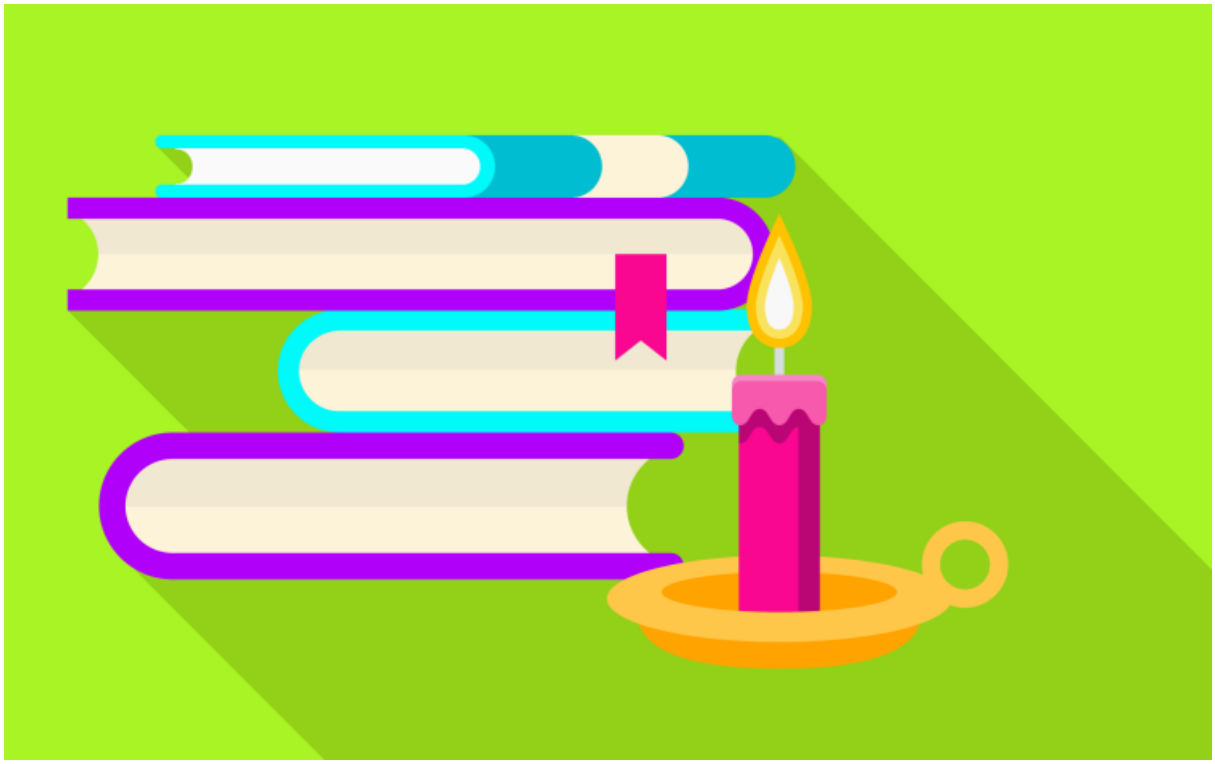
EasyAdmin! For an Awesomely Powerful Admin Area



With <3 from SymfonyCasts

Chapter 1: Installing EasyAdmin

Well hey friends! We are in for a *treat* with this tutorial! It's EasyAdmin: my favorite admin generator for Symfony. It just... gives you so many features out of the box. And it looks great! This shouldn't really be a surprise because its creator is the always-impressive Javier Eguiluz.

So let's have some fun and learn how to bend EasyAdmin to our will. Because getting a lot of features for free is great... as long as we can extend it to do crazy things when we need to.

Project Setup

To squeeze the most "easy" out of EasyAdmin, you should definitely code along with me. You probably know the drill: download the course code from this page and unzip it to find a `start/` directory with the same code that you see here. Check out the `README.md` file for all the setup goodies. I've already done all of these steps except for two.

For the first, find your terminal and run:

```
yarn install
```

I ran this already to save time... so I'll skip to compiling my assets with:

```
yarn watch
```

You can also run:

```
yarn dev-server
```

Which can do cool things like update your CSS without refreshing.

Perfect! For the *second* thing, open up another tab and run:

```
symfony serve -d
```

This fires up a local web server - using the Symfony binary - at https://127.0.0.1:8000. I'll be lazy by holding `Cmd` and clicking the link to pop open my browser. Say "hello" to... Cauldron Overflow! If you've been doing our Symfony 5 series, you're definitely familiar with this project. But, this is a *Symfony 6* project, not Symfony 5:

100 lines | composer.json

```
{
... lines 2 - 3
    "require": {
... lines 5 - 15
        "symfony/asset": "6.0.*",
        "symfony/console": "6.0.*",
        "symfony/dotenv": "6.0.*",
... line 19
        "symfony/framework-bundle": "6.0.*",
... line 21
        "symfony/runtime": "6.0.*",
        "symfony/security-bundle": "6.0.*",
        "symfony/stopwatch": "6.0.*",
        "symfony/twig-bundle": "6.0.*",
... line 26
        "symfony/yaml": "6.0.*",
... lines 28 - 29
    },
    "require-dev": {
... line 32
        "symfony/debug-bundle": "6.0.*",
... line 34
        "symfony/var-dumper": "6.0.*",
        "symfony/web-profiler-bundle": "6.0.*",
... line 37
    },
... lines 39 - 98
}
```

Oooo. If you *are* using Symfony 5, don't worry: very little will be different.

You don't need to worry too much about the majority of the code inside the project. The most important thing is probably our `src/Entity/` directory. Our site has questions, and each `Question` has a number of answers. Each `Question` belongs to a single `Topic` ... and then we have a `User` entity.

Our goal in *this* tutorial is to create a rich admin section that allows our admin users to manage *all* of this data.

Installing EasyAdmin

So let's get EasyAdmin installed! Find your terminal and run:

```
composer require admin
```

This is a Flex alias for `easycorp/easyadmin-bundle`. Notice that it downloads the shiny new version *4* of EasyAdmin, which only works with Symfony 6. So if you're using Symfony 5, run:

```
composer require admin:^3
```

to get version 3. Right now, version 4 and version 3 are identical, so you won't notice any differences. But going forward, new features will only be added to version 4.

Cool! Now that this is installed, what's next? Ship it!? Well, before we start deploying and celebrating our success... if we want to actually *see* something on our site, we're going to need a dashboard. Let's generate that next!

Chapter 2: Admin Dashboard

Run:

```
git status
```

Installing EasyAdmin didn't do anything fancy: it doesn't have a recipethat adds config files or a button that makes cute kittens appear. Darn. It just added itself and registered its bundle.So simply installing the bundle didn't give us any new routes or pages.

For example, if I try to go to `/admin`, we see "Route Not Found."That's because the first step after installing EasyAdmin is to create an admin dashboard: a sort of "landing page" for your admin.You'll typically have only one of these in your app, but you *can* have multiple, like for different admin user types.

And we don't even need to create this dashboard thingy by hand!Back at your terminal, run:

```
symfony console make:admin:dashboard
```

As a reminder, `symfony console` is exactly the same as running `php bin/console`. The only difference is that running `symfony console` allows the Docker environment variablesto be injected into this command. It typically makes no difference unless you're running a command that requires database access. So, in this case, `php bin/console` would work just fine.

I'll stick with `symfony console` throughout this tutorial. So say:

```
symfony console make:admin:dashboard
```

We'll call it `DashboardController`, generate it into `src/Controller/Admin` and... done! This created one new file: `src/Controller/Admin/DashboardController.php`. Let's go check it out!

When I open it...

31 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 2
namespace App\Controller\Admin;

use EasyCorp\Bundle\EasyAdminBundle\Config\Dashboard;
use EasyCorp\Bundle\EasyAdminBundle\Config\MenuItem;
use EasyCorp\Bundle\EasyAdminBundle\Controller\AbstractDashboardController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class DashboardController extends AbstractDashboardController
{
    #[Route('/admin', name: 'admin')]
    public function index(): Response
    {
        return parent::index();
    }

    public function configureDashboard(): Dashboard
    {
        return Dashboard::new()
            ->setTitle('EasyAdminBundle');
    }

    public function configureMenuItems(): iterable
    {
        yield MenuItem::linkToDashboard('Dashboard', 'fa fa-home');
        // yield MenuItem::linkToCrud('The Label', 'fas fa-list', EntityClass::class);
    }
}
```

Hmm. There's not much here yet. But one thing you might notice is that it has a *route* for `/admin`:

31 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 10
class DashboardController extends AbstractDashboardController
{
    #[Route('/admin', name: 'admin')]
    public function index(): Response
    {
... line 16
    }
... lines 18 - 29
}
```

So now, if we find our browser and go to `/admin` ... we *do* hit the admin dashboard!

Since version `4.0.3` of EasyAdmin, this welcome page looks a bit different! For example, it won't have the side menu that you see in the video. To see the links - and follow better with the tutorial - create a new dashboard template that will extend the base layout from EasyAdmin:

```
{# templates/admin/index.html.twig #}

{% extends '@EasyAdmin/page/content.html.twig' %}
```

Then, comment out the `return parent::index();` line in `DashboardController::index()` and instead render this template:

```
class DashboardController extends AbstractDashboardController
{
    #[Route('/admin', name: 'admin')]
    public function index(): Response
    {
        return $this->render('admin/index.html.twig');
    }
}
```

We'll talk much more later about how to use and design this dashboard page!

I want to point out a few important things. The first is that we *do* have a `/admin` route... and there's nothing fancy or "EasyAdmin" about it. This is just... how we create routes in Symfony. This is a PHP 8 *attribute* route, which you may or may not be familiar with. I've typically used annotations until now. But because I'm using PHP 8, I'll be using attributes insteadof annotations throughout the tutorial. Don't worry though! They work exactly the same. If you're still using PHP 7, you can use annotations just fine.

The second important thing is that `DashboardController` is just a normal controller. Though, it *does* extend `AbstractDashboardController`:

31 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Controller\AbstractDashboardController;
... lines 8 - 10
class DashboardController extends AbstractDashboardController
{
... lines 13 - 29
}
```

Hold `Cmd` or `Ctrl` and click to jump into that class.

This implements `DashboardControllerInterface`. So this *is* a normal controller, but by implementing this interface, EasyAdmin knows that we're inside the admin section... and boots up its engine. We'll learn *all* about what that means throughout the tutorial.

Most importantly, this class has a number of methods that we can overrideto configure what our dashboard looks like. We'll *also* be doing *that* throughout this tutorial.

Securing the Dashboard

And because this is just a normal route and controller, it *also* follows the normal security rules that we would expect. Right now, this means that *no* security is being applied. I mean, check it out: I'm not even logged in, but I*am* successfully on the admin dashboard!

> In Symfony 6.2, you can use the `#[IsGranted()]` attribute without installing SensioFrameworkExtraBundle. It's now part of the core!

So let's secure it! I'll also do this with an attribute. I already have SensioFrameworkExtraBundle installed, so I can say `#[IsGranted()]` and hit "tab" to auto-complete that. Let's require any user accessing this controller to have `ROLE_ADMIN` ... that's kind of a base admin role that all admin users have in my app:

33 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 7
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
... lines 9 - 11
class DashboardController extends AbstractDashboardController
{
    #[IsGranted('ROLE_ADMIN')]
... line 15
    public function index(): Response
    {
... line 18
    }
... lines 20 - 31
}
```

Now when we refresh... beautiful! We bounced back over to the login page!

