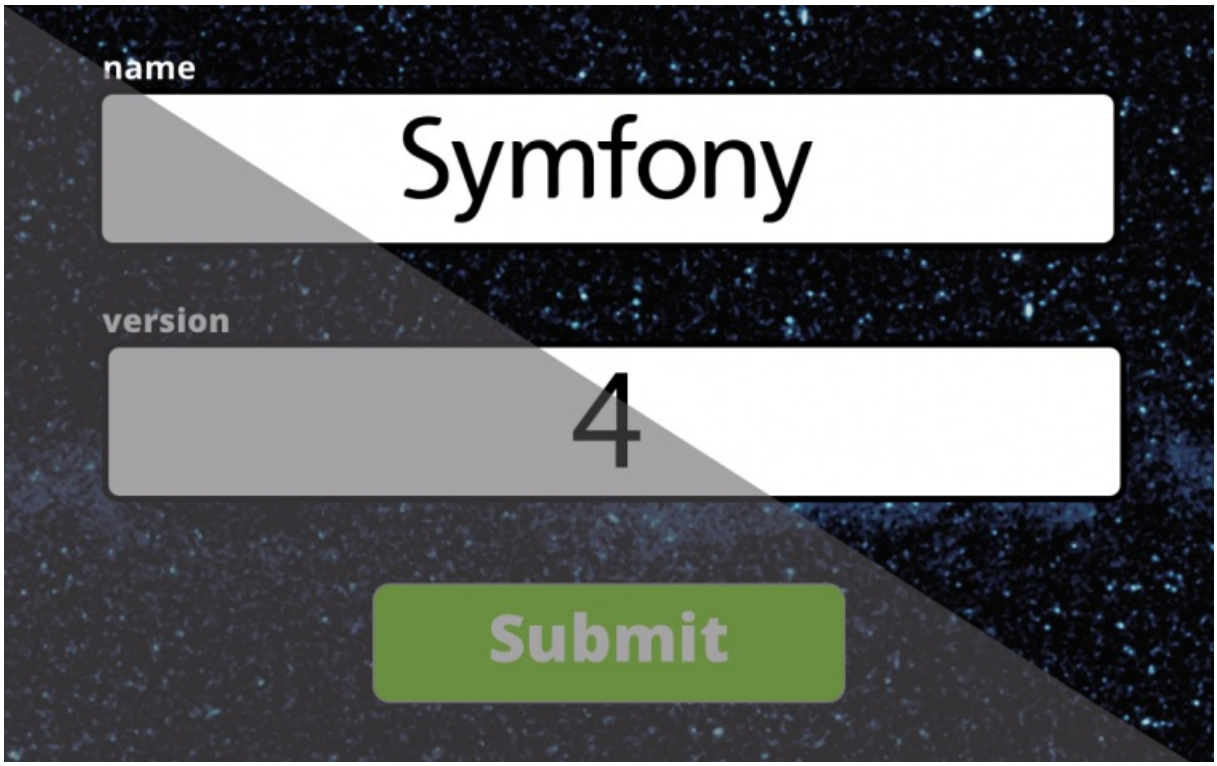
Symfony 4 Forms: Build, Render & Conquer!



With <3 from SymfonyCasts

Chapter 1: Form Type Class

Hey friends! And welcome to, what *I* think will be, a *super* fun tutorial: the one about cookies! Um, forms!

The *first* question you *might* ask is: forms? Do we even *need* forms anymore in this age of JavaScript frontends? Aren't forms *so* 2016? The answer is... it depends. I get to talk to a lot of developers and, honestly, it comes down to what you're building. Yea, some apps *are* using modern JavaScript frontends. But just as many are building form-rich interfaces. So, if that's you - hi! o/.

There is *no* more powerful form system on the planet than Symfony's Form component. Oh, and there are *so* many pieces to a form: rendering the form, handling the submit, validating data, normalizing data, and other things that you don't even *think* about, like CSRF protection. Here's the truth about Symfony's Form component: yes, it *is* crazy powerful. And when you learn to harness that power, you will be incredibly productive. At the same time, in some situations, the form system can be *really* hard & complex. It can make your job *harder* than if you didn't use it at all!

So here is our big goal: to learn how to do almost *everything* you can think of with a form *and* to identify those complex scenarios, and find the simplest path through them. After all, *even* if you use and love the form system, it doesn't mean that you *have* to use it in *every* single situation.

Project Setup

As always, to become the *master* of form tags, inputs & textareas, you should totally code along with me. Download the course code from this page. When you unzip it, you'll find a start/ directory inside with the same files that you see here. Open up the README.md file for instructions on how to get the site set up. The last step will be to find a terminal, move into the project, sip some coffee, and run:

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

to start the built-in web server. Woo! Now, find your browser and head to http://localhost:8000. Welcome to our work-in-progress masterpiece: The Space Bar! Our intergalactic news site where aliens *everywhere* can quickly catch up on only the *most* important news... after a 500 year nap in cryosleep.

Thanks to our last tutorial, we can even log in! Use admin2@thespacebar.com, password engage. Then head over to /admin/article/new to see.... oh! A big TODO!

Yep! We can display articles but... we can't actually create or edit them yet. The code behind this lives in src/Controller/ArticleAdminController.php and, sure enough, *past* us got lazy and just left a TODO.

Creating a Form Class

Time to get to work! The first step to building a form is always to create a form *class*. Inside src, add a new Form/ directory... though, like normal, you can put this stuff *wherever* you want. Inside, a new PHP class called ArticleFormType. Form classes are usually called form "types", and the only rule is that they must extend a class called AbstractType. Oh! But of course! I can't find that class because... *we* haven't installed the form system yet! No problem!

Find your terminal, open a new tab, have *another* well-deserved sip of coffee, and run:

```
$ composer require form
```

Perfect! Back in our editor, once PhpStorm finishes indexing, we *should* be able to find the AbstractType class from the Form component.

Got it! Now, go to the Code -> generate menu, or Cmd+N on a Mac, and click override methods. There are several methods that you can override to control different parts of your form. But, by *far*, the most important is buildForm().

14 lines | src/Form/ArticleFormType.php

```
... lines 1 - 4
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class ArticleFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {

    }
}
```

Inside this method, our job is pretty simple: use this $builder object to, um... build the form! Use $builder->add() to add two fields right now: title and content. These are the two most important fields inside the Article entity class.

18 lines | src/Form/ArticleFormType.php

```
... lines 1 - 9
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title')
            ->add('content')
        ;
    }
... lines 17 - 18
```

And... that's it! We'll do more work here later, but this is enough.

Creating the Form Object

Next, find your controller so we can render the form. Start by saying $form = and using a shortcut: $this->createForm(). Pass the *class* that you want to create: ArticleFormType::class. I'll delete the return response stuff and, instead, render a template with return $this->render('article_admin/new.html.twig'). To render the form, we need to pass that in. Let's call the variable articleForm and set it to - this is tricky - $form->createView(). Yep: don't pass the $form object directly to Twig: always call createView(). This transforms the Form object into another object that is *super* good at rendering forms and telling funny stories at parties.

37 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 12
class ArticleAdminController extends AbstractController
{
... lines 15 - 18
    public function new(EntityManagerInterface $em)
    {
        $form = $this->createForm(ArticleFormType::class);

        return $this->render('article_admin/new.html.twig', [
            'articleForm' => $form->createView()
        ]);
    }
... lines 27 - 35
}
```

Rendering the Form

To create the template, I'll cheat! Ha! Thanks to the Symfony plugin, I can put my cursor on the template name, hit alt+enter,

click "Create Twig Template" and hit enter again to confirm the location. There's no real magic here: that just created the file for us at templates/article_admin/new.html.twig.

Oh, and you *might* remember from previous tutorials that, in addition to the normal base.html.twig, we also have a content_base.html.twig, which gives us a *little* bit of real markup and a content_body block that we can override. Let's use that: {% extends 'content_base.html.twig %} and then, override the block content_body, with {% endblock %}. Add an <h1>Launch a new Article</h1> with, of course, a rocket emoji! Zoom!

12 lines | templates/article_admin/new.html.twig

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

{% block content_body %}
    <h1>Launch a new Article! ?</h1>
... lines 5 - 10
{% endblock %}
```

To render the form, we get to use a few special form *rendering* functions: {{ form_start() }} and pass that the articleForm variable. At the end {{ form_end(articleForm }}. And in the middle, {{ form_widget(articleForm) }}. Oh, and for the submit button, you *can* build this into your form class, but I prefer to add it manually: <button type="submit">, some classes: btn btn-primary, and then Create!

12 lines | templates/article_admin/new.html.twig

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

{% block content_body %}
    <h1>Launch a new Article! ?</h1>

    {{ form_start(articleForm) }}
        {{ form_widget(articleForm) }}

        <button type="submit" class="btn btn-primary">Create!</button>
    {{ form_end(articleForm) }}
{% endblock %}
```

And... we're done! We create a form class, create a Form object from that in the controller, pass the form to Twig, then render it. We'll learn a *lot* more about these rendering functions. But, more or less, form_start() renders the opening form tag, form_end() renders the form *closing* tag... plus a little extra magic, and form_widget() renders *all* of the fields.

Try it! Find your browser and refresh! Woohoo! Just like that, we have a functional form. Sure, it's a bit ugly - but that will be *super* easy to fix. Before we get there, however, we need to talk about handling the form submit.

Chapter 2: Handling the Form Submit

Creating the form class and rendering was... easy! Now it's time to talk about handling the form *submit*. Notice: we haven't configured *anything* on our form about what *URL* it should submit to. When we rendered it, we used form_start() and... that's it! Inspect element on the form. By default, form_start() creates a form tag with *no* action attribute. And when a form tag has *no* action=, it means that it will submit right back to this *same* URL.

That's the most common way of handling forms in Symfony: the same controller is responsible for both *rendering* the form on a GET request *and* handling the form *submit* on a POST request. The way you *do* this always follows a similar pattern.

The Form Submit Logic

First, get the $request object by type-hinting Request - the one from HttpFoundation. Next, add $form->handleRequest($request) and then if ($form->isSubmitted() && $form->isValid()). Inside the if, dd($form->getData().

43 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 13
class ArticleAdminController extends AbstractController
{
... lines 16 - 19
    public function new(EntityManagerInterface $em, Request $request)
    {
... lines 22 - 23
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
            dd($form->getData());
        }
... lines 28 - 31
    }
... lines 33 - 41
}
```

Okay, so... this requires a *little* bit of explanation. First, yea, the $form->handleRequest() makes it *look* like the submitted data is being read and processed on *every* request, even the initial GET request that renders the form. But, that's not true! By default, handleRequest() *only* processes the data when this is a POST request. So, when the form is being submitted. When the form is originally loaded, handleRequest() sees that this is a GET request, does nothing, $form->isSubmitted() returns false, and then the un-submitted form is rendered by Twig.

But, when we POST the form, ah, that's when handleRequest() does its magic. Because the form knows all of its fields, *it* grabs all of the submitted data from the $request automatically and isSubmitted() returns true. Oh, and later, we'll talk about adding validation to our form. As you can guess, when validation fails, $form->isValid() returns false.

So, wow! This controller does a lot, with very little code. And there are *three* possible flows. One: if this is a GET request, isSubmitted() returns false and so the form is passed to Twig. Two, if this is a POST request but validation fails, isValid() returns false and so the form is *again* passed to Twig, but *now* it will render with errors. We'll see that later. And three: if this is a POST request and validation *passes*, both isSubmitted() *and* isValid() are true, and we finally get into the if block. $form->getData() is how *we* access the final, normalized data that was submitted.

Phew! So, let's try it! Find your browser and create a very important article about the booming tourism industry on Mercury. Submit!

Yes! It dumps out *exactly* what we probably expected: an array with title and content keys. It's not too fancy yet, but it works nicely.

Saving the Form Data

To insert a new article into the database, we need to use this data to create an Article object. There is a *super* cool way to do

this automatically with the form system. But, to start, let's do it the manual way. Add $data = $form->getData(). Then, create that object: $article = new Article(), $article->setTitle($data['title']);, $article->setContent($data['content']), and the author field is also required. How about, $article->setAuthor() with $this->getUser(): the current user will be the author.

52 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 19
20      public function new(EntityManagerInterface $em, Request $request)
21      {
... lines 22 - 24
25          if ($form->isSubmitted() && $form->isValid()) {
26              $data = $form->getData();
27              $article = new Article();
28              $article->setTitle($data['title']);
29              $article->setContent($data['content']);
30              $article->setAuthor($this->getUser());
... lines 31 - 35
36          }
... lines 37 - 40
41      }
... lines 42 - 52
```

To save this to the database, we need the entity manager. And, hey! We *already* have it thanks to our EntityManagerInterface argument. Save with the normal $em->persist($article), $em->flush().

Awesome! The *last* thing we *always* do after a successful form submit is redirect to another page. Let's use return this->redirectToRoute('app_homepage').

52 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 24
25          if ($form->isSubmitted() && $form->isValid()) {
... lines 26 - 31
32              $em->persist($article);
33              $em->flush();
34
35              return $this->redirectToRoute('app_homepage');
36          }
... lines 37 - 52
```

Time to test this puppy out! Refresh to re-post the data. Cool! I... *think* it worked? Scroll down... Hmm. I don't see my article. Ah! But that's because only *published* articles are shown on the homepage.

Adding an Article List Page

What we *really* need is a way to see *all* of the articles in an admin area. We have a "new" article page and a work-in-progress edit page. Now, create a new method: public function list(). Above it, add the annotation @Route("/admin/article"). To fetch all of the articles, add an argument: ArticleRepository $articleRepo, and then say $articles = $articleRepo->findAll(). At the bottom, render a template - article_admin/list.html.twig- and pass this an articles variable.

65 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 52
    /**
     * @Route("/admin/article")
     */
    public function list(ArticleRepository $articleRepo)
    {
        $articles = $articleRepo->findAll();

        return $this->render('article_admin/list.html.twig', [
            'articles' => $articles,
        ]);
    }
... lines 64 - 65
```

Oh, and I'll cheat again! If you have the Symfony plugin installed, you can put your cursor in the template name and press Alt+Enter to create the Twig template, right next to the other one.

Because we're *awesome* at Twig, the contents of this are pretty boring. In fact, I'm going to cheat again! I'm on a roll! I'll paste a template I already prepared. You can get this from the code block on this page.

30 lines | templates/article_admin/list.html.twig

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

{% block content_body %}
    <a href="{{ path('admin_article_new') }}" class="btn btn-primary pull-right">
        Create <span class="fa fa-plus-circle"></span>
    </a>
    <h1>All Articles</h1>

    <table class="table">
        <thead>
            <tr>
                <th>Title</th>
                <th>Author</th>
                <th>Published?</th>
            </tr>
        </thead>
        <tbody>
            {% for article in articles %}
                <tr>
                    <td>{{ article.title }}</td>
                    <td>{{ article.author.email }}</td>
                    <td>
                        <span class="fa fa-{{ article.isPublished ? 'check' : 'times' }}"></span>
                    </td>
                </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

And... yea! Beautifully boring! This loops over the articles and prints some basic info about each. I also added a link on top to the new article form page.

Oh, there is *one* interesting part: the article.isPublished code, which I use to show a check mark or an "x" mark. That's

interesting because... we don't have an isPublished property or method on Article! Add public function isPublished(), which will return a bool, and very simply, return $this->publishedAt !== null.

255 lines | src/Entity/Article.php

```
... lines 1 - 15
class Article
{
... lines 18 - 127
    public function isPublished(): bool
    {
        return $this->publishedAt !== null;
    }
... lines 132 - 253
}
```

If you want to be fancier, you could check to see if the publishedAt date is not null and also not a *future* date. It's up to how you want your app to work.

Time to try it! Manually go to /admin/article and... woohoo! *There* is our new article on the bottom.

And... yea! We've *already* learned enough to create, render *and* process a form submit! Nice work! Next, let's make things a bit fancier by rendering a success message after submitting.

Chapter 3: Success (Flash) Messages

Our form submits and saves! But... it's not all *that* obvious that it works... because we redirect to the homepage... and there's not even a success message to tell us it worked! We can do better!

In ArticleAdminController, give the list endpoint route a name="admin_article_list". After a successful submit, we can redirect there. That makes more sense.

67 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
class ArticleAdminController extends AbstractController
{
... lines 17 - 54
    /**
     * @Route("/admin/article", name="admin_article_list")
     */
    public function list(ArticleRepository $articleRepo)
... lines 59 - 65
}
```

Adding a Flash Message

With that done, I *next* want to add a "success" message. Like, after I submit, there's a giant, happy-looking green bar on top that says "Article created! You're a modern-day Shakespeare!".

And... great news! Symfony has a feature that's *made* for this. It's called a *flash* message. Oooooo. After a successful form submit, say $this->addFlash(). Pass this the key success - we'll talk about that in a moment - and then an inspirational message!

> Article Created! Knowledge is power

67 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 20
    public function new(EntityManagerInterface $em, Request $request)
    {
... lines 23 - 25
        if ($form->isSubmitted() && $form->isValid()) {
... lines 27 - 35
            $this->addFlash('success', 'Article Created! Knowledge is power!');
... lines 37 - 38
        }
... lines 40 - 43
    }
... lines 45 - 67
```

That's *all* we need in the controller. The addFlash() method is a shortcut to set a message in the *session*. But, flash messages are special: they only live in the session until they are *read* for the first time. As soon as we read a flash message, poof! In a... *flash*, it disappears. It's the *perfect* place to store temporary messages.

Rendering the Flash Message

Oh, and the success key? I just made that up. That's sort of a "category" or "type", and we'll use it to *read* the message and render it. And... *where* should we read and render the message? The *best* place is in your base.html.twig *layout*. Why? Because no matter *what* page you redirect to after a form submit, your flash message will then be rendered.

Scroll down a little bit and find the block body. Right *before* this - so that it's not overridden by our child templates, add {% for message in app.flashes() %} and pass this our type: success. Remember: Symfony adds *one* global variable to Twig called app, which comes in handy here.

Inside the for, add a div with class="alert alert-success" and, inside, print message.

```
92 lines | templates/base.html.twig
    ... lines 1 - 15
16      <body>
    ... lines 17 - 66
67          {% for message in app.flashes('success') %}
68              <div class="alert alert-success">
69                  {{ message }}
70              </div>
71          {% endfor %}
    ... lines 72 - 89
90      </body>
    ... lines 91 - 92
```

Done! Oh, but, why do we need a for loop here to read the message? Well, it's not *too* common, but you can technically put as *many* messages onto your success flash type as you want. So, in theory, there could be *5* success messages that we need to read and print... but you'll usually have just one.

Anyways, let's try this crazy thang! Move back so we can create another important article:

> Ursa Minor: Major Construction Planned

Hit enter and... hello nice message! I don't like that weird margin issue - but we'll fix that in a minute. When you refresh, yep! The message disappears in a flash... because it was *removed* from the session when we read it the first time.

Peeking at the Flash Messages

Ok, let's fix that ugly margin issue... it's actually interesting. Inspect element on the page and find the navbar. Ah, *it* has some bottom margin thanks to this mb-5 class. Hmm. To make this look right, we *don't* want to render that mb-5 class when there is a flash message. How can we do that?

Back in base.html.twig, scroll up a bit to find the navbar. Ok: we could *count* the number of success flash messages, and if there are *more* than 0, do *not* print the mb-5 class. That's *pretty* simple, except for one huge problem! If we read the flash messages here to count them, that would also *remove* them! Our loop below would *never* do *anything*!

How can we work around that? By *peeking* at the flash messages. Copy the class. Then, say app.session.flashbag.peek('success'). Pipe that to the length filter and if this is greater than zero, print nothing. Otherwise, print the mb-5 class.

```
93 lines | templates/base.html.twig
    ... lines 1 - 23
24      <nav class="navbar navbar-expand-lg navbar-dark navbar-bg {{ app.session.flashbag.peek('success')|length > 0 ? '' : 'mb-5' }}">
    ... lines 25 - 93
```

This... deserves some explanation. First, the global app variable is actually an *object* called, conveniently, AppVariable! Press Shift+Shift and search for this so we can see *exactly* what it looks like.

Before, we used the getFlashes() method, which handles all the details of working with the Session object. But, if we need to "peek", we need to work with the Session directly via the getSession() shortcut. It turns out, the "flash messages" are stored on a sub-object called the "flash bag". This new longer code fetches the Session, gets that "FlashBag" and calls peek() on it.

Ok, let's see if that fixed things! Move back over and click to author *another* amazing article:

> Mars: God of War? Or Misunderstood?

Hit enter to submit and... got it! Flash message *and* no extra margin.

Next, let's learn how we can do... less work! By bossing around the form system and forcing it to create and populate our Article object so we don't have to.

Chapter 4: Bind Your Form to a Class

We created a form type class, used it in the controller to process the form submit *and* rendered it. This is pretty basic, but the form system is already doing a lot for us!

But... I think the form component can do more! Heck, I think it's been downright *lazy*. $data = $form->getData() gives us an associative array with the submitted & normalized data. That's cool... but it *does* mean that we need to set all of that data onto the Article object manually. Lame!

Setting the data_class Option

But, no more! Open ArticleFormType. Then, go back to the Code -> Generate menu - or Cmd+N on a Mac - select "Override Methods" and choose configureOptions(). Just like with buildForm(), we don't need to call the parent method because it's empty. Inside add $resolver->setDefaults() and pass an array. This is where you can set *options* that control how your form behaves. And, well... there aren't actually very many options. The most important, by *far*, is data_class. Set it to Article::class. This *binds* the form to that class.

27 lines | src/Form/ArticleFormType.php

```
... lines 1 - 9
class ArticleFormType extends AbstractType
{
... lines 12 - 19
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Article::class
        ]);
    }
}
```

And... yep! *That* little option changes *everything*. Ready to see how? Back in your controller, dd($data).

68 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
class ArticleAdminController extends AbstractController
{
... lines 17 - 20
    public function new(EntityManagerInterface $em, Request $request)
    {
... lines 23 - 25
        if ($form->isSubmitted() && $form->isValid()) {
... line 27
            dd($data);
... lines 29 - 39
        }
... lines 41 - 44
    }
... lines 46 - 66
}
```

Now, move back to your browser. Watch closely: right now both fields are simple text inputs... because we haven't configured them to be anything else. But, refresh!

Form Field Type Guessing

Whoa! The content is now a textarea! We haven't talked about it yet, but we can, of course, configure how each field is rendered. By default, if you do nothing, everything renders as a text input. But, when you bind your form to a class, a special system - called the "form type guessing" system - tries to *guess* the proper "type" for each field. It notices that the $content property on Article is a longer text Doctrine type. And so, it basically says:

> Hey peeps! This content field looks pretty big! So, let's use a textarea field type by default.

Anyways, form field type guessing is a *cool* feature. But, it is actually *not* the super important thing that just happened.

What was? Create another breaking news story:

> Orion's Belt: for Fashion or Function?

Click Create and... yes! Check it out! $form->getData() is now an Article object! And the title and content properties are already set! *This* is the power of the data_class option.

When the form submits, it notices the data_class and so creates a new Article() object for us. Then, it uses the *setter* methods to populate the data. For example, the form has two fields: title and content. When we submit the form, it calls setTitle() and then setContent(). It's basically just an automatic way to do what we are *already* doing manually in our controller. This is *awesome* because we can remove code! Just say $article = $form->getData(), done. To help PhpStorm I'll add some inline documentation that says that this is an Article.

