

Symfony Forms: Build, Render & Conquer!

With <3 from KnpUniversity

Chapter 1: The Form Type Class

Hey guys! You're back! Awesome! Because this tutorial is all about *forms*: the good, the bad, and the ugly.

Quit Hatin' on the Forms

The truth is: the form component is super controversial: some people love it, some people hate it - and a lot of people have trouble learning it. In fact, its documentation on symfony.com is read far more than any other section.

Why? Because honestly, it *is* complex - too complex sometimes. But you know what? The form component is going to allow you to get a *lot* of work done really quickly. And when I *do* see someone suffering with forms, most of the time, it's their fault. They create situations that are far more complicated than they need to be.

So let's *not* do that - let's *enjoy* forms, and turn them into a weapon.

~~Sing Along~~ Code along!

Ok, you guys know the drill: download the course code and unzip it to code along with me. Inside, you'll find the answers to life's questions and a `start/` directory that has the same code I have here. Make sure to check out the README for all the setup details.

Once you're ready, start the built-in web server with:

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

I made a *few* changes to the site since last time. For example, go to `localhost:8000/genus`. It looks the same, but see the "Sub Family"? Before, that was a string field on `Genus`. But now, I've added a `SubFamily` entity:

```
40 lines | src/AppBundle/Entity/SubFamily.php
1  ... lines 1 - 6
7  /**
8   * @ORM\Entity
9   * @ORM\Table(name="sub_family")
10  */
11  class SubFamily
12  {
13      /**
14       * @ORM\Id
15       * @ORM\GeneratedValue(strategy="AUTO")
16       * @ORM\Column(type="integer")
17       */
18      private $id;
19
20      /**
21       * @ORM\Column(type="string")
22       */
23      private $name;
24  ... lines 24 - 38
39  }
```

And created a `ManyToOne` relation from `Genus` to `SubFamily` :

```
119 lines | src/AppBundle/Entity/Genus.php
```

```

12 class Genus
13 {
14     ... lines 14 - 25
15     /**
16      * @ORM\ManyToOne(targetEntity="AppBundle\Entity\SubFamily")
17      * @ORM\JoinColumn(nullable=false)
18      */
19     private $subFamily;
20     ... lines 31 - 117
21 }

```

So every **Genus** belongs to one **SubFamily**.

I also started a new admin section - see it at **/admin/genus**. But, it needs some work - like the ability to add *a new* genus. That'll be our job. And the code will live in the new **GenusAdminController**.

Creating a new Form

To create a form, you'll add a class where you'll *describe* what the form looks like.

💡 Tip

Actually, you can build a form directly in the controller if you want to.

In PhpStorm, select anywhere in your bundle and press **command + N**, or right click and select "New". Find "Form" and call it **GenusFormType**:

```

21 lines | src/AppBundle/Form/GenusFormType.php
22 ... lines 1 - 2
23 namespace AppBundle\Form;
24
25 use Symfony\Component\Form\AbstractType;
26 use Symfony\Component\Form\FormBuilderInterface;
27 use Symfony\Component\OptionsResolver\OptionsResolver;
28
29 class GenusFormType extends AbstractType
30 {
31     public function buildForm(FormBuilderInterface $builder, array $options)
32     {
33     }
34
35     public function configureOptions(OptionsResolver $resolver)
36     {
37     }
38 }

```

Cool! This just gave us a basic skeleton and put the class in a **Form** directory. Sensible! These classes are called "form types"... which is the worst name that we could come up with when the system was created. Sorry. Really, these classes are "form recipes".

Here's how it works: in **buildForm()**: start adding fields: **\$builder->add()** and then **name** to create a "name" field:

```

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

```

```

1  ... lines 1 - 8
9  class GenusFormType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder
14             ->add('name')
15     }
16
17     ;
18 }
19
20 ... lines 19 - 23
24 }

```

Keep going: `add('speciesCount')` and `add('funFact')` :

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

```

1  ... lines 1 - 12
13     $builder
14         ->add('name')
15         ->add('speciesCount')
16         ->add('funFact')
17     ;
18
19 ... lines 18 - 25

```

Right now, those field names can be anything - you'll see why in a second.

And that's it! The form is built after writing about 4 lines of code! Let's go render it!

Chapter 2: Render that Form Pretty (Bootstrap)

Head into the `GenusAdminController`. I gave us a *tiny* head start - there's already a `newAction()`:

```
39 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 11
12 class GenusAdminController extends Controller
13 {
1  ... lines 14 - 27
28 /**
29  * @Route("/genus/new", name="admin_genus_new")
30  */
31 public function newAction()
32 {
1  ... lines 33 - 37
38 }
39 }
```

In the admin area, the "Add" button points here. Click it!

The controller must return a response (`null` given). Did you forget to add a return statement somewhere in your controller?

Yep, a great explosion!

Instantiating the Form Object

To create the form object, add `$form = $this->createForm()`. This is a shortcut method in the base `Controller` that calls a method on the `form.factory` service. That's my friendly reminder to you that *everything* - including form magic - is done by a service.

To `createForm()`, pass it the form type class name - `GenusFormType::class`:

```
39 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 4
5  use AppBundle\Form\GenusFormType;
1  ... lines 6 - 11
12 class GenusAdminController extends Controller
13 {
1  ... lines 14 - 30
31 public function newAction()
32 {
33     $form = $this->createForm(GenusFormType::class);
1  ... lines 34 - 37
38 }
39 }
```

And because I used autocomplete, that *did* just add the `use` statement for me. The `::class` syntax is new in PHP 5.5 - and we're going to use it a lot.

Rendering the Form

Now that we have a form object, just render a template a pass it in: `return $this->render()` with `admin/genus/new.html.twig` to somewhat follow the directory structure of the controller:

```
39 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
```

```

1  ... lines 1 - 11
12 class GenusAdminController extends Controller
13 {
1  ... lines 14 - 30
31     public function newAction()
32     {
33         $form = $this->createForm(GenusFormType::class);
34
35         return $this->render('admin/genus/new.html.twig', [
1  ... line 36
37             ]);
38     }
39 }

```

Pass in one variable `genusForm` set to `$form->createView()` :

```

39 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 11
12 class GenusAdminController extends Controller
13 {
1  ... lines 14 - 30
31     public function newAction()
32     {
33         $form = $this->createForm(GenusFormType::class);
34
35         return $this->render('admin/genus/new.html.twig', [
36             'genusForm' => $form->createView()
37         ]);
38     }
39 }

```

Don't forget that `createView()` part - it's something we'll talk about more in a future course about form theming.

Hold `command` + `click` to jump into the template. Yep, I took the liberty of already creating this for us in the `app/Resources/views/admin/genus` directory:

```

17 lines | app/Resources/views/admin/genus/new.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6              <div class="col-xs-12">
7                  <h1>New Genus</h1>
1  ... lines 8 - 13
14              </div>
15          </div>
16      </div>
17  {% endblock %}

```

Here, we know we have a `genusForm` variable. So... how can we render it? You can't render! I'm kidding - you totally can, by using several special Twig functions that Symfony gives us.

The Form Twig Functions

First, we need an opening form tag. Render that with `form_start(genusForm)` :

```

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

```

```

1 ... lines 1 - 2
3 {% block body %}
4     <div class="container">
5         <div class="row">
6             <div class="col-xs-12">
7                 <h1>New Genus</h1>
8
9                 {{ form_start(genusForm) }}
10 ... lines 10 - 13
14             </div>
15         </div>
16     </div>
17 {% endblock %}

```

To add a closing form tag, add `form_end(genusForm)` :

```

17 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 2
3 {% block body %}
4     <div class="container">
5         <div class="row">
6             <div class="col-xs-12">
7                 <h1>New Genus</h1>
8
9                 {{ form_start(genusForm) }}
10 ... lines 10 - 12
13                 {{ form_end(genusForm) }}
14             </div>
15         </div>
16     </div>
17 {% endblock %}

```

I know, having functions to create the HTML form tag seems a little silly. But, wait `form_start()` is cool because it will add the `enctype="multipart/form-data"` attribute if the form has an upload field. And the `form_end()` function takes care of rendering hidden fields. So, these guys are my friends.

Between them, render all three fields at once with `form_widget(genusForm)` :

```

17 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 2
3 {% block body %}
4     <div class="container">
5         <div class="row">
6             <div class="col-xs-12">
7                 <h1>New Genus</h1>
8
9                 {{ form_start(genusForm) }}
10                 {{ form_widget(genusForm) }}
11 ... lines 11 - 12
13                 {{ form_end(genusForm) }}
14             </div>
15         </div>
16     </div>
17 {% endblock %}

```

And finally, we need a button! We can do that by hand: `<button type="submit">`, give it a few Bootstrap classes and call it "Save":

```

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

```

```

1  ... lines 1 - 2
3  {% block body %}
4      <div class="container">
5          <div class="row">
6              <div class="col-xs-12">
7                  <h1>New Genus</h1>
8
9                  {{ form_start(genusForm) }}
10                 {{ form_widget(genusForm) }}
11
12                 <button type="submit" class="btn btn-primary">Save</button>
13                 {{ form_end(genusForm) }}
14             </div>
15         </div>
16     </div>
17 {% endblock %}

```

💡 Tip

You *can* also add buttons as fields to your form. But this is helpful in very few cases, so I prefer just to render them by hand.

that's it! Head to the browser and refresh.

