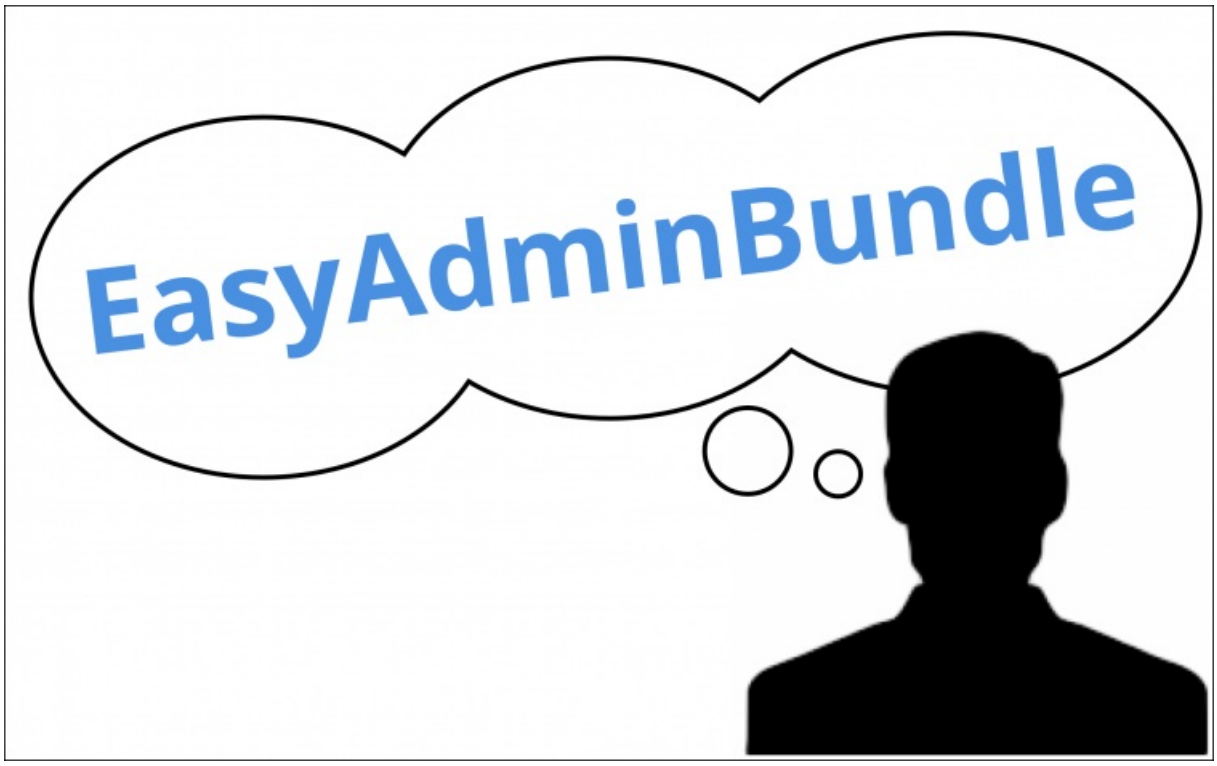
EasyAdminBundle for an Amazing Admin Interface



With <3 from SymfonyCasts

Chapter 1: Installation and First Admin

Hello! And welcome to another *amazing* episode of: "Recipes with Ryan". Today we'll be baking a positively mind-blowing, mile-high cinnamon bread!

Huh? It's not the recipe video? EasyAdminBundle? Ok, cool!

Um... we'll be baking a positively mind-blowing admin interface in Symfony with the wonderful EasyAdminBundle.

What about SonataAdminBundle

Wait, but what about SonataAdminBundle? SonataAdminBundle *is* super powerful... more powerful than EasyAdminBundle. But it's also a bit of a beast - a lot bigger and more difficult to use. If it has a feature that you *need*... go for it! Otherwise, jump into the easy side with EasyAdminBundle.

Code with Me!

To make your cinnamon bread a hit, code along with me. Download the course code from this page and unzip it. Inside, you'll find a directory called `start/`, which will have the same code you see here. Open `README.md` for step-by-step baking instructions... and also details on how to get your project setup!

The last step will be to find your favorite terminal, move into the project and run:

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

to start the built-in web server. Find your fanciest browser and open that: http://localhost:8000. Welcome to AquaNote! The project we've been building in our Symfony series.

Actually, in that series, we spent some serious time making an admin area for one of our entities: Genus. Go to `/admin/genus`... and then login: username `weaverryan+1@gmail.com`, password: `iliketurtles`.

A genus is a type of animal classification. And after all our work, we can create and edit them really nicely.

That's great... but now I need an admin interface for a bunch of other entities: `GenusNote`, `SubFamily`, and `User`. Doing that by hand... well... that would take a while... and we've got baking to do! So instead, we'll turn to EasyAdminBundle.

Installing EasyAdminBundle

Google for EasyAdminBundle and find its GitHub page. Ah, it's made by our friend Javier! Hi Javier! Let's get this thing installed. Copy the `composer require` line, go back to your terminal, open a new tab, and paste:

```
$ composer require javiereguiluz/easyadmin-bundle
```

While Jordi is downloading that package, let's keep busy!

Copy the new bundle line, find `app/AppKernel.php`, and put it here!

```
61 lines | app/AppKernel.php
... lines 1 - 2
use JavierEguiluz\Bundle\EasyAdminBundle\EasyAdminBundle;
... lines 4 - 6
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 12 - 25
            new EasyAdminBundle(),
        );
... lines 28 - 38
    }
... lines 40 - 59
}
```

Unlike most bundles, this bundle actually gives us a new *route*, which we need to import. Copy the routing import lines, find our `app/config/routing.yml` and paste anywhere:

```
14 lines | app/config/routing.yml
... lines 1 - 9
easy_admin_bundle:
    resource: "@EasyAdminBundle/Controller/"
    type: annotation
    prefix: /easyadmin
```

Since we already have some pages that live under `/admin`, let's change the prefix to `/easyadmin`:

```
14 lines | app/config/routing.yml
... lines 1 - 9
easy_admin_bundle:
... lines 11 - 12
    prefix: /easyadmin
```

Finally, one last step: run the `assets:install` command. This should run automatically after Composer is finished... but just in case, run it again:

```
$ php bin/console assets:install --symlink
```

This bundle comes with some CSS and JavaScript, and we need to make sure it's available.

Setting up your First Admin

Ok, we are installed! So... what did we get for our efforts? Try going to `/easyadmin`. Well... it's not much to look at yet... but it will be! I promise! We just need to configure what admin sections we need... and Javier gave us a great error message about this!

In a nut shell, EasyAdminBundle can create an admin CRUD for any entity with almost zero configuration. In the docs, it shows an example of this minimal config. Copy that, find `config.yml`, scroll to the bottom, and paste. Change these to our entity names: `AppBundle\Entity\Genus`, `AppBundle\Entity\GenusNote`, `AppBundle\Entity\SubFamily`, skip `GenusScientist` - we'll embed that inside one of the other forms, and add `AppBundle\Entity\User`:

87 lines | app/config/config.yml

```yaml
... lines 1 - 80
easy_admin:
    entities:
        - AppBundle\Entity\Genus
        - AppBundle\Entity\GenusNote
        - AppBundle\Entity\SubFamily
        - AppBundle\Entity\User
```

We have arrived... at the moment of truth. Head back to your browser and refresh Ah, hah! Yes! A full CRUD for each entity: edit, delete, add, list, show, search, party! For a wow factor, it's even responsive: if you pop it into iPhone view, it looks pretty darn good!

This is exactly what I want for my admin interface: I want it to be amazing and I want it to let me be lazy!

Of course the trick with this bundle is learning to configure and extend it. We're 80% of the way there with no config... now let's go further.

Chapter 2: Design Config & Security Setup

With about six lines of code, we got a fully functional admin area. It must be our birthday. But now... we need to learn how to take control and *really* customize!

And for most things... it's... um... easy... because EasyAdminBundle let's us control almost anything via configuration. Back on its README, scroll up and click to view the full docs.

The bundle has great docs, so we'll mostly cover the basics and then dive into the really tough stuff. Let's start with design.

Right now, in the upper left corner, it says, "EasyAdmin"... which is probably *not* what I want. Like most stuff, this can be changed in the config. Add a `site_name` key set to `AquaNote`:

88 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
    site_name: 'AquaNote'
... lines 83 - 88
```

Actually, to be fancy, add a bit of HTML in this:

88 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
    site_name: 'Aqua<i>Note</i>'
... lines 83 - 88
```

Refresh! Sweet! What else can we do?

The design Config Key

Well, eventually, we'll learn how to override the EasyAdminBundle templates... which pretty much means you can customize anything. But a lot of things can be controlled here, under a `design` key.

For example, `brand_color`. This controls the blue used in the layout. Set it to `#819b9a` to match our front-end a bit better:

90 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
... line 82
    design:
        brand_color: '#81b9ba'
... lines 85 - 90
```

Try that! Got it!

But what if we need to change something more specific... like if we want the branding name to be a darker gray color? Let's see... that means we want to set the color of the `.main-header .logo` anchor tag. So... how can we do that? The way you normally would: in CSS. Create a new file in `web/css`: `custom_backend.css`. Add the `.main-header .logo` and make its color a bit darker:

4 lines | web/css/custom_backend.css

```
.main-header .logo {
    color: #3a3a3a;
}
```

Simple... but how do we include this on the page... because we don't have control over any of these templates yet. Well, like most things... it's configurable: add `assets`, `css` then pass an array with the path: `css/custom_backend.css`:

```
92 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... line 84
        assets:
            css: ['css/custom_backend.css']
... lines 87 - 92
```

And yes! There *is* a `js` key and it works the same. We'll use it a bit later. Refresh to see our sweet styling skills. Woohoo!

There are a few other keys under `design` and we'll use some of them. But they're all pretty simple and this stuff is documented under "Basic configuration" and "Design Configuration".

Adding Security

Before we keep going... we need to talk security! Because right now, I can log out and go back to `/easyadmin` with no problem. This is *totally* open to the public. Fun!

How can we configure security in EasyAdminBundle? We don't! This is just a normal page... so we can use Symfony's normal security to protect it.

The easiest way is in `security.yml` via `access_control`:

```
41 lines | app/config/security.yml
... lines 1 - 2
security:
... lines 4 - 38
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
```

Uncomment the example, use `^/easyadmin` and check for `ROLE_ADMIN`:

```
41 lines | app/config/security.yml
... lines 1 - 2
security:
... lines 4 - 38
    access_control:
        - { path: ^/easyadmin, roles: ROLE_ADMIN }
```

That is it!

When we refresh... yep! We're bounced back to the login page. Log back in with `weaverryan+1@gmail.com` password `iliketurtles`. And we're back! We *can* also secure things in a more granular way... and I'll show you how later.

Now, let's start customizing the list page.

Chapter 3: Views & entities Config

With EasyAdminBundle, you can configure just about *everything*... in multiple different ways. It's *great*! But also confusing. So, let's get it straight!

There are two different axis for configuring things. First, every entity has 5 different "views": `list` - which is this one - `search`, `new`, `edit` and `show`. Each view can be configured. Second, each *entity* can *also* be configured. And sometimes, these overlap: you can tweak something for all list views, but then override that list tweak for *one* specific entity.

Global list Config

Let's see this in action! In `config.yml`, under `easy_admin`, you can configure the list view by adding a `list` key. Set `title: 'List'`:

```
94 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 86
    list:
        title: 'List'
... lines 89 - 94
```

And yep! *Each* view can be configured like this, by adding `search`, `show`, `edit` or `new`. Some config, like `title`, will work under all of these. But mostly, each section has its own config.

When you add `list` at this root level, it applies to *all* entities. Try it out: yes! We just added a boring title to *all* the list pages!

For more context, add a magic wildcard: `%%entity_label%%`:

```
94 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 86
    list:
        title: 'List of %%entity_label%%'
... lines 89 - 94
```

Try it! Much better!

There are just a *few* magic wildcards: `entity_label`, `entity_name` and `entity_id`. The `entity_name` is the geeky, machine-name for your entity - like `GenusNote`. The `entity_label` defaults to that same value... but we can change it to something better.

Configuring at the Entity Level

So far, we just have a simple list of all the entity admin sections we want. That's great to get started... but not anymore! As *soon* as you need to configure an entity further, you need to use an *expanded* format. Basically, instead of `- Genus`, use `Genus` as the key and add a new line with `class` set to the full class name: `AppBundle\Entity\Genus`:

```
98 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 88
    entities:
        Genus:
            class: AppBundle\Entity\Genus
... lines 92 - 98
```

Repeat this for everything else: `GenusNote`, `SubFamily` and `User`:

```
98 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 88
    entities:
        Genus:
            class: AppBundle\Entity\Genus
        GenusNote:
            class: AppBundle\Entity\GenusNote
        SubFamily:
            class: AppBundle\Entity\SubFamily
        User:
            class: AppBundle\Entity\User
```

This didn't change anything... but now we are, oh yes, dangerous! Oh, *so* much can be configured for each entity. Start simple: `label: Genuses`:

```
99 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 88
    entities:
        Genus:
            class: AppBundle\Entity\Genus
            label: Genuses
... lines 93 - 99
```

Try that! Nice! The label is of course used in the title, but also in the navigation.

Overriding the List view under an Entity

And here's where things get *really* interesting. The `list` config we added applies to *all* entities. But we can also customize the `list` view for just *one* entity. Under `GenusNote`, add a `label: 'Genus Notes'`:

```
102 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 88
    entities:
... lines 90 - 92
        GenusNote:
            class: AppBundle\Entity\GenusNote
            label: Genus Notes
... lines 96 - 102
```

But more importantly, add `list`, then `title: 'List of notes'`:

```
102 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 88
    entities:
... lines 90 - 92
        GenusNote:
            class: AppBundle\Entity\GenusNote
            label: Genus Notes
            list:
                title: List of notes
... lines 98 - 102
```

Ok, check this out! The left navigation uses the new label. But the list page's title is `List of notes`.

Woohoo! To review: there are *5* views, and each view can be configured globally, but also beneath each entity. If that makes sense, you're in *great* shape.

The Search View

While we're here, go type a few letters into that search box. Yep! This is the `search` view. It's almost identical to `list`: it re-uses its template and has almost identical config keys.

Overriding Entity View Config

In addition to `title`, one other key that *every* view has is `help`. First, set this below the `Genus` section: "Genuses are not covered by warranty!":

```
103 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 88
89      entities:
90          Genus:
... lines 91 - 92
93              help: Genuses are not covered under warranty!
... lines 94 - 103
```

Notice, this is directly under the *entity* config, not under a specific view. Thanks to this, it will apply to *all* views for this entity. And... yep! It's at the top of the search page, on list and on edit.

In the spirit of EasyAdminBundle, you can override this for each view below. For `list`, set `help` to `Do not feed!`:

```
105 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 88
89      entities:
90          Genus:
... lines 91 - 92
93              help: Genuses are not covered under warranty!
94              list:
95                  help: Do not feed!
... lines 96 - 105
```

Nice!

And the `search` view still uses the default message. If you want to turn *off* the help message here, you can totally do that. Under `search`, set `help` to `null`:

```
107 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 88
89      entities:
90          Genus:
... lines 91 - 92
93              help: Genuses are not covered under warranty!
94              list:
95                  help: Do not feed!
96              search:
97                  help: null
... lines 98 - 107
```

Refresh! Gone!

Ok, so we're not going to talk about *all* of the different keys available: it's all pretty easy. The most important thing is to realize

that each of the 5 views can be configured on two different levels.

Next, let's talk about actions and the different ones that we can enable.

Chapter 4: Actions Config

We now know that there are *5* views: `list`, `search`, `edit`, `new` and ~~banana~~... `show`. EasyAdminBundle also has an idea of "actions", which are basically the *links* and buttons that show up on each view. For example, on `list`, we have a `search` action, a `new` action and `edit` & `delete` actions. In the docs, they have a section called Actions Configuration that shows the "default" actions for each view. For example, from the "Edit" view, you have a delete button and a list link... but you do *not* have a link to create a new entity or eat a banana.

These actions can be customized a lot: we can add some, take some away, tweak their design and - gasp - create custom actions.

Adding the show Action

In the docs, it says that the `list` view does *not* have the `show` action by default. Yep, there's no little "show" link next to each item. *If* you want that, add it! Under the global `list`, add `actions`, then `show`:

```
108 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 86
    list:
... line 88
        actions: ['show']
... lines 90 - 108
```

We *only* need to say `show`, we don't need to re-list all the actions. That's because the, `actions` configuration is a *merge*: it takes all of the original actions and *adds* whatever we have here.

Adding __toString() Methods

So... how would we *remove* an action? We'll get there! But first, if you click, "Show"... wow. Geez. Dang! The page is *very* broken:

> Object of class `GenusNote` could not be converted to string.

In a *lot* of places, EasyAdminBundle tries to convert your entity objects into strings... ya know, so it can print them in a list or create a drop-down menu. To get this working, we need to add `__toString()` methods to each entity. Let's do that real quick!

In `Genus`, add `public function __toString()` and return `$this->getName()`:

```
243 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 17
class Genus
{
... lines 20 - 237
    public function __toString()
    {
        return (string) $this->getName();
    }
}
```

I'm casting this into a string just to be safe - if it's null, you'll get an error. In `GenusNote`, do the same thing: return `$this->getNote()`:

```
121 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 10
class GenusNote
{
... lines 13 - 115
    public function __toString()
    {
        return (string) $this->getNote();
    }
}
```

In `GenusScientist`, we can return `$this->getUser()`:

```
85 lines | src/AppBundle/Entity/GenusScientist.php
... lines 1 - 17
class GenusScientist
{
... lines 20 - 79
    public function __toString()
    {
        return (string) $this->getUser();
    }
}
```

That's an object... but we're about to add a `__toString()` for it too!

`SubFamily`, well hey! It already has a `__toString()`. I'll just add the string cast:

```
45 lines | src/AppBundle/Entity/SubFamily.php
... lines 1 - 10
class SubFamily
{
... lines 13 - 39
    public function __toString()
    {
        return (string) $this->getName();
    }
}
```

And finally, in `User` ... make this a bit fancier. Return `$this->getFullName()` or `$this->getEmail()`, in case the user doesn't have a first or last name in the database:

```
271 lines | src/AppBundle/Entity/User.php
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 233
    public function __toString()
    {
        return (string) $this->getFullName() ? $this->getFullName() : $this->getEmail();
    }
... lines 238 - 269
}
```

Try the show page again! Nice! It renders *all* of the properties, including the *relations*, which is why it needed that `__toString()` method.

And because we have a `SubFamily` admin section, the `SubFamily` is a link that takes us to its `show` view.

Removing Actions

Speaking of `SubFamily`, as you can see... there's not much to it: just an id and a name. And the "show" view, well, it's just the id and name again. Not too interesting. In this case, I think it's overkill to have the `show` action for the `SubFamily` entity. So let's kill it!

Back in `config.yml`, we just added the, `show` action to the `list` view globally. Now, under `SubFamily`, we can *override* that `list` config. Add `actions` and - to *remove* an action - use `-show`:

