Starting in Symfony2: Course 4 (2.4+)

With <3 from SymfonyCasts

Chapter 1: Introduction

THE FINAL STRETCH¶

Ok guys, welcome to the last episode. We'll be working on the project from where we left off in episode 3. You can get that from the code download on this tutorial. I'm not going to bore you with a lot of introduction, I'd rather get to work. But I will say that you've come a long way through the first 3 episodes and you now understand all the really important parts of Symfony, like routes, controllers and services. Here in episode 4, we're going to round things out by learning more about forms, handling assets and how we can get our app up to production.

And hey, don't get lazy on me now. Code along with the tutorial - it'll make a huge difference.

Ok, I can't wait any longer - let's go!

Chapter 2: Assets and Cache Busting

WE (MOSTLY) DON'T CARE ABOUT YOUR CSS/JS¶

We'll start by talking about CSS and JS files, and just how much Symfony *doesn't* care about these. I mean that in a good way - you don't necessarily need a PHP Framework to help you include a JavaScript file.

Open up your base template and find the weird `stylesheets` tag there:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% block stylesheets %}
    {# link tag for bootstrap... #}

    {% stylesheets
        'bundles/event/css/event.css'
        'bundles/event/css/events.css'
        'bundles/event/css/main.css'
        filter='cssrewrite'
    %}
        <link rel="stylesheet" href="{{ asset_url }}" />
    {% endstylesheets %}
{% endblock %}
```

Symfony does have some *optional* tricks for assets, and this is one of them. For now, just remove this whole block and replace it with 3 good, old-fashioned `link` tags:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% block stylesheets %}
    {# link tag for bootstrap... #}

    <link rel="stylesheet" href="/bundles/event/css/event.css" />
    <link rel="stylesheet" href="/bundles/event/css/events.css" />
    <link rel="stylesheet" href="/bundles/event/css/main.css" />
{% endblock %}
```

Login with Wayne and password waynepass (party on) and then open up the HTML source on the homepage.

No Symfony magic here - this is just pure frontend code that points to real files in the `web/bundles/event/css` directory. And since the `web/` directory is the document root, we don't include that part.

Making Bundle Assets Public¶

The only thing Symfony is doing is helping move these files from their original location inside EventBundle's `Resources/public` directory. But remember from episode 1 that Symfony has an `assets:install` console command. Run this again with a `symlink` option:

```
php app/console assets:install --symlink
```

Note

Symbolically links `src/Yoda/EventBundle/Resources/public` to `web/bundles/event`.

This creates a symbolic link from `web/bundles/event` to that `Resources/public` directory. This is just a cheap trick to expose CSS or JS files to the `web/` directory that live inside a bundle. This lets us point at a real, physical file with the `link` tag.

Tip

The `--sylmink` option may not work on all Windows setups (depending) on your permissions.

You can also just put your CSS and JS files directly into the `web/` directory. In fact, that's a great idea.

The Twig asset Function¶

Take your simple link tag `href` and wrap it in a Twig `asset` function:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% block stylesheets %}
    {# link tag for bootstrap... #}

    <link rel="stylesheet" href="{{ asset('bundles/event/css/event.css') }}" />
    <link rel="stylesheet" href="{{ asset('bundles/event/css/events.css') }}" />
    <link rel="stylesheet" href="{{ asset('bundles/event/css/main.css') }}" />
{% endblock %}
```

I want you to notice that the path isn't changing, except that we don't need the first `/` anymore. When you've got this, refresh. The site still looks great and the HTML source looks exactly as it did before, so `asset` isn't doing anything . . . yet.

Chapter 3: Busting Browser Cache and Using a CDN

BUSTING BROWSER CACHE AND USING A CDN¶

But the `asset` function *does* give us some super-powers, like being able to bust CSS and JS browser cache.

Open up `app/config/config.yml` and find the `framework` `templating` key. Uncomment out the `assets_version` key and set it to your favorite star wars episode:

```
# app/config/config.yml
# ...

framework:
    # ...
    templating:
        engines: ['twig']
        assets_version: 5-return-of-the-jedi
```

Note

We realized later that the number 6 (or even better VI) would have made a *little* bit more sense here...

When we view the source code, we've got a `?5-return-of-the-jedi` at the end of the CSS file paths. That's handy!