65 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 25
26          if ($form->isSubmitted() && $form->isValid()) {
27              /** @var Article $article */
28              $article = $form->getData();
... lines 29 - 36
37          }
... lines 38 - 65
```

That's great! Our controller is tiny and, when we submit, bonus! It even works!

Model Classes & Complex Forms

In most cases, *this* is how I use the form system: by binding my forms to a class. But! I *do* want you to remember one thing: if you have a super complex form that looks different than your entity, it's perfectly okay to *not* use data_class. Sometimes it's simpler to build the form exactly how you want, call $form->getData() and use that associative array in your controller to update what you need.

Oh, and while we *usually* see form types bound to an *entity* class, that's not required! This class could be *any* PHP class. So, if you have a form that doesn't match up well with any of your entities, you *can* still use data_class. Yep! Create a new model class that has the same properties as your form, set the data_class to that class, submit the form, get *back* that model object from the form, and use it inside your controller to do whatever you want!

Oh, and if this isn't *quite* making sense: no worries - we'll practice this later.

Form Theme: Making your Form Beautiful

Before we keep going, let's take 30 seconds to make our ugly form... beautiful! So far, we're not controlling the markup that's rendered in *any* way: we call a few form rendering functions and... somehow... we get a form!

Behind the scenes, *all* of this markup comes from a set of special Twig templates called "form themes". And yea, we *can* and totally *will* mess with these. If you're using Bootstrap CSS or Foundation CSS, ah, you're in luck! Symfony comes with a built-in form theme that makes your forms render *exactly* how these systems want.

Open config/packages/twig.yaml. Add a new key called form_themes with one element that points to a template called bootstrap_4_layout.html.twig.

7 lines | config/packages/twig.yaml

```yaml
twig:
    ... lines 2 - 4
    form_themes:
        - bootstrap_4_layout.html.twig
```

This template lives deep inside the core of Symfony. And we'll check it out later when we talk more about form themes. Because right now... we get to celebrate! Move over and refresh. Ha! Our form is instantly pretty! The form system is now rendering with Bootstrap-friendly markup.

Next: let's talk about customizing the "type" of each field so we can make it look and act exactly how we need.

Chapter 5: Field Types & Options

Our form has an input text field and a textarea... which is *interesting* because, in our form class, all *we've* done is give each field a name. We... actually have *not* said *anything* about what "type" of field each should be.

But, we *now* know that, because we've bound our form to our entity, the form type "guessing" system is able to *read* the Doctrine metadata, notice that content looks like a big field, and "guess" that it should be a textarea.

Setting the Field Type

Field type guessing is cool! But... it's not meant to be perfect: we need a way to take control. How? It turns out that the add() method has *three* arguments: the field name, the field *type* and some options.

For title pass, TextType::class - the one from Form\Extensions. Go back to the form refresh and... absolutely *nothing* changes! Symfony was already "guessing" that this was a TextType field.

So Many Built-in Field Types

Google for "Symfony forms", click into the Symfony form documentation and find a section called "Built-in Field Types". Woh. It turns out that there are a *ton* of built-in field types. Yep, there's a field type for *every* HTML5 field that exists, as well as a few other, special ones.

Click into TextType. In addition to choosing which *type* you need, every type is super configurable. Many of these options are global, meaning, the options can be used for *any* field type. A good example is the label option: you can set a label option for *any* field, regardless of its type.

But other options are specific to that field type. We'll see an example in a minute.

Check out this help option: we can define a "help" message for any field. That sounds awesome! Back in the form class, add a third argument: an options array. Pass help and set it to "Choose something catchy".

30 lines | src/Form/ArticleFormType.php

```
... lines 1 - 10
class ArticleFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title', TextType::class, [
                'help' => 'Choose something catchy!'
            ])
... lines 19 - 20
    }
... lines 22 - 28
}
```

Let's go check it out! Refresh! I like that! Nice little help text below the field.

The Form Profiler

This *finally* gives us a reason to check out one of the *killer* features of Symfony's form system. Look down at the web debug toolbar and find the little clipboard icon. Click that.

Yes! Say hello to the form profiler screen! We can see our *entire* form and the individual fields. Click on the title field to get all sorts of information about it. We're going to look at this *several* more times in this tutorial and learn about each piece of information.

Form Options in Profiler

Under "Passed Options" you can see the help option that we just passed. But, what's *really* cool are these "Resolved Options". This is a list of *every* option that was used to control the rendering & behavior of this *one* field. A lot of these are low-level and don't directly do anything for *this* field - like the CSRF stuff. That's why the official docs can sometimes be easier to look at. But, this is an *amazing* way to see what's *truly* going on under the hood: what are *all* the options for this field and their values.

And, yea! We can override *any* of these options via the third argument to the add() method. Without even reading the docs, we can see that we could pass a label option, label_attr to set HTML attributes on your label, or a required option... which actually has nothing to do with validation, but controls whether or not the HTML5 required attribute should be rendered. More on that later.

Anyways, let's add another, more *complex* field - and use its options to totally transform how it looks and works.

Chapter 6: DateTimeType & Data "Transforming"

Let's use our new powers to add another field to the form. Our Article class has a publishedAt *DateTime* property. Depending on your app, you might *not* want this to be a field in your form. You might just want a "Publish" button that sets this to today's date when you click it.

But, in *our* app, I want to allow whoever is writing the article to specifically set the publish date. So, add publishedAt to the form... but *don't* set the type.

31 lines | src/Form/ArticleFormType.php

```
... lines 1 - 10
class ArticleFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 16 - 19
            ->add('publishedAt')
        ;
    }
... lines 23 - 29
}
```

So... ah... this is interesting! How will Symfony render a "date" field? Let's find out! Refresh! Woh... it's a bunch of dropdowns for the year, month, day and time. That... will *technically* work... but that's not my favorite.

Which Field Type was Guessed?

Go back to the list of field types. Obviously, this is working because the field guessing system guessed... *some* field type. But... which one? To find out, go back to the web debug toolbar, click to open the profiler and select publishedAt. Ha! Right on top: DateTimeType. Nice!

Let's click into the DateTimeType documentation. Hmm... it has a *bunch* of options, and *most* of these are special to *this* type. For example, you can't pass a with_seconds option to a TextType: it makes no sense, and Symfony will yell at you.

Anyways, *one* of the options is called widget. Ah! This defines how the field is *rendered*. And if you did a little bit of digging, you would learn that we can set this to single_text to get a more user-friendly field.

Passing Options but No Type

To set an option on the publishedAt field, pass null as the second argument and set up the array as the third. null just tells Symfony to continue "guessing" this field type. Basically, I'm being lazy: we could pass DateTimeType::class ... but we don't need to!

Under the options, set widget to single_text.

33 lines | src/Form/ArticleFormType.php

```
... lines 1 - 19
            ->add('publishedAt', null, [
                'widget' => 'single_text'
            ])
... lines 23 - 33
```

Let's see what that did! Find your form, refresh and... cool! It's a text field! Right click and "Inspect Element" on that. Double cool! It's an <input type="datetime-local" ...>. That's an HTML5 input type that gives us a cool calendar widget. Unfortunately, while this will work on *most* browsers, it will not work on *all* browsers. If the user's browser has no idea how to handle a

datetime-local input, it will fall back to a normal text field.

If you need a fancy calendar widget for *all* browsers, you'll need to add some JavaScript to do that. We did that in our Symfony 3 forms tutorial and, later, we'll talk a bit about JavaScript and forms in Symfony 4.

Data Transforming

But, the reason I wanted to show you the DateTimeType was *not* because of this HTML5 fanciness. Nope! The *really* important thing I want you to notice is that, regardless of browser support, when we submit this form, it will send this field as a simple, date *string*. But... wait! *We* know that, on submit, the form system will call the setPublishedAt() method. And... that requires a DateTime *object*, not a string! Won't this totally explode?

Actually... no! It will work *perfectly*.

In reality, each field type - like DateTimeType - has *two* superpowers. First, it determines *how* the field is rendered. Like, an input type="text" field or, a bunch of drop-downs, or a fancy datetime-local input. Second... and this is the *real* superpower, a field type is able to *transform* the data to and from your object and the form. This is called "data transformation".

I won't do it now, but when we submit, the DateTimeType will *transform* the submitted date *string* into a DateTime object and *then* call setPublishedAt(). Later, when we create a page to edit an *existing* Article, the form system will call getPublishedAt() to fetch the DateTime object, and then the DateTimeType will transform *that* into a string so it can be rendered as the value of the input.

We'll talk more about data transformers later. Heck, we're going to create one! Right now, I just want you to realize that this is happening behind the scenes. Well, not *all* fields have transformers: simple fields that hold text, like an input text field or textarea don't need one.

Next: let's talk about one of Symfony's most important and most versatile field types: ChoiceType. It's the over-achiever in the group: you can use it to create a select drop down, multi-select, radio buttons or checkboxes. Heck, I'm pretty sure it even knows how to fix a flat tire.

Let's work with it - and its brother the EntityType - to create a drop-down list populated from the database.

Chapter 7: EntityType: Drop-downs from the Database

On submit, we set the author to *whoever* is currently logged in. I want to change that: sometimes the person who *creates* the article, isn't the author! They need to be able to *select* the author.

ChoiceType: Maker of select, radio & checkboxes

Go to the documentation and click back to see the list of form field types. One of the most *important* types in *all* of Symfony is the ChoiceType. It's kind of the loud, confident, over-achiever in the group: it's able to create a select drop-down, a multi-select list, radio buttons *or* checkboxes. It even works on weekends! Phew!

If you think about it, that makes sense: those are all different ways to *choose* one or more items. You pass this type a choices option - like "Yes" and "No" - and, by default, it will give you a select drop-down. Want radio buttons instead? Brave choice! Just set the expanded option to true. Need to be able to select "multiple" items instead of just one? Totally cool! Set multiple to true to get checkboxes. The ChoiceType is awesome!

But... we have a special case. Yes, we *do* want a select drop-down, but we want to *populate* that drop-down from a table in the database. We *could* use ChoiceType, but a much easier, ah, choice, is EntityType.

Hello EntityType

EntityType is kind of a "sub-type" of choice - you can see that right here: parent type ChoiceType. That means it basically works the same way, but it makes it easy to get the choices from the database and has a few different options.

Head over to ArticleFormType and add the new author field. I'm calling this author because that's the name of the property in the Article class. Well, actually, that doesn't matter. I'm calling this author because this class has setAuthor() and getAuthor() methods: *they* are what the form system will call behind the scenes.

34 lines | src/Form/ArticleFormType.php

```
... lines 1 - 10
class ArticleFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 16 - 22
            ->add('author')
... line 24
    }
... lines 26 - 32
}
```

As *soon* as we add this field, go try it! Refresh! Hello drop-down! It *is* populated with all the users from the database... but... it might look a little weird. By default, the EntityType queries for all of the User objects and then uses the __toString() method that we have on that class to figure out what display value to use. So, firstName. If we did *not* have a __toString() method, we would get a huge error because EntityType wouldn't know what to do. Anyways, we'll see in a minute how we can control what's displayed here.

Set the Type, Options are Not Guessed

So... great first step! It looks like the form guessing system correctly sees the Doctrine relation to the User entity and configured the EntityType for us. Go team!

But now, pass the type manually: EntityType::class. That should make no difference, right? After all, the guessing system was *already* setting that behind the scenes!

35 lines | src/Form/ArticleFormType.php

```
... lines 1 - 23
            ->add('author', EntityType::class)
... lines 25 - 35
```

Well... we're programmers. And so, we know to expect the unexpected. Try it! Surprise! A huge error!

> The required option class is missing

But, why? First, the EntityType has *one* required option: class. That makes sense: it needs to know which entity to query for. Second, the form type guessing system does *more* than just guess the form *type*: it can also guess certain field *options*. Until now, it was guessing EntityType *and* the class option!

But, as *soon* as you pass the field type explicitly, it stops guessing *anything*. That means that *we* need to manually set class to User::class. This is why I often *omit* the 2nd argument if it's being guessed correctly. And, we *could* do that here.

38 lines | src/Form/ArticleFormType.php

```
... lines 1 - 24
            ->add('author', EntityType::class, [
                'class' => User::class,
            ])
... lines 28 - 38
```

Try it again. Got it!

Controlling the Option Display Value

Let's go see what *else* we can do with this field type. Because EntityType's parent is ChoiceType, they share a lot of options. One example is choice_label. If you're not happy with using the __toString() method as the display value for each option... too bad! I mean, you can *totally* control it with this option!

Add choice_label and set it to email, which means it should call getEmail() on each User object. Try this. I like it! Much more obvious.

39 lines | src/Form/ArticleFormType.php

```
... lines 1 - 24
            ->add('author', EntityType::class, [
... line 26
                'choice_label' => 'email',
            ])
... lines 29 - 39
```

Want to get fancier? I thought you would. You can *also* pass this option a callback, which Symfony will call for each item and pass it the data for that option - a User object in this case. Inside, we can return whatever we want. How about return sprintf('(%d) %s') passing $user->getId() and $user->getEmail().

41 lines | src/Form/ArticleFormType.php

```
... lines 1 - 24
            ->add('author', EntityType::class, [
... line 26
                'choice_label' => function(User $user) {
                    return sprintf('(%d) %s', $user->getId(), $user->getEmail());
                }
            ])
... lines 31 - 41
```

Cool! Refresh that! Got it!

The "Choose an Option" Empty Value

Another useful option that EntityType shares with ChoiceType is placeholder. This is how you can add that "empty" option on top - the one that says something like "Choose your favorite color". It's... a little weird that we *don't* have this now, and so the first author is auto-selected.

Back on the form, set placeholder to Choose an author. Try that: refresh. Perfecto!

42 lines | src/Form/ArticleFormType.php

```
... lines 1 - 24
            ->add('author', EntityType::class, [
... lines 26 - 29
                'placeholder' => 'Choose an author'
            ])
... lines 32 - 42
```

With all of this set up, go back to our controller. And... remove that setAuthor() call! Woo! We don't need it anymore because the form will call that method *for* us and pass the selected User object.

We just learned how to use the EntityType. But... well... we haven't talked about the most *important* thing that it does for us! Data transforming. Let's talk about that next and learn how to create a custom query to select and order the users in a custom way.

Chapter 8: EntityType: Custom Query

Right click and "Inspect Element". Look at the *value* of each option: it's the id of that user in the database. So, when we choose an author, *this* is the value that will be submitted to the server: this *number*. Just remember that.

Time to author another award-winning article:

> Pluto: I didn't want to be a Planet Anyways

Set the publish date to today at any time, select an author and... create! Yes! The author is spacebar3@example.com and it *is* published.

This is *way* more amazing than it might look at first! Sure, the EntityType is cool because it makes it easy to create a drop-down that's populated from the database. Blah, blah, blah. That's fine. But the *truly* amazing part of EntityType is its *data transformer*. It's the fact that, when we submit a *number* to the server - like 17 - it queries the database and *transforms* that into a User object. That's important because the form system will *eventually* call setAuthor(). And this method *requires* a User object as an argument - not the number 17. The data transformer is the magic that makes that happen.

Creating a Custom Query

We can use this new knowledge to our advantage! Go back to the create form. What if we don't want to show *all* of the users in this drop-down? Or, what if we want to control their *order*. How can we do that?

Normally, when you use the EntityType, you don't need to pass the choices option. Remember, if you look at ChoiceType, the choices option is how you specify which, ah, *choices* you want to show in the drop-down. But EntityType queries for the choices and basically sets this option *for* us.

To control that query, there's an option called query_builder. *Or*, you can do what I do: be less fancy and simply override the choices option entirely. Yep, you basically say:

> Hey EntityType! Thanks... but I can handle querying for the choices myself. But, have a *super* day.

Injecting Dependencies

To do this, we need to execute a *query* from inside of our form class. And to do *that*, we need the UserRepository. But... great news! Form types are services! So we can use our favorite pattern: dependency injection.

Create an __construct() method with an UserRepository argument. I'll hit alt+enter, and select "Initialize Fields" to create that property and set it. Down below, pass choices set to $this->userRepository and I'll call a new method ->findAllEmailAlphabetical().

52 lines | src/Form/ArticleFormType.php

```
... lines 1 - 14
class ArticleFormType extends AbstractType
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }
... line 23
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 27 - 33
            ->add('author', EntityType::class, [
... lines 35 - 39
                'choices' => $this->userRepository->findAllEmailAlphabetical(),
            ])
... line 42
    }
... lines 44 - 50
}
```

Copy that name, go to src/Repository/, open UserRepository, and create that method. Use the query builder: return $this->createQueryBuilder('u') and then ->orderBy('u.email', 'ASC'). Finish with ->getQuery() and ->execute().

Above the method, *we* know that this will return an array of User objects. So, let's advertise that!

63 lines | src/Repository/UserRepository.php

```
... lines 1 - 14
class UserRepository extends ServiceEntityRepository
{
... lines 17 - 21
    /**
     * @return User[]
     */
    public function findAllEmailAlphabetical()
    {
        return $this->createQueryBuilder('u')
            ->orderBy('u.email', 'ASC')
            ->getQuery()
            ->execute()
        ;
    }
... lines 33 - 61
}
```

I love it! This makes our ArticleFormType class happy. I think we should try it! Refresh! Cool! The admin users are first, then the others.

So... is EntityType Still Needed?

But... wait. Now that we're *manually* setting the choices option... do we even need to use EntityType anymore? Couldn't we switch to ChoiceType instead?

Actually... no! There is one *super* critical thing that EntityType is still giving us: data transformation. When we submit the form,

we *still* need the submitted *id* to be transformed back into the correct User object. So, even though we're querying for the options manually, it is *still* doing this very important job for us. Remember: the *true* power of a field type is this data transformation ability.

Next: let's add some form validation! It might work a little differently than you expect.

Chapter 9: HTML5 & "Sanity" Validation

Does our form have any validation yet? Well... sort of? The form *is* going through a validation process. When we POST to this endpoint, handleRequest() reads the data *and* executes Symfony's validation system. If validation fails, then $form->isValid() returns false and we immediately render the template, except that *now* errors will be displayed by each field with an error.

Of course we haven't *seen* this yet... because we haven't added any validation rules!

HTML5 Validation

But, check this out: leave the form completely blank and try to submit. It stops us! Wait... who... stopped us? Actually, it was the *browser*. Many of you may recognize this: it's HTML5 validation.

When Symfony renders a field, depending on our config, it often adds a required="required" attribute. This isn't *real* validation - there's *nothing* on our server that's checking to make sure this value isn't blank. It's just nice client-side validation. HTML5 is cool... but limited. There *are* a few other things it can validate. Like, a datetime-local field will require you to enter a valid date. Or, an <input type="number"> will require a number. But, not much more.

The Annoying required Attribute

To control whether or not you want that required attribute, *every* field type has an option called required: just set it to true or false. Actually, this option is kinda confusing. It defaults to *true* for *every* field... no matter what... which can be kind of annoying & surprising. However, when you bind your form to an entity class, the form field guessing system uses the nullable Doctrine option to choose the correct required option value for you. In fact, if we look at the textarea field... yep! This has no required attribute. Oh, by the way, all those extra attributes are coming from a browser plugin I have installed - not the form system.

To make things a bit *more* confusing, the required option is only "guessed" from your Doctrine config if you omit or pass null to the second argument of add(). If you specify the type manually, the form type guessing system does nothing and you'll need to configure the required option manually. Honestly, the required option is kind of a pain in the butt. Be careful to make sure that an optional field doesn't accidentally have this attribute.

Installing Validation

Anyways, even if you use HTML5 validation, you will *still* need proper server-side validation so that a "bad" user can't just disable that validation and send weird data. To do that, well, first we need to install the validator!

Find your terminal and run:

```
$ composer require validator
```

Validation is a separate component in Symfony, which is *cool* because it means that you can use it independent of the form system.

And... done! There are actually *two* types of server-side validation: what I call "sanity validation" versus "business rules validation".

Form Field Sanity Validation

Let's talk about sanity validation first. Sanity validation is built into the form fields themselves and makes sure that the submitted value isn't completely... insane! For text fields like title and content, there is no sanity validation: we can submit anything to those fields and it basically makes sense: it's a string. But the EntityType *does* have built-in sanity validation.

Check this out: inspect element in your browser and find the select field. Let's change one of these values to be something that's *not* in the database, like value=100.

Select this user and hit Create. Oh, duh! The HTML5 validation on the other fields stops us. To work around this, find the form class and add a novalidate attribute: that tells the browser to get a hobby and skip HTML5 validation. It's a nice trick when

you're testing your server-side validation. Hit Create again.

Yay! Our first, *real* validation error!

> This value is not valid

This error comes from the "sanity" validation that's built into EntityType: if you try to submit a value that should *not* be in the drop-down, boom! You get an error. Sanity validation is great: it saves us, *and*... we don't need to think about it! It just works.

To control the message, pass an option called invalid_message. Set it to:

> Symfony is too smart for your hacking!

53 lines | src/Form/ArticleFormType.php

```
... lines 1 - 14
15  class ArticleFormType extends AbstractType
16  {
... lines 17 - 23
24      public function buildForm(FormBuilderInterface $builder, array $options)
25      {
26          $builder
... lines 27 - 33
34              ->add('author', EntityType::class, [
... lines 35 - 40
41                  'invalid_message' => 'Symfony is too smart for your hacking!'
42              ])
... line 43
44      }
... lines 45 - 51
52  }
```

Move back and refresh to re-POST the form. Nice! I don't *usually* set the invalid_message, only because, for most fields, you only see these errors if you're doing something *really* weird - like messing with the HTML.

We've talked about HTML5 validation and learned about sanity validation. Next, let's get to the *good* stuff: the *real* validation that *we* need to add.

Chapter 10: Validation Constraints with @Assert

When you talk about validation, what you're *really* talking about is business rules validation. That's where you tell Symfony that the title is required and needs to be a certain length, or that some field should be a valid email address, or that the password must contain 2 upper case letters, 2 lower case letters, 3 unicode characters and at least 4 emojis. It's about making the data constrain to *your* application's rules.

Adding your First Assert Annotation

Symfony's validation is kinda interesting because you do *not* apply the validation rules to the form. Nope, you apply them to your class via annotations. Check this out: I want the title field to be required. To do that, type @NotBlank and hit tab to autocomplete to @Assert\NotBlank. Because I have the PHP annotations plugin installed, when I auto-completed that, it added a use statement on top that we need: use Symfony\Component\Validator\Constraints as Assert.

257 lines | src/Entity/Article.php

```
... lines 1 - 11
use Symfony\Component\Validator\Constraints as Assert;
... lines 13 - 16
class Article
{
... lines 19 - 27
    /**
... line 29
     * @Assert\NotBlank()
     */
    private $title;
... lines 33 - 255
}
```

Without doing *anything* else, refresh the form - the title field *is* empty. Yes! That's our error!

> This value should not be blank.

To customize that, add a message key to the annotation:

> Get creative and think of a title!

257 lines | src/Entity/Article.php

```
... lines 1 - 27
    /**
... line 29
     * @Assert\NotBlank(message="Get creative and think of a title!")
     */
    private $title;
... lines 33 - 257
```

Try it again - refresh and... nice!

The Built-in Validation Constraints

On the docs, click to go back to the documentation homepage. Then, under guides, find the "Validation" guide. Just like with the form fields, there are a *bunch* of built-in validation constraints that... can help you validate almost anything! And... just like with form field types, each validation constraint has different options that control its behavior.

For example - check out Length: you can set the min length with the min option, or max with max. Control their error messages with minMessage and maxMessage.

Oh, *another* way to see what the options are is to remember that every annotation has a concrete PHP *class* behind it. Thanks to the PHP annotations plugin, I can hold Command or Ctrl and click the annotation to jump to that class.

Nice! Every property becomes an *option* that you can pass to the annotation. We'll see this again later when we create our *own* custom validation constraint.

Anyways, we won't talk too much about validation constraints because... they're honestly pretty easy: it's usually a matter of finding which validation constraint you need and the options to pass to it.

The Callback Constraint

Oh, but there *is* one really cool constraint called Callback. This is *the* tool when you need to go rogue and do something totally custom. Check it out: create a method in your class and add @Assert\Callback() above it. Then, during validation, Symfony will call your method!

Let's copy this, find our Article class, go all the way to the bottom, and paste. Oh, I need to retype the end of ExecutionContextInterface and auto-complete it to get the use statement. Then, inside... it's awesome! We can do whatever we want!

```
270 lines | src/Entity/Article.php
    ... lines 1 - 12
13  use Symfony\Component\Validator\Context\ExecutionContextInterface;
    ... lines 14 - 17
18  class Article
19  {
    ... lines 20 - 257
258     /**
259      * @Assert\Callback
260      */
261     public function validate(ExecutionContextInterface $context, $payload)
262     {
    ... lines 263 - 267
268     }
269 }
```

Let's make sure that the title of this Article doesn't contain the string the borg... cause they're scary. So, if stripos() of $this->getTitle() and the borg does not equal false... error! To create the error, use $context->buildViolation():

> Um.. the Borg kinda makes us nervous

Apparently *so* nervous that I typed "the Bork" instead! Resistance to typos is futile...

Next, choose which field to attach the error to with ->atPath('title') and finish with ->addViolation(). That's it!

```
270 lines | src/Entity/Article.php
    ... lines 1 - 260
261     public function validate(ExecutionContextInterface $context, $payload)
262     {
263         if (stripos($this->getTitle(), 'the borg') !== false) {
264             $context->buildViolation('Um.. the Bork kinda makes us nervous')
265                 ->atPath('title')
266                 ->addViolation();
267         }
268     }
    ... lines 269 - 270
```

Go back to our form, write an article about how you *really* want to join the borg and Create!

Got it! Custom validation logic with a custom error.

Next: let's talk a little more about how we can control the *rendering* of these fields. Because, right now, we're just sort of

rendering them all at once... without much control over their look and feel.

Chapter 11: Form Rendering Functions: form_*

To render the form, we're using a few form functions: one that makes the form start tag, one that makes the end tag and one that renders all the fields, labels and errors inside.

This was *easy* to set up. The *problem* is that we have almost *no* control over the HTML markup that's used! Sure, we were able to activate a "form theme" that told it to use Bootstrap-friendly markup. But, what if you need more control?

This is probably *the* hardest part of Symfony's form system. But don't worry: we're going to learn several different strategies to help you get the markup you need... without going crazy... probably.

The Form Rendering Functions

Go to your other tab and Google for "Symfony form rendering functions" to find a page that talks all about the functions we're using and a few others.

First, form_start(), yes, this *does* just render the form start tag, which might seem kind of silly, but it can come in handy when you add a file upload field to your form: it automatically add the enctype attribute.

Oh, but notice: form_start() has a second argument: an array of *variables* that can be passed to customize it. Apparently you can pass method to change the method attribute or attr to add any *other* attributes to the form tag - like a class.

Next: find form_end(). This one seems even *sillier* because it literally prints... yep! The form closing tag! But, it has a hidden superpower: it *also* renders any fields that we forgot to render. Now, that might not make sense yet because this magic form_widget() function seems to be rendering *everything* automatically. But, in a moment, we'll render the fields one-by-one. When we do that, *if* we've forgotten to render any of the fields, form_end() will render them for us... and *then* the closing tag.

That *still* may not seem like a good feature... and, in many ways, it's not! In reality, the *purpose* of this is *not* so that we can be lazy and form_end() will save us. Nope - the *true* purpose is that form_end() will render any hidden fields automatically, without us needing to even think about them. Most importantly, it will render your form's CSRF token

CSRF Token

Inspect element near the bottom of the form. Woh! Without us doing *anything*, we have a hidden input tag called _token. *This* is a CSRF token and it was automatically added by Symfony. And, even *cooler*, when we submit, Symfony automatically validates it.

Without even knowing it, all of our forms are protected from CSRF attacks.

form_widget and form_row()

Back to the form rendering goodness! To print the form fields themselves, the *easiest* way is to call form_widget() and pass the entire form. But, if you need a *little* bit more control, instead of form_widget(), you can call form_row() and render each field individually. For example, articleForm.title. Copy that and paste it three more times. Render articleForm.content, articleForm.publishedAt and articleForm.author.

15 lines | templates/article_admin/new.html.twig

```
... lines 1 - 5
6       {{ form_start(articleForm) }}
7           {{ form_row(articleForm.title) }}
8           {{ form_row(articleForm.author) }}
9           {{ form_row(articleForm.content) }}
10          {{ form_row(articleForm.publishedAt) }}
11
... line 12
13      {{ form_end(articleForm) }}
... lines 14 - 15
```

Before we talk about this function, move over, refresh and... ok! It looks *exactly* the same. That's no accident! Calling form_widget() and passing it the entire form is just a shortcut for calling form_row() on each field individually.

This introduces an important concept in Symfony's form rendering system: the "row". The form_row() function basically adds a "wrapper" around the field - like a div - then renders the *4* components of each field: the label, the "widget" - that's the form field itself, the help text and, if needed, the validation errors.

At first, it looks like using form_row() isn't much more flexible than what we had before, except that we can reorder the fields. But, in reality, we've just unlocked quite a *lot* of control via a system called "form variables". Let's check those out next!

Chapter 12: Form Rendering Variables

Find the form_row() documentation. There is one *super* important thing that almost all of these functions share: their last argument is something called variables.

These variables are *key* to controlling how each part of each field is rendered. And it's explained a bit more at the bottom. Yep - this table describes the most common "variables" - which are kind of like "options" - that you can pass to most fields, including label.

Let's override that variable for the title. Pass a second argument to form_row(): an array with a label key. How about, Article title.

17 lines | templates/article_admin/new.html.twig

