Getting Crazy with Form Themes

With <3 from SymfonyCasts

Chapter 1: Form Parts & Functions Reference

Yo peeps! It's time to jump into a topic that's actually, super fun! Yep, we're going to learn to *bend* Symfony forms to our will: controlling *exactly* how they render... and believe me, by the end of this course, you'll be able to render a field in whatever *weird* way you want to.

Grab the Course Code!

To make forms great again, let's code together! Download the code from this page and unzip it. Inside, you'll find a trusty start/ directory, which will hold the exact code that I already have here.

To get the project running, open the README.md file and follow all the amazing details there. The last step will be open a terminal, move into the project directory - mine is called aqua_note - and then start the built-in PHP web server with:

```
$ bin/console server:run
```

Find a browser and pull up the address - http://localhost:8000 - to find our awesome project: Aquanote!

For this tutorial, there's just *one* thing you need to know: our database has a *genus* table, which is a type of animal classification. That table holds a bunch of different types of sea animals. To manage this, we have an admin section: login with weaverryan+1@gmail.com and password iliketurtles.

The admin section lives at /admin/genus. Click edit and... here's our starting form!

Form Type Basics

And so far, our form has all the parts you'd expect: a form class: GenusFormType:

50 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 13
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('subFamily', EntityType::class, [
                'placeholder' => 'Choose a Sub Family',
                'class' => SubFamily::class,
                'query_builder' => function(SubFamilyRepository $repo) {
                    return $repo->createAlphabeticalQueryBuilder();
                }
            ])
            ->add('speciesCount')
            ->add('funFact')
            ->add('isPublished', ChoiceType::class, [
                'choices' => [
                    'Yes' => true,
                    'No' => false,
                ]
            ])
            ->add('firstDiscoveredAt', DateType::class, [
                'widget' => 'single_text',
                'attr' => ['class' => 'js-datepicker'],
                'html5' => false,
            ])
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => 'AppBundle\Entity\Genus'
        ]);
    }
}
```

And a controller that builds the form and passes it into the template:

86 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
... lines 1 - 15
class GenusAdminController extends Controller
{
... lines 18 - 34
    public function newAction(Request $request)
    {
        $form = $this->createForm(GenusFormType::class);

        // only handles data on POST
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
            $genus = $form->getData();

            $em = $this->getDoctrine()->getManager();
            $em->persist($genus);
            $em->flush();

            $this->addFlash(
                'success',
                sprintf('Genus created by you: %s!', $this->getUser()->getEmail())
            );

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

        return $this->render('admin/genus/new.html.twig', [
            'genusForm' => $form->createView()
        ]);
    }
... lines 60 - 85
}
```

This gives us a genusForm variable inside of new.html.twig:

14 lines | app/Resources/views/admin/genus/new.html.twig

```
{% extends 'admin/genus/formLayout.html.twig' %}

{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-xs-12">
                <h1>New Genus</h1>

                {{ include('admin/genus/_form.html.twig') }}
            </div>
        </div>
    </div>
{% endblock %}
```

But the real work is done via an included template: _form.html.twig:

13 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
    {{ form_row(genusForm.name) }}
    {{ form_row(genusForm.subFamily) }}
    {{ form_row(genusForm.speciesCount, {
        'label': 'Number of Species'
    }) }}
    {{ form_row(genusForm.funFact) }}
    {{ form_row(genusForm.isPublished) }}
    {{ form_row(genusForm.firstDiscoveredAt) }}

    <button type="submit" class="btn btn-primary" formnovalidate>Save</button>
{{ form_end(genusForm) }}
```

Let's start there.

The Form Rendering Functions

The genusForm variable is an *object*, but you can't just print it. Instead, Symfony gives us a bunch of form *functions*: each renders a different part of the form.

To get all the deets, head to Symfony.com. Click into the Documentation and then find the Reference section. This holds a *wonderful* page called Twig Template Function and Variable Reference. This lists all the functions we'll be using *and* their arguments. Let's dive into these... and then, extend the heck out of them.

Chapter 2: Form Rendering Functions

I want to render this form, but be as *lazy* as humanly possible.

The Lazy Way: form()

Copy the existing code - we'll put it back in a second. Then, use our first form rendering function: form() and pass it the genusForm variable:

1 lines | app/Resources/views/admin/genus/_form.html.twig

```
1  {{ form(genusForm) }}
```

That's it! Refresh! This prints all the fields... but dang! What happened to my submit button!? The form() function renders *everything* in your form... but nothing else. If you like this function, you'll actually need to add a submit button as a field to your form. That's totally supported, but I like rendering my submit buttons by hand. So I like being lazy, but not this lazy.

A Little Less Lazy

If you're feeling just *one* level less lazy, try this: start with form_start(genusForm) form_end(genusForm) and a submit button:

8 lines | app/Resources/views/admin/genus/_form.html.twig

```
1  {{ form_start(genusForm) }}
   ... lines 2 - 5
6      <button type="submit" class="btn btn-primary" formnovalidate>Save</button>
7  {{ form_end(genusForm) }}
```

Then, render all the fields with form_errors(genusForm) and form_widget(genusForm):

8 lines | app/Resources/views/admin/genus/_form.html.twig

```
1  {{ form_start(genusForm) }}
2      {{ form_errors(genusForm) }}
3
4      {{ form_widget(genusForm) }}
5
6      <button type="submit" class="btn btn-primary" formnovalidate>Save</button>
7  {{ form_end(genusForm) }}
```

form_start() adds the starting form tag, but *with* the all-important enctype="multipart/form-data" attribute if your form has a file field. The form_end() function prints the closing form tag *plus* any fields that you forgot to render. That's handy for automatically printing out hidden fields.

Next, this form_errors() is a little strange. Usually, when you have a validation error, it's for one specific field - like "Name is required". But occasionally, you might have an error that doesn't really belong to *one* field, but the form as a whole. This line prints out those rare, but possible, global form errors.

Finally, if you call form_widget() and pass it your *entire* form, it loops over and prints each one.

If you look at the reference now, we've already covered *most* of the form functions.

The Nice Middle ground: form_row

Now, I like to render my forms a bit different. Let me undo my changes. Between the form start and end tags, I usually render each field individually with form_row():

15 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 3
    {{ form_row(genusForm.name) }}
    {{ form_row(genusForm.subFamily) }}
    {{ form_row(genusForm.speciesCount, {
        'label': 'Number of Species'
    }) }}
    {{ form_row(genusForm.funFact) }}
    {{ form_row(genusForm.isPublished) }}
    {{ form_row(genusForm.firstDiscoveredAt) }}

    <button type="submit" class="btn btn-primary" formnovalidate>Save</button>
{{ form_end(genusForm) }}
```

Oh, and we *should* still have the global form errors line: form_errors(genusForm):

15 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
    {{ form_errors(genusForm) }}

    {{ form_row(genusForm.name) }}
    {{ form_row(genusForm.subFamily) }}
    {{ form_row(genusForm.speciesCount, {
        'label': 'Number of Species'
    }) }}
    {{ form_row(genusForm.funFact) }}
    {{ form_row(genusForm.isPublished) }}
    {{ form_row(genusForm.firstDiscoveredAt) }}

    <button type="submit" class="btn btn-primary" formnovalidate>Save</button>
{{ form_end(genusForm) }}
```

It's so rarely needed, I actually forgot to include it before. So... shame on me.

In reality, each field - like name or subFamily - consists of *three* parts: the label, the widget - like an input field, textarea or select element, and the validation errors, if there are any. If you leave "Name" blank and submit, bam! Validation error.

The form_row() function renders all three parts at the same time, *and* wraps them up in some markup that we can control. We'll talk soon about *how* to do that.

Getting Specific

But, if you already need more control, you can skip form_row() and render the three pieces individually: form_label(), form_widget() and form_errors() - passing each the genusForm.name field:

18 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 3
    {{ form_label(genusForm.name) }}
    {{ form_widget(genusForm.name) }}
    {{ form_errors(genusForm.name) }}
... lines 7 - 16
{{ form_end(genusForm) }}
```

Refresh that! It's *almost* the same: the three parts are there, but the red error styling is gone. That makes sense: form_row() prints the three parts, but also surrounds them in some markup, which until now, was giving us some fancy error styling

thanks to Bootstrap CSS.

So let's switch *back* to using form_row() so that *every* field is rendered in a consistent way, *unless* you actually need to do something custom.

So those are your brave and valiant form rendering functions. Each function's first argument is something called "view" - that's just the field - like genusForm.name. We call it a view, and you'll find out why later.

But check *this* out! Most of these functions *also* have an argument called variables. I know, that's a *really* generic-sounding argument. But it turns out that when it comes to kicking butt with form rendering, these *variables* are the key.

Chapter 3: Form Variables are the Bomb

My favorite form rendering function is form_row() - pass the field as the first argument and... um... something weird called *variables* as the second argument. What are these variables?

Overriding the label Variable

Apparently, it's an array, and one variable you can pass is called label. So if you want to override a field's label, one way is with variables.