```
<link href="/bundles/event/css/main.css?1" rel="stylesheet" />
```

Actually, this query parameter will be at the end of *everything* that uses the the `asset` function. Since browser cache problems suck, increment this number before you deploy and crush the problem. These aren't the assets you're looking for.

If you want to get fancy, add an `assets_version_format` configuration option:

```
# app/config/config.yml
# ...

framework:
    # ...
    templating:
        engines: ['twig']
        assets_version: 5-return-of-the-jedi
        assets_version_format: "%%s?v=%%s"
```

This looks a little funny, but has 2 `%s` placeholders. The first will be filled in with the path to the asset and the second will get the version. Refresh again and check out the path in the source code now:

```
<link href="/bundles/event/css/main.css?1" rel="stylesheet" />
```

Head over to the Reference section of the Symfony docs and click into the `framework` page. This shows you all the options that can live under the `framework` key in `config.yml`.

Find the `assets_version_format`. If you want to go really crazy, you can follow the directions here and create URLs where the version is part of the path, instead of a query parameter. You'd need to do some extra work with rewrite rules to get things to load still, but some CDN's need this type of cache busting.

Using a CDN¶

And on that note, we can use a CDN with pretty much no extra work. Add a new `assets_base_url` key and give it some

imaginary domain:

```
# app/config/config.yml
# ...

framework:
    # ...
    templating:
        engines: ['twig']
        assets_version: 5-return-of-the-jedi
        assets_version_format: "%%s?v=%%s"
        assets_base_url: http://evilempireassets.com
```

Refresh! All the styling is gone, that's great! All the CSS files are prefixed with my make-believe hostname.

```
<link rel="stylesheet"
    href="http://myfancycdn.com/bundles/event/css/event.css?v=5-return-of-the-jedi" />
```

All I'd need to do to make this work is upload my files to this CDN host. And actually, most CDN's support an "origin pull" configuration, where it automatically downloads the files from your real server. There's no uploading involved at all. Super easy.

Take the `http:` part off of the host name and view the source:

```
# app/config/config.yml
# ...

framework:
    # ...
    templating:
        engines: ['twig']
        assets_version: 5-return-of-the-jedi
        assets_version_format: "%%s?v=%%s"
        assets_base_url: //myfancycdn.com
```

```
<link rel="stylesheet"
    href="//myfancycdn.com/bundles/event/css/event.css?v=5-return-of-the-jedi" />
```

This is a valid URL and makes sure that if the user is on an `https` page on your site, that the CSS file is also downloaded via `https`. This avoids the annoying warnings about "non-secure" assets.

Ok, unbreak the site by commenting out this option:

```
# app/config/config.yml
# ...

framework:
    # ...
    templating:
        engines: ['twig']
        assets_version: 5-return-of-the-jedi
        assets_version_format: "%%s?v=%%s"
        # assets_base_url: //myfancycdn.com
```

Chapter 4: Assetic: Filters, Combination and Minification

FILTERING, COMBINING AND OTHER CRAZINESS WITH ASSETIC¶

Life is simple, but things can get crazy with CSS and JS. If you use LESS or SASS, you'll need to process those into CSS before seeing your changes. On deploy, you'll probably also want to combine your CSS into a single file and remove all the extra whitespace to speed up your user's experience. There are also tools like RequireJS, really the list goes on and on.

Frontend Tools: Grunt¶

These days, tools exist outside of PHP to help solve these problems. For example, Grunt is a tool to help you build your assets, like processing through SASS, minifiying and combining. If you're a frontend developer or have one on your team and are comfortable using these tools, go for it. We even have a blog post Evolving RequireJS, Bower and Grunt with code that shows you an approach of using some of this with Symfony.

Assetic: For the Backend Guy¶

But if you're more of a backend dev and just want some help with minifying and combining files, it's all good. Symfony uses a tool called Assetic which makes this *almost* painless :).

Using the stylesheets Tag¶

Open up your base template and add a new `stylesheets` tag. This has the strangest syntax, but should include the path to our 3 CSS files, a filter called `cssrewrite`, and an actual `link` tag. Remove the 3 hard-coded link tags we just added:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% block stylesheets %}
    {# link tag for bootstrap... #}

    {% stylesheets
        'bundles/event/css/event.css'
        'bundles/event/css/events.css'
        'bundles/event/css/main.css'
        filter='cssrewrite'
    %}
        <link rel="stylesheet" href="{{ asset_url }}" />
    {% endstylesheets %}
{% endblock %}
```

Refresh the page. Ok, things still work. Now view the source.

```
<link rel="stylesheet" href="/css/8e49901_event_1.css" />
<link rel="stylesheet" href="/css/8e49901_events_2.css" />
<link rel="stylesheet" href="/css/8e49901_main_3.css" />
```

Hmm. So we still have 3 link tags, but the location has changed. What's even stranger is that these 3 files don't exist - we don't even have a `web/css` directory.

When the browser requests these files, they actually hit our Symfony app and are processed by an internal Assetic controller that renders the CSS code. And I can even prove it!

Run the `router:debug` console task:

```
php app/console router:debug
```

At the top, you'll see actual routes that match the CSS files:

> Name Path _assetic_8e49901_0 /css/8e49901_event_1.css _assetic_8e49901_1 /css/8e49901_events_2.css _assetic_8e49901_2 /css/8e49901_main_3.css

These routes showed up automatically, just by adding the `stylesheets` tag. And if we change any of these CSS files and refresh, these routes will return the updated file.