```
110 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 103
104         SubFamily:
... line 105
106             list:
107                 actions: ['-show']
... lines 108 - 110
```

Yep, use "minus" to take an action away.

Refresh! Ah, ha! The show link is gone.

Disabling Actions

But... even though the link is gone, you can *totally* still get to the show page! For example, if we click, "Edit", we can be annoying and change the `action` in the URL to `show`. Genus!

Or... you can just go to the show page for any `Genus`: there is *still* a link to the `SubFamily` show page.

That might be ok, but if you *truly* want to disable the show view, there's a special config key for that. Under the entity itself, add `disabled_actions` set to an array with `show` inside:

```
111 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 103
104         SubFamily:
... line 105
106             list:
107                 actions: ['-show']
108             disabled_actions: ['show']
... lines 109 - 111
```

As *soon* as we do that ... the link vanishes in dramatic fashion! Let's be annoying again: I'll go forward in my browser to get back to the show page URL. Refresh! Dang! Now we get a huge error. The show view is *totally* gone.

Customizing Action Design

There are two more things you can do with actions: custom actions - we'll talk about those later - and making your actions pretty. That's probably even more important!

I want to tweak how the list actions look... but *only* for the `Genus` section. Ok, find its config, go under `list` and add `actions`:

```
114 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
        Genus:
... lines 92 - 94
            list:
... line 96
                actions:
... lines 98 - 114
```

This time, rather than adding or removing actions, we want to *customize* them. So instead of using a simple string like `show`, use an expanded configuration with `name: edit`. We are now proudly configuring the `edit` action.

There are a few things we can do here, like `icon: pencil` and `label: Edit`:

```
114 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
        Genus:
... lines 92 - 94
            list:
... line 96
                actions:
                    - { name: 'edit', icon: 'pencil', label: 'Edit' }
... lines 99 - 114
```

The `icon` option - which shows up in a few places - allows you to add Font Awesome icons. For the value, just use whatever comes *after* the `fa-`. So, to get the `fa-pencil` icon, say `pencil`.

Make the `show` action just as fancy, with `icon: info-circle` and a *blank* label:

```
114 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
        Genus:
... lines 92 - 94
            list:
... line 96
                actions:
                    - { name: 'edit', icon: 'pencil', label: 'Edit' }
                    - { name: 'show', icon: 'info-circle', label: '' }
... lines 100 - 114
```

OoooOooo. Refresh to see how that looks!

Oh man, it's actually kind of ugly... I need to work on my styling skills. But it totally works!

Chapter 5: Configuring the List Fields

We've been tweaking a bunch of stuff on the list view. But... what about this big giant table in the middle!? How can we customize that?

Actually, EasyAdminBundle did a pretty good job with it: it guesses which fields to show, humanizes the column header labels and renders things nicely. Good job Javier!

The EasyAdminBundle Profiler

Before we tweak all of this, see that magic wand in the web debug toolbar? Say "Alohomora" and click to open that in a new tab. This is the `EasyAdminBundle` profiler... and it's *awesome*. Here, under "Current entity configuration", we can see all of the config we've been building for this entity, *including* default values that it's guessing for us. This is a *sweet* map for knowing what can be changed and how.

Under `list`, then `fields`, it shows details about all the columns used for the table. For example, under `name`, you can see `type => string`. Actually, `dataType` is the really important one.

Here's the deal: each field that's rendered in the table has a different *data type*, like `string`, `float`, `date`, `email` and a bunch others. EasyAdminBundle guesses a type, and it affects how the data for that field is rendered. We can change the type... and *anything* else you see here.

Controller Fields

How? Under `Genus` and `list`, add `fields`. Now, list the exact fields that you want to display, like `id`, `name` and `isPublished`:

121 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
        Genus:
... lines 92 - 94
            list:
... lines 96 - 99
                fields:
                    - 'id'
                    - 'name'
                    - 'isPublished'
... lines 104 - 121
```

These 3 fields were already shown before.

Let's also show `firstDiscoveredAt`... but! I want to tweak it a little. Just like with actions, there is an "expanded" config format. Add `{ }` with `property: firstDiscoveredAt`.

Now... what configuration can we put here? Because this is a `date` field, it has a `format` option. Set it to `M Y`. And, *all* fields have a `label` option. Use "Discovered":

```
121 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
91          Genus:
... lines 92 - 94
95              list:
... lines 96 - 99
100                 fields:
101                     - 'id'
102                     - 'name'
103                     - 'isPublished'
104                     - { property: 'firstDiscoveredAt', format: 'M Y', label: 'Discovered' }
... lines 105 - 121
```

Keep going! Add `funFact` and then one more expanded property: `property: speciesCount`. This is an `integer` type, which also has a `format` option. For fun, set it to `%b` - binary format!

```
121 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
91          Genus:
... lines 92 - 94
95              list:
... lines 96 - 99
100                 fields:
101                     - 'id'
102                     - 'name'
103                     - 'isPublished'
104                     - { property: 'firstDiscoveredAt', format: 'M Y', label: 'Discovered' }
105                     - 'funFact'
106                     - { property: 'speciesCount', format: '%b' }
... lines 107 - 121
```

Yea know because, scientists are nerds and like puzzles.

Tip

The `format` option for number fields is passed to the `sprintf()` function.

If your head is starting to spin with all of these types and options that I'm pulling out of the air, don't worry! Almost *all* of the options - like `label` - are shared across all the types. There are very few type-specific options like `format`.

And more importantly, in a few minutes, we'll look at a list of *all* of the valid types and their options.

Ok! Close the profiler tab and refresh. Bam! The table has our 6 columns!

Customizing the Search View

Try out the search again: look for "quo". Ok nice! Without any work, the search view re-uses the `fields` config from list.

You *can* add a `fields` key under `search`, but it means something different. Add `fields: [id, name]`:

```yaml
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
        Genus:
... lines 92 - 106
            search:
... line 108
                fields: ['id', 'name']
... lines 110 - 122
```

Out-of-the-box, the bundle searches *every* field for the search string. You can see that in the queries. But now, it *only* searches `id` and `name`.

Next, let's dive into some of the more interesting field types and their config.

Chapter 6: More about List Field Types

Navigate to the `User` entity section... open up the EasyAdminBundle profiler... and check out the list fields.

I want to talk a bit more about these field "data types". Check out `isScientist`. Its data type is set to `toggle`, my favorite type! Go back to the main page of the documentation and open List, Search and Show Views Configuration.

List Field Types and Options

Down the page a little, it talks about how to "Customize the Properties Appearance". This is *great* stuff. First, it lists the valid *options* for all the fields, like `property`, `label`, `css_class`, `template` - which we'll talk about later - and yes! The `type`, which controls that `dataType` config. There are a *bunch* of built-in types, including all of the `Doctrine` field types and a few special fancy ones from EasyAdminBundle, like *toggle*. The "toggle" type is actually *super* cool: it renders the field as a little switch that turns the value on and off via AJAX.

Changing to the boolean data type

Let's take control of the User fields. Below that entity, add `list`, then `fields`, with `id` and `email`:

130 lines | app/config/config.yml

```
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 119
120         User:
... line 121
122             list:
123                 fields:
124                     - id
125                     - email
... lines 126 - 130
```

Let's customize `isScientist` and set its label to `Is scientist?`. And as *cool* as the toggle field is, for the great sake of learning, change it to `boolean`:

130 lines | app/config/config.yml

```
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 119
120         User:
... line 121
122             list:
123                 fields:
124                     - id
125                     - email
126                     - { property: 'isScientist', label: 'Is scientist?', type: 'boolean' }
... lines 127 - 130
```

Then add, `firstName`, `lastName`, and `avatarUri`:

```
130 lines | app/config/config.yml
... lines 1 - 80
81 easy_admin:
... lines 82 - 89
90     entities:
... lines 91 - 119
120         User:
... line 121
122             list:
123                 fields:
124                     - id
125                     - email
126                     - { property: 'isScientist', label: 'Is scientist?', type: 'boolean' }
127                     - firstName
128                     - lastName
129                     - avatarUri
```

Try that! Ok! The `isScientist` field is now a "less-cool" Yes/No label. Open up the `EasyAdminBundle` config to see the difference. Under list... fields... `isScientist`, yep! `dataType` is now boolean... *and* it's using a different template to render it. More on that later.

Virtual Fields

Back in the config, obviously, these are all property names on the `User` entity. But... that's not required. As long as there is a "getter" method, you can invent new, "virtual" fields. Our `User` does *not* have a `fullName` property... but it *does* have a `getFullName()` method. So, check this out: remove `firstName` and `lastName` and replace it with `fullName`:

```
129 lines | app/config/config.yml
... lines 1 - 80
81 easy_admin:
... lines 82 - 89
90     entities:
... lines 91 - 119
120         User:
... line 121
122             list:
123                 fields:
... lines 124 - 126
127                     - fullName
... lines 128 - 129
```

Try that out! Magic!

The email and url Fields

As we saw earlier, EasyAdminBundle also has a few *special* types. For example, expand the `email` property and set its type to `email`:

```
129 lines | app/config/config.yml
... lines 1 - 80
81 easy_admin:
... lines 82 - 89
90     entities:
... lines 91 - 119
120         User:
... line 121
122             list:
123                 fields:
... line 124
125                     - { property: 'email', type: 'email' }
... lines 126 - 129
```

While we're here, do the same for `avatarUri`, setting it to `url`:

```
129 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 119
120         User:
... line 121
122             list:
123                 fields:
... line 124
125                     - { property: 'email', type: 'email' }
... lines 126 - 127
128                     - { property: 'avatarUri', type: 'url' }
```

Try that! I know, it's not earth-shattering, but it is nice: the email is now a `mailto` link and the `avatarUri` is a link to open in a new tab.

The image Type

Of course, `avatarUri` is an image... so it would be *way* trendier to... ya know... actually render an image! Yea! But let's do it somewhere else: go to the `GenusNote` section. Then, in `config.yml`, under the entity's `list` key - add `fields`. Let's show `id` and `username`:

```
135 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 109
110         GenusNote:
... lines 111 - 112
113             list:
... line 114
115                 fields:
116                     - id
117                     - username
... lines 118 - 135
```

One of the fields is called the `userAvatarFileName`, which is a simple text field that stores an image filename, like `leanna.jpeg` or `ryan.jpeg`. I want that to show up as an image thumbnail. To do that, add `property: userAvatarFilename`, `label: User avatar` and... `type: image`:

```
135 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 109
110         GenusNote:
... lines 111 - 112
113             list:
... line 114
115                 fields:
116                     - id
117                     - username
118                     - { property: 'userAvatarFilename', label: 'User avatar', type: 'image' }
... lines 119 - 135
```

Before we try that, also add `createdAt` and `genus`:

```
135 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 109
110         GenusNote:
... lines 111 - 112
113             list:
... line 114
115                 fields:
116                     - id
117                     - username
118                     - { property: 'userAvatarFilename', label: 'User avatar', type: 'image' }
119                     - createdAt
120                     - genus
... lines 121 - 135
```

Actually, `genus` is the property that points to the related `Genus` *object*, which is pretty cool:

```
121 lines | src/AppBundle/Entity/GenusNote.php
... lines 1 - 10
11  class GenusNote
12  {
... lines 13 - 43
44      /**
45       * @ORM\ManyToOne(targetEntity="Genus", inversedBy="notes")
46       * @ORM\JoinColumn(nullable=false)
47       */
48      private $genus;
... lines 49 - 100
101     public function getGenus()
102     {
103         return $this->genus;
104     }
105
106     public function setGenus(Genus $genus)
107     {
108         $this->genus = $genus;
109     }
... lines 110 - 119
120 }
```

That will *totally* work because our `Genus` class has a `__toString()` method:

```
243 lines | src/AppBundle/Entity/Genus.php
... lines 1 - 17
18  class Genus
19  {
... lines 20 - 237
238     public function __toString()
239     {
240         return (string) $this->getName();
241     }
242 }
```

Refresh! Ok, it *kinda* works... there *is* an image tag! Yea... it's broken, but let's try to be positive! Right click to open that in a new tab. Ah, it's look for the image at `http://localhost:8000/leanna.jpeg`. In our simple system, those images are actually stored in a

`web/images` directory. In a more complex app, you might store them in an `uploads/` directory or - even better - up on something like S3. But no matter where you store your images, you'll need to configure this field to point to the right path.

How? Via a special option on the `image` type: `base_path` set to `/images/`:

135 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
... lines 91 - 109
        GenusNote:
... lines 111 - 112
            list:
... line 114
                fields:
... lines 116 - 117
                    - { property: 'userAvatarFilename', label: 'User avatar', type: 'image', base_path: '/images/' }
... lines 119 - 135
```

You can of course also use an absolute URL.

Try it! There it is! And it's even got a fancy lightbox.

Next up, let's finish talking about the list view by taking crazy control of filtering and ordering.

Chapter 7: DQL Filtering & Sorting

What else could we *possibly* configure with the list view? How about sorting, or *filtering* list via DQL. OoooOOooo.

Configuring Sort

First, sorting... which we get for free. Already, the genuses are sorted by id, but we can click any column to sort by that. But this isn't sticky: when you come back to the genus list page, it's back to filtering by id.

Sorting by *name* would be a bit more awesome. And you can probably *guess* what the config looks like to do this. Under `Genus` and `list`, add `sort: name`:

```
136 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
91          Genus:
... lines 92 - 94
95              list:
... lines 96 - 106
107                 sort: 'name'
... lines 108 - 136
```

This is the *new* default field for sorting.

Sorting via Relations

Oh, but we can get fancier. Under `GenusNote`, what if I told you I wanted to sort by the name of the `Genus` it's related to? Yea, that would mean sorting *across* a relation. But that's *totally* possible: `sort: ['genus.name', 'ASC']`:

```
137 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 110
111         GenusNote:
... lines 112 - 113
114             list:
... lines 115 - 121
122                 sort: ['genus.name', 'ASC']
... lines 123 - 137
```

This also controls the direction. It sorts descending by default.

Try it! Nice! This works... just don't get *too* confident and try to do this across *multiple* relationships... that's not going to work.

Disabling Sort Fields

The ability to sort via *any* field with no setup is great! Though... sometimes it doesn't make sense - like with the "User avatar" field. To tighten things up, you can disable sorting. Find that field's `list` config and add a new option at the end: `sortable: false`:

```
137 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 110
111         GenusNote:
... lines 112 - 113
114             list:
... line 115
116                 fields:
... lines 117 - 118
119                     - { property: 'userAvatarFilename', label: 'User avatar', type: 'image', base_path: '/images/', sortable: false }
... lines 120 - 137
```

And... gone!

DQL Filtering

Ok, let's turn to something fun: DQL filtering. Like, what if we want to *hide* some genuses entirely from the list and search page?

But first, so far, it *seems* like we're limited to one entity section per entity. That's a lie! Let me show you: add a new section under `entities` called `GenusHorde` - I just made that up. Below, set its class to `AppBundle\Entity\Genus`:

```
113 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 110
111         GenusHorde:
112             class: AppBundle\Entity\Genus
```

You see, some scientists are worried that certain genuses are becoming too large... and threaten the survival of mankind. They want a new `GenusHorde` section where they can keep track of all of the genuses that have a lot of species. It's scary stuff, so we'll add a label: `HORDE of Genuses` with a scary icon:

```
140 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 110
111         GenusHorde:
112             class: AppBundle\Entity\Genus
113             label: HORDE of Genuses 😱 !!!
... lines 114 - 140
```

Tip

Fun fact! You can press `Control` + `Command` + `Space` to open up the icon menu on a Mac.

And all of a sudden... ah! We have a new "Horde of Genuses" section! Run!!!

Of course, this still shows *all* genuses. I want to filter this to *only* list genuses that have a *lot* of species. Start by adding a `list` key and a new, awe-inspiring option: `dql_filter`. For the value, pretend that you're building a query in Doctrine. So, `entity.speciedCount >= 50000`:

```
142 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 110
111         GenusHorde:
... lines 112 - 113
114             list:
115                 dql_filter: 'entity.speciesCount >= 50000'
... lines 116 - 142
```

The alias will *always* be `entity`.

Try it! Ten down to... only 7 menacing genuses!

And just like any query, you can get more complex. How about: `AND entity.isPublished = true`:

```
144 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 110
111         GenusHorde:
... lines 112 - 113
114             list:
115                 dql_filter: 'entity.speciesCount >= 50000 AND entity.isPublished = true'
... lines 116 - 144
```

And to *really* focus on the genuses that are certain to overtake humanity, sort it by `speciesCount` and give the section a helpful message: `Run for your life!!!` Add scary icons for emphasis:

```
144 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 89
90      entities:
... lines 91 - 110
111         GenusHorde:
... lines 112 - 113
114             list:
115                 dql_filter: 'entity.speciesCount >= 50000 AND entity.isPublished = true'
116                 sort: 'speciesCount'
117                 help: Run for your life!!! 😱😱😱
... lines 118 - 144
```

Ok... refresh! Ah... now only *three* genuses are threatening mankind.

Oh, and search automatically re-uses the `dql_filter` from list: these are 2 results from the possible 3. And like always, you can override this. Under `search`, set the `dql_filter` to the same value, but without the `isPublished` check:

146 lines | app/config/config.yml

```yaml
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
... lines 91 - 110
        GenusHorde:
... lines 112 - 117
            search:
                dql_filter: 'entity.speciesCount >= 50000'
... lines 120 - 146
```

Try that. Boom! 3 more genuses that - when published - will spell certain doom for all.

Next! We'll save humanity by learning how to override the many templates that EasyAdminBundle uses.

Chapter 8: Customize all the Templates!

We can do *a lot* via config... but eventually... we're going to need to *really* dig in. And that will probably mean overriding the templates used by the bundle.

Exploring the Templates

First, let's go look at those templates! Open `vendor/javiereguiluz/easyadmin-bundle/Resources/views/default`. Ah, ha! These are the *many* templates used to render every single part of our admin. We can override *any* of these. But even better! We can override any of these for specific entities: using different customized templates for different sections. Or even... different templates to control how individual *fields* render.

Check out `layout.html.twig` ... this is the full admin page layout. It's awesome because it's *filled* with blocks. So instead of *completely* replacing the layout, you could extend this and override only the blocks you need. We won't do that for the layout, but we will for `list.html.twig`.

This is responsible for the `list` view we've been working on. And not surprisingly, there are also new, show and edit templates.

But *most* of the templates start with `field_` ... interesting. Remember how each field on the list page has a "data type"? We saw this in the EasyAdminBundle configuration. The "data type" is used to determine which template should render the data in that column. `firstDiscoveredAt` is a `date` type... and hey! It has a `template` option that defaults to `field_date.html.twig`. And by opening that template, you can see how the `date` type is rendered.

How to Override Templates

Ok, let's *finally* override some stuff! How!? On the same List, Search and Show Views Configuration page, near the bottom, you'll see an "Advanced Design Configuration" section. There are a *bunch* of different ways to override a template... ah... too many options! Let's simplify: (A) you can override a template via configuration - which are options 1 and 2 - or (B) by putting a file in an `easy_admin` directory - options 3 and 4. We'll try both.

Ok, first challenge! I want to override the way the `id` field is rendered for `Genus`: add a little `key` icon next to the number... ya know, because it's the primary key.

This means we need to override the `field_id.html.twig` template, because `id` is actually a data type. Copy `field_id.html.twig`. Then, in `app/Resources/views`, I already have an `admin` directory. So inside that, create a new `fields` directory and paste the file there, as `_id.html.twig`. Now, add the icon: `fa fa-key`:

2 lines | app/Resources/views/admin/fields/_id.html.twig

```
<i class="fa fa-key"></i> {{ value }}
```

Cool! I put the file here... just because I already have an `admin` directory. But EasyAdminBundle doesn't automagically know it's there. Nope, we need to tell it. In `config.yml`, to use this *only* for `Genus`, add a `templates` key, then `field_id` - the name of the original template - set to `admin/fields/_id.html.twig`:

115 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
... lines 82 - 89
    entities:
        Genus:
... lines 92 - 110
            templates:
                field_id: 'admin/fields/_id.html.twig'
... lines 113 - 115
```

Try that! Yes! It *is* using our template... and only in the `Genus` section. But this key thing is pretty excellent, so we should use it everywhere. Copy the templates config and comment it out:

```
117 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 91
92      entities:
93          Genus:
... lines 94 - 112
113 #           templates:
114 #               field_id: 'admin/fields/_id.html.twig'
... lines 115 - 117
```

Just like with almost anything, we can also control the template globally: paste this under `design`:

```
117 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... lines 84 - 86
87          templates:
88              field_id: 'admin/fields/_id.html.twig'
... lines 89 - 117
```

Now the key icon shows up *everywhere*.