```
... lines 1 - 5
{{ form_start(articleForm) }}
    {{ form_row(articleForm.title, {
        label: 'Article title'
    }) }}
... lines 10 - 14
{{ form_end(articleForm) }}
... lines 16 - 17
```

Try that! Reload the form. Boom! Label changed!

Discovering the Form Variables

There are *tons* of variables that you can pass to the form rendering functions to change how the field is rendered. And the *list* of those variables will be *slightly* different for each field *type*. The *best* way to see *all* the possibilities is back inside our best friend: the form profiler.

Click on the field you want to customize - like title. I'll collapse the options stuff. Remember: options are what we can pass to the third argument of the add() function in our form class.

For rendering, we're interested in the "View Variables". Behind the scenes, each part of each field is rendered by a mini Twig template that lives inside Symfony. We'll see this later. These variables are *passed* to those Twig templates and used to control, well, almost everything.

Hey! There's the label variable we just overrode! Notice, it's null: the values inside the profiler represent the values at the moment the form object is passed into the Twig template. So, if you override a value, it won't show up here. No big deal - just don't let that surprise you.

Ah, and there's a help message and a whole bunch of other things that help the form do its job, like full_name, which will be the name attribute, and even the id attribute.

By the way, if it's useful, in addition to *overriding* these variables, you can access them directly in your template. I don't need it here, but you could, for example print articleForm.title.vars.id.

If you go back and look at your form, that will print the id attribute that's used for the form field. Pretty cool, though, the *real* purpose of variables is to *override* them via the form functions.

form_label(), form_widget(), form_help(), form_errors()

Using form_row() gave us more flexibility, because we can reorder the fields and override the field's variables. If you need a little bit *more* flexibility, another option is to render the 4 components of each field independently.

For example, get rid of form_row. And, instead, render each part manually: {{ form_label(articleForm.title) }}, and for this function, the second argument is the label. Then {{ form_errors(articleForm.title) }} for the validation errors, {{ form_widget(articleForm.title }} to print the input field, and finally {{ form_help(articleForm.title }}.

19 lines | templates/article_admin/new.html.twig

```
... lines 1 - 6
        {{ form_label(articleForm.title, 'Article title') }}
        {{ form_errors(articleForm.title) }}
        {{ form_widget(articleForm.title) }}
        {{ form_help(articleForm.title) }}
... lines 11 - 19
```

Let's see how this compares to using form_row(). Refresh! Hmm - it's *almost* the same. But if you look more closely, of course! The other fields are wrapped in a form-group div, but the title no longer has that!

When you render things at this level, you start to lose some of the special formatting that form_row() gives you. Sure, it's easy to re-add that div. But form_row also adds a special error class to that div when the field has a validation error.

For that reason, let's go back to using form_row().

17 lines | templates/article_admin/new.html.twig

```
... lines 1 - 6
        {{ form_row(articleForm.title, {
            label: 'Article title'
        }) }}
... lines 10 - 17
```

A little bit later, we're going to learn how we can use form_row(), but completely customize how it looks for one specific form, or across your entire site! We'll do this by creating a "form theme". It's kind of the best of both worlds: you can render things in the lazy way, but still have the control you need.

But before we get there - let's learn how to create an "edit" form so we can update articles!

Chapter 13: The Edit Form

We know what it looks like to create a *new* Article form: create the form, process the form request, and save the article to the database. But what does it look like to make an "edit" form?

The answer is - delightfully - almost identical! In fact, let's copy all of our code from the new() action and go down to edit(), where the only thing we're doing so far is allowing Symfony to query for our article. Paste! Excellent.

.

81 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
class ArticleAdminController extends AbstractController
{
... lines 17 - 46
    public function edit(Article $article)
    {
        $form = $this->createForm(ArticleFormType::class);

        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
            /** @var Article $article */
            $article = $form->getData();

            $em->persist($article);
            $em->flush();

            $this->addFlash('success', 'Article Created! Knowledge is power!');

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

        return $this->render('article_admin/new.html.twig', [
            'articleForm' => $form->createView()
        ]);
    }
... lines 68 - 79
}
```

Oh, but we need a few arguments: the Request and EntityManagerInterface $em. This is now *exactly* the same code from the new form. So... how can we make this an edit form? You're going to love it! Pass $article as the second argument to ->createForm().

80 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 46
    public function edit(Article $article, Request $request, EntityManagerInterface $em)
    {
        $form = $this->createForm(ArticleFormType::class, $article);
... lines 50 - 65
    }
... lines 67 - 80
```

We're done! Seriously! When you pass $article, this object - which we just got from the database becomes the *data* attached to the form. This causes two things to happen. First, when Symfony renders the form, it calls the *getter* methods on that Article

object and uses those values to fill in the values for the fields.

Heck, we can see this immediately! This is using the new template, but that's fine temporarily. Go to /article/1/edit. Dang - I *don't* have an article with id

1. Let's go find a real id. In your terminal, run:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article'
```

Perfect! Let's us id 26. Hello, completely pre-filled form!

The *second* thing that happens is that, when we submit, the form system calls the *setter* methods on that *same* Article object. So, we *can* still say $article = $form->getData()... But these two Article objects will be the *exact* same object. So, we don't need this.

So.. ah... yea! Like I said, we're done! By passing an existing object to createForm() our "new" form becomes a perfectly-functional "edit" form. Even Doctrine is smart enough to know that it needs to *update* this Article in the database instead of creating a new one. Booya!

Tweaks for the Edit Form

The *real* differences between the two forms are all the small details. Update the flash message:

> Article updated! Inaccuracies squashed!

80 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 51
52          if ($form->isSubmitted() && $form->isValid()) {
... lines 53 - 55
56              $this->addFlash('success', 'Article Updated! Inaccuracies squashed!');
... lines 57 - 60
61          }
... lines 62 - 80
```

And then, instead of redirecting to the list page, give this route a name="admin_article_edit". Then, redirect right back here! Don't forget to pass a value for the id route wildcard: $article->getId().

80 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 42
43      /**
44       * @Route("/admin/article/{id}/edit", name="admin_article_edit")
... line 45
46       */
47      public function edit(Article $article, Request $request, EntityManagerInterface $em)
48      {
... lines 49 - 51
52          if ($form->isSubmitted() && $form->isValid()) {
... lines 53 - 57
58              return $this->redirectToRoute('admin_article_edit', [
59                  'id' => $article->getId(),
60              ]);
61          }
... lines 62 - 65
66      }
... lines 67 - 80
```

Controller, done!

Next, even though it worked, we don't *really* want to re-use the same Twig template, because it has text like "Launch a new

article" and "Create". Change the template name to edit.html.twig. Then, down in the templates/article_admin directory, copy the new.html.twig and name it edit.html.twig, because, there's not *much* that needs to be different.

Update the h1 to Edit the Article and, for the button, Update!.

17 lines | templates/article_admin/edit.html.twig

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

{% block content_body %}
    <h1>Edit the Article! ?</h1>

    {{ form_start(articleForm) }}
        {{ form_row(articleForm.title, {
            label: 'Article title'
        }) }}
        {{ form_row(articleForm.author) }}
        {{ form_row(articleForm.content) }}
        {{ form_row(articleForm.publishedAt) }}

        <button type="submit" class="btn btn-primary">Update!</button>
    {{ form_end(articleForm) }}
{% endblock %}
```

Cool! Let's try this - refresh! Looks perfect! Let's change some content, hit Update and... we're back!

Reusing the Form Rendering Template

Cool *except*... I don't *love* having all this duplicated form rendering logic - especially if we start customizing more stuff. To avoid this, create a *new* template file: _form.html.twig. I'm prefixing this by _ *just* to help me remember that this template will render a little bit of content - not an entire page.

Next, copy the *entire* form code and paste! Oh, but the button needs to be different for each page! No problem: render a new variable: {{ button_text }}.

10 lines | templates/article_admin/_form.html.twig

```
{{ form_start(articleForm) }}
    {{ form_row(articleForm.title, {
        label: 'Article title'
    }) }}
    {{ form_row(articleForm.author) }}
    {{ form_row(articleForm.content) }}
    {{ form_row(articleForm.publishedAt) }}

    <button type="submit" class="btn btn-primary">{{ button_text }}</button>
{{ form_end(articleForm) }}
```

Then, from the edit template, use the include() function to include article_admin/_form.html.twig and pass one *extra* variable as a second argument: button_text set to Update!.

10 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 2
{% block content_body %}
... lines 4 - 5
    {{ include('article_admin/_form.html.twig', {
        button_text: 'Update!'
    }) }}
{% endblock %}
```

Copy this and repeat it in new: remove the duplicated stuff and say Create!.

10 lines | templates/article_admin/new.html.twig

```
... lines 1 - 2
{% block content_body %}
... lines 4 - 5
    {{ include('article_admin/_form.html.twig', {
        button_text: 'Create!'
    }) }}
{% endblock %}
```

I love it! Let's double-check that it works. No problems on edit! And, if we go to /admin/article/new... nice!

Adding an Edit Link

And just to make our admin section *even* more awesome, back on the list page, let's add a link to edit each article. Open list.html.twig, add a new empty table header, then, in the loop, create the link with href="path('admin_article_edit')" passing an id wildcard set to article.id. For the text, print an icon using the classes fa fa-pencil.

38 lines | templates/article_admin/list.html.twig

```
... lines 1 - 9
        <thead>
            <tr>
... lines 12 - 14
                <th> </th>
            </tr>
        </thead>
        <tbody>
            {% for article in articles %}
                <tr>
... lines 21 - 25
                    <td>
                        <a href="{{ path('admin_article_edit', {
                            id: article.id
                        }) }}">
                            <span class="fa fa-pencil"></span>
                        </a>
                    </td>
                </tr>
            {% endfor %}
        </tbody>
... lines 36 - 38
```

Cool! Try that out - refresh the list page. Hello pencil icon! Click any of these to hop right into that form.

We just saw one of the most pleasant things about the form component: edit and new pages are almost identical. Heck, the Form component can't even tell the difference! All it knows is that, if we *don't* pass an Article object, it needs to create one. And if we *do* pass an Article object, it says, okay, I'll just update that object instead of making a new one. In both cases, Doctrine is smart enough to INSERT or UPDATE correctly.

Next: let's turn to a *super* interesting form use-case: our highly-styled registration form.

Chapter 14: Registration Form

Head back over to /register. We built this in our security tutorial. It *does* work... but we kind of cheated. Back in your editor, open src/Controller/SecurityController.php and find the register() method. Yep, it's pretty obvious: we did *not* use the form component. Instead, we manually read and handled the POST data. The template - templates/security/register.html.twig - is just a hardcoded HTML form.

Ok, first: *even* if you use and love the Form component, you do *not* need to use it in *every* single situation. If you have a simple form and want to skip it, sure! You can totally do that. But... our registration form is missing one key thing that *all* forms should have: CSRF protection. When you use the Form component. you get CSRF protection for free! And, usually, that's enough of a reason for me to use it. But, you *can* add CSRF protection without the form system: check out our login from for an example.

make:form

Let's refactor our code to use the form system. Remember step 1? Create a form class... like we did with ArticleFormType. That's pretty easy. But to be even *lazier*, we can generate it! Find your terminal and run:

```
$ php bin/console make:form
```

Call the class, UserRegistrationFormType. This will ask if you want this form to be bound to a *class*. That's *usually* what we want, but it's optional. Bind our form to the User class.

Nice! It created one new file. Find that and open it up!

30 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 9
class UserRegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('email')
            ->add('roles')
            ->add('firstName')
            ->add('password')
            ->add('twitterUsername')
        ;
    }

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

Customizing & Using UserRegistrationFormType

Cool. It set the data_class to User and even looked at the properties on that class and pre-filled the fields! Let's see: we don't want roles or twitterUsername for registration. And, firstName is something that I won't include either - the current form has just these two fields: email and password.

27 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 11
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('email')
            ->add('password')
        ;
    }
... lines 19 - 27
```

Ok: step 2: go the controller and create the form! And, yes! I get to remove a "TODO" in my code - that never happens! Use the normal $form = this->createForm() and pass this UserRegistrationFormType::class. But don't pass a second argument: we *want* the form to create a new User object.

Then, add $form->handleRequest($request) and, for the if, use $form->isSubmitted() && $form->isValid().

75 lines | src/Controller/SecurityController.php

```
... lines 1 - 14
class SecurityController extends AbstractController
{
... lines 17 - 44
    public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
    {
        $form = $this->createForm(UserRegistrationFormType::class);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
... lines 51 - 72
    }
}
```

Beautiful, boring, normal, code. And now that we're using the form system, instead of creating the User object like chumps, say $user = $form->getData(). I'll add some inline documentation so that PhpStorm knows what this variable is. Oh, and we don't need to set the email directly anymore: the form will do that! And I'll remove my firstName hack: we'll fix that in a minute.

75 lines | src/Controller/SecurityController.php

```
... lines 1 - 44
    public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
    {
... lines 47 - 49
        if ($form->isSubmitted() && $form->isValid()) {
            /** @var User $user */
            $user = $form->getData();
... lines 53 - 67
        }
... lines 69 - 72
    }
... lines 74 - 75
```

About the password: we *do* need to encode the password. But now, the plain text password will be stored on $user->getPassword(). Hmm. That *is* a little weird: the form system is setting the plaintext password on the password field. And then, a moment later, we're encoding that and setting it *back* on that *same* property! We're going to change this in a few minutes - but, it should work.

75 lines | src/Controller/SecurityController.php

```
... lines 1 - 49
50          if ($form->isSubmitted() && $form->isValid()) {
... lines 51 - 52
53              $user->setPassword($passwordEncoder->encodePassword(
54                  $user,
55                  $user->getPassword()
56              ));
... lines 57 - 67
68          }
... lines 69 - 75
```

Down below when we render the template, pass a new registrationForm variable set to $form->createView().

75 lines | src/Controller/SecurityController.php

```
... lines 1 - 44
45      public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
46      {
... lines 47 - 69
70          return $this->render('security/register.html.twig', [
71              'registrationForm' => $form->createView(),
72          ]);
73      }
... lines 74 - 75
```

Awesome! Let's find that template and get to work. Remove the TODO - we're killing it - then comment out all the old markup: I want to keep it for reference. Render with {{ form_start(registrationForm) }}, form_end(registrationForm) and, in the middle, render *all* of the fields with form_widget(registrationForm). Oh, and we need a submit button. Steal that from the old code and move it here.

45 lines | templates/security/register.html.twig

```
... lines 1 - 10
11  {% block body %}
... lines 12 - 13
14          <div class="col-sm-12">
15              {{ form_start(registrationForm) }}
16                  {{ form_widget(registrationForm) }}
17
18                  <button class="btn btn-lg btn-primary btn-block" type="submit">
19                      Register
20                  </button>
21              {{ form_end(registrationForm) }}
... lines 22 - 40
41          </div>
... lines 42 - 43
44  {% endblock %}
```

Perfect! Let's go check this thing out! Refresh! Oh... wow... it looks *terrible*! Our old form code *was* using Bootstrap... but it was *pretty* customized. We *will* need to talk about how we can get back our good look.

Making firstName Optional

But, other than that... it seems to render fine! Before we test it, open your User entity class. We originally made the firstName field not nullable. That's the *default* value for nullable. So if you *don't* see nullable=true, it means that the field *is* required in the database.

Now, I *do* want to allow users to register *without* their firstName. No problem: set nullable=true.

247 lines | src/Entity/User.php

```
... lines 1 - 13
class User implements UserInterface
{
... lines 16 - 33
    /**
     * @ORM\Column(type="string", length=255, nullable=true)
... line 36
     */
    private $firstName;
... lines 39 - 245
}
```

Then, find your terminal and run:

```
$ php bin/console make:migration
```

Let's go check out that new file. Yep! No surprises: it just makes the column not required.

29 lines | src/Migrations/Version20181018165320.php

```
... lines 1 - 10
final class Version20181018165320 extends AbstractMigration
{
    public function up(Schema $schema) : void
    {
... lines 15 - 17
        $this->addSql('ALTER TABLE user CHANGE first_name first_name VARCHAR(255) DEFAULT NULL');
    }
... lines 20 - 27
}
```

Move back over and run this with:

```
$ php bin/console doctrine:migrations:migrate
```

Excellent! Let's try to register! Register as geordi@theenterprise.org, password, of course, engage. Hit enter and... nice! We are even logged in as Geordi!

Next: we have a problem! We're temporarily storing the plaintext password on the password field... which is a *big* no no! If something goes wrong, we might *accidentally* save the user's plaintext password to the database.

To fix that, we, for the *first* time, will add a field to our form that does *not* exist on our entity. An awesome feature called mapped will let us do that.

Chapter 15: Adding Extra "Unmapped" Fields

UserRegistrationFormType has a password field. But *that* means, when the user types in their password, the form component will call setPassword() and pass it that *plaintext* property, which will be stored on the password property.

That's both *weird* - because the password field should *always* be encrypted - *and* a potential security issue: if we somehow accidentally save the user at this moment, that plaintext password will go into the database.

And, yea before we save, we *do* encrypt that plaintext password and set that *back* on the password property. But, I don't like doing this: I don't like *ever* setting the plaintext password on a property that could be persisted: it's just risky, and, kind of strange to use this property in two ways.

Go back to UserRegistrationFormType. Change the field to plainPassword. Let's add a comment above about why we're doing this.

29 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 9
class UserRegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... line 15
            // don't use password: avoid EVER setting that on a
            // field that might be persisted
            ->add('plainPassword')
        ;
    }
... lines 21 - 27
}
```

But... yea! This *will* break things! Go back to the form and try to register with a different user. Boom!

> Neither the property plainPassword nor one of the methods getPlainPassword() blah, blah, blah, exist in class User.

And we know why this is happening! Earlier, we learned that when you add a field to your form called email, the form system, calls getEmail() to read data off of the User object. And when we submit, it calls setEmail() to set the data back *on* the object. Oh, and, it *also* calls getEmail() on submit to so it can *first* check to see if the data changed at all.

Anyways, the form is basically saying:

> Hey! I see this plainPassword field, but there's no way for me to get or set that property!

There are two ways to fix this. First, we *could* create a plainPassword property on User, but make it *not* persist it to the database. So, *don't* put an @ORM\Column annotation on it. Then, we could add normal getPlainPassword() and setPlainPassword() methods... and we're good! That solution is simple. But it also means that we've added this extra property to the class *just* to help make the form work.

Unmapped (mapped => false) Fields

The *second* solution is... a bit more interesting: we can mark the field to not be "mapped". Check it out: pass null as the second argument to add() so it continues guessing the field type for now. Then, pass a new option: mapped set to false.

That changes everything. This tells the form system that we *do* want to have this plainPassword field on our form, but that it should *not* get or set its data back onto the User object. It means that we *no* longer need getPlainPassword() and setPlainPassword() methods!

Accessing Unmapped Fields

Woo! Except... wait, if the form doesn't set this data onto the User object... how the heck can we access that data? After all, when we call $form->getData(), it gives us the User object. Where will that plainPassword data live?

In your controller, dd($form['plainPassword']->getData()).

```
76 lines | src/Controller/SecurityController.php
... lines 1 - 14
15  class SecurityController extends AbstractController
16  {
... lines 17 - 44
45      public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
46      {
... lines 47 - 49
50          if ($form->isSubmitted() && $form->isValid()) {
51              dd($form['plainPassword']->getData());
... lines 52 - 68
69          }
... lines 70 - 73
74      }
75  }
```

Then move over, refresh and... oh! Form contains extra fields. My fault: I never fully refreshed the form after renaming password to plainPassword. So, we were *still* submitting the old password field. By default, if you submit *extra* fields to a form, you get this validation error.

Let's try that again. This time... Yes! It hits our dump and die and *there* is our plain password!

This uncovers a *really* neat thing about the form system. When you call $this->createForm(), it creates a Form object that represents the whole form. But also, each individual *field* is *also* represented as its *own* Form object, and it's a *child* of that top-level form. Yep, $form['plainPassword'] gives us a Form object that knows everything about this *one* field. When we call ->getData() on it, yep! That's the value for this *one* field.

This is a *super* nice solution for situations where you need to add a field to your form, but it doesn't map cleanly to a property on your entity. Copy this, remove the dd() and, down below, use *that* to get the plain password.

```
75 lines | src/Controller/SecurityController.php
... lines 1 - 49
50          if ($form->isSubmitted() && $form->isValid()) {
... lines 51 - 52
53              $user->setPassword($passwordEncoder->encodePassword(
54                  $user,
55                  $form['plainPassword']->getData()
56              ));
... lines 57 - 67
68          }
... lines 69 - 75
```

Let's try it! Move back over, refresh and... got it! We are *registered*!

Using the PasswordType Field

Go back to /register - there is *one* more thing I want to fix before we keep going: the password field is a normal, plaintext input. That's not ideal.

Find your form class. The form field guessing system has *no* idea what type of field plainPassword is - it's not even a property on our entity! When guessing fails, it falls back to TextType.

Change this to PasswordType::class. This won't change how the field *behaves*, only how it's rendered. Yep! A proper <input type="password"> field.

32 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 12
13      public function buildForm(FormBuilderInterface $builder, array $options)
14      {
15          $builder
... lines 16 - 18
19              ->add('plainPassword', PasswordType::class, [
... line 20
21              ]);
... line 22
23      }
... lines 24 - 32
```

Next: time to add validation! Which, hmm, is going to be a bit interesting. First, we need to validate that the user is unique in the database. And second, for the first time, we need to add validation to a form field where there is *no* corresponding property on our class.

Chapter 16: UniqueEntity & Validation Directly on Form Fields

The registration form works, but we have a few problems. First, geez, it looks terrible. We'll fix that a bit later. More *importantly*, it *completely* lacks validation... except, of course, for the HTML5 validation that we get for free. But, we can't rely on that.

No problem: let's add some validation constraints to email and plainPassword! We know how to do this: add annotations to the class that is bound to this form: the User class. Find the email field and, above, add @Assert\NotBlank(). Make sure to hit tab to auto-complete this so that PhpStorm adds the use statement that we need on top. Also add @Assert\Email().

250 lines | src/Entity/User.php

```
... lines 1 - 14
class User implements UserInterface
{
... lines 17 - 23
    /**
... lines 25 - 26
     * @Assert\NotBlank()
     * @Assert\Email()
     */
    private $email;
... lines 31 - 248
}
```

Nice! Move back to your browser and inspect the form. Add the novalidate attribute so we can skip HTML5 validation. Then, enter "foo" and, submit! Nice! Both of these validation annotations have a message option - let's customize the NotBlank message: "Please enter an email".

250 lines | src/Entity/User.php

```
... lines 1 - 23
    /**
... lines 25 - 26
     * @Assert\NotBlank(message="Please enter an email")
... line 28
     */
    private $email;
... lines 31 - 250
```

Cool! email field validation, done!

Unique User Validation

But... hmm... there's one *other* validation rule that we need that's related to email: when someone registers, we need to make sure their email address isn't already registered. Try geordi@theenterprise.org again. I'll add the novalidate attribute so I can leave the password empty. Register! It *explodes*!

> Integrity constraint violation: duplicate entry "geordi@theenterprise.org

Ok, *fortunately*, we *do* have the email column marked as unique in the database. But, we *probably* don't want a 500 error when this happens.

This is the *first* time that we need to add validation that's not just as simple as "look at this field and make sure it's not blank", "or a valid email string". This time we need to look into the database to see if the value is valid.

When you have more complex validation situations, you have two options. First, try the Callback constraint! This allows you do *whatever* you need. Well, *mostly*. Because the callback lives inside your entity, you don't have access to any services. So,

you couldn't make a query, for example. If Callback doesn't work, the solution that *always* works is to create your very own custom validation constraint. That's something we'll do later.

Fortunately, we don't need to do that here, because validating for uniqueness is *so* common that Symfony has a built-in constraint to handle it. But, instead of adding this annotation above your property, it lives above your *class*. Add @UniqueEntity. Oh, and notice! This added a *different* use statement because this class happens to live in a different namespace than the others.

This annotation needs at least one option: the fields that, when combined, need to be unique. For us, it's just email. You'll probably want to control the message too. How about: I think you've already registered.

255 lines | src/Entity/User.php

```
... lines 1 - 7
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 9 - 12
/**
... line 14
 * @UniqueEntity(
 *     fields={"email"},
 *     message="I think you're already registered!"
 * )
 */
class User implements UserInterface
... lines 21 - 255
```

Oh, and just a reminder: if you have the PHP annotations plugin installed, you can hold command or control and click the annotation to open its class and see all its options.

Let's try it! Move over and refresh! Got it! That's a *much* nicer error.

Adding Validation Directly to Form Fields

There is *one* last piece of validation that's missing: the plainPassword field. At the very least, it needs to be *required*. But, hmm. In the form, this field is set to 'mapped' => false. There *is* no plainPassword property inside User that we can add annotations to!

No problem. Yes, we *usually* add validation rules via annotations on a class. But, if you have a field that's not mapped, you can add *its* validation rules directly to the form field via a constraints array option. What do you put inside? Remember how each annotation is represented by a concrete class? That's the key! *Instantiate* those as objects here: new NotBlank(). To pass options, use an array and set message to Choose a password!.

Heck, while we're here, let's also add new Length() so we can require a minimum length. Hold command or control and click to open that class and see the options. Ah, yea: min, max, minMessage, maxMessage. Ok: set min to, how about 5 and minMessage to Come on, you can think of a password longer than that!

43 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 12
class UserRegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 18 - 20
            ->add('plainPassword', PasswordType::class, [
                'mapped' => false,
                'constraints' => [
                    new NotBlank([
                        'message' => 'Choose a password!'
                    ]),
                    new Length([
                        'min' => 5,
                        'minMessage' => 'Come on, you can think of a password longer than that!'
                    ])
                ]
            ]);
        ;
    }
... lines 35 - 41
}
```

Done! These constraint options will work *exactly* the same as the annotations. To prove it, go back and refresh! Got it! Now, validating an unmapped field is no problem. We rock!

Next: the registration form is missing one *other* field: the boring, but, unfortunately, all-important "Agree to terms" checkbox. The solution... is interesting.

Chapter 17: Agree to Terms Database Field

If you compare our old registration form and our new one, we're missing one annoying, piece: the "Agree to terms" checkbox, which, if you're like me, is just one of my *favorite* things in the world - right behind a fine wine or day at the beach.