Give that a try with the subFamily field: add a second argument - a Twig array or hash - and say label set to Taxonomic Subfamily:

18 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
    ... lines 2 - 5
6       {{ form_row(genusForm.subFamily, {
7           'label': 'Taxonomic Subfamily'
8       }) }}
    ... lines 9 - 16
17  {{ form_end(genusForm) }}
```

Try that - refresh! Okay, that's cool.

Overriding the attr Variable

So what *else* can we do with these variables? It also turns out that every field has a variable called attr, which is itself, an array. These are attributes that you want set on the widget, meaning the actual field itself.

So, you can give your field a class foo, or set disabled to disabled:

22 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
    ... lines 2 - 5
6       {{ form_row(genusForm.subFamily, {
7           'label': 'Taxonomic Subfamily',
8           'attr': {
9               'class': 'foo',
10              'disabled': 'disabled'
11          }
12      }) }}
    ... lines 13 - 20
21  {{ form_end(genusForm) }}
```

Try that out. Perfect! The field is disabled and if you dig a little, there's the foo class.

These variables give us *huge* control over our fields. But, this still does *not* answer my original question: what are the variables that I can use on a field beyond just label and attr?

Listing all Available Variables

It turns out, the answer is hiding right down here in your web debug toolbar. Click the clipboard icon to go to the forms section of the profiler. Check this out: if you select a single field - like subFamily - you'll get a *ton* of good information. It shows you stuff about the submitted data, the options you passed when you originally created the field, and - most importantly - View Variables! Yes! This is your master list of *every* variable that's being used to render *this* field. And you can override

- get this - *everything*.

This gives you access to the id attribute, the name attribute, the label we just overrode and even a way to add an attribute to your label element! Heck there's even a variable called disabled: we can just use that instead of setting the attribute.

Remove the disabled attribute, and then set disabled to true:

22 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 5
    {{ form_row(genusForm.subFamily, {
        'label': 'Taxonomic Subfamily',
        'attr': {
            'class': 'foo'
        },
        'disabled': 'disabled',
    }) }}
... lines 13 - 20
{{ form_end(genusForm) }}
```

That'll have the same effect: the field is still disabled.

Field Options Versus Variables

Head back into the profiler. The subFamily field has a variable called placeholder set to "Choose a Sub Family". To see what this does, remove the disabled variable. Then, refresh. There it is! The placeholder is the option that appears at the *top* of the select element.

Why is this set to "Choose a Sub Family"? Because that's what *we* passed as the placeholder *option* when we configured the field:

50 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 6
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 8 - 13
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... line 19
            ->add('subFamily', EntityType::class, [
                'placeholder' => 'Choose a Sub Family',
... lines 22 - 25
            ])
... lines 27 - 39
        ;
    }
... lines 42 - 48
}
```

Back in the template, override the variable, set placeholder to Select a Subfamily:

22 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 5
    {{ form_row(genusForm.subFamily, {
        'label': 'Taxonomic Subfamily',
        'attr': {
            'class': 'foo'
        },
        'placeholder': 'Select a Subfamily'
    }) }}
... lines 13 - 20
{{ form_end(genusForm) }}
```

So, which will win? The placeholder option, or the placeholder variable? Let's find out! Refresh!

It's "Select a Subfamily": the *variable* wins.

I wanted to show you this because this touches on a *really* important thing. When you configure a field in your form class, each field has a set of options. These are *not* the same thing as the variables you can override in your template.

Nope: your form class holds field *options*, and your rendering functions have *variables*. Occasionally, a field has an option and a variable with the same name, like placeholder. But for the most part, these are two totally separate ideas: a field has a set of *options*, which mostly influence how the field should *function*, and a different set of *variables*, which help decide how the field will be *rendered*.

Dumping Form Variables

Before we move into form theming, there's *one* other way to get a list of the variables for a field: dump them!

When we write genusForm.subFamily, this is actually an instance of an object called FormView. A FormView object doesn't really have much information on it, *except* for a public $vars property that holds all of its variables:

163 lines | vendor/symfony/symfony/src/Symfony/Component/Form/FormView.php

```
... lines 1 - 11
namespace Symfony\Component\Form;
... lines 13 - 18
class FormView implements \ArrayAccess, \IteratorAggregate, \Countable
{
    /**
     * The variables assigned to this view.
     *
     * @var array
     */
    public $vars = array(
        'value' => null,
        'attr' => array(),
    );
... lines 30 - 161
}
```

Print them with dump(genusForm.subFamily.vars):

24 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 5
    {{ dump(genusForm.subFamily.vars) }}
... lines 7 - 22
{{ form_end(genusForm) }}
```

Head back, refresh, and boom! Check out this beautiful list. This will become even more important later. Ok, let's talk about form theming.

Remove that dump. Then refresh to look at our nice, normal form. In the template, all we do is call form_row() and then - magically - a whole lot of markup is printed onto the page. So here's the million dollar question: where the heck is that markup coming from? I mean, *somewhere* deep in the core of Symfony, there must be a file that decides what the HTML for an input field is, or what markup to use when printing errors? So, where is that?

The King of Form Markup: form_div_layout.html.twig

The answer: a *single* file that - indeed - lives in the deepest, darkest corners of Symfony. I'll use the Navigate->File shortcut to look for form_div_layout.html.twig:

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
1   {# Widgets #}
2
3   {%- block form_widget -%}
4       {% if compound %}
5           {{- block('form_widget_compound') -}}
6       {% else %}
7           {{- block('form_widget_simple') -}}
8       {% endif %}
9   {%- endblock form_widget -%}
    ... lines 10 - 209
210 {%- block form_label -%}
211     {% if label is not same as(false) -%}
212         {% if not compound -%}
213             {% set label_attr = label_attr|merge({'for': id}) %}
214         {%- endif -%}
215         {% if required -%}
216             {% set label_attr = label_attr|merge({'class': (label_attr.class|default('') ~ ' required')|trim}) %}
217         {%- endif -%}
218         {% if label is empty -%}
219             {%- if label_format is not empty -%}
220                 {% set label = label_format|replace({
221                     '%name%': name,
222                     '%id%': id,
223                 }) %}
224             {%- else -%}
225                 {% set label = name|humanize %}
226             {%- endif -%}
227         {%- endif -%}
228         <label{% for attrname, attrvalue in label_attr %} {{ attrname }}="{{ attrvalue }}"{% endfor %}>{{ translation_domain is same as(fal
229     {%- endif -%}
230 {%- endblock form_label -%}
    ... lines 231 - 243
244 {%- block form_row -%}
245     <div>
246         {{- form_label(form) -}}
247         {{- form_errors(form) -}}
248         {{- form_widget(form) -}}
249     </div>
250 {%- endblock form_row -%}
```

```
... lines 251 - 263
{%- block form -%}
    {{ form_start(form) }}
        {{- form_widget(form) -}}
    {{ form_end(form) }}
{%- endblock form -%}

{%- block form_start -%}
    {% set method = method|upper %}
    {%- if method in ["GET", "POST"] -%}
        {% set form_method = method %}
    {%- else -%}
        {% set form_method = "POST" %}
    {%- endif -%}
    <form name="{{ name }}" method="{{ form_method|lower }}"{% if action != '' %} action="{{ action }}"{% endif %}{% for attrname, attr
    {%- if form_method != method -%}
        <input type="hidden" name="_method" value="{{ method }}" />
    {%- endif -%}
{%- endblock form_start -%}

{%- block form_end -%}
    {%- if not render_rest is defined or render_rest -%}
        {{ form_rest(form) }}
    {%- endif -%}
    </form>
{%- endblock form_end -%}

{%- block form_errors -%}
    {%- if errors|length > 0 -%}
    <ul>
        {%- for error in errors -%}
            <li>{{ error.message }}</li>
        {%- endfor -%}
    </ul>
    {%- endif -%}
{%- endblock form_errors -%}
... lines 299 - 372
```

This is probably the *weirdest* Twig template you'll ever see. It defines a bunch of blocks that, together, contain *every* little bit or markup that's used for *any* part of a form.

And here's how it works: when you render a piece of your form, Symfony opens this template looks for a specific block, which varies depending on *what* you're rendering, and then renders *just* that block to get that *one* little part of your form.

For example, when you render the *widget* for a TextareaType field, it looks for a block called textarea_widget and executes *just* its code:

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 32
{%- block textarea_widget -%}
    <textarea {{ block('widget_attributes') }}>{{ value }}</textarea>
{%- endblock textarea_widget -%}
... lines 36 - 372
```

Symfony never renders this *whole* template at once, just each block, as it needs them. It's almost like a list of small functions, where each function, or block, renders just a small part of the form.

The Bootstrap Theme

But in reality, not all of *our* markup is coming from this one file. In an earlier tutorial, we started using the Bootstrap form *theme*. Open app/config/config.yml and find the twig.form_themes key:

74 lines | app/config/config.yml

```
... lines 1 - 36
# Twig Configuration
twig:
... lines 39 - 42
    form_themes:
        - bootstrap_3_layout.html.twig
... lines 45 - 74
```

By adding bootstrap_3_layout.html.twig, we told Symfony to *also* look at this template, which again, lives deep dark in the core, black heart of Symfony. I'm kidding - the core is cool.

246 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig

```
{% use "form_div_layout.html.twig" %}

{# Widgets #}

{% block form_widget_simple -%}
    {% if type is not defined or type not in ['file', 'hidden'] %}
        {%- set attr = attr|merge({class: (attr.class|default('') ~ ' form-control')|trim}) -%}
    {% endif %}
    {{- parent() -}}
{%- endblock form_widget_simple %}
... lines 11 - 246
```

This template overrides some blocks from form_div_layout.html.twig. For example, form_widget_simple in the bootstrap templates *overrides* the other one. It adds an extra form-control class.

Decrypting the Block Names

You see, there's a trick to these block names. There are three parts to every field: the label, the widget and the error. Well, *four* parts if you also count the "row".

