` to that option. Note that it is important to use the variable name `type` inside the variables and not `html5_type`. This is because a text widget will be rendered by the default `form_widget_simple` block, as there is no more specific `text_widget` block in the default templates. That block renders the actual `<input>` tag and uses the `type` template variable as the actual input type. If no type is specified, it defaults to `text`.

This means, that if we DO specify the type, that actual type will be used to render the input tag. So if we define our `html5_type` to be `range`, then the template variable `type` will become `range`, and the actual rendered input element will be `<input type='range' ....>`.

If you take a look at the `email_widget` and `url_widget` (and other) twig templates in the `form_div_layout.html.twig` layout file from Symfony2, you'll notice that they only set the specific type to respectively `email` and `url` before calling the `form_widget_simple` block.

Once the extension has been created, all we need to do is register this form type extension in our form factory. This is easy as Symfony automatically does this as soon as you register your form type extension as a service and tag it with the tag `form.type_extension`.

Make sure that when registering your form type extensions, you must alias the form type extension with the NAME of the form type you want to EXTEND and not the name of your actual form type extension. This is easily overlooked!

### Specifying HTML5 attributes

It's possible to specify custom attributes if they are needed. For instance, the `range` type allows you to define a `min`, `max` and `step` attribute to control the range. These options can easily be added with the default `attr` option.

Note that HTML5 specific input types are rendered differently per browser. Not all browsers support all HTML5 input elements (for instance, the `search` type is not defined in most browsers). However,

they will automatically fall back to a regular input text type so you don't have to worry about unsupported types.

Change the `setDefaultOptions()` so only valid html5 types are allowed in the `html5_type` option.

# Code

**src/Rainbow/FormBundle/Form/Extension/HtmlTextTypeExtension.php**

```php
<?php

namespace Rainbow\FormBundle\Form\Extension;

use Symfony\Component\Form\AbstractTypeExtension;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\Form\FormView;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class HtmlTextTypeExtension extends AbstractTypeExtension
{
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        if (!empty($options['html5_type'])) {
            // Using 'type' is important, as this is the variable used by the form view blocks
            $view->vars['type'] = $options['html5_type'];
        }
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'html5_type' => '',
        ));
    }

    public function getExtendedType()
    {
        return 'text';
    }
}
```

**src/Rainbow/FormBundle/Resources/config/services.yml**

```yaml
services:
    rainbow_form_type_extension:
        class: Rainbow\FormBundle\Form\Extension\HtmlTextTypeExtension
        tags:
            - { name: form.type_extension, alias: text }
```

**src/Rainbow/FormBundle/Controller/Recipe6Controller.php**

```php
<?php

namespace Rainbow\FormBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class Recipe6Controller extends Controller
{
    public function indexAction(Request $request)
    {
        $form = $this->createFormBuilder()
            ->add('favorite_color', 'text', array(
                'html5_type' => 'color',
            ))
            ->add('delivery_dt', 'text', array(
                'html5_type' => 'datetime',
            ))
            ->add('quantity', 'text', array(
                'html5_type' => 'range',
                'attr' => array('min' => 10, 'max' => 90, 'step' => 4),
            ))
            ->add('phone', 'text', array(
                'html5_type' => 'tel',
            ))
            ->add('weeknr', 'text', array(
                'html5_type' => 'week',
            ))
            ->getForm();

        $form->add('submit', 'submit');

        $form->handleRequest($request);
        if ($form->isValid()) {
            var_dump($form->getData());
        }

        return $this->render('RainbowFormBundle:Recipe6:index.html.twig', array(
            'recipe6form' => $form->createView(),
        ));
    }
}
```

# Recipe 7: Using multiple forms on the same page

The difficulty level for this recipe is **easy**

It's possible that you want to use multiple forms on the same page. This is normally no problem, but it can become so, if you want to use two of SIMILAR forms.

```
$form1 = $this->createForm(new ContactType());
$form2 = $this->createForm(new ContactType());
```

Another problem will arise when you create simple forms on the fly through the `getFormBuilder()` method directly from a controller.

```
$form1 = $this->createformBuilder()
        ->add('country', 'country')
        ->getForm();
$form2 = $this->createformBuilder()
        ->add('contact', 'text')
        ->getForm();
```

The main problem with these forms, is that the forms share the same "name". This is an important aspect, as the form name is used to identify forms from others when submitting data, but also when setting form themes, and even the web debug toolbar will get confused (it will only display information for the first form in such cases).

This recipe tells you how you can easily create multiple forms of the same type, or create forms through the form builder directly without name collisions.

# Recipe 8: Extending templates

The difficulty level for this recipe is **easy**

Often you like to use a form theme in which you want to change just a small part, for instance, the `form_label` block. However, you want to extend this not from the default form theme, but from another custom form theme, which could be for instance the `form_table_layout.html.twig`.

# Recipe 9: Adding a captcha after multiple submission failures

The difficulty level for this recipe is **medium**

Suppose you have a custom login form. Within this form, you like to make sure that nobody is "brute-forcing" their way in, by randomly testing usernames and passwords. For this, you can use a captcha to make it harder to automatically submit the form, but this will also annoy valid users, as everybody needs to enter a captcha whenever they want to log in.

This recipe will create a form extension that will only add a captcha field when it has been submitted and validated incorrectly after a number of attempts. This makes it more friendly to users who might have mistyped their passwords and users who log in for the first time, while still be able to mitigate brute-force attempts.

# Recipe 10: Creating a simple form wizard

The difficulty level for this recipe is **hard**

Sometimes you would like to display your form not as a long list of elements, but rather view the form on different pages. Based on information you submit on page 1, you might even get different questions on page 2 etc. This recipe provides a simple form wizard system to give you some idea on how to incorporate such a system.

Even though you can always expand your own form wizard from this recipe, there is a very good form wizard bundle available at https://github.com/craue/CraueFormFlowBundle[5]

[5] https://github.com/craue/CraueFormFlowBundle

# Recipe 11: Dynamic data mapper for value objects

The difficulty level for this recipe is **hard**

The previous recipe on creating a data mappers for value objects had one big drawback: it could only be used when we had a one-on-one relation between the form fields and the value object properties: they had to be present and the form fields must have been added in the correct order. Changing the order of the form element would also change the order in which the parameters to the value object's constructor are passed.

This recipe is a more advanced version of the value object data mapper, where the order in which you create your form elements does not matter, but where the actual order of constructor arguments is found by inspecting the constructor through reflection. This way, you can safely change the order (which can be important when adding form elements dynamically), and it even allows you to skip certain arguments in your value objects, provided you add default values to the arguments in the constructor (although this might not be likely when dealing with value objects).

# Recipe 12: Dynamic data validation

The difficulty level for this recipe is **easy**

It's not uncommon that your validation is more dynamic than a simple "this field must not be blank", or "this field must have at least 10 characters". Sometimes, validation is based on other information, like external information: only a user with certain rights can check this box, or sometimes it depends on other form fields: if this field is set, than another field must not be set etc.

This recipe will define form with a start and end date. It will add a custom validation constraint to make sure that the end date is always higher than the start date.