Legally speaking, this field *is* important. So let's code it up correctly.

Adding the "Agreed Terms" Persisted Date Field

A few years ago, we might have added this as a simple unmapped checkbox field with some validation to make sure it was checked. But *these* days, to be compliant, we need to save the *date* the terms were agreed to.

Let's start by adding a new property for that! Find your terminal and run:

```
$ php bin/console make:entity
```

Update the User class and add a new field called agreedTermsAt. This will be a datetime field and it *cannot* be nullable in the database: we need this to always be set. Hit enter to finish.

272 lines | src/Entity/User.php

```
... lines 1 - 19
class User implements UserInterface
{
... lines 22 - 68
    /**
     * @ORM\Column(type="datetime")
     */
    private $agreedTermsAt;
... lines 73 - 270
}
```

Adding the Checkbox Field

Before we worry about the migration, let's think about the form. What we want is very simple: a checkbox. Call it, how about, agreeTerms. Notice: this creates a familiar problem: the form field is called agreeTerms but the *property* on User is agreedTermsAt. We *are* going to need more setup to get this working.

45 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 13
class UserRegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 19 - 33
            ->add('agreeTerms', CheckboxType::class)
        ;
    }
... lines 37 - 43
}
```

But first, Google for "Symfony form types" and click the "Form Type Reference" page. Let's see if we can find a checkbox field - ah: CheckboxType. Interesting: it says that this field type should be used for a field that has a boolean value. If the box

is checked, the form system will set the value to true. If the box is unchecked, the value will be set to false. That makes sense! That's the *whole* point of a checkbox!

Back on the form, set the type to CheckboxType::class.

45 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 15
16      public function buildForm(FormBuilderInterface $builder, array $options)
17      {
18          $builder
... lines 19 - 33
34              ->add('agreeTerms', CheckboxType::class)
35          ;
36      }
... lines 37 - 45
```

Nice start! Before I forget, find your terminal and make the migration:

```
$ php bin/console make:migration
```

As usual, go to the migrations directory, open that file and... yep! It adds the one field. Run it with:

```
$ php bin/console doctrine:migrations:migrate
```

Oh no! Things are *not* happy. We have *existing* users in the database! When we suddenly create a new field that is NOT NULL, MySQL has a hard time figuring out what datetime value to use for the existing user rows!

Migrating Existing User Data

Our migration needs to be smarter. First: when a migration fails, Doctrine does *not* record it as having been executed. That makes sense. And because there is only *one* statement in this migration, we know that it completely failed, and we can try it again as soon as we fix it. In other words, the agreed_terms_at column was *not* added.

If a migration has *multiple* statements, it's possible that the first few queries *were* successful, and *then* one failed. When that happens, I usually delete the migration file entirely, fully drop the database, then re-migrate to get back to a "clean" migration state. But also, some database engines like PostgreSQL are smart enough to rollback the first changes, if a later change fails. In other words, those database engines avoid the problem of partially-executed-migrations.

Anyways, to fix the migration, change the NOT NULL part to DEFAULT NULL temporarily. Then add another statement: $this->addSql('UPDATE user SET agreed_terms_at = NOW()');.

30 lines | src/Migrations/Version20181016183947.php

```
... lines 1 - 10
11  final class Version20181016183947 extends AbstractMigration
12  {
13      public function up(Schema $schema) : void
14      {
... lines 15 - 17
18          $this->addSql('ALTER TABLE user ADD agreed_terms_at DATETIME DEFAULT NULL');
19          $this->addSql('UPDATE user SET agreed_terms_at = NOW()');
20      }
... lines 21 - 28
29  }
```

Great! First, let's run *just* this migration

```
$ php bin/console doctrine:migrations:migrate
```

This time... it works! To finish the change, make one more migration:

```
$ php bin/console make:migration
```

29 lines | src/Migrations/Version20181016184244.php

```
... lines 1 - 10
final class Version20181016184244 extends AbstractMigration
{
    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        $this->addSql('ALTER TABLE user CHANGE agreed_terms_at agreed_terms_at DATETIME NOT NULL');
    }
... lines 20 - 27
}
```

Go check it out! Perfect! This gives us the *last* piece we need: changing the column back to NOT NULL, which will *work* because each existing user now has a real value for this field. Oh, but, for *legal* purposes, on a real site - it may not be proper to automatically set the agreed_terms_at for existing users. Yep, you've gotta check with a lawyer on that kind of stuff.

But from a database migration standpoint, this should fix everything! Run the last migration:

```
$ php bin/console doctrine:migrations:migrate
```

Excellent! Next: we have a CheckboxType field on the form... which is good at setting true/false values. And, we have an agreedTermsAt DateTime field on the User class. Somehow, those need to work together!

Chapter 18: Agree to Terms Checkbox Field

The User class has an agreedTermsAt property that expects a DateTime object. But, our form has an agreeTerms field that, on submit, will give us a true/false boolean value. How can we make these work together? As I *so* often like to say: there are two options.

First, we could be clever! There is *no* agreeTerms property on User. But, we *could* create a setAgreeTerms() *method* on User. When that's called, we would *actually* set the agreedTermsAt property to the current date. We would also need to create a getAgreeTerms() method that would return a boolean based on whether or not the agreedTermsAt property was set.

This is a *fine* solution. But, this is *also* a good example of how the form system can start to make your life *harder* instead of *easier*. When your form and your class don't look the same, *sometimes* you can find a simple and natural solution. But sometimes, you might need to dream up something *crazy* to make it all work. If the solution isn't obvious to you, move on to option two: make the field unmapped.

Let's try that: set agreeTerms to mapped false. To *force* this to be checked, add constraints set to a new IsTrue()... because we *need* the underlying value of this field to be true, not false. Set a custom message:

> I know, it's silly, but you must agree to our terms

53 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 14
class UserRegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 20 - 34
            ->add('agreeTerms', CheckboxType::class, [
                'mapped' => false,
                'constraints' => [
                    new IsTrue([
                        'message' => 'I know, it\'s silly, but you must agree to our terms.'
                    ])
                ]
            ])
        ;
    }
... lines 45 - 51
}
```

Excellent! Thanks to the mapped = false, the form should *at least* load. Try it - refresh! Yes! Well... oh boy - our styling is *so* bad, the checkbox is hiding off the screen! Let's worry about that in a minute.

Thanks to the mapped => false, the data from the checkbox does *not* affect our User object in any way when we submit. No problem: in SecurityController, let's handle it manually with if (true === $form['agreeTerms']->getData()). Wait... that looks redundant! We *already* have form validation that *forces* the box to be checked. You're totally right! I'm just being *extra* careful... ya know... for legal reasons.

79 lines | src/Controller/SecurityController.php

```
... lines 1 - 44
45      public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
46      {
... lines 47 - 49
50          if ($form->isSubmitted() && $form->isValid()) {
... lines 51 - 56
57              // be absolutely sure they agree
58              if (true === $form['agreeTerms']->getData()) {
... line 59
60              }
... lines 61 - 71
72          }
... lines 73 - 76
77      }
... lines 78 - 79
```

Inside, we *could* call $user->setAgreedTermsAt() and pass the current date. *Or*, we can do something a bit cleaner. Find the setAgreedTermsAt() method and rename it to agreeTerms(), but with no arguments. Inside say $this->agreedTermsAt = new \DateTime().

270 lines | src/Entity/User.php

```
... lines 1 - 19
20  class User implements UserInterface
21  {
... lines 22 - 264
265     public function agreeTerms()
266     {
267         $this->agreedTermsAt = new \DateTime();
268     }
269 }
```

This gives us a clean, *meaningful* method. In SecurityController, call that: $user->agreeTerms().

79 lines | src/Controller/SecurityController.php

```
... lines 1 - 57
58              if (true === $form['agreeTerms']->getData()) {
59                  $user->agreeTerms();
60              }
... lines 61 - 79
```

Ok team, let's try this. Refresh the page. *Annoyingly*, I still can't see the checkbox. Let's hack that for now: add a little extra padding on this div. There it is!

Register as geordi3@theenterprise.org, password engage, hit enter, and... yes! We *know* the datetime column was just set correctly in the database because it's *required*.

Here's the big takeaway: whenever you need a field on your form that doesn't exist on your entity, there *may* be a clever solution. But, *if* it's not obvious, make the field unmapped and add a little bit of glue code in your controller that does whatever you need.

Later, we'll discuss a *third* option: creating a custom *model* class for your form.

Fixing your Fixtures

Before we move on, try to reload the fixtures:

```
$ php bin/console doctrine:fixtures:load
```

It... explodes! Duh! I made the new agreedTermsAt field *required* in the database, but forgot to update it in the fixtures. No problem: open UserFixture. In the first block, add $user->agreeTerms(). Copy that, and do the same for the admin users.

62 lines | src/DataFixtures/UserFixture.php

```
... lines 1 - 9
class UserFixture extends BaseFixture
{
... lines 12 - 18
    protected function loadData(ObjectManager $manager)
    {
        $this->createMany(10, 'main_users', function($i) use ($manager) {
... lines 22 - 24
            $user->agreeTerms();
... lines 26 - 41
        });
... line 43
        $this->createMany(3, 'admin_users', function($i) {
... lines 45 - 48
            $user->agreeTerms();
... lines 50 - 56
        });
... lines 58 - 59
    }
}
```

Cool! Try it again:

```
$ php bin/console doctrine:fixtures:load
```

And.... all better!

Next: let's fix the styling in our registration form by creating our very own form *theme*.

Chapter 19: All about Form Themes

There's just one problem left with our registration form - it looks terrible! It does *not* look like our original form, which was styled pretty nicely. One of the *trickiest* things to do with the form system is to style, or *theme* your forms. The system is super powerful: we just need to unlock its potential!

Adding Attributes to the <form> Tag

Here's the goal: make our form render the same markup that we had before. Let's start with something simple: the form tag had a class called form-signin. Google for "Symfony form function reference". Hey, we know this page! It lists all of the functions that we can call to render each part of our form! And it will give us a clue about how we can customize each part.

For example, the first argument to form_start() is called view. When you see "view" on this page, it's referring to your form variable. The *really* important argument is the second one: variables. We saw this before: almost *every* function has this mysterious variables argument. This is an array of, literally, Twig variables! They're used to render each part of the form.

For example, there is apparently a variable called method that you can set to control the method attribute on the form. But, it's not as simple as: every variable becomes an attribute. For example, we *can't* just pass a class variable to add a class attribute.

Scroll all the way down to the bottom of this page. Remember this table? It shows the most common variables that we can override. One of the most important ones is attr. Let's try that one! Add a second argument - an array, with an attr key set to *another* array with class set to form-signin. Phew! And while we're here, we also had an <h1> before. Add that right at the beginning of the form.

48 lines | templates/security/register.html.twig

```
... lines 1 - 14
15          {{ form_start(registrationForm, {
16              'attr': {'class': 'form-signin'}
17          }) }}
18              <h1 class="h3 mb-3 font-weight-normal">Register</h1>
... lines 19 - 48
```

Two small steps forward! Let's try it! Oh, it's already, *so* much better. Heck, I can even see my agree to terms checkbox again!

The Core Bootstrap Form Theme

Now... things get more interesting. The original fields were just a label and an input. The input has a class on it, but otherwise, it's pretty basic. But the Bootstrap form theme renders everything inside of a form-group div. *Then* there's the <label> and the <input>. So, hmm: we need to change how all of this markup is rendered. To do that we need to dive deep: we need to learn *how* the form theme system works under the hood.

Earlier, we opened config/packages/twig.yaml and added a form_themes line that pointed to a core template called bootstrap_4_layout.html.twig. This... instantly, made everything pretty! But... what did that *really* do?

Whenever Symfony renders *any* part of your form, there is a Twig template deep in the core that contains the markup for that one piece: like the label, the widget or the errors. Once we know how this work, we can *override* it!

Press Shift+Shift to open this template: bootstrap_4_layout.html.twig. This is probably the *strangest* Twig template that you'll ever see. It's, huh, just a *ton* of blocks: block time_widget, percent_widget, file_widget and many, many more.

Form Theme Template & Block System

Here's how it works: every field has five different components: the row and the 4 things it contains: the widget label, errors and help. When you render each part, Symfony opens this template, selects the correct block for the thing it's rendering, and renders it like a mini-template. It passes all the variables into that block. Yea, it's a *totally* cool, but weird use of Twig.

Go back to our form class to see an example. Oh, I totally forgot! We can set email to EmailType::class. That will make it render as <input type="email"> instead of text. And *that* will give us some extra HTML5 validation.

54 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 17
18      public function buildForm(FormBuilderInterface $builder, array $options)
19      {
20          $builder
21              ->add('email', EmailType::class)
... lines 22 - 44
45      }
... lines 46 - 54
```

Back to the theming stuff. Here's the key: to render the "widget" part of an "email" field type, Symfony looks for a block called email_widget. That's the pattern: the block is named by combining the field type - email, an underscore, then the "part" - widget: email_widget. Ok... so let's find it! Search for the block email_widget.

Oh... boo - it doesn't exist? What? Actually, this block lives in *another* template that lives right next to this. I'll click the Form directory on top, then open a *super* important template called form_div_layout.html.twig.

This is Symfony's default form theme template. And *even* if you don't list this in your twig.yaml file, Symfony *always* uses it. What I mean is: when Symfony searches for a block - like email_widget - it will look in bootstrap_4_layout.html.twig first. But if it is *not* there, it will *also* look here.

Let's search again for email_widget. Boom! *This* is the block that's responsible for rendering the widget part of the email field. Want to find the block that renders the widget part of PasswordType? There it is: password_widget. Both of these execute *another* block - form_widget_simple - to do the real work.

So... cool! By understanding the naming system for these blocks, we can create our *own* form theme template and create blocks that override *any* part of *any* field type. Sweet!

But... there is a surprise ahead! What is the name of the block that Symfony looks for when rendering the label part of a password type field? We know! It's password_label, right? Search for that. It's not here! And you won't find it in bootstrap_4_layout.html.twig either.

To understand why, we need to learn a bit about field type hierarchy. Then, we'll be ready to create our own form theme. That's next!

Chapter 20: Form Theme Block Naming & Creating our Theme!

When Symfony renders the "label" part of a password field type... it *should* be looking for a password_label block name. And... it *is*. But... that block doesn't exist! What's going on?

Here's the situation: the label will look the same for probably *every* field type: there's no difference between how a label should render for a text field versus a choice drop-down. To avoid duplicating the label code over and over again, the block system has a fallback mechanism.

Block Prefixes

Go back to your browser, click on the form icon on the web debug toolbar and select plainPassword. Go check out the "View Variables". Ah, here it is: the very special block_prefixes variable! This is an array that Symfony uses when trying to find which block to use. For example, to render the "widget" for this field, Symfony *first* looks for a block named _user_registration_form_plainPassword_widget.

This *super* specific block name will allow us to change how the widget looks for just *one* field of the form. We'll do this a bit later. If it does *not* find that block, it *next* looks for password_widget, then text_widget, and finally form_widget. There *is* a password_widget block but, when the label is being rendered, there is *not* a password_label block. Ok, so it next looks for text_label. Let's see if that exists. Nope! Finally, it looks for form_label. Search for that. Got it!

This is the block that used to render *every* label for *every* field type.

The Form Rendering Big Picture

Open up register.html.twig: let's back up and make sure this all makes sense. When we call form_widget(registrationForm), that's a shortcut for calling form_row() on each field. That means that the "row" part of each field is rendered. Not surprisingly, the "row" looks exactly the same for all field types. In other words, in bootstrap_4_layout.html.twig, you probably won't find a password_row block, but you *will* find a form_row block. Keep searching until you find it... there it is!

Ah, I love it! It has some special logic on top, but then! Yes: it renders a div with a form-group class then calls the form_label(), form_widget() and form_help() functions! The reason you don't see form_errors() here is that it's called from inside of form_label() so we can get the correct Bootstrap markup.

Creating our Form Theme

We *now* know enough to be dangerous! If we could override this form_row block *just* for the registration form, we could simplify the markup to match what we need. How do we do that? By creating our own form theme... which is just a template that contains these fancy blocks.

If you create a form theme in its own template file - like bootstrap_4_layout.html.twig - you can reuse it across your *entire* app by adding it to twig.yaml after bootstrap. Or, you can add some code to your Twig template to use a specific form theme template only on certain forms.

But, we actually will *not* create a separate template for our form theme. Why not? If you only need to customize a *single* form, there's an easier way. At the top of the template where you form lives, add {% form_theme %}, the name of your form variable - registrationForm - and then _self.

63 lines | templates/security/register.html.twig

```
... line 1
{% form_theme registrationForm _self %}
... lines 3 - 63
```

This says:

> Yo form system! I want to use *this* template as a form theme template for the registrationForm object.

As *soon* as we do this, when Symfony renders the form, it will *first* look for form theme blocks right inside of *this* template. Yep, we could copy that form_row block from Bootstrap, paste it, and start customizing!

Let's do that! But, actually, the Bootstrap form_row block is a bit fancier than I need. Instead, open form_div_layout.html.twig and find the block there. Copy that and, in register.html.twig, paste this anywhere.

63 lines | templates/security/register.html.twig

```
... lines 1 - 3
{% block form_row %}
    {%- set widget_attr = {} -%}
    {%- if help is not empty -%}
        {%- set widget_attr = {attr: {'aria-describedby': id ~"_help"}} -%}
    {%- endif -%}

    <div>
        {{- form_label(form) -}}
        {{- form_errors(form) -}}
        {{- form_widget(form, widget_attr) -}}
        {{- form_help(form) -}}
    </div>
{% endblock %}
... lines 17 - 63
```

Hmm - let's remove the wrapping and see if this works! Deep breath - refresh! I saw something move! Inspect the form and... yes! That wrapping div is gone!

61 lines | templates/security/register.html.twig

```
... lines 1 - 3
{% block form_row %}
    {%- set widget_attr = {} -%}
    {%- if help is not empty -%}
        {%- set widget_attr = {attr: {'aria-describedby': id ~"_help"}} -%}
    {%- endif -%}

    {{- form_label(form) -}}
    {{- form_errors(form) -}}
    {{- form_widget(form, widget_attr) -}}
    {{- form_help(form) -}}
{% endblock %}
... lines 15 - 61
```

When Symfony looks for the form_row() block it finds *our* block and uses it. All the other parts - like the widget and label blocks - are *still* coming from the Bootstrap theme. It's *perfect*.

But, we have more work to do! Next, let's learn a lot more about what we can do inside of these form theme blocks.

Chapter 21: Form Theming & Variables

We now know that when Symfony renders any part of your form, it looks for a specific block in this core form_div_layout.html.twig template. For example, to render the "row" part of any field, it looks for form_row. We *also* learned that this system has some hierarchy to it: to render the label part of a TextType field, it first looks for text_label and then falls back to using form_label.

Heck, there is even a form_start block that controls the open form tag!

We used this new knowledge to create our first form theme: we told Twig to look *right* inside *this* template for blocks to use when rendering the registration form. Our form_row block is now hooked into the form rendering process.

The Bizarre World of a Form Theme Block

When you're inside of a block that's used by the form theming system... your world is... weird. You *really* need to pretend like this block doesn't even exist in this template - like it lives all by itself in its *own*, isolated template. Why? Because these blocks are passed a *completely* different set of variables that come from the form system: this block doesn't work like *any* of the other blocks in this template.

I mean, look inside: there is apparently a help variable and a form variable. So, the *big* question is: when you're in a form theme block, what variables do you have access to?

The easiest answer is just to dump() inside one of these blocks.

62 lines | templates/security/register.html.twig

```
... lines 1 - 3
{% block form_row %}
... lines 5 - 9
    {{ dump() }}
... lines 11 - 14
{% endblock %}
... lines 16 - 62
```

Move over and refresh. Woh! Yes - we see *giant* dumps for *each* row that's rendered! There's attr, id and full_name. Do these... look familiar? These are the *exact* variables that we have been *overriding* when rendering our fields!

Look back at article_admin/_form.html.twig. We learned earlier that there is a variable called label and that the second argument of form_row() is an array of variables that you want to override. You can see this in the docs: when I search for form_row(), the second argument is variables.

Here's the point: when a field is rendered, the form system creates a bunch of variables to help that process, and we can override them. And *those* variable are ultimately passed... as variables, to your form theme blocks!

For example, remember how we passed a method variable to the form_start() function? Check out the form_start block in the bootstrap theme. Surprise! There is a local method variable that it uses to render. We *literally* override these variables via the form rendering functions.

The point is: when you're inside a form theme block, you have access to a lot of variables... which is *great*, because we can use those variables to do, well, whatever we need to!

Adding a label_attr

Back in register.html.twig, remove the dump(). On the old form, each label had an sr-only class. That stands for "screen reader only" and it makes the labels invisible.

How can we make *our* label tag have this? Hmm. Well, inside our block, we call form_label() and pass in the form object - which represents the form object for whatever field is currently being rendered.

Look back at the form function reference and search for form_label(). Ah yes: the second argument is the label itself. But the

third argument is an array of variables! And, apparently, there is a variable called label_attr! If we set that, we can control the attributes on the label tag.

In fact, we can see this: open form_div_layout.html.twig and search for form_label to find that block. There it is! It does some complex processing, but it *does* use this variable.

Actually, this is a great example of one, not-so-great thing about these templates: they can be crazy complex!

Anyways, back on register.html.twig, let's customize the label attributes! Pass null as the label text so it continues to use whatever the normal label is. Then pass an array with label_attr set to another array, and class equals sr-only.

63 lines | templates/security/register.html.twig

```
... lines 1 - 3
{% block form_row %}
... lines 5 - 9
    {{- form_label(form, null, {
            label_attr: { class: 'sr-only' }
    }) -}}
... lines 13 - 15
{% endblock %}
... lines 17 - 63
```

Phew! Let's try that. Move over refresh and... yes! They're gone! They now have an sr-only class! But, hmm... we now have *no* idea what these fields are! No worries: that was handled before via a placeholder attribute. New question: how can we set this for each field? Well... it's kind of the same thing: we want a custom attribute on each input.

The form_widget() function is being passed this widget_attr variable as its array of variables. So, we *could* add an attr key to it! Except... we don't know what the label should be! You *might* think that we could use the label variable. This *does* exist, but, unless you set the label explicitly, at this point, it's null. The form_label block holds the logic that turns the field name into a humanized label, if it wasn't set explicitly.

No problem: there's another simple solution. Refactor the form_widget() call into three, separate form_row() calls. Let me close a few files and - that's right! The fields are email plainPassword and agreeTerms. Use .email, copy those, paste twice, then plainPassword and agreeTerms.

For email pass a second argument with attr then placeholder set to Email. Do the same thing for the one other text field: placeholder set to "Password".

69 lines | templates/security/register.html.twig

```
... lines 1 - 29
            {{ form_start(registrationForm, {
... lines 31 - 33
                {{ form_row(registrationForm.email, {
                    attr: { placeholder: 'Email' }
                }) }}
                {{ form_row(registrationForm.plainPassword, {
                    attr: { placeholder: 'Password' }
                }) }}
                {{ form_row(registrationForm.agreeTerms) }}
... lines 41 - 44
            {{ form_end(registrationForm) }}
... lines 46 - 69
```

That should be it! And yea, we *could* have been less fancy and *also* passed this label_attr variable directly to form_row(). That would have worked *fine*.

Anyways, let's try it! Move over, refresh and... woohoo! The placeholders pop into place. And other than my obvious typo... I think it looks pretty good!

Next: there's one field left that isn't rendering correctly: the terms checkbox. Let's learn how to customize how a *single* field renders.

Chapter 22: Form Theming a Single Field

The last thing we need to do is fix this "agree to terms" checkbox. It doesn't look that bad... but this markup is *not* the markup that we had before.

This fix for this is... interesting. We want to override how the form_row is rendered... but *only* for this *one* field - not for everything. Sure, we could override the checkbox_row block... because this is the *only* checkbox on this form. But... could we get even *more* specific? Can we create a form theme block that only applies to a *single* field? Totally!

Go back and open the web debug toolbar for the form system. Click on the agreeTerms field and scroll down to the "View Variables". A few minutes ago we looked at this block_prefixes variable. When you render the "row" for a field, Symfony will *first* look for a block that starts with _user_registration_form_agreeTerms. So, _user_registration_form_agreedTerms_row. If it doesn't find that, which of course it will not, it falls back to the other prefixes, and eventually uses form_row.

Creating the Form Theme Block

To customize *just* this one field, copy that long block name and use it to create a new{% block _user_registration_form_agreeTerms_row %}, then {% endblock %}. Inside, let's *literally* copy the old HTML and paste.

Try it! Find the main browser tab and refresh. Whoops!

> A template that extends another cannot include content outside Twig blocks.

Yep, I pasted that in the wrong spot. Let's move it *into* the block. Come back and try that again. Yea! The checkbox moved back into place. Yep, the markup is exactly what we just pasted in.

Customizing with Variables

This is nice... but it's totally hardcoded! For example, if there's a validation error, it would *not* show up! No problem! Remember all of those variables we have access to inside form theme blocks? Let's put those to use!

First, inside, call {{ form_errors(form) }} to make sure any validation errors show up. I can also call form_help() if I wanted to, but we're not using that feature on this field.

Second: this name="_terms" is a problem because the form is expecting a different name. And so, this field won't process correctly. Replace this with the very handy full_name variable.

78 lines | templates/security/register.html.twig

```
... lines 1 - 17
{% block _user_registration_form_agreeTerms_row %}
    <div class="checkbox mb-3">
        {{ form_errors(form) }}
        <label>
            <input type="checkbox" name="{{ full_name }}" required> Agree to terms I for sure read
        </label>
    </div>
{% endblock %}
... lines 26 - 78
```

And... I think that's all I care about! Yes, we *could* get fancier, like using the id variable... if we cared. Or, we could use the errors variable to print a special error class if errors is not empty. It's all up to you.

The point is: get as fancy as your situation requires. Try the page one more time. It looks good *and* it will play nice with our form.

Next: let's learn how to create our own, totally custom field type! We'll eventually use it to create a special email text box with autocompletion to replace our author select drop-down.

Chapter 23: Custom Field Type

Go back to /admin/article/new and click to create a new article. Oh, duh! We're not logged in as an admin anymore. Log out, then log *back* in with admin2@thespacebar.com password engage. Cool. Try /admin/article/new again.