There it is! A rendered form with almost no work. They're all text boxes now: we'll customize them soon.

The Bootstrap Form Theme

Of course, it *is* pretty ugly... Symfony has default HTML markup that it uses to render everything you're seeing: the labels, the inputs and any validation errors.

Since we're using Bootstrap, it would be *really* cool if Symfony could automatically render the fields using Bootstrap-friendly markup.

Yep, that's built-in. Open `app/config/config.yml`. Under `twig`, add `form_themes` and then below that `- bootstrap3_layout.html.twig`. Actually, make that `bootstrap_3_layout.html.twig`:

```

74 lines | app/config/config.yml
1  ... lines 1 - 36
37 # Twig Configuration
38 twig:
1  ... lines 39 - 42
43     form_themes:
44         - bootstrap_3_layout.html.twig
1  ... lines 45 - 74

```

Form themes are how we can control the markup used to render forms. The `bootstrap_3_layout.html.twig` template lives in the core of Symfony and now, our form markup will change to use HTML bits that live inside of it.

Try it out. Beautiful. Now, let's submit this form and do something with its data.

Chapter 3: Process that Form!

Inspect the HTML and check out the `<form>` element. Notice: this does *not* have an `action` attribute. This means that the form will submit right back to the same route and controller that renders it. You *can* totally change this, but we won't.

In other words, our single action method will be responsible for *rendering* the form *and* processing it when the request method is POST.

Before we do any processing, we need the request. Type-hint it as an argument:

```
46 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 7
8  use Symfony\Component\HttpFoundation\Request;
1  ... lines 9 - 12
13 class GenusAdminController extends Controller
14 {
1  ... lines 15 - 31
32 public function newAction(Request $request)
33 {
1  ... lines 34 - 44
45 }
46 }
```

`$form->handleRequest()`

Next, to actually handle the submit, call `$form->handleRequest()` and pass it the `$request` object:

```
46 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 12
13 class GenusAdminController extends Controller
14 {
1  ... lines 15 - 31
32 public function newAction(Request $request)
33 {
34     $form = $this->createForm(GenusFormType::class);
35
36     // only handles data on POST
37     $form->handleRequest($request);
1  ... lines 38 - 44
45 }
46 }
```

This is really cool: the `$form` knows what fields it has on it. So `$form->handleRequest()` goes out and grabs the post data off of the `$request` object for those specific fields and processes them.

The *confusing* thing is that this *only* does this for POST requests. If this is a GET request - like the user simply navigated to the form - the `handleRequest()` method does nothing and our form renders just like it did before.



Tip

You *can* configure the form to submit on GET requests if you want. This is useful for search forms

But if the form was just submitted, then we'll want to do something with that information, like save a new `Genus` to the database. Add an if statement: `if ($form->isSubmitted() && $form->isValid()) :`

```
46 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
```

```

1  ... lines 1 - 12
13 class GenusAdminController extends Controller
14 {
1  ... lines 15 - 31
32 public function newAction(Request $request)
33 {
34     $form = $this->createForm(GenusFormType::class);
35
36     // only handles data on POST
37     $form->handleRequest($request);
38     if ($form->isSubmitted() && $form->isValid()) {
1  ... line 39
40     }
1  ... lines 41 - 44
45 }
46 }

```

In other words, if this is a POST request and if the form passed all validation. We'll add validation soon.

Fetching \$form->getData()

If we get inside this `if` statement, life is good. For now, just dump the submitted data: `dump($form->getData())` and then `die;`:

```

46 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 12
13 class GenusAdminController extends Controller
14 {
1  ... lines 15 - 31
32 public function newAction(Request $request)
33 {
34     $form = $this->createForm(GenusFormType::class);
35
36     // only handles data on POST
37     $form->handleRequest($request);
38     if ($form->isSubmitted() && $form->isValid()) {
39         dump($form->getData());die;
40     }
1  ... lines 41 - 44
45 }
46 }

```

OK, let's see what this dumps out!

Fill out the form with very realistic data and submit. Check that out! It dumps an associative array with the three fields we added to the form. That's so simple! We added 3 fields to the form, rendered them in a template, and got those three values as an associative array.

Now, it would be very, very easy to use that associative array to create a new `Genus` object, populate it with the data, and save it via Doctrine.

But, it would be awesomesauce if the form framework could do our job for us. I mean, use the data to automatically create the `Genus` object, populate it, and return *that* to us. Let's do that.

Chapter 4: Binding Forms to Objects: data_class

Open `GenusFormType` and find the `configureOptions()` method. Add `$resolver->setDefaults()`. Pass that an array with a single key called `data_class` set to `AppBundle\Entity\Genus`:

```
27 lines | src/AppBundle/Form/GenusFormType.php
... lines 1 - 8
9 class GenusFormType extends AbstractType
10 {
... lines 11 - 19
20 public function configureOptions(OptionsResolver $resolver)
21 {
22     $resolver->setDefaults([
23         'data_class' => 'AppBundle\Entity\Genus'
24     ]);
25 }
26 }
```

Do *nothing* else: refresh to re-submit the form.

💡 Tip

You can also use a newer syntax in PHP for this:

```
'data_class' => Genus::class
```

Most editors will auto-complete this for you!

Boom! Now we have a brand-new `Genus` object that's just waiting to be saved. Thanks to the `data_class` option, the form creates a new `Genus` object behind the scenes. And then it sets the data on it.

Earlier, when we got back an associative array, these field names - `name`, `speciesCount` and `funFact` – could have been anything:

```
27 lines | src/AppBundle/Form/GenusFormType.php
... lines 1 - 8
9 class GenusFormType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder
14             ->add('name')
15             ->add('speciesCount')
16             ->add('funFact')
17         ;
18     }
... lines 19 - 25
26 }
```

But as soon as you bind your form to a class, `name`, `speciesCount` and `funFact` need to match property names inside of your class:

```
134 lines | src/AppBundle/Entity/Genus.php
```

```

12 class Genus
13 {
24     private $name;
35     private $speciesCount;
40     private $funFact;
133 }

```

Actually, that's *kind of* a lie. These properties are private, so the form component *can't* set them directly. In reality, it guesses a setter function for each field and call that: `setName()` , `setSpeciesCount()` and `setFunFact()` :

```

134 lines | src/AppBundle/Entity/Genus.php
12 class Genus
13 {
68     public function setName($name)
69     {
70         $this->name = $name;
71     }
91     public function setSpeciesCount($speciesCount)
92     {
93         $this->speciesCount = $speciesCount;
94     }
101    public function setFunFact($funFact)
102    {
103        $this->funFact = $funFact;
104    }
133 }

```

Technically, you could add a form field call `outOnAMagicalJourney` as long as you had a public method in your class called `setOutOnAMagicalJourney()` .

Form Field Guessing!

Head back to your browser, highlight the URL and hit enter. This just made a GET request, which skipped form processing and just rendered the template.

Let's add a few more field we need: like `subFamily` :

```

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

```

```

1 ... lines 1 - 8
9 class GenusFormType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder
14             ->add('name')
15             ->add('subFamily')
16             ->add('speciesCount')
17             ->add('funFact')
18 ... lines 18 - 19
20     };
21 }
19 ... lines 22 - 28
29 }

```

Hey, we're even getting auto-complete now: PhpStorm knows `Genus` has a `subFamily` property!

Also add `isPublished` - that should eventually be a checkbox - and `firstDiscoveredAt` - that will need to be some sort of date field:

```

30 lines | src/AppBundle/Form/GenusFormType.php
1 ... lines 1 - 8
9 class GenusFormType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array $options)
12     {
13         $builder
14             ->add('name')
15             ->add('subFamily')
16             ->add('speciesCount')
17             ->add('funFact')
18             ->add('isPublished')
19             ->add('firstDiscoveredAt')
20     };
21 }
19 ... lines 22 - 28
29 }

```

Cool, try it out!

Huge error!

Catchable Fatal Error: Object of class `SubFamily` could not be converted to string

Okay: that's weird. What's going on?

Until now, it looked like Symfony renders every field as an input text field by default. But that's not true! There's a lot more coolness going on behind the scenes!

In reality, the form system looks at each field and tries to *guess* what type of field it should be. For example, for `subFamily`, it sees that this is a `ManyToOne` relationship to `SubFamily`:

```

134 lines | src/AppBundle/Entity/Genus.php

```

```

12 class Genus
13 {
14     ... lines 14 - 25
26     /**
27      * @ORM\ManyToOne(targetEntity="AppBundle\Entity\SubFamily")
28      * @ORM\JoinColumn(nullable=false)
29      */
30     private $subFamily;
31     ... lines 31 - 132
133 }

```

So, it tries to render this as a select drop-down of sub families. That's amazing, because it's *exactly* what we want.

But, it needs to be able to turn a `SubFamily` object into a string so it can render the text for each option in the select. That's the source of the error.

To help it, add a `public function __toString()` to the `SubFamily` class:

```

45 lines | src/AppBundle/Entity/SubFamily.php
11 class SubFamily
12 {
13     ... lines 13 - 39
40     public function __toString()
41     {
42         return $this->getName();
43     }
44 }

```

Refresh again!