To log in, open src/DataFixtures/AppFixtures.php :

67 lines | src/DataFixtures/AppFixtures.php

```
... lines 1 - 11
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // Load Users
        UserFactory::new()
            ->withAttributes([
                'email' => 'superadmin@example.com',
                'plainPassword' => 'adminpass',
            ])
            ->promoteRole('ROLE_SUPER_ADMIN')
            ->create();

        UserFactory::new()
            ->withAttributes([
                'email' => 'admin@example.com',
                'plainPassword' => 'adminpass',
            ])
            ->promoteRole('ROLE_ADMIN')
            ->create();

        UserFactory::new()
            ->withAttributes([
                'email' => 'moderatoradmin@example.com',
                'plainPassword' => 'adminpass',
            ])
            ->promoteRole('ROLE_MODERATOR')
            ->create();

        UserFactory::new()
            ->withAttributes([
                'email' => 'tisha@symfonycasts.com',
                'plainPassword' => 'tishapass',
                'firstName' => 'Tisha',
                'lastName' => 'The Cat',
                'avatar' => 'tisha.png',
            ])
            ->create();
... lines 50 - 64
    }
}
```

I have a bunch of dummy users in the database: there's a super admin, a normal admin, and then somebody known as a moderator. We'll talk more about these later when we get deeper into how to secure different parts of your admin for different roles.

Anyways, log in with `admin@example.com` ... password `adminpass`, and... beautiful! We're back to our dashboard!

Of course, if you want to, instead of using the `IsGranted` PHP attribute, you could also say `$this->denyAccessUnlessGranted()`. And you could *also* go to `config/packages/security.yaml` and, down at the bottom, add an `access_control` that protects the entire `/admin` section:

55 lines | config/packages/security.yaml

```
security:
... lines 2 - 36
    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
        - { path: ^/profile, roles: ROLE_USER }
... lines 42 - 55
```

Actually, adding this `access_control` is basically *required*: using only the `IsGranted` attribute is *not* enough. We'll learn why a bit later.

Configuring the Dashboard

So our dashboard is the "jumping off point" for our admin, but there's nothing particularly special here. The page has a title, some menu items, and a nice little user menu over here. Eventually, we'll render something cool on this page - like some stats and graphs - instead of this message from EasyAdmin. Oh, and all of this styling is done with Bootstrap 5 and FontAwesome. More on tweaking the design later.

Before we move on, let's see if we can customize the dashboard a little bit. One of the absolute *best* things about EasyAdmin is that all the config is done in PHP. Yay! It's usually done via methods in your controller. For example: want to configure the dashboard? There's a `configureDashboard()` method for that!

33 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 11
class DashboardController extends AbstractDashboardController
{
... lines 14 - 20
    public function configureDashboard(): Dashboard
    {
        return Dashboard::new()
            ->setTitle('EasyAdminBundle');
    }
... lines 26 - 31
}
```

We can change the title of the page to "Cauldron Overflow Admin":

33 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 11
class DashboardController extends AbstractDashboardController
{
... lines 14 - 20
    public function configureDashboard(): Dashboard
    {
        return Dashboard::new()
            ->setTitle('Cauldron Overflow Admin');
    }
... lines 26 - 31
}
```

When we refresh... we see "Cauldron Overflow Admin"! And there are a number of other methods...just look at the auto-complete from your editor. There are methods related to the favicon path...and something about the sidebar being minimized. That's referring to a nice feature where you can click on the separatorfor the sidebar to collapse or expand it.

The *main* part of the dashboard is really these menu items. And, we only have one right now. This is controlled by `configureMenuItems()` :

33 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 11
12  class DashboardController extends AbstractDashboardController
13  {
... lines 14 - 26
27      public function configureMenuItems(): iterable
28      {
29          yield MenuItem::linkToDashboard('Dashboard', 'fa fa-home');
30          // yield MenuItem::linkToCrud('The Label', 'fas fa-list', EntityClass::class);
31      }
32  }
```

Just to prove that we can, let's change the icon to `fa-dashboard` :

33 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 11
12  class DashboardController extends AbstractDashboardController
13  {
... lines 14 - 26
27      public function configureMenuItems(): iterable
28      {
29          yield MenuItem::linkToDashboard('Dashboard', 'fa fa-dashboard');
... line 30
31      }
32  }
```

This leverages the FontAwesome library. When we refresh, new icon!

So we can *definitely* do more with our dashboard, but that's enough for now. Because what we're *really* here for are the "CRUD controllers". These are the sections of our site where we will be able to create, readµpdate, and delete all of our entities. Let's get those going next!

Chapter 3: Hello CRUD Controller

The true reason to use EasyAdmin is for its CRUD controllers. Each CRUD controller will give us a rich set of pages to create, read, update, and delete a single entity. This is where EasyAdmin *shines*, and the next few minutes are going to be *critically* important to understand how EasyAdmin works. So, buckle up!

Generating the CRUD Controller

We have four entities. Let's generate a CRUD controller for `Question` first. Find your terminal and run:

```
symfony console make:admin:crud
```

As you can see, it recognizes our four entities. I'll hit 1 for `App\Entity\Question`, let this generate into the default directory... and with default namespace.

Sweet! This did exactly *one* thing: it created a new `QuestionCrudController.php` file. Let's... go open it up!

26 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 2
namespace App\Controller\Admin;

use App\Entity\Question;
use EasyCorp\Bundle\EasyAdminBundle\Controller\AbstractCrudController;

class QuestionCrudController extends AbstractCrudController
{
    public static function getEntityFqcn(): string
    {
        return Question::class;
    }

    /*
    public function configureFields(string $pageName): iterable
    {
        return [
            IdField::new('id'),
            TextField::new('title'),
            TextEditorField::new('description'),
        ];
    }
    */
}
```

Linking to the CRUD Controller

Cool. But before we look too deeply into this, head over to the admin page and refresh to see... absolutely no difference! We *do* have a new `QuestionCrudController`, but these CRUD controllers are *totally* useless until we link to them from a dashboard. So, back over in `DashboardController`, down at the bottom... `yield MenuItem` ... but instead of `linkToDashboard()`, there are a number of other things that we can link to. We want `linkToCrud()`. Pass this the label - so "Questions" - and some FontAwesome icon classes: `fa fa-question-circle`. Then, most importantly, pass the entity's class name: `Question::class`:

37 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 5
use App\Entity\Question;
... lines 7 - 15
class DashboardController extends AbstractDashboardController
{
... lines 18 - 30
    public function configureMenuItems(): iterable
    {
... line 33
        yield MenuItem::linkToCrud('Questions', 'fa fa-question-circle', Question::class);
    }
}
```

Behind the scenes, when we click this new link, EasyAdmin will recognize that there is only *one* CRUD controller for the entity - `QuestionCrudController` - and will know to use it. And yes, in theory, we *can* have multiple CRUD controllers for a single entity... and that's something we'll talk about later.

Okay, go refresh to reveal our new link, click and... whoa! This is *amazingly* cool! We have a slider for the `isApproved` field, which saves automatically. We also have a search bar on top... and sortable columns to help us find whatever we're looking for.

We can delete, edit... and the form even has a nice calendar widget. This is *loaded* with rich features out-of-the-box.

Generating All the CRUD Controllers

So let's repeat this for our other three controllers. Head back to your terminal and, once again, run:

```
symfony console make:admin:crud
```

This time generate a CRUD for `Answer` ... with the default stuff... one for `Topic` with the defaults... I'll clear my screen... and finally generate one for `User` .

Beautiful! The *only* thing this did was add three more CRUD controller classes. But to make those useful, we need to link to them. I'll paste 3 more links... then customize the label, font icons and class on each of them:

40 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 4
use App\Entity\Answer;
... line 6
use App\Entity\Topic;
use App\Entity\User;
... lines 9 - 15
class DashboardController extends AbstractDashboardController
{
... lines 18 - 30
    public function configureMenuItems(): iterable
    {
... lines 33 - 34
        yield MenuItem::linkToCrud('Answers', 'fas fa-comments', Answer::class);
        yield MenuItem::linkToCrud('Topics', 'fas fa-folder', Topic::class);
        yield MenuItem::linkToCrud('Users', 'fas fa-users', User::class);
    }
}
```

Super fast!

Let's go check it out! Refresh and... look! Simply by running that command four times, we now have four different fully-featured admin sections!

The Main configure() Methods of your CRUD Controller

I want to look a little deeper into *how* this is working behind the scenes. Go to `QuestionCrudController` and look at its base class:

26 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 5
use EasyCorp\Bundle\EasyAdminBundle\Controller\AbstractCrudController;

class QuestionCrudController extends AbstractCrudController
{
... lines 10 - 24
}
```

Hold `Cmd` or `Ctrl` to jump into `AbstractCrudController`. We saw earlier that our dashboard extends `AbstractDashboardController`. CRUD controllers extend `AbstractCrudController`.

Pretty much everything about how our CRUD controller works is going to be controlled by overriding the configure methods that you see inside of here. We'll learn about all of these as we go along. But on a high level, `configureCrud()` helps you configure things about the CRUD section as a whole, `configureAssets()` allows you to add custom CSS and JavaScript to the section, and `configureActions()` allows you to control the *actions* you want, where an action is a button or link. So, you can control whether or not you have delete, edit or index links on different pages. More on that later.

The last super important method is `configureFields()`, which controls the fields we see on both the index page *and* on the form. But don't worry about those too much yet. We'll master each method along the way.

Below this, super cool... we can see the actual code that executes for each page! The `index()` method is the *real* action for the index, or "list" page. `detail()` is an action that shows the details of a single item, and `edit()` is the edit form. I *love* that we can see the full code that runs all of this. It'll be *super* useful when we're figuring out how to extend things.

But... wait a second. If you scroll back up to the configure methods, a few of these look familiar. Some of these *also* exist in the dashboard base controller class. And it turns out, understanding *why* some methods live in both classes is the *key* to being able to make changes to your *entire* admin section *or* changes to just *one* CRUD section. Let's dive into that next.

Chapter 4: Global vs CRUD-Specific Configuration

The methods `configureAssets()`, `configureCrud()`, `configureActions()` and `configureFilters()` all live here inside of `AbstractCrudController`. And each gives us a way to control different parts of the CRUD section.