And, every field has a type, like the entity type, the choice type or - for the name field - the text type:

50 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 6
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
... lines 10 - 13
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('subFamily', EntityType::class, [
... lines 21 - 25
            ])
... lines 27 - 28
            ->add('isPublished', ChoiceType::class, [
... lines 30 - 33
            ])
... lines 35 - 39
        ;
    }
... lines 42 - 48
}
```

To render the *widget* for a *text* type, Symfony looks for a block called text_widget. Or, to render the widget for a textarea type, Symfony uses textarea_widget, which looks exactly like we'd expect!

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 32
{%- block textarea_widget -%}
    <textarea {{ block('widget_attributes') }}>{{ value }}</textarea>
{%- endblock textarea_widget -%}
... lines 36 - 372
```

What about rendering the *label* for a *textarea*? We'd expect this to be textarea_label. Find that. Ooh, it's not here! This is because the field types follow a *hierarchy*. First, Symfony looks for textarea_label. But if that's not there, it'll fallback to its parent type: text. So, text_label. And if *that* doesn't exist, it'll finally look for - and find - form_label:

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 209
{%- block form_label -%}
    {% if label is not same as(false) -%}
        {% if not compound -%}
            {% set label_attr = label_attr|merge({'for': id}) %}
        {%- endif -%}
        {% if required -%}
            {% set label_attr = label_attr|merge({'class': (label_attr.class|default('') ~ ' required')|trim}) %}
        {%- endif -%}
        {% if label is empty -%}
            {%- if label_format is not empty -%}
                {% set label = label_format|replace({
                    '%name%': name,
                    '%id%': id,
                }) %}
            {%- else -%}
                {% set label = name|humanize %}
            {%- endif -%}
        {%- endif -%}
        <label{% for attrname, attrvalue in label_attr %} {{ attrname }}="{{ attrvalue }}"{% endfor %}>{{ translation_domain is same as(fal
    {%- endif -%}
{%- endblock form_label -%}
... lines 231 - 372
```

Form is the parent type for all fields.

And this system makes sense! The label for a textarea is no different from a label for any other field. So, *all* labels are rendered via this block.

Another way to see this fallback mechanism is back in the web profiler. Click the name field and then find "View Variables". Every field will have a variable called block_prefixes. *This* shows us the options: after trying text_label, text_widget or text_errors - depending on which part of the field we're rendering, it'll fallback to form_label, form_widget or form_errors.

And actually, there's *also* a way to override the block for just *one* field in your *one* form, by giving it a very specific name. In this case, if you had an _genus_form_name_label block, that would override the label for *only* the name field in this form. Pretty cool.

With *all* this new fun stuff in mind, let's extend this by creating our *own* form theme. The goal: when a field has a validation error, add a cute "X" icon inside the text field. Let's do it!

Chapter 5: Form Theming: Add an Error Icon

Checkout Bootstrap's form documentation. Under validation, they have a cool feature: when your field has an error, you can add a cute icon. I want a cute icon! To get it, we just need to add a has-feedback class to the div around the entire field *and* add the icon itself.

Right now, each field is surrounded by a div with a form-group class. How can we *also* add a has-feedback class to this? Answer: override the block that's responsible for rendering the *row* part of every field. In other words, the form_row block.

In form_div_layout.html.twig, search for the form_row block. There it is!

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 243
{%- block form_row -%}
    <div>
        {{- form_label(form) -}}
        {{- form_errors(form) -}}
        {{- form_widget(form) -}}
    </div>
{%- endblock form_row -%}
... lines 251 - 372
```

But, we might be overriding this in the bootstrap theme - so check there too. Yep, we are: and this is where the form-group class comes from:

246 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig

```
... lines 1 - 184
{% block form_row -%}
    <div class="form-group{% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}">
        {{- form_label(form) -}}
        {{- form_widget(form) -}}
        {{- form_errors(form) -}}
    </div>
{%- endblock form_row %}
... lines 192 - 246
```

Overriding a Block

Ok! So... how can we override this? Very simple. First, copy the block. Second, go to your templates directory and create a new file called _formTheme.html.twig. The name of this isn't file is not important. And just so we know when this is working, add a class: cool-class:

8 lines | app/Resources/views/_formTheme.html.twig

```
{% block form_row -%}
    <div class="form-group cool-class{% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}">
        {{- form_label(form) -}}
        {{- form_widget(form) -}}
        {{- form_errors(form) -}}
    </div>
{%- endblock form_row %}
```

Finally, we need to point Symfony to this new form theme template. And we already know where to do this: right inside config.yml. After the bootstrap template, add a new line with _formTheme.html.twig:

75 lines | app/config/config.yml

```
... lines 1 - 36
# Twig Configuration
twig:
... lines 39 - 42
    form_themes:
        - bootstrap_3_layout.html.twig
        - _formTheme.html.twig
... lines 46 - 75
```

Because this is *after* bootstrap, its blocks will override those from bootstrap. Oh, and even though we don't have it explicitly listed here, Symfony *always* uses form_div_layout.html.twig as the fallback file.

Ok, go back, and refresh! Inspect any form element. There it is! *Our* block is now being used.

Using Variables in Blocks

And here's where things get *really* interesting. We need to add a class to the div, but *only* if this field has a validation error. Well check this out: this block is *already* using a few variables, like compound, force_error and valid:

8 lines | app/Resources/views/_formTheme.html.twig

```
{% block form_row -%}
    <div class="form-group cool-class{% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}">
... lines 3 - 5
    </div>
{%- endblock form_row %}
```

But, where are those coming from? And what other stuff can we use?

It turns out that these are the *same* form variables that we can override from the main, _form.html.twig template. Once you're inside of a form theme block, these become *local* variables.

To see this in action, call dump() with no arguments:

9 lines | app/Resources/views/_formTheme.html.twig

```
{% block form_row -%}
    {{ dump() }}
    <div class="form-group cool-class{% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}">
... lines 4 - 6
    </div>
{%- endblock form_row %}
```

This will print *all* the variables we can use.

Refresh the page. Ah, now we have a big dump before *every* single field: revealing all of the variables we have access to. And it doesn't matter *which* block you're overriding: you always have access to this same, big group of variables. We can use these to only add that has-feedback class *if* there is an error.

Remove the dump. Then, set a new variable called showErrorIcon. Copy *all* of the logic from the if statement below that controls whether or not the has-error class is added and paste it here:

12 lines | app/Resources/views/_formTheme.html.twig

```
{% block form_row -%}
    {% set showErrorIcon = (not compound or force_error|default(false)) and not valid %}
... lines 3 - 10
{%- endblock form_row %}
```

The most important variable is valid: if this is false, the field failed validation. Don't worry about the compound variable - we'll talk about that soon.

Next, at the end of the div, use an inline if statement so that if showErrorIcon is true, we add the has-feedback class:

```
12 lines | app/Resources/views/_formTheme.html.twig
1   {% block form_row -%}
2       {% set showErrorIcon = (not compound or force_error|default(false)) and not valid %}
3       <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
    ... lines 4 - 9
10      </div>
11  {%- endblock form_row %}
```

Then, to add the icon, add that same if statement *after* printing the widget. Add a span with the necessary classes to make this an icon:

```
12 lines | app/Resources/views/_formTheme.html.twig
1   {% block form_row -%}
2       {% set showErrorIcon = (not compound or force_error|default(false)) and not valid %}
3       <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
4           {{- form_label(form) -}}
5           {{- form_widget(form) -}}
6           {% if showErrorIcon %}
7               <span class="glyphicon glyphicon-remove form-control-feedback" aria-hidden="true"></span>
8           {% endif %}
9           {{- form_errors(form) -}}
10      </div>
11  {%- endblock form_row %}
```

Ok, time to try it. Refresh! There's nothing yet, but there also aren't any validation errors. Empty the name field and submit. Our beautiful "X"!

But now, set the Subfamily field to "Select a Subfamily" and submit. Ok, the drop-down looks a little funny - the "X" is on top of the arrow. In fact, the Bootstrap docs warn you about this: this icon should only be added to *text* fields. And other fields, like checkboxes, will look even worse!

So, it's time to get a little smarter, and only add the cute icon to text fields.

Chapter 6: Complex Blocks & the parent() Function

We've just added an X glyphicon to *every* field that has an error...

12 lines | app/Resources/views/_formTheme.html.twig

```
{% block form_row -%}
    {% set showErrorIcon = (not compound or force_error|default(false)) and not valid %}
    <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
... lines 4 - 5
        {% if showErrorIcon %}
            <span class="glyphicon glyphicon-remove form-control-feedback" aria-hidden="true"></span>
        {% endif %}
... line 9
    </div>
{%- endblock form_row %}
```

And then found out that we *really* only want to add this to input fields.

How can we do that? I know that this is a *text* type. So maybe, instead of adding the icon to form_row, we could override the text_widget block and add it there. That would *only* affect text fields.

Go into form_div_layout.html.twig and look for text_widget. Woh! It's not here! That means Symfony must be using form_widget. That block *does* exist:

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 2
{%- block form_widget -%}
    {% if compound %}
        {{- block('form_widget_compound') -}}
    {% else %}
        {{- block('form_widget_simple') -}}
    {% endif %}
{%- endblock form_widget -%}
... lines 10 - 372
```

Remember that compound variable I refused to explain before. Well, here it is again! We normally think of a field as just, well, a field: like a text box, or a select element. But sometimes, a field is actually a collection of *sub-fields*. An easy example is Symfony's DateType, which by default renders as 3 select elements for year, month and day. In that case, the DateType is said to be *compound*: it's just a wrapper for its three child fields.

In our form, all of our fields right now are simple: so, *not* compound. The block() function says:

> Hey! Go render this other block called form_widget_simple.

After following all of this, it turns out that if we want to override the text widget, we need to override form_widget_simple:

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 10
{%- block form_widget_simple -%}
    {%- set type = type|default('text') -%}
    <input type="{{ type }}" {{ block('widget_attributes') }} {% if value is not empty %}value="{{ value }}" {% endif %}/>
{%- endblock form_widget_simple -%}
... lines 15 - 372
```

In fact, *all* input fields - like the number field, search field or URL field - use this same block.