Look at this! A free drop-down with almost no work. It also noticed that `isPublished` should be a checkbox because that's a `boolean` field in Doctrine:

```

134 lines | src/AppBundle/Entity/Genus.php
12 class Genus
13 {
14     ... lines 14 - 41
42     /**
43      * @ORM\Column(type="boolean")
44      */
45     private $isPublished = true;
46
47     /**
48      * @ORM\Column(type="date")
49      */
50     private $firstDiscoveredAt;
51     ... lines 51 - 132
133 }

```

And since `firstDiscoveredAt` is a date, it rendered it with year-month-day drop-down boxes. Now, those three boxes are *totally* ugly and we'll fix it later, but isn't it cool that it's *guessing* the right field types?

Fill out the form again with super-realistic data and submit. Woh! One more error:

Neither the property `isPublished` nor one of the methods `getIsPublished()` exist and have public access in class `Genus`

Remember how every form field needs a setter function on your class? Like `name` and `setName()` ? Every field *also* needs a *getter* function - like `getIsPublished()` or one of these other variations.

This was my bad: when I set this up, I added an `isPublished` property, a `setIsPublished()` method, but no getter! I'll use the "Code" -> "Generate" menu - or `command + N` - to generate that getter:

```
139 lines | src/AppBundle/Entity/Genus.php
12 class Genus
13 {
14 ... lines 14 - 115
116     public function getIsPublished()
117     {
118         return $this->isPublished;
119     }
120 ... lines 120 - 137
138 }
```

Refresh! It dumps the `Genus` object of course, but check out the `subFamily` field! It's not the `SubFamily ID` - the form field took the submitted ID, queried the database for the `SubFamily` object and set *that* on the property. That's HUGE.

We're ready to save this!

Chapter 5: Save, Redirect, setFlash (and Dance)

We already have the finished `Genus` object. So what do we do now? Whatever we want!

Probably... we want to save this to the database. Add `$genus = $form->getData()`. Get the entity manager with `$em = $this->getDoctrine()->getManager()`. Then, the classic `$em->persist($genus)` and `$em->flush()`:

```
52 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 12
13 class GenusAdminController extends Controller
14 {
15  ... lines 15 - 31
32 public function newAction(Request $request)
33 {
34  ... lines 34 - 37
38     if ($form->isSubmitted() && $form->isValid()) {
39         $genus = $form->getData();
40
41         $em = $this->getDoctrine()->getManager();
42         $em->persist($genus);
43         $em->flush();
44     ... lines 44 - 45
46     }
47  ... lines 47 - 50
51 }
52 }
```

Always Redirect!

Next, we *always* redirect after a successful form submit - ya know, to make sure that the user can't just refresh and re-post that data. That'd be lame.

To do that, `return $this->redirectToRoute()`. Hmm, generate a URL to the `admin_genus_list` route - that's the main admin page I created before the course:

```
52 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 12
13 class GenusAdminController extends Controller
14 {
15  ... lines 15 - 31
32 public function newAction(Request $request)
33 {
34  ... lines 34 - 37
38     if ($form->isSubmitted() && $form->isValid()) {
39  ... lines 39 - 44
45         return $this->redirectToRoute('admin_genus_list');
46     }
47  ... lines 47 - 50
51 }
52 }
```

Because `redirectToRoute()` returns a `RedirectResponse`, we're done!

Time to try it out. I'll be lazy and refresh the POST. We *should* get a brand new "Sea Monster" genus. There it is! Awesome!

Adding a Super Friendly (Flash) Message

Now, it worked... but it lack some spirit! There was no "Success! You're amazing! You created a new genus!" message.

And I want to build a friendly site, so let's add that message. Back in `newAction()` , add some code *right* before the redirect:
`$this->addFlash('success')` - you'll see where that key is used in a minute - then `Genus created - you are amazing!` :

```
54 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 12
13 class GenusAdminController extends Controller
14 {
1  ... lines 15 - 31
32 public function newAction(Request $request)
33 {
1  ... lines 34 - 37
38 if ($form->isSubmitted() && $form->isValid()) {
1  ... lines 39 - 44
45     $this->addFlash('success', 'Genus created!');
46
47     return $this->redirectToRoute('admin_genus_list');
48 }
1  ... lines 49 - 52
53 }
54 }
```

It's good to encourage users.

But let's be curious and see what this does behind the scenes. Hold `command` and click into the `addFlash()` method:

```
398 lines | vendor/symfony/symfony/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php
1  ... lines 1 - 38
39 abstract class Controller implements ContainerAwareInterface
40 {
1  ... lines 41 - 102
103 /**
104  * Adds a flash message to the current session for type.
105  *
106  * @param string $type The type
107  * @param string $message The message
108  *
109  * @throws \LogicException
110  */
111 protected function addFlash($type, $message)
112 {
113     if (!$this->container->has('session')) {
114         throw new \LogicException("You can not use the addFlash method if sessions are disabled.");
115     }
116
117     $this->container->get('session')->getFlashBag()->add($type, $message);
118 }
1  ... lines 119 - 396
397 }
```

Okay, cool: it uses the `session` service, fetches something called a "flash bag" and adds our message to it. So the flash bag is a special part of this session where you can store messages that will automatically disappear *after* one redirect. If we store a message here and then redirect, on the next page, we can read those messages from the flash bag, and print them on the page. And because the message automatically disappears after one redirect, we won't accidentally show it to the user more than once.

And actually, it's even a little bit cooler than that. A message will *actually* stay in the flash bag until you ask for it. Then it's removed. This is good because if you - for some reason - redirect *twice* before rendering the message, no problem! It'll stay in there and wait for you.

Rendering the Flash Message

All we need to do now is render the flash message. And the best place for this is in your base template. Because then, you can set a flash message, redirect to *any* other page, and it'll always show up.

Right above the `body` block, add `for msg in app.session` - the shortcut to get the `session` service - `.flashbag.get()` and then the `success` key. Add the `endfor` :

```
49 lines | app/Resources/views/base.html.twig
1  <!DOCTYPE html>
2  <html>
  ... lines 3 - 13
14  <body>
  ... lines 15 - 28
29      <div class="main-content">
30          {% for msg in app.session.flashBag.get('success') %}
  ... lines 31 - 33
34          {% endfor %}
  ... lines 35 - 36
37      </div>
  ... lines 38 - 46
47  </body>
48  </html>
49
```

Why `success` ? Because that's what we used in the controller - but this string is arbitrary:

```
54 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
  ... lines 1 - 12
13  class GenusAdminController extends Controller
14  {
  ... lines 15 - 31
32      public function newAction(Request $request)
33      {
  ... lines 34 - 37
38          if ($form->isSubmitted() && $form->isValid()) {
  ... lines 39 - 44
45              $this->addFlash('success', 'Genus created!');
  ... lines 46 - 47
48          }
  ... lines 49 - 52
53      }
54  }
```

Usually I have one for `success` that I style green and happy, and one called `error` that style to be red and scary.

I'll make this happy with the `alert-success` from bootstrap and then render `msg` :

```
49 lines | app/Resources/views/base.html.twig
```

```

1  <!DOCTYPE html>
2  <html>
  ... lines 3 - 13
14  <body>
  ... lines 15 - 28
29  <div class="main-content">
30    {% for msg in app.session.flashBag.get('success') %}
31      <div class="alert alert-success">
32        {{ msg }}
33      </div>
34    {% endfor %}
  ... lines 35 - 36
37  </div>
  ... lines 38 - 46
47  </body>
48  </html>
49

```

Cool! Go back and create `Sea Monster2`. Change its subfamily, give it a species count and save that sea creature! Ocean conservation has never been so easy.

And, I'm feeling the warm and fuzzy from our message.

Next, let's *really* start to control how the fields are rendered.

Chapter 6: Field Types and Options

So far, we set the property name of the form fields and Symfony tries to *guess* the best field type to render. Since `isPublished` is a `boolean` in the database, it guessed a checkbox field:

```
139 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 11
12 class Genus
13 {
... lines 14 - 41
42 /**
43  * @ORM\Column(type="boolean")
44  */
45 private $isPublished = true;
... lines 46 - 137
138 }
```

But obviously, that's not always going to work.

Google for "Symfony form field types", and find a document called [Form Types Reference](#).

These are *all* the different field types we can choose from. It's got stuff you'd expect - like `text` and `textarea`, some HTML5 fields - like `email` and `integer` and some more unusual types, like `date`, which helps render date fields, either as three drop-downs or a single text field.

Right now, `isPublished` is a checkbox. But instead, I'd rather have a select drop-down with "yes" and "no" options. But... you won't see a "select" type in this list. Instead, it's called `ChoiceType`.

Using ChoiceType

`ChoiceType` is a little special: it can render a drop-down, radio buttons or checkboxes based on what options you pass to it. One of the options is `choices`, which, if you look down at the example, you can see controls the actual items in the drop-down.

Let's use this! In `GenusFormType`, the optional second argument to the `add` function is the field *type* you want to use. Set it to `ChoiceType::class`:

```
36 lines | src/AppBundle/Form/GenusFormType.php
... lines 1 - 5
6 use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
... lines 7 - 9
10 class GenusFormType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array $options)
13     {
14         $builder
... lines 15 - 18
19         ->add('isPublished', ChoiceType::class, [
... lines 20 - 23
24         ])
... line 25
26     ;
27 }
... lines 28 - 34
35 }
```

The third argument is an array of options to configure that field. What can you pass here? Well, each field type has different options... though there are a lot of options shared by *all* types, like the ability to customize the label.