But, these methods *also* live inside of `AbstractDashboardController`. Here's `configureAssets()`, and then further down, we see the methods for, CRUD, actions and filters.

But... that doesn't make sense! The dashboard controller just renders... the dashboard page. And that page doesn't have any actions or any CRUD to configure. What's going on here?

One Route for All of your Admin

Click "Questions" and look at the URL. It starts with `/admin` and then has a bunch of query parameters. It turns out that *everything* in EasyAdmin is handled by a single giant route. It all runs through the `DashboardController` route - the `/admin` route that's above `index()`:

40 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 15
class DashboardController extends AbstractDashboardController
{
... line 18
    #[Route('/admin', name: 'admin')]
    public function index(): Response
    {
... line 22
    }
... lines 24 - 38
}
```

So when we go to `QuestionCrudController`, it's actually matching *this* route here with extra query parameters to say which CRUD controller and which action to run. You can see `crudController` and `crudAction` hiding in the URL. Yup, we're rendering `QuestionCrudController`, but in the *context* of `DashboardController`.

And when we go to this page, in order to get the CRUD config, EasyAdmin *first* calls `configureCrud()` on our *dashboard* controller. *Then* it calls `configureCrud()` on the specific CRUD controller, in this case, `QuestionCrudController`. This is *incredibly* powerful. It means that we can configure things inside of our dashboard - and have those apply to *every* section in our admin - or configure things inside of one specific CRUD controller to only change the behavior for that *one* section.

Understanding Pages and Actions

We can prove it! Go back to `AbstractDashboardController`. Look at `configureCrud()`. Every CRUD section has 4 pages. Hold `Cmd` or `Ctrl` and click to open this `Crud` class. Check out the constants on top. Every CRUD section has an index page - that's this - an edit page, a new page, and also a detail page. Each page can then have links and buttons to a set of *actions*. For example, on the index page, right now, we have an action for editing, an action for deleting... and also an action on top to add a *new* question. And, of course, this is all something we can control.

You can see how this is configured down in `configureActions()`. Because we're inside of the *dashboard* controller class, this is the default action configuration that applies to *every* CRUD section. You can see that, for the index page, it adds `NEW`, `EDIT` and `DELETE` actions. For the detail page, there's `EDIT`, `INDEX`, and `DELETE`. And if you're on the edit page, you have the actions `SAVE_AND_RETURN` and `SAVE_AND_CONTINUE`.

Adding an Action Globally

If you look closely, you'll notice that while we *do* have a detail page, nobody links to it! We don't see an action called `DETAIL` on any of these pages. So the page exists, but it's not really used out-of-the-box. Let's change that!

Go back to `DashboardController`. It doesn't matter where, but I'll go down to the bottom, go to "Code"->"Generate..." or `Cmd` +

N on a Mac - click "Override Methods" and select configureActions() :

49 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 9
use EasyCorp\Bundle\EasyAdminBundle\Config\Actions;
... lines 11 - 18
class DashboardController extends AbstractDashboardController
{
... lines 21 - 42
    public function configureActions(): Actions
    {
        return parent::configureActions()
... line 46
    }
}
```

We *do* want to call the parent method so that it can create the Actions object and set up all of those default actions for us.Let's add a link to the "detail" page from the "index" page. In EasyAdmin language, this means we want to add a detail*action* to the index page. Do that by saying ->add() , passing the page name - Crud::PAGE_INDEX - and then the action: Action::DETAIL :

49 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 8
use EasyCorp\Bundle\EasyAdminBundle\Config\Action;
... line 10
use EasyCorp\Bundle\EasyAdminBundle\Config\Crud;
... lines 12 - 18
class DashboardController extends AbstractDashboardController
{
... lines 21 - 42
    public function configureActions(): Actions
    {
        return parent::configureActions()
            ->add(Crud::PAGE_INDEX, Action::DETAIL);
    }
}
```

Thanks to this, when we refresh the index page of QuestionCrudController ... we have a "Show" link that goes to the DETAIL action! And you'll see this on *every* section of our admin! Yup, we just modified *every* CRUD controller in the system!

Overriding Actions Config for One CRUD

But, since the Topic entity is so simple, let's disable the DETAIL action for *just* this section. To do that, open up TopicCrudController , and, just like before, go to "Code"->"Generate..." - or Cmd + N on a Mac - hit "Override Methods" and select configureActions() :

34 lines | src/Controller/Admin/TopicCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Config\Actions;
... lines 8 - 9
class TopicCrudController extends AbstractCrudController
{
... lines 12 - 16
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
... line 20
    }
... lines 22 - 32
}
```

By the time this method is called, it will pass us the Actions object that was already set up by our dashboard.So it will already have the detail action enabled for the index page. But *now*, we can change that by saying ->disable(Action::DETAIL) :

34 lines | src/Controller/Admin/TopicCrudController.php

```
... lines 1 - 5
use EasyCorp\Bundle\EasyAdminBundle\Config\Action;
... lines 7 - 9
class TopicCrudController extends AbstractCrudController
{
... lines 12 - 16
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
            ->disable(Action::DETAIL);
    }
... lines 22 - 32
}
```

We'll talk more about the actions configuration later. But these are the main things that you can do inside of them: add a new action to a page, *or* completely disable an action. Now, when we refresh, our `DETAIL` action is gone! But if we go to any other section, it's *still* there.

The big takeaway is that everything is processed *through* our `DashboardController`, which means that we can configure things on a dashboard-level, which will apply to all of our CRUDs, *or* we can configure things for one *specific* CRUD.

Re-Visiting Securing your Admin

The fact that all of the CRUD controllers go through this `/admin` URL has one other effect related to security. It means that all of our controllers *are* already secure. That's thanks to our `access_control`.

Remember, back in `config/packages/security.yaml`, we added an `access_control` that said if the URL starts with `/admin`, require `ROLE_ADMIN`:

55 lines | config/packages/security.yaml

```
security:
... lines 2 - 38
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
... lines 41 - 55
```

This means that without doing *anything* else, *every* CRUD controller and action in our admin already requires `ROLE_ADMIN`. We'll talk more later about how to secure different admin controllers with different roles... but at the very least, you need to have `ROLE_ADMIN` to get anywhere, which is awesome.

But one important point: adding this `access_control` *was* necessary. Why? The `index()` action in our dashboard is what holds the *one* route. When we go to a CRUD controller, like this, it *does* match this route.... but EasyAdmin does something crazy. Instead of allowing Symfony to call this controller, it sees this `crudController` query parameter and magically *switches* the controller to be the *real* controller. In this case, it changes it to `QuestionCrudController::index()`.

You can see this down on the web debug toolbar. If you hover over "@admin", this tells you that the matched route name was `admin`. So, yes, the route *is* matching the main dashboard route. But the controller is `QuestionCrudController::index()`.

This means that the method in your *CRUD* controller is what Symfony ultimately executes. In this case, it's the `index()` method in this `AbstractCrudController` ... down here. *This* is the *real* controller for the page.

Why does that matter? First, it's nice to know that, even with all the EasyAdmin coolness and magic, at the end of the day, the actions in our controller are *real* actions that are called like any normal action. And second, this is important for security. Because if we had *only* put the `IsGranted` above `index()` and *not* added the `access_control`, that would *not* have been enough. Why? Because this `isGranted` attribute is *only* enforced when you execute *this* action. So, when we go to the dashboard page.

Anyways, if some of this is still a bit fuzzy, no worries! This was a blast of EasyAdmin theory that'll help us understand things better as we dig and experiment.

Next, before we go deeper into our CRUD controllers, let's mess around a bit more with our dashboard by adding some custom

links to our admin menu and user menu.

Chapter 5: Controlling the Dashboard Menu

There are two things that we can do from our `DashboardController`. The first is to configure the dashboard itself, which is mostly just the title, menu links, and also controlling the user menu. The second is that we can configure things that *affect* the CRUD controllers. And we saw an example of that with `configureActions()` where we globally added a `DETAIL` action to every index page.

That was an *awesome* start, but let's look a bit more at some ways that we can configure the dashboard itself.

Linking to the Frontend

The `configureMenuItems()` method, as we already know, can link to a dashboard and another CRUD section. Now let's add a link to the homepage. Say `yield MenuItem::linkToRoute()`, passing "Homepage", an icon... and then the route name and optionally route parameters. The name of our homepage route is `app_homepage`:

50 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 18
19 class DashboardController extends AbstractDashboardController
20 {
... lines 21 - 33
34     public function configureMenuItems(): iterable
35     {
... lines 36 - 40
41         yield MenuItem::linkToRoute('Homepage', 'fas fa-home', 'app_homepage');
42     }
... lines 43 - 48
49 }
```

Cool. Head over, refresh and... yea! We now have a nice homepage link on every page. And if we click, it works!

linkToRoute() vs linkToUrl()

But whoa... check out the URL! That does *not* look like the homepage. The URL starts with `/admin` and then has a bunch of query parameters. Yup, it's rendering our homepage controller *through* the admin dashboard... much like how our CRUD controllers render through the dashboard. This works, but it's *not* what we intended.

So let's try again. The `linkToRoute()` method really means:

> Link to somewhere... but run that controller *through* the admin section.

This can be useful if you have a custom controller but want to leverage some of the EasyAdmin tools from inside it. If you just want to link to a page, use `linkToUrl()` instead. This will have the same label and icon... but instead of passing the route name, say `$this->generateUrl()` and pass `app_homepage`:

50 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 18
19 class DashboardController extends AbstractDashboardController
20 {
... lines 21 - 33
34     public function configureMenuItems(): iterable
35     {
... lines 36 - 40
41         yield MenuItem::linkToUrl('Homepage', 'fas fa-home', $this->generateUrl('app_homepage'));
42     }
... lines 43 - 48
49 }
```

Go back to the admin page, refresh...click and... much better!

Linking to the Admin from the Frontend

But what about a link *back* to the admin page, like up here in the header?For that, open `templates/base.html.twig` and scroll down to the navbar... here it is.