On the surface, nothing has changed. But the magic is coming...

The cssrewrite Filter¶

Assetic exists for 2 reasons, and the first is to apply filters to your CSS and JS. For example, Assetic has a `less` filter that processes your less files into CSS before returning them.

If you look back at the `stylesheets` tag, you can see that we *do* have one filter called `cssrewrite`.

Open up the generated `event_1.css` file in your browser *and* the original `event.css` in your editor. Now, find the background image for `pinpoint.png` in each. Huh, the paths are a bit different!

The original event.css:

```
background: url(../images/pinpoint.png) no-repeat -5px -7px;
```

The event.css that's served in (generated for) the browser:

```
background: url(../../bundles/event/images/pinpoint.png) no-repeat -5px -7px;
```

Why? In the browser's eyes, the file lives in `/css`, but the original lived in `/bundles/event/css`. If the generated file used the original url, it would point to `/images/pinpoint.png` instead of `/bundles/event/images/pinpoint.png`. The `cssrewrite` filter dynamically changes the url so that things still work. Crazy, right?

This filter is less of a cool feature and more of a necessity. But Assetic supports a number of other filters. As a fair warning, a lot of them aren't documented.

Chapter 5: Combining and Minifying CSS & JS

COMBINING AND MINIFYING CSS & JS¶

The second big feature of Assetic is its ability to combine our CSS or JS into a single file. First, clear your cache and switch over to the `prod` environment:

```
php app/console cache:clear --env=prod
```

```
http://localhost:8000/app.php
```

Things still look nice. But view the source. Woh! Our 3 CSS files are now one:

```
<link rel="stylesheet" href="/css/8e49901.css?v=5-return-of-the-jedi" />
```

Tip

If your page does *not* look fine. that's actually normal! Keep reading about how to dump your assets.

In the `dev` environment, Symfony keeps our 3 files so we can debug more easily. In `prod`, it puts them all together.

More Speed: assetic:dump¶

But when your browser requests this one CSS file, it's still being executed through a dynamic Symfony route. For production, that's *way* too slow. And depending on your setup, it may not even be working in the `prod` environment.

The secret? The `assetic:dump` console command. Run it in the `prod` environment.

```
php app/console assetic:dump --env=prod
```

This wrote a physical file to the `web/css` directory. And when we refresh, the web server loads this file instead of going through Symfony.

When we deploy our application, this command will be part of our deploy process.

Controlling the Output Filename¶

Assetic gave our CSS file a weird name - `8e49901.css` for me, which is just a random name it created. But we can control this by adding an `output` option to the `stylesheets` tag:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% stylesheets
    'bundles/event/css/event.css'
    'bundles/event/css/events.css'
    'bundles/event/css/main.css'
    filter='cssrewrite'
    output='css/built/layout.css'
%}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}
```

Refresh and look at the source. Woops, nothing changed! I can't forget to clear my cache when I'm in the `prod` environment:

```
php app/console cache:clear --env=prod
```

Now the link tag points to this exact spot:

```
<link rel="stylesheet"
   href="/css/built/layout.css?v=5-return-of-the-jedi" />
```

And of course, if we dump assetic, it writes this file instead of the one with the funny name:

```
php app/console assetic:dump --env=prod
```

I also like to put all my built files into `css/built` and `js/built` directories. Add both of these to your `.gitignore` file. There's no need to commit these - we can build them at any time:

```
# .gitignore
# ...

/web/css/built
/web/js/built
```

Chapter 6: Applying a Minification Filter

APPLYING A MINIFICATION FILTER¶

Open up the built CSS file. Ugh. All that nasty whitespace that my user's are going to download. Is there nothing we can do?

Reason #1 to use Assetic was because of its filters, like `cssrewrite`. It also has filters to minify assets. Your best option is to use a binary called uglifycss through Assetic. There's also an `uglify-js`.

Intalling uglifycss with npm¶

We're also going to get a crash-course in `npm`, the Composer for node.js. *Very* Rebel hipster of us.

First, create a nearly empty `package.json` file - this is like the `composer.json` for node libraries:

```
{
}
```

Next, install uglify!

```
npm install uglifycss --save
```

If you don't have `npm`, install node.js to get it. This installs `uglifycss` into a `node_modules` directory. It also updated our `package.json` file to have this library. Another developer on the project only needs to run `npm install` to download uglify. Nice. In fact, let's add `node_modules/` to our `.gitignore` file, just like we did for the `vendor/` directory:

```
# .gitignore
# ...