Now, open ArticleFormType so we can take a closer look at the field *types*. Right now, we're using TextType, this is TextareaType, this is a DateTimeType and the author drop-down is an EntityType. We learned earlier that the purpose of each field type is really two things. First: it controls how the field is rendered, like <input type="text">, <textarea>, <input type="datetime-local"> or a select drop down. The *second* purpose of a field type is more important: it determines how the field's data is transformed.

For example, the publishedAt field has a nice date widget that was added by my browser. But, really, this is just an input text field. What I mean is: the data from this field is submitted as a raw text string. But ultimately, on my Article entity, the setPublishedAt method requires a DateTime object! That's the job of the DateTimeType: to convert that specially-formatted date *string* into a DateTime object.

And *just* as important, it *also* transforms the other direction. Go to the list page and click to edit an existing, published article. Inspect the published at field. Yep! When the form loaded, the DateTimeType took the DateTime object from the Article and transformed it *back* into the string format that's used for the value attribute.

Custom Field for Author

Why are we talking about this? Because I want to *completely* replace this author dropdown, to avoid a future problem. Imagine if we had 10,000 users. Hmm, in that case, it wouldn't be very easy to find the person we want - that would be a *big* drop-down! Plus, querying for 10,000 users and rendering them would be *pretty* slow!

So, new plan: I want to convert this into a *text* field where I can type the author's email. That's... *easy*! We could use EmailType for that! But, there's a catch: when we submit, we need to create a data transformer that's able to take that email address string and query for the User object. Because, ultimately, when the form calls setAuthor(), the value needs to be a User object.

Creating the Custom Form Type

To do all of this, we're going to create our first, custom form field type. Oh, and it's really cool: it looks almost *identical* to the normal *form* classes that we've already been building.

Create a new class: let's call it UserSelectTextType. Make it extend that same AbstractType that we've been extending in our other form classes. Then, go to the Code + Generate menu, or Command + N on a Mac, and select override methods. But this time, instead of overriding buildForm(), override getParent(). Inside, return TextType::class. Well, actually, EmailType::class might be better: it will make it render as an <input type="email">, but either will work fine.

15 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 7
class UserSelectTextType extends AbstractType
{
    public function getParent()
    {
        return TextType::class;
    }
}
```

Internally, the form fields have an inheritance system. For, not-too-interesting technical reasons, the form classes don't use *real* class inheritance - we don't literally extend the TextType class. But, it works in a similar way.

By saying that TextType is our parent, we're saying that, unless we say otherwise, we want this field to look and behave like a normal TextType.

And... yea! We're basically set up. We're not doing anything *special* yet, but this should work! Go back over to

ArticleFormType. Remove *all* of this EntityType stuff and say UserSelectTextType::class.

Let's try it! Move over, refresh and... it actually works! It's a text field filled with the firstName of the current author.

But... it only works thanks to some luck. When this field is rendered, the author field is a User object. The <input type="text"> field needs a *string* that it can use for its value attribute. By *chance*, our User class has a __toString() method. And so, we get the first name!

But check this out: when we submit! Big, hairy, giant error:

> Expected argument of type User or null, string given

When that first name string is submitted, the TextType has *no* data transformer. And so, the form system ultimately calls setAuthor() and tries to pass it the *string* first name!

We'll fix this next with a data transformer.

Chapter 24: Data Transformer

We built a custom field type called UserSelectTextType and we're already using it for the author field. That's cool, except, thanks to getParent(), it's really just a TextType in disguise!

Internally, TextType basically has no data transformer: it takes whatever value is on the object and tries to print it as the value for the HTML input! For the author field, it means that it's trying to echo that property's value: an entire User object! Thanks to the __toString() method in that class, this prints the first name.

Let's remove that and see what happens. Refresh! Woohoo! A big ol' error:

> Object of class User could not be converted to string

More importantly, *even* if we put this back, yes, the form would render. But when we submitted it, we would just get a *different* huge error: the form would try to take the submitted *string* and pass *that* to setAuthor().

To fix this, our field needs a data transformer: something that's capable of taking the User object and rendering its email field. And on submit, transforming that email string back into a User object.

Creating the Data Transformer

Here's how it works: in the Form/ directory, create a new DataTransformer/ directory, but, as usual, the location of the new class won't matter. Then add a new class: EmailToUserTransformer.

The only rule for a data transformer is that it needs to implement a DataTransformerInterface. I'll go to the Code -> Generate menu, or Command+N on a Mac, select "Implement Methods" and choose the two from that interface.

I love data transformers! Let's add some debug code in each method so we can see when they're called and what this value looks like. So dd('transform', $value) and dd('reverse transform', $value).

19 lines | src/Form/DataTransformer/EmailToUserTransformer.php

```
... lines 1 - 7
class EmailToUserTransformer implements DataTransformerInterface
{
    public function transform($value)
    {
        dd('transform', $value);
    }

    public function reverseTransform($value)
    {
        dd('reverse transform', $value);
    }
}
```

To make UserSelectTextType use this, head back to that class, go to the Code -> Generate menu again, or Command + N on a Mac, and override one more method: buildForm().

Hey! We know this method! This is is the method that we override in our *normal* form type classes: it's where we add the fields! It turns out that there are a few *other* things that you can do with this $builder object: one of them is $builder->addModelTransformer(). Pass this a new EmailToUserTransformer().

22 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 9
class UserSelectTextType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->addModelTransformer(new EmailToUserTransformer());
    }
... lines 16 - 20
}
```

The transform() Method

Let's try it! I'll hit enter on the URL in my browser to re-render the form with a GET request. And... boom! We hit the transform() method! And the value is our User *object*.

This is awesome! That's the whole point of transform()! This method is called. when the form is *rendering*: it takes the raw data for a field - in our case the User object that lives on the author property - and our job is to transform that into a representation that can be used for the form field. In other words, the email string.

First, if null is the value, just return an empty string. Next, let's add a sanity check: if (!$value instanceof User), then *we*, the developer, are trying to do something crazy. Throw a new LogicException() that says:

> The UserSelectTextType can only be used with User objects.

Finally, at the bottom, so nice, return $value - which we now know is a User object ->getEmail().

28 lines | src/Form/DataTransformer/EmailToUserTransformer.php

```
... lines 1 - 8
class EmailToUserTransformer implements DataTransformerInterface
{
    public function transform($value)
    {
        if (null === $value) {
            return '';
        }

        if (!$value instanceof User) {
            throw new \LogicException('The UserSelectTextType can only be used with User objects');
        }

        return $value->getEmail();
    }
... lines 23 - 27
}
```

Let's rock! Move over, refresh and.... hello email address!

The reverseTransform() Method

Now, let's submit this. Boom! This time, we hit reverseTransform() and *its* data is the literal string email address. Our job is to use that to query for a User object and return it. And to do *that*, this class needs our UserRepository.

Time for some dependency injection! Add a constructor with UserRepository $userRepository. I'll hit alt+enter and select "Initialize Fields" to create that property and set it.

42 lines | src/Form/DataTransformer/EmailToUserTransformer.php

```
... lines 1 - 9
class EmailToUserTransformer implements DataTransformerInterface
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }
... lines 18 - 41
}
```

Normally... that's all we would need to do: we could instantly use that property below. But... this object is *not* instantiated by Symfony's container. So, we *don't* get our cool autowiring magic. Nope, in this case, *we* are creating this object ourselves! And so, *we* are responsible for passing it whatever it needs.

It's no big deal, but, we do have some more work. In the field type class, add an identical __construct() method with the same UserRepository argument. Hit Alt+Enter again to initialize that field. The form type classes *are* services, so autowiring *will* work here.

30 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 10
class UserSelectTextType extends AbstractType
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }
... lines 19 - 28
}
```

Thanks to that, in buildForm() pass $this->userRepository manually into EmailToUserTransformer.

30 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 19
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->addModelTransformer(new EmailToUserTransformer($this->userRepository));
    }
... lines 24 - 30
```

Back in reverseTransform(), let's get to work: $user = $this->userRepository and use the findOneBy() method to query for email set to $value. If there is *not* a user with that email, throw a new TransformationFailedException(). This is important - and its use statement was even pre-added when we implemented the interface. Inside, say:

> No user found with email %s

and pass the value. At the bottom, return $user.

42 lines | src/Form/DataTransformer/EmailToUserTransformer.php

```
... lines 1 - 9
class EmailToUserTransformer implements DataTransformerInterface
{
... lines 12 - 31
    public function reverseTransform($value)
    {
        $user = $this->userRepository->findOneBy(['email' => $value]);

        if (!$user) {
            throw new TransformationFailedException(sprintf('No user found with email "%s"', $value));
        }

        return $user;
    }
}
```

The TransformationFailedException is special: when this is thrown, it's a signal that there is a *validation* error.

Check it out: find your browser and refresh to resubmit that form. Cool - it *looks* like it worked. Try a different email: spacebar3@example.com and submit! Nice! If I click enter on the address to get a fresh load... yep! It *definitely* saved!

But now, try an email that does *not* exist, like spacebar300@example.com. Submit and... validation error! *That* comes from our data transformer. This TransformationFailedException causes a validation error. Not the type of validation errors that we get from our annotations - like @Assert\Email() or @NotBlank(). Nope: this is what I referred to early as "sanity" validation: validation that is built right into the form field itself.

We saw this in action back when we were using the EntityType for the author field: if we hacked the HTML and changed the value attribute of an option to a non-existent id, we got a sanity validation error message.

Next: let's see how we can customize this error and learn to do a few other fancy things to make our custom field more flexible.

Chapter 25: Custom Field: configureOptions() & Allowing Empty Input

Thanks to our data transformer - specifically the fact that it throws a TransformationFailedException when a bad email is entered - our UserSelectTextType has some built-in sanity validation!

But, the message we passed to the exception is *not* what's shown to the user. That's just internal. To control the message, well, we already know the answer! Add an invalid_message option when we create the field.

configureOptions(): Defining Field Options / Default

Or... instead of configuring that option when we're adding the specific field, we can give this option a default value for our custom field type. Open UserSelectTextType, go back to the Code -> Generate menu, or Command + N on a Mac, and this time, override configureOptions(). Inside, add $resolver->setDefaults() and give the invalid_message option a different default: "User not found".

38 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 11
class UserSelectTextType extends AbstractType
{
... lines 14 - 30
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'invalid_message' => 'Hmm, user not found!',
        ]);
    }
}
```

Try that out! Go back, refresh and... very nice!

And hey! We've seen this configureOptions() method before inside our normal form classes! When you're building an entire form, configureOptions() is used to set some options on your... whole form. There aren't very many common things to configure at that level.

But when you're creating a custom field type: configureOptions() is used to set the options for that specific *field*. We've just changed the default value for the invalid_message option. The *cool* thing is that this can *still* be overridden if we want: we could add an invalid_message option to the author field and *it* would win!

Fixing Empty Value Case in the Data Transformer

I want to talk more about field options because they can unlock some serious possibilities. But first, there is a *teenie*, tiny bug with our data transformer. Clear out the author text box and try to submit. Duh - disable HTML5 validation by adding the novalidate attribute. Hit update!

Oh! Our sanity validation *still* fails: User not found. That's not *quite* what we want. Instead of failing, our data transformer should probably just return null.

Go back to EmailToUserTransformer. In reverseTransform(), if $value is empty, just return. So, if the field is submitted empty, null should be passed to setAuthor().

But, hmm... the problem *now* is that, while it's *technically* ok to call setAuthor() with a null argument, we *want* that field to be required!

Re-submit the form! Oof - an integrity constraint violation: it's trying to save to the database with null set as the author_id column. We purposely made this required in the database and this is a *great* example of... messing up! We forget to add an important piece of business validation: to make the author required. No worries: open the Article class, find the $author field

and, above it, add @Assert\NotNull(). Give it a message: Please set an author.

271 lines | src/Entity/Article.php

```
... lines 1 - 17
class Article
{
... lines 20 - 71
    /**
... lines 73 - 74
     * @Assert\NotNull(message="Please set an author")
     */
    private $author;
... lines 78 - 269
}
```

Try that again. Excellent! *This* is the behavior - and error - we expect.

Next: how could we make our custom field type behave *differently* if it was used in different forms? Like, what if in one form, we want the user to be able to enter *any* user's email address but in *another* form we only want to allow the user to enter the email address of an admin user. Let's learn more about the power of form field options.

Chapter 26: Leveraging Custom Field Options

Our UserSelectTextType field work great! I've been high-fiving people all day about this! But now, imagine that you want to use this field on multiple forms in your app. That part is easy. Here's the catch: on some forms, we want to allow the email address of *any* user to be entered. But on *other* forms, we need to use a custom query: we only want to allow *some* users to be entered - maybe only admin users.

To make this possible, our field needs to be more flexible: instead of looking for *any* User with this email, we need to be able to *customize* this query each time we use the field.

Adding a finderCallback Option

Let's start inside the transformer first. How about this: add a new argument to the constructor a callable argument called $finderCallback. Hit the normal Alt+Enter to create that property and set it.

49 lines | src/Form/DataTransformer/EmailToUserTransformer.php

```
... lines 1 - 9
class EmailToUserTransformer implements DataTransformerInterface
{
... line 12
    private $finderCallback;
... line 14
    public function __construct(UserRepository $userRepository, callable $finderCallback)
    {
... line 17
        $this->finderCallback = $finderCallback;
    }
... lines 20 - 48
}
```

Here's the idea: whoever instantiates this transformer will pass in a callback that's responsible for querying for the User. Down below, instead of fetching it directly, say $callback = $this->finderCallback and then, $user = $callback(). For convenience, let's pass the function $this->userRepository. And of course, it will need the $value that was just submitted.

49 lines | src/Form/DataTransformer/EmailToUserTransformer.php

```
... lines 1 - 33
    public function reverseTransform($value)
    {
... lines 36 - 39
        $callback = $this->finderCallback;
        $user = $callback($this->userRepository, $value);
... lines 42 - 47
    }
```

Cool! We've now made this class a little bit more flexible. But, that doesn't *really* help us yet. How can we allow this $finderCallback to be customized each time we use this field? By creating a brand new field *option*.

Check this out: we know that invalid_message is *already* an option in Symfony and we're changing its default value. But, we can invent *new* options too! Add a new option called finder_callback and give it a default value: a callback that accepts a UserRepository $userRepository argument and the value - which will be a string $email. Inside return the normal $userRepository->findOneBy() with ['email' => $email].

44 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 11
class UserSelectTextType extends AbstractType
{
... lines 14 - 33
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
... line 37
            'finder_callback' => function(UserRepository $userRepository, string $email) {
                return $userRepository->findOneBy(['email' => $email]);
            }
        ]);
    }
}
```

Next, check out the buildForm() method. See this array of $options? That will *now* include finder_callback, which will either be our default value, or some other callback if it was overridden.

Let's break this onto multiple lines and, for the second argument to EmailToUserTransformer, pass $options['finder_callback'].

44 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 20
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->addModelTransformer(new EmailToUserTransformer(
            $this->userRepository,
            $options['finder_callback']
        ));
    }
... lines 28 - 44
```

Ok! Let's make sure it works. I'll hit enter on the URL to reload the page. Then, change to spacebar2@example.com, submit and... yes! It saves!

The *real* power of this is that, in ArticleFormType, when we use UserSelectTextType, we can pass a finder_callback option if we need to do a custom query. *If* we did that, it would override the default value and, when we instantiate EmailToUserTransformer, the second argument would be the callback that *we* passed from ArticleFormType.

Investigating the Core Field Types

This is how options are used internally by the core Symfony types. Oh, and you probably noticed by now that *every* field type in Symfony is represented by a normal, PHP class! If you've ever want to know more about how a specific field or option works, just open up the class!

For example, we know that this field is a DateTimeType. Press Shift+Shift and look for DateTimeType - open the one from the Form component. I love it - these classes will look a lot like our own custom field type class! This one has a buildForm() method that adds some transformers. And if you scroll down far enough, cool! Here is the configureOptions() method where *all* of the valid options are defined for this field.

Want to know how one of these options is used? Copy its name and find out! Search for the with_seconds option. No surprise: it's used in buildForm(). If you looked a little further, you'd see that this is eventually used to configure how the data transformer works.

These core classes are a great way to figure out how to do something advanced or to get inspiration for your own custom field type. Don't' be afraid to dig!

Next: let's hook up some auto-complete JavaScript to this field.

Chapter 27: Autocomplete JavaScript

From a backend perspective, the custom field is done! When the user submits a string email address, the data transformer turns that into the proper User object, *with* built-in validation.

But from a *frontend* perspective, it could use some help. It would be *way* more awesome if this field had some cool JavaScript auto-completion magic where it suggested valid emails as I typed. So... let's do it!

Google for "Algolia autocomplete". There are a lot of autocomplete libraries, and this one is pretty nice. Click into their documentation and then to the GitHub page for autocomplete.js.

Many of you might know that Symfony comes with a great a JavaScript tool called Webpack Encore, which helps you create organized JavaScript and build it all into compiled files. We have *not* been using Encore in this tutorial yet. So I'm going to keep things simple and continue without it. Don't worry: the most *important* part of what we're about to do is the same no matter what: it's how you connect custom JavaScript to your form fields.

Adding the autocomplete.js JavaScript

Copy the script tag for jQuery, open templates/article_admin/edit.html.twig and override {% block javascripts %} and {% endblock %}. Call the {{ parent() }} function to keep rendering the parent JavaScript. Then paste in that new <script> tag.

```
23 lines | templates/article_admin/edit.html.twig
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script src="https://cdn.jsdelivr.net/autocomplete.js/0/autocomplete.jquery.min.js"></script>
... line 7
{% endblock %}
... lines 9 - 23
```

Yes, we *are* also going to need to do this in the new template. We'll take care of that in a little bit.

Now, if you scroll down a little on their docs... there it is! This page has some CSS that helps make all of this look good. Copy that, go to the public/css directory, and create a new file: algolia-autocomplete.css. Paste this there.

Include this file in our template as well: override {% block stylesheets %} and {% endblock %}. This time add a <link> tag that points to that file: algolia-autocomplete.css. Oh, and don't forget the parent() call - I'll add that in a second.

```
23 lines | templates/article_admin/edit.html.twig
... lines 1 - 9
{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="{{ asset('css/algolia-autocomplete.css') }}">
{% endblock %}
... lines 15 - 23
```

Finally, for the custom JavaScript logic, in the js/ directory, create a new file called algolia-autocomplete.js. Before I fill *anything* in here, include that in the template: a <script> tag pointing to js/algolia-autocomplete.js.

23 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 6
    <script src="{{ asset('js/algolia-autocomplete.js') }}"></script>
{% endblock %}
... lines 9 - 23
```

Implementing autocomplete.js

Initial setup done! Head back to their documentation to find where it talks about how to use this with jQuery. It looks *kinda* simple: select an element, call .autcomplete() on it, then... pass a ton of options that tell it how to fetch and process the autocomplete data.

Cool! Let's do something similar! I'll start with the document.ready() block from jQuery just to make sure the DOM is fully loaded. Now: here is the key moment: how can we write JavaScript that can *connect* to our custom field? Should we select it by the id? Something else?

I like to select with a *class*. Find all elements with, how about, some .js-user-autocomplete class. Nothing has this class yet, but our field will soon. Call .autocomplete() on this, pass it that same hint: false and then an array. This looks a bit complex: add a JavaScript object with a source option set to a function() that receives a query argument and a callback cb argument.

Basically, as we're typing in the text field, the library will call this function and pass whatever we've entered into the text box so far as the query argument. *Our* job is to determine which results match this "query" text and pass those back by calling the cb function.

To start... let's hardcode something and see if it works! Call cb() and pass it an array where each entry is an object with a value key... because that's how the library wants the data to be structured by default.

13 lines | public/js/algolia-autocomplete.js

```
$(document).ready(function() {
    $('.js-user-autocomplete').autocomplete({hint: false}, [
        {
            source: function(query, cb) {
                cb([
                    {value: 'foo'},
                    {value: 'bar'}
                ])
            }
        }
    ]);
});
```

Thanks to my imaginative code, no matter *what* we type, foo and bar *should* be suggested.

Adding the js- Class to the Field

And... we're almost... sorta done! In order for this to be applied to our field, all *we* need to do is add this class to the author field. No problem! Copy the class name and open UserSelectTextType. Here, we can set a *default* value for the attr option to an array with class set to js-user-autocomplete.

47 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 11
class UserSelectTextType extends AbstractType
{
... lines 14 - 33
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
... lines 37 - 40
            'attr' => [
                'class' => 'js-user-autocomplete'
            ]
        ]);
    }
}
```

Field Options vs View Variables

Up until now, if we've wanted to add a class attribute, we've done it from inside of our Twig template. For example, open security/register.html.twig. For the form start tag, we're passing an attr variable with a class key. Or, for the fields, we're adding a placeholder attribute.

attr is one of a few things that can be passed either as a view variable or *also* as a field *option*. But, I want to be clear: options and variables are *two* different things. Go back and open the profiler. Click on, how about, the author field. We know that there is a set of options that we can pass to the field from inside the form class. And then, when you're rendering in your template, there is a *different* set of view variables. These are two *different* concepts. However, there *is* some overlap, like attr.

Behind the scenes, when you pass the attr option, that simply becomes the default value for the attr view variable. The attr option, just like the label and help options - exists *just* for the added convenience of being able to set these in your form class *or* in your template.

Anyways, thanks to the code in UserSelectTextType, our field *should* have this class. Let's try it! Close the profiler, refresh and... ah! I killed my page! The CSS is gone! I *always* do that! Go back to the template and add the missing parent() call: I don't want to completely *replace* the CSS from our layout.

Ok, try it again. Much better. And when we type into the field... yes! We get foo and bar no matter what we type. Awesome!

Next, hey: I like foo and bar as much as the next programmer. But we should probably make an AJAX call to fetch a *true* list of matching email addresses.

Chapter 28: Autocomplete Endpoint & Serialization Group

To get our autocomplete fully working, we need an API endpoint that returns a list of user information - specifically user *email* addresses. We can do that! Create a new controller for this: AdminUtilityController. Make that extend the normal AbstractController and add a public function getUsersApi(). To make this a real page, add @Route("/admin/utility/users"). And, just to be extra fancy, let's also add methods="GET".

24 lines | src/Controller/AdminUtilityController.php

```
... lines 1 - 10
class AdminUtilityController extends AbstractController
{
    /**
     * @Route("/admin/utility/users", methods="GET")
     */
    public function getUsersApi(UserRepository $userRepository)
    {
... lines 18 - 22
    }
}
```

The *job* of this endpoint is pretty simple: return an array of User objects as JSON: I'm not even going to worry about filtering them by a search term yet.

Add the UserRepository $userRepository argument and fetch every user with $users = $userRepository->findAllEmailAlphabetical(). Finish this with return $this->json() and, it doesn't really matter, but let's set the user objects into a users key.

24 lines | src/Controller/AdminUtilityController.php

```
... lines 1 - 15
    public function getUsersApi(UserRepository $userRepository)
    {
        $users = $userRepository->findAllEmailAlphabetical();
... lines 19 - 22
    }
```

Cool! Copy that URL, open a new tab paste and.... boo! A circular reference has been detected. This is a common problem with the serializer and Doctrine objects. Check it out: open the User class. By default, the serializer will serialize *every* property... or more accurately, every property that has a getter method.

Serialization Groups to the Rescue

But *that* means that it's serializing the apiTokens property. And, well, when it tries to serialize that, it notices its user property and so, tries to serialize the User object. You can see the problem. Eventually, before our CPU causes our computer fan to quit & our motherboard to catch on fire, the serializer notices this loop and throws this exception.

What's the fix? Well, the thing is, we don't really want to serialize *all* of the fields anyway! We really only need the email, but we could also just serialize the same basic fields that we serialized earlier.

Remember: in AccountController, we created an API endpoint that returns one User object. When we did that, we told the serializer to only serialize the groups called main. Look back in the User class. Ah, yes: we used the @Groups() annotation to "categorize" the fields we wanted into a group called main.

In AdminUtilityController, we can serialize that *same* group. Pass 200 as the second argument - this is the status code - we don't need any custom headers, but we *do* want to pass a groups option set to main... I know a lot of square brackets to do this.

24 lines | src/Controller/AdminUtilityController.php

```
... lines 1 - 15
    public function getUsersApi(UserRepository $userRepository)
    {
... lines 18 - 19
        return $this->json([
            'users' => $users
        ], 200, [], ['groups' => ['main']]);
    }
```

Now go back and refresh. Got it! We could add a *new* serialization group to return even *less* - like maybe just the email. It's up to you.

Adding Security

But no matter *what* we do, we probably need to make sure this endpoint is secure: we don't want *anyone* to be able to search our user database. But... hmm.. this is tricky. In ArticleAdminController, the new() endpoint requires ROLE_ADMIN_ARTICLE.

Copy that role, go back to AdminUtilityController and, above the method, add @IsGranted() and paste to use the same role.

25 lines | src/Controller/AdminUtilityController.php

```
... lines 1 - 12
    /**
... line 14
     * @IsGranted("ROLE_ADMIN_ARTICLE")
     */
    public function getUsersApi(UserRepository $userRepository)
... lines 18 - 25
```

This is a *little* weird because, in ArticleAdminController, the edit endpoint is protected by a custom voter that allows access if you have that same ROLE_ADMIN_ARTICLE role *or* if you are the author of this article. In other words, it's possible that an author could be editing their article, but the AJAX call to our new endpoint would *fail* because they don't have that role!

I wanted to point this out, but we won't need to fix it because, later, we're going to *disable* this field on the edit form anyways. In other words: we will eventually *force* the author to be set at the moment an article is created.

Next: let's finish this! Let's hook up our JavaScript to talk to the new API endpoint and *then* make it able to filter the user list based on the user's input.

Chapter 29: Hooking up the AJAX Autocomplete

We now have an endpoint that returns all users as JSON. *And* we have some autocomplete JavaScript that... ya know... autocompletes entries for us. I have a crazy idea: let's *combine* these two so that our autocomplete uses that Ajax endpoint!

Adding a data-autocomplete-url Attribute

First: inside of the JavaScript, we need to know what the URL is to this endpoint. We *could* hardcode this - I wouldn't judge you for doing that - this is a no-judgment zone. But, there *is* a simple, clean solution.

In AdminUtilityController, let's give our new route a name: admin_utility_users. Now, idea time: when we render the field, what if we added a "data" attribute onto the input field that pointed to this URL? If we did that, it would be *super* easy to read that from JavaScript.

25 lines | src/Controller/AdminUtilityController.php

```
... lines 1 - 10
class AdminUtilityController extends AbstractController
{
    /**
     * @Route("/admin/utility/users", methods="GET", name="admin_utility_users")
... line 15
     */
    public function getUsersApi(UserRepository $userRepository)
... lines 18 - 24
}
```

Let's do it! In UserSelectTextType, add another attribute: how about data-autocomplete-url set to... hmm. We need to *generate* the URL to our new route. How do we generate a URL from inside of a service? Answer: by using the router service. Add a second argument to the constructor: RouterInterface $router. I'll hit Alt+Enter to add that property and set it.

51 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 12
class UserSelectTextType extends AbstractType
{
... line 15
    private $router;
... line 17
    public function __construct(UserRepository $userRepository, RouterInterface $router)
    {
... line 20
        $this->router = $router;
    }
... lines 23 - 49
}
```

Oh, and if you can't remember the type-hint to use, at least make sure that you remember that you can run:

```
$ php bin/console debug:autowiring
```

to see a full list of type-hints. By the way, in Symfony 4.2, this output will look a little bit different, but contains the same info. If you search for the word "route" without the e... cool! We have a few different type-hints, but they all return the same service anyways.