Either way, the reference section will tell you what you can use. Pass in the `choices` option and set that to an array. Add `Yes` mapped to `true` and `No` mapped to `false`. The keys - `Yes` and `No` will be the text in the drop down:

```
36 lines | src/AppBundle/Form/GenusFormType.php
10 class GenusFormType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array $options)
13     {
14         $builder
15             ->add('isPublished', ChoiceType::class, [
16                 'choices' => [
17                     'Yes' => true,
18                     'No' => false,
19                 ]
20             ])
21     }
22 }
23
24 ... lines 28 - 34
35 }
```

The values - `true` and `false` will be the value that's passed to `setIsPublished()` if that option is chosen.

Try it out! Head back to the form and refresh! Perfect.

The EntityType

Let's keep going with this. This `subFamily` field is also a drop-down. So that's probably being *guessed* as a `ChoiceType`, right? *Almost*. Check out the `EntityType`. This is *just* like the `ChoiceType`, except it's really good at fetching the options by querying an entity.

In this case, it's automatically querying for the `SubFamily` entity: it guessed which type to use *and* auto-configured it. And yeah, I know - these sub-families are totally silly. They're actually just the last names of people - coming from the Faker library - I was being lazy.

Form Field Options

But I do have one problem with this `EntityType` field: it auto-selects the first option. That's lame - I'd rather have an option on top that says "Choose a Sub-Family".

Check out the options for `EntityType`: there's one called `placeholder`. Click to read more about that. Yes! This is exactly what we want!

Open the form class back up. We know that Symfony is guessing the `EntityType` for the `subFamily` field. We *could* now manually pass `EntityType::class` as the second argument. But don't! Pass `null` instead. Now, add a `placeholder` option set to `Choose a Sub-Family`:

```
39 lines | src/AppBundle/Form/GenusFormType.php
```

```

11 class GenusFormType extends AbstractType
12 {
13     public function buildForm(FormBuilderInterface $builder, array $options)
14     {
15         $builder
16         ... line 16
17         ->add('subFamily', null, [
18             'placeholder' => 'Choose a Sub Family'
19         ])
20         ... lines 20 - 28
29     };
30 }
31 ... lines 31 - 37
38 }

```

But wait, why did I pass `null` as the second argument? Well first, because I can! I can be lazy here: if I pass `null`, Symfony will guess the `EntityType`. That's cool.

Second, when you use the `EntityType`, there's a required option called `class`. Let me show you an example: `'class' => 'AppBundle:User'`. You have to tell it *which* entity to query from. But if you let Symfony *guess* the field type for you, then it will also guess any options it can, including this one. So by being lazy and passing `null`, it will continue to guess the field type *and* a few other options for me, like `class`.

Anyways, go back, refresh, and there it is. Here are the key takeaways. First, you have a giant dictionary of built-in form field types. And second: you configure each field by passing a third argument to the `add()` function.

Now, how could we *further* control the query that's made for the `SubFamily` options? What if we need them to be listed alphabetically?

Chapter 7: Custom Query in EntityType

As cool as it is that it's guessing my field type, I am actually going to add `EntityType::class` to use this type *explicitly*.

```
43 lines | src/AppBundle/Form/GenusFormType.php
1  ... lines 1 - 4
5  use Symfony\Bridge\Doctrine\Form\Type\EntityType;
1  ... lines 6 - 10
11 class GenusFormType extends AbstractType
12 {
13     public function buildForm(FormBuilderInterface $builder, array $options)
14     {
15         $builder
16
17         ->add('subFamily', EntityType::class, [
18
19         ... lines 18 - 22
23     ])
24
25     ... lines 24 - 32
33     ;
34 }
35
36 ... lines 35 - 41
42 }
```

I don't *have* to do this: I just want to show you guys what a traditional setup looks like.

Now, do *nothing* else and refresh. It'll still work, right? Right? Nope!

The required `class` option is missing! As I just finished saying, we *must* pass a `class` option to the `EntityType`. We got away with this before, because when it's `null`, Symfony guesses the form "type" *and* the `class` option.

Set the option to `SubFamily::class` - and alternate syntax to the normal `AppBundle:SubFamily` :

```
45 lines | src/AppBundle/Form/GenusFormType.php
1  ... lines 1 - 4
5  use AppBundle\Entity\SubFamily;
1  ... lines 6 - 12
13 class GenusFormType extends AbstractType
14 {
15     public function buildForm(FormBuilderInterface $builder, array $options)
16     {
17         $builder
18
19         ->add('subFamily', EntityType::class, [
20
21         ... line 20
22         'class' => SubFamily::class,
23
24         ... lines 22 - 24
25     ])
26
27     ... lines 26 - 34
35     ;
36 }
37
38 ... lines 37 - 43
44 }
```

The query_builder Option

Now that our form is put back together, I have a second challenge: make the select order alphabetically. In other words, I want

to customize the query that's made for the SubFamily's and add an ORDER BY.

Head back to the `EntityType` docs. One option jumps out at me: `query_builder`. Click to check it out. OK, it says:

If specified, this is used to query the subset of options that should be used for the field.

And actually, I need to search for `query_builder`: I know there's a better example on this page. Here it is!

So, if you pass a `query_builder` option and set it to an anonymous function, Doctrine will pass that the *entity repository* for this specific entity. All we need to do is create whatever query builder we want and return it.

In the form, add `query_builder` and enjoy that auto-completion. Set this to a function with a `$repo` argument:

```
45 lines | src/AppBundle/Form/GenusFormType.php
13 class GenusFormType extends AbstractType
14 {
15     public function buildForm(FormBuilderInterface $builder, array $options)
16     {
17         $builder
18             ->add('subFamily', EntityType::class, [
19             'query_builder' => function(SubFamilyRepository $repo) {
20             }
21         ])
22     }
23 }
24 ;
25 }
26 ... lines 26 - 34
27 }
28 ... lines 28 - 43
29 }
```

Now, I like to keep all of my queries inside of repository classes because I don't want queries laying around in random places, like in a form class. But, if you look, I don't have a `SubFamilyRepository` yet.

No worries - copy the `GenusRepository`, paste it as `SubFamilyRepository`. Rename that class and clear it out:

```
16 lines | src/AppBundle/Repository/SubFamilyRepository.php
3 namespace AppBundle\Repository;
6 use Doctrine\ORM\EntityRepository;
7
8 class SubFamilyRepository extends EntityRepository
9 {
10 ... lines 10 - 14
11 }
```

Open the `SubFamily` entity and hook it up with `@ORM\Entity(repositoryClass="AppBundle\Repository\SubFamilyRepository")` and fill in `SubFamilyRepository`:

```
45 lines | src/AppBundle/Entity/SubFamily.php
7 /**
8  * @ORM\Entity(repositoryClass="AppBundle\Repository\SubFamilyRepository")
9  */
10
11 class SubFamily
12 ... lines 12 - 45
```

Great! Back in our form, we know this repo will be an instance of `SubFamilyRepository`:

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

```
1 ... lines 1 - 5
6 use AppBundle\Repository\SubFamilyRepository;
1 ... lines 7 - 12
13 class GenusFormType extends AbstractType
14 {
15     public function buildForm(FormBuilderInterface $builder, array $options)
16     {
17         $builder
18
19         ->add('subFamily', EntityType::class, [
20
21         'query_builder' => function(SubFamilyRepository $repo) {
22
23
24         }
25     ])
26
27
28
29
30
31
32
33
34
35 ;
36 }
37
38
39
40
41
42
43
44 }
```

Return `$repo->` and a new method that we're about to create called `createAlphabeticalQueryBuilder()` :

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

```
1 ... lines 1 - 12
13 class GenusFormType extends AbstractType
14 {
15     public function buildForm(FormBuilderInterface $builder, array $options)
16     {
17         $builder
18
19         ->add('subFamily', EntityType::class, [
20
21         'query_builder' => function(SubFamilyRepository $repo) {
22             return $repo->createAlphabeticalQueryBuilder();
23         }
24     ])
25
26
27
28
29
30
31
32
33
34
35 ;
36 }
37
38
39
40
41
42
43
44 }
```

Copy that name and head into the repository to create that function. Inside, return `$this->createQueryBuilder('sub_family')` and then order by `sub_family.name`, `ASC` :

16 lines | src/AppBundle/Repository/SubFamilyRepository.php

```
1 ... lines 1 - 7
8 class SubFamilyRepository extends EntityRepository
9 {
10     public function createAlphabeticalQueryBuilder()
11     {
12         return $this->createQueryBuilder('sub_family')
13             ->orderBy('sub_family.name', 'ASC');
14     }
15 }
```

Done! The `query_builder` method points here, and we handle the query.

Alright, try it out! Nailed it! As far as form options go, we probably just conquered one of the most complex.

Chapter 8: Creating a Date Picker Field

What's the ugliest part of the form? Yeah, we all know. It's this crazy 3 drop-downs used for the date field.

In modern times, if we need a date field, we're going to render it as a text field and use a fancy JavaScript widget to help the user fill it in.

Head back to the list of fields. It doesn't take long to find the one that's being guessed for the "First Discovered At" field: it's `DateTime`.