/node_modules
```

CONFIGURING AND USING THE FILTER¶

The rest is a breeze. Configure the filter in `config.yml` under the `assetic` key. Basically, add an `uglifycss` filter and point it to where the new executable lives:

```
# app/config/config.yml
# ...

assetic:
    # ...
    filters:
        cssrewrite: ~
        uglifycss:
            bin: %kernel.root_dir%/../node_modules/.bin/uglifycss
```

That `node_modules.bin/uglifycss` is a physical binary that was downloaded. The `%kernel.root_dir%` is a parameter that points to `app/`. We'll talk about parameters in a second.

To actually use uglify, add it to the `stylesheets` block:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% stylesheets
    'bundles/event/css/event.css'
    'bundles/event/css/events.css'
    'bundles/event/css/main.css'
    filter='cssrewrite'
    filter='uglifycss'
    output='css/built/layout.css'
%}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}
```

Head back to the `dev` environment and refresh. And when we look at one of the CSS files, no more nasty whitespace.

APPLYING A FILTER ONLY IN THE PROD ENVIRONMENT¶

Ok, I got a little over-excited about whitespace and made working with CSS hell. Our browser thinks that every style is coming from line 1 of these files... because there's only one line in each. Good luck frontend people!

Really, I want the `uglifycss` filter to *only* run in the `prod` environment. We can do just this by adding a `?` before the filter name:

```
{# app/Resources/views/base.html.twig #}
{# ... #}

{% stylesheets
    'bundles/event/css/event.css'
    'bundles/event/css/events.css'
    'bundles/event/css/main.css'
    filter='cssrewrite'
    filter='?uglifycss'
    output='css/built/layout.css'
%}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}
```

Refresh in the `dev` environment. Cool, whitespace restored. Now switch over to the `prod` environment, clear your cache and re-dump the assets:

```
php app/console cache:clear --env=prod
php app/console assetic:dump --env=prod
```

Now, `layout.css` is a physical file *and* has no whitespace. That's perfect.

ASSETIC WITH JAVASCRIPT FILES¶

We just did this all with CSS, but it's all the same with JavaScript. Instead of a `stylesheets` tag, there's a `javascripts` tag that works exactly the same. Symfony has a cookbook entry about this, but seriously, it's no different at all. Even the minification is the same, except that the library is called `uglify-js`.

In other words, you now know pretty much everything you need to about Assetic. If you start using it a lot and notice your pages loading slower and slower, check out the `use_controller` option that's mentioned on that same page.

Ok, back to work!

Chapter 7: Form Template Customizations

FORM THEMING: MAKING FORMS PRETTY(ISH)¶

Where Form Markup comes from¶

In episode 2, we built a registration form. Cool! Open up the `register.html.twig` template for that page. Twig's `form_row` function renders the label, input widget and any errors for each field. And with a few other Twig functions, we can render each part individually. That's all old news, way back from episode 2.

But where does the markup actually come from? Why is the row surrounded in a `div` and the errors in a `ul`?

The answer lives deep inside Symfony, in a file called form_div_layout.html.twig. Open it up in your editor.

Tip

The location of this file is deep inside Symfony in the vendor directory:

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

This odd little file holds a lot of blocks and each renders a different part of the form. There's a block for input fields, labels, errors, and everything else. Every piece of markup for a form is somewhere in here.

Customizing form_row¶

Find the `form_row` block. I know it's shocking, but this is what's used when we call `form_row`.

Let's change it! You should be reminding me that we can't just modify this file. So, let's go with your idea and copy this block and create a new `form_theme.html.twig` file inside `app/Resources/views`. Copy in the block and add your favorite tag to it, just to see if it's working:

```
{# app/Resources/views/form_theme.html.twig #}

{% block form_row %}
    <marquee>It looks like it's working</marquee>
    <div>
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock form_row %}
```

To tell Symfony about this, go to `config.yml` and find the `twig` key. Add `form` and `resources` keys and then the name of this template. Since it lives in `app/Resources`, we use the double-colon syntax, just like when we reference our base template:

```
# app/config/config.yml
# ...

twig:
    # ...
    form:
        resources:
            - "::form_theme.html.twig"
```

Refresh! For some reason my Marquee takes its time, but there it is! Now, we can override *any* of the blocks from Symfony's core `form_div_layout.html.twig` file.

Twitter Bootstrap Form Theming¶

Let's do something useful. A few bundles exist that can help you style your forms for Twitter Bootstrap. Just go to KnpBundles.com and look for them.

To learn a few things, we'll do some of this by hand. Find the Bootstrap Form Docs.

Every field should have a `form-group` div around it. As cool as it is, let's take out the marquee and give the div this class:

```
{# app/Resources/views/form_theme.html.twig #}

{% block form_row %}
    <div class="form-group">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock form_row %}
```

Refresh! It's minor, but we've got a little extra margin now. Let's keep going.