There's nothing here yet. Add `<li class="navbar-nav></li>` and a few other classes. Inside, add an `<a>` with `href=""` set to `path()`. To link to the admin section, there's nothing special.Our `DashboardController` has a *real* route, and its name is `admin`:

50 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 18
class DashboardController extends AbstractDashboardController
{
... line 21
    #[Route('/admin', name: 'admin')]
    public function index(): Response
    {
... line 25
    }
... lines 27 - 48
}
```

So we can just link to that: `admin`. Give our anchor a class to make it look good...and I'll say "Admin":

84 lines | templates/base.html.twig

```
... line 1
<html lang="en">
... lines 3 - 14
    <body>
        <nav
            class="navbar navbar-expand-lg navbar-light bg-light px-1"
            {{ is_granted('ROLE_PREVIOUS_ADMIN') ? 'style="background-color: red !important"' }}
        >
            <div class="container-fluid">
... lines 21 - 30
                <div class="collapse navbar-collapse" id="navbar-collapsable">
                    <ul class="navbar-nav me-auto mb-2 mb-lg-0">
... line 33
                        <li class="nav-item">
                            <a class="nav-link" href="{{ path('admin') }}">Admin</a>
                        </li>
... line 37
                    </ul>
... lines 39 - 72
                </div>
            </div>
        </nav>
... lines 76 - 81
    </body>
</html>
```

And we really only want to render this link if we have `ROLE_ADMIN`. So `{% if isGranted('ROLE_ADMIN') %}` ... then `{% endif %}` on the other side:

84 lines | templates/base.html.twig

```
... line 1
<html lang="en">
... lines 3 - 14
    <body>
        <nav
            class="navbar navbar-expand-lg navbar-light bg-light px-1"
            {{ is_granted('ROLE_PREVIOUS_ADMIN') ? 'style="background-color: red !important"' }}
        >
            <div class="container-fluid">
... lines 21 - 30
                <div class="collapse navbar-collapse" id="navbar-collapsable">
                    <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                        {% if is_granted('ROLE_ADMIN') %}
                            <li class="nav-item">
                                <a class="nav-link" href="{{ path('admin') }}">Admin</a>
                            </li>
                        {% endif %}
                    </ul>
... lines 39 - 72
                </div>
            </div>
        </nav>
... lines 76 - 81
    </body>
</html>
```

Beautiful! Let's test it. Refresh... there's the link and... we're right back on our admin section!

Customizing the User Menu

One other thing that the dashboard controls is this nice little user menu up here. It shows who you're logged in as, an avatar that doesn't work yet, and a "Sign out" link. In our system, users actually *do* have avatars on the frontend. You can see this: this is an avatar for a user... and my user's avatar shows up in the upper right. *But* EasyAdmin doesn't *know* that our users have avatars. We need to tell it.

Back in `DashboardController` ... it doesn't matter where... go to "Code"->"Generate..." or `Cmd + N` on a Mac, click "Override Methods", and select `configureUserMenu()` :

60 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 13
use EasyCorp\Bundle\EasyAdminBundle\Config\UserMenu;
... lines 15 - 18
use Symfony\Component\Security\Core\User\UserInterface;

class DashboardController extends AbstractDashboardController
{
... lines 23 - 48
    public function configureUserMenu(UserInterface $user): UserMenu
    {
        return parent::configureUserMenu($user);
    }
... lines 53 - 58
}
```

This has several methods on it. We can add other menu items (we'll do that in second), set the avatar URL, and a few other things. I'll say `setAvatarUrl()` and pass it `$user->getAvatarUrl()` : this is a custom method on our `User` class:

61 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 20
class DashboardController extends AbstractDashboardController
{
... lines 23 - 48
    public function configureUserMenu(UserInterface $user): UserMenu
    {
        return parent::configureUserMenu($user)
            ->setAvatarUrl($user->getAvatarUrl());
    }
... lines 54 - 59
}
```

Notice that I'm not getting auto-completion on the method.That's because PhpStorm doesn't know that this is our *custom* `User` class. So if you want to code defensively, add `if (!$user instanceof User)` , then `throw new \Exception('Wrong user');` .

```
use App\Entity\User;

class DashboardController extends AbstractDashboardController
{
    public function configureUserMenu(UserInterface $user): UserMenu
    {
        if (!$user instanceof User) {
            throw new \Exception('Wrong user');
        }

        return parent::configureUserMenu($user)
            ->setAvatarUrl($user->getAvatarUrl());
    }
}
```

Or you can just add a PHPDoc instead that will help PhpStorm to show more methods for autocompletion:

61 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 7
use App\Entity\User;
... lines 9 - 18
use Symfony\Component\Security\Core\User\UserInterface;

class DashboardController extends AbstractDashboardController
{
... lines 23 - 45
    /**
     * @param UserInterface|User $user
     */
    public function configureUserMenu(UserInterface $user): UserMenu
    {
        return parent::configureUserMenu($user)
            ->setAvatarUrl($user->getAvatarUrl());
    }
... lines 54 - 59
}
```

That won't ever happen, but now if I retype `$user->getAvatarUrl()` ... that fixes it!

And when we refresh... perfect! We have an avatar!

Adding a Link to the User Menu

The last thing I want to add is a link on the user menu that goes to my profile.We noticed before that another method you can call is `setMenuItems()` ... where you pass it in array of `MenuItem` objects. These items are the *same* ones that we've been

building in `configureMenuItems()`. So we can say, for example, `MenuItem::linkToUrl` with "My Profile"... some icons, and then `$this->generateUrl()`. The name of the route for my profile page is `app_profile_show`:

64 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 20
class DashboardController extends AbstractDashboardController
{
... lines 23 - 48
    public function configureUserMenu(UserInterface $user): UserMenu
    {
        return parent::configureUserMenu($user)
... line 52
            ->addMenuItems([
                MenuItem::linkToUrl('My Profile', 'fas fa-user', $this->generateUrl('app_profile_show'))
            ]);
    }
... lines 57 - 62
}
```

That's it! Refresh and... new link! Click and... that works too!

So there's nothing *too* complicated here: we can very easily control *all* of the menus in the admin.

So next, let's talk about *assets* inside of EasyAdmin. This is how we can add custom CSS and custom JavaScript to *any* section, including assets that are processed through Webpack Encore.

Chapter 6: Assets: Custom CSS and JS

The EasyAdmin interface looks pretty great out of the box.But what if we want to customize the way something looks?For example, if I want to change the background on the sidebar. How can we do that?

This type of stuff can be controlled via the `configureAssets()` method. As a reminder, this is one of those methods that exists inside both our dashboard controller *and* each individual CRUD controller. So we can control assets on a global level *or* for just one section.

Let's make our change globally so that we can change the color of the sidebar on every page.

Hello configureAssets()

Anywhere inside of `DashboardController`, go back to the "Code"->"Generate..." menu,select "Override Methods" and override `configureAssets()`:

70 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 10
use EasyCorp\Bundle\EasyAdminBundle\Config\Assets;
... lines 12 - 21
class DashboardController extends AbstractDashboardController
{
... lines 24 - 64
    public function configureAssets(): Assets
    {
        return parent::configureAssets();
    }
}
```

This has a lot of cool methods.There are some simple ones like `->addCssFile()`. If you said `->addCssFile('foo.css')`, that will include a `link` tag to `/foo.css`. As long as we have `foo.css` inside of our `public/` directory, that would work.

The same thing goes for `->addJsFile()`. And you can also `->addHtmlContentToBody()` or `->addHtmlContentToHead()`. There are *tons* of interesting methods!

Creating a Custom Admin Encore Entry

Our application uses Webpack Encore. Go check out the `webpack.config.js` file: it's pretty standard. We have just one entry called `app`:

76 lines | webpack.config.js

```
... lines 1 - 8
Encore
... lines 10 - 16
    /*
     * ENTRY CONFIG
     *
     * Each entry will result in one JavaScript file (e.g. app.js)
     * and one CSS file (e.g. app.css) if your JavaScript imports CSS.
     */
    .addEntry('app', './assets/app.js')
... lines 24 - 72
;
... lines 74 - 76
```

It's responsible for loading all of the JavaScript *and* CSS:

16 lines | assets/app.js

```
/*
 * Welcome to your app's main JavaScript file!
 *
 * We recommend including the built version of this JavaScript file
 * (and its CSS file) in your base layout (base.html.twig).
 */

// any CSS you import will output into a single css file (app.css in this case)
import './styles/app.css';

// start the Stimulus application
import './bootstrap';

// activates collapse functionality
import { Collapse } from 'bootstrap';
```

and we include this entry on our *frontend* to get everything looking and working well.

You probably noticed that, in `configureAssets()`, there's an `addWebpackEncoreEntry()` method. If we said `app` here, that would pull in the CSS and JavaScript from our `app` entry. *But*.... that makes things look a little crazy... because we do *not* want *all* of our frontend styles and JavaScript to show up in the admin section. Nope, we just want to be able to add *a little bit* of new stuff.

So here's what we'll do instead. Inside the `assets/styles/` directory, create an entirely new file called `admin.css`. This will be our CSS solely for styling the admin section. And just to see if things are working, I'll add a very lovely body background of "lightcyan":

4 lines | assets/styles/admin.css

```
body {
    background: lightcyan;
}
```

Fancy!

Over in `webpack.config.js`, add a second entry for *just* the admin. But, right now, since we only have a CSS file (we don't need JavaScript), I'll say `.addStyleEntry()` ... and point it to `./assets/styles/admin.css`. I should also change `app` to `admin` ... but I'll catch that in a minute:

77 lines | webpack.config.js

```
... lines 1 - 8
Encore
... lines 10 - 23
    .addStyleEntry('admin', './assets/styles/admin.css')
... lines 25 - 73
;
... lines 75 - 77
```

Because we just modified our webpack file, we need to go over to our terminal, find where we're running encore, hit `Ctrl` + `C`, and then rerun it:

```
yarn watch
```

And... it exploded! That's from my mistake! I need to give my entry a unique name. Change `app` to `admin`:

77 lines | webpack.config.js

```
... lines 1 - 8
Encore
... lines 10 - 23
    .addStyleEntry('admin', './assets/styles/admin.css')
... lines 25 - 73
;
... lines 75 - 77
```

Run it again, and... beautiful!