Let's see if there's a way to render this as a text field instead.

Setting widget to `single_text`

Check out the `widget` option:

The basic way in which this field should be rendered.

It can be either choice - the three select fields, which is lame - 3 text fields, or *a* single text field with `single_text`. Ah hah!

Back in the form, let's pass in `DateTime::class` even though we could be lazy and pass `null`. Create the array for the third argument and add `widget` set to `single_text`:

```
48 lines | src/AppBundle/Form/GenusFormType.php
10 use Symfony\Component\Form\Extension\Core\Type\DateTime;
14 class GenusFormType extends AbstractType
15 {
16     public function buildForm(FormBuilderInterface $builder, array $options)
17     {
18         $builder
19             ... lines 19 - 34
20             ... lines 19 - 34
35         ->add('firstDiscoveredAt', DateTime::class, [
36             'widget' => 'single_text'
37         ])
38     ;
39 }
47 }
```

Check it out.

HTML5 Date Types are Cool(ish)

Boom! It looks great. In fact, check this out: *italready* has some widget coolness: with a drop-down and a nice calendar.

💡 Tip

If you don't see a fancy widget, don't worry! Not all browsers support this, which is why we'll use a true JavaScript widget soon!

This is *not* coming from Symfony: it's coming from my browser. Because as soon as we made this a `single_text` widget, Symfony rendered it as an `<input type="date">` HTML5 field. Most browsers see this and add their own little date widget functionality.

Ready for the lame news? Not all browsers do this. And that means that users *without* this feature will have a pretty tough time trying to figure out what date format to pass.

Adding a JavaScript Date Picker

Instead, let's add a proper JavaScript widget. Google for "Bootstrap Date Picker". Ok, this first result looks pretty awesome - let's go for it!

First, we need to import new CSS and JS files. In `new.html.twig`, override the block `stylesheets` from the base layout. Add `{% endblock %}` and print `{{ parent() }}`:

```
35 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 2
3 {% block stylesheets %}
4     {{ parent() }}
1 ... lines 5 - 6
7 {% endblock %}
1 ... lines 8 - 35
```

Because I'm lazy, I'll paste the URL to a CDN that hosts this CSS file:

```
35 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 2
3 {% block stylesheets %}
4     {{ parent() }}
5
6     <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.6.0/css/bootstrap-datepicker.css">
7 {% endblock %}
1 ... lines 8 - 35
```

But, you can download it if you want.

Do the same thing for `block javascripts`. Add `{% endblock %}` and call `parent()`:

```
35 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 8
9 {% block javascripts %}
10     {{ parent() }}
1 ... lines 11 - 18
19 {% endblock %}
1 ... lines 20 - 35
```

I've got a CDN URL ready for the JavaScript file too. Go me!

```
35 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 8
9 {% block javascripts %}
10     {{ parent() }}
11
12     <script src="//cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.6.0/js/bootstrap-datepicker.min.js"></script>
1 ... lines 13 - 18
19 {% endblock %}
1 ... lines 20 - 35
```

Adding a class Attribute

Next, how do we activate the plugin? According to their docs: it's pretty simple: select the input and call `.datepicker()` on it.

Personally, whenever I want to target an element in JavaScript, I give that element a class that starts with `js-` and use that with jQuery.

So the question is, how do we give this text field a class? You can't!

I mean you can! In 2 different ways! The first is by passing another option to the field in the form class. Add an `attr` option to an array. And give that array a `class` key set to `js-datepicker`:

```
50 lines | src/AppBundle/Form/GenusFormType.php
1 ... lines 1 - 10
11 ... lines 11 - 20
12 ... lines 21 - 30
13 ... lines 31 - 40
14 ... lines 41 - 50
15 ... lines 51 - 60
16 ... lines 61 - 70
17 ... lines 71 - 80
18 ... lines 81 - 90
19 ... lines 91 - 100
20 ... lines 101 - 110
21 ... lines 111 - 120
22 ... lines 121 - 130
23 ... lines 131 - 140
24 ... lines 141 - 150
25 ... lines 151 - 160
26 ... lines 161 - 170
27 ... lines 171 - 180
28 ... lines 181 - 190
29 ... lines 191 - 200
30 ... lines 201 - 210
31 ... lines 211 - 220
32 ... lines 221 - 230
33 ... lines 231 - 240
34 ... lines 241 - 250
35 ... lines 251 - 260
36 ... lines 261 - 270
37 ... lines 271 - 280
38 ... lines 281 - 290
39 ... lines 291 - 300
40 ... lines 301 - 310
41 ... lines 311 - 320
42 ... lines 321 - 330
43 ... lines 331 - 340
44 ... lines 341 - 350
45 ... lines 351 - 360
46 ... lines 361 - 370
47 ... lines 371 - 380
48 ... lines 381 - 390
49 ... lines 391 - 400
50 ... lines 401 - 410
```

```

14 class GenusFormType extends AbstractType
15 {
16     public function buildForm(FormBuilderInterface $builder, array $options)
17     {
18         $builder
19             ... lines 19 - 34
20             ... line 36
21             ... line 38
22             ... lines 42 - 48
23     }

```

Setting up the JavaScript

Next, in our template, add the `jQuery(document.ready)` block. Hook it up with `$('.js-datepicker').datepicker()` :

```

35 lines | app/Resources/views/admin/genus/new.html.twig
9 {% block javascripts %}
10     {{ parent() }}
11
12     <script src="//cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.6.0/js/bootstrap-datepicker.min.js"></script>
13
14     <script>
15         jQuery(document).ready(function() {
16             $('.js-datepicker').datepicker();
17         });
18     </script>
19 {% endblock %}
20 ... lines 20 - 35

```

Easy. Give it a try.

Scroll down and... hey! There it is! I can see the cool widget. And if I click... um... if I click... then - why is nothing happening?

HTML5 versus DatePicker: Fight!

It turns out, the HTML5 date functionality from my browser is *fighting* with the date picker. Silly kids. This doesn't happen in all browsers, it's actually something special to Chrome.

To fix this, we need to turn *off* the HTML5 date functionality. In other words, we want render this as a true `<input type="text">` field, *not* a `date` field.

To do that, open the form type. There's one last option that will help us: set `html5` to `false` :

```

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

```

```

14 class GenusFormType extends AbstractType
15 {
16     public function buildForm(FormBuilderInterface $builder, array $options)
17     {
18         $builder
19             ... lines 19 - 34
20             ->add('firstDiscoveredAt', DateType::class, [
21                 ... lines 36 - 37
22                 'html5' => false,
23             ])
24         ;
25     }
26     ... lines 42 - 48
27 }

```

Try it one last time. HTML5 is out of the way and the date picker is in charge.

Pretty awesome.

Chapter 9: Date Format & "Sanity" Validation

Let's use the fancy new date picker. Fill in the other fields and hit submit.

Whoa! Validation Error!

First, the good news: the error looks nice! And we didn't do any work for that.

Now, the bad news: do you remember adding any validation? Because I don't! So, where is that coming from?

The Format of the Date Field

What you're seeing is a really amazing automatic thing that Symfony does. And I invented a term for it: "sanity validation".

Remember, even though this has a fancy date picker widget, this is *just* a text field, and it needs to be filled in with a very specific format. In this case, the widget fills in month/day/year.

But what if Symfony expects a different format - like year/month/day?

Well, that's *exactly* what's happening: Symfony expects the date in one format, but the widget is generating something else. When Symfony fails to parse the date, it adds the validation error.

Sanity Validation

So, it turns out that many fields have sanity validation. It basically makes sure that a *sane* value is submitted by the user. If an *insane* value is sent, it blocks the way and shows a message.

For example, the `speciesCount` is using Symfony's `NumberType`, which renders as an HTML5 field with up and down arrows on some browsers:

```
139 lines | src/AppBundle/Entity/Genus.php
12 class Genus
13 {
14     ... lines 14 - 31
32     /**
33      * @ORM\Column(type="integer")
34      */
35     private $speciesCount;
36     ... lines 36 - 137
138 }
```

If we tried to type a word here and submit, the `NumberType` would throw a validation error thanks to sanity validation.

Our drop-down fields *also* have sanity validation. If a jerk user tries to hack the system by adding an option that we *did not* originally include, the field will fail validation. 99% of the time, you don't know or care that this is happening. Just know, Symfony has your back.

Making the Date Formats Match

But how do we fix the date field? We just need to make Symfony's expected format match the format used by the datepicker. In fact, the `format` itself is an option on the `DateType`. I'll hold `command` and click into `DateType`. When you use the `single_text` widget, it expects this `HTML5_FORMAT`: so year-month-day.

Let's update the JavaScript widget to match this.

How? On its docs, you'll see that it *also* has a `format` option. Cool!

Now, unfortunately, the format string used by the `DateType` and the format string used by the datepicker widget are not *exactly* the same format - each has its own system, unfortunately. So, you may need to do some digging or trial and error. It turns out, the correct format is `yyyy-mm-dd`:

```
37 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 8
9 {% block javascripts %}
1 ... lines 10 - 13
14 <script>
15     jQuery(document).ready(function() {
16         $('#js-datepicker').datepicker({
17             format: 'yyyy-mm-dd'
18         });
19     });
20 </script>
21 {% endblock %}
1 ... lines 22 - 37
```

OK, go back and refresh that page. Fill out the top fields... and then select a date. Moment of truth. Got it!