Chapter 8: Error Formatting for Twitter Bootstrap

ERROR FORMATTING FOR TWITTER BOOTSTRAP¶

Submit the form with some bad data. Oh, it's terrible. The errors, they're so ugly. We must fix this.

Go back to `form_div_layout.html.twig`. We don't know which block renders errors, but if you search for the word "errors", you'll find it: `form_errors`.

Copy it into our template:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

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

Here's the plan. Give the `ul` a `help-block` class. This class is from Twitter Bootstrap:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_errors %}
    {% if errors|length > 0 %}
    <ul class="help-block">
        {% for error in errors %}
            <li>{{ error.message }}</li>
        {% endfor %}
    </ul>
    {% endif %}
{% endblock form_errors %}
```

Refresh. It's a very minor improvement, but we've at least modified our second form block. I'll leave the bullet point, but if you want to add some CSS to get rid of it, be my guest. It *is* ugly.

Next, let's see if we can highlight the error message in red. Hardcode a `has-error` field to the div in `form_row`:

```
{# app/Resources/views/form_theme.html.twig #}

{% block form_row %}
    <div class="form-group has-error">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock form_row %}
```

Refresh. This worked, we have red error text but in a second this class is also going to turn the fields red. But we don't want every field to always look like an emergency, so what can we do?

Form Variables: The Holy Grail of Form Rendering Control¶

Inside the `form_errors` block, we have access to some `errors` variable. In fact, in each block we have access to a bunch of variables, like `label`, `value`, `name`, `full_name` and `required`.

Let's use a trick to see *all* of the variables we have access to in `form_errors`:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_errors %}
    {{ dump(_context|keys) }}

    {% if errors|length > 0 %}
    <ul class="help-block">
        {% for error in errors %}
            <li>{{ error.message }}</li>
        {% endfor %}
    </ul>
    {% endif %}
{% endblock form_errors %}
```

Tip

`dump` is a Twig debugging function, like `var_dump`. You can pass it any variable to print it out.

Refresh! For each field, you now see a giant list - for me, 27 things. *All* of these are variables that you magically have access to inside a form theme block. And the variables are the same no matter what block you're in.

Remove the `dump` call. So we can finally use the `errors` variable in `form_row` to *only* print the class if the field has errors:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_row %}
    <div class="form-group {{ errors|length > 0 ? 'has-error' : '' }}">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock form_row %}
{# ... #}
```

Re-submit, fill in some fields correctly. Cool, we still see the red errors, but the other fields are missing this class. That's awesome.

Chapter 9: Adding form-control to the input

ADDING FORM-CONTROL TO THE INPUT¶

Look back at the Bootstrap docs. Every input field should have a `form-control` class. Cool, let's override something else! In `form_div_layout.html.twig`, the block we want is called `form_widget`:

```
{# vendor/symfony/symfony/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig #}
{# ... #}

{% block form_widget %}
{% spaceless %}
    {% if compound %}
        {{ block('form_widget_compound') }}
    {% else %}
        {{ block('form_widget_simple') }}
    {% endif %}
{% endspaceless %}
{% endblock form_widget %}
```

A compound field is one that is actually several fields, like the repeated password we're using on this form. When each individual field is actually rendered, `form_widget_simple` is used.

Copy the block into `form_theme.html.twig`.

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_widget_simple %}
    {% spaceless %}
        {% set type = type|default('text') %}
        <input type="{{ type }}" {{ block('widget_attributes') }} {% if value is not empty %}value="{{ value }}" {% endif %}/>
    {% endspaceless %}
{% endblock form_widget_simple %}
```

One of the variables floating around right now is an array called `attr`. And if it has a `class` key, that'll be printed out by the `widget_attributes` block. Let's add our class to this variable. The code leverages the heck out of Twig. I know it looks strange:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_widget_simple %}
    {% spaceless %}
        {% set attr = attr|merge({ 'class': (attr.class|default('') ~ ' form-control')|trim }) %}
        {% set type = type|default('text') %}
        <input type="{{ type }}" {{ block('widget_attributes') }} {% if value is not empty %}value="{{ value }}" {% endif %}/>
    {% endspaceless %}
{% endblock form_widget_simple %}
```

Before we try this, open up the `login.css` file in `UserBundle` and remove the form-related styles:

```
/* src/Yoda/UserBundle/Resources/public/css/login.css */
/* ... */

.login article h1 {
    margin-top: 0;
    font-family:Arial;
}

/* Remove everything after this */
```

Yes, this will make our login page terrible-looking, but we can add some Bootstrap classes on *that* form later manually, since it doesn't use the form component.

Refresh! Cool! Things are looking better and better.

Adding a Class to the Label¶

Let's do one more thing! The labels *also* need a class: `control-label`. This should be getting easy now. Find the `form_label` block in `form_div_layout.html.twig` but *don't* copy it. Instead, add a blank `form_label` block to our template:

```
{# app/Resources/views/form_theme.html.twig #} {# ... #}

{% block form_label %} {% endblock form_label %}
```

Of course, if we refresh now, the label disappears completely. I want to add a class to the label, but I'd rather not have to copy the *entire* `form_label` block - it's kind of big!

Instead, we can *call* the parent block from inside our template. First, add a Twig `use` tag that points at `form_div_layout.html.twig`:

```
{# app/Resources/views/form_theme.html.twig #} {% use 'form_div_layout.html.twig' with form_label as
base_form_label %}

{# ... #}
```

Now, we can call the parent block inside `form_label`:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_label %}
    {{ block('base_form_label') }}
{% endblock form_label %}
```

Refresh! The labels are back. I know, we're doing craziness with blocks. This is something you'll only see with forms.

But it's also cool! To add a class, just modify the `label_attr` variable, just like we did with `attr`:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_label %}
    {% set label_attr = attr|merge({ 'class': (attr.class|default('') ~ ' control-label')|trim }) %}

    {{ block('base_form_label') }}
{% endblock form_label %}
```

Hey! Now the labels are red, and they will be for *every* form on the site.

Want to know more? You're crazy! Ok, we'll see more cool stuff next. But there's also a cookbook article.

The Block Names (e.g. form_row versus textarea_widget)¶

So far, we've been able to guess which block renders which piece of the form. But there's a science to it.

First, there are 4 parts to any field:

1. label
2. widget
3. errors
4. row

So when you're customizing part of a field, you're always cusotmizing one of these four. That's important because each block name *ends* in the part being modified.

The first part of the block name is the "field type" that you used when building your form. Field types are the things like `text`, `email`, `repeated` and `password`.

Let's put this together. What is the block name to render the "widget" for a "textarea" field type?

Answer? `textarea_widget`. And if you search in Symfony's base template, you'll find this block.

Field type	Which part	Block name
textarea	widget	textarea_widget

So to customize the `errors` of a `textarea` field, you'd look for a `textarea_errors` block. Ah, it doesn't exist!

But there *is* `form_errors` block. Symfony looks for `textarea_errors` first. And if it doesn't find it, it falls back to `form_errors`.

Field type	Which part	Block name
textarea	widget	textarea_widget
textarea	errors	form_errors

Tweak all the things! Just find the right block, copy it into your template, use the variables and customize it.

Chapter 10: More Form Customizations (Form Theming)

CHANGING AND USING FORM VARIABLES¶

So we know that we have access to a bunch of variables from within the form blocks. Awesome.

Overriding Form Variables¶

Open up `register.html.twig`. Remember that `attr` variable we have access to in our form theme blocks? We can override that variable, or any other, right when we render the field. Give the username field a clever class:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

{{ form_row(form.username, {
    'attr': { 'class': 'a-clever-class' }
}) }}
```

Refresh and inspect the field to see the class. In addition to the trick I showed you earlier, Symfony has a reference page called Twig Template Form Function and Variable Reference that lists *most* of these variables. Really you can customize almost anything when rendering a field.

Adding a Help Feature¶

I want to be able to add a little bit of help text beneath any form field. I'll open `form_theme.html.twig` and just hardcode a message in so you can see what I mean:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_row %}
    <div class="form-group {{ errors|length > 0 ? 'has-error' : '' }}">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}

        <div class="help-block">This is the field you're looking for.</div>
    </div>
{% endblock form_row %}
```

I know - it's pointless so far. The same message shows up for every field. How can we customize this?

Inventing a New Form Variable¶

Why not just pass in a new variable? Go back to `register.html.twig` and add a `help` variable to the username field:

```
{# src/Yoda/UserBundle/Resources/views/Register/register.html.twig #}
{# ... #}

{{ form_row(form.username, {
    'attr': { 'class': 'the-username-field' },
    'help': 'Choose something unique and clever'
}) }}
```

In normal Symfony, there is no `help` variable - I totally just made that up. But even though it doesn't normally exist, it *is* being

passed into the form theme blocks. So use it!

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_row %}
    <div class="form-group {{ errors|length > 0 ? 'has-error' : '' }}">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}

        <div class="help-block">{{ help }}</div>
    </div>
{% endblock form_row %}
```

Alright, time to try it. Woh, BIG error:

> Variable “help” does not exist in kernel.root_dir/Resources/views/form_theme.html.twig at line 9

I promise, I wasn’t lying! The problem is that the *other* fields like email and password *aren’t* passing in this variable, so we need to code defensively in the block. Add an `if` statement to make sure the variable is defined and actually set to some real value:

```
{# app/Resources/views/form_theme.html.twig #}
{# ... #}

{% block form_row %}
    <div class="form-group {{ errors|length > 0 ? 'has-error' : '' }}">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}

        {% if help is defined and help %}
            <div class="help-block">{{ help }}</div>
        {% endif %}
    </div>
{% endblock form_row %}
```

Try it again. It works! We can pass in a `help` variable to *any* field on *any* form to use this.