Next, I want to override something bigger: the entire list template. And we'll use a different convention to do that.

Chapter 9: Dynamically Remove the delete Action Link

Another chapter, another problem to solve! I need to hide the delete button on the list page if an entity is published. So... nope! We can't just go into `config.yml` and add `-delete`. We need to override the `list.html.twig` template and take control of those actions manually.

Copy that file. Then, up inside our `views` directory, I want to show the *other* way of overriding templates: by convention. Create a new `easy_admin` directory, and paste the template there and... that's it! EasyAdminBundle will automatically know to use *our* list template.

Dumping Variables

The *toughest* thing about overriding a template is... well... figuring out what variables you can use! In `list.html.twig` ... how about in the `content_header` block, add `{{ dump() }}`:

```
245 lines | app/Resources/views/easy_admin/list.html.twig
... lines 1 - 38
39 {% block content_header %}
40     {{ dump() }}
... lines 41 - 93
94 {% endblock content_header %}
... lines 95 - 245
```

And in `_id.html.twig`, do the same:

```
2 lines | app/Resources/views/admin/fields/_id.html.twig
1 {{ dump() }}<i class="fa fa-key"></i> {{ value }}
```

I want to see what the variables look like in each template.

Ok, refresh the genus list page! Awesome! This first dump is from `list.html.twig`. It has the same `fields` configuration we've been looking at in the profiler, a `paginator` object and a few other things, including configuration for this specific section.

The other dumps come from `_id.html.twig`. The big difference is that we're rendering *one* `Genus` each time this template is called. So it has an `item` variable set to the `Genus` object. That will be super handy. If some of the other keys are tough to look at, remember, a lot of this already lives in the EasyAdminBundle profiler area.

Extending the Original Template

Ok, take out those dumps! So, how can we hide the delete button for published genuses? It's actually a bit tricky.

In `list.html.twig`, if you search, there is a variable called `_list_item_actions`:

```
244 lines | app/Resources/views/easy_admin/list.html.twig
... lines 1 - 94
95 {% block main %}
96     {% set _list_item_actions = easyadmin_get_actions_for_list_item(_entity_config.name) %}
... lines 97 - 189
190 {% endblock main %}
... lines 191 - 244
```

This contains information about the actions that should be rendered for each row. It's used further below, in a block called `item_actions`:

```
244 lines | app/Resources/views/easy_admin/list.html.twig
... lines 1 - 94
{% block main %}
    {% set _list_item_actions = easyadmin_get_actions_for_list_item(_entity_config.name) %}

    <div class="table-responsive">
    <table class="table">
... lines 100 - 129
        <tbody>
        {% block table_body %}
            {% for item in paginator.currentPageResults %}
... line 133
                <tr data-id="{{ _item_id }}">
... lines 135 - 143
                    {% if _list_item_actions|length > 0 %}
... line 145
                        <td data-label="{{ _column_label }}" class="actions">
                        {% block item_actions %}
                            {{ include('@EasyAdmin/default/includes/_actions.html.twig', {
... lines 149 - 153
                            }, with_context = false) }}
                        {% endblock item_actions %}
                        </td>
                    {% endif %}
                </tr>
... lines 159 - 164
            {% endfor %}
        {% endblock table_body %}
        </tbody>
    </table>
    </div>
... lines 170 - 189
{% endblock main %}
... lines 191 - 244
```

The template it renders - `_actions.html.twig` - generates a link at the end of the row for each action.

Let's dump `_list_item_actions` to see *exactly* what it looks like.

Ah, ok! It's an array with 3 keys: `edit`, `show` and `delete`. We need to remove that `delete` key, *only* if the entity is published. But how?

Here's my idea: if we override the `item_actions` block, we could *remove* the `delete` key from the `_list_item_actions` array and then call the parent `item_actions` block. It would use the new, smaller `_list_item_actions`.

Start by deleting *everything* and extending the base layout: `@EasyAdmin/default/list.html.twig` ... so that we don't need to duplicate everything:

```
8 lines | app/Resources/views/easy_admin/list.html.twig
1 {% extends '@EasyAdmin/default/list.html.twig' %}
... lines 2 - 8
```

Next, add `block item_actions` and `endblock`:

```
8 lines | app/Resources/views/easy_admin/list.html.twig
1 {% extends '@EasyAdmin/default/list.html.twig' %}
2
3 {% block item_actions %}
... lines 4 - 6
7 {% endblock %}
```

Twig isn't really meant for complex logic like removing keys from an array. But, to accomplish our goal, we don't have any other choice. So, set `_list_item_actions = _list_item_actions|filter_admin_actions(item)` :

8 lines | app/Resources/views/easy_admin/list.html.twig

```
1  {% extends '@EasyAdmin/default/list.html.twig' %}
2
3  {% block item_actions %}
4      {% set _list_item_actions = _list_item_actions|filter_admin_actions(item) %}
   ... lines 5 - 6
7  {% endblock %}
```

That filter does *not* exist yet: we're about to create it.

Just to review, open up the original `list.html.twig`. The `_list_item_actions` variable is set up here:

244 lines | app/Resources/views/easy_admin/list.html.twig

```
   ... lines 1 - 94
95  {% block main %}
96      {% set _list_item_actions = easyadmin_get_actions_for_list_item(_entity_config.name) %}
   ... lines 97 - 189
190 {% endblock main %}
   ... lines 191 - 244
```

Later, the `for` loop creates an `item` variable...

244 lines | app/Resources/views/easy_admin/list.html.twig

```
   ... lines 1 - 94
95  {% block main %}
96      {% set _list_item_actions = easyadmin_get_actions_for_list_item(_entity_config.name) %}
97
98      <div class="table-responsive">
99      <table class="table">
   ... lines 100 - 129
130         <tbody>
131         {% block table_body %}
132             {% for item in paginator.currentPageResults %}
   ... lines 133 - 164
165             {% endfor %}
166         {% endblock table_body %}
167         </tbody>
168     </table>
169     </div>
   ... lines 170 - 189
190 {% endblock main %}
   ... lines 191 - 244
```

which we have access to in the `item_actions` block:

244 lines | app/Resources/views/easy_admin/list.html.twig

```
... lines 1 - 94
{% block main %}
    {% set _list_item_actions = easyadmin_get_actions_for_list_item(_entity_config.name) %}

    <div class="table-responsive">
    <table class="table">
... lines 100 - 129
        <tbody>
        {% block table_body %}
            {% for item in paginator.currentPageResults %}
... line 133
                <tr data-id="{{ _item_id }}">
... lines 135 - 143
                    {% if _list_item_actions|length > 0 %}
... line 145
                        <td data-label="{{ _column_label }}" class="actions">
                        {% block item_actions %}
                            {{ include('@EasyAdmin/default/includes/_actions.html.twig', {
                                actions: _list_item_actions,
                                request_parameters: _request_parameters,
                                translation_domain: _entity_config.translation_domain,
                                trans_parameters: _trans_parameters,
                                item_id: _item_id
                            }, with_context = false) }}
                        {% endblock item_actions %}
                        </td>
                    {% endif %}
                </tr>
... lines 159 - 164
            {% endfor %}
        {% endblock table_body %}
        </tbody>
    </table>
    </div>
... lines 170 - 189
{% endblock main %}
... lines 191 - 244
```

Creating the Filter Twig Exension

Phew! All *we* need to do now is create that filter! In `src/AppBundle/Twig`, create a new PHP class: `EasyAdminExtension`. To make this a Twig extension, extend `\Twig_Extension`:

28 lines | src/AppBundle/Twig/EasyAdminExtension.php

```
... lines 1 - 2
namespace AppBundle\Twig;
... lines 4 - 6
class EasyAdminExtension extends \Twig_Extension
{
... lines 9 - 26
}
```

Then, go to the `Code` -> `Generate` menu - or `Command` + `N` on a Mac - and override the `getFilters()` method:

```
28 lines | src/AppBundle/Twig/EasyAdminExtension.php
... lines 1 - 2
namespace AppBundle\Twig;
... lines 4 - 6
class EasyAdminExtension extends \Twig_Extension
{
    public function getFilters()
    {
... lines 11 - 16
    }
... lines 18 - 26
}
```

Here, return an array with the filter we need: `new \Twig_SimpleFilter('filter_admin_actions', [$this, 'filterActions'])` :

```
28 lines | src/AppBundle/Twig/EasyAdminExtension.php
... lines 1 - 2
namespace AppBundle\Twig;
... lines 4 - 6
class EasyAdminExtension extends \Twig_Extension
{
    public function getFilters()
    {
        return [
            new \Twig_SimpleFilter(
                'filter_admin_actions',
                [$this, 'filterActions']
            )
        ];
    }
... lines 18 - 26
}
```

Down below, create `public function filterActions()` with two arguments. First, it will be passed an `$itemActions` array - that's the `_list_item_actions` variable. And second, `$item` : whatever entity is being listed at that moment:

```
28 lines | src/AppBundle/Twig/EasyAdminExtension.php
... lines 1 - 6
class EasyAdminExtension extends \Twig_Extension
{
... lines 9 - 18
    public function filterActions(array $itemActions, $item)
    {
... lines 21 - 25
    }
}
```

Ok, let's fill in the logic: `if $item instanceof Genus && $item->getIsPublished()` , then `unset($itemActions['delete'])` . At the bottom, `return $itemActions` :

28 lines | src/AppBundle/Twig/EasyAdminExtension.php

```php
... lines 1 - 4
use AppBundle\Entity\Genus;

class EasyAdminExtension extends \Twig_Extension
{
... lines 9 - 18
    public function filterActions(array $itemActions, $item)
    {
        if ($item instanceof Genus && $item->getIsPublished()) {
            unset($itemActions['delete']);
        }

        return $itemActions;
    }
}
```

Phew! That should do it! This project uses the new Symfony 3.3 autowiring, auto-registration and autoconfigure `services.yml` goodness:

32 lines | app/config/services.yml

```yaml
... lines 1 - 5
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true
        autoconfigure: true
... lines 11 - 12
    AppBundle\:
        resource: '../../src/AppBundle/*'
        exclude: '../../src/AppBundle/{Entity,Repository,Tests}'
... lines 16 - 32
```

So... we don't need to configure anything: `EasyAdminExtension` will automatically be registered as a service and tagged with `twig.extension`. In other words... it should just work.

Let's go. Refresh... and hold your breath.

Haha, it *kind* of worked! Delete *is* gone... but so is everything else. And you may have noticed why. We *did* change the `_list_item_actions` variable... but we forgot to call the parent block. Add `{{ parent() }}`:

8 lines | app/Resources/views/easy_admin/list.html.twig

```twig
... lines 1 - 2
{% block item_actions %}
    {% set _list_item_actions = _list_item_actions|filter_admin_actions(item) %}

    {{ parent() }}
{% endblock %}
```

Try it again. Got it! The delete icon is only there when the item is *not* published. This was a *tricky* example... which is why we did it! But usually, customizing things is easier. Technically, the user could still go directly to the URL to delete the `Genus`, but we'll see how to close that down later.

Chapter 10: Customize Template for One Field

We already customized the template used for *every* field whose data type is id. But you can also go deeper, and customize the way that just *one* specific field is rendered.

For example, let's say we need to customize how the "full name" field is rendered. No problem: in `config.yml`, find `User`, `list`, `fields`, and change `fullName` to the expanded configuration. To control the template add... surprise! A `template` option set to, how about, `_field_user_full_name.html.twig`:

```
117 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 91
92      entities:
... lines 93 - 117
```

Copy that name. It expects this to live in the standard `easy_admin` directory. Create it there!

Since this is a template for one field, it will have access to the `User` object as an `item` variable. And that makes life easy. Add `if item.isScientist` - that's a property on `User` - then add a cool graduation cap:

```
6 lines | app/Resources/views/easy_admin/_field_user_full_name.html.twig
1  {% if item.isScientist %}
2      <i class="fa fa-graduation-cap"></i>
3  {% endif %}
... lines 4 - 6
```

Below, print the full name. To do that, you can use a `value` variable. Pipe it through `|default('Stranger')`, just in case the user doesn't have any name data:

```
6 lines | app/Resources/views/easy_admin/_field_user_full_name.html.twig
1  {% if item.isScientist %}
2      <i class="fa fa-graduation-cap"></i>
3  {% endif %}
4
5  {{ value|default('Stranger') }}
```

Try it! Yes! We now know how to customize *entire* page templates - like `list.html.twig`, templates for a specific field *type*, or the template for just *one* field.

Time to move into forms!

Chapter 11: Form Field Customization

We can pretty much do anything to the list page. But we have *totally* ignored the two most important views: edit and new. Basically, we've ignored all the forms!

Ok, the `edit` and `new` views have *some* of the same configuration as `list` and `search` ... like `title` and `help`. They also both have a `fields` key... but it's quite different than `fields` under `edit` and `new`.

Start simple: for `Genus`, I want to control which fields are shown on the form: I don't want to show *every* field. But instead of adding an `edit` or `new` key, add `form`, `fields` below that, and the fields we want: `id`, `name`, `speciesCount`, `funFact`, `isPublished`, `firstDiscoveredAt`, `subFamily` and `genusScientists`:

133 lines | app/config/config.yml

```
... lines 1 - 80
81  easy_admin:
... lines 82 - 91
92      entities:
93          Genus:
... lines 94 - 114
115             form:
116                 fields:
117                     - id
118                     - name
119                     - speciesCount
120                     - funFact
121                     - isPublished
122                     - firstDiscoveredAt
123                     - subFamily
124                     - genusScientists
... lines 125 - 133
```

Before we talk about this, try it! Yes! Behind the scenes, EasyAdminBundle uses Symfony's form system... it just creates a normal form object by using our config. Right now, it's adding these 8 fields with no special config. Then, the form system is *guessing* which field *types* to use.

That's great... but why did we put `fields` under `form`, instead of `edit` or `new`? Where the heck did `form` come from? First, there is *not* a `form` view. But, since `edit` and `new` are *so* similar, EasyAdminBundle allows us to configure a "fake" view called `form`. Any config under `form` will automatically be used for `new` and `edit`. Then, you can keep customizing. Under `new` and `fields`, we can *remove* the id field with `-id`:

133 lines | app/config/config.yml

```
... lines 1 - 80
81  easy_admin:
... lines 82 - 91
92      entities:
93          Genus:
... lines 94 - 114
115             form:
... lines 116 - 124
125             new:
126                 fields:
127                     - '-id'
... lines 128 - 133
```

And under `edit`, to include the `slug` field - which is *not* under `form`, just add `slug`:

```yaml
133 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 91
    entities:
        Genus:
... lines 94 - 114
            form:
... lines 116 - 124
            new:
                fields:
                    - '-id'
            edit:
                fields:
                    - slug
... lines 131 - 133
```

Ok, refresh the edit page. Yep! *Now* we have a `slug` field... but it's all the way at the bottom. This is because the fields from `form` are added first, and *then* any `edit` fields are added. We'll fix the order later.

And the `new` view does *not* have `id`.

Customizing Field Types, Options

Go back into the EasyAdminBundle profiler. Under `new` and then `fields`, we can see each field *and* its `fieldType`. That corresponds to the Symfony *form* type that's being used for this field. Open up Symfony's form documentation and scroll down to the built-in fields list.

Yes, we know these: `TextType`, `TextareaType`, `EntityType`, etc. When you use these in a normal form class, you reference them by their full class name - like `EntityType::class`. EasyAdminBundle re-uses these form types... but lets us use a shorter string name... like just `entity`.

The most *important* way to customize a field is to change its type. For example, see `funFact`? It's just a text field... but sometimes, fun facts are *so* fun... a mere text box cannot contain them. No problem. Just like we did under `list`, we can *expand* each field: set `property: funFact`, then `type: textarea`:

```yaml
138 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 91
    entities:
        Genus:
... lines 94 - 114
            form:
                fields:
... lines 117 - 121
                    - { property: 'funFact', type: 'textarea' }
... lines 123 - 138
```

You can picture what this is doing internally: EasyAdminBundle now calls `$builder->add('funFact', TextareaType::class)`.

It even works! From working with forms, we also know that `$builder->add()` has a *third* argument: an options array. And yep, those are added here too. One normal form option is called `disabled`. Let's use that on the `id` field. Change it to use the expanded configuration - I'll even get fancy with multiple lines. Then, add `type_options` set to an array with `disabled: true`:

```
138 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 91
92      entities:
93          Genus:
... lines 94 - 114
115             form:
116                 fields:
117                     -
118                         property: id
119                         type_options: {disabled: true}
... lines 120 - 138
```

Do the same thing below on the `slug` field. Oh, and EasyAdminBundle also has one special config key called `help: Unique auto-generated value`:

```
138 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 91
92      entities:
93          Genus:
... lines 94 - 129
130             edit:
131                 fields:
132                     -
133                         property: 'slug'
134                         help: 'unique auto-generated value'
135                         type_options: { disabled: true }
... lines 136 - 138
```

Find your browser and go edit a genus. Yea... `id` is disabled... and so is `slug`. *And*, we have a super cool help message below!

The cool thing about EasyAdminBundle is that if you're comfortable with the form system... well... there's not much new to learn. You're simply customizing your form fields in YAML instead of in PHP.

For example, the `firstDiscoveredAt` field is a `DateType`. And that means, under `type_options`, we could set `widget` to `single_text` to render that in one text field:

```
138 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 91
92      entities:
93          Genus:
... lines 94 - 114
115             form:
116                 fields:
... lines 117 - 123
124                     - { property: 'firstDiscoveredAt', type_options: { widget: 'single_text' }}
... lines 125 - 138
```

If your browser supports it, you'll see a cute calendar widget.

Chapter 12: The Autocomplete Field

EasyAdminBundle re-uses all the form stuff... but also comes with *one new* form field... and it's pretty bananas! From the main documentation page, click Edit and New Views Configuration. Down a ways, find a section called "Autocomplete". Ah, lovely! This is a lot like the `EntityType` ... except that it renders as a fancy AJAX auto-complete box instead of a select drop down.