Ok, let's override it! But wait - check to see if it's in the Bootstrap template first:

246 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig

```
... lines 1 - 4
{% block form_widget_simple -%}
    {% if type is not defined or type not in ['file', 'hidden'] %}
        {%- set attr = attr|merge({class: (attr.class|default('') ~ ' form-control')|trim}) -%}
    {% endif %}
    {{- parent() -}}
{%- endblock form_widget_simple %}
... lines 11 - 246
```

It is! Copy that version, and paste it into _formTheme.html.twig.

The Craziness of Twig in Form Themes

Now, check out this logic: form theme templates will have some of the *craziest* Twig code you'll ever see! In normal words, this says:

> If type is not defined or file does not equal type, add a new form-control class.

To make this happen, it uses the attr variable that we were playing with before and *merges* in the new class, adding a space in case there was already a class.

Stealing Parent Blocks with use

Before *we* make any changes, go back and refresh. Woh! It doesn't work - that's surprising. The problem is this parent() call:

246 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig

```
... lines 1 - 4
{% block form_widget_simple -%}
... lines 6 - 8
    {{- parent() -}}
{%- endblock form_widget_simple %}
... lines 11 - 246
```

We understand that in normal Twig templates, you can override parent blocks and use the parent() function. But check this out: our template does *not* extend any Twig template... and we don't want it to! For reasons that honestly aren't very important, a form theme template should never extend anything.

But wait, then, how did this code work in the Bootstrap template? Go look: at the top, it has a use for form_div_layout.html.twig:

246 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig

```
{% use "form_div_layout.html.twig" %}
... lines 2 - 246
```

The use says:

> I don't *actually* want to extend this other template. But, please allow me to call the parent() function as *if* I were extending it.

The use statement is an awesome Twig feature that allows you to just, grab and use random blocks from a different template. It's advanced, but now it's in your toolkit! Go you!

At the top of our template, use 'bootstrap_3_layout.html.twig':

21 lines | app/Resources/views/_formTheme.html.twig

```
{% use "bootstrap_3_layout.html.twig" %}
... lines 2 - 21
```

And as soon as we do that, life is good.

And actually, we don't need this logic anymore: that's done in the parent() block:

21 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 14
{% block form_widget_simple -%}
    {% if type is not defined or type not in ['file', 'hidden'] %}

    {% endif %}
    {{- parent() -}}
{%- endblock form_widget_simple %}
```

If you refresh, everything still looks great.

The Error Icon in the Widget

Finally, we can move the icon to this block. First, keep that showErrorIcon variable: we need that to add the has-feedback class. But copy it and move it down into form_widget_simple, inside the if statement... because, it turns out, we probably also don't want to show the error icon if this is a file upload field:

23 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 11
{% block form_widget_simple -%}
... line 13
    {% if type is not defined or type not in ['file', 'hidden'] %}
        {# show error icon for these types #}
        {% set showErrorIcon = (not compound or force_error|default(false)) and not valid %}
    {% endif %}
    {{- parent() -}}
... lines 19 - 21
{%- endblock form_widget_simple %}
```

Above, set showErrorIcon to false by default:

23 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 11
{% block form_widget_simple -%}
    {% set showErrorIcon = false %}
... lines 14 - 21
{%- endblock form_widget_simple %}
```

Finally, copy the span icon code, remove it, and paste it right *after* the parent call, to put this *after* the widget.

That should do it! Resubmit the form! Got it! One fancy error icon on the name text field, and zero fancy error icons on the select field.

In a nutshell, form theming means:

- (A) finding the right block to override and then;
- (B) leveraging your variables to do cool stuff.

Next, we'll add a missing feature to Symfony: field help text.

Chapter 7: Adding Form Field Help Text

I think we should becomes legends by adding a *totally* new feature to Symfony!

I want to be able to add some help text under each field - ya know to give the user a little bit more info about that field.

Bootstrap already has markup and CSS for this:

```
<span class="help-block">
    A block of help text that breaks onto a new line and may extend beyond one line.
</span>
```

If we add a span and give it a help-block class, it should look great, for free.

Inventing a new help Variable

Tip

Since Symfony 4.1 there is help option you can use on every form field:

```
$builder->add('price', null, [
    'help' => 'In cents',
]);
```

For more information, check this doc: https://symfony.com/blog/new-in-symfony-4-1-form-field-help

This feature doesn't exist in Symfony... yet, but here's how I *want* it to work: I want to be able to pass a variable called help and just have it show up:

24 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
    ... lines 2 - 16
17      {{ form_row(genusForm.isPublished, {
18          help: 'Should this genus be shown on the site?'
19      }) }}
    ... lines 20 - 22
23  {{ form_end(genusForm) }}
```

As promised, nothing happens yet. But! Now that we are passing in a new, invented variable called help, this variable is *already* available in our form theme templates. Head into the form_row block, render dump() with no arguments:

24 lines | app/Resources/views/_formTheme.html.twig

```
    ... lines 1 - 2
3   {% block form_row -%}
    ... line 4
5       <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
    ... lines 6 - 7
8           {{ dump() }}
    ... line 9
10      </div>
11  {%- endblock form_row %}
    ... lines 12 - 24
```

And then refresh.

Search for help: it's alive! The isPublished field is *rocking* that variable.

So yea, you're free to pass in *whatever* variables you want. And that makes us, super dangerous. If the help message is set, then add the span, give it the help-block class and print help:

```
26 lines | app/Resources/views/_formTheme.html.twig
    ... lines 1 - 2
3   {% block form_row -%}
    ... line 4
5       <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
    ... lines 6 - 7
8           {% if help %}
9               <span class="help-block">{{ help }}</span>
10          {% endif %}
    ... line 11
12      </div>
13  {%- endblock form_row %}
    ... lines 14 - 26
```

Using the default Filter Everywhere

Try that out. Oh... and it does not work. Of course: the form_row block is called for *every* field, but the help variable *only* exists for the isPublished field. We need to code defensively! How? Just pipe the help variable to the default filter:

```
26 lines | app/Resources/views/_formTheme.html.twig
    ... lines 1 - 2
3   {% block form_row -%}
    ... line 4
5       <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
    ... lines 6 - 7
8           {% if help|default %}
9               <span class="help-block">{{ help }}</span>
10          {% endif %}
    ... line 11
12      </div>
13  {%- endblock form_row %}
    ... lines 14 - 26
```

Yep, that's it. This says:

> If the help variable does *not* exist, don't throw a big error, just default the variable to null and, have a nice day.

When you use default, you can either pass it a default value - like we did above - or let it use null, which is cool for us.

Try this! Suhweet!

Next, we'll do a bit more work to make our forms friendly for screen readers. Pulling this off gets *really* interesting.

Chapter 8: Go Deeper: Vars, Twig merge & Form Functions

Look closer at the Bootstrap help feature: to be nice to screen readers, we should add an aria-describedby attribute to the field that points to an id that we add to the help span:

```
<label class="sr-only" for="inputHelpBlock">Input with help text</label>
<input type="text" id="inputHelpBlock" class="form-control" aria-describedby="helpBlock">
...
<span id="helpBlock" class="help-block">
  A block of help text that breaks onto a new line and may extend beyond one line.
</span>
```

That way, when a screen reader focuses on the text box, it will read the help text, which is pretty rad. It also turns out that pulling this off is a cool challenge!

Let's start with a plan: when the form_widget() function is called inside form_row:

26 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 2
{% block form_row -%}
... line 4
    <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
... line 6
        {{- form_widget(form) -}}
... lines 8 - 11
    </div>
{%- endblock form_row %}
... lines 14 - 26
```

we want the attr variable to have a new key called aria-describedby. We've seen magic like this before: in the Bootstrap layout, the form_widget_simple block *modifies* the attr variable before calling the parent block:

246 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig

```
... lines 1 - 4
{% block form_widget_simple -%}
    {% if type is not defined or type not in ['file', 'hidden'] %}
        {%- set attr = attr|merge({class: (attr.class|default('') ~ ' form-control')|trim}) -%}
    {% endif %}
    {{- parent() -}}
{%- endblock form_widget_simple %}
... lines 11 - 246
```

That's what we want to do!

Modifying the attr variable

Back in our block, before form_widget(), add another if help|default. Inside, set attr = attr|merge() with an array argument. The core merge filter will array_merge() the argument back into the attr variable. Add aria-describedby set to... well, nothing yet:

30 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 2
{% block form_row -%}
... line 4
    <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
... line 6
        {% if help|default %}
            {# set the aria-describedby attribute #}
            {%- set attr = attr|merge({'aria-describedby': 'help-block-'~id }) -%}
        {% endif %}
... lines 11 - 15
    </div>
{%- endblock form_row %}
... lines 18 - 30
```

First, we need to give our help span an id. Do that: set it to help-block- then print the id variable to make sure this is unique:

30 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 2
{% block form_row -%}
... line 4
    <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
... line 6
        {% if help|default %}
            {# set the aria-describedby attribute #}
            {%- set attr = attr|merge({'aria-describedby': 'help-block-'~id }) -%}
        {% endif %}
... line 11
        {% if help|default %}
            <span class="help-block" id="help-block-{{ id }}">{{ help }}</span>
        {% endif %}
... line 15
    </div>
{%- endblock form_row %}
... lines 18 - 30
```

The id variable will become the id attribute on the field itself.

Now set the aria-describedby to help-block-, a ~ then id:

30 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 2
{% block form_row -%}
... line 4
    <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
... line 6
        {% if help|default %}
            {# set the aria-describedby attribute #}
            {%- set attr = attr|merge({'aria-describedby': 'help-block-'~id }) -%}
        {% endif %}
... lines 11 - 15
    </div>
{%- endblock form_row %}
... lines 18 - 30
```

The ~ is Twig's rarely-used concatenation operator, so, it's like . in PHP.