Data Transformers

So I keep telling you that the purpose of the field "types" is to control how a field is rendered. But that's only *half* of it. Behind the scenes, many fields have a "data transformer".

Basically, the job of a data transformer is to transform the data that's inside of your PHP code to a format that's visible to your user. For example, the `firstDiscoveredAt` value on `Genus` is actually a `DateTime` object:

```
139 lines | src/AppBundle/Entity/Genus.php
1 ... lines 1 - 11
12 class Genus
13 {
1 ... lines 14 - 46
47 /**
48  * @ORM\Column(type="date")
49  */
50 private $firstDiscoveredAt;
1 ... lines 51 - 137
138 }
```

The data transformer internally changes that into the string that's printed in the box.

Then, when a date string is submitted, that same data transformer does its work in reverse: changing the string back into a `DateTime` object.

The data transformer is also kicking butt on the `subFamily` field. The `id` of the selected `SubFamily` is submitted. Then, the data transformer uses that to query for a `SubFamily` object and set that on `Genus`.

You don't need to know more about data transformers right now, I just want you to realize that this awesomeness is happening.

Chapter 10: Form Rendering and Form Variables

Let's talk form rendering.

Sure, things looks nice right now, especially considering we're rendering all the fields in one line!

The problem? First, we can't change the order of the fields. And second, we won't be able to control the labels or anything else that we're going to talk about.

Using form_row

So, in reality, I don't use `form_widget` to render all my fields at once. Replace this with a different function `form_row` and pass it `genusForm.` and then the name of one of the fields - like `name` :

```
43 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 22
23 {% block body %}
24     <div class="container">
25         <div class="row">
26             <div class="col-xs-12">
27 ... lines 27 - 28
29                 {{ form_start(genusForm) }}
30                 {{ form_row(genusForm.name) }}
31 ... lines 31 - 37
38                 {{ form_end(genusForm) }}
39             </div>
40         </div>
41     </div>
42 {% endblock %}
```

Since this form has 1, 2, 3, 4, 5, 6 fields, I'll copy that and paste it six times. Now, fill in all the field names `subFamily` , `speciesCount` , `funFact` , `isPublished` and `firstDiscoveredAt` :

```
43 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 22
23 {% block body %}
24     <div class="container">
25         <div class="row">
26             <div class="col-xs-12">
27 ... lines 27 - 28
29                 {{ form_start(genusForm) }}
30                 {{ form_row(genusForm.name) }}
31                 {{ form_row(genusForm.subFamily) }}
32                 {{ form_row(genusForm.speciesCount) }}
33                 {{ form_row(genusForm.funFact) }}
34                 {{ form_row(genusForm.isPublished) }}
35                 {{ form_row(genusForm.firstDiscoveredAt) }}
36 ... lines 36 - 37
38                 {{ form_end(genusForm) }}
39             </div>
40         </div>
41     </div>
42 {% endblock %}
43
```

Refresh! OMG - it's the *exact* same thing as before!!! This is what `form_widget` was doing behind the scenes: looping over the fields and calling `form_row`.

All the Form Rendering Functions

So, at this point, you're probably asking:

What else can I do? What other functions are there? What options can I pass to these functions?

Look, I don't know! But I bet Google does: search for "form functions twig" to find another reference section called [Form Function and Variable Reference](#).

Ah hah! This is our cheatsheet!

First - we already know about `form_row`: it renders the 3 parts of a field, which are the label, the HTML widget element itself and any validation errors.

If you ever need more control, you can render those individually with `form_widget`, `form_label` and `form_errors`. Use these *instead* of `form_row`, *only* when you need to.

Form Variables

Now notice, most of these functions - including `form_row` - have a second argument called "variables". And judging by this code example, you can apparently control the `label` with this argument.

Listen closely: these "variables" are the most *powerful* part of form rendering. By passing different values, you can override almost *every* part of how a field is rendered.

So what *can* you pass here? Scroll down near the bottom to find a big beautiful table called [Form Variables Reference](#).

This gives you a big list of all the variables that you can override when rendering a field, including `label`, `attr`, `label_attr`, and other stuff.

To show this off, find the `speciesCount` field and add a second argument set to `{}` - the Twig array syntax. Override the `label` variable: set it to `Number of Species`:

```
45 lines | app/Resources/views/admin/genus/new.html.twig
↑ ... lines 1 - 22
23 {% block body %}
24     <div class="container">
25         <div class="row">
26             <div class="col-xs-12">
↑ ... lines 27 - 28
29                 {{ form_start(genusForm) }}
↑ ... lines 30 - 31
32                 {{ form_row(genusForm.speciesCount, {
33                     'label': 'Number of Species'
34                 }) }}
↑ ... lines 35 - 39
40                 {{ form_end(genusForm) }}
41             </div>
42         </div>
43     </div>
44 {% endblock %}
```

Refresh! It's just that easy.

The Amazing Form Profiler

There's one *huge* form tool that we haven't looked at yet. Your web debug toolbar *should* have a clipboard icon. Click it.

This is the profiler for your form, and it's *packed* with stuff that's going to make your form life better. If you click `speciesCount`, you can see the different data for your species - which isn't too interesting on this blank form. You can see any submitted data and all of the variables that can be passed to the field.

It turns out, the reference section we looked at has *most* of the variables but *this* will have *all* of them. This is your place to answer:

What are the values for my variables? And, what can I override?

This also shows you "Resolved Options": these are the final values of the options that can be passed as the third argument to the `add()` function.

CSRF Protection

Oh, and check out this `_token` field. Do you remember adding this?

Hopefully not - because we never did! This is a CSRF token that's automatically added to the form. It's rendered ~~for~~ us when we call `form_end` and validated behind the scenes along with all the other fields. It's free CSRF protection.

Chapter 11: Disable HTML5 Validation

Leave everything blank and hit save.

Oh, it doesn't submit. Instead we get this validation error. So where's that coming from?

The Famous required Attribute

Hint: it's not Symfony!. It's our friend HTML5. When Symfony renders the field, it's adding a `required="required"` attribute, and this activates HTML5 validation.

But, there are a few problems with this. First, Symfony always adds the `required` attribute... even if it's not actually required in the database. It's a borderline bug in Symfony.

And actually, that's not totally fair. If you use field-type-guessing, Symfony *will* guess whether or not it should render this by looking at your database and validation config. But as soon as you set your field type, it stops doing that. Boo!

Here's the second problem: even if we like this HTML5 client-side validation, we still need to add true server-side validation. Otherwise, nasty users can go crazy on our site.

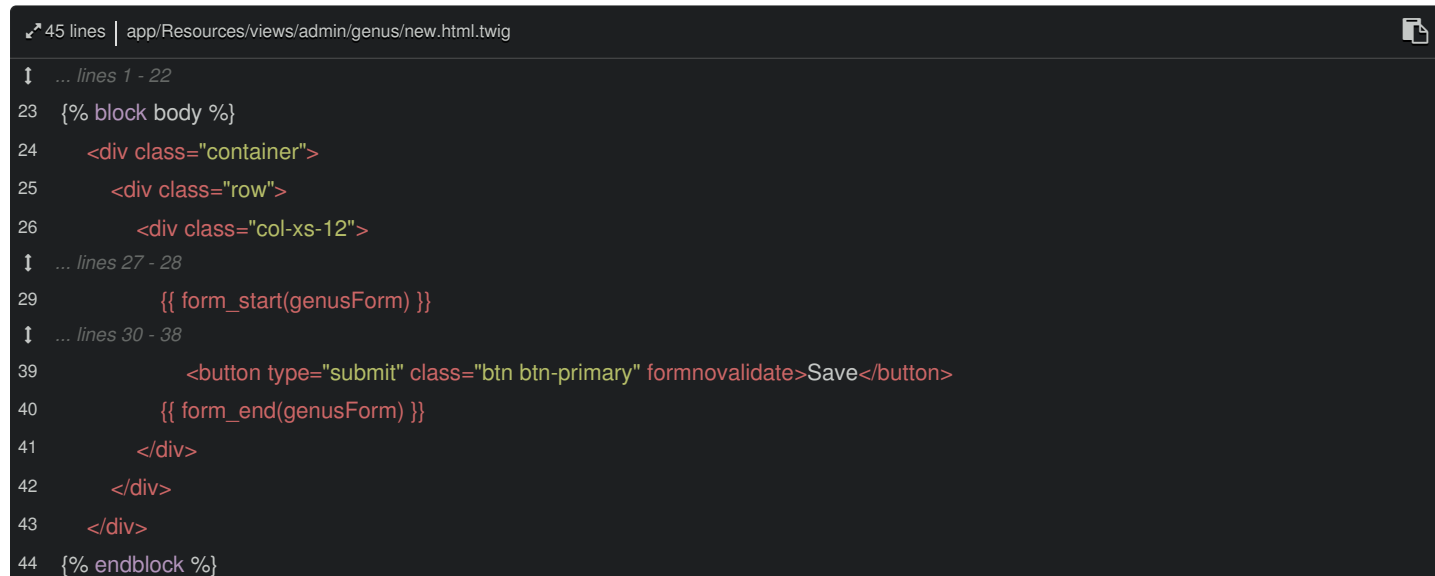
Disable HTML5 Validation

So here's what I do: I disable HTML5 validation and rely purely on server-side validation.