Now that we've injected the router, down below, use $this->router->generate() and pass it the new route name: admin_utility_users.

```
51 lines | src/Form/UserSelectTextType.php
... lines 1 - 36
37      public function configureOptions(OptionsResolver $resolver)
38      {
39          $resolver->setDefaults([
... lines 40 - 43
44              'attr' => [
... line 45
46                  'data-autocomplete-url' => $this->router->generate('admin_utility_users')
47              ]
48          ]);
49      }
... lines 50 - 51
```

Let's check it out! Refresh, inspect that field and ... perfect! We have a shiny new data-autocomplete-url attribute.

Making the AJAX Call

Let's head to our JavaScript! I'm going to write this a little bit different - though it would work either way: let's find all of the elements... there will be just one in this case... and loop over them with .each(). Indent the inner code, then close the extra function.

```
20 lines | public/js/algolia-autocomplete.js
1   $(document).ready(function() {
2       $('.js-user-autocomplete').each(function() {
... lines 3 - 17
18      });
19  });
```

Now we can change the selector to this and... yea! We're basically doing the same thing as before. Inside the loop, fetch the URL with var autocompleteUrl = $(this).data() to read that new attribute.

```
20 lines | public/js/algolia-autocomplete.js
... line 1
2       $('.js-user-autocomplete').each(function() {
3           var autocompleteUrl = $(this).data('autocomplete-url');
... lines 4 - 17
18      });
... lines 19 - 20
```

Finally, clear out the source attribute. Since we're using jQuery already, let's use it to make the AJAX call: $.ajax() with a url option set to autocompleteUrl. That's it!

To handle the result, chain a .then() onto the Promise and pass a callback with a data argument. Let's see: our job is to execute the cb callback and pass it an array of the results.

Remember: in the controller, I'm returning all the user information on a users key. So, let's return data.users: that should return this entire array of data.

20 lines | public/js/algolia-autocomplete.js

```
... lines 1 - 4
        $(this).autocomplete({hint: false}, [
            {
                source: function(query, cb) {
                    $.ajax({
                        url: autocompleteUrl
                    }).then(function(data) {
                        cb(data.users);
                    });
                },
... lines 14 - 15
            }
        ])
... lines 18 - 20
```

But *also* remember that, by default, the autocomplete library expects each result to have a value key that it uses. Obviously, *our* key is called email. To change that behavior, add displayKey: 'email'. I'll also add debounce: 500 - that will make sure that we don't make AJAX requests faster than once per half a second.

20 lines | public/js/algolia-autocomplete.js

```
... lines 1 - 4
        $(this).autocomplete({hint: false}, [
            {
... lines 7 - 13
                displayKey: 'email',
                debounce: 500 // only request every 1/2 second
            }
        ])
... lines 18 - 20
```

Ok... I think we're ready! Let's try this! Move back to your browser, refresh the page and clear out the author field... "spac"... we got it! Though... it *still* returns *all* of the users - the geordi users should not match.

Filtering the Users

That's no surprise: our endpoint *always* returns *every* user. No worries - this is the easiest part! Go back to the JavaScript. The source function is passed a query argument: that's equal to whatever is typed into the input box at that moment. Let's use that! Add a '?query='+query to the URL.

20 lines | public/js/algolia-autocomplete.js

```
... lines 1 - 6
                source: function(query, cb) {
                    $.ajax({
                        url: autocompleteUrl+'?query='+query
... lines 10 - 11
                    });
                },
... lines 14 - 20
```

Back in AdminUtilityController, to read that, add a second argument, the Request object from HttpFoundation. Then, let's call a new method on UserRepository, how about findAllMatching(). Pass this the ?query= GET parameter by calling $request->query->get('query').

26 lines | src/Controller/AdminUtilityController.php

```
... lines 1 - 8
use Symfony\Component\HttpFoundation\Request;
... lines 10 - 11
class AdminUtilityController extends AbstractController
{
... lines 14 - 17
    public function getUsersApi(UserRepository $userRepository, Request $request)
    {
        $users = $userRepository->findAllMatching($request->query->get('query'));
... lines 21 - 24
    }
}
```

Nice! Copy the method name and then open src/Repository/UserRepository.php. Add the new public function findAllMatching() and give it a string $query argument. Let's also add an optional int $limit = 5 argument, because we probably shouldn't return 1000 users if 1000 users match the query. Advertise that this will return an array of User objects.

76 lines | src/Repository/UserRepository.php

```
... lines 1 - 14
class UserRepository extends ServiceEntityRepository
{
... lines 17 - 33
    /**
     * @return User[]
     */
    public function findAllMatching(string $query, int $limit = 5)
    {
... lines 39 - 44
    }
... lines 46 - 74
}
```

Inside, it's pretty simple: return $this->createQueryBuilder('u'), ->andWhere('u.email LIKE :query') and bind that with ->setParameter('query') and, this is a little weird, '%'.$query.'%'.

Finish with ->setMaxResults($limit), ->getQuery() and ->getResult().

76 lines | src/Repository/UserRepository.php

```
... lines 1 - 36
    public function findAllMatching(string $query, int $limit = 5)
    {
        return $this->createQueryBuilder('u')
            ->andWhere('u.email LIKE :query')
            ->setParameter('query', '%'.$query.'%')
            ->setMaxResults($limit)
            ->getQuery()
            ->getResult();
    }
... lines 46 - 76
```

Done! Unless I've *totally* mucked things up, I think we should have a working autocomplete setup! Refresh to get the new JavaScript, type "spac" and... woohoo! Only 5 results! Let's get the web debug toolbar out of the way. I love it!

Next: there's one other important method you can override in your custom form field type class to control how it renders. We'll use it to *absolutely* make sure our autocomplete field has the HTML attributes it needs, *even* if we override the attr option

when using the field.

Chapter 30: The buildView() Method

The autocomplete setup works nicely on the edit page. But, if you click to create an article... it *looks* like it's working, but it's not! This is just the normal autocomplete from my browser.

There's no JavaScript error and we *do* have the class and the data- attribute. We expected this: we just... haven't added the JavaScript to this page!

In edit.html.twig, the javascripts and stylesheets blocks bring in the magic. Let's solve this in the simplest way possible. Copy both of these blocks. Open new.html.twig and paste! Oh, and I mentioned earlier, that we're going to eventually tweak things so that the author field is *only* filled in on *create*: we're going to disable it on edit.

23 lines | templates/article_admin/new.html.twig

```
... lines 1 - 2
{% block javascripts %}
    {{ parent() }}

    <script src="https://cdn.jsdelivr.net/autocomplete.js/0/autocomplete.jquery.min.js"></script>
    <script src="{{ asset('js/algolia-autocomplete.js') }}"></script>
{% endblock %}

{% block stylesheets %}
    {{ parent() }}

    <link rel="stylesheet" href="{{ asset('css/algolia-autocomplete.css') }}">
{% endblock %}
... lines 15 - 23
```

That means... we *won't* need any of this stuff on the edit page. Let's delete it now. But, if you *did* need some JavaScript and CSS on both templates and you did *not* want to duplicate the blocks, you could create a new template, like article_admin_base.html.twig. *It* would extend content_base.html.twig and include the javascripts and stylesheets blocks. Then, edit.html.twig and new.html.twig would extend this.

Anyways, now that the JavaScript and CSS live in the new template, when we refresh, we have autocomplete.

The buildView() Form Class Method

Before we move on, I have one more cool thing I want to show you! And, it solves a *real* problem... just not a problem we realize we had yet. Close a few files then go to UserSelectTextType. The whole autocomplete system works because we are setting the attr option with class and data-autocomplete-url keys. Now open ArticleFormType where we use this field type. One of the things that we're *allowed* to do here is *override* that attr option. But, if we did that, our custom attr option would completely *replace* the attr default from the type class! In other words, we would lose all of the special attributes that we need!

To fix this, at the bottom of UserSelectTextType, go to the Code -> Generate menu, or command+N on a Mac, select Override methods and choose buildView(). Oh, there's also a method called finishView() and its purpose is *almost* identical to buildView() - it's just called a bit later.

60 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 14
class UserSelectTextType extends AbstractType
{
... lines 17 - 48
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
... lines 51 - 57
    }
}
```

Here's what's going on: to render each field, Symfony creates a bunch of *variables* that are used in the form theme system. We already knew that: in register.html.twig we're overriding the attr variable. And in our form theme blocks, we use different variables to do our work.

And, of course, we know that, thanks to the profiler, we can see the *exact* view variables that exist for *each* field. But... where do these variables come from? For example, why does each field have a full_name variable? Who added that?

The answer is buildView(): Symfony calls this method on *every* field, and it is *the* place where these variables are created and can be changed.

We do that with this $view variable, which is kind of a strange object. Start with $attr = $view->vars['attr'];. This $view object has a public ->vars array property that holds *all* of the things that will eventually become the "variables". At this moment, the core form system has *already* set this variable up for us: it will either be equal to the attr option passed for this field, or an empty array.

60 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 48
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        $attr = $view->vars['attr'];
... lines 52 - 57
    }
... lines 59 - 60
```

Next: grab the class: if class is set on $attr, use it, but add a space on the end. If there is no class yet, set this to be blank. Now, here's the key: let's *always* append js-user-autocomplete: that's the class we're using above. Call $attr['class'] = to set the new class string back on.

60 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 48
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
... line 51
        $class = isset($attr['class']) ? $attr['class'].' ' : '';
        $class .= 'js-user-autocomplete';

        $attr['class'] = $class;
... lines 56 - 57
    }
... lines 59 - 60
```

Oh, and we *also* need to add the data-autocomplete-url attribute. Copy that from above and say $attr['data-autocomplete-url'] equals the generated URL. Perfect! *Finally*, set *all* of this back onto the view object with $view->vars['attr'] = $attr.

60 lines | src/Form/UserSelectTextType.php

```
... lines 1 - 48
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
... lines 51 - 55
        $attr['data-autocomplete-url'] = $this->router->generate('admin_utility_users');
        $view->vars['attr'] = $attr;
    }
... lines 59 - 60
```

Phew! We're done! Now that we're setting the attr variable directly, we don't need to set the option anymore. And the *best* part is that we know our attributes will be rendered no matter *what* the user passes to the attr option.

Let's try it! Move over, refresh and cool! Nice work team! The element still has the attributes we need.

Oh, and open the profiler for this form. Click on the author field and check out the View Variables. So cool! That's *exactly* what we set!

Next: the form component has a *crazy* powerful plugin system. Want to make some tweak to *every* form or even every *field* in your entire app? That's possible, and it's fun!

Chapter 31: Form Type Extension

Symfony's form system has a feature that gives us the *massive* power to modify *any* form or *any* field across our *entire* app! Woh! It's called "form type extensions" and working with them is *super* fun.

To see how this works, let's talk about the textarea field. Forget about Symfony for a moment. In HTML land, one of the features of the textarea element is that you can give it a rows attribute. If you set rows="10", it gets longer.

If we wanted to set that attribute in Symfony, we could, of course, pass an attr option with rows set to some value. But, here's the *real* question: could we automatically set that option for *every* textarea across our entire app? Absolutely! We can do anything!

Creating the Form Type Extension

In your Form/ directory, create a new directory called TypeExtension, then inside a class called TextareaSizeExtension. Make this implement FormTypeExtensionInterface. As the name implies, this will allow us to *extend* existing form types.

38 lines | src/Form/TypeExtension/TextareaSizeExtension.php

```
... lines 1 - 6
use Symfony\Component\Form\FormTypeExtensionInterface;
... lines 8 - 10
class TextareaSizeExtension implements FormTypeExtensionInterface
{
... lines 13 - 36
}
```

Next, go to the Code -> Generate menu, or Command+N on a Mac, and choose "Implement Methods" to implement everything we need. Woh! We know these methods! These are almost the *exact* same methods that we've been implementing in our form type classes! And... that's on purpose! These methods work pretty much the same way.

38 lines | src/Form/TypeExtension/TextareaSizeExtension.php

```
... lines 1 - 10
class TextareaSizeExtension implements FormTypeExtensionInterface
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // TODO: Implement buildForm() method.
    }

    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        // TODO: Implement buildView() method.
    }

    public function finishView(FormView $view, FormInterface $form, array $options)
    {
        // TODO: Implement finishView() method.
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        // TODO: Implement configureOptions() method.
    }

    public function getExtendedType()
    {
        // TODO: Implement getExtendedType() method.
    }
}
```

Registering the Form Type Extension

The only *new* method is getExtendedType() - we'll talk about that in a second. To tell Symfony that this form type extension exists *and* to tell it that we want to extend the TextareaType, we need a little bit of config. This might look confusing at first. Let's code it up, then I'll explain.

Open config/services.yaml. And, at the bottom, we need to give our service a "tag". First, put the form class and below, add tags. The syntax here is a bit ugly: add a dash, open an array and set name to form.type_extension. Then I'll create a new line for my own sanity and add one more option extended_type. We need to set this to the form type class that we want to extend - so TextareaType. Let's cheat real quick: I'll use TextareaType, auto-complete that, copy the class, then delete that. Go paste it in the config. Oh, and I forgot my comma!

43 lines | config/services.yaml

```
... lines 1 - 6
services:
... lines 8 - 38
    App\Form\TypeExtension\TextareaSizeExtension:
        tags:
            - { name: form.type_extension,
                extended_type: Symfony\Component\Form\Extension\Core\Type\TextareaType }
```

As *soon* as we do this, every time a TextareaType is created in the system, *every* method on our TextareaSizeExtension will be called. It's almost as if each of these methods *actually* lives *inside* of the TextareaType class! If we add some code to buildForm(), it's pretty much identical to opening up the TextareaType class and adding code right there!

The form.type_extension Tag & autoconfigure

Now, two important things. If you're using Symfony 4.2, then you do *not* need to add *any* of this code in services.yaml. Whenever you need to "plug into" some part of Symfony, internally, you do that by registering a service and giving it a "tag". The form.type_extension tag says:

> Hey Symfony! This isn't just a normal service! It's a form type extension! So make sure you use it for that!

But these days, you don't see "tags" much in Symfony. The reason is simple: for most things, Symfony looks at the interfaces that your service implements, and adds the correct tags automatically. In Symfony 4.1 and earlier, this does *not* happen for the FormTypeExtensionInterface. But in Symfony 4.2... it does! So, no config needed... at all.

But then, how does Symfony know *which* form type we want to extend in Symfony 4.2? The getExtendedType() method! Inside, return TextareaType::class. And yea, we *also* need to fill in this method in Symfony 4.1... it's a bit redundant, which is why Symfony 4.2 will be *so* much cooler.

Tip

Since Symfony 4.2 getExtendedType() method is deprecated in favor of getExtendedTypes() but you still need a dummy implementation of getExtendedType()

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

public static function getExtendedTypes(): iterable
{
    return [SomeType::class];
}
```

36 lines | src/Form/TypeExtension/TextareaSizeExtension.php

```
... lines 1 - 11
class TextareaSizeExtension implements FormTypeExtensionInterface
... lines 13 - 30
    public function getExtendedType()
    {
        return TextareaType::class;
    }
}
```

Filling in the Form Type Extension

Ok! Let's remove the rest of the TODOs in here and then get to work! We can fill in whichever methods we need. In our case, we want to modify the *view* variables. That's easy for us: in buildView(), say $view->vars['attr'], and then add a rows attribute equal to 10.

36 lines | src/Form/TypeExtension/TextareaSizeExtension.php

```
... lines 1 - 17
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        $view->vars['attr']['rows'] = 10;
    }
... lines 22 - 36
```

Done! Move over, refresh and... yea! I think it's bigger! Inspect it - yes: rows="10". *Every* <textarea> on our *entire* site will now have this.

Modifying "Every" Field?

By the way, instead of modifying just one field type, *sometimes* you may want to modify literally *every* field type. To do that, you can choose to extend FormType::class. That works because of the form field inheritance system. All field types ultimately extend FormType::class, except for a ButtonType that I don't usually use anyways. So if you override FormType, you can

modify everything. Just keep in mind that this will *also* include your entire *form* classes, like ArticleFormType.

Adding a new Field Option

But wait, there's more! Instead of hardcoding 10, could we make it possible to configure this value *each* time you use the TextareaType? Why, of course! In ArticleFormType, pass null to the content field so it keeps guessing it. Then add a new option: rows set to 15.

47 lines | src/Form/ArticleFormType.php

```
... lines 1 - 14
class ArticleFormType extends AbstractType
{
... lines 17 - 23
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 27 - 29
            ->add('content', null, [
                'rows' => 15
            ])
... lines 33 - 36
        ;
    }
... lines 39 - 45
}
```

Try this out - refresh! Giant error!

> The option "rows" does not exist

It turns out that you can't just "invent" new options and pass them: each field has a concrete set of valid options. But, in TextareaSizeExtension, we *can* invent new options. Do it down in configureOptions(): add $resolver->setDefaults() and invent a new rows option with a default value of 10.

39 lines | src/Form/TypeExtension/TextareaSizeExtension.php

```
... lines 1 - 11
class TextareaSizeExtension implements FormTypeExtensionInterface
{
... lines 14 - 26
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'rows' => 10
        ]);
    }
... lines 33 - 37
}
```

Now, up in buildView(), notice that almost every method is passed the final array of $options for this field. Set the rows attribute to $options['rows'].

39 lines | src/Form/TypeExtension/TextareaSizeExtension.php

```
... lines 1 - 17
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        $view->vars['attr']['rows'] = $options['rows'];
    }
... lines 22 - 39
```

Done. The rows will default to 10, but we can override that via a brand, new shiny form field option. Try it! Refresh, inspect the textarea and... yes! The rows attribute is set to 15.

How CSRF Protection Works

This is the power of form type extensions. And these are *even* used in the core of Symfony to do some cool stuff. For example, remember how every form automatically has an _token CSRF token field? How does Symfony magically add that? The answer: a form type extension. Press Shift+Shift and look for a class called FormTypeCsrfExtension.

Cool! It extends an AbstractTypeExtension class, which implements the same FormTypeExtensionInterface but prevents you from needing to override *every* method. We also could have used this same class.

Anyways, in buildForm() it adds an "event listener", which activates some code that will *validate* the _token field when we submit. We'll talk about events in a little while.

In finishView() - which is very similar to buildView() - it adds a few variables to help render that hidden field. And finally, in configureOptions(), it adds some options that allow us to control things. For example, inside the configureOptions() method of any form class - like ArticleFormType - we could set a csrf_protection option to false to disable the CSRF token.

Next: how could we make our form look or act differently based on the *data* passed to it? Like, how could we make the author field *disabled*, only on the edit form? Let's find out!

Chapter 32: Tweak your Form based on the Underlying Data

New goal team! Remember this author field? It's where we added all this nice auto-complete magic. I want this field to be *fully* functional on the "new form", but *disabled* on the edit form: as *wonderful* as they are, some of our alien authors get nervous and sometimes try to change an article to look like it was written by someone else.

This is the first time that we want the *same* form to behave in two different ways, based on *where* it is used.

Let's see: on our new endpoint, the form creates the new Article object behind the scenes for us. But on the edit page, the form is *modifying* an *existing* Article: we pass this *to* the form.

So, hmm, in the buildForm() method of our form class, if we could get *access* to the data that was passed to the form - either the existing Article object or maybe nothing - then we could use that info to build the fields differently.

Accessing Data via $options

Fortunately... that's *easy*. The secret is the $options argument that's passed to us. Let's see what this looks like: dd($options) and then go back and refresh the edit page.

48 lines | src/Form/ArticleFormType.php

```
... lines 1 - 14
class ArticleFormType extends AbstractType
{
... lines 17 - 23
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        dd($options);
... lines 27 - 38
    }
... lines 40 - 46
}
```

Wow! There are a *ton* of options. And *all* of these are things that we could configure down in configureOptions(). But, the majority of this stuff isn't all that important. However, there is *one* super-helpful key: data. It's set to our Article object! Bingo!

Now, open another tab and go to /admin/article/new.

Oh. This time there is *no* data... which makes sense because we never passed anything to the form. That's great! We can use the data key to get the underlying data. How about: $article = $options['data'] ?? null;

49 lines | src/Form/ArticleFormType.php

```
... lines 1 - 23
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $article = $options['data'] ?? null;
        dd($article);
... lines 28 - 39
    }
... lines 41 - 49
```

If you don't know that syntax, it basically says that I want the $article variable to be equal to $options['data'] if it *exists* and is not null. But if it does *not* exist, set it to null. Let's dump that and make sure it's what we expect.

Refresh the new article page - yep - null. Try the edit page... there's the Article object. *Now*, we are dangerous. Remove the dd() and create a new variable: $isEdit = $article && $article->getId().

52 lines | src/Form/ArticleFormType.php

```
... lines 1 - 23
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $article = $options['data'] ?? null;
        $isEdit = $article && $article->getId();
... lines 28 - 42
    }
... lines 44 - 52
```

You *might* think that it's enough just to check whether $article is an object. But actually, on our new endpoint, if we wanted, we *could* instantiate a new Article() object and pass *it* as the second argument to createForm(). You do this sometimes if you want to pre-fill a "new" form with some default data. The form system would *update* that Article object, but Doctrine would still be smart enough to insert a new row when we save.

Anyways, *that's* why I'm checking not only that the Article is an object, but that it also has an id.

Dynamically disabling a Field

This is great, because, our goal was to *disable* the author field on the edit form. To do that, we can take advantage of an option that every field type has: disabled. Set it to $isEdit.

52 lines | src/Form/ArticleFormType.php

```
... lines 1 - 23
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 26 - 38
            ->add('author', UserSelectTextType::class, [
                'disabled' => $isEdit
            ])
        ;
    }
... lines 44 - 52
```

Ok, let's try that out! Refresh the edit page. Disabled! Now try the new page: *not* disabled. Perfect!

Oh, by the way, this disabled option does *two* things. First, obviously, it adds a disabled attribute so that the browser prevents the user from modifying it. But it *also* now *ignores* any submitted data for this field. So, if a nasty user removed the disabled attribute and updated the field, meh - no problem - our form will ignore that submitted data.

Conditionally Hiding / Showing a Field

I want to do *one* more thing. The publishedAt field: I want to *only* show that on the *edit* page. Because, when we're creating a new article, I don't want the admin to be able to publish it immediately. To do that, instead of just disabling it, I want to remove the field *entirely* from the new form.

So, yea - we *could* leverage this $isEdit variable: that would totally work. But, let's make things more interesting: I want the ability to choose whether or not the publishedAt field should be shown when we *create* our form in the controller.

Here's the trick: go down to the edit form. The createForm() method actually has a *third* argument: an array of options that you can pass to your form. Let's invent a new one called include_published_at set to true.