Ok! Now that attr is changed *before* we call form_widget, it'll hopefully render on that widget. Time to give it a try. Refresh!

Ok, go dig into the source to see if the attribute is there. Umm... it's not! There is *not* any aria-describedby. This tutorial is a LIE!

To Pass or Not Pass Variables

No no, it's cool. It turns out that there's a very subtle, but important detail that I'm neglecting. Let me show you: click to open the parent form_div_layout.html.twig template. We're letting Symfony *guess* this, but the speciesCount is a NumberType, meaning it'll render as an <input type="number" /> field:

50 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 13
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 19 - 26
            ->add('speciesCount')
... lines 28 - 39
        ;
    }
... lines 42 - 48
}
```

Inside the layout file, find the number_widget block that renders this:

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 133
{%- block number_widget -%}
    {# type="number" doesn't work with floats #}
    {%- set type = type|default('text') -%}
    {{ block('form_widget_simple') }}
{%- endblock number_widget -%}
... lines 139 - 372
```

Ok, check it out: it sets a type variable, and then calls the form_widget_simple block. Then, when form_widget_simple executes, type is set to number:

372 lines | vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```
... lines 1 - 10
{%- block form_widget_simple -%}
    {%- set type = type|default('text') -%}
    <input type="{{ type }}" {{ block('widget_attributes') }} {% if value is not empty %}value="{{ value }}" {% endif %}/>
{%- endblock form_widget_simple -%}
... lines 15 - 372
```

So, why does *that* work, but not our code? Well look at that code again: it sets a variable and then calls the *block* function. When you call block(), all variables flow through to that block.

But now check out our code: we set a variable, but then we don't execute a block! We call form_widget(). Hey! that's a form rendering function - the same kind that we use inside our normal templates. In this case, the variables *do not* magically flow through. But that's ok! We already know how to pass variables into form_widget(). Add a second argument, and pass attr set to attr:

32 lines | app/Resources/views/_formTheme.html.twig

```
... lines 1 - 2
{% block form_row -%}
... line 4
    <div class="form-group {% if (not compound or force_error|default(false)) and not valid %} has-error{% endif %}{{ showErrorIcon ? '
... lines 6 - 10
        {{- form_widget(form, {
            'attr': attr
        }) -}}
... lines 14 - 17
    </div>
{%- endblock form_row %}
... lines 20 - 32
```

Let's refresh! Inspect the field, well, not *any* field - inspect the isPublished field. This time, we got it!

So not only are you a form-theming pro, but you're quickly becoming a Twig all star.

Chapter 9: Form Options & Variables: Dream Team

We now know that these form variables kick butt, *and* we know how to override them from inside a template. But, could we *also* control these from inside of our form class?

Earlier, I mentioned that the *options* for a field are totally different than the *variables* for a field. Occasionally, a field has an option - like placeholder - *and* a variable with the same name, but that's not always true. But clearly, there must be *some* connection between options and variables. So what is it?!

Form Type Classes & Options

First, behind every field type is a class. Obviously, for the subFamily field, the class behind this is EntityType:

50 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 6
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 8 - 13
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('subFamily', EntityType::class, [
... lines 21 - 25
            ])
... lines 27 - 39
        ;
    }
... lines 42 - 48
}
```

name is a *text* type, so the class behind it, is, well, TextType. I'll use the Shift+Shift shortcut in my editor to open the TextType file, from the Symfony Form component:

69 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/TextType.php

```
... lines 1 - 11
12  namespace Symfony\Component\Form\Extension\Core\Type;
13
14  use Symfony\Component\Form\AbstractType;
15  use Symfony\Component\Form\DataTransformerInterface;
16  use Symfony\Component\Form\FormBuilderInterface;
17  use Symfony\Component\OptionsResolver\OptionsResolver;
18
19  class TextType extends AbstractType implements DataTransformerInterface
20  {
21      public function buildForm(FormBuilderInterface $builder, array $options)
22      {
23          // When empty_data is explicitly set to an empty string,
24          // a string should always be returned when NULL is submitted
25          // This gives more control and thus helps preventing some issues
26          // with PHP 7 which allows type hinting strings in functions
27          // See https://github.com/symfony/symfony/issues/5906#issuecomment-203189375
28          if ('' === $options['empty_data']) {
29              $builder->addViewTransformer($this);
30          }
31      }
... lines 32 - 35
36      public function configureOptions(OptionsResolver $resolver)
37      {
38          $resolver->setDefaults(array(
39              'compound' => false,
40          ));
41      }
... lines 42 - 45
46      public function getBlockPrefix()
47      {
48          return 'text';
49      }
... lines 50 - 53
54      public function transform($data)
55      {
56          // Model data should not be transformed
57          return $data;
58      }
... lines 59 - 63
64      public function reverseTransform($data)
65      {
66          return null === $data ? '' : $data;
67      }
68  }
```

Now, unlike variables, there is a specific *set* of valid options for a field. If you pass an option that doesn't exist, Symfony will scream at you. The *valid* options for a field are determined by this configureOptions() method:

69 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/TextType.php

```
... lines 1 - 16
use Symfony\Component\OptionsResolver\OptionsResolver;

class TextType extends AbstractType implements DataTransformerInterface
{
... lines 21 - 35
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'compound' => false,
        ));
    }
... lines 42 - 67
}
```

Apparently the TextType has a compound option, and it defaults to false.

Form Type Inheritance

Earlier, when we talked about form theme blocks, I mentioned that the field types have a built-in *inheritance* system. Well, technically, TextType extends AbstractType, but behind-the-scenes, the TextType's parent type is FormType. In fact, *every* field ultimately inherits options from FormType. Open that class:

195 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/FormType.php

```
... lines 1 - 11
namespace Symfony\Component\Form\Extension\Core\Type;
... lines 13 - 21
use Symfony\Component\PropertyAccess\PropertyAccess;
use Symfony\Component\PropertyAccess\PropertyAccessorInterface;

class FormType extends BaseType
{
    /**
     * @var PropertyAccessorInterface
     */
    private $propertyAccessor;

    public function __construct(PropertyAccessorInterface $propertyAccessor = null)
    {
        $this->propertyAccessor = $propertyAccessor ?: PropertyAccess::createPropertyAccessor();
    }
... lines 36 - 193
}
```

Tip

Wondering how you would know what the "parent" type of a field is? Each *Type class has a getParent() method that will tell you. If you don't see one, then it's defaulting to FormType.

This is cool because it *also* has a configureOptions() method that adds a *bunch* of options:

195 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/FormType.php

```
... lines 1 - 24
class FormType extends BaseType
{
... lines 27 - 120
```

```php
    public function configureOptions(OptionsResolver $resolver)
    {
        parent::configureOptions($resolver);

        // Derive "data_class" option from passed "data" object
        $dataClass = function (Options $options) {
            return isset($options['data']) && is_object($options['data']) ? get_class($options['data']) : null;
        };

        // Derive "empty_data" closure from "data_class" option
        $emptyData = function (Options $options) {
            $class = $options['data_class'];

            if (null !== $class) {
                return function (FormInterface $form) use ($class) {
                    return $form->isEmpty() && !$form->isRequired() ? null : new $class();
                };
            }

            return function (FormInterface $form) {
                return $form->getConfig()->getCompound() ? array() : '';
            };
        };

        // For any form that is not represented by a single HTML control,
        // errors should bubble up by default
        $errorBubbling = function (Options $options) {
            return $options['compound'];
        };

        // If data is given, the form is locked to that data
        // (independent of its value)
        $resolver->setDefined(array(
            'data',
        ));

        $resolver->setDefaults(array(
            'data_class' => $dataClass,
            'empty_data' => $emptyData,
            'trim' => true,
            'required' => true,
            'property_path' => null,
            'mapped' => true,
            'by_reference' => true,
            'error_bubbling' => $errorBubbling,
            'label_attr' => array(),
            'inherit_data' => false,
            'compound' => true,
            'method' => 'POST',
            // According to RFC 2396 (http://www.ietf.org/rfc/rfc2396.txt)
            // section 4.2., empty URIs are considered same-document references
            'action' => '',
            'attr' => array(),
            'post_max_size_message' => 'The uploaded file was too large. Please try to upload a smaller file.',
        ));
```

```
176
177         $resolver->setAllowedTypes('label_attr', 'array');
178     }
    ... lines 179 - 193
194 }
```

These are the options that are available to *every* field type. And actually, the parent class - BaseType - has even more:

125 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php

```
    ... lines 1 - 11
12  namespace Symfony\Component\Form\Extension\Core\Type;
    ... lines 13 - 27
28  abstract class BaseType extends AbstractType
29  {
    ... lines 30 - 109
110     public function configureOptions(OptionsResolver $resolver)
111     {
112         $resolver->setDefaults(array(
113             'block_name' => null,
114             'disabled' => false,
115             'label' => null,
116             'label_format' => null,
117             'attr' => array(),
118             'translation_domain' => null,
119             'auto_initialize' => true,
120         ));
121
122         $resolver->setAllowedTypes('attr', 'array');
123     }
124 }
```

There are easier ways to find out the valid options for a field - like the documentation or the form web profiler tab. But sometimes, being able to see how an option is *used* in these classes, might help you find the right value.