Right now, the `subFamily` field is a standard `EntityType` . But, it doesn't look that way at first... it's fancy! And has a search! We get this automatically thanks to some JavaScript added by EasyAdminBundle. It works *wonderfully*... as long as your drop down list is short. Because if there were hundreds or thousands of sub families... then *all* of them would need to be rendered on page load... which will *really* slow - or even break - your page.

Let's use the autocomplete field instead. Expand the `subFamily` configuration and set `type: easyadmin_autocomplete` :

138 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
... lines 82 - 91
    entities:
        Genus:
... lines 94 - 114
            form:
                fields:
... lines 117 - 124
                    - { property: 'subFamily', type: 'easyadmin_autocomplete' }
... lines 126 - 138
```

That is *all* we need: it will look at the `subFamily` field and know which entity to query. So.... it just works! Watch the web debug toolbar as a I type. Ha! There be AJAX happening!

Next, let's add a `CollectionType` to our form.

Chapter 13: CollectionType Field

The form system does a pretty good job guessing the correct field types... but nobody is perfect. For example, the `genusScientists` field is *not* setup correctly. Click the clipboard icon to open the form profiler.

Yep, `genusScientists` is currently an `EntityType` with `multiple` set to true. Thanks to EasyAdminBundle, it renders this as a cool, tag-like, auto-complete box. Fancy!

But... that's not going to work here: the `GenusScientist` entity has an extra field called `yearsStudied`:

85 lines | src/AppBundle/Entity/GenusScientist.php

```
... lines 1 - 17
18 class GenusScientist
19 {
... lines 20 - 38
39     /**
40      * @ORM\Column(type="integer")
41      * @Assert\NotBlank()
42      */
43     private $yearsStudied;
... lines 44 - 83
84 }
```

When you link a `Genus` and a `User`, we need to allow the admin to *also* fill in how many *years* the `User` has studied the `Genus`.

In the Symfony series, we did a *lot* of work to create a `CollectionType` field that used `GenusScientistEmbeddedForm`:

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

```
... lines 1 - 20
21 class GenusFormType extends AbstractType
22 {
23     public function buildForm(FormBuilderInterface $builder, array $options)
24     {
25         $builder
... lines 26 - 48
49             ->add('genusScientists', CollectionType::class, [
50                 'entry_type' => GenusScientistEmbeddedForm::class,
51                 'allow_delete' => true,
52                 'allow_add' => true,
53                 'by_reference' => false,
54             ])
55         ;
... lines 56 - 57
58     }
... lines 59 - 106
107 }
```

Thanks to that, in the admin, we just need to update the form to look like this.

Change `genusScientists` to use the expanded syntax. From here, you can guess what's next! Set `type: collection` and then add `type_options` with the 4 options you see here: `entry_type: AppBundle\Form\GenusScientistEmbeddedForm`, `allow_delete: true`, `allow_add: true`, `by_reference: false`:

```
145 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 91
    entities:
        Genus:
... lines 94 - 114
            form:
                fields:
... lines 117 - 125
                    -
                        property: 'genusScientists'
                        type: 'collection'
                        type_options:
                            entry_type: AppBundle\Form\GenusScientistEmbeddedForm
                            allow_delete: true
                            allow_add: true
                            by_reference: false
... lines 134 - 145
```

Let's see what happens! Woh! Ignore how *ugly* it is for a minute. It *does* work! We can remove items and add new ones.

But it looks weird. When we created this form for our custom admin area - we *hid* the user field when editing... which looks really odd now. Open the `GenusScientistEmbeddedForm`. We used a form event to accomplish this: if the `GenusScientist` had an id, we unset the `user` field. Comment that out for now and refresh:

```
52 lines | src/AppBundle/Form/GenusScientistEmbeddedForm.php
... lines 1 - 14
class GenusScientistEmbeddedForm extends AbstractType
{
... lines 17 - 34
    public function onPostSetData(FormEvent $event)
    {
        if ($event->getData() && $event->getData()->getId()) {
            $form = $event->getForm();
            // unset($form['user']);
        }
    }
... lines 42 - 50
}
```

Cool: this section *at least* makes more sense now.

The CollectionType Problems

But... there are *still* some problems! First, it's *ugly*! I know this is just an admin area... but wow! If you want to use the `CollectionType`, you'll probably need to create a custom form theme for this *one* field and render things in a more intelligent way. We'll do something similar in a few minutes.

Second... this only works because we already did a lot of hard work setting up the relationships to play well with the CollectionType. Honestly, the CollectionType is both the best and worst form type: you can do some really complex stuff... but it requires some seriously tough setup. You need to worry about the owning and the inverse sides of the relationship, and things called `orphanRemoval` and cascading. There is some significant Doctrine magic going on behind the scenes to get it working.

So in a few minutes, we're going to look at a more custom alternative to using the collection type.

Virtual Form Field

But first, I want to show you one more thing. Go to the `User` section and edit a User. We haven't touched *any* of this config yet. In `config.yml`, under `User`, add `form` then `fields`. Let's include `email` and `isScientist`:

```
145 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 91
    entities:
... lines 93 - 145
```

Right now, the form has `firstName` and `lastName` fields... which makes sense: there are `firstName` and `lastName` properties in `User`. But just like we did earlier under the `list` view, instead of having `firstName` and `lastName`, we could actually have, just `fullName`. And nope... there is *not* a `fullName` property. But as long as we create a `setFullName()` method, we can *totally* add it to the form:

```
281 lines | src/AppBundle/Entity/User.php
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 225
    public function setFullName($fullName)
    {
        $names = explode(' ', $fullName);
        $firstName = array_shift($names);
        $lastName = implode(' ', $names);

        $this->setFirstName($firstName);
        $this->setLastName($lastName);
    }
... lines 235 - 279
}
```

Actually, this isn't special to `EasyAdminBundle`, it's just how the form system works!

Now... this example is a little crazy. This code will take everything *before* the first space as the first name, and everything after as the last name. Totally imperfect, but you guys get the idea.

And now that we have `getFullName()` and `setFullName()`, add that as a field: `property: fullName`, `type: text` and a help message:

```
145 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 91
    entities:
... lines 93 - 145
```

Keep going to add `avatarUri` and `universityName`:

```
145 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 91
    entities:
... lines 93 - 145
```

Try it out! Yes! It looks great... *and*... it even submits! Next up, let's add a field that needs custom JavaScript to work.

Chapter 14: Custom Fields with JavaScript

99% of stuff in EasyAdminBundle is really easy. But it's that last, pesky 1% that can be so tough! And usually, that last 1% involves a form.

Sometimes, you have a form field that needs to be *really* customized. Maybe you need to write a bunch of special JavaScript, your own HTML or even create some custom routes and make AJAX calls back to them. Well... so far, all we can do is... just use the built-in form field types. And while that system is super extensible, it can also be super complex. So, we're going to dive into two examples where we do something very custom, without pulling our hair out.

Adding the JS Markdown Field

On the front-end of our site, this `funFact` field is processed through Markdown. That means we can *totally* use Markdown syntax inside its textarea. But... our admin users aren't very comfortable with Markdown... and being the awesome programmers that we are, we want to help them! I want to embed a JavaScript markdown previewer library called Snarkdown. You type some markdown, and Snarkdown shows you a preview.

So how can we transform our boring `textarea` field to include this?

In `config.yml`, under `Genus`, find the `funFact` field and add a `css_class` option set to `js-markdown-input`:

148 lines | app/config/config.yml

```
... lines 1 - 80
81  easy_admin:
... lines 82 - 94
95      entities:
96          Genus:
... lines 97 - 117
118             form:
119                 fields:
... lines 120 - 124
125                     - { property: 'funFact', type: 'textarea', css_class: 'js-markdown-input' }
... lines 126 - 148
```

This will be our new best friend. Because now that the element has this CSS class, we can write JavaScript to do *whatever* insane crazy things we want!

How do we include JavaScript on the page? We already know how! Up at the top, under `design` and `assets`, add `js`. Let's add 2 JavaScript files. First, include the Snarkdown library:

148 lines | app/config/config.yml

```
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... line 84
85          assets:
... line 86
87              js:
88                  - 'https://unpkg.com/snarkdown@1.2.2/dist/snarkdown.umd.js'
... lines 89 - 148
```

We could also download it locally. And include a new `js/custom_backend.js` file:

148 lines | app/config/config.yml

```yaml
... lines 1 - 80
easy_admin:
... line 82
    design:
... line 84
        assets:
... line 86
            js:
                - 'https://unpkg.com/snarkdown@1.2.2/dist/snarkdown.umd.js'
                - 'js/custom_backend.js'
... lines 90 - 148
```

To save some time, if you downloaded the code that came with this tutorial, you'll have a `tutorial/` directory with a `custom_backend.js` file inside. Copy that, go into `web/js` and paste:

14 lines | web/js/custom_backend.js

```js
$(document).ready(function () {
    var $markdownInputs = $('.js-markdown-input .form-control')

    $markdownInputs.after('<div class="markdown-preview"></div>');

    $markdownInputs.on('keyup', function (e) {
        var html = snarkdown(e.target.value);

        e.target.nextElementSibling.innerHTML = html;
    });

    $markdownInputs.trigger('keyup');
});
```

It's pretty simple: inside a `$(document).ready()` block, it finds all the `js-markdown-input` classes, gets their `.form-control` element, adds a new `markdown-preview` div and then processes that through Snarkdown. Basically, it creates an element that will show the preview version of our Markdown.

In that same `tutorial/` directory, there's also a `custom_backend.css` file. Just copy its contents. Open our `custom_backend.css` file and paste at the bottom... to make things *just* a little prettier:

14 lines | web/css/custom_backend.css

```css
... lines 1 - 4
.markdown-preview {
    margin-top: 10px;
    border: 2px dashed #da3735;
    padding: 5px;
}

.markdown-preview::before {
    content: "Preview: ";
}
```

I think we're ready! Refresh the page. Bam! There's our preview below the field. We can add some bold... and use some ticks. We rock!

I want you to realize how *powerful* this is. You can easily add a css class to any field. And then, by writing JavaScript, you can do *anything* you want! You could render a text field, hide it and then *entirely* replace the area with some crazy, custom JavaScript widget that updates the hidden text field in the background. This is your Swiss Army Knife.

In fact, we're going to do something similar next.

Chapter 15: Form Theming For a Completely Custom Field

Let's look at one more way to make a *ridiculously* custom field. Right now, we're using the `CollectionType` ... which works... but is totally ugly. And the *only* reason it works is that we did a lot of work in a previous tutorial to get the relationship setup properly.

And even if you *can* get the `CollectionType` working, you may want to add more bells and whistles to the interface. So here's the plan: we're not going to use the `CollectionType` ... at all. Instead, we'll write our own HTML and JavaScript to create our *own* widget, which will use AJAX to delete and add new entries. Actually, we won't do all of that right now - but I'll show you how to get things setup so you can get back to writing that custom code.

Configuring a Fake Field

Back in `config.yml`, find the `genusScientists` field, change its type to `text` and delete the 4 options:

148 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
... lines 82 - 94
    entities:
        Genus:
... lines 97 - 117
            form:
                fields:
... lines 120 - 128
                    -
                        property: 'genusScientists'
                        type: 'collection'
                        type_options:
                            entry_type: AppBundle\Form\GenusScientistEmbeddedForm
                            allow_delete: true
                            allow_add: true
                            by_reference: false
... lines 137 - 148
```

Whaaaat? Won't this break like crazy!? The `genusScientists` field holds a collection of `GenusScientist` objects... not just some text!

Totally! Except that we're going to add one magic config: `mapped: false`:

149 lines | app/config/config.yml

```
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
        Genus:
... lines 100 - 120
            form:
                fields:
... lines 123 - 131
                    -
... lines 133 - 134
                        type_options:
                            mapped: false
... lines 137 - 149
```

As *soon* as I do that, this is no longer a real field. I mean, when the form renders, it will *not* call `getGenusScientists()`. And when we submit, it will *not* call `setGenusScientists()`. In fact, you could even change the field name to something totally fake... and it

would work fine! This field *will* live in the form... but it doesn't read or set data on your entity. It's simply a way for us to "sneak" a fake field into our form.

Like we did in the last chapter, add a CSS class to the field: `js-scientists-field`:

```
149 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
        Genus:
... lines 100 - 120
            form:
                fields:
... lines 123 - 131
                    -
... lines 133 - 134
                        type_options:
                            mapped: false
                            attr: { class: 'js-genus-scientists-field' }
... lines 138 - 149
```

This time I'll use the standard `attr` option under `type_options`... but not for any particular reason.

Let's go see what this looks like! Yep, it's just a normal, empty text field: empty because it's not bound to our entity... at all - so it has no data.

Form Theme for One Field

Here's the goal: I want to replace this text field with our own Twig code, where we can build whatever crazy genus scientist-managing widget we want! How? The answer is to work with the form system: create a custom form theme that *just* overrides the widget for this *one* field.

To find out how, click the clipboard icon to get into the form profiler. Under `genusScientists`, open up the view variables. See this one called `block_prefixes`? This is the *key* for knowing the name of the block you should create in a form theme to customize this field. For example, to customize the `widget` for this field, we could create a block called `form_widget`, `text_widget` or `_genus_genusScientists_widget`. The last block would *only* affect this one field.

Copy that name. Then, in `app/Resources/views/easy_admin`, create a new file called `_form_theme.html.twig`. Add the block: `_genus_genusScientists_widget` with its `endblock`:

```
4 lines | app/Resources/views/easy_admin/_form_theme.html.twig
{% block _genus_genusScientists_widget %}
    Are you feeling powerful?
{% endblock %}
```

Are you feeling powerful yet? If not, you will soon. Before we start writing our awesome code, we need to tell Symfony to use this form theme. In previous tutorials, we learned how to add a custom form theme to our entire app... but in this case, we really only need this inside of our easy admin area.

EasyAdminBundle gives us a way to do this. In `config.yml`, under `design`, add `form_theme`. We're actually going to add two: `horizontal` and `easy_admin/_form_theme.html.twig`:

```
149 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 91
        form_theme:
            - horizontal
            - easy_admin/_form_theme.html.twig
... lines 95 - 149
```

EasyAdminBundle actually ships with two custom form themes: `horizontal` and `vertical` ... the difference is just whether the labels are next to, or above the fields. By default, `horizontal` is used. When you add your own custom form theme, you need to include `horizontal` or `vertical` ... to keep using it.

Ok... let's kick the tires! Close the profiler and refresh. Ahhhhhhh!

> Unrecognized option "form_themes" under "easy_admin.design"

Ok, my bad. It's `form_theme` :

```
149 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 91
        form_theme:
... lines 93 - 149
```

Thank you validation.

Now... we've got it! Our text shows up where the field should be. We can put anything here: like some HTML or an empty div that JavaScript fills in. Heck, we could create a React or Vue.js app and point it at this div. It's simple... but the possibilities are endless.

Rendering the Genus Scientists

Let's see a quick example to get the creative juices flowing! Let's create a table that lists all of the genus scientists:

```
16 lines | app/Resources/views/easy_admin/_form_theme.html.twig
{% block _genus_genusScientists_widget %}
    <table class="table">
        <tbody>
... lines 4 - 12
        </tbody>
    </table>
{% endblock %}
```

Inside a `tbody` , we're ready to loop over the scientists! But... uh... how can I get them? What variables do I have access to right here?

Go back to the form profiler, find `genusScientists` and look again at the view variables. These are all the variables that we have access to from within our form theme. But because we set the field to `mapped` false... um... we actually don't have access to our `Genus` object! That's a problem. But! Because we're inside EasyAdminBundle, it gives us a special `easyadmin` variable... with an `item` key equal to our `Genus` ! Phew!

Ok! In the table, loop: `for genusScientist in easyadmin.item.genusScientists` :

```
16 lines | app/Resources/views/easy_admin/_form_theme.html.twig
1  {% block _genus_genusScientists_widget %}
2      <table class="table">
3          <tbody>
4              {% for genusScientist in easyadmin.item.genusScientists %}
... lines 5 - 11
12             {% endfor %}
13         </tbody>
14     </table>
15 {% endblock %}
```

Add the `tr` and print out a few fields: `genusScientist.user` and `genusScientist.yearsStudied`:

```
16 lines | app/Resources/views/easy_admin/_form_theme.html.twig
1  {% block _genus_genusScientists_widget %}
2      <table class="table">
3          <tbody>
4              {% for genusScientist in easyadmin.item.genusScientists %}
5                  <tr>
6                      <td>{{ genusScientist.user }}</td>
7                      <td>{{ genusScientist.yearsStudied }} years</td>
... lines 8 - 10
11                 </tr>
12             {% endfor %}
13         </tbody>
14     </table>
15 {% endblock %}
```

Let's also add a fake delete link with a class and a `data-url` attribute. But leave it blank:

```
16 lines | app/Resources/views/easy_admin/_form_theme.html.twig
1  {% block _genus_genusScientists_widget %}
2      <table class="table">
3          <tbody>
4              {% for genusScientist in easyadmin.item.genusScientists %}
5                  <tr>
6                      <td>{{ genusScientist.user }}</td>
7                      <td>{{ genusScientist.yearsStudied }} years</td>
8                      <td>
9                          <a href="#" class="js-delete-scientist" data-url="">&times;</a>
10                     </td>
11                 </tr>
12             {% endfor %}
13         </tbody>
14     </table>
15 {% endblock %}
```

In your app, you might create a delete AJAX endpoint and use the `path()` function to put that URL here so you can read it in JavaScript.

Cool! To make this a *bit* more realistic, open `custom_backend.js`. Let's find those `.js-delete-scientist` elements and, on click, call a function. Add the normal `e.preventDefault()` and... an `alert('to do')`:

20 lines | web/js/custom_backend.js

```
1  $(document).ready(function () {
   ... lines 2 - 13
14     $('.js-delete-scientist').on('click', function(e) {
15         e.preventDefault();
16
17         alert('todo');
18     });
19 });
```

The rest, is homework!

Let's try it! There it us! A nice table with a delete icon. There's more work to do, but you can totally do it! This is just normal coding: create a delete endpoint, call it via JavaScript and celebrate!

With form stuff behind us, let's turn to adding custom *actions*, like, a publish button.

Chapter 16: Adding a Custom Action

We know there are a bunch of built-in *actions*, like "delete" and "edit. But sometimes you need to manipulate an entity in a different way! Like, how could we add a "publish" button next to each Genus?

There are... two different ways to do that. Click into the show view for a Genus. On show, the actions show up at the bottom. Before we talk about publishing, I want to add a new button down here called "feed"... ya know... because Genuses get hungry. When we click that, it should send the user to a custom controller where we can write whatever crazy code we want.

Custom Route Actions

The first step should feel very natural. We already know how to add actions, remove actions and customize how they look. Under `Genus`, add a new `show` key and `actions`. Use the expanded configuration, with `name: genus_feed` and `type: route`:

```
157 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
        Genus:
... lines 100 - 118
            show:
                actions:
                    -
                        name: 'genus_feed'
                        type: 'route'
... lines 124 - 157
```

There are two different custom action "types": `route` and `action`. Route is simple: it creates a new link to the `genus_feed` route. And you can use any of the normal action-configuring options, like `label`, `css_class: 'btn btn-info` or an `icon`:

```
157 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
        Genus:
... lines 100 - 118
            show:
                actions:
                    -
                        name: 'genus_feed'
                        type: 'route'
                        label: 'Feed genus'
                        css_class: 'btn btn-info'
                        icon: 'cutlery'
... lines 127 - 157
```

Adding the Route Action Endpoint

Next, we need to actually *create* that route and controller. In `src/AppBundle/Controller`, open `GenusController`. At the top, add `feedAction()` with `@Route("/genus/feed")` and `name="genus_feed` to match what we put in the config:

```
169 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 15
16 class GenusController extends Controller
17 {
18     /**
19      * @Route("/genus/feed", name="genus_feed")
20      */
21     public function feedAction(Request $request)
22     {
... lines 23 - 36
37     }
... lines 38 - 167
168 }
```

Notice the URL for this is just `/genus/feed`. It does not start with `/easyadmin`. And so, it's not protected by our `access_control` security.