In addition to the original stuff, you can see that it also dumped an `admin.css` file. Thanks to this, over in our `DashboardController`, say `->addWebpackEncoreEntry('admin')`:

71 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 21
class DashboardController extends AbstractDashboardController
{
... lines 24 - 64
    public function configureAssets(): Assets
    {
        return parent::configureAssets()
            ->addWebpackEncoreEntry('admin');
    }
}
```

Refresh and... it works! That's a... well... interesting-looking page.

If you View the page source, you can see how this works. There's really nothing special. The `app.css` file gives us all of the EasyAdmin styling that we've been enjoying... and then *here* is our new `admin.css` file.

CSS Properties

At this point, we're dangerous! We can add whatever CSS we want to the new `admin.css` file and it will override *any* of the EasyAdmin styles. Cool! But EasyAdmin makes it even easier than that!

Inspect the element on the sidebar. The goal is to change the sidebar background. Find the actual element with the `sidebar` class. If you look over at the styles on the right... I'll make this a little bit bigger... you can see that the `.sidebar` class has a `background` style. But instead of it being set to a color, it's set to this `var(--sidebar-bg)` thing. If you hover over it, apparently, this is equal to `#f8fafc`.

If you haven't seen this before, this is a CSS property. It has nothing to do with EasyAdmin or Symfony. In CSS, you can create variables (called "CSS properties") and reference them somewhere else. EasyAdmin, apparently, created a `--sidebar-bg` variable and is referencing it here. So, instead of trying to override the background of `.sidebar` - which we *could* do - we can override this CSS property and it will have the same effect.

How? Let's cheat a little bit by digging deep into EasyAdmin itself.

Open `vendor/easycorp/easyadmin-bundle/assets/css/easyadmin-theme/`. Inside, there's a file called `variables-theme.scss`. *This* is where all of these CSS properties are defined. And there's *tons* of stuff here, for font sizes, different widths, and... `--sidebar-bg`! This `--sidebar-bg` variable, or property, is apparently set to *another* variable via the `var` syntax. You'll find *that* variable in another file called `./color-palette.scss` ... which is right here. These are SCSS files, but this CSS property system has *nothing* to do with Sass. This is a *pure* CSS feature.

There's a lot here, but if you follow the logic, `--sidebar-bg` is set to `--gray-50` ... then *all* the way at the bottom, `--gray-50` is set to `--blue-gray-50` ... then *that*... if we keep looking... yes! It's set to the color we expected!

This is a great way to learn what these values are, how they relate to one another and how to override them. Copy the `--sidebar-bg` syntax.

The way you define CSS variables is typically under this `:root` pseudo-selector. We're going to do the same thing.

In our CSS file, remove the `body`, add `:root` and then paste. And while it's *totally* legal to reference CSS properties from here, let's replace that with a normal hex color:

2 lines | assets/styles/admin.css

```
1  :root { --sidebar-bg: #deebff; }
```

Let's try it! Watch the sidebar closely... the change is subtle. Refresh and... it changed! To prove it, if you find the `--sidebar-bg` on the styles and hover... that property *is* now set to `#deebff`. It's subtle, but it *is* loading the correct color!

So we just customized the assets globally for our entire admin section. But we *could* override `configureAssets()` in a specific CRUD controller to make changes that *only* apply to that section.

Next, let's start digging into what is quite possibly the *most* important part of configuring EasyAdmin: Fields. These control which fields show up on the index page, as well as the form pages.

Chapter 7: Configuring Fields

Open up the "Users" section. EasyAdmin has a concept of *fields*. A field controls how a property is displayed on the index and detail pages, but *also* how it renders inside of a form. So the field *completely* defines the property inside the admin. By default, EasyAdmin just... guesses which fields to include. But *usually* you'll want to control this. How? Via the `configureFields()` method in the CRUD controller.

In this case, open `UserCrudController.php` ... and you can see that it already has a commented-out `configureFields()` method:

26 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 7
class UserCrudController extends AbstractCrudController
{
... lines 10 - 14
    /*
    public function configureFields(string $pageName): iterable
    {
        return [
            IdField::new('id'),
            TextField::new('title'),
            TextEditorField::new('description'),
        ];
    }
    */
}
```

Go ahead and uncomment that.

Notice that you can either return an `array` or an `iterable`. I usually return an `iterable` by saying `yield Field::new()` and passing the property name, like `id`:

21 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\Field;

class UserCrudController extends AbstractCrudController
{
... lines 11 - 15
    public function configureFields(string $pageName): iterable
    {
        yield Field::new('id');
    }
}
```

When I refresh... we have "ID" and nothing else.

Field Types

So EasyAdmin has *many* different *types* of fields, like text fields, boolean fields, and association fields... and it does its best to guess which type to use. In this case, you can't really see it, but when we said `id`, it guessed that this is an `IdField`. Instead of just saying `Field::new()` and letting it guess, I often prefer being explicit: `IdField::new()`:

22 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 7
use EasyCorp\Bundle\EasyAdminBundle\Field\IdField;

class UserCrudController extends AbstractCrudController
{
... lines 12 - 16
    public function configureFields(string $pageName): iterable
    {
        yield IdField::new('id');
    }
}
```

Watch: when we refresh... that makes absolutely no difference! It was *already* guessing that this was an `IdField`.

Cool! So how do we figure out what all of the field types are? Documentation is the most obvious way. If you look on the web debug toolbar, there's a little EasyAdmin icon.

> If you're using EasyAdmin 4.4.2 or later, you won't find an EasyAdmin icon on the Web Debug Toolbar. Instead, click on any link on the toolbar to get to the Profiler, then look for the "EasyAdmin" section near the bottom of the left sidebar.

Click into that... to see some basic info about the page... *with* a handy link to the documentation. Open that up. It has a "Field Types" section down a ways. Yup, there's your big list of all the different field types inside of EasyAdmin.

Or, if you want to go rogue, you find this directly in the source code. Check out `vendor/easycorp/easyadmin-bundle/src/Field`. *Here* is the directory that holds *all* the different possible field types.

Back in our CRUD controller, let's add a few more fields.

If you look in the `User` entity, you can see `$id`, `$email`, `$roles`, `$password`, `$enabled`, `$firstName`, `$lastName`, `$avatar` ... and then a couple of association fields:

283 lines | src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    use TimestampableEntity;

    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id;

    #[ORM\Column(length: 180, unique: true)]
    private ?string $email;

    #[ORM\Column(type: Types::JSON)]
    private array $roles = [];

    /**
     * The hashed password
     */
    #[ORM\Column]
    private ?string $password;

    /**
     * The plain non-persisted password
     */
    private ?string $plainPassword;

    #[ORM\Column]
    private bool $enabled = true;

    #[ORM\Column]
    private ?string $firstName;

    #[ORM\Column]
    private ?string $lastName;

    #[ORM\Column(nullable: true)]
    private ?string $avatar;

    #[ORM\OneToMany('askedBy', Question::class)]
    private Collection $questions;

    #[ORM\OneToMany('answeredBy', Answer::class)]
    private Collection $answers;
... lines 59 - 281
}
```

We won't need to manage *all* of these in the admin, but we*will* want most of them.

Add `yield TextField::new('firstName')` ... repeat that for `$lastName` ... and then for the `$enabled` field, let's `yield BooleanField::new('enabled')`. We also have a `$createdAt` field... so `yield DateField::new('createdAt')`:

30 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\BooleanField;
use EasyCorp\Bundle\EasyAdminBundle\Field\DateField;
... lines 9 - 10
use EasyCorp\Bundle\EasyAdminBundle\Field\TextField;

class UserCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... line 22
        yield TextField::new('email');
        yield TextField::new('firstName');
        yield TextField::new('lastName');
        yield BooleanField::new('enabled');
        yield DateField::new('createdAt');
    }
}
```

So I'm just listing the same properties that we see in the entity.Well, we don't see `$createdAt` ... but that's only because it lives inside of the `TimestampableEntity` trait:

283 lines | src/Entity/User.php

```
... lines 1 - 9
use Gedmo\Timestampable\Traits\TimestampableEntity;
... lines 11 - 15
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    use TimestampableEntity;
... lines 19 - 281
}
```

Anyways, with just this config, if we move over and refresh...beautiful! The text fields render normal text, the `DateField` knows how to print dates and the `BooleanField` gives us this nice little switch!

Using "Pseudo Properties"

As a challenge, instead of rendering "First Name" and "Last Name" columns,could we combine them into a single "Full Name" field? Let's try it!

I'll say `yield TextField::new('fullName')` :

29 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 12
class UserCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 23
        yield TextField::new('fullName');
... lines 25 - 26
    }
}
```

This is *not* a real property. If you open `User` , there is *no* `$fullName` property. *But*, I do have a `getFullName()` method:

283 lines | src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
... lines 18 - 194
    public function getFullName(): ?string
    {
        return $this->firstName.' '.$this->lastName;
    }
... lines 199 - 281
}
```

So the question is: is it smart enough - because the field is called `fullName` - to call the `getFullName()` method?

Let's find out. I bet you can guess the answer. Yup! That works!

Behind the scenes, EasyAdmin uses the PropertyAccess Component from Symfony. It's the same component that's used inside of the form system... and it's *really* good at reading properties by leveraging their getter method.

Field Options

Back in `configureFields()`, I forgot to add an "email" field. So, `yield TextField::new('email')`:

29 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 12
class UserCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... line 22
        yield TextField::new('email');
... lines 24 - 26
    }
}
```

And... no surprise, it renders correctly. *But*, this is a case where there's actually a more *specific* field for this: `EmailField`:

30 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... line 23
        yield EmailField::new('email');
... lines 25 - 27
    }
}
```

The only difference is that it renders with a link to the email. *And*, when you look at the form, it will now be rendering as an `<input type="email">`.

The *real* power of fields is that each has a *ton* of options. Some field options are shared by *all* field types. For example, you can call `->addCssClass()` on any field to add a CSS class to it. That's super handy. But *other* options are specific to the field *type* itself. For example, `BooleanField` has a `->renderAsSwitch()` method... and we can pass this `false`:

31 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 25
        yield BooleanField::new('enabled')
            ->renderAsSwitch(false);
... line 28
    }
}
```

Now, instead of rendering this cute switch, it just says "YES". This... is probably a good idea anyways... because it was a bit *too* easy to accidentally disable a user before this.

So... this is great! We can control which fields are displayed *and* we know that there are methods we can call on each field object to configure its behavior. But remember, fields control both how things are rendered on the index and detail pages *and* how they're rendered on the *form*. Right now, if we go to the form... yup! That's what I expected: these are the five fields that we've configured.

It's not perfect, though. I *do* like having an "ID" column on my index page, but I do *not* like having an "ID" field in my form.

So next, let's learn how to *only* show certain fields on certain pages. We'll also learn a few more tricks for configuring them.

Chapter 8: Fields on some Pages, not Others

As we discussed earlier, `configureFields()` controls how each field is rendered on both the list page *and* the form pages. That leaves us with a situation that... isn't exactly "ideal". For example, we don't want an ID field on our form. But I *do* like having it on the index page!

To fix this, there are a bunch of useful methods on these field classes that we can utilize. For instance, we can call `->onlyOnIndex()`:

32 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
        yield IdField::new('id')
            ->onlyOnIndex();
... lines 25 - 29
    }
}
```

And... just like that, it's gone from the form page, but we still have it on the index page. As you're playing with these methods, I invite you to be curious: dive in and check out the code behind the scenes. It's a *great* way to learn more about how EasyAdmin works on a deeper level.

Methods like `->onlyOnIndex()` give us a lot of control. But also notice that `configureFields()` is passed the `$pageName`, which will be a string like `index`, `detail`, or `edit`. So in the end, you can always just put `if` statements inside of this method and conditionally yield - or *don't* yield - different fields.

Hiding on the Form

The other problem on our form is that we have this `fullName` field. In the database, we have `firstName` and `lastName` fields. It *is* kind of nice to render them as "Full Name" on the index page. But ultimately, when we go to the form, we really need *separate* `firstName` and `lastName` fields.

And, at the moment, this doesn't even work! If I change something on the form and submit... error! It says:

> Could not determine access type for property `fullName` ...