FormView: Customizing Form Variables from your Form Type¶

Ok, but one more challenge. Could we set this help message from inside our form class?

Open up `RegisterFormType`. The `buildForm` method adds the fields and `setDefaultOptions` does exactly that. To customize the form variables directly, create a third method called `finishView`. I’ll use my IDE to generate this for me. Don’t forget the `use` statements for `FormView` and `FormInterface`:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...
use Symfony\Component\Form\FormInterface;
use Symfony\Component\Form\FormView;
// ...

public function finishView(FormView $view, FormInterface $form, array $options)
{

}
```

This method is called right before we start rendering the form. We can use the `FormView` object to change any variable on any field. Use it to add a help message to the email field:

```
// src/Yoda/UserBundle/Form/RegisterFormType.php
// ...

public function finishView(FormView $view, FormInterface $form, array $options)
{
    $view['email']->vars['help'] = 'Hint: it will have an @ symbol';
}
```

Refresh! Yep, you're one dangerous form customizer.

Tip

Most of the core built-in form view variables come from a `FormType::buildView` method: http://bit.ly/sf2-form-build-view

Chapter 11: An Aside: Dependency Injection Parameters

AN ASIDE: DEPENDENCY INJECTION PARAMETERS¶

Cleanse the palette of all the forms stuff and open `config.yml`. Under the `doctrine` key, we see a bunch of percent sign values:

```
# app/config/config.yml
# ...

doctrine:
    dbal:
        driver:   "%database_driver%"
        host:     "%database_host%"
        port:     "%database_port%"
        dbname:   "%database_name%"
        user:     "%database_user%"
        password: "%database_password%"
        # ...
```

Whenever you see something surrounded by two percent signs in a config file, it's a parameter. Parameters are like variables: you set them somewhere and then use them with this syntax. So where are these being set?

Open up `parameters.yml` to find the answer:

```
# app/config/parameters.yml
# ...

# This file is auto-generated during the composer install
    database_driver: pdo_mysql
    database_host: 127.0.0.1
    database_port: null
    database_name: knp_events
    database_user: root
    database_password: null
    # ...
```

In episode 1, we talked about how this file is special because it holds any server-specific configuration. This works because it's in our `.gitignore` file so that every developer and server can have their own. So we set parameters here and use them anywhere else.

Adding More Parameters¶

But technically, we can add parameters to *any* configuration file. Go back to `config.yml` and add a new `parameters` key anywhere in the file. Below it, create a new parameter called `our_assets_version`, and set it to the `assets_version` value we're using below:

```yaml
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }
    - { resource: "@EventBundle/Resources/config/services.yml" }
    - { resource: "@UserBundle/Resources/config/services.yml" }

parameters:
    our_assets_version: 5-return-of-the-jedi

framework:
    # ...
```

Now, just use it under the `framework` key:

```yaml
# app/config/config.yml
# ...

framework:
    # ...
    templating:
        engines: ['twig']
        assets_version: %our_assets_version%
        assets_version_format: "%%s?v=%%s"
    # ...
```

See, they work just like variables. Refresh to make sure we didn't break anything.

So now you know what these percent signs are all about. Spoiler alert! You can also access parameters from a controller using `$this->container->getParameter`, which might come in handy.

Chapter 12: Deployment

DEPLOYMENT: THE ART OF UPLOADING YOUR CODE¶

This wouldn't be much of a tutorial if we didn't at least help show you how to share your project with the world! There are a lot of neat deployment tools out there and I'm sorry, we're not going to show you any of them. At least not in this screencast. Instead, we'll go through the exact steps you'll need for deployment. If you want to automate them, awesome!

To keep things simple, I'm going to "deploy" to a different directory right on my local machine. So, just pretend this is our server and I've already ssh'ed into it.

We already have MySQL, PHP and Apache up and running.

Step 1) Upload the Files¶

First, we've gotta get the files up to the server! The easiest way is just to clone your git repository right on the server. To do this, you'll need to push your code somewhere accessible, like GitHub. The finished code for this tutorial already lives on GitHub, under a branch called `episode4-finish`.

Let's clone this repository:

```
git clone
```

Move into the directory. If your code lives anywhere other than the master branch, you'll need to switch to that branch:

```
git checkout -b episode4-finish origin/episode4-finish
```

GitHub *might* ask you to authenticate yourself or give you some public key error. If that happens, you'll need to register the public key of your server as a deploy key for your repository. This is what gives your server permission to access the code.

GitHub has great articles on deploy keys and generating a public key.

Step 2) Configuring the Web Server¶

Code, check! Next, let's configure the web server. I'm using Apache, but Symfony has a cookbook article about using Nginx. Find your Apache configuration and add a new VirtualHost that points to the `web/` directory of our project. In our case, `/var/www/knpevents.com/web`:

```
<VirtualHost *:80>
    ServerName knpevents.com
    DocumentRoot /var/www/knpevents.com/web

    <Directory /var/www/knpevents.com/web>
        Options Indexes FollowSymlinks
        AllowOverride All

        # Use these 2 lines for Apache 2.3 and below
        Order allow,deny
        allow from all

        # Use this line for Apache 2.4 and above
        Require all granted
    </Directory>

    ErrorLog /var/log/apache2/events_error.log
    CustomLog /var/log/apache2/events_access.log combined
</VirtualHost>
```

The VirtualHost is pretty simple and needs `ServerName`, `DocumentRoot` and `Directory` keys.

Restart your webserver. For many servers, this is done by calling service restart apache:

```
sudo service restart apache2
```

Project: First-Time Setup¶

Code, check! VirtualHost, check!

Since this is the first time we've deployed, we need to do some one-time setup.

First, download Composer and use it to install our vendor files:

```
curl -sS https://getcomposer.org/installer | php
php composer.phar install
```

At the end, it'll ask you for values to fill into your `parameters.yml` file. You'll need to have a database user and password ready.

Speaking of, let's create the database and insert the schema. I'll even run the fixtures to give our site some starting data:

```
php app/console doctrine:database:create
php app/console doctrine:schema:create
php app/console doctrine:fixtures:load
```

In this pretend scenario, I've already pointed the DNS for knpevents.com to my server. So let's try it:

> http://knpevents.com

It's alive! And with a big error, which might just show up as the white screen of death on your server. Symfony can't write to the cache directory. We need to do a one-time `chmod` on it and the `logs` dir:

```
sudo chmod -R 777 app/cache/ app/logs/
```

Let's try again. Ok, we have a site, and we can even login as Wayne. But it's missing all the styles. Ah, right, dump the assetic assets:

```
php app/console assetic:dump --env=prod
```

Crap! Scroll up. This failed when trying to run uglifycss. I don't have Uglifycss installed on this machine yet. To get ugly Just run `npm install` to fix this.

```
php app/console assetic:dump --env=prod
```

Now, the dump works, AND the site looks great!

Things to do on each Deploy¶

On your next deploy, things will be even easier. Here's a simple guide:

1. Update your Code. With our method, that's as simple as running a git pull:

```
git pull origin
```

2. Just in case we added any new libraries to Composer, run the install command:

```
php composer.phar install
```

3. Update your database schema. The easy, but maybe dangerous way is with the schema update console command:

```
php app/console doctrine:schema:update --force
```

Why dangerous? Let's say you rename a property from `name` to `firstName`. Instead of renaming the column, this task may just `drop` name and add `firstName`. That would mean that you'd lose all that data!

There's a library called Doctrine Migrations that helps do this safely.

4. Clear your production cache:

```
php app/console cache:clear --env=prod
```

5. Dump your Assetic assets:

```
php app/console assetic:dump --env=prod
```

That's it! As your site grows, you may have more and more things you need to setup. But for now, it's simple.

Performance Setup you Need¶

One more thing. There are a few really easy wins to maximize Symfony's performance.

First, when you deploy, dump Composer's optimized autoloader:

```
php composer.phar dump-autoload --optimize
```

This helps Composer's autoloader find classes faster, sometimes much faster. And hey, there's no downside at all to this!

Tip

> If you add the `–optimize-autoloader` flag, Composer will generate a class map, which will give your whole application a performance boost. Using the APC ClassLoader may give you an even bigger boost.

Next, make sure you have a byte code cache installed on your server. For PHP 5.4 and earlier, this was called APC. For 5.5 and later, it's called OPcache. In the background, these cache the compiled PHP files, making your site *much* faster. Again, there's no downside here - make sure you have one of these on your server.

And on that note, PHP typically gets faster from version to version. So staying on the latest version is good for more than just security and features. Thanks PHPeeps!

Ok, that's it! Now google around for some deployment tools to automate this!

Chapter 13: Goodbye Friend!

GOODBYE FRIEND!¶

Young Jedi, now that you know how to deploy your application, why are you still listening to me?

Seriously, thanks for joining me, I'm excited to see what you'll build! You've touched on just about every part of Symfony, including some more advanced topics. So start coding!

Of course, you'll certainly run into new problems that will require new solutions. When you do, be sure to check out Symfony's cookbook, which is packed with articles on specific, and often much more advanced topics.

We also hope that you'll join us again in the future as we cover more PHP and Symfony topics. Have an idea? We'd love to hear it.

Thank you, and see ya next time!