That should be enough to get started. Refresh! There's our link! Click it and... good! Error! I love errors! Our action is still empty.

So here's the question: when we click feed on the `Genus` show page... the EasyAdminBundle must *somehow* pass us the id of that genus... right? Yes! It does it via query parameters... which are a bit ugly! So I'll open up my profiler and go to "Request / Response". Here are the GET parameters. We have `entity` and `id`!

Now that we know that, this will be a pretty traditional controller. I'll type-hint the `Request` object as an argument:

```
169 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 12
13 use Symfony\Component\HttpFoundation\Request;
... lines 14 - 15
16 class GenusController extends Controller
17 {
... lines 18 - 20
21     public function feedAction(Request $request)
22     {
... lines 23 - 36
37     }
... lines 38 - 167
168 }
```

Then, fetch the entity manager and the `$id` via `$request->query->get('id')`. Use that to get the `$genus` object: `$em->getRepository(Genus::class)->find($id)`.

```
169 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 15
16 class GenusController extends Controller
17 {
... lines 18 - 20
21     public function feedAction(Request $request)
22     {
23         $em = $this->getDoctrine()->getManager();
24         $id = $request->query->get('id');
25         $genus = $em->getRepository('AppBundle:Genus')->find($id);
... lines 26 - 36
37     }
... lines 38 - 167
168 }
```

Cool! To feed the `Genus`, we'll re-use a `feed()` method from a previous tutorial. Start by creating a menu of delicious food: `shrimp`, `clams`, `lobsters` and... `dolphin`! Then choose a random food, add a flash message and call `$genus->feed()`:

```
169 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 15
16 class GenusController extends Controller
17 {
... lines 18 - 20
21     public function feedAction(Request $request)
22     {
23         $em = $this->getDoctrine()->getManager();
24         $id = $request->query->get('id');
25         $genus = $em->getRepository('AppBundle:Genus')->find($id);
26
27         $menu = ['shrimp', 'clams', 'lobsters', 'dolphin'];
28         $meal = $menu[random_int(0, 3)];
29
30         $this->addFlash('info', $genus->feed([$meal]));
... lines 31 - 36
37     }
... lines 38 - 167
168 }
```

Now that all this hard work is done, I want to redirect back to the show view for this genus. Like normal, `return $this->redirectToRoute()`. And actually, EasyAdminBundle only has *one* route... called `easyadmin`:

```
169 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 15
16 class GenusController extends Controller
17 {
... lines 18 - 20
21     public function feedAction(Request $request)
22     {
23         $em = $this->getDoctrine()->getManager();
24         $id = $request->query->get('id');
25         $genus = $em->getRepository('AppBundle:Genus')->find($id);
26
27         $menu = ['shrimp', 'clams', 'lobsters', 'dolphin'];
28         $meal = $menu[random_int(0, 3)];
29
30         $this->addFlash('info', $genus->feed([$meal]));
31
32         return $this->redirectToRoute('easyadmin', [
... lines 33 - 35
36         ]);
37     }
... lines 38 - 167
168 }
```

We tell it where to go via query parameters, like `action` set to `show`, `entity` set to `$request->query->get('entity')` ... or we could just say `Genus`, and `id` set to `$id`:

```
169 lines | src/AppBundle/Controller/GenusController.php
... lines 1 - 15
16 class GenusController extends Controller
17 {
... lines 18 - 20
21     public function feedAction(Request $request)
22     {
... lines 23 - 31
32         return $this->redirectToRoute('easyadmin', [
33             'action' => 'show',
34             'entity' => $request->query->get('entity'),
35             'id' => $id
36         ]);
37     }
... lines 38 - 167
168 }
```

That is it! Refresh the show page! And feed the genus. Got it! We can hit that over and over again. Hello custom action.

Custom Controller Action

There's also *another* way of creating a custom action. It's a bit simpler and a bit stranger... but has one advantage: it allows you to create different implementations of the action for different entities.

Let's try it! In `config.yml`, add another action. This time, set the name to `changePublishedStatus` with a `css_class` set to `btn`:

```
158 lines | app/config/config.yml
... lines 1 - 80
81 easy_admin:
... lines 82 - 97
98     entities:
99         Genus:
... lines 100 - 118
119            show:
120                actions:
... lines 121 - 126
127                    - { name: 'changePublishedStatus', css_class: 'btn' }
... lines 128 - 158
```

Let's do as *little* work as possible! So...refresh! We have a button! Click it! Bah! Big error! But, it explains how the feature works:

> Warning: `call_user_func_array()` expects parameter 1 to be a valid callback, class `AdminController` does not have a method `changePublishedStatusAction()`.

Eureka! All we need to do is create that method... then celebrate!

Overriding the AdminController

To do that, we need to sub-class the core `AdminController`. Create a new directory in `Controller` called `EasyAdmin`. Then inside, a new PHP class called `AdminController`. To make this extend the normal `AdminController`, add a `use` statement for it: use `AdminController as BaseAdminController`. Extend that: `BaseAdminController`:

```
27 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php
... lines 1 - 2
namespace AppBundle\Controller\EasyAdmin;

use JavierEguiluz\Bundle\EasyAdminBundle\Controller\AdminController as BaseAdminController;

class AdminController extends BaseAdminController
{
... lines 9 - 25
}
```

Next, create that action method: `changePublishedStatusAction()`:

```
27 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php
... lines 1 - 6
class AdminController extends BaseAdminController
{
    public function changePublishedStatusAction()
    {
... lines 11 - 24
    }
}
```

Notice the config key is just `changePublishedStatus` - EasyAdminBundle automatically expects that `Action` suffix.

And now that we're in a controller method... we're comfortable! I mean, we could write a killer action in our sleep. But... there's a gotcha. This method is not, exactly, like a traditional controller. That's because it's not called by Symfony's routing system... it's called directly by EasyAdminBundle, which is trying to "fake" things.

In practice, this means one important thing: we *cannot* add a `Request` argument. Actually, *all* of the normal controller argument tricks will not work.. because this isn't *really* a real controller.

Fetching the Request & the Entity

Instead, the base `AdminController` has a few surprises for us: protected properties with handy things like the entity manager, the request and some EasyAdmin configuration.

Let's use this! Get the id query parameter via `$this->request->query->get('id')`. Then, fetch the object with `$entity = $this->em->getRepository(Genus::class)->find($id)`:

```
27 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php
... lines 1 - 6
class AdminController extends BaseAdminController
{
    public function changePublishedStatusAction()
    {
        $id = $this->request->query->get('id');
        $entity = $this->em->getRepository('AppBundle:Genus')->find($id);
... lines 13 - 24
    }
}
```

Now things are easier. Change the published status to whatever it is *not* currently. Then, `$this->em->flush()`:

```
27 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php
... lines 1 - 6
class AdminController extends BaseAdminController
{
    public function changePublishedStatusAction()
    {
        $id = $this->request->query->get('id');
        $entity = $this->em->getRepository('AppBundle:Genus')->find($id);

        $entity->setIsPublished(!$entity->getIsPublished());

        $this->em->flush();
... lines 17 - 24
    }
}
```

Set a fancy flash message that says whether the genus was just published or unpublished:

```
27 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php
... lines 1 - 6
class AdminController extends BaseAdminController
{
    public function changePublishedStatusAction()
    {
        $id = $this->request->query->get('id');
        $entity = $this->em->getRepository('AppBundle:Genus')->find($id);

        $entity->setIsPublished(!$entity->getIsPublished());

        $this->em->flush();

        $this->addFlash('success', sprintf('Genus %spublished!', $entity->getIsPublished() ? '' : 'un'));
... lines 19 - 24
    }
}
```

And finally, at the bottom, I want to redirect back to the show page. Let's go steal that code from `GenusController`. The one difference of course is that `$request` needs to be `$this->request`:

27 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php

```
... lines 1 - 6
class AdminController extends BaseAdminController
{
    public function changePublishedStatusAction()
    {
        $id = $this->request->query->get('id');
        $entity = $this->em->getRepository('AppBundle:Genus')->find($id);

        $entity->setIsPublished(!$entity->getIsPublished());

        $this->em->flush();

        $this->addFlash('success', sprintf('Genus %spublished!', $entity->getIsPublished() ? '' : 'un'));

        return $this->redirectToRoute('easyadmin', [
            'action' => 'show',
            'entity' => $this->request->query->get('entity'),
            'id' => $id,
        ]);
    }
}
```

Pointing to our AdminController Classs

Ok friends. Refresh! It works! Ahem... I mean, we totally get the exact same error! What!?

This is because we haven't told Symfony to use *our* AdminController yet: it's still using the one from the bundle. The fix is actually in `routing.yml`:

14 lines | app/config/routing.yml

```
... lines 1 - 9
easy_admin_bundle:
    resource: "@EasyAdminBundle/Controller/"
    type: annotation
    prefix: /easyadmin
```

This tells Symfony to import the annotation routes from the bundle's `AdminController` class... which means *that* class is used when we go to those routes. Change this to import routes from `@AppBundle/Controller/EasyAdmin/AdminController.php` instead:

14 lines | app/config/routing.yml

```
... lines 1 - 9
easy_admin_bundle:
    resource: "@AppBundle/Controller/EasyAdmin/AdminController.php"
... lines 12 - 14
```

It will *still* read the same route annotations from the base class, because we're extending it. But now, it will use *our* class when that route is matched.

That should be all we need. Try it. Boom! Genus published. Do it again! Genus unpublished! The power... it's intoxicating...

Next! We're going to go rogue... and start adding our own custom hooks... like right before or after an entity is inserted or updated.

Chapter 17: Override Controllers

When we submit a form, obviously, EasyAdminBundle is taking care of *everything*: handling validation, saving and adding a flash message. That's the best thing ever! Until... I need to hook into that process... then suddenly, it's the *worst* thing ever! What if I need to do some custom processing right before an entity is created or updated?

There are 2 main ways to hook into EasyAdminBundle...and I want you to know both. Open the `User` entity. It has an `updatedAt` field:

281 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
17 class User implements UserInterface
18 {
... lines 19 - 82
83     /**
84      * @ORM\Column(name="updated_at", type="datetime", nullable=true)
85      */
86     private $updatedAt;
... lines 87 - 279
280 }
```

To set this, we could use Doctrine lifecycle callbacks or a Doctrine event subscriber.

But, I want to see if we can set this instead, by hooking into EasyAdminBundle. In other words, when the user submits the form for an update, we need to run some code.

The protected AdminController Methods

The first way to do this is by adding some code to the controller. Check this out: open up the base `AdminController` from the bundle and search for `protected function`. Woh... There are a *ton* of methods that we can override, beyond just the actions. Like, `createNewEntity()`, `prePersistEntity()` and `preUpdateEntity()`.

If we override `preUpdateEntity()` in *our* controller, that will be called right before *any* entity is updated. There are a few other cool things that you can override too.

Per-Entity Override Methods

Ok, easy! Just add `preUpdateEntity()` to our `AdminController`, right? Yep... but we can do better! If we override `preUpdateEntity()`, it will be called whenever *any* entity is updated. But we really *only* want it to be called for the `User` entity.

Once again, EasyAdminBundle has our back. Inside the base controller, search for `preUpdate`. Check this out: right before saving, it calls some `executeDynamicMethod` function and passes it `preUpdate`, a weird `<EntityName>` string, then `Entity`.

Actually, the bundle does this type of thing all over the place. Like above, when it calls `createEditForm()`. Whenever you see this, it means that bundle will *first* look for an entity-specific version of the method - like `preUpdateUserEntity()` - and call it. If that doesn't exist, it will call the normal `preUpdateEntity()`.

This is *huge*: it means that each entity class can have its *own* set of hook methods in our `AdminController`!

One Controller per Entity

And now that I've told you that... we're going to do something completely different. Instead of having one controller - `AdminController` - full of entity-specific hook methods like `preUpdateUserEntity` or `createGenusEditForm` - I prefer to create a custom controller class for each entity.

Try this: in the `EasyAdmin` directory, copy `AdminController` and rename it to `UserController`. Then, remove the function. Use the "Code"->"Generate menu" - or `Command` + `N` on a Mac - to override the `preUpdateEntity()` method. And don't forget to update your class name to `UserController`:

```
18 lines | src/AppBundle/Controller/EasyAdmin/UserController.php
... lines 1 - 2
namespace AppBundle\Controller\EasyAdmin;

use AppBundle\Entity\User;
use JavierEguiluz\Bundle\EasyAdminBundle\Controller\AdminController as BaseAdminController;

class UserController extends BaseAdminController
{
    /**
     * @param User $entity
     */
    protected function preUpdateEntity($entity)
    {
... line 15
    }
}
```

We're going to configure things so that this `UserController` is used *only* for the `User` admin section. And that means we can safely assume that the `$entity` argument will *always* be a `User` object:

```
18 lines | src/AppBundle/Controller/EasyAdmin/UserController.php
... lines 1 - 4
use AppBundle\Entity\User;
... lines 6 - 7
class UserController extends BaseAdminController
{
    /**
     * @param User $entity
     */
    protected function preUpdateEntity($entity)
    {
... line 15
    }
}
```

And that makes life easy: `$entity->setUpdatedAt(new \DateTime())`:

```
18 lines | src/AppBundle/Controller/EasyAdmin/UserController.php
... lines 1 - 7
class UserController extends BaseAdminController
{
... lines 10 - 12
    protected function preUpdateEntity($entity)
    {
        $entity->setUpdatedAt(new \DateTime());
    }
}
```

But how does `EasyAdminBundle` know to use this controller *only* for the `User` entity? That happens in `config.yml`. Down at the bottom, under `User`, add `controller: AppBundle\Controller\EasyAdmin\UserController`:

```
158 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
... lines 99 - 158
```

And *just* like that! We have one controller that's used for *just* our `User`.

Try it out! Let's go find a user... how about ID 20. Right now, its `updateAt` is null. Edit it... make some changes... and save! Go back to show and... we got it!

Organizing into a Base AdminContorller

This little trick unlocks a lot of hook points. But if you look at `AdminController`, it's a little messy. Because, `changePublishedStatusAction()` is *only* meant to be used for the `Genus` class:

27 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php

```
... lines 1 - 6
7  class AdminController extends BaseAdminController
8  {
9      public function changePublishedStatusAction()
10     {
11         $id = $this->request->query->get('id');
12         $entity = $this->em->getRepository('AppBundle:Genus')->find($id);
... lines 13 - 24
25     }
26 }
```

But *technically*, this controller is being used by *all* entities, except `User`.

So let's copy `AdminController` and make a new `GenusController`! Empty `AdminController` completely:

10 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php

```
... lines 1 - 2
3  namespace AppBundle\Controller\EasyAdmin;
4
5  use JavierEguiluz\Bundle\EasyAdminBundle\Controller\AdminController as BaseAdminController;
6
7  class AdminController extends BaseAdminController
8  {
9  }
```

Then, make sure you rename the new controller class to `GenusController`:

25 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php

```
... lines 1 - 2
3  namespace AppBundle\Controller\EasyAdmin;
4
5  class GenusController extends AdminController
6  {
7      public function changePublishedStatusAction()
8      {
9          $id = $this->request->query->get('id');
10         $entity = $this->em->getRepository('AppBundle:Genus')->find($id);
11
12         $entity->setIsPublished(!$entity->getIsPublished());
13
14         $this->em->flush();
15
16         $this->addFlash('success', sprintf('Genus %spublished!', $entity->getIsPublished() ? '' : 'un'));
17
18         return $this->redirectToRoute('easyadmin', [
19             'action' => 'show',
20             'entity' => $this->request->query->get('entity'),
21             'id' => $id,
22         ]);
23     }
24 }
```

But before we set this up in config, change the extends to `extends AdminController`, and remove the now-unused `use` statement:

```
25 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php
... lines 1 - 2
3  namespace AppBundle\Controller\EasyAdmin;
4
5  class GenusController extends AdminController
6  {
... lines 7 - 23
24 }
```

Repeat that in `UserController`:

```
17 lines | src/AppBundle/Controller/EasyAdmin/UserController.php
... lines 1 - 2
3  namespace AppBundle\Controller\EasyAdmin;
... lines 4 - 6
7  class UserController extends AdminController
8  {
... lines 9 - 15
16 }
```

Yep, now *all* of our sections share a common base `AdminController` class. And even though it's empty now, this could be *really* handy later if we ever need to add a hook that affects *everything*.

Love it! `UserController` has only the stuff it needs, `GenusController` holds only things that relate to `Genus`, and if we need to override something for all entities, we can do that inside `AdminController`.

Don't forget to go back to your config to tell the bundle about the `GenusController`. All the way on top, set the `Genus` controller to `AppBundle\Controller\EasyAdmin\GenusController`:

```
159 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
99          Genus:
... line 100
101             controller: AppBundle\Controller\EasyAdmin\GenusController
... lines 102 - 159
```

Now we're setup to do some really, really cool stuff.

Chapter 18: Event Hooks

There's one other *major* way to hook into things with EasyAdminBundle... and it's my favorite! Go back to the base `AdminController` and search for "event". You'll see a *lot* in here! Whenever EasyAdminBundle does, well, pretty much anything... it dispatches an event: `PRE_UPDATE`, `POST_UPDATE`, `POST_EDIT`, `PRE_SHOW`, `POST_SHOW`... yes we get the idea already!

And this means that we can use standard Symfony event subscribers to totally kick EasyAdminBundle's butt!

Creating an Event Subscriber

Create a new `Event` directory... though, this could live anywhere. Then, how about, `EasyAdminSubscriber`. Event subscribers always implement `EventSubscriberInterface`:

23 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 2
namespace AppBundle\Event;
... lines 4 - 5
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
... lines 7 - 8
class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 11 - 21
}
```

I'll go to the "Code"->"Generate" menu - or `Command` + `N` on a Mac - and choose "Implement Methods" to add the one required method: `getSubscribedEvents()`:

23 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 8
class EasyAdminSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
... lines 13 - 15
    }
... lines 17 - 21
}
```

EasyAdminBundle dispatches a *lot* of events... but fortunately, they all live as constants on a helper class called `EasyAdminEvents`. We want to use `PRE_UPDATE`. Set that to execute a new method `onPreUpdate` that we will create in a minute:

23 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 4
use JavierEguiluz\Bundle\EasyAdminBundle\Event\EasyAdminEvents;
... lines 6 - 8
class EasyAdminSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return [
            EasyAdminEvents::PRE_UPDATE => 'onPreUpdate',
        ];
    }
... lines 17 - 21
}
```

But first, I'll hold `Command` and click into that class. Dude, this is cool: this puts *all* of the possible hook points right in front of us. There are a few different categories: most events are either for customizing the actions and views or for hooking into the entity saving process.