82 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
class ArticleAdminController extends AbstractController
{
... lines 17 - 46
    public function edit(Article $article, Request $request, EntityManagerInterface $em)
    {
        $form = $this->createForm(ArticleFormType::class, $article, [
            'include_published_at' => true
        ]);
... lines 52 - 67
    }
... lines 69 - 80
}
```

Before doing *anything* else, try this. A huge error! *Just* like with the options you pass to an individual *field*, you can't just *invent* new options to pass to your form! The error says: look - the form does *not* have this option!

So... we'll add it! Copy the option name, go into ArticleFormType and, down in configureOptions(), add include_published_at set to false. *This* is enough to make this a valid option... with a default value.

56 lines | src/Form/ArticleFormType.php

```
... lines 1 - 14
class ArticleFormType extends AbstractType
{
... lines 17 - 47
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
... line 51
            'include_published_at' => false,
        ]);
    }
}
```

Now, up in buildForm(), the $options array will *always* have an include_published_at key. We can use that below to say if ($options['include_published_at']), then we want that field. Remove it from above, then say $builder paste and... clean that up a little bit.

56 lines | src/Form/ArticleFormType.php

```
... lines 1 - 23
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 26 - 40
        if ($options['include_published_at']) {
            $builder->add('publishedAt', null, [
                'widget' => 'single_text',
            ]);
        }
    }
... lines 47 - 56
```

I love it! On the edit form, because we've overridden that option to be true, when we refresh... yes! We have the field! Open up the profiler for your form and click on the top level. Nice! You can see that a passed option include_published_at was set to true.

For the new page, we should *not* have that field. Try it! Woh! An error from Twig:

> Neither the property publishedAt nor one of the methods publishedAt(), blah blah blah, exist in some FormView class.

It's blowing up inside form_row() because we're trying to render a field that doesn't exist! Go open that template: templates/article_admin/_form.html.twig, and wrap this in an if statement: {% if articleForm.publishedAt is defined %}, then we'll render the field.

12 lines | templates/article_admin/_form.html.twig

```
1  {{ form_start(articleForm) }}
   ... lines 2 - 6
7      {% if articleForm.publishedAt is defined %}
8          {{ form_row(articleForm.publishedAt) }}
9      {% endif %}
   ... lines 10 - 11
12 {{ form_end(articleForm) }}
```

Try it again. The field is gone! And because it's *completely* gone from the form, when we submit, the form system will *not* call the setPublishedAt() method at all.

Next: let's talk about another approach to handling the situation where your form looks different than your entity class: data transfer objects.

Chapter 33: Form Model Classes (DTOs)

I want to talk about a *different* strategy that we could have used for the registration form: a strategy that many people really love. The form class behind this is UserRegistrationFormType and it's bound to our User class. That makes sense: we ultimately want to create a User object. But this was an interesting form because, out of its three fields, *two* of them don't map back to a property on our User class! There is no plainPassword property or agreeTerms property on User. To work around this, we used a nice trick - setting mapped to false - which allowed us to have these fields without getting an error. Then, in our controller, we just need to read that data in a different way: like with $form['plainPassword']->getData()

This is a *great* example of a form that doesn't look *exactly* like our entity class. And when your form starts to look different than your entity class, or maybe it looks more like a combination of several entity classes, it might *not* make sense to try to bind your form to your entity at all! Why? Because you might have to do all *sorts* of crazy things to get that to work, including using *embedded* forms, which isn't even something I like to talk about.

What's the better solution? To create a model class that looks just like your form.

Creating the Form Model Class

Let's try this out on our registration form. In your Form/ directory, I like to create a Model/ directory. Call the new class UserRegistrationFormModel. The purpose of this class is *just* to hold data, so it doesn't need to extend anything. And because our form has three fields - email, plainPassword and agreeTerms - I'm going to create three *public* properties: email, plainPassword, agreeTerms.

13 lines | src/Form/Model/UserRegistrationFormModel.php

```
... lines 1 - 4
class UserRegistrationFormModel
{
    public $email;

    public $plainPassword;

    public $agreeTerms;
}
```

Wait, why public? We *never* make public properties! Ok, yes, we *could* make these properties private and then add getter and setter methods for them. That *is* probably a bit better. But, because these classes are *so* simple and have just this *one* purpose, I often cheat and make the properties public, which works fine with the form component.

Next, in UserRegistrationFormType, at the bottom, instead of binding our class to User::class, bind it to UserRegistrationFormModel::class.

52 lines | src/Form/UserRegistrationFormType.php

```
... lines 1 - 15
class UserRegistrationFormType extends AbstractType
{
... lines 18 - 44
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => UserRegistrationFormModel::class
        ]);
    }
}
```

And... that's it! Now, instead of creating a new User object and setting the data onto it, it will create a new

UserRegistrationFormModel object and put the data there. And *that* means we can remove both of these 'mapped' => false options: we *do* want the data to be mapped back onto that object.

In the controller, the *big* difference is that $form->getData() will *not* be a User object anymore - it will be a $userModel. I'll update the inline doc above this to make that obvious.

80 lines | src/Controller/SecurityController.php

```
... lines 1 - 15
class SecurityController extends AbstractController
{
... lines 18 - 45
    public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
    {
... lines 48 - 50
        if ($form->isSubmitted() && $form->isValid()) {
            /** @var UserRegistrationFormModel $userModel */
            $userModel = $form->getData();
... lines 54 - 72
        }
... lines 74 - 77
    }
}
```

When you use a model class, the downside is that you need to do a bit more work to *transfer* the data from our model object into the entity object - or *objects* - that actually need it. That's why these model classes are often called "data transfer objects": they just hold data and help transfer it between systems: the form system and our entity classes.

Add $user = new User() and $user->setEmail($userModel->email). For the password field, it's almost the same, but now the data comes from $userModel->plainPassword. Do the same thing for $userModel->agreeTerms.

83 lines | src/Controller/SecurityController.php

```
... lines 1 - 45
    public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guard
    {
... lines 48 - 50
        if ($form->isSubmitted() && $form->isValid()) {
... lines 52 - 54
            $user = new User();
            $user->setEmail($userModel->email);
            $user->setPassword($passwordEncoder->encodePassword(
... line 58
                $userModel->plainPassword
            ));
... line 61
            if (true === $userModel->agreeTerms) {
... line 63
            }
... lines 65 - 75
        }
... lines 77 - 80
    }
... lines 82 - 83
```

The benefit of this approach is that we're using this nice, concrete PHP class, instead of referencing specific array keys on the form for unmapped fields. The downside is... just more work! We need to transfer *every* field from the model class back to the User.

And also, if there were an "edit" form, we would need to create a new UserRegistrationFormModel object, populate *it* from the existing User object, and pass *that* as the second argument to ->createForm() so that the form is pre-filled. The best solution is up to you, but these data transfer objects - or DTO's, are a pretty clean solution.

Let's see if this actually works! I'll refresh just to be safe. This time, register as WillRyker@theenterprise.org, password engage, agree to the terms, register and... got it!

Validation Constraints

Mission accomplished! Right? Wait, no! We forgot about validation! For example, check out the email field on User: we *did* add some @Assert constraints above this! But... now that our form is not bound to a User object, these constraints are *not* being read! It is *now* reading the annotations off of *these* properties... and we don't have any!

Go back to your browser, inspect element on the form and add the novalidate attribute. Hit register to submit the form blank. Ah! We *do* have *some* validation: for the password and agree to terms fields. Why? Because those constraints were added into the form class itself.

Let's start fixing things up. Above the email property, paste the two existing annotations. I *do* need a use statement for this: I'll cheat - add another @Email, hit tab - there's the use statement - and then delete that extra line.

26 lines | src/Form/Model/UserRegistrationFormModel.php

```
... lines 1 - 6
class UserRegistrationFormModel
{
    /**
     * @Assert\NotBlank(message="Please enter an email")
     * @Assert\Email()
     */
    public $email;
... lines 14 - 24
}
```

At this point, if you *want* to, you can remove these annotations from your User class. But, because we might use the User class on a form somewhere else - like an edit profile form - I'll keep them there.

One of the really nice things about using a form model class is that we can remove the constraints from the form and put them in the model class so that we have everything in one place. Above $plainPassword, add @Assert\NotBlank() and @Assert\Length(). Let's pass in the same options: message="" and copy that from the form class. Then copy the minMessage string, add min=5, minMessage= and paste.

Finally, above agreeTerms, go copy the message from the form, and add the same @Assert\IsTrue() with message= that message.

26 lines | src/Form/Model/UserRegistrationFormModel.php

```
... lines 1 - 14
    /**
     * @Assert\NotBlank(message="Choose a password!")
     * @Assert\Length(min=5, minMessage="Come on, you can think of a password longer than that!")
     */
    public $plainPassword;

    /**
     * @Assert\IsTrue(message="I know, it's silly, but you must agree to our terms.")
     */
    public $agreeTerms;
... lines 25 - 26
```

Awesome! Let's celebrate by removing these from our form! Woo! Time to try it! Find your browser, refresh and... ooook - annotations parse error! It's a Ryan mistake! Let's go fix that - ah - what can I say? I love quotes!

Try it again. Much better! All the validation constraints are being cleanly read from our model class.

Except... for one. Go back to your User class: there was *one* more validation annotation on it: @UniqueEntity(). Copy this, go back into UserRegistrationFormModel and paste this above the class. We need a special use statement for this, so I'll re-type it, hit tab and... there it is! This annotation happens to live in a different namespace than all the others.

33 lines | src/Form/Model/UserRegistrationFormModel.php

```
... lines 1 - 4
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 6 - 7
/**
 * @UniqueEntity(
 *     fields={"email"},
 *     message="I think you're already registered!"
 * )
 */
class UserRegistrationFormModel
... lines 15 - 33
```

Let's try this - refresh. Woh! Huge error!

> Unable to find the object manager associated with an entity of class UserRegistrationFormModel

It thinks our model class is an entity! And, bad news friends: it is *not* possible to make UniqueEntity work on a class that is *not* an entity class. That's a bummer, but we *can* fix it: by creating our very-own custom validation constraint. Let's do that next!

Chapter 34: Custom Validator

Unfortunately, you can't use the @UniqueEntity() validation constraint above a class that is *not* an entity: it's just a known limitation. But, *fortunately*, this gives us the *perfect* excuse to create a custom validation constraint! Woo!

When you can't find a built-in validation constraint that does what you need, the *next* thing to try is the @Assert\Callback constraint. We use this in the Article class. But, it has one limitation: because the method lives inside an entity class - we do *not* have access to any services. In our case, in order to know whether or not the email is taken yet, we need to make a query and so we *do* need to access a service.

Generating the Constraint Validator

When that's your situation, it's time for a custom validation constraint. They're awesome anyways *and* we're going to cheat! Find your terminal and run:

```
$ php bin/console make:validator
```

Call the class, how about, UniqueUser. Oh, this created *two* classes: UniqueUser and UniqueUserValidator. You'll find these inside a new Validator/ directory. Look at UniqueUser first: it's basically a dumb configuration object. *This* will be the class we use for our annotation.

18 lines | src/Validator/UniqueUser.php

```
... lines 1 - 6
/**
 * @Annotation
 */
class UniqueUser extends Constraint
{
    /*
     * Any public properties become valid options for the annotation.
     * Then, use these in your validator class.
     */
    public $message = 'The value "{{ value }}" is not valid.';
}
```

The actual validation is handled by UniqueUserValidator: Symfony will pass it the value being validated *and* a Constraint object - which will be that UniqueUser object we just saw. We'll use it to read some options to help us get our job done. For example, in the generated code, it reads the message property from the $constraint and sets that as the validation error. That's literally reading this public $message property from UniqueUser.

19 lines | src/Validator/UniqueUserValidator.php

```
... lines 1 - 7
class UniqueUserValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        /* @var $constraint App\Validator\UniqueUser */

        $this->context->buildViolation($constraint->message)
            ->setParameter('{{ value }}', $value)
            ->addViolation();
    }
}
```

Configuring the Annotation

Ok: let's bring this generated code to life! Step 1: make sure your annotation class - UniqueUser - is ready to go. In general, an annotation can either be added above a class *or* above a property. Well, you can *also* add annotations above methods - that works pretty similar to properties.

If you add a validation annotation above your class, then during validation, the *value* that's passed to that validator is the entire *object*. If you add it above a property, then the value that's passed is *just* that property's value. So, if you need access to multiple fields on an object for validation, then you'll need to create an annotation that can be used above the class. In this situation, I'm going to delete @UniqueEntity and, instead, add the new annotation above my $email property: @UniqueUser. Hit tab to auto-complete that and get the use statement.

28 lines | src/Form/Model/UserRegistrationFormModel.php

```
... lines 1 - 4
use App\Validator\UniqueUser;
... lines 6 - 7
class UserRegistrationFormModel
{
    /**
... lines 11 - 12
     * @UniqueUser()
     */
    public $email;
... lines 16 - 26
}
```

Nice! Now, go back to your annotation class, we need to do a bit more work. To follow an example, press shift+ shift and open the core NotBlank annotation class. See that @Target() annotation above the class? This is a special annotation... that configures, um, the annotation system! @Target tells the annotation system *where* your annotation is allowed to be used. Copy that and paste it above our class. This says that it's okay for this annotation to be used above a property, above a method or even inside of another annotation... which is a bit more of a complex case, but we'll leave it.

19 lines | src/Validator/UniqueUser.php

```
... lines 1 - 6
/**
... line 8
 * @Target({"PROPERTY", "ANNOTATION"})
 */
class UniqueUser extends Constraint
... lines 12 - 19
```

What if you instead want your annotation to be put above a class? Open the UniqueEntity class as an example. Yep, you would use the CLASS target. The *other* thing you would need to do is override the getTargets() method. Wait, why is there an

@Target annotation *and* a getTargets() method - isn't that redundant? Basically, yep! These provide more or less the same info to two different systems: the annotation system and the validation system. The getTargets() method defaults to PROPERTY - so you only need to override it if your annotation should be applied to a class.

Configuring your Annotation Properties

Phew! The *last* thing we need to do inside of UniqueUser is give it a better default $message: we'll set it to the same thing that we have above our User class: I think you've already registered. Paste that and... cool!

19 lines | src/Validator/UniqueUser.php

```
... lines 1 - 10
class UniqueUser extends Constraint
{
... lines 13 - 16
    public $message = 'I think you\'re already registered!';
}
```

If you need to be able to configure more things on your annotation - just create more public properties on UniqueUser. Any properties on this class can be set or overridden as options when using the annotation. In UserRegistrationFormModel, I won't do it now, but we *could* add a message= option: that string would ultimately be set on the message property.

Before we try this, go to UniqueUserValidator. See the setParameter() line? The makes it possible to add wildcards to your message - like:

> The email {{ value }} is already registered

We could keep that, but since I'm not going to use it, I'll remove it. And... cool! With this setup, when we submit, this validator will be called and it will *always* fail. That's a good start. Let's try it!

Filling in the Validator Logic

Move over and refresh to resubmit the form. Yes! Our validator *is* working... it just doesn't have any logic yet! This is the easy part! Let's think about it: we need to make a query from inside the validator. Fortunately, these validator classes are *services*. And so, we can use our *favorite* trick: dependency injection!

Add an __construct() method on top with a UserRepository $userRepository argument. I'll hit alt+Enter to create that property and set it. Below, let's say $existingUser = $this->userRepository->findOneBy() to query for an email set to $value. Remember: because we put the annotation above the email property, $value will be that property's value.

Next, very simply, if (!$existingUser) then return. That's it.

34 lines | src/Validator/UniqueUserValidator.php

```
... lines 1 - 8
class UniqueUserValidator extends ConstraintValidator
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }
... line 17
    public function validate($value, Constraint $constraint)
    {
        $existingUser = $this->userRepository->findOneBy([
            'email' => $value
        ]);

        if (!$existingUser) {
            return;
        }
... lines 27 - 31
    }
}
```

One note: if this were an edit form where a user could *change* their email, this validator would need to make sure that the existing user wasn't actually just *this* user, if they submitted without changing their email. In that case, we would need $value to be the entire object so that we could use the id to be sure of this. To do that, you would need to change UniqueUser so that it lives above the *class*, instead of the property. You would also need to add an id property to UserRegistrationFormModel.

But, for us, this is it! Move back over, refresh and... got it! Try entering a new user and adding the novalidate attribute so we can be lazy and keep the other fields blank. Submit! Error gone. Try WillRyker@theenterprise.org with the same novalidate trick. And... the error is back.

Custom validation constraints, check! Next, we're going to update our Article form to add a few new drop-down select fields, but... with a catch: when the user selects an option from the first drop-down, the options of the *second* drop-down will need to update dynamically. Woh.

Chapter 35: Setup: For Dependent Select Fields

We're going to tackle one of the most *annoying* things in Symfony's form system, and, I hope, make it as *painless* as possible... because the end result is pretty cool!

Log in as admin2@thespacebar.com, password engage and then go to /admin/article. Click to create a new article. Here's the goal: on this form, I want to add two new drop-down select elements: a location drop-down - so you can choose where in the galaxy you are - and a second dropdown with more *specific* location options depending on what you chose for the location. For example, if you select "Near a star" for your location, the next drop-down would update to be a list of stars. Or, if you select "The Solar System", the next drop-down will be a list of planets.

Adding the First Select Field

This is called a "dependent form field", and, unfortunately, it's one of the trickier things to do with the form system - which is *exactly* why we're talking about it! Let's add the first new field. Find your terminal and run

```
$ php bin/console make:entity
```

Modify the Article entity and create a new field called location. Make it a string field with "yes" to nullable in the database: the location will be optional. Now run:

```
$ php bin/console make:migration
```

and open the Migrations/ directory to check out that new file.

29 lines | src/Migrations/Version20181031150638.php

```
... lines 1 - 12
13      public function up(Schema $schema) : void
14      {
15          // this up() migration is auto-generated, please modify it to your needs
16          $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys
17
18          $this->addSql('ALTER TABLE article ADD location VARCHAR(255) DEFAULT NULL');
19      }
... lines 20 - 29
```

No surprises here, so let's go back and run it:

```
$ php bin/console doctrine:migrations:migrate
```

Perfect!

Next, open the ArticleFormType so we can add the new field. Add location and set it to a ChoiceType to make it a drop-down. Pass the choices option set to just three choices. The Solar System set to solar_system, Near a star set to star and Interstellar Space set to interstellar_space.

65 lines | src/Form/ArticleFormType.php

```
... lines 1 - 15
class ArticleFormType extends AbstractType
{
... lines 18 - 24
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 27 - 29
        $builder
... lines 31 - 39
            ->add('location', ChoiceType::class, [
                'choices' => [
                    'The Solar System' => 'solar_system',
                    'Near a star' => 'star',
                    'Interstellar Space' => 'interstellar_space'
                ],
... line 46
            ])
        ;
... lines 49 - 54
    }
... lines 56 - 65
```

The choices on the ChoiceType can look confusing at first: the *key* for each item will be what's actually *displayed* in the drop down. And the *value* will be what's *set* onto our entity if this option is selected. So, this is the string that will ultimately be saved to the database.

Let's also add one more option: required set to false.

65 lines | src/Form/ArticleFormType.php

```
... lines 1 - 29
        $builder
... lines 31 - 39
            ->add('location', ChoiceType::class, [
... lines 41 - 45
                'required' => false,
            ])
        ;
... lines 49 - 65
```

Remember: as soon as we pass the field type as the second argument, the form field type guessing stops and does nothing. Lazy! It would normally guess that the required option *should* be false - because this field is not required in the database, but that won't happen. So, we set it explicitly.

Cool! Let's try it - go refresh the form. Ha! It works... but in a surprising way: the location field shows up... all the way at the bottom of the form.

The reason? We forgot to render it! Open templates/article_admin/_form.html.twig. When you forget to render a field, {{ form_end() }} renders it for you. It's kind of a nice reminder that I forgot it. Of course, we don't *really* want to render it all the way at the bottom like this. Instead, add {{ form_row(articleForm.location) }}

13 lines | templates/article_admin/_form.html.twig

```
{{ form_start(articleForm) }}
... lines 2 - 5
    {{ form_row(articleForm.location) }}
... lines 7 - 12
{{ form_end(articleForm) }}
```

Oh, and I forgot: we'll want an "empty" choice at the top of the select. In the form, add one more option: placeholder set to Choose a location.

66 lines | src/Form/ArticleFormType.php

```
... lines 1 - 24
25      public function buildForm(FormBuilderInterface $builder, array $options)
26      {
... lines 27 - 39
40              ->add('location', ChoiceType::class, [
41                  'placeholder' => 'Choose a location',
... lines 42 - 47
48              ])
49          ;
... lines 50 - 55
56      }
... lines 57 - 66
```

Refresh! So much nicer! And if we submitted the form, it *would* save.

Adding the Second Field

So, let's add the second field! Go back to your terminal and run:

```
$ php bin/console make:entity
```

Update the Article entity again and create a new field called specificLocationName, which will store a string like "Earth" or "Mars". Make this "yes" to nullable in the database - another optional field.

When you're done, make the migration:

```
$ php bin/console make:migration
```

And... I'm *pretty* confident that migration won't have any surprises, so let's just run it:

```
$ php bin/console doctrine:migrations:migrate
```

Sweet! Back in ArticleFormType, copy the location field, paste, and call it specificLocationName. For the placeholder, use Where exactly?. And for the choices... hmm - this is where things get interesting. I'll just add a dummy "TODO" option to start.

73 lines | src/Form/ArticleFormType.php

```
... lines 1 - 24
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 27 - 48
            ->add('specificLocationName', ChoiceType::class, [
                'placeholder' => 'Where exactly?',
                'choices' => [
                    'TODO' => 'TODO'
                ],
                'required' => false,
            ])
        ;
... lines 57 - 62
    }
... lines 64 - 73
```

Back in the form template, copy the location render line, paste it right below, and change it to specificLocationName.

14 lines | templates/article_admin/_form.html.twig

```
{{ form_start(articleForm) }}
... lines 2 - 6
    {{ form_row(articleForm.specificLocationName) }}
... lines 8 - 13
{{ form_end(articleForm) }}
```

When we refresh now... no surprise: it works. Here's our location and here's our specificLocationName. But... this is not how we want this to work. When "solar system" is selected, I want this second drop-down to contain a list of planets. If "Near a star" is selected, this should be a list of stars. And if "Interstellar space" is selected, I don't want this field to be in the form at all. Woh.

The way to solve this is a combination of form events, JavaScript and luck! Ok, I hope we won't need too much of that. Let's start jumping into these topics next!

Chapter 36: Form Events & Dynamic ChoiceType choices

Let's focus on the edit form first - it'll be a little bit easier to get working. Go to /admin/article and click to edit one of the existing articles. So, based on the location, we need to make this specificLocationName field have different options.

Determining the specificLocationName Choices

Open ArticleFormType and go to the bottom. I'm going to paste in a function I wrote called getLocationNameChoices(). You can copy this function from the code block on this page. But, it's fairly simple: We pass it the $location string, which will be one of solar_system, star or interstellar_space, and it returns the choices for the specificLocationName field. If we choose "solar system", it returns planets. If we choose "star", it returns some popular stars. And if we choose "Interstellar space", it returns null, because we actually don't want the drop-down to be displayed at *all* in that case.

108 lines | src/Form/ArticleFormType.php

```
... lines 1 - 75
    private function getLocationNameChoices(string $location)
    {
        $planets = [
            'Mercury',
            'Venus',
            'Earth',
            'Mars',
            'Jupiter',
            'Saturn',
            'Uranus',
            'Neptune',
        ];

        $stars = [
            'Polaris',
            'Sirius',
            'Alpha Centauari A',
            'Alpha Centauari B',
            'Betelgeuse',
            'Rigel',
            'Other'
        ];

        $locationNameChoices = [
            'solar_system' => array_combine($planets, $planets),
            'star' => array_combine($stars, $stars),
            'interstellar_space' => null,
        ];

        return $locationNameChoices[$location];
    }
... lines 107 - 108
```

Oh, and I'm using array_combine() just because I want the display values and the values set back on my entity to be the same. This is equivalent to saying 'Mercury' => 'Mercury'... but saves me some duplication.

Dynamically Changing the Options

The first step to get this working is not *so* different from something we did earlier. To start, *forget* about trying to use fancy JavaScript to instantly reload the specificLocationName drop-down when we select a new location. Yes, we *are* going to do that - but later.

Hit "Update" the save the location to "The Solar System". The first goal is this: when the form loads, because the location field is already set, the specificLocationName should show me the planet list. In other words, we should be able to use the underlying Article data inside the form to figure out which choices to use.

I'll add some inline documentation just to tell my editor that this is an Article object or null. Then, $location = , if $article is an object, then $article->getLocation(), otherwise, null.

```
108 lines | src/Form/ArticleFormType.php
... lines 1 - 24
25      public function buildForm(FormBuilderInterface $builder, array $options)
26      {
27          /** @var Article|null $article */
28          $article = $options['data'] ?? null;
... line 29
30          $location = $article ? $article->getLocation() : null;
... lines 31 - 65
66      }
... lines 67 - 108
```

Down below, copy the entire specificLocationName field and remove it. Then *only* if ($location) is set, add that field. For choices, use $this->getLocationNameChoices() and pass that $location.

```
108 lines | src/Form/ArticleFormType.php
... lines 1 - 24
25      public function buildForm(FormBuilderInterface $builder, array $options)
26      {
... lines 27 - 52
53          if ($location) {
54              $builder->add('specificLocationName', ChoiceType::class, [
55                  'placeholder' => 'Where exactly?',
56                  'choices' => $this->getLocationNameChoices($location),
57                  'required' => false,
58              ]);
59          }
... lines 60 - 65
66      }
... lines 67 - 108
```

Cool! Again, no, if we change the location field, it will *not* magically update the specificLocationName field... not yet, at least. With this code, we're saying: when we originally load the form, if there is already a $location set on our Article entity, let's add the specificLocationName field with the correct choices. If there is *no* location, let's not load that field at *all*, which means in _form.html.twig, we need to render this field conditionally: {% if articleForm.specificLocationName is defined %}, then call form_row().