This is because, inside of our `User` class, we have a `getFullName()` method, but we do *not* have a `setFullName()` method (and I don't really want one). The *point* is that, over inside `configureFields()`, we need to change `fullName` to render `->onlyOnIndex()`:

33 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 25
        yield TextField::new('fullName')
            ->onlyOnIndex();
... lines 28 - 30
    }
}
```

Now we'll have "Full Name" on our index, but we won't have one on the form.

And actually, instead of `->onlyOnIndex()`, we can use `->hideOnForm()`:

33 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 25
        yield TextField::new('fullName')
            ->hideOnForm();
... lines 28 - 30
    }
}
```

What's the difference? Using `->hideOnForm()` still allows "Full Name" to show on the detail page. If I go back to "Users" and click "Show"... there it is!

Now that "Full Name" is gone from the form, let's put "First Name" and "Last Name" *back*. So, `yield Field::new('firstName')` ... copy this, paste, and replace `firstName` with `lastName`:

35 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 27
        yield Field::new('firstName');
        yield Field::new('lastName');
... lines 30 - 32
    }
}
```

If we refresh... looks good! Over on the list page... looks weird! We *don't* want those here.

But now, we know what to do. There's a nice method for this: `->onlyOnForms()`. Copy that, repeat it for `lastName`:

37 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 27
        yield Field::new('firstName')
            ->onlyOnForms();
        yield Field::new('lastName')
            ->onlyOnForms();
... lines 32 - 34
    }
}
```

And now... perfect!

Finally, let's do something similar for "Created At". I like having this on the list, but I do *not* like having it inside the form because it should be set automatically. So, down here, add `->hideOnForm()`:

38 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
class UserCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 33
        yield DateField::new('createdAt')
            ->hideOnForm();
    }
}
```

Beautiful!

Next, I want to dive a bit further into fields.We're going to take one of these fields and configure its*form type* in a different way. As we do, we're going to accidentally learnabout an important concept called field configurators.

Chapter 9: Deep Field Configuration

One other property that we have inside of `User` is `$roles`, which actually stores an *array* of the roles this user should have:

283 lines | src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
... lines 18 - 27
    #[ORM\Column(type: Types::JSON)]
    private array $roles = [];
... lines 30 - 281
}
```

That's *probably* a good thing to include on our admin page. And fortunately, EasyAdmin has an `ArrayField`!

ArrayField

Check it out! Say `yield ArrayField::new('roles')`:

40 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\ArrayField;
... lines 8 - 14
class UserCrudController extends AbstractCrudController
{
... lines 17 - 21
    public function configureFields(string $pageName): iterable
    {
... lines 24 - 36
        yield ArrayField::new('roles');
    }
}
```

And then head back to your browser. Over on the index page... nice! It renders as a comma-separated list. And on the "Edit" page... oh, that's really cool! It added a nice widget for adding and removing roles!

Adding Help Text to Fields

The only tricky part might be remembering *which* roles are available. Right now, you have to type each in manually. We can at least help our admins by going back to our array field and implementing a method called `->setHelp()`. Add a message that includes the available roles:

41 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\ArrayField;
... lines 8 - 14
class UserCrudController extends AbstractCrudController
{
... lines 17 - 21
    public function configureFields(string $pageName): iterable
    {
... lines 24 - 36
        yield ArrayField::new('roles')
            ->setHelp('Available roles: ROLE_SUPER_ADMIN, ROLE_ADMIN, ROLE_MODERATOR, ROLE_USER');
    }
}
```

Now when we refresh... much better!

->setFormType() and ->setFormTypeOptions()

But, hmm. Now that I see this, it might look *even* better if we had check boxes. So let's see if we can *change* the `ArrayField` to display check boxes. Hold `Cmd` and open this core class.

This is *really* interesting, because you can actually *see* how the field is configured inside of its `new()` method. It sets the template name (we'll talk about templates later), but it *also* sets the form type. Behind the scenes, the `ArrayField` uses a `CollectionType`. If you're familiar with the Symfony Form Component, you know that, to render check boxes, you need the `ChoiceType`. I wonder if we can *use* `ArrayField` ... but override its form type to be `ChoiceType`.

Let's... give it a try!

First, above this, add `$roles = []` and list our roles. Then, down here, after `->setHelp()`, one of the methods we can call is `->setFormType()` ... there's also `->setFormTypeOptions()`. Select `->setFormType()` and set it to `ChoiceType::class`:

49 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 13
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;

class UserCrudController extends AbstractCrudController
{
... lines 18 - 22
    public function configureFields(string $pageName): iterable
    {
... lines 25 - 38
        $roles = ['ROLE_SUPER_ADMIN', 'ROLE_ADMIN', 'ROLE_MODERATOR', 'ROLE_USER'];
        yield ArrayField::new('roles')
            ->setFormType(ChoiceType::class)
... lines 42 - 46
    }
}
```

Then `->setFormTypeOptions()` ... because one of the options that you *must* pass to this form type is `choices`. Set this to `array_combine()` and pass `$roles` twice:

49 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 15
class UserCrudController extends AbstractCrudController
{
... lines 18 - 22
    public function configureFields(string $pageName): iterable
    {
... lines 25 - 38
        $roles = ['ROLE_SUPER_ADMIN', 'ROLE_ADMIN', 'ROLE_MODERATOR', 'ROLE_USER'];
        yield ArrayField::new('roles')
            ->setFormType(ChoiceType::class)
            ->setFormTypeOptions([
                'choices' => array_combine($roles, $roles),
... lines 44 - 45
            ]);
    }
}
```

I love rolls!

I know, that looks weird. This will create an array where these are both the keys *and* the values. The result is that these will be *both* the values that are saved to the database if that field is checked *and* what is displayed to the user. Lastly, set `multiple` to `true` - because we can select multiple roles - and `expanded` to `true` ... which is what makes the `ChoiceType` render as check boxes:

49 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 15
class UserCrudController extends AbstractCrudController
{
... lines 18 - 22
    public function configureFields(string $pageName): iterable
    {
... lines 25 - 38
        $roles = ['ROLE_SUPER_ADMIN', 'ROLE_ADMIN', 'ROLE_MODERATOR', 'ROLE_USER'];
        yield ArrayField::new('roles')
            ->setFormType(ChoiceType::class)
            ->setFormTypeOptions([
                'choices' => array_combine($roles, $roles),
                'multiple' => true,
                'expanded' => true,
            ]);
    }
}
```

Alrighty! Let's see what happens. Refresh and... it... explodes! Exciting!

> An error occurred resolving the options of `ChoiceType` : The options `allow_add` , `allow_delete` , `delete_empty` , `entry_options` and `entry_type` do not exist.

Hmm... I recognize these options as options that belong to the `CollectionType` , which is the type that the `ArrayField` was *originally* using. This tells me that something, *somewhere* is trying to add these options to our form type...which we don't want because... we're not using `CollectionType` anymore!

So... who *is* setting those options? This is tricky. You might expect to see them set inside of `ArrayField` . But... it's not here! What mysterious being is messing with our field?

Hello Field Configurators

The answer is something called a *Configurator*.

Scroll back down to `vendor/` . I've already opened `easycorp/easyadmin-bundle/src/` . Earlier, we were looking at the `Field/` directory: these are all the built-in fields.

After a field is created, EasyAdmin runs each through a `Configurator` system that can make *additional* changes to it. This `Configurator/` directory holds *those*. There are a couple of them - like `CommonPreConfigurator` - that are applied to *every* field. It returns `true` from `supports()` ... and does various normalizations on the field. `CommonPostConfigurator` is another that applies to every field.

But *then*, there are also a bunch of configurators that are specific to just *one*... or maybe a few... field types, including `ArrayConfigurator` . This configurator does its work when the `$field` is an `ArrayField` . The `$field->getFieldFqcn()` is basically helping to ask:

> Hey, is the current field that's being configured an `ArrayField` ? If it is, then call my `configure()` method so I can do some stuff!

And... yup! *Here* is where those options are being added. The Configurator system is something we're going to look at more later. Heck we're even going to create our own! For now, just be aware it exists.

Refactoring to ChoiceField

So, hmm. In our situation, we *don't* want the `ArrayConfigurator` to do its work. But, unfortunately, we don't really have a choice! The Configurator is *always* going to apply its logic if we're dealing with an `ArrayField` .

And actually, that's fine! Back in `UserCrudController.php` , I didn't realize it at first, but there's also a `ChoiceField` !

45 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 7
use EasyCorp\Bundle\EasyAdminBundle\Field\ChoiceField;
... lines 9 - 14
class UserCrudController extends AbstractCrudController
{
... lines 17 - 21
    public function configureFields(string $pageName): iterable
    {
... lines 24 - 38
        yield ChoiceField::new('roles')
... lines 40 - 42
    }
}
```

Hold Cmd or Ctrl to open it. Yup, we can see that it already uses ChoiceType . *So*, we don't need to take ArrayField and try to turn it *into* a choice... there's already a built-in ChoiceField *made* for this!

And now we don't need to set the form type...and we don't need the help or the form type options.I probably *could* set the choices that way, but the ChoiceField has a special method called ->setChoices() . Pass that same thing: array_combine($roles, $roles) . For the other options, we can say ->allowMultipleChoices() and ->renderExpanded() :

45 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 14
class UserCrudController extends AbstractCrudController
{
... lines 17 - 21
    public function configureFields(string $pageName): iterable
    {
... lines 24 - 37
        $roles = ['ROLE_SUPER_ADMIN', 'ROLE_ADMIN', 'ROLE_MODERATOR', 'ROLE_USER'];
        yield ChoiceField::new('roles')
            ->setChoices(array_combine($roles, $roles))
            ->allowMultipleChoices()
            ->renderExpanded();
    }
}
```

How nice is that?

Let's try this thing. Refresh and... *that* is what I was hoping for!Back on the index... ChoiceType *still* renders as a nice comma-separated list.

Oh, and by the way: if you want to see the logic that makes ChoiceType render as a comma-separated list, there a ChoiceConfigurator.php . If you open that... and scroll to the bottom - beyond a lot of normalization code - here it is: $field->setFormattedValue() where it implodes the $selectedChoices with a comma.

Rendering ChoiceList as Badges

Oh, and speaking of this type - let me close some core classes -one other method we can call is ->renderAsBadges() :

46 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 14
class UserCrudController extends AbstractCrudController
{
... lines 17 - 21
    public function configureFields(string $pageName): iterable
    {
... lines 24 - 38
        yield ChoiceField::new('roles')
... lines 40 - 42
            ->renderAsBadges();
    }
}
```

That affects the "formatted value" that we just saw...and turns it into these little guys. Cute!

Next, let's handle our user's `$avatar` field, which needs to be an upload field!