That difference is important, because our subscriber method will be passed *slightly* different information based on which event it's listening to.

Back in our subscriber, we need to create `onPreUpdate()`. That's easy, but it's Friday and I'm *so* lazy. So I'll hit the `Alt` + `Enter` shortcut and choose "Create Method":

23 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 6
use Symfony\Component\EventDispatcher\GenericEvent;

class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 11 - 17
    public function onPreUpdate(GenericEvent $event)
    {
... line 20
    }
}
```

Thank you PhpStorm Symfony plugin!

Notice that it added a `GenericEvent` argument. In EasyAdminBundle, *every* event passes you this same object... just with different data. So, you kind of need to dump it to see what you have access to:

23 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 8
class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 11 - 17
    public function onPreUpdate(GenericEvent $event)
    {
        dump($event);die;
    }
}
```

Since we're using Symfony 3.3 and the new service configuration, my event subscriber will automatically be loaded as a service and tagged as an event subscriber:

32 lines | app/config/services.yml

```
... lines 1 - 5
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true
        autoconfigure: true
        public: false

    AppBundle\:
        resource: '../../src/AppBundle/*'
        exclude: '../../src/AppBundle/{Entity,Repository,Tests}'

    AppBundle\Controller\:
        resource: '../../src/AppBundle/Controller'
        public: true
        tags: ['controller.service_arguments']
... lines 21 - 32
```

If that just blew your mind, check out our Symfony 3.3 series!

This means we can just... try it! Edit a user and submit. Bam!

Fetching Info off the Event

For this event, the important thing is that we have a `subject` property on `GenericEvent` ... which holds the `User` object. We can get this via `$event->getSubject()` :

41 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 10
11 class EasyAdminSubscriber implements EventSubscriberInterface
12 {
... lines 13 - 26
27     public function onPreUpdate(GenericEvent $event)
28     {
29         $entity = $event->getSubject();
... lines 30 - 38
39     }
40 }
```

Remember though, this `PRE_UPDATE` event will be fired for *every* entity - not just `User` . So, we need to check for that: if `$entity instanceof User` , then we know it's safe to work our magic:

41 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 10
11 class EasyAdminSubscriber implements EventSubscriberInterface
12 {
... lines 13 - 26
27     public function onPreUpdate(GenericEvent $event)
28     {
29         $entity = $event->getSubject();
30
31         if ($entity instanceof User) {
... lines 32 - 37
38         }
39     }
40 }
```

Since we already took care of setting the `updatedAt` in the controller, let's do something different. The `User` class *also* has a `lastUpdatedBy` field, which should be a `User` object:

281 lines | src/AppBundle/Entity/User.php

```
... lines 1 - 16
17 class User implements UserInterface
18 {
... lines 19 - 87
88     /**
89      * @ORM\OneToOne(targetEntity="User")
90      * @ORM\JoinColumn(name="last_updated_by_id", referencedColumnName="id", nullable=true)
91      */
92     private $lastUpdatedBy;
... lines 93 - 279
280 }
```

Let's set that here.

That means we need to get the currently-logged-in `User` object. To get that from inside a service, we need to use another service. At the top, add a constructor. Then, type-hint the first argument with `TokenStorageInterface` . Watch out: there are two of them... and oof, it's impossible to know which is which. Choose either of them for now. Then, name the argument and hit `Alt` + `Enter` to create and set a new property:

```
41 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 8
use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface;

class EasyAdminSubscriber implements EventSubscriberInterface
{
    private $tokenStorage;

    public function __construct(TokenStorageInterface $tokenStorage)
    {
        $this->tokenStorage = $tokenStorage;
    }
... lines 19 - 39
}
```

Back on top... this is not the right `use` statement. I'll re-add `TokenStorageInterface`: make sure you choose the one from `Security\Core\Authentication`:

```
41 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 8
use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface;

class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 13 - 39
}
```

In our method, fetch the user with `$user = $this->tokenStorage->getToken()->getUser()`. And if the `User` is *not* an `instanceof` our `User` class, that means the user isn't actually logged in. In that case, set `$user = null`:

```
41 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 10
class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 13 - 26
    public function onPreUpdate(GenericEvent $event)
    {
        $entity = $event->getSubject();

        if ($entity instanceof User) {
            $user = $this->tokenStorage->getToken()->getUser();
            if (!$user instanceof User) {
                $user = null;
            }
... lines 36 - 37
        }
    }
}
```

Then, `$entity->setLastUpdatedBy($user)`:

41 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 10
class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 13 - 26
    public function onPreUpdate(GenericEvent $event)
    {
        $entity = $event->getSubject();

        if ($entity instanceof User) {
            $user = $this->tokenStorage->getToken()->getUser();
            if (!$user instanceof User) {
                $user = null;
            }

            $entity->setLastUpdatedBy($user);
        }
    }
}
```

Woohoo! Thanks to the new auto-wiring stuff in Symfony 3.3, we don't need to configure *anything* in `services.yml`. Yep, with some help from the type-hint, Symfony already knows what to pass to our `$tokenStorage` argument.

So go back, refresh and... no errors! It's always creepy when things work on the first try. Go to the show page for the User id 20. Last updated by is set!

Next, we're going to hook into the bundle further and learn how to completely disable actions based on security permissions.

Chapter 19: Conditional Actions

Ok, new challenge! I *only* want this edit button to be visible and accessible if the user has `ROLE_SUPERADMIN`. This turns out to be a bit complicated... in part because there are two sides to it.

First, we need truly block *access* to that action... so that a clever user can't just hack the URL and start editing! And second, we need to actually hide the link... so that our less-than-super-admin users don't get confused.

Preventing Action Access by Role

First, let's lock down the actual controller action. How? Now we know two ways: by overriding the `editAction()` in `UserController` and adding a security check *or* by adding a `PRE_EDIT` event listener. Let's use events!

Subscribe to a *second* event: `EasyAdminEvents::PRE_EDIT` set to `onPreEdit`:

61 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 5
use JavierEguiluz\Bundle\EasyAdminBundle\Event\EasyAdminEvents;
... lines 7 - 12
class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 15 - 23
    public static function getSubscribedEvents()
    {
        return [
            EasyAdminEvents::PRE_EDIT => 'onPreEdit',
... line 28
        ];
    }
... lines 31 - 59
}
```

Once again, I'll hit `Alt` + `Enter` as a shortcut to create that method for me:

61 lines | src/AppBundle/Event/EasyAdminSubscriber.php

```
... lines 1 - 7
use Symfony\Component\EventDispatcher\GenericEvent;
... lines 9 - 12
class EasyAdminSubscriber implements EventSubscriberInterface
{
... lines 15 - 31
    public function onPreEdit(GenericEvent $event)
    {
... lines 34 - 37
    }
... lines 39 - 59
}
```

And just like before... we don't really know what the `$event` looks like. So, dump it!

Now, as *soon* as I hit edit... we see the dump! Check this out: this time, the `subject` property is actually an *array*. But, it has a `class` key set to the `User` class. We can use *that* to make sure we only run our code when we're editing a user.

In other words, `$config = $event->getSubject()` and `if $config['class']` is equal to our `User` class, then we want to check security:

```
61 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 12
13 class EasyAdminSubscriber implements EventSubscriberInterface
14 {
... lines 15 - 31
32     public function onPreEdit(GenericEvent $event)
33     {
34         $config = $event->getSubject();
35         if ($config['class'] == User::class) {
... line 36
37         }
38     }
... lines 39 - 59
60 }
```

Let's call a new method... that we'll create in a moment: `$this->denyAccessUnlessSuperAdmin()` :

```
61 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 12
13 class EasyAdminSubscriber implements EventSubscriberInterface
14 {
... lines 15 - 31
32     public function onPreEdit(GenericEvent $event)
33     {
34         $config = $event->getSubject();
35         if ($config['class'] == User::class) {
36             $this->denyAccessUnlessSuperAdmin();
37         }
38     }
... lines 39 - 59
60 }
```

At the bottom, add that: `private function denyAccessUnlessSuperAdmin()` :

```
61 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 12
13 class EasyAdminSubscriber implements EventSubscriberInterface
14 {
... lines 15 - 53
54     private function denyAccessUnlessSuperAdmin()
55     {
... lines 56 - 58
59     }
60 }
```

Now... we just need to check to see if the current user has `ROLE_SUPERADMIN` . How? Via the "authorization checker" service. To get it, type-hint a new argument with `AuthorizationCheckerInterface` . Hit `Alt` + `Enter` to create and set that property:

```
61 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 9
10 use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface;
... lines 11 - 12
13 class EasyAdminSubscriber implements EventSubscriberInterface
14 {
... line 15
16     private $authorizationChecker;
17
18     public function __construct(TokenStorageInterface $tokenStorage, AuthorizationCheckerInterface $authorizationChecker)
19     {
... line 20
21         $this->authorizationChecker = $authorizationChecker;
22     }
... lines 23 - 59
60 }
```

Then, back down below, `if (!$this->authorizationChecker->isGranted('ROLE_SUPERADMIN')`, then throw a new `AccessDeniedException()`:

```
61 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 10
11 use Symfony\Component\Security\Core\Exception\AccessDeniedException;
12
13 class EasyAdminSubscriber implements EventSubscriberInterface
14 {
... lines 15 - 53
54     private function denyAccessUnlessSuperAdmin()
55     {
56         if (!$this->authorizationChecker->isGranted('ROLE_SUPERADMIN')) {
57             throw new AccessDeniedException();
58         }
59     }
60 }
```

Make sure you use the class from the Security component:

```
61 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 10
11 use Symfony\Component\Security\Core\Exception\AccessDeniedException;
... lines 12 - 61
```

Oh, and don't forget the `new`!

```
61 lines | src/AppBundle/Event/EasyAdminSubscriber.php
... lines 1 - 12
13 class EasyAdminSubscriber implements EventSubscriberInterface
14 {
... lines 15 - 53
54     private function denyAccessUnlessSuperAdmin()
55     {
56         if (!$this->authorizationChecker->isGranted('ROLE_SUPERADMIN')) {
57             throw new AccessDeniedException();
58         }
59     }
60 }
```

See, normally, in a controller, we call `$this->denyAccessUnlessGranted()`. When we do that, *this* is actually the exception that is thrown behind the scenes. In other words, we're *really* doing the *same* thing that we normally do in a controller.

And... we're done! The service is set to be autowired, so Symfony will know to pass us the authorization checker automatically. Refresh!

Great news! Access denied! Woohoo! I've never been so happy to get kicked out of something. Our user does *not* have `ROLE_SUPERADMIN` - just `ROLE_ADMIN` and `ROLE_USER`. To double-check our logic, open `app/config/security.yml`, and, temporarily, for anyone who has `ROLE_ADMIN`, also give them `ROLE_SUPERADMIN`:

```
security:
    role_hierarchy:
        ROLE_ADMIN: [ROLE_MANAGE_GENUS, ROLE_ALLOWED_TO_SWITCH, ROLE_SUPERADMIN]
```

Now we should have access. Try it again!

Access granted! Comment-out that `ROLE_SUPERADMIN`.

Hiding the Edit Button

Time for step 2! On the list page, we need to hide the edit link, unless I have the role. This is trickier: there's no official hook inside of EasyAdminBundle to conditionally hide or show actions. But don't worry! Earlier, we overrode the list template so that we could control *exactly* what actions are displayed. Our new `filter_admin_actions()` filter lives in `EasyAdminExtension`:

8 lines | app/Resources/views/easy_admin/list.html.twig

```
... lines 1 - 2
{% block item_actions %}
    {% set _list_item_actions = _list_item_actions|filter_admin_actions(item) %}
... lines 5 - 6
{% endblock %}
```

And we added logic there to hide the delete action for any published genuses:

28 lines | src/AppBundle/Twig/EasyAdminExtension.php

```
... lines 1 - 6
class EasyAdminExtension extends \Twig_Extension
{
... lines 9 - 18
    public function filterActions(array $itemActions, $item)
    {
        if ($item instanceof Genus && $item->getIsPublished()) {
            unset($itemActions['delete']);
        }

        return $itemActions;
    }
}
```

In other words, we added our *own* hook to control which actions are displayed. We rock!

To hide the edit action, we'll need the authorization checker again. No problem! Add `public function __construct()` with one argument: `AuthorizationCheckerInterface`. Set that on a new property:

```
41 lines | src/AppBundle/Twig/EasyAdminExtension.php
... lines 1 - 6
use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface;

class EasyAdminExtension extends \Twig_Extension
{
    private $authorizationChecker;

    public function __construct(AuthorizationCheckerInterface $authorizationChecker)
    {
        $this->authorizationChecker = $authorizationChecker;
    }
... lines 17 - 39
}
```

Then, down below, we'll add some familiar code: if `$item instanceof User` and `!$this->authorizationChecker->isGranted('ROLE_SUPERADMIN')`, then unset the `edit` action:

```
41 lines | src/AppBundle/Twig/EasyAdminExtension.php
... lines 1 - 8
class EasyAdminExtension extends \Twig_Extension
{
... lines 11 - 27
    public function filterActions(array $itemActions, $item)
    {
... lines 30 - 33
        if ($item instanceof User && !$this->authorizationChecker->isGranted('ROLE_SUPERADMIN')) {
            unset($itemActions['edit']);
        }
... lines 37 - 38
    }
}
```

Phew! It's not the easiest thing ever... EasyAdminBundle... but it *does* get the job done!

Except for one... *minor* problem... there is *also* an edit button on the show page. Oh no! It looks like we need to repeat all of this for the *show* template!

Controlling the Actions in the show Template

But don't worry! With all our knowledge, this should be quick and painless.

Inside of the bundle, find the show template. And inside it, search for "actions". Here we go: block `item_actions`. To control the actions, we can do a very similar thing as the list template. In fact, copy the list template, and paste it as `show.html.twig`. Because it's in the right location, it should automatically override the one from the bundle.

Extend that base `show.html.twig` template:

```
15 lines | app/Resources/views/easy_admin/show.html.twig
{% extends '@EasyAdmin/default/show.html.twig' %}
... lines 2 - 15
```

Before, we overrode the `_list_item_actions` variable and then called the `parent()` function to render the parent block.

But... that actually won't work here! Bananas! Why not? In this case, the variable we need to override is called `_show_actions`. And... well... it's set right inside the block. That's different from `list.html.twig`, where the variable was set *above* the block. This means that if we override `_show_actions` and then call the parent block, the parent block will re-override our value! Lame!!!

No worries, it just means that we need to override the *entire* block, and avoid calling parent. Copy the block and, in `show.html.twig`, paste:

```
15 lines | app/Resources/views/easy_admin/show.html.twig
... lines 1 - 2
{% block item_actions %}
    {% set _show_actions = easyadmin_get_actions_for_show_item(_entity_config.name) %}
... line 5
    {% set _request_parameters = { entity: _entity_config.name, referer: app.request.query.get('referer') } %}

    {{ include('@EasyAdmin/default/includes/_actions.html.twig', {
        actions: _show_actions,
        request_parameters: _request_parameters,
        translation_domain: _entity_config.translation_domain,
        trans_parameters: _trans_parameters,
        item_id: _entity_id
    }, with_context = false) }}
{% endblock item_actions %}
```

Next, add our filter: `set _show_actions = _show_actions|filter_admin_actions`:

```
15 lines | app/Resources/views/easy_admin/show.html.twig
... lines 1 - 2
{% block item_actions %}
    {% set _show_actions = easyadmin_get_actions_for_show_item(_entity_config.name) %}
    {% set _show_actions = _show_actions|filter_admin_actions(entity) %}
    {% set _request_parameters = { entity: _entity_config.name, referer: app.request.query.get('referer') } %}
... lines 7 - 14
{% endblock item_actions %}
```

Remember, we need to pass the entity object as an argument to `filter_admin_actions` ... and that's another difference between show and list. Since this template is for a page that represents one entity, the variable is not called `item`, it's called `entity`.

As crazy as that looks, it should do it! Hold you breath, do a dance, and refresh!

Hey! No edit button! Go back to `security.yml` and re-add `ROLE_SUPERADMIN`:

```
security:
    role_hierarchy:
        ROLE_ADMIN: [ROLE_MANAGE_GENUS, ROLE_ALLOWED_TO_SWITCH, ROLE_SUPERADMIN]
```

Refresh now. Edit button is back. And we can even use it. One of the least-easy things in EasyAdminBundle is now done!

Chapter 20: CSV Export

When we started this tutorial, we kept getting the same question:

> Ryan, would you rather have rollerblades for feet or chopsticks for hands for the rest of your life?

I don't know the answer, but Eddie from Southern Rail can definitely answer this.

We also got this question:

> How can I add a "CSV Export" button to my admin list pages?

And now we know *why* you asked this question: it's tricky! I *want* to add a button on top that says "Export". But, when you add a custom action to the list page... those create links *next* to each item, not on top. That's *not* what we want!

Defining the Custom Action

So let's see if we can figure this out. First, in `config.yml`, under the global `list`, we *are* going to add a new action, called `export`:

159 lines | app/config/config.yml

```
... lines 1 - 80
81 easy_admin:
... lines 82 - 94
95     list:
... line 96
97         actions: ['show', 'export']
... lines 98 - 159
```

Now, if we refresh... not surprisingly, this adds an "export" link next to *every* item. And if we click it, it tries to execute a new `exportAction()` method.

So this is a bit weird: we do *not* want this new link on each row - we'll fix that in a few minutes. But we *do* need the new `export` action. Why? Because as *soon* as we add this, it's now *legal* to create a link that executes `exportAction()`. And that means that we could manually add this link somewhere else... like on top of the list page.

Adding the Custom Link (Conditionally)

Open up our custom `list.html.twig`:

8 lines | app/Resources/views/easy_admin/list.html.twig

```
1 {% extends '@EasyAdmin/default/list.html.twig' %}
2
3 {% block item_actions %}
4     {% set _list_item_actions = _list_item_actions|filter_admin_actions(item) %}
5
6     {{ parent() }}
7 {% endblock %}
```

I'll also hold `Command` and click to open the parent `list.html.twig` from the bundle. If you scroll down a little bit, you'll find a block called `global_actions`. Ah, it looks like it's rendering the search field. The `global_actions` block represents this area on top.

In other words, if we want to add a new link here, `global_actions` is the place to do it! Copy that block name and override it inside of our template: `global_actions` and `endlock`:

```
22 lines | app/Resources/views/easy_admin/list.html.twig
1  {% extends '@EasyAdmin/default/list.html.twig' %}
...lines 2 - 8
9  {% block global_actions %}
...lines 10 - 20
21 {% endblock global_actions %}
```

Inside, we'll add the Export button.

But wait! I have an idea. What if we only want to add the export button to *some* entities? Sure, I added the `export` action in the `global` section... but we could still remove it from any other entity by saying `-export`. Basically, I want this button to be smart: I only want to show it *if* the `export` action is enabled for this entity.

How can we figure that out? In the parent template, you'll find a really cool if statement that checks to see if an action is enabled. Steal it!