```
16 lines | templates/article_admin/_form.html.twig
1   {{ form_start(articleForm) }}
... lines 2 - 6
7       {%  if articleForm.specificLocationName is defined %}
8           {{ form_row(articleForm.specificLocationName) }}
9       {% endif %}
... lines 10 - 15
16  {{ form_end(articleForm) }}
```

Let's try this! Refresh the page. The Solar System is selected and so... sweet! There is our list of planets! And we can totally

save this. Yep! It saved as Earth. Open a second tab and go to the new article form. No surprise: there is *no* specificLocationName field here because, of course, the location isn't set yet.

Our system now... sort of works. We can change the data... but we need to do it little-by-little. We can go to "Near a Star", hit "Update" and *then* change the specificLocationName field and save that. But I can't do it all at once: I need to fully reload the page... which kinda sucks!

Can you Hack the Options to Work?

Heck, we can't even be clever! Change location to "The Solar System". Then, inspect element on the next field and change the "Betelgeuse" option to "Earth". In theory, that should work, right? Earth *is* a valid option when location is set to solar_system, and so this should *at least* be a hacky way to work with the system.

Hit Update. Woh! It does *not* work! We get a validation error: This value is not valid. Why?

Think about it: when we submit, Symfony *first* builds the form based on the Article data that's stored in the *database*. Because location is set to star in the database, it builds the specificLocationName field with the *star* options. When it sees earth being submitted for that field, it looks invalid!

Our form needs to be even smarter: when we submit, the form needs to *realize* that the location field changed, and rebuild the specificLocationName choices before processing the data. Woh.

We can do that by leveraging form events.

Chapter 37: Dynamic Form Events

Alright, here's the issue and it is *super* technical. If we change the Location from "Near a Star" to "Solar System", *even* if we "hack" the specificLocationName field so that it submits the value "Earth", it doesn't work! It fails validation!

This *is* a real problem, because, in a few minutes, we're going to add JavaScript to the page so that when we change location to "The Solar System", it will dynamically update the specificLocationName dropdown down to be the list of planets. But for that to work, our form system needs to be smart enough to realize - at the *moment* we're submitting - that the location has changed. And then, before it validates the ChoiceType, it needs to *change* the choices to be the list of planets.

Don't worry if this doesn't make complete sense yet - let's see some code!

Adding an Event Listener

There's one piece of the form system that we haven't talked about yet: it has an *event* system, which we can use to hook into the form loading & submitting process.

At the end of the form, add $builder->get('location')->addEventListener() and pass this FormEvents::POST_SUBMIT. This FormEvents class holds a constant for each "event" that we can hook into for the form system. Pass a callback as a second argument: Symfony will pass that a FormEvent object.

Let's dd() the $event so we can see what it looks like.

117 lines | src/Form/ArticleFormType.php

```
... lines 1 - 26
27      public function buildForm(FormBuilderInterface $builder, array $options)
28      {
... lines 29 - 68
69          $builder->get('location')->addEventListener(
70              FormEvents::POST_SUBMIT,
71              function(FormEvent $event) {
72                  dd($event);
73              }
74          );
75      }
... lines 76 - 117
```

But before we check it out, two important things. First, when you build a form, it's actually a big form tree. We've seen this inside of the form profiler. There's a Form object on top and then each individual field below is itself a full Form object. The same is true with the "form builder": we normally just interact with the top-level $builder by adding fields to it. When we call $builder->add(), that creates another "form builder" object for that field, and you can fetch it later by saying $builder->get().

Second, we're attaching the event to only the location *field* - not the entire form. So, when the form submits, Symfony will call this function, but the $event object will only have information about the location field - not the entire form.

Let's actually *see* this! Refresh to re-submit the form. There it is! The FormEvent contains the raw, submitted data - the solar_system string - *and* the entire Form object for this one field.

Dynamically Updating the Field

This gives us the hook we need: we can use the submitted data to *dynamically* change the specificLocationName field to use the correct choices, *right* before validation occurs. Actually, this hook happens *after* validation - but we'll use a trick where we remove and re-add the field, to get around this.

To start, create a new private function called setupSpecificLocationNameField(). The job of this function will be to dynamically add the specificLocationName field with the correct choices. It will accept a FormInterface - we'll talk about that in a minute - and a ?string $location, the ? part so this can be null.

145 lines | src/Form/ArticleFormType.php

```
... lines 1 - 81
    private function setupSpecificLocationNameField(FormInterface $form, ?string $location)
    {
... lines 84 - 102
    }
... lines 104 - 145
```

Inside, first check if $location is null. If it is, take the $form object and actually ->remove() the specificLocationName field and return. Here's the idea: if when I originally rendered the form there was a location set, then, thanks to our logic in buildForm(), there *will* be a specificLocationName field. But if we changed it to "Choose a location", meaning we are *not* selecting a location, then we want to *remove* the specificLocationName field before we do any validation. We're kind of trying to do the same thing in *here* that our future JavaScript will do instantly on the frontend: when we change to "Choose a location" - we will want the field to disappear.

145 lines | src/Form/ArticleFormType.php

```
... lines 1 - 81
    private function setupSpecificLocationNameField(FormInterface $form, ?string $location)
    {
        if (null === $location) {
            $form->remove('specificLocationName');

            return;
        }
... lines 89 - 102
    }
... lines 104 - 145
```

Next, get the $choices by using $this->getLocationNameChoices() and pass that $location. Then, similar to above, if (null === $choices) remove the field and return. This is needed for when the user selects "Interstellar Space": that doesn't have any specific location name choices, and so we don't want that field at all.

145 lines | src/Form/ArticleFormType.php

```
... lines 1 - 81
    private function setupSpecificLocationNameField(FormInterface $form, ?string $location)
    {
... lines 84 - 89
        $choices = $this->getLocationNameChoices($location);

        if (null === $choices) {
            $form->remove('specificLocationName');

            return;
        }
... lines 97 - 102
    }
... lines 104 - 145
```

Finally, we *do* want the specificLocationName field, but we want to use our new choices. Scroll up and copy the $builder->add() section for this field, paste down here, and change $builder to $form - these two objects have an identical add() method. For choices pass $choices.

145 lines | src/Form/ArticleFormType.php

```
... lines 1 - 81
    private function setupSpecificLocationNameField(FormInterface $form, ?string $location)
    {
... lines 84 - 97
        $form->add('specificLocationName', ChoiceType::class, [
            'placeholder' => 'Where exactly?',
            'choices' => $choices,
            'required' => false,
        ]);
    }
... lines 104 - 145
```

Nice! We created this new function so that we can call it from inside of our listener callback. Start with $form = $event->getForm(): that gives us the actual Form object for this one field. Now call $this->setupSpecificLocationNameField() and, for the first argument, pass it $form->getParent().

145 lines | src/Form/ArticleFormType.php

```
... lines 1 - 27
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 30 - 69
        $builder->get('location')->addEventListener(
            FormEvents::POST_SUBMIT,
            function(FormEvent $event) {
                $form = $event->getForm();
                $this->setupSpecificLocationNameField(
                    $form->getParent(),
... line 76
                );
            }
        );
    }
... lines 81 - 145
```

This is tricky. The $form variable is the Form object that represents just the location field. But we want to pass the *top* level Form object into the function so that the specificLocationName field can be added or removed from *it*.

The second argument is the location itself, which will be $form->getData(), or $event->getData().

145 lines | src/Form/ArticleFormType.php

```
... lines 1 - 73
                $this->setupSpecificLocationNameField(
                    $form->getParent(),
                    $form->getData()
                );
... lines 78 - 145
```

Okay guys, I know this is craziness, but we're ready to try it! Refresh to resubmit the form. It saves. Now change the Location to "Near a Star". In a few minutes, our JavaScript will reload the specificLocationName field with the new options. To fake that, inspect the element. Let's go copy a real star name - how about Sirius. Change the selected option's value to that string.

Hit update! Yes! It saved! We were able to change both the location *and* specificLocationName fields at the same time.

And *that* means that we're ready to swap out the field dynamically with JavaScript. But first, we're going to leverage another form event to remove some duplication from our form class.

Chapter 38: PRE_SET_DATA: Data-based Dynamic Fields

On our form class, we're creating the specificLocationName field in two places: it's up in buildForm() and duplicated down inside of setupSpecificLocationNameField(). Because duplication is a *bummer*, let's fix it by calling $this->setupSpecificLocationNameField() from buildForm().

Except... hmm, there's a minor mismatch: in buildForm(), we're working with a form *builder* object, but the method expects a FormInterface object. It's a weird situation where these two objects *happen* to have the same add() method, but they are two totally different classes.

We're going to work around this by leveraging another form event. Remove the block where we first add the specificLocationName field. Oh, and we can remove the $location variable now too.

Let's think about how we could re-add this field using events: we basically want Symfony to call our callback, the moment the underlying "data" is set onto the form - the Article object. Use $builder->addEventListener() and listen on an event called FormEvents::PRE_SET_DATA. Two things: first, this time, we're attaching the event to the entire *form*, which means our callback will be passed info about the entire form. That's *usually* want you want: listening to a single field like we did before was a bit of a hack to allow us to remove and re-add the field at *just* the right moment.

152 lines | src/Form/ArticleFormType.php

```
... lines 1 - 27
28      public function buildForm(FormBuilderInterface $builder, array $options)
29      {
... lines 30 - 60
61          $builder->addEventListener(
62              FormEvents::PRE_SET_DATA,
... lines 63 - 74
75          );
... lines 76 - 86
87      }
... lines 88 - 152
```

Second, how do we know to use PRE_SET_DATA? When exactly is that called? Open ArticleAdminController: in the edit() action, we pass createForm() an Article object. When that happens, Symfony dispatches this PRE_SET_DATA event. In general, the FormEvents class itself is a *great* resource for finding out when each event is called and what you can do by listening to it. I won't do it here, but if you hold Command or Ctrl and click the event name to open that class, you'll find great documentation above each constant.

Add the callback with the same FormEvent $event argument. Then, get the underlying data with $data = $event->getData(). *We* know that this must be either an Article object or possibly null. If there is *no* data, just return and do nothing: we don't want to add the field at *all* for the new form.

152 lines | src/Form/ArticleFormType.php

```
... lines 1 - 60
61          $builder->addEventListener(
... line 62
63              function (FormEvent $event) {
64                  /** @var Article|null $data */
65                  $data = $event->getData();
66                  if (!$data) {
67                      return;
68                  }
... lines 69 - 73
74              }
75          );
... lines 76 - 152
```

If there *is* data, call $this->setupSpecificLocationNameField() and pass it $event->getForm(). This time, $event->getForm() will be the top-level form, because we added the listener to the top-level builder. For the location, pass $data->getLocation().

152 lines | src/Form/ArticleFormType.php

```
... lines 1 - 60
61          $builder->addEventListener(
... line 62
63              function (FormEvent $event) {
... lines 64 - 69
70                  $this->setupSpecificLocationNameField(
71                      $event->getForm(),
72                      $data->getLocation()
73                  );
74              }
75          );
... lines 76 - 152
```

Cool! This code *should* work just like before. But actually, while events are nice, if I need to tweak my form based on the underlying data - like we're doing here - I prefer to *avoid* using events and just use the $options['data'] key. It's just a bit simpler. But, both solutions are fine.

Anyways, let's try it! I'll hit enter on the address bar to get a fresh page. And... yep! Because "Near a star" is selected as the location, the next field loaded with the correct list of stars.

We are now *fully* ready for the last, fancy step: adding JavaScript and AJAX to dynamically change the specificLocationName select options when the location changes. And... that's probably the easiest part!

Chapter 39: JS to Auto-Update the Select Options

Thanks to these event listeners, no matter what data we start with - or what data we submit - for the location field, the specificLocationName field choices will update so that everything saves.

The last step is to add some JavaScript! When the form loaded, the location was set to "Near a star". When I change it to "The Solar System", we need to make an Ajax call that will fetch the list of planets and update the option elements.

Adding the Options Endpoint

In ArticleAdminController, let's add a new endpoint for this: public function getSpecificLocationSelect(). Add Symfony's Request object as an argument. Here's the idea: our JavaScript will send the location that was just selected to this endpoint and *it* will return the new HTML needed for the entire specificLocationName field. So, this won't be a pure API endpoint that returns JSON. We *could* do that, but because the form is already rendering our HTML, returning HTML simplifies things a bit.

```
101 lines | src/Controller/ArticleAdminController.php
    ... lines 1 - 14
15  class ArticleAdminController extends AbstractController
16  {
    ... lines 17 - 72
73      public function getSpecificLocationSelect(Request $request)
74      {
    ... lines 75 - 86
87      }
    ... lines 88 - 99
100 }
```

Above the method add the normal @Route() with /admin/article/location-select. And give it a name="admin_article_location_select".

```
101 lines | src/Controller/ArticleAdminController.php
    ... lines 1 - 69
70      /**
71       * @Route("/admin/article/location-select", name="admin_article_location_select")
72       */
73      public function getSpecificLocationSelect(Request $request)
    ... lines 74 - 101
```

Inside, the logic is kinda cool: create a new Article: $article = new Article(). Next, we need to set the new location *onto* that. When we make the AJAX request, we're going to add a ?location= query parameter. Read that here with $request->query->get('location').

```
101 lines | src/Controller/ArticleAdminController.php
    ... lines 1 - 72
73      public function getSpecificLocationSelect(Request $request)
74      {
75          $article = new Article();
76          $article->setLocation($request->query->get('location'));
    ... lines 77 - 86
87      }
    ... lines 88 - 101
```

But, let's back up: we're *not* creating this Article object so we can save it, or anything like that. We're going to build a temporary *form* using this Article's data, and render *part* of it as our response. Check it out:

$form = $this->createForm(ArticleFormType::class, $article). We know that, thanks to our event listeners - specifically our PRE_SET_DATA event listener - this form will now have the correct specificNameLocation options based on whatever location was just sent to us.

101 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 72
    public function getSpecificLocationSelect(Request $request)
    {
... lines 75 - 76
        $form = $this->createForm(ArticleFormType::class, $article);
... lines 78 - 86
    }
... lines 88 - 101
```

Or, the field may have been removed! Check for that first: if (!$form->has('specificLocationName') then just return new Response() - the one from HttpFoundation - with no content. I'll set the status code to 204, which is a fancy way of saying that the call was successful, but we have no content to send back.

101 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 72
    public function getSpecificLocationSelect(Request $request)
    {
... lines 75 - 78
        // no field? Return an empty response
        if (!$form->has('specificLocationName')) {
            return new Response(null, 204);
        }
... lines 83 - 86
    }
... lines 88 - 101
```

If we *do* have that field, we want to render it! Return and render a new template: article_admin/_specific_location_name.html.twig. Pass this the form like normal 'articleForm' => $form->createView(). Then, I'll put my cursor on the template name and press alt+enter to make PhpStorm create that template for me.

101 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 72
    public function getSpecificLocationSelect(Request $request)
    {
... lines 75 - 83
        return $this->render('article_admin/_specific_location_name.html.twig', [
            'articleForm' => $form->createView(),
        ]);
    }
... lines 88 - 101
```

Inside, just say: {{ form_row(articleForm.specificLocationName) }} and that's it.

1 lines | templates/article_admin/_specific_location_name.html.twig

```
{{ form_row(articleForm.specificLocationName) }}
```

Yep, we're literally returning *just* the form row markup for this *one* field. It's a weird way to use a form, but it works!

Let's go try this out! Copy the new URL, open a new tab and go to http://localhost:8000/admin/article/location-select?location=star

Cool! A drop-down of stars! Try solar_system and... that works too. Excellent!

JS Setup: Adding data- Attributes & Classes

Next, open _form.html.twig. Our JavaScript will need to be able to *find* the location select element so it can read its value *and* the specificLocationName field so it can replace its contents. It also needs to know the URL to our new endpoint.

No problem: for the location field, pass an attr array variable. Add a data-specific-location-url key set to path('admin_article_location'). Then, add a class set to js-article-form-location.

23 lines | templates/article_admin/_form.html.twig

```
{{ form_start(articleForm) }}
... lines 2 - 5
    {{ form_row(articleForm.location, {
        attr: {
            'data-specific-location-url': path('admin_article_location_select'),
            'class': 'js-article-form-location'
        }
    }) }}
... lines 12 - 22
{{ form_end(articleForm) }}
```

Next, *surround* the specificLocationName field with a new . I'm adding this as a new element *around* the field instead of *on* the select element so that we can remove the field without losing this target element.

23 lines | templates/article_admin/_form.html.twig

```
... lines 1 - 11
    <div class="js-specific-location-target">
        {%  if articleForm.specificLocationName is defined %}
... line 14
        {% endif %}
    </div>
... lines 17 - 23
```

Adding the JavaScript

Ok, we're ready for the JavaScript! Open up the public/ directory and create a new file: admin_article_form.js. I'm going to paste in some JavaScript that I prepped: you can copy this from the code block on this page.

27 lines | public/js/admin_article_form.js

```
$(document).ready(function() {
    var $locationSelect = $('.js-article-form-location');
    var $specificLocationTarget = $('.js-specific-location-target');

    $locationSelect.on('change', function(e) {
        $.ajax({
            url: $locationSelect.data('specific-location-url'),
            data: {
                location: $locationSelect.val()
            },
            success: function (html) {
                if (!html) {
                    $specificLocationTarget.find('select').remove();
                    $specificLocationTarget.addClass('d-none');

                    return;
                }

                // Replace the current field and show
                $specificLocationTarget
                    .html(html)
                    .removeClass('d-none')
            }
        });
    });
});
```

Before we talk about the specifics, let's include this with the script tag. Unfortunately, we can't include JavaScript directly in _form.html.twig because that's an included template. So, in the edit template, override {% block javascripts %}, call the {{ parent() }} function and then add a <script> tag with src="{{ asset('js/admin_article_form.js') }}.

16 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 10
{% block javascripts %}
    {{ parent() }}

    <script src="{{ asset('js/admin_article_form.js') }}"></script>
{% endblock %}
```

Copy that, open the new template, and paste this at the bottom of the javascripts block.

24 lines | templates/article_admin/new.html.twig

```
... lines 1 - 2
{% block javascripts %}
... lines 4 - 7
    <script src="{{ asset('js/admin_article_form.js') }}"></script>
{% endblock %}
... lines 10 - 24
```

Before we try this, let's check out the JavaScript so we can see the entire flow. I made the code here as simple, and unimpressive as possible - but it gets the job done. First, we select the two elements: $locationSelect is the actual select element and $specificLocationTarget represents the div that's around that field. The $ on the variables is meaningless - I'm just using it to indicate that these are jQuery elements.

Next, when the location select changes, we make the AJAX call by reading the data-specific-location-url attribute. The location key in the data option will cause that to be set as a query parameter.

Finally, on success, if the response is empty, that means that we've selected an option that should *not* have a specificLocationName dropdown. So, we look inside the $specificLocationTarget for the select and remove it to make sure it doesn't submit with the form. On the wrapper div, we also need to add a Bootstrap class called d-none: that stands for display none. That will hide the entire element, including the label.

If there *is* some HTML returned, we do the opposite: replace the entire HTML of the target with the new HTML and remove the class so it's not hidden. And... that's it!

There are a *lot* of moving pieces, so let's try it! Refresh the edit page. The current location is "star" and... so far, no errors in my console. Change the option to "The Solar System". Yes! The options updated! Try "Interstellar Space"... gone!

If you look deeper, the js-specific-location-target div *is* still there, but it's hidden, and only has the label inside. Change back to "The Solar System". Yep! The d-none is gone and it now has a select field inside.

Try saving: select "Earth" and Update! We got it! We can keep changing this all day long - all the pieces are moving perfectly.

I'm super happy with this, but it *is* a complex setup - I totally admit that. If you have this situation, you need to choose the best solution: if you have a big form with 1 dependent field, what we just did is probably a good option. But if you have a small form, or it's even more complex, it might be better to skip the form component and code everything with JavaScript and API endpoints. The form component is a great tool - but not the best solution for every problem.

Next: there are a *few* small details we need to clean up before we are *fully* done with this form. Let's squash those!

Chapter 40: Clear that Location Name Data

When we change to the solar system, great! It loads the planets. We can even change to "Interstellar Space" and it disappears. We're amazing! And when we change it to "Choose a Location"... uhhh oh! Nothing happened? Ah, the Ajax part of the web debug toolbar is trying to tell me that there was a 500 error!

By the way, this is one of the *coolest* features of the web debug toolbar: when you get a 500 error on an AJAX call, you can click this link to jump straight into the profiler for that request! It takes us straight to the Exception screen so we can see exactly what we messed up, I mean, what went wrong... that may or may not be our fault.

Fixing our Empty Value Bug

Apparently ArticleFormType line 125 has an undefined "empty string" index. Let's go check that out. This is the method that we call to get the correct specificLocationName choices. But, in this case, the location is an empty string, and that's a *super* not valid key.

To fix this, add ?? null. This says, *if* the location key is set, use it, else use null.

152 lines | src/Form/ArticleFormType.php

```
... lines 1 - 18
19  class ArticleFormType extends AbstractType
20  {
... lines 21 - 119
120     private function getLocationNameChoices(string $location)
121     {
... lines 122 - 148
149         return $locationNameChoices[$location] ?? null;
150     }
151 }
```

Let's make sure that worked: on your browser, switch back to the solar system, and then back to "Choose a Location". Nice! The field disappears and *no* 500 error this time.

Forcing specificLocationName to null

There's *one* other subtle problem with our setup. To see it, refresh this page. In the database, this article's location is star, specificLocationName is Rigel and id is 28. Let's go verify this in the database: find your terminal and run:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article WHERE id = 28'
```

Yep! All the data looks like we expected! But *now*, change the location to "Interstellar Space" and hit update. It works... but try that query again:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article WHERE id = 28'
```

Ok: the location is interstellar_space, but ah! The specific_location_name is *still* Rigel! This may or may *not* be a real problem - depending on how you use this data. But it's *for sure* technically wrong: when we change the location to interstellar_space, the specific_location_name should be set *back* to null: we are *not* at Rigel.

The reason this did *not* happen is subtle. When we change the location to "Interstellar Space" and submit, our POST_SUBMIT listener function calls setupSpecificLocationNameField(), which sees that there are no choices for this location and so removes the field entirely. The end result is that the form makes *no* changes to the specificLocationName property on Article: it just *never* calls setSpecificLocationName() at all... because that field isn't part of the form!

That *is* the correct behavior. But, it means that we need to do a *little* bit more work to clean things up. There are a few ways to fix this inside the form itself. But, honestly, they're overly-complex. The solution *I* like lives entirely in Article. Open that class and find the setLocation() method. Inside, if there is *no* location, or if the location equals interstellar_space, call $this->setSpecificLocationName(null).

309 lines | src/Entity/Article.php

```
... lines 1 - 17
18  class Article
19  {
... lines 20 - 285
286     public function setLocation(?string $location): self
287     {
... lines 288 - 289
290         if (!$this->location || $this->location === 'interstellar_space') {
291             $this->setSpecificLocationName(null);
292         }
... lines 293 - 294
295     }
... lines 296 - 307
308 }
```

Simple! Oh, and in a real app, I'd probably add some class constants in Article to represent these special location keys so we could use something like Article::INTERSTELLAR_SPACE instead of just the string interstellar_space.

Let's try this people! First change the data back to a planet. Then, change it to "Interstellar Space" and update. Cool! Spin back over to our terminal and run that same query:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article WHERE id = 28'
```

Now it's set to null. Awesome. Next: we're pretty much done! There's just *one* last piece of homework left - and it's related to securing one of our endpoints.

Chapter 41: A bit of Security Cleanup

There's *one* last piece of business that we need to clean up. In ArticleAdminController, we created this endpoint... but we didn't add any security on it! It's open entirely to the world: there's no @IsGranted annotation above the method *or* above the class.

Securing the new Endpoint

Now... this *might* be ok - this endpoint just returns some boring, non-sensitive HTML anyways. But, let's be cautious.

The *tricky* thing is that this endpoint is used on both the new and edit form pages. On the new page, we require you to have ROLE_ADMIN_ARTICLE. But on the edit page, we use a special voter that gives you access if you have ROLE_ADMIN_ARTICLE *or* if you are the author of the article.

So, hmm - our endpoint needs to be available to anyone that has ROLE_ADMIN_ARTICLE *or* is the author of at least one article. A little odd, but we can make that happen!

The *proper* way to solve this is to create a new voter and call @IsGranted() with a new attribute we invent, like ADMIN_ARTICLE_FORM. The voter would handle that attribute and have all the logic inside.

But... because we only need to use this security logic on this *one* endpoint, and because I'm feeling lazy, let's instead put the logic right in the controller. We can always move it to a voter later if we need to re-use it.

First, add @IsGranted("ROLE_USER") to *at least* make sure the user is logged in. Then, inside the method, if *not* $this->isGranted('ROLE_ADMIN_ARTICLE') *and* $this->getUser()->getArticles() === 0, then we should not have access. Wait, but the ->getArticles() method is not auto-completing for me.

107 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
15  class ArticleAdminController extends BaseController
16  {
... lines 17 - 69
70      /**
... line 71
72       * @IsGranted("ROLE_USER")
73       */
74      public function getSpecificLocationSelect(Request $request)
75      {
76          // a custom security check
77          if (!$this->isGranted('ROLE_ADMIN_ARTICLE') && $this->getUser()->getArticles()->isEmpty()) {
... lines 78 - 92
93      }
... lines 94 - 105
106 }
```

Oh, I know why! Go to the top of this class and change the base class from extends AbstractController to extends BaseController.

107 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
15  class ArticleAdminController extends BaseController
... lines 16 - 107
```

Reminder: BaseController is a controller that *we* created. It extends AbstractController but it adds a return type to getUser() with *our* User class so we get auto-completion.

Back down in our method, we can say $this->getUser()->getArticles()->isEmpty(), which is a method on Doctrine's Collection object. So, if we don't have ROLE_ADMIN_ARTICLE *and* we are not the author of any articles, throw $this->createAccessDeniedException().

107 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 73
74      public function getSpecificLocationSelect(Request $request)
75      {
... line 76
77          if (!$this->isGranted('ROLE_ADMIN_ARTICLE') && $this->getUser()->getArticles()->isEmpty()) {
78              throw $this->createAccessDeniedException();
79          }
... lines 80 - 92
93      }
... lines 94 - 107
```

Done! And just to make sure I didn't completely break things, if I change the location to "Near a star"... yea! It *still* loads.

Fetch EXTRA_LAZY

What *really* made adding this security easy was being able to call $this->getUser()->getArticles(). The *problem* is that if this user is the author of 200 articles, then this will query for 200 rows of articles *and* hydrate those into 200 full objects, *just* to figure out that, yes, we *are* the author of at least one article. All we *really* need is a quick *count* query of the articles.

Fortunately, we can tell isEmpty() to do that! Open User and look for that articles property. At the end of the @OneToMany annotation, add fetch="EXTRA_LAZY". We talked about this option in our Doctrine relations tutorial. With this set, if we simply try to *count* the articles - which is what isEmpty() does - then Doctrine will make a quick COUNT query instead of fetching all the data. Nice!

270 lines | src/Entity/User.php

```
... lines 1 - 19
20  class User implements UserInterface
21  {
... lines 22 - 63
64      /**
65       * @ORM\OneToMany(targetEntity="App\Entity\Article", mappedBy="author", fetch="EXTRA_LAZY")
66       */
67      private $articles;
... lines 68 - 268
269 }
```

Using the @method in BaseController

Ok, *one* more thing - and it's also unrelated to forms. Open BaseController. By extending AbstractController, this class gives us all the great shortcut method we love but it *also* overrides getUser() so that our editor knows that this method will return *our* specific User class.

After we did this, a wonderful SymfonyCasts user pointed out that the getUser() method on the parent class is marked as final with @final. When something is final it means that we are *not* allowed to override it. Symfony *could* enforce this by changing the method to be final protected function getUser(). Then, we would get an error! But, Symfony often uses the softer @final comment, which is just documentation, either to prevent breaking backward compatibility or because it's harder for Symfony to unit test code that has final methods.

Anyways, the method is *intended* to be final, which means that we're not supposed to override it. So, delete the method in our class. There's another nice solution anyways: above the class add @method User getUser().

13 lines | src/Controller/BaseController.php

```
... lines 1 - 7
/**
 * @method User getUser()
 */
abstract class BaseController extends AbstractController
... lines 12 - 13
```

That's it! That does the *exact* same thing: it hints to our IDE that the getUser() method returns our User object. Back in ArticleAdminController, if we delete getArticles() and re-type... yep! It works!

Phew! Amazing job people! That was a *huge* topic to get through. Seriously, congrats!

The Symfony form system is both massively powerful and, in some places, quite complex. It has the power to make you incredibly productive or just as *unproductive* if you use it in the wrong place or the wrong ways. So, be smart: and follow these two rules.

One: if your form looks quite different than your entity, either *remove* the data_class option and use the associative array the form gives you to do your work *or* bind your form to a model class. Two: if your form has a complex frontend with a lot of AJAX and updating, it might be easier - *and* a better user experience - if you skip the form and write everything with JavaScript. Use this tool in the right places, and you'll be happy.

Let me know what you guys are building! And, as always, if you have any questions, ask us down in the comments.

Alright friends, see ya next time.