The label Option versus Variable

Let's see an example. In the form, we add a subFamily field:

50 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 6
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 8 - 13
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... line 19
            ->add('subFamily', EntityType::class, [
                'placeholder' => 'Choose a Sub Family',
                'class' => SubFamily::class,
                'query_builder' => function(SubFamilyRepository $repo) {
                    return $repo->createAlphabeticalQueryBuilder();
                }
            ])
... lines 27 - 39
        ;
    }
... lines 42 - 48
}
```

Then, in the template, we override the label variable:

24 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 5
    {{ form_row(genusForm.subFamily, {
        'label': 'Taxonomic Subfamily',
... lines 8 - 11
    }) }}
... lines 13 - 22
{{ form_end(genusForm) }}
```

But, according to BaseType, this field, well *any* field, also has a label *option*:

125 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php

```
... lines 1 - 27
abstract class BaseType extends AbstractType
{
... lines 30 - 109
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
... lines 113 - 114
            'label' => null,
... lines 116 - 119
        ));
... lines 121 - 122
    }
}
```

The Form to FormView Transition

That's interesting! Let's see if we can figure out how the option and variable work together. Scroll up in BaseType. These

classes also have another function called buildView():

125 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php

```
... lines 1 - 27
abstract class BaseType extends AbstractType
{
... lines 30 - 41
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        $name = $form->getName();
        $blockName = $options['block_name'] ?: $form->getName();
        $translationDomain = $options['translation_domain'];
        $labelFormat = $options['label_format'];

        if ($view->parent) {
            if ('' !== ($parentFullName = $view->parent->vars['full_name'])) {
                $id = sprintf('%s_%s', $view->parent->vars['id'], $name);
                $fullName = sprintf('%s[%s]', $parentFullName, $name);
                $uniqueBlockPrefix = sprintf('%s_%s', $view->parent->vars['unique_block_prefix'], $blockName);
            } else {
                $id = $name;
                $fullName = $name;
                $uniqueBlockPrefix = '_'.$blockName;
            }

            if (null === $translationDomain) {
                $translationDomain = $view->parent->vars['translation_domain'];
            }

            if (!$labelFormat) {
                $labelFormat = $view->parent->vars['label_format'];
            }
        } else {
            $id = $name;
            $fullName = $name;
            $uniqueBlockPrefix = '_'.$blockName;

            // Strip leading underscores and digits. These are allowed in
            // form names, but not in HTML4 ID attributes.
            // http://www.w3.org/TR/html401/struct/global.html#adef-id
            $id = ltrim($id, '_0123456789');
        }

        $blockPrefixes = array();
        for ($type = $form->getConfig()->getType(); null !== $type; $type = $type->getParent()) {
            array_unshift($blockPrefixes, $type->getBlockPrefix());
        }
        $blockPrefixes[] = $uniqueBlockPrefix;

        $view->vars = array_replace($view->vars, array(
            'form' => $view,
            'id' => $id,
            'name' => $name,
            'full_name' => $fullName,
            'disabled' => $form->isDisabled(),
            'label' => $options['label'],
```

```
90              'label' => $options['label'],
91              'label_format' => $labelFormat,
92              'multipart' => false,
93              'attr' => $options['attr'],
94              'block_prefixes' => $blockPrefixes,
95              'unique_block_prefix' => $uniqueBlockPrefix,
96              'translation_domain' => $translationDomain,
97              // Using the block name here speeds up performance in collection
98              // forms, where each entry has the same full block name.
99              // Including the type is important too, because if rows of a
100             // collection form have different types (dynamically), they should
101             // be rendered differently.
102             // https://github.com/symfony/symfony/issues/5038
103             'cache_key' => $uniqueBlockPrefix.'_'.$form->getConfig()->getType()->getBlockPrefix(),
104         ));
105     }
    ... lines 106 - 123
124 }
```

In a controller, when you pass your form into a template, you always call createView() on it first:

86 lines | src/AppBundle/Controller/Admin/GenusAdminController.php

```
    ... lines 1 - 15
16  class GenusAdminController extends Controller
17  {
    ... lines 18 - 63
64      public function editAction(Request $request, Genus $genus)
65      {
    ... lines 66 - 81
82          return $this->render('admin/genus/edit.html.twig', [
83              'genusForm' => $form->createView()
84          ]);
85      }
86  }
```

That line turns out to be *very* important: it transforms your Form object into a FormView object.

In fact, your form is a big tree, with a Form on top and fields below it, which themselves are *also* Form objects. When you call createView(), all of the Form objects are transformed into FormView objects.

To do that, the buildView() method is called on each individual field. And one of the arguments to buildView() is an array of the final *options* passed to this field:

125 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php

```
    ... lines 1 - 15
16  use Symfony\Component\Form\FormInterface;
17  use Symfony\Component\Form\FormView;
    ... lines 18 - 27
28  abstract class BaseType extends AbstractType
29  {
    ... lines 30 - 41
42      public function buildView(FormView $view, FormInterface $form, array $options)
43      {
    ... lines 44 - 104
105     }
    ... lines 106 - 123
124 }
```

For example, for subFamily, we're passing three options:

50 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 13
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... line 19
            ->add('subFamily', EntityType::class, [
                'placeholder' => 'Choose a Sub Family',
                'class' => SubFamily::class,
                'query_builder' => function(SubFamilyRepository $repo) {
                    return $repo->createAlphabeticalQueryBuilder();
                }
            ])
... lines 27 - 39
        ;
    }
... lines 42 - 48
}
```

We could also pass a label option here.

These values - merged with any other default values set in configureOptions() - are then passed to buildView() and... if you scroll down a little bit, many of them are used to populate the *vars* on the FormView object of this field. Yep, these are the same *vars* that become so important when rendering that field:

125 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php

```
... lines 1 - 27
28  abstract class BaseType extends AbstractType
29  {
... lines 30 - 41
42      public function buildView(FormView $view, FormInterface $form, array $options)
43      {
... lines 44 - 83
84          $view->vars = array_replace($view->vars, array(
85              'form' => $view,
86              'id' => $id,
87              'name' => $name,
88              'full_name' => $fullName,
89              'disabled' => $form->isDisabled(),
90              'label' => $options['label'],
91              'label_format' => $labelFormat,
92              'multipart' => false,
93              'attr' => $options['attr'],
94              'block_prefixes' => $blockPrefixes,
95              'unique_block_prefix' => $uniqueBlockPrefix,
96              'translation_domain' => $translationDomain,
97              // Using the block name here speeds up performance in collection
98              // forms, where each entry has the same full block name.
99              // Including the type is important too, because if rows of a
100             // collection form have different types (dynamically), they should
101             // be rendered differently.
102             // https://github.com/symfony/symfony/issues/5038
103             'cache_key' => $uniqueBlockPrefix.'_'.$form->getConfig()->getType()->getBlockPrefix(),
104         ));
105     }
... lines 106 - 123
124 }
```

To put it simply: every field has options and *sometimes* these options are used to set the form *variables* that control how the field is rendered:

125 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/BaseType.php

```
... lines 1 - 27
28  abstract class BaseType extends AbstractType
29  {
... lines 30 - 41
42      public function buildView(FormView $view, FormInterface $form, array $options)
43      {
... lines 44 - 83
84          $view->vars = array_replace($view->vars, array(
... lines 85 - 89
90              'label' => $options['label'],
... lines 91 - 92
93              'attr' => $options['attr'],
... lines 94 - 103
104         ));
105     }
... lines 106 - 123
124 }
```

Symfony gives us a label *option* as a convenient way to ultimately set the label variable.

Close up all those classes. Here's a question: we know how to set the help variable from inside of a Twig template. But could we somehow set this variable from inside of the GenusFormType class? Yes, and there are actually *two* cool ways to do it. Let's check them out.

Chapter 10: Controlling Vars with finishView()

Question: can we control form *view* variables directly from inside GenusFormType? Of course! Use the "Code"->"Generate" menu, or Command+N on a Mac, click "Override Methods" and then select a method called finishView().

There are actually *two* methods that are called when your form is transformed into a FormView object: buildView() and finishView(): one is called at the beginning, and the other at the end.

In this case, we want finishView():

57 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 11
use Symfony\Component\Form\FormInterface;
use Symfony\Component\Form\FormView;
... lines 14 - 15
class GenusFormType extends AbstractType
{
... lines 18 - 44
    public function finishView(FormView $view, FormInterface $form, array $options)
    {
... line 47
    }
... lines 49 - 55
}
```

Don't worry about calling the parent function, it's empty.