In our case, change `search` to `export`:

```
22 lines | app/Resources/views/easy_admin/list.html.twig
1  {% extends '@EasyAdmin/default/list.html.twig' %}
...lines 2 - 8
9  {% block global_actions %}
...lines 10 - 11
12     {% if easyadmin_action_is_enabled_for_list_view('export', _entity_config.name) %}
...lines 13 - 19
20     {% endif %}
21 {% endblock global_actions %}
```

At this point, we can do *whatever* we want. So, very simply, let's add a new link that points to the `export` action. Add a `button-action` div for styling:

```
22 lines | app/Resources/views/easy_admin/list.html.twig
1  {% extends '@EasyAdmin/default/list.html.twig' %}
...lines 2 - 8
9  {% block global_actions %}
...lines 10 - 11
12     {% if easyadmin_action_is_enabled_for_list_view('export', _entity_config.name) %}
13
14         <div class="button-action">
...lines 15 - 18
19         </div>
20     {% endif %}
21 {% endblock global_actions %}
```

Then, inside, a link with `btn btn-primary` and an `href`. How can we point to the `exportAction()`? Remember, the bundle only has one route: `easyadmin`. For the parameters, use a special variable called `_request_parameters`. This is something that EasyAdminBundle gives us, and it contains all of the query parameters. You'll see why that's cool in a minute.

But the *most* important thing is to add another query parameter called `action` set to `export`:

```
22 lines | app/Resources/views/easy_admin/list.html.twig
1  {% extends '@EasyAdmin/default/list.html.twig' %}
... lines 2 - 8
9  {% block global_actions %}
... lines 10 - 11
12     {% if easyadmin_action_is_enabled_for_list_view('export', _entity_config.name) %}
13
14         <div class="button-action">
15             <a class="btn btn-primary" href="{{ path('easyadmin', _request_parameters|merge({ action: 'export' })) }}">
... lines 16 - 17
18             </a>
19         </div>
20     {% endif %}
21 {% endblock global_actions %}
```

Oh boy, that's ugly. But, it works great: it generates a route to `easyadmin` where `action` is set to `export` and all the existing query parameters are maintained.

Phew! Inside, add a download icon and say "Export":

```
22 lines | app/Resources/views/easy_admin/list.html.twig
1  {% extends '@EasyAdmin/default/list.html.twig' %}
... lines 2 - 8
9  {% block global_actions %}
... lines 10 - 11
12     {% if easyadmin_action_is_enabled_for_list_view('export', _entity_config.name) %}
13
14         <div class="button-action">
15             <a class="btn btn-primary" href="{{ path('easyadmin', _request_parameters|merge({ action: 'export' })) }}">
16                 <i class="fa fa-download"></i>
17                 Export
18             </a>
19         </div>
20     {% endif %}
21 {% endblock global_actions %}
```

Try it! Woh! We have an export button... but nothing else. I *love* to forget the `parent()` call:

```
22 lines | app/Resources/views/easy_admin/list.html.twig
1  {% extends '@EasyAdmin/default/list.html.twig' %}
... lines 2 - 8
9  {% block global_actions %}
10     {{ parent() }}
11
12     {% if easyadmin_action_is_enabled_for_list_view('export', _entity_config.name) %}
13
14         <div class="button-action">
15             <a class="btn btn-primary" href="{{ path('easyadmin', _request_parameters|merge({ action: 'export' })) }}">
16                 <i class="fa fa-download"></i>
17                 Export
18             </a>
19         </div>
20     {% endif %}
21 {% endblock global_actions %}
```

Try it again. Beautiful!

When I click export, it of course looks for `exportAction` in our controller... in this case, `GenusController`.

Adding the Custom Action

Remember: we're not going to support this export action for all of our entities. And to make this error clearer, open `AdminController` - our base controller - and create a `public function exportAction()` that simply throws a new `RuntimeException`: "Action for exporting an entity is not defined":

14 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php

```
... lines 1 - 6
class AdminController extends BaseAdminController
{
    public function exportAction()
    {
        throw new \RuntimeException('Action for exporting an entity not defined');
    }
}
```

If we configure everything correctly, and implement this method for all entities that need it, we should never see this error. But... just in case.

Now, to the *real* work. To add an export for genus, we have two options. First, in `AdminController`, we *could* create a `public function exportGenusAction()`. Remember, whenever EasyAdminBundle calls *any* of our actions - even custom actions - it always looks for that specially named method: `export<EntityName>Action()`. *Or*, we can be a bit more organized, and create a custom controller for each entity. That's what we've done already. So, in `GenusController`, add `public function exportAction()`:

56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php

```
... lines 1 - 7
class GenusController extends AdminController
{
... lines 10 - 34
    public function exportAction()
    {
... lines 37 - 53
    }
}
```

Adding the CSV Export Logic

To save time, we've already done most of the work for the CSV export. If you downloaded the starting code, in the `Service` directory, you should have a `CsvExporter` class:

85 lines | src/AppBundle/Service/CsvExporter.php

```
... lines 1 - 2
namespace AppBundle\Service;

use AppBundle\Entity\Genus;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\ORM\QueryBuilder;
use Symfony\Component\HttpFoundation\StreamedResponse;

class CsvExporter
{
    public function getResponseFromQueryBuilder(QueryBuilder $queryBuilder, $columns, $filename)
    {
        $entities = new ArrayCollection($queryBuilder->getQuery()->getResult());
        $response = new StreamedResponse();

        if (is_string($columns)) {
            $columns = $this->getColumnsForEntity($columns);
        }

        $response->setCallback(function () use ($entities, $columns) {
            $handle = fopen('php://output', 'w+');

            // Add header
            fputcsv($handle, array_keys($columns));

            while ($entity = $entities->current()) {
                $values = [];

                foreach ($columns as $column => $callback) {
                    $value = $callback;

                    if (is_callable($callback)) {
                        $value = $callback($entity);
                    }

                    $values[] = $value;
                }

                fputcsv($handle, $values);

                $entities->next();
            }

            fclose($handle);
        });

        $response->headers->set('Content-Type', 'text/csv; charset=utf-8');
        $response->headers->set('Content-Disposition', 'attachment; filename="' . $filename . '"');

        return $response;
    }

    private function getColumnsForEntity($class)
    {
... lines 56 - 82
    }
}
```

Basically, we pass it a `QueryBuilder`, an array of column information, or the entity's class name - which is mapped to an array of column info thanks to this special function, and the filename we want. Then, it creates the CSV and returns it as a

`StreamedResponse`. So all we need to do is call this method and return it from our controller!

I'll paste a little bit of code in the action to get us started:

```
56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php
... lines 1 - 7
8  class GenusController extends AdminController
9  {
... lines 10 - 34
35     public function exportAction()
36     {
37         $sortDirection = $this->request->query->get('sortDirection');
38         if (empty($sortDirection) || !in_array(strtoupper($sortDirection), ['ASC', 'DESC'])) {
39             $sortDirection = 'DESC';
40         }
... lines 41 - 53
54     }
55 }
```

When we created the export link, we kept the existing query parameters. That means we should have a `sortDirection` parameter... which is a nice way of making the export order match the list order.

To create the query builder, we can actually use a protected function on the base class called `createListQueryBuilder()`:

```
56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php
... lines 1 - 7
8  class GenusController extends AdminController
9  {
... lines 10 - 34
35     public function exportAction()
36     {
37         $sortDirection = $this->request->query->get('sortDirection');
38         if (empty($sortDirection) || !in_array(strtoupper($sortDirection), ['ASC', 'DESC'])) {
39             $sortDirection = 'DESC';
40         }
41
42         $queryBuilder = $this->createListQueryBuilder(
... lines 43 - 46
47         );
... lines 48 - 53
54     }
55 }
```

Pass this the entity class, either `Genus::class` or `$this->entity['class']` ... in case you want to make this method reusable across multiple entities:

```
56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php
... lines 1 - 7
8  class GenusController extends AdminController
9  {
... lines 10 - 34
35     public function exportAction()
36     {
... lines 37 - 41
42         $queryBuilder = $this->createListQueryBuilder(
43             $this->entity['class'],
... lines 44 - 46
47         );
... lines 48 - 53
54     }
55 }
```

Next, pass the sort direction and then the sort field: `$this->request->query->get('sortField')` :

```
56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php
... lines 1 - 7
8  class GenusController extends AdminController
9  {
... lines 10 - 34
35     public function exportAction()
36     {
... lines 37 - 41
42         $queryBuilder = $this->createListQueryBuilder(
43             $this->entity['class'],
44             $sortDirection,
45             $this->request->query->get('sortField'),
... line 46
47         );
... lines 48 - 53
54     }
55 }
```

Finally, pass in the `dql_filter` option: `$this->entity['list']['dql_filter']` :

```
56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php
... lines 1 - 7
8  class GenusController extends AdminController
9  {
... lines 10 - 34
35     public function exportAction()
36     {
... lines 37 - 41
42         $queryBuilder = $this->createListQueryBuilder(
43             $this->entity['class'],
44             $sortDirection,
45             $this->request->query->get('sortField'),
46             $this->entity['list']['dql_filter']
47         );
... lines 48 - 53
54     }
55 }
```

This is kind of cool. We're using the `entity` configuration array - which is always full of goodies - to actually read the `list` key and the `dql_filter` key below it. If we have a DQL filter on this entity, the CSV export will know about it!

Ok, *finally*, we're ready to use the `CsvExporter` class. Because I'm using the new Symfony 3.3 service configuration, the

`CsvExporter` is already registered as a *private* service:

32 lines | app/config/services.yml

```yaml
... lines 1 - 5
services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true
        autoconfigure: true
        public: false

    AppBundle\:
        resource: '../../src/AppBundle/*'
        exclude: '../../src/AppBundle/{Entity,Repository,Tests}'

    AppBundle\Controller\:
        resource: '../../src/AppBundle/Controller'
        public: true
        tags: ['controller.service_arguments']
... lines 21 - 32
```

Using DI in a Fake Action

The Symfony 3.3 way of accessing a service from a controller is as an argument to the action. But... remember: this is *not* a real action. I mean, it's not called by the normal core, Symfony controller system. Nope, it's called by EasyAdminBundle... and none of the normal controller argument tricks work. You can't type-hint the Request or any services.

Because of this, we're going to use *classic* dependency injection. We can do this because this controller - well *any* controller if you're using the Symfony 3.3 configuration - is registered as a service. Add a `__construct()` function and type-hint the `CsvExporter` class. I'll press `Alt` + `Enter` to create a property and set it:

56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php

```php
... lines 1 - 5
use AppBundle\Service\CsvExporter;

class GenusController extends AdminController
{
    private $csvExporter;

    public function __construct(CsvExporter $csvExporter)
    {
        $this->csvExporter = $csvExporter;
    }
... lines 16 - 54
}
```

Back down below, just return `$this->csvExporter->getResponseFromQueryBuilder()` and pass it the `$queryBuilder`, `Genus::class`, and `genuses.csv` - the filename:

```
56 lines | src/AppBundle/Controller/EasyAdmin/GenusController.php
... lines 1 - 7
class GenusController extends AdminController
{
... lines 10 - 34
    public function exportAction()
    {
... lines 37 - 41
        $queryBuilder = $this->createListQueryBuilder(
... lines 43 - 46
        );

        return $this->csvExporter->getResponseFromQueryBuilder(
            $queryBuilder,
            Genus::class,
            'genuses.csv'
        );
    }
}
```

Deep breath... refresh! It downloaded! Ha! In my terminal. I'll:

```
$ cat ~/Downloads/genuses.csv
```

There it is!

Hiding the Extra

There's just *one* last problem: on the list page... we still have those weird export links on each row. That's technically fine... but it's super confusing. The *only* reason we added this `export` action was so that it would be a valid action to call:

```
159 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... lines 82 - 94
    list:
... line 96
        actions: ['show', 'export']
... lines 98 - 159
```

Unfortunately, this *also* gave us those links!

No worries, we just need to hide that link manually... and we already have a filter to do this! Open `EasyAdminExtension` and `filterActions()`. Now, just unset `$itemActions['export']`. That looks a little crazy, so I'll add a comment: "This action is rendered manually":

```
44 lines | src/AppBundle/Twig/EasyAdminExtension.php
... lines 1 - 27
    public function filterActions(array $itemActions, $item)
    {
... lines 30 - 37
        // export action is rendered by us manually
        unset($itemActions['export']);
... lines 40 - 41
    }
}
```

Try it! Yes! We have the export button on top... but *not* on each row. This is a tricky - but *valid* - use-case for custom actions.

Chapter 21: Tweaking the Form Layout

We've talked a lot about customizing the forms... which mostly means using the config to change the field types or adding a custom form theme to control how the individual fields look.

But, what about the form *layout*? Like, what if I wanted to put the email & full name fields in a section that floats to the left, and these other two fields in a section that floats to the right?

Well... that's not really part of the form component: we usually do this by adding markup to our template. But of course, the template lives inside EasyAdminBundle!

In turns out, there are two great ways to control your form's layout. First..., well, you could just override the template and go *crazy*! For example, inside the bundle, find `new.html.twig`. It has a block called `entity_form`. And, to render the form, it just calls the `form()` function. This means that there's no form layout at *all* by default: it just barfs out all the fields.

But... that's awesome! Because we could override this template, replace *just* the `entity_form` block, and go bonkers by rendering the form *however* we need. And since we can override each template on an entity-by-entity basis... well... suddenly it's super easy to customize the exact form layout for each section.

Form Layout Config Customizations

Phew! But... there is an *even* easier way that works for about 90% of the use-cases. And that is... of course... with configuration. EasyAdminBundle comes with a bunch of different ways to add dividers, sections and groups inside the form.

So, let's do it! Start with `User`. Let's reorganize things: put `fullName` on top, then add a new type called `divider`. Put `avatarUri` after the divider, then another divider, `email`, divider, `isScientist` and `universityName`:

```
159 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
... lines 99 - 159
```

On a technical level... this is kind of geeky cool: `divider` is a *fake* form field! I mean, in the bundle itself, it is *literally* a form field. And you can see a few others, like `section` and `group`. We'll use all three.

Try out the page! Hello divider! It's nothing too fancy, but it's nice.

Adding a Section

To go further, we can divide things into sections by using `type: section`. For example, start here by saying `type: section, label: 'User Details'`:

```
159 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
... lines 99 - 159
```

And then, inside, we'll have `fullName`, keep the `divider`, keep `avatarUri`, but replace the next `divider` with the expanded syntax: `type: section, label: 'Contact Information'`. And like many other places, you can add an `icon`, a `help` message and a `css_class`:

```
159 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
... lines 99 - 159
```

With `email` in its own section, change the last divider to `type: section, label: Education`:

```
159 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
... lines 99 - 159
```

Ok, let's see how this looks!

Not bad! Each field appears inside whatever section is above it.

Reorganizing all Fields under form

The last organizational trick is the *group*, which gives you *mad* control over the width and float of a bunch of fields.

To see an example, go up to the `Genus` form.

And first, remember how we organized most fields under the `form` key... but then tweaked a few final things under `new` and `edit`? Well, when EasyAdminBundle reads this, all of the fields under `form` are added first... and *then* any extra fields under `new` or `edit` are added. That means, in `edit`, our `slug` field is printed *last* on the form. And... there's not really a good way to control that. This gets even a little bit more problematic when you want to organize fields into sections or groups. How could you organize the `slug` field into the same section as `name`? Right now, you can't!

For that reason, it's best to configure *all* of your fields under `form`. Then, use `new` and `edit` *only* to *remove* fields you don't want. Copy the `slug` field and remove `edit` entirely. Then, under `form`, paste this near the top:

```
158 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
99          Genus:
... lines 100 - 130
131             form:
132                 fields:
133                     -
134                         property: id
135                         type_options: {disabled: true}
136                     -
137                         property: 'slug'
138                         help: 'unique auto-generated value'
139                         type_options: { disabled: true }
... lines 140 - 158
```

To keep `slug` off of the `new` form, just add `-slug`:

```
158 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
99          Genus:
... lines 100 - 151
152             new:
153                 fields:
... line 154
155                     - '-slug'
... lines 156 - 158
```

The end result is the same, but with complete control over the field order.

Adding Form Groups

Ok, back to adding *groups*. First, move `id` and `slug` to the end of the form. Then, on top, add a new group: `type: group, css_class: 'col-sm-6', label: 'Basic Information'`:

```
170 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
99          Genus:
... lines 100 - 130
131             form:
132                 fields:
133                     - { type: 'group', css_class: 'col-sm-6', label: 'Basic information' }
... lines 134 - 170
```

You can picture what this is doing: adding a div with `col-sm-6`, putting a header inside of it, and then printing any fields below that, but in the div.

And that's huge! Because thanks to the `col-sm-6` CSS class, we can really start organizing how things look.

Move `funFact` and `isPublished` a bit further down. Then, after `subFamily`, add a `section` labeled `Optional`:

```
170 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... lines 82 - 97
98      entities:
99          Genus:
... lines 100 - 130
131             form:
132                 fields:
133                     - { type: 'group', css_class: 'col-sm-6', label: 'Basic information' }
134                     - name
135                     - speciesCount
136                     - { property: 'firstDiscoveredAt', type_options: { widget: 'single_text' }}
137                     - { property: 'subFamily', type: 'easyadmin_autocomplete' }
138                     - { type: 'section', label: 'Optional' }
... lines 139 - 170
```

Yep, you can totally mix-and-match groups and sections.

At this point, `funFact` and `isPublished` *will* still be in the group, but they'll also be in a section within that group. And since `genusScientists` is pretty big, let's put that in their own group with `css_class: col-sm-6` and `label: Studied by...`:

170 lines | app/config/config.yml

```yaml
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
        Genus:
... lines 100 - 130
            form:
                fields:
                    - { type: 'group', css_class: 'col-sm-6', label: 'Basic information' }
                    - name
                    - speciesCount
                    - { property: 'firstDiscoveredAt', type_options: { widget: 'single_text' }}
                    - { property: 'subFamily', type: 'easyadmin_autocomplete' }
                    - { type: 'section', label: 'Optional' }
                    - { property: 'funFact', type: 'textarea', css_class: 'js-markdown-input' }
                    - isPublished

                    - { type: 'group', css_class: 'col-sm-6', label: 'Studied by ...' }
... lines 143 - 170
```

Finally, at the bottom, add one more group. I'll use the expanded format this time: `css_class: col-sm-6` and `label: Identification`. And yep, groups can have `icon` and `help` keys:

170 lines | app/config/config.yml

```yaml
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
        Genus:
... lines 100 - 130
            form:
                fields:
                    - { type: 'group', css_class: 'col-sm-6', label: 'Basic information' }
                    - name
                    - speciesCount
                    - { property: 'firstDiscoveredAt', type_options: { widget: 'single_text' }}
                    - { property: 'subFamily', type: 'easyadmin_autocomplete' }
                    - { type: 'section', label: 'Optional' }
                    - { property: 'funFact', type: 'textarea', css_class: 'js-markdown-input' }
                    - isPublished

                    - { type: 'group', css_class: 'col-sm-6', label: 'Studied by ...' }
                    -
                        property: 'genusScientists'
                        type: 'text'
                        type_options:
                            mapped: false
                            attr: { class: 'js-genus-scientists-field' }

                    -
                        type: 'group'
                        css_class: 'col-sm-6'
                        label: 'Identification'
                        icon: 'id-card-o'
                        help: 'For administrators'
                    -
                        property: id
                        type_options: {disabled: true}
                    -
                        property: 'slug'
                        help: 'unique auto-generated value'
                        type_options: { disabled: true }
... lines 163 - 170
```

Phew! While I did this, I added some line breaks *just* so that this all looks a bit more clear: here's one group, here's a second group, the last group is at the bottom.