If you *do* want some fancy client-side validation, I recommend adding it with a JavaScript library. These give you more features and control than HTML5 validation.

There's even a bundle - [JsFormValidatorBundle](#) - that can dump your server-side rules into JavaScript.

So how do we disable HTML5 validation? Very simple: find the submit button and add `formnovalidate` :



```
45 lines | app/Resources/views/admin/genus/new.html.twig
1 ... lines 1 - 22
23 {% block body %}
24     <div class="container">
25         <div class="row">
26             <div class="col-xs-12">
27 ... lines 27 - 28
29         {{ form_start(genusForm) }}
30 ... lines 30 - 38
39             <button type="submit" class="btn btn-primary" formnovalidate>Save</button>
40         {{ form_end(genusForm) }}
41     </div>
42 </div>
43 </div>
44 {% endblock %}
```

That's it. Refresh the page now and submit. No more HTML5 validation. But of course now, we need server-side validation!

Chapter 12: Beautiful Form Validation

Guess what! Server-side validation is really, really fun. Google for Symfony validation, and find the book chapter.

There is *one* weird thing about validation... which I love. Here it is: you don't apply validation to your form. Nope, there will be *no* validation code inside of `GenusFormType`. Instead, you add validation to the *class* that is bound to your form. When the form is submitted, it automatically reads those validation rules and uses them.

Validation is added with annotations. So copy the `use` statement from the code block, find `Genus` and paste it on top:

```
141 lines | src/AppBundle/Entity/Genus.php
1 ... lines 1 - 6
7 use Symfony\Component\Validator\Constraints as Assert;
1 ... lines 8 - 141
```

The Giant List of Constraints

Good start! Next, we'll add validation rules - called constraints - above each property. On the left side bar, find the "Constraints" link.

Check out this menu of validation rules: `NotBlank`, `NotNull`, `Email`, `Length`, `Regex` ... so many things! Pretty much anything you can dream up is inside of this list.

Let's start with an easy one: above the `name` property, add `@Assert\NotBlank`:

```
141 lines | src/AppBundle/Entity/Genus.php
1 ... lines 1 - 12
13 class Genus
14 {
1 ... lines 15 - 21
22 /**
23  * @Assert\NotBlank()
1 ... line 24
25 */
26 private $name;
1 ... lines 27 - 139
140 }
```

Without doing anything else, refresh. Boom! Validation error. And, it looks nice.

Let's add some more. For `subFamily` - that should be required, so add `@NotBlank`:

```
145 lines | src/AppBundle/Entity/Genus.php
1 ... lines 1 - 12
13 class Genus
14 {
1 ... lines 15 - 27
28 /**
29  * @Assert\NotBlank()
1 ... lines 30 - 31
32 */
33 private $subFamily;
1 ... lines 34 - 143
144 }
```

For `speciesCount`, add `@NotBlank` again:

```
145 lines | src/AppBundle/Entity/Genus.php
```

```

13 class Genus
14 {
15     ... lines 15 - 34
35     /**
36      * @Assert\NotBlank()
37     ... lines 37 - 38
39     */
40     private $speciesCount;
41     ... lines 41 - 143
144 }

```

But in addition to that, we want `speciesCount` to be a positive number: we don't want some funny biologist entering negative 10.

Constraint Options

On the constraints list, there's one called `Range`. Check that out.

Ok cool: just like the form field types, you can pass options to the constraints. The `Range` constraint has several: `min`, `max`, `minMessage` and `maxMessage`. Add `@Assert\Range` with `min=0` and `minMessage="Negative species! Come on..."`:

```

145 lines | src/AppBundle/Entity/Genus.php
13 class Genus
14 {
15     ... lines 15 - 34
35     /**
36      * @Assert\NotBlank()
37      * @Assert\Range(min=0, minMessage="Negative species! Come on...")
38     ... line 38
39     */
40     private $speciesCount;
41     ... lines 41 - 143
144 }

```

Ok, let's finish up. It's ok if `funFact` is empty - so don't add anything there. The same is true for `isPublished`: we *could* add a constraint to make sure this is a boolean, but the sanity validation on the form already takes care of that.

Finally, let's make sure `firstDiscoveredAt` is also `NotBlank`:

```

145 lines | src/AppBundle/Entity/Genus.php
13 class Genus
14 {
15     ... lines 15 - 51
52     /**
53      * @Assert\NotBlank()
54     ... line 54
55     */
56     private $firstDiscoveredAt;
57     ... lines 57 - 143
144 }

```

Ok, refresh! Leave everything blank and put -10 for the number of species. I love it!

Chapter 13: Changing Validation Messages

Check out the web debug toolbar: it's highlighted with the number of validation errors, which is pretty cool. It's even cooler because we can see where those are coming from.

The default message for the `NotBlank` constraint is obviously

```
This value should not be blank
```

We can easily change this by passing a `message` option to the annotation. But wait a second: we're going to use `NotBlank` a lot. Is there a way to customize the default message across the whole system?

Yea! All of these strings are passed through Symfony's translator. And we can take advantage of that to customize this message, even in English.

Enabling the Translator

First, in case you don't already have it enabled, open up `app/config/config.yml`. Activate the `translator` service by uncommenting out the `translator` key under `framework`:

```
74 lines | app/config/config.yml
12 framework:
13     translator: { fallbacks: ["%locale%"] }
```

Refresh this page and watch the web debug toolbar. Suddenly, there's an extra icon that's coming from the translator. It's reporting that there are ten missing messages. In other words, we are apparently already sending ten messages through the translator that are missing translation strings.

This is because every form field and all validation errors are automatically sent through the translator. Nothing looks weird, because those strings are already English, so it's not really a problem that they aren't translated.

Changing Validation Messages

But, if you want to customize this message, copy it. And, notice, the domain is `validators`: that's basically a translation "category", and it's important for what we do next.

We don't have any translation files yet, so create a new directory called `translations` in `app/Resources`. Inside, add a new file: `validators.en.yml`. This is `validators` because the message is being translated in that domain.

Inside, paste the string and set it to "Hi! Please enter something for this field.":

```
1 lines | app/Resources/translations/validators.en.yml
1 "This value should not be blank.": Hi! Please enter *something* for this field :)
```

And that's it! Go back and refresh! Oh no, it didn't work! Well, I kind of expected that. Find your terminal, open a new tab, and run:

```
./bin/console cache:clear
```

You almost *never* need to worry about clearing your cache while developing but very occasionally, you'll find a quirk in Symfony that needs this. Sometimes, when you add a new translation file, this happens.

Let's refresh again. There it is!

So yay validation! There's one more constraint I want you to see: `Callback`. This is your Swiss army knife: it allows you to write whatever custom validation logic you want inside a method. There, you can create different validation errors and map them to any field.

Chapter 14: Easy Edit Form

When you get to the form, it's completely blank. How could we add *default* data to the form?

Well, it turns out answering that question is exactly the same as answering the question

How do we create an edit form?

Let's tackle that.

In `GenusAdminController`, I'm going to be lazy: copy the entire `newAction()` and update the URL to `/genus/{id}/edit`. Give it a different route name: `admin_genus_edit` and call it `editAction()`:

```
81 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 13
14 class GenusAdminController extends Controller
15 {
16  ... lines 16 - 55
56  /**
57   * @Route("/genus/{id}/edit", name="admin_genus_edit")
58   */
59  public function editAction(Request $request, Genus $genus)
60  {
61  ... lines 61 - 79
80  }
81 }
```

Our first job should be to query for a `Genus` object. I'll be lazy again and just type-hint an argument with `Genus`:

```
81 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
1  ... lines 1 - 4
5  use AppBundle\Entity\Genus;
6  ... lines 6 - 8
9  use Symfony\Component\HttpFoundation\Request;
10 ... lines 10 - 13
14 class GenusAdminController extends Controller
15 {
16  ... lines 16 - 58
59  public function editAction(Request $request, Genus $genus)
60  {
61  ... lines 61 - 79
80  }
81 }
```

Thanks to the param converter from `SensioFrameworkExtraBundle`, this will automatically query for `Genus` by using the `{id}` value.

Passing in Default Data

This form needs to be pre-filled with all of the data from the database. So again, how can I pass default data to a form? It's as simple as this: the second argument to `createForm` is the default data. Pass it the entire `$genus` object:

```
81 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
```



```

14 class GenusAdminController extends Controller
15 {
16     ... lines 16 - 58
59     public function editAction(Request $request, Genus $genus)
60     {
61         $form = $this->createForm(GenusFormType::class, $genus);
62         ... lines 62 - 79
80     }
81 }

```

Why the entire object? Because remember: our form is bound to the `Genus` class:

```

50 lines | src/AppBundle/Form/GenusFormType.php
14 class GenusFormType extends AbstractType
15 {
16     ... lines 16 - 42
43     public function configureOptions(OptionsResolver $resolver)
44     {
45         $resolver->setDefaults([
46             'data_class' => 'AppBundle\Entity\Genus'
47         ]);
48     }
49 }

```

That means that its output will be a `Genus` object, but its input should *also* be a `Genus` object.