Chapter 10: Upload Fields

Our `User` class also has a property called `$avatar`:

283 lines | src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
... lines 18 - 50
    #[ORM\Column(nullable: true)]
    private ?string $avatar;
... lines 53 - 281
}
```

In the database, this stores a simple filename, like `avatar.png`. Then, thanks to a `getAvatarUrl()` method that I created before the tutorial, you can get the full URL to the image, which is `/uploads/avatars/the-file-name`:

283 lines | src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
... lines 18 - 204
    public function getAvatarUrl(): ?string
    {
        if (!$this->avatar) {
            return null;
        }

        if (strpos($this->avatar, '/') !== false) {
            return $this->avatar;
        }

        return sprintf('/uploads/avatars/%s', $this->avatar);
    }
... lines 217 - 281
}
```

To get this to work, if you create a form that has an upload field,we need to *move* the uploaded file *into* this `public/uploads/avatars/` directory and then store whatever the filename is onto the `avatar` property.

Let's add this to *our* admin area as an "Upload" field and...see if we can get it all working. Fortunately, EasyAdmin makes this pretty easy! It's like it's in the name or something...

The ImageField

Back over in `UserCrudController` (it doesn't matter where, you can have this in whatever order you want), I'm going to say `yield ImageField::new('avatar')`:

48 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 12
use EasyCorp\Bundle\EasyAdminBundle\Field\ImageField;
... lines 14 - 15
class UserCrudController extends AbstractCrudController
{
... lines 18 - 22
    public function configureFields(string $pageName): iterable
    {
... lines 25 - 26
        yield ImageField::new('avatar');
... lines 28 - 45
    }
}
```

If you have an upload field that is *not* an image, there isn't a generic `FileField` or anything like that. But you *could* use a `TextField`, then override its form type to be a special `FileUploadType` that comes from EasyAdmin. Check the `ImageField` to see what it does internally for more details.

Anyways, let's see what this does. Head back to the user index page and...ah! Broken image tags! But they *shouldn't* be broken: those image files *do* exist!

Setting the Base Path

Inspect element on an image. Ah: *every* image tag literally has just `/` then the filename. It's missing the `/uploads/avatars/` part! To configure that, we need to call `->setBasePath()` and pass `uploads/avatars` so it knows where to look:

49 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 15
class UserCrudController extends AbstractCrudController
{
... lines 18 - 22
    public function configureFields(string $pageName): iterable
    {
... lines 25 - 26
        yield ImageField::new('avatar')
            ->setBasePath('uploads/avatars');
... lines 29 - 46
    }
}
```

If you're storing images on a CDN, you can put the full URL to your CDN right here instead. Basically, put whatever path needs to come right *before* the actual filename.

Setting the Upload Dir

Head back over, refresh and... got it! Now edit the user and...error!

> The "avatar" image field must define the directory where the images are uploaded using the `setUploadDir()` method.

That's a pretty great error message! According to this, we need to tell the `ImageField()` that when we upload, we want to store the files in the `public/uploads/avatar/` directory. We can do that by saying `->setUploadDir()` with `public/avatars/uploads`:

50 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 15
class UserCrudController extends AbstractCrudController
{
... lines 18 - 22
    public function configureFields(string $pageName): iterable
    {
... lines 25 - 26
        yield ImageField::new('avatar')
            ->setBasePath('uploads/avatars')
            ->setUploadDir('public/avatars/uploads');
... lines 30 - 47
    }
}
```

Um, actually that path isn't quite right.

And when I refresh... EasyAdmin tells me! The directory *actually* is `public/uploads/avatars`. Now that I've fixed that... it works. And that's nice!

The field renders as an upload field, but with a "delete" link,the current filename and even its size! Click the file icon and choose a new image. I'll choose my friend Molly! Hit save and... *another* error.

> You cannot guess the extension as the Mime component is not installed.Try running `composer require symfony/mime`.

The Mime component helps Symfony look inside of a file to make sure it's *really* an image... or whatever type of file you're expecting. So, head over to your terminal and run:

```
composer require symfony/mime
```

Once that finishes, spin back over, hit refresh to resubmit the form and...yes! There's Molly! She's adorable! And if you look over in our `public/uploads/avatars/` directory, *there's* the file! It has the *same* filename as it did on my computer.

Tweaking the Uploaded Filename

That's... not actually perfect... because if someone *else* uploaded an image with the same name -some other fan of Molly - it would *replace* mine! So let's control how this file is named to avoid any mishaps.

Do that by calling `->setUploadedFileNamePattern()`. Before I put anything here, hold `Cmd` or `Ctrl` to open that up... because this method has *really* nice documentation. There are a bunch of wildcards that we can use to get*just* the filename we want. For example, I'll pass `[slug]-[timestamp].[extension]`, where `[slug]` is, sort of a cleaned-up version of the original filename:

51 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 15
class UserCrudController extends AbstractCrudController
{
... lines 18 - 22
    public function configureFields(string $pageName): iterable
    {
... lines 25 - 26
        yield ImageField::new('avatar')
... lines 28 - 29
            ->setUploadedFileNamePattern('[slug]-[timestamp].[extension]');
... lines 31 - 48
    }
}
```

By including the time it was uploaded, that will keep things unique!

Ok, edit that same user again, re-upload "Molly", hit "Save" and... beautiful! It *still* works! And over in the file location... awesome! We now have a "slugified" version of the new file, the timestamp, then `.jpg`. And notice that the old file is gone! That's another nice feature of EasyAdmin. When we uploaded the new file, it deleted the original since we're not using it anymore. I love that!

Handling Non-Local Files & FileUploadType

Oh, and many people like to upload their files to something like Amazon S3 instead of uploading them locally to the server. Does EasyAdmin support that? Totally! Though, you'll need to hook parts of this up by yourself. Hold `Cmd` or `Ctrl` to open `ImageField`. Behind the scenes, its form type is something called `FileUploadType`. Hold `Cmd` or `Ctrl` *again* to jump into that.

This is a custom EasyAdmin form type for uploading. Scroll down a bit to find `configureOptions()`. This declares all of the options that *we* can pass to this form type. Notice there's a variable called `$uploadNew`, which is set to a callback and `$uploadDelete`, which is *also* set to a callback. Down here, these become the `upload_new` and `upload_delete` options: two of the *many* options that you can see described here.

So if you needed to do something *completely* custom when a file is uploaded - like moving it to S3 - you could call `->setFormTypeOption()` and pass `upload_new` set to a callback that contains that logic.

So it's *very* flexible. And if you dig into the source a bit, you'll be able to figure out exactly what you need to do.

Next, it's time to learn about the purpose of the *formatted value* for each field and how to control it. That will let us render *anything* we want on the index and detail page for each field.

Chapter 11: Controlling the "Formatted Value"

Head back to the index page. One of the nice things about the `ImageField` is that you can click to see a bigger version of it. But let's pretend that we *don't* want that for some reason... like because these are meant to be tiny avatars.

Actually, EasyAdmin has a field that's made specifically *for* avatars. It's called `AvatarField`!

Back in our code, `yield AvatarField::new()` and pass it `avatar`:

53 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\AvatarField;
... lines 8 - 16
class UserCrudController extends AbstractCrudController
{
... lines 19 - 23
    public function configureFields(string $pageName): iterable
    {
... lines 26 - 27
        yield AvatarField::new('avatar');
... lines 29 - 50
    }
}
```

Yes, we *do* temporarily have two fields for `avatar`. Go refresh and... the original works, but the `AvatarField` is broken!

Inspect the image. Yup! This looks like the same problem as before: it's dumping out the filename instead of the full path to it. To fix this, the `ImageField` has a `->setBasePath()` method. Does that method exist on `AvatarField`? Apparently not!

Controlling the "Formatted Value"

So let's back up. No matter which field type you use, when a field is ultimately printed onto the page, what's printed is something called the *formatted value*. For some fields - like text fields - that *formatted value* is just rendered by itself. But for other fields, it's wrapped inside some markup. For example, if you dug into the template for the `AvatarField` - something we'll learn to do soon - you'd find that the formatted value is rendered as the `src` attribute of an `img` tag.

Anyways, the formatted value is something we can control. Do that by calling `->formatValue()` and passing a callback. I'll use a `static function()` that will receive a `$value` argument - whatever EasyAdmin would *normally* render as the formatted `$value` - and then our entity: `User $user`. Inside, we can return whatever value should be printed inside the `src` of the `img`. So, `return $user->getAvatarUrl()`:

56 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 16
class UserCrudController extends AbstractCrudController
{
... lines 19 - 23
    public function configureFields(string $pageName): iterable
    {
... lines 26 - 27
        yield AvatarField::new('avatar')
            ->formatValue(static function ($value, User $user) {
                return $user->getAvatarUrl();
            });
... lines 32 - 53
    }
}
```

The `static` isn't important... it's just kind of a "cool kid" thing to do if your callback does *not* need to leverage the `$this` variable.

Anyways, go back to your browser and refresh. Yay! We have a nice little avatar! But, if you go the the form for this user, interesting! It only renders *one* of our avatar fields. This is expected: even though we can *display* two avatar fields on the index page, we can't have two avatar fields in the *form*. The second one always wins. And that's fine. We don't actually *want* two fields... it's just nice to understand why that's happening.

If we *deleted* the `ImageField` and used the `AvatarField` on the form, you'd see that the `AvatarField` renders as a text input! Not very helpful. Ultimately, we want to use `ImageField` on the form and `AvatarField` when rendering. And we already know how to do that!

Down here... on `ImageField`, add `->onlyOnForms()`. And above, on `AvatarField`, do the opposite: `->hideOnForm()`:

58 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 16
class UserCrudController extends AbstractCrudController
{
... lines 19 - 23
    public function configureFields(string $pageName): iterable
    {
... lines 26 - 27
        yield AvatarField::new('avatar')
... lines 29 - 31
            ->hideOnForm();
        yield ImageField::new('avatar')
... lines 34 - 36
            ->onlyOnForms();
... lines 38 - 55
    }
}
```

This gives us the *exact* result we want.

Allowing Null in formatValue

Oh, and I almost forgot! In the `->formatValue()` callback, technically the `User` argument should be allowed to be null. We'll learn *why* later when we talk about entity permissions. In a real project, I would make the function look like this:

```
->formatValue(static function($value, ?User $user) {
    return $user?->getAvatarUrl();
})
```

That has a nullable `User` argument and uses a PHP 8 syntax that basically says:

> If we have a `User`, then call `getAvatarUrl()` and return that string. But if we *don't* have a user, skip calling the method and just return `null`.

I'm actually going to remove this for now... because we'll re-add it later when we hit an error.

Next, I want to customize *more* fields inside of our admin! In particular, I'm excited to check out the very powerful `AssociationField`.

Chapter 12: The AssociationField

Let's configure the fields for some of our other CRUD controllers.Go to the "Questions" page.This shows the default field list. We can do better. Open `QuestionCrudController`, uncomment `configureFields()`, and then... let's yield some fields! I'm going to write that down in my poetry notebook.

Let's yield a field for `IdField` ... and call `->onlyOnIndex()`. Then `yield Field::new('name')`:

27 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\Field;
use EasyCorp\Bundle\EasyAdminBundle\Field\IdField;

class QuestionCrudController extends AbstractCrudController
{
... lines 12 - 16
    public function configureFields(string $pageName): iterable
    {
        yield IdField::new('id')
            ->onlyOnIndex();
        yield Field::new('name');
... lines 22 - 24
    }
}
```

Yea, yea... I'm being lazy. I'm using `Field::new()` and letting it guess the field type for me.This *should* be good enough most of the time, unless you need to configure something *specific* to a field type.