But what does it actually *look* like? Let's find out! Refresh!

Oh, this feels good. The "Basic Information" group is on the left with the "Optional section" at the bottom. The other two groups float to the right.

Now, sometimes, you might want to force the "Identification" group to go onto its own line. Basically, you want to add a CSS `clear` after the first two groups.

To do that, on the group, add a special CSS class, called `new-row`:

170 lines | app/config/config.yml

```yaml
... lines 1 - 80
easy_admin:
... lines 82 - 97
    entities:
        Genus:
... lines 100 - 130
            form:
                fields:
... lines 133 - 149
                    -
                        type: 'group'
                        css_class: 'col-sm-6 new-row'
                        label: 'Identification'
                        icon: 'id-card-o'
                        help: 'For administrators'
... lines 156 - 170
```

And *now* it floats to the next line. So, groups are a really, really neat way to control how things are rendered. It adds some nice markup, and we can add whatever classes we need. So, there's not much you can't do.

Chapter 22: Dashboard & Menu Customizations

The *only* thing we have *not* talked about is this big, giant menu on the left! This menu is *actually* the key to one other commonly-asked question: how do I create an admin dashboard?

The answer... like always... lives in the configuration! In `config.yml`, under `design`, add a `menu` key:

```
177 lines | app/config/config.yml
... lines 1 - 80
81 easy_admin:
... line 82
83     design:
... lines 84 - 94
95         menu:
... lines 96 - 177
```

This works like many other config keys. First, it has a simple format: just list the sections in the order you want them: `User`, `Genus`, `GenusHorde` and `SubFamily`:

```
177 lines | app/config/config.yml
... lines 1 - 80
81 easy_admin:
... line 82
83     design:
... lines 84 - 94
95         menu:
96             - User
97             - Genus
98             - GenusHorde
99             - GenusNote
100            - SubFamily
... lines 101 - 177
```

These keys are coming from the keys that we chose for each section's configuration:

```
177 lines | app/config/config.yml
... lines 1 - 80
81 easy_admin:
... lines 82 - 104
105    entities:
106        Genus:
... lines 107 - 174
175        GenusHorde:
... lines 176 - 177
```

These could have been anything.

Thanks to this, the `User` link will move from the bottom all the way to the top. There are a *lot* of other customizations you can make to the menu... but before we get there, I want a dashboard! Yea know, an admin homepage full of important-looking graphs!

Adding a Dashboard

If you downloaded the course code, you should have a `tutorial/` directory. Inside, it has an `AdminController` with a `dashboardAction()`. Copy that. Then, in `src/AppBundle/Controller/EasyAdmin`, open our `AdminController` and paste it there:

31 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php

```
... lines 1 - 8
class AdminController extends BaseAdminController
{
... lines 11 - 15
    /**
     * @Route("/dashboard", name="admin_dashboard")
     */
    public function dashboardAction()
    {
        $em = $this->getDoctrine()->getManager();
        $genusRepository = $em->getRepository(Genus::class);

        return $this->render('easy_admin/dashboard.html.twig', [
            'genusCount' => $genusRepository->getGenusCount(),
            'publishedGenusCount' => $genusRepository->getPublishedGenusCount(),
            'randomGenus' => $genusRepository->findRandomGenus()
        ]);
    }
}
```

Thanks to the prefix on the route import:

14 lines | app/config/routing.yml

```
... lines 1 - 9
easy_admin_bundle:
... lines 11 - 12
    prefix: /easyadmin
```

This creates a new `/easyadmin/dashboard` route named `admin_dashboard`. Oh, I'm missing my `use` statement for `@Route`. I'll re-type that and hit enter so that it auto-completes the `use` statement on top:

31 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php

```
... lines 1 - 6
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class AdminController extends BaseAdminController
{
... lines 11 - 15
    /**
     * @Route("/dashboard", name="admin_dashboard")
     */
    public function dashboardAction()
    {
... lines 21 - 28
    }
}
```

Perfect!

This renders a template, which I will *also* grab from the `tutorial/` directory. Paste that in `app/Resources/views/easy_admin`:

84 lines | app/Resources/views/easy_admin/dashboard.html.twig

```
{% extends '@EasyAdmin/default/layout.html.twig' %}

{% set _content_title = 'Admin dashboard' %}

{% block page_title -%}
    {{ _content_title }}
{%- endblock %}
```

```
{% block content_header %}
    <h1 class="title">{{ _content_title }}</h1>
{% endblock %}

{% block main %}
    <div class="row">
        <div class="col-sm-4">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3 class="panel-title">Stats</h3>
                </div>

                <div class="panel-body">
                    <ul class="list-group" style="margin-left: 0;">
                        <li class="list-group-item">
                            <span class="badge">{{ genusCount }}</span>
                            Genus count
                        </li>

                        <li class="list-group-item">
                            <span class="badge">{{ publishedGenusCount }}</span>
                            Published genus count
                        </li>
                    </ul>
                </div>
            </div>
        </div>

        <div class="col-sm-4">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3 class="panel-title">Chart</h3>
                </div>

                <div class="panel-body">
                    <script type="text/javascript"
                            src="https://ssl.gstatic.com/trends_nrtr/1015_RC10/embed_loader.js"></script>
                    <script type="text/javascript">
                        trends.embed.renderExploreWidget("TIMESERIES", {
                            "comparisonItem": [{
                                "keyword": "funny cat videos",
                                "geo": "",
                                "time": "2012-05-17 2017-05-17"
                            }], "category": 0, "property": ""
                        }, {
                            "exploreQuery": "q=funny%20cat%20videos",
                            "guestPath": "https://trends.google.com:443/trends/embed/"
                        });
                    </script>
                </div>
            </div>
        </div>

        <div class="col-sm-4">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3 class="panel-title">{{ randomGenus.name }}</h3>
                </div>

                <div class="panel-body">
```

```
                <p>
                    <img src="{{ asset('images/octopus-rubescens.jpg') }}" />
                </p>

                <p>{{ randomGenus.funFact }}</p>

                <a href="{{ path('easyadmin', {entity: 'Genus', action: 'show', id: randomGenus.id}) }}"
                   class="btn btn-primary">
                    Show details
                </a>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

At this point... the page *should* work. Cool... but how can I tell EasyAdminBundle to show this page when we go to the admin section's homepage? Right now, if you go directly to `/easyadmin`, it will take you to whatever the first-defined entity section is... so Genus.

Adding the Dashboard Menu Link

But... add a new menu item and use the *expanded* config format with `label: Dashboard`, `route: admin_dashboard` and - here is the key - `default: true`:

178 lines | app/config/config.yml

```
... lines 1 - 80
81 easy_admin:
... line 82
83     design:
... lines 84 - 94
95         menu:
96             - { label: 'Dashboard', route: 'admin_dashboard', default: true }
... lines 97 - 178
```

Thanks to `default: true`, when you click on the AquaNote logo to go to the admin homepage... ah! You'll get an error! That was *not* the dramatic success moment I was hoping for.

But... look! It *did* redirect to `/easyadmin/dashboard`! The error is just a Ryan mistake: I forgot a `use` statement for my `Genus` class. Add that on top:

31 lines | src/AppBundle/Controller/EasyAdmin/AdminController.php

```
... lines 1 - 4
5 use AppBundle\Entity\Genus;
... lines 6 - 8
9 class AdminController extends BaseAdminController
10 {
... lines 11 - 18
19     public function dashboardAction()
20     {
... line 21
22         $genusRepository = $em->getRepository(Genus::class);
... lines 23 - 28
29     }
30 }
```

Try it again! Hello super fancy dashboard! Where apparently, somehow, interest in funny cat videos has been *decreasing*. Well, anyways, say hello to your new dashboard. Where hopefully, you will build infinitely more useful graphs than this.

Now, back to customizing that menu...

Chapter 23: Customizing the Menu

What else can we do with the menu? Well, of course, we can *expand* the simple entity items to get more control. To expand it, add `entity: User`. Now we can go *crazy* with `label: Users`:

```
177 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... lines 84 - 94
95          menu:
... line 96
97              - { entity: 'User', label: 'Users' }
... lines 98 - 177
```

Remember, you can *also* specify the label under the `User` entity itself. If we added a `label` key here, it would apply to the menu, the header to that section and a few other places. But under `menu`, it *just* changes the menu text.

Do the same thing for `GenusNote`, with `label: Notes`, and also for sub families: `entity: SubFamily, label: Sub-Families`:

```
177 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... lines 84 - 94
95          menu:
... lines 96 - 99
100             - { entity: 'GenusNote', label: 'Notes' }
101             - { entity: 'SubFamily', label: 'Sub-Families' }
... lines 102 - 177
```

At this point, it should be *no* surprise that we can control the *icon* for each menu. Like, `icon: user`, `icon: sticky-note` and `icon: ''`:

```
177 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... lines 84 - 94
95          menu:
... line 96
97              - { entity: 'User', label: 'Users', icon: 'user' }
... lines 98 - 99
100             - { entity: 'GenusNote', label: 'Notes', icon: 'sticky-note' }
101             - { entity: 'SubFamily', label: 'Sub-Families', icon: '' }
... lines 102 - 177
```

Before configuring anything, each item has a little arrow icon. With empty quotes, even that icon is gone.

Adding Menu Separators & Sub-Menus

Oh, but the fanciness does, not, stop! The menu does a *lot* more than just simple links: it has separators, groups and sub-links. Above `Genus`, create a new item that *only* has a label:

```
178 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 94
        menu:
... line 96
            - { entity: 'User', label: 'Users', icon: 'user' }
            - { label: 'Genus' }
... lines 99 - 178
```

Yep, this has *no* `route` key and no `entity` key. We're not linking to anything.

Instead, this just adds a nice separator. Or, you can go a step further and create a sub-menu. Change this new menu item to use the expanded format. Then, add a `children` key. Indent *all* the other links so that they live under this:

```
181 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 94
        menu:
... line 96
            - { entity: 'User', label: 'Users', icon: 'user' }
            -
                label: 'Genus'
                children:
                    - Genus
                    - GenusHorde
... line 103
                    - { entity: 'GenusNote', label: 'Notes', icon: 'sticky-note' }
                    - { entity: 'SubFamily', label: 'Sub-Families', icon: '' }
... lines 106 - 181
```

And just to make it even nicer, add a separator called `Related`:

```
181 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 94
        menu:
... line 96
            - { entity: 'User', label: 'Users', icon: 'user' }
            -
                label: 'Genus'
                children:
                    - Genus
                    - GenusHorde
                    - { label: 'Related' }
                    - { entity: 'GenusNote', label: 'Notes', icon: 'sticky-note' }
                    - { entity: 'SubFamily', label: 'Sub-Families', icon: '' }
... lines 106 - 181
```

Try it! Try it! Nice! The `Genus` menu expands to show the sub-items and the `Related` separator.

Custom Links!

We can also link to the different entity sections, but with different *sort* options. We already have a `Genus` link that will take us to that page with the normal sort. But let's not limit ourselves! We could also add another link to that *same* section, with a different label: `Genuses (sorted by ID)` and a `params` key:

```
188 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... lines 84 - 94
95          menu:
... lines 96 - 97
98              -
99                  label: 'Genus'
100                 children:
101                     - Genus
102                     -
103                         entity: 'Genus'
104                         label: 'Genuses (sorted by ID)'
105                         params:
... lines 106 - 188
```

Here, we can control *whatever* query parameters we want, like `sortField: id`, `sortDirection: ASC` and... heck `pizza: delicious`:

```
188 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... lines 84 - 94
95          menu:
... lines 96 - 97
98              -
99                  label: 'Genus'
100                 children:
101                     - Genus
102                     -
103                         entity: 'Genus'
104                         label: 'Genuses (sorted by ID)'
105                         params:
106                             sortField: 'id'
107                             sortDirection: 'ASC'
108                             pizza: 'delicious'
... lines 109 - 188
```

That last query parameter won't do anything... but it doesn't make it any less true!

Ok, refresh! Then try out that new link. Yea! We're sorting by id and you might also notice in the address bar that `pizza=delicious`.

On that note, one of the other query parameters is `action`, which we can also set to anything. Copy this entire new menu link and - at the top of `children` - paste it. This time, let's link to the show page of 1 *specific* genus... our favorite "Pet genus". To do that, set `action` to `show` and `id` to some id in the database, like 2:

```
195 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 94
        menu:
... lines 96 - 97
            -
                label: 'Genus'
                children:
                    -
                        entity: 'Genus'
                        label: 'Pet genus'
                        icon: 'paw'
                        params:
                            action: 'show'
                            id: 2
                    - Genus
... lines 109 - 195
```

This isn't anything special, we're just taking advantage of how the query parameters work in EasyAdminBundle.

And while we're here, it might also be nice to add a link to the front-end of our app. This is also nothing special: add a new link that points to the `app_genus_list` route called "Open front-end":

```
196 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 94
        menu:
... line 96
            - { label: 'Open front-end', route: 'app_genus_list' }
... lines 98 - 196
```

Refresh! And try that link. Nice!

External Links

In addition to routes, if you want, you can just link to external URLs. Go to the bottom of the list... and make sure we're at the root level. Add a new section called "Important stuff" with `icon: explanation` and a `children` key:

```
208 lines | app/config/config.yml
... lines 1 - 80
easy_admin:
... line 82
    design:
... lines 84 - 94
        menu:
... lines 96 - 120
            -
                label: 'Important stuff'
                icon: 'exclamation'
                children:
... lines 125 - 208
```

I'll paste a couple of *very* important external links for silly kittens and wet cats:

```
208 lines | app/config/config.yml
... lines 1 - 80
81  easy_admin:
... line 82
83      design:
... lines 84 - 94
95          menu:
... lines 96 - 120
121             -
122                 label: 'Important stuff'
123                 icon: 'exclamation'
124                 children:
125                     -
126                         label: 'Silly kittens'
127                         url: 'https://www.youtube.com/results?search_query=silly+kittens'
128                         target: '_blank'
129                     -
130                         label: 'Wet cats'
131                         url: 'http://www.boredpanda.com/funny-wet-cats/'
132                         target: '_blank'
... lines 133 - 208
```

Yep, instead of `entity` or `route` keys, you can skip all of that and just add `url`. And of course, you can set the `target` on any item.

Re-organizing the Config

Ok team, our admin menu is complete! The *last* thing I want to show you isn't anything special to this bundle: it's just a nice way to organize any configuration. In fact, this trick will become the standard way to organize things in Symfony 4.

Right now, well, our admin configuration goes from line 81 of `config.yml` to line

1. Wow! It's *huge*!

To clear things up, I'd like to create a new file called `admin.yml`. Copy *all* of this config, remove it, and add it to `admin.yml`:

```
128 lines | app/config/admin.yml
easy_admin:
    site_name: 'Aqua<i>Note</i>'
    design:
        brand_color: '#81b9ba'
        assets:
            css: ['css/custom_backend.css']
            js:
                - 'https://unpkg.com/snarkdown@1.2.2/dist/snarkdown.umd.js'
                - 'js/custom_backend.js'
        templates:
            field_id: 'admin/fields/_id.html.twig'
        form_theme:
            - horizontal
            - easy_admin/_form_theme.html.twig
        menu:
            - { label: 'Dashboard', route: 'admin_dashboard', default: true }
            - { label: 'Open front-end', route: 'app_genus_list' }
            - { entity: 'User', label: 'Users', icon: 'user' }
            -
                label: 'Genus'
                children:
                    -
                        entity: 'Genus'
                        label: 'Pet genus'
                        icon: 'paw'
```

```yaml
                    params:
                        action: 'show'
                        id: 2
                - Genus
                -
                    entity: 'Genus'
                    label: 'Genuses (sorted by ID)'
                    params:
                        sortField: 'id'
                        sortDirection: 'ASC'
                        pizza: 'delicious'
                - GenusHorde
                - { label: 'Related' }
                - { entity: 'GenusNote', label: 'Notes', icon: 'sticky-note' }
                - { entity: 'SubFamily', label: 'Sub-Families', icon: '' }
        -
            label: 'Important stuff'
            icon: 'exclamation'
            children:
                -
                    label: 'Silly kittens'
                    url: 'https://www.youtube.com/results?search_query=silly+kittens'
                    target: '_blank'
                -
                    label: 'Wet cats'
                    url: 'http://www.boredpanda.com/funny-wet-cats/'
                    target: '_blank'
    list:
        title: 'List of %%entity_label%%'
        actions: ['show', 'export']
    entities:
        Genus:
            class: AppBundle\Entity\Genus
            controller: AppBundle\Controller\EasyAdmin\GenusController
            label: Genuses
            help: Genuses are not covered under warranty!
            list:
                help: Do not feed!
                actions:
                - { name: 'edit', icon: 'pencil', label: 'Edit' }
                - { name: 'show', icon: 'info-circle', label: '' }
                fields:
                - 'id'
                - 'name'
                - 'isPublished'
                - { property: 'firstDiscoveredAt', format: 'M Y', label: 'Discovered' }
                - 'funFact'
                - { property: 'speciesCount', format: '%b' }
                sort: 'name'
            search:
                help: null
                fields: ['id', 'name']
            show:
                actions:
                -
                    name: 'genus_feed'
                    type: 'route'
                    label: 'Feed genus'
                    css_class: 'btn btn-info'
```

```
                    icon: 'cutlery'
                - { name: 'changePublishedStatus', css_class: 'btn' }
#           templates:
#               field_id: 'admin/fields/_id.html.twig'
            form:
                fields:
                - { type: 'group', css_class: 'col-sm-6', label: 'Basic information' }
                - name
                - speciesCount
                - { property: 'firstDiscoveredAt', type_options: { widget: 'single_text' }}
                - { property: 'subFamily', type: 'easyadmin_autocomplete' }
                - { type: 'section', label: 'Optional' }
                - { property: 'funFact', type: 'textarea', css_class: 'js-markdown-input' }
                - isPublished

                - { type: 'group', css_class: 'col-sm-6', label: 'Studied by ...' }
                -
                    property: 'genusScientists'
                    type: 'text'
                    type_options:
                        mapped: false
                        attr: { class: 'js-genus-scientists-field' }

                -
                    type: 'group'
                    css_class: 'col-sm-6 new-row'
                    label: 'Identification'
                    icon: 'id-card-o'
                    help: 'For administrators'
                -
                    property: id
                    type_options: {disabled: true}
                -
                    property: 'slug'
                    help: 'unique auto-generated value'
                    type_options: { disabled: true }

            new:
                fields:
                - '-id'
                - '-slug'
        GenusHorde:
            class: AppBundle\Entity\Genus
            label: HORDE of Genuses
```

Perfect!

Now, we just need to make sure that Symfony loads this file. At the top of `config.yml`, just load another resource: `admin.yml`:

81 lines | app/config/config.yml

```
1   imports:
    ... lines 2 - 4
5       - { resource: admin.yml }
    ... lines 6 - 81
```

And that is it! When we refresh, everything still works!

Phew, we're done! EasyAdminBundle is *great*. But of course, depending on *how* custom you need things, you might end up overriding a *lot* of different parts. Many things can be done via configuration. But by using the tools that we've talked about, you can really override everything. Ultimately, customizing things will *still* be a *lot* faster than building all of this on your own.

All right guys, thank you *so* much for joining me! And a *huge* thanks to my co-author Andrew for doing all the *actual* hard work.

Ok, seeya next time!