Behind the scenes, it will use the getter functions on `Genus` to pre-fill the form: like `getName()`. And everything else is exactly the same. Well, I'll tweak the flash message but you get the idea:

```

81 lines | src/AppBundle/Controller/Admin/GenusAdminController.php
14 class GenusAdminController extends Controller
15 {
16     ... lines 16 - 58
59     public function editAction(Request $request, Genus $genus)
60     {
61         ... lines 61 - 64
65         if ($form->isSubmitted() && $form->isValid()) {
66             ... lines 66 - 71
72             $this->addFlash('success', 'Genus updated!');
73             ... lines 73 - 74
75         }
76         ... lines 76 - 79
80     }
81 }

```

Rendering the Edit Form

Update the template to `edit.html.twig`:

```

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

```

```

14 class GenusAdminController extends Controller
15 {
16 ... lines 16 - 58
59 public function editAction(Request $request, Genus $genus)
60 {
61 ... lines 61 - 76
77 return $this->render('admin/genus/edit.html.twig', [
78     'genusForm' => $form->createView()
79 ]);
80 }
81 }

```

I'm still feeling lazy, so I'll completely duplicate the new template and update the `h1` to say "Edit Genus":

45 lines | app/Resources/views/admin/genus/edit.html.twig

```

23 {% block body %}
24     <div class="container">
25         <div class="row">
26             <div class="col-xs-12">
27                 <h1>Edit Genus</h1>
28 ... lines 28 - 40
41             </div>
42         </div>
43     </div>
44 {% endblock %}
45

```

Don't worry, this duplication is temporary.

Finally, in the admin list template, I already have a spot ready for the edit link. Fill that in with `path('admin_genus_edit')` and pass it the single wildcard value: `id: genus.id` :

33 lines | app/Resources/views/admin/genus/list.html.twig

```

3 {% block body %}
4     <div class="container">
5         <div class="row">
6
7             <div class="col-xs-12">
8 ... lines 8 - 13
14         <table class="table table-striped">
15 ... lines 15 - 19
20             {% for genus in genres %}
21                 <tr>
22 ... lines 22 - 23
24                     <td>
25                         <a href="{{ path('admin_genus_edit', {'id': genus.id}) }}" class="btn btn-xs btn-success"><span class="fa fa-pencil"></span>
26                     </td>
27                 </tr>
28             {% endfor %}
29         </table>
30     </div>
31 </div>
32 </div>
33 {% endblock %}

```

LOVE it. Open up `/admin/genus` in your browser.

Ah good, an explosion - I felt like things were going too well today:

```
Method id for object Genus does not exist in list.html.twig at line 25.
```

So apparently I do *not* have a `getId()` function on `Genus`. Let's check it out. And indeed, when I created this class, I did not add a getter for ID. I'll use `command + N`, or the "Code"->"Generate" menu to add it:

```
150 lines | src/AppBundle/Entity/Genus.php
↑ ... lines 1 - 12
13 class Genus
14 {
↑ ... lines 15 - 68
69 public function getId()
70 {
71     return $this->id;
72 }
↑ ... lines 73 - 148
149 }
```

All right. Let's try it again. Refresh. No errors!

Edit the first genus. Check that out: it completely pre-filled the form for us. In fact, check out this weird "TEST" text inside of `funFact`. That is left over from an earlier tutorial. I hacked in this TEST string temporarily in `getFunFact()` so we could play with markdown. This *proves* that the form is using the getter functions to pre-fill things.

So, that's really interesting but let's take it out:

```
150 lines | src/AppBundle/Entity/Genus.php
↑ ... lines 1 - 12
13 class Genus
14 {
↑ ... lines 15 - 106
107 public function getFunFact()
108 {
109     return $this->funFact;
110 }
↑ ... lines 111 - 148
149 }
```

Refresh. Change the "Fun fact" to be even more exciting, hit enter, and there it is:

```
Genus updated - you are amazing!
```

Edit that Genus again: there's the new fun fact. This is *areally* cool thing about the form framework: the new and edit endpoints are *identical*. The only difference is that one is passed default data.

So this is great! Except for the template duplication. That's not great still.

Chapter 15: Sharing Form Templates with include()

Adding edit was quick! But the *entire* template is now duplicated. This includes the code to render the form *and* the other blocks that include the needed CSS and JS files.

First, copy the form rendering code and move that into a new file: `_form.html.twig` :

```
13 lines | app/Resources/views/admin/genus/_form.html.twig
1  {{ form_start(genusForm) }}
2    {{ form_row(genusForm.name) }}
3    {{ form_row(genusForm.subFamily) }}
4    {{ form_row(genusForm.speciesCount, {
5      'label': 'Number of Species'
6    }) }}
7    {{ form_row(genusForm.funFact) }}
8    {{ form_row(genusForm.isPublished) }}
9    {{ form_row(genusForm.firstDiscoveredAt) }}
10
11    <button type="submit" class="btn btn-primary" formnovalidate>Save</button>
12  {{ form_end(genusForm) }}
13
```

Paste it here.

In edit, just include that template: `include('admin/genus/_form.html.twig')` :

```
34 lines | app/Resources/views/admin/genus/edit.html.twig
1  ... lines 1 - 22
23  {% block body %}
24    <div class="container">
25      <div class="row">
26        <div class="col-xs-12">
27          <h1>Edit Genus</h1>
28
29          {{ include('admin/genus/_form.html.twig') }}
30        </div>
31      </div>
32    </div>
33  {% endblock %}
34
```

Copy that, open `new.html.twig` , and paste it there:

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

```

1 ... lines 1 - 22
23 {% block body %}
24     <div class="container">
25         <div class="row">
26             <div class="col-xs-12">
27                 <h1>New Genus</h1>
28
29                 {{ include('admin/genus/_form.html.twig') }}
30             </div>
31         </div>
32     </div>
33 {% endblock %}
34

```

Ok, I'm feeling better. Refresh now: everything still looks good.

And by the way, if there *were* any customizations you needed to make between new and edit, I would pass a variable in through the second argument of the `include` function and use that to control the differences.

Using a Form Layout

So let's fix the last problem: the duplicated block overrides.

To solve this, we'll need a shared layout between these two templates. Create a new file called `formLayout.html.twig`. This will *just* be used by these two templates.

Copy the `extends` code all the way through the `javascripts` block and delete it from `edit.html.twig` :

```

14 lines | app/Resources/views/admin/genus/edit.html.twig
1 ... lines 1 - 2
3 {% block body %}
4     <div class="container">
5         <div class="row">
6             <div class="col-xs-12">
7                 <h1>Edit Genus</h1>
8
9                 {{ include('admin/genus/_form.html.twig') }}
10            </div>
11        </div>
12    </div>
13 {% endblock %}
14

```

Paste it in `formLayout.html.twig` :

```

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

```

```

1 {% extends 'base.html.twig' %}
2
3 {% block stylesheets %}
4     {{ parent() }}
5
6     <link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.6.0/css/bootstrap-datepicker.css">
7 {% endblock %}
8
9 {% block javascripts %}
10    {{ parent() }}
11
12    <script src="//cdnjs.cloudflare.com/ajax/libs/bootstrap-datepicker/1.6.0/js/bootstrap-datepicker.min.js"></script>
13
14    <script>
15        jQuery(document).ready(function() {
16            $('.js-datepicker').datepicker({
17                format: 'yyyy-mm-dd'
18            });
19        });
20    </script>
21 {% endblock %}
22

```

So this template *itself* will extend `base.html.twig`, but not before adding some stylesheets and some JavaScripts. In edit, re-add the `extends` to use this template: `admin/genus/formLayout.html.twig`:

```

14 lines | app/Resources/views/admin/genus/edit.html.twig
1 {% extends 'admin/genus/formLayout.html.twig' %}
... lines 2 - 14

```

Copy that, open `new.html.twig` and repeat: delete the `javascripts` and `stylesheets` and paste in the new extends:

```

14 lines | app/Resources/views/admin/genus/new.html.twig
1 {% extends 'admin/genus/formLayout.html.twig' %}
2
3 {% block body %}
4     <div class="container">
5         <div class="row">
6             <div class="col-xs-12">
7                 <h1>New Genus</h1>
8
9                 {{ include('admin/genus/_form.html.twig') }}
10            </div>
11        </div>
12    </div>
13 {% endblock %}
14

```

Try it! Cool! We're using our Twig tools to get rid of duplication!

A Word of Caution

Congrats team - that's it for our first form episode. You should feel dangerous. Most of the time, forms are easy, and amazing! They do *a lot* of work for you.

Let me give you one last word of warning: because this is how I see people get into trouble.

Right now, our form is bound to our entity and that makes this form super easy to use. But eventually, you'll need to build a form that does *not* look exactly like your entity: perhaps it has a few extra fields or is a combination of fields from several entities.

When you run into this: here's what I want you to do. Don't bind your form to your entity class. Instead, create a brand new class: I usually put these classes inside my `Form` directory. For example, `GenusModel`. This class will have the exact properties that your form needs.

Bind *this* class to your form and add all your validation rules like normal. After you submit your form `$form->getData()` will return this *other* object. Then, it'll be your job to write a little bit of extra code that reads this data, updates your entities - or whatever else you need that data for - and saves things.

If you have questions, let me know in the comments.

There's certainly more to learn, but don't wait! Get out there and build something crazy cool!

Seeya guys next time!