The FormView object that's passed to this method is the final, top-level FormView object that represents the entire form. Now, check this out: use $view['funFact']->vars['help'] = and then type a message, like, a nice fun fact suggestion:

57 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 15
class GenusFormType extends AbstractType
{
... lines 18 - 44
    public function finishView(FormView $view, FormInterface $form, array $options)
    {
        $view['funFact']->vars['help'] = 'For example, Leatherback sea turtles can travel more than 10,000 miles every year!';
    }
... lines 49 - 55
}
```

Congratulations: you've just set that view variable. But let's break it down. The $view variable is the *top* of our FormView tree. To get a FormView for a specific field, access it like an array key.

At this point, $view['funFact'] gives you the *same* FormView object that you would get in a template by calling genusForm.funFact. Then, we access the public vars array property and add a help key to it. Ultimately, this adds that view variable.

Refresh to check it out. It works! And now there's nothing we can't change!

But let's do something even harder. Copy the help string, then comment out the finishView() method entirely. Find the funFact field above, pass null as the second option so that Symfony keeps guessing the field type, then add a new help *option*:

54 lines | src/AppBundle/Form/GenusFormType.php

```
<?php
... lines 2 - 15
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 21 - 29
            ->add('funFact', null, [
                'help' => 'For example, Leatherback sea turtles can travel more than 10,000 miles every year!'
            ])
... lines 33 - 43
        ;
    }
... lines 46 - 52
}
```

I want *this* to ultimately set a help variable for me.

But if you try it now, *huge* error!

> The option "help" does not exist.

That's no surprise: I just invented this option! But, we *can* make this work.

Chapter 11: Form Type Extension Magic

The only way to make an option valid for a field is to, well, hack the core class and add it! For example, since funFact is a TextType, I *could* - if I were feeling crazy - open TextType, scroll down, and hack that help option into configureOptions():

69 lines | vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/TextType.php

```
... lines 1 - 16
use Symfony\Component\OptionsResolver\OptionsResolver;

class TextType extends AbstractType implements DataTransformerInterface
{
... lines 21 - 35
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'compound' => false,
        ));
    }
... lines 42 - 67
}
```

With a few other hacks, we could have this feature working!

Form Plugins: Type Extensions

Obviously, this is *not* the right path to take, and that's ok, but there is *another* way to add an option to a field. It's called a form *type extension*, and it's basic a plugin to the Symfony form field system. By leveraging a form type extension, you can modify *any* field in the system. You could, I don't know, add a new attribute to literally every single field on your entire site.

Let's find out how.

Creating a Form Type Extension

In your Form directory, create a new directory called TypeExtension and then a new class called HelpFormExtension:

14 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 2
namespace AppBundle\Form\TypeExtension;
... lines 4 - 6
class HelpFormExtension extends AbstractTypeExtension
{
... lines 9 - 12
}
```

The goal of this class will be to to allow for a help option to be passed to any field, *and* to turn that help option into a help variable.

First, all form type extensions should extend AbstractTypeExtension:

14 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 2
namespace AppBundle\Form\TypeExtension;

use Symfony\Component\Form\AbstractTypeExtension;

class HelpFormExtension extends AbstractTypeExtension
{
... lines 9 - 12
}
```

Next, use the "Code"->"Generate" menu, or Command+N on a Mac, and click "Implement Methods". This abstract class requires us to have one method: getExtendedType():

14 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 2
namespace AppBundle\Form\TypeExtension;

use Symfony\Component\Form\AbstractTypeExtension;

class HelpFormExtension extends AbstractTypeExtension
{
    public function getExtendedType()
    {
        // TODO: Implement getExtendedType() method.
    }
}
```

You see, when you create a form type extension, you could make it modify *every* field in your entire system, *or* just one type, like the FileType. To modify *every* field, return FormType::class:

22 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 5
use Symfony\Component\Form\Extension\Core\Type\FormType;
... lines 7 - 9
class HelpFormExtension extends AbstractTypeExtension
{
... lines 12 - 16
    public function getExtendedType()
    {
        return FormType::class;
    }
}
```

Because remember, FormType is the parent for all fields.

Tip

Technically, FormType is the parent type for *all* fields, except for buttons. But I don't like adding buttons to my form anyways!

Overriding Field Behavior

Here's the cool thing about these classes: they have all the same functions as a normal form class, like buildForm() or configureOptions(). The difference is that whatever modifications we make to this class will literally be applied to *every* field in the system.

For example, go back to the "Code"->"Generate" menu, click "Override Methods", then select buildView():

22 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 6
use Symfony\Component\Form\FormInterface;
use Symfony\Component\Form\FormView;

class HelpFormExtension extends AbstractTypeExtension
{
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
... line 14
    }
... lines 16 - 20
}
```

When we're done setting things up, whenever *any* field is transformed into a FormView object, this method will be called and *we* will be able to add variables to anything!

Try it: add $view - which represents whatever one field is being setup - $view->vars['help'] set to TURTLES!

22 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 9
class HelpFormExtension extends AbstractTypeExtension
{
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        $view->vars['help'] = 'TURTLES!';
    }
... lines 16 - 20
}
```

That's a ridiculous, and yet, fully-functional type extension.

Registering a Type Extension

To tell Symfony about this, you guys can probably guess, we need to register this as a service. In app/config/services.yml, add a new service: call it app.form.help_form_extension. Set its class to HelpFormExtension and then I'll set autowire to true... even though the class doesn't have any constructor arguments, at least not yet:

33 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 27
    app.form.help_form_extenion:
        class: AppBundle\Form\TypeExtension\HelpFormExtension
        autowire: true
... lines 31 - 33
```

Then, to actually tell Symfony: "Hey! This is a form type extension!", add a tag, set to form.type_extension. Also give this an extended_type option. This needs to match whatever you're returning from getExtendedType(). FormType::class returns the long string in the use statement, so copy that, and paste it into your service:

33 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 27
    app.form.help_form_extenion:
        class: AppBundle\Form\TypeExtension\HelpFormExtension
        autowire: true
        tags:
            - { name: form.type_extension, extended_type: Symfony\Component\Form\Extension\Core\Type\FormType }
```

That's it team! Temporarily remove the help option from GenusFormType, ya know, so the page doesn't explode. Then, refresh! OMG, everyone is screaming about TURTLES! Well, everyone except for the isPublished field, because we're overriding that help variable at the last possible second: from inside the template.

Using the Option to Fuel the help Variable

Finally, uncomment the help option. So, how can we make this a valid option? Go back to HelpFormExtension, use the "Code"->"Generate" menu one last time, click "Override Methods", and select configureOptions():

30 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 8
use Symfony\Component\OptionsResolver\OptionsResolver;

class HelpFormExtension extends AbstractTypeExtension
{
... lines 13 - 19
    public function configureOptions(OptionsResolver $resolver)
    {
... line 22
    }
... lines 24 - 28
}
```

Our job here is *so* simple: $resolver->setDefault('help', null):

30 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 10
class HelpFormExtension extends AbstractTypeExtension
{
... lines 13 - 19
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefault('help', null);
    }
... lines 24 - 28
}
```

Just by doing that, you are now allowed to have a help option on any field. It also means that when buildView() is called, the $options array will have a key called help. All we need to say is if $options['help'], then set the help variable to $options['help']:

30 lines | src/AppBundle/Form/TypeExtension/HelpFormExtension.php

```
... lines 1 - 10
class HelpFormExtension extends AbstractTypeExtension
{
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        if ($options['help']) {
            $view->vars['help'] = $options['help'];
        }
    }
... lines 19 - 28
}
```

And that takes care of it. Try this puppy out.

And consider yourself very, very dangerous.

Chapter 12: Compound & Embedded Forms

Right now, this is a pretty simple form. We have our top level form, and then each field below it is its own Form object. And we now know that when you pass this into the template, all of those Form objects become FormView objects.

But this will still just be *2* levels: the FormView object on top, and the children FormView object for each field. But, it can get a lot more complicated than that.

To show you, go into GenusFormType. For now, change the firstDiscoveredAt options: comment out widget and attr:

54 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 15
16  class GenusFormType extends AbstractType
17  {
18      public function buildForm(FormBuilderInterface $builder, array $options)
19      {
20          $builder
... lines 21 - 38
39              ->add('firstDiscoveredAt', DateType::class, [
40                  //'widget' => 'single_text',
41                  //'attr' => ['class' => 'js-datepicker'],
42                  'html5' => false,
43              ])
44          ;
45      }
... lines 46 - 52
53  }
```

Refresh this immediately. Ok, the widget option defaults to choice, which means that this renders as three select fields. I know, it's horribly ugly, hard to look at... but it's a perfect example! Click into the profiler for this form to see something *really* interesting. The firstDiscoveredAt has a "+" next to it... and three fields below it!

Compound Fields!

You see, firstDiscoveredAt is no longer a "simple" field: it's now a field that consists of 3 sub-fields: year, month, and day. Each of these is their own ChoiceType field. Oh, and if you select firstDiscoveredAt, under "View Variables", for the first time, the compound variable is set to true.

We saw this compound variable in a few places earlier. And now we know what it means! A field is compound if it's not really its own field, but is instead just a container for sub-fields.

In the _form.html.twig template, when we call form_row() on genusForm.firstDiscoveredAt, Symfony tries to render the parent field, notices that it's compound and so, calls form_row() on each of its three sub-fields:

24 lines | app/Resources/views/admin/genus/_form.html.twig

```
1   {{ form_start(genusForm) }}
... lines 2 - 19
20      {{ form_row(genusForm.firstDiscoveredAt) }}
... lines 21 - 22
23  {{ form_end(genusForm) }}
```

The result is the nice output we're already seeing.

Rendering Sub-Fields

To get more control, you could instead call form_row on each individual field: for year, month and day:

27 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 20
    {{ form_row(genusForm.firstDiscoveredAt.year) }}
    {{ form_row(genusForm.firstDiscoveredAt.month) }}
    {{ form_row(genusForm.firstDiscoveredAt.day) }}
... lines 24 - 25
{{ form_end(genusForm) }}
```

But notice that if this field fails validation, the error is attached to the *parent* field. So you might want to keep rendering form_label(genusForm.firstDiscoveredAt) and you definitely want to keep rendering form_errors(genusForm.firstDiscoveredAt), so that the error shows up.

If you go back and refresh, you basically see the same thing as before. It's ugly, but you just learned how to take control of *any* level of a complex form tree.

Chapter 13: Rendering Fields Manually

Finally, let's look at the Swiss Army knife of form rendering: instead of using the form-rendering functions, we'll build the field entirely by hand.

For example, suppose we need to do something *crazy* with the "year" drop-down field. That's fine! We'll still render the label and errors normally, but let's handle the widget ourselves. Yep, I literally mean: create a select tag and start filling in the details.

The first detail is the id attribute. Every field has a unique id, which ties that field to its label. And this is where form variables help us out *big*.

Referencing Field Variables Directly

Go back into the Form tab of the web profiler and click the year field. There are a lot of variables, but there are a few that are *especially* important, like id and full_name, which normally becomes the name attribute.

In your template, reference the id variable with: genusForm.firstDiscoveredAt.year.vars.id. Repeat that for the name attribute set to genusForm.firstDiscoveredAt.year.vars.full_name:

35 lines | app/Resources/views/admin/genus/_form.html.twig

```
1  {{ form_start(genusForm) }}
   ... lines 2 - 20
21     {{ form_label(genusForm.firstDiscoveredAt.year) }}
22     <select id="{{ genusForm.firstDiscoveredAt.year.vars.id }}"
23             name="{{ genusForm.firstDiscoveredAt.year.vars.full_name }}">
   ... lines 24 - 26
27     </select>
28     {{ form_errors(genusForm.firstDiscoveredAt.year) }}
29
30     {{ form_row(genusForm.firstDiscoveredAt.month) }}
31     {{ form_row(genusForm.firstDiscoveredAt.day) }}
   ... lines 32 - 33
34 {{ form_end(genusForm) }}
```

Now that we understand the FormView tree and how variables are stored, this actually makes sense.

Printing the Options

Next, what about the options that should go inside the select tag? Head back to the web profiler to see which variable might help us. Ah, here's one called choices, and each item is a ChoiceView object. Use the Shift+Shift shortcut to open *that* file from Symfony:

65 lines | vendor/symfony/symfony/src/Symfony/Component/Form/ChoiceList/View/ChoiceView.php

```
... lines 1 - 11
12  namespace Symfony\Component\Form\ChoiceList\View;
... lines 13 - 18
19  class ChoiceView
20  {
21      /**
22       * The label displayed to humans.
23       *
24       * @var string
25       */
26      public $label;
27
28      /**
29       * The view representation of the choice.
30       *
31       * @var string
32       */
33      public $value;
34
35      /**
36       * The original choice value.
37       *
38       * @var mixed
39       */
40      public $data;
41
42      /**
43       * Additional attributes for the HTML tag.
44       *
45       * @var array
46       */
47      public $attr;
48
49      /**
50       * Creates a new choice view.
51       *
52       * @param mixed  $data  The original choice
53       * @param string $value The view representation of the choice
54       * @param string $label The label displayed to humans
55       * @param array  $attr  Additional attributes for the HTML tag
56       */
57      public function __construct($data, $value, $label, array $attr = array())
58      {
59          $this->data = $data;
60          $this->value = $value;
61          $this->label = $label;
62          $this->attr = $attr;
63      }
64  }
```

Cool! Each ChoiceView is a simple object, with a public label property and a public value property:

65 lines | vendor/symfony/symfony/src/Symfony/Component/Form/ChoiceList/View/ChoiceView.php

```
... lines 1 - 18
class ChoiceView
{
    /**
     * The label displayed to humans.
     *
     * @var string
     */
    public $label;

    /**
     * The view representation of the choice.
     *
     * @var string
     */
    public $value;
... lines 34 - 63
}
```

That's exactly what we need.

Add a loop: for choice in genusForm.firstDiscoveredAt.year.vars.choices:

35 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 20
    {{ form_label(genusForm.firstDiscoveredAt.year) }}
    <select id="{{ genusForm.firstDiscoveredAt.year.vars.id }}"
            name="{{ genusForm.firstDiscoveredAt.year.vars.full_name }}">
        {% for choice in genusForm.firstDiscoveredAt.year.vars.choices %}
... line 25
        {% endfor %}
    </select>
    {{ form_errors(genusForm.firstDiscoveredAt.year) }}

    {{ form_row(genusForm.firstDiscoveredAt.month) }}
    {{ form_row(genusForm.firstDiscoveredAt.day) }}
... lines 32 - 33
{{ form_end(genusForm) }}
```

Inside, add <option value=""> then print choice.value.

We also need to know if this option should be currently selected. We can do that by comparing the value to the data variable that's attached to the year field. Why not do this in one big giant line:
choice.value == genusForm.firstDiscoveredAt.year.vars.data. Wow. Then, ? ' selected' or empty quotes. Finally, for the option text, use choice.label:

35 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 20
    {{ form_label(genusForm.firstDiscoveredAt.year) }}
    <select id="{{ genusForm.firstDiscoveredAt.year.vars.id }}"
            name="{{ genusForm.firstDiscoveredAt.year.vars.full_name }}">
        {% for choice in genusForm.firstDiscoveredAt.year.vars.choices %}
            <option value="{{ choice.value }}" {{ choice.value == genusForm.firstDiscoveredAt.year.vars.data ? 'selected' : '' }}>{{ choice.la
        {% endfor %}
    </select>
    {{ form_errors(genusForm.firstDiscoveredAt.year) }}

    {{ form_row(genusForm.firstDiscoveredAt.month) }}
    {{ form_row(genusForm.firstDiscoveredAt.day) }}
... lines 32 - 33
{{ form_end(genusForm) }}
```

That's it! Go back to your browser, then refresh. Ah, error!

That's me being careless: the sub-field is called year, not years:

35 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 21
    <select id="{{ genusForm.firstDiscoveredAt.year.vars.id }}"
            name="{{ genusForm.firstDiscoveredAt.year.vars.full_name }}">
        {% for choice in genusForm.firstDiscoveredAt.year.vars.choices %}
            <option value="{{ choice.value }}" {{ choice.value == genusForm.firstDiscoveredAt.year.vars.data ? 'selected' : '' }}>{{ choice.la
        {% endfor %}
    </select>
... lines 28 - 33
{{ form_end(genusForm) }}
```

Refresh again.

It works! It's not styled because we've taken complete control of rendering it. But you *can* see the errors, and the options look correct. Cool!

Marking Fields as Rendered

So, we're done! Wait... except for this random field at the bottom of my form. What the heck!? That's my year field! What's going on?

See that form_end() at the bottom of our form?

35 lines | app/Resources/views/admin/genus/_form.html.twig

```
... lines 1 - 33
{{ form_end(genusForm) }}
```

Remember how it renders *any* field that we forgot to render? Well, now it thinks that *we* forgot to render the year field. The nerve!

So, could we just *tell* it that the field *was* actually rendered? Yep, and the code is both simple and strange. Use a rare *do* tag from Twig and say genusForm.firstDiscoveredAt.year.setRendered():

36 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 28
    {% do genusForm.firstDiscoveredAt.year.setRendered() %}
... lines 30 - 34
{{ form_end(genusForm) }}
```

Whaaaat? Well, every field is a FormView object. And if you open that class, it has a setRendered() method!

163 lines | vendor/symfony/symfony/src/Symfony/Component/Form/FormView.php

```
... lines 1 - 18
class FormView implements \ArrayAccess, \IteratorAggregate, \Countable
{
... lines 21 - 86
    /**
     * Marks the view as rendered.
     *
     * @return FormView The view object
     */
    public function setRendered()
    {
        $this->rendered = true;

        return $this;
    }
... lines 98 - 161
}
```

And by calling it, we're saying:

> Yo, we rendered this already. So, you know, don't try to render it again.

Refresh now! Whoops! Another Ryan mistake - make sure your variable is genusForm, not genus:

36 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 28
    {% do genusForm.firstDiscoveredAt.year.setRendered() %}
... lines 30 - 34
{{ form_end(genusForm) }}
```

Now that extra field is gone.

Wrap it Up!

Congrats team: you have the power to render your forms in whatever crazy, insane, creative way you want! But with power, comes great responsibility. I'll delete all the code we just added and go back to simply rendering genusForm.firstDiscoveredAt:

24 lines | app/Resources/views/admin/genus/_form.html.twig

```
{{ form_start(genusForm) }}
... lines 2 - 20
    {{ form_row(genusForm.firstDiscoveredAt) }}
... line 22
{{ form_end(genusForm) }}
```

54 lines | src/AppBundle/Form/GenusFormType.php

```
... lines 1 - 15
class GenusFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 21 - 38
            ->add('firstDiscoveredAt', DateType::class, [
                'widget' => 'single_text',
                'attr' => ['class' => 'js-datepicker'],
                'html5' => false,
            ])
        ;
    }
... lines 46 - 52
}
```

Don't use your new skills unless you actually need to.

Ok guys, that's it! If you still have some questions, or want to tell me about something really cool you did, or share vacation photos, whatever, you can do it in the comments - it's always great to hear from you.

All right guys, see you next time.