Copy that... and paste this two more times for `votes` and `createdAt`. For `createdAt`, don't forget to add `->hideOnForm()`:

27 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 9
class QuestionCrudController extends AbstractCrudController
{
... lines 12 - 16
    public function configureFields(string $pageName): iterable
    {
        yield IdField::new('id')
            ->onlyOnIndex();
        yield Field::new('name');
        yield Field::new('votes');
        yield Field::new('createdAt')
            ->hideOnForm();
    }
}
```

Cool! Find your browser, refresh and...good start!

More Field Configuration

There are *a lot* of things that we can configure on these fields, and we've already seen several.If you check the auto-completion, wow! That's a great list: `addCssClass()`, `setPermission()` (which we'll talk about later) and more.We can also control the field *label*. Right now, the label for votes is..."Votes". Makes sense! But we can change that with `->setLabel('Total Votes')`.

Or, "label" is the second argument to the `new()` method, so we could shorten this by passing it there:

27 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 9
class QuestionCrudController extends AbstractCrudController
{
... lines 12 - 16
    public function configureFields(string $pageName): iterable
    {
... lines 19 - 21
        yield Field::new('votes', 'Total Votes');
... lines 23 - 24
    }
}
```

And... that works perfectly! But I think these numbers would look better if they were right-aligned.That is, *of course*, another method: `->setTextAlign('right')` :

28 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 9
class QuestionCrudController extends AbstractCrudController
{
... lines 12 - 16
    public function configureFields(string $pageName): iterable
    {
... lines 19 - 21
        yield Field::new('votes', 'Total Votes')
            ->setTextAlign('right');
... lines 24 - 25
    }
}
```

This... yup! Scooches our numbers to the right!

These are just a *few* examples of the crazy things you can do when you configure each field.And of course, many field classes have *more* methods that are specific to them.

Back on the question section, let's edit one of these.Not surprisingly, it just lists "Name" and "Total Votes".But our `Question` entity has more fields that we want here, like the `$question` text itself... and `$askedBy` and `$topic` which are both relationships:

208 lines | src/Entity/Question.php

```
... lines 1 - 13
class Question
{
    use TimestampableEntity;

    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id;

    #[ORM\Column]
    private ?string $name;

    /**
     * @Gedmo\Slug(fields={"name"})
     */
    #[ORM\Column(length: 100, unique: true)]
    private ?string $slug;

    #[ORM\Column(type: Types::TEXT)]
    private ?string $question;

    #[ORM\ManyToOne(inversedBy: 'questions')]
    #[ORM\JoinColumn(nullable: false)]
    private User $askedBy;

    #[ORM\Column]
    private int $votes = 0;

    #[ORM\OneToMany('question', Answer::class)]
    private Collection $answers;

    #[ORM\ManyToOne(inversedBy: 'questions')]
    #[ORM\JoinColumn(nullable: false)]
    private Topic $topic;

    #[ORM\Column]
    private bool $isApproved = false;

    #[ORM\ManyToOne]
    private User $updatedBy;
... lines 54 - 206
}
```

Back in `QuestionCrudController`, the `question` field will hold a lotof text, so it should be a textarea.For this, there is a (surprise!) `TextareaField`. Yield `TextareaField::new('question')` ... and then `->hideOnIndex()`:

31 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 8
use EasyCorp\Bundle\EasyAdminBundle\Field\TextareaField;

class QuestionCrudController extends AbstractCrudController
{
... lines 13 - 17
    public function configureFields(string $pageName): iterable
    {
... lines 20 - 22
        yield TextareaField::new('question')
            ->hideOnIndex();
... lines 25 - 28
    }
}
```

Because we definitely do *not* want a wall of text in the list.

Back on the form... excellent!

Hello AssociationField

Let's do the `$topic` field! This is an interesting one because it's a *relation* to the `Topic` entity. How can we handle that in EasyAdmin? With the *super* powerful `AssociationField`. Yield `AssociationField::new()` and pass `topic`:

33 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\AssociationField;
... lines 8 - 11
class QuestionCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
... lines 21 - 22
        yield AssociationField::new('topic');
... lines 24 - 30
    }
}
```

That's it!

Click "Questions" to go back to the index page.Hmm. We *do* have a "Topic" column, but it's not very descriptive.It's just "Topic" and then the ID. And if you click to edit a question, it explodes!

> Object of class `App\Entity\Topic` could not be converted to string

On both the index page *and* on the form, it's trying to find a string representation of the `Topic` object. On the index page, it guesses by using its id. But on the form... it just... gives up and explodes. The easiest way to fix this is to open the `Topic` entity and add a `__toString()` method.

Scroll down a bit... and, after the `__construct` method, add `public function __toString()`, which will return a `string`. Inside `return $this->name`:

81 lines | src/Entity/Topic.php

```
... lines 1 - 10
class Topic
{
... lines 13 - 28
    public function __toString(): string
    {
        return $this->name;
    }
... lines 33 - 79
}
```

Now when we refresh... got it! And check it out!It renders a really cool select element with a search bar on it.For free? No way!

The important thing to know about this is that it's really just a `select` element that's made to look and work fabulously.But when you type, no AJAX calls are made to build the list. *All* of the possible topics are loaded onto the page in the HTML.And *then* this JavaScript widget helps you select them.

And over on the index page for Questions,our `__toString()` method now gives us better text in the list.And EasyAdmin even renders a link to jump right *to* that Topic.

The only problem is that, when we click, it's busted!It goes to the "detail" action of `TopicCrudController` ... which we *disabled* earlier. Whoops. In a real app, you probably *won't* disable the "detail" action... it's pretty harmless. So I'm not going to worry

about this. But you *could* argue that this is a *tiny* bug in EasyAdmin because it doesn't check the permissions correctly before generating the link.

Anyways, let's repeat this `AssociationField` for the `$askedby` property in `Question`, which is *another* relationship. Over in the controller, down near the bottom... because it's less important... `yield AssociationField::new('askedBy')`:

34 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 11
class QuestionCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
... lines 21 - 28
        yield AssociationField::new('askedBy');
... lines 30 - 31
    }
}
```

As *soon* as we do that, it shows up the index page...but just with the id... and on the form, we get the same error.No problem. Pop open `User` ... I'll scroll up, then add `public function __toString(): string` ... and `return $this->getFullName()`:

288 lines | src/Entity/User.php

```
... lines 1 - 15
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
... lines 18 - 65
    public function __toString(): string
    {
        return $this->getFullName();
    }
... lines 70 - 286
}
```

Back over on the form... nice! It's way at the bottom, but works great!

Adding some Field Margin

Well, it's *so* far at the bottom that there's not much space!It's hard to see the entire list of users.Let's add some "margin-bottom" to the page. We can do that *very* easily now thanks to the `assets/styles/admin.css` file.

Let's do some digging. Ah! There's a section up here called `main-content`, which holds this entire body area.This time, instead of overriding a CSS property - since there *is* no CSS property that controls the bottom margin for this element - we can do it the normal way. Add `.main-content` with `margin-bottom: 100px`:

5 lines | assets/styles/admin.css

```
... line 1
.main-content {
    margin-bottom: 100px;
}
```

Let's check it! Refresh. Ah, that's much better! If the change didn't show up for you, try a force refresh.

Ok, the `AssociationField` is *great*. But ultimately, what it renders is just a fancy-looking `select` field... which means that *all* the users in the entire database are being rendered into the HTML right now. Watch! I'll view the page source, and search for "Tisha". Yup! The server loaded *all* of the options onto the page.If you only have a few users or topics, no biggie!.But in a real app, we're going to have hundreds, thousands, maybe even millions of users, and we *cannot* load all of those onto the page. That will absolutely break things.

But no worries: the `AssociationField` has a trick up its sleeve.

Chapter 13: Auto-complete Association Field & Controlling the Query

The `AssociationField` creates these pretty cool `select` elements. But these are really just normal, boring `select` elements with a fancy UI. *All* of the options, in this case, every user in the database, is loaded onto the page in the background to build the `select`. This means that if you have even a hundred users in your database, this page is going to start slowing down, and eventually, explode.

AssociationField::autoComplete()

To fix this, head over and call a custom method on the `AssociationField` called `->autocomplete()`:

35 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 11
class QuestionCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
... lines 21 - 28
        yield AssociationField::new('askedBy')
            ->autocomplete();
... lines 31 - 32
    }
}
```

Yes, this *is* as nice as it sounds. Refresh. It *looks* the same, but when we type in the search bar... and open the Network Tools... check it out! That made an AJAX request! So instead of loading *all* of the options onto the page, it leverages an AJAX endpoint that handles the autocompletion. Problem solved!

Controlling the Autocomplete Items with formatValue()

And as you can see, it uses the `__toString()` method on `User` to display the option, which is the same thing it does on the index page in the "Asked By" column. We *can* control that, however. How? We already know: it's our old friend `->formatValue()`. As you might remember, this takes a callback function as its argument: `static function()` with `$value` as the *first* argument and `Question` as the second:

42 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 11
class QuestionCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
... lines 21 - 28
        yield AssociationField::new('askedBy')
... line 30
            ->formatValue(static function ($value, Question $question): ?string {
... lines 32 - 36
            });
... lines 38 - 39
    }
}
```

The `$value` argument will be the formatted value that it's *about* to print onto the page. And then `Question` is the current `Question` object. We'll eventually need to make this argument `nullable` and I'll explain *why* later. But for now, just pretend that we always have a `Question` to work with.

Inside: `if (!$question->getAskedBy())` - if for some reason that field is `null`, we'll `return null`. If that *is* set, `return sprintf()` - with `%s`, ` ` for a space, and then `%s` inside of parentheses. For the first wildcard, pass `$user->getEmail()`.

Oh, whoops! In the if statement, I meant to say if `!$user =`. This, fancily, assigns the `$user` variable *and* checks to see if there *is* an askedBy user all at once. Finish the `->getEmail()` method and use `$user->getQuestions()->count()` for the second wildcard:

42 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 11
class QuestionCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
... lines 21 - 28
        yield AssociationField::new('askedBy')
... line 30
            ->formatValue(static function ($value, Question $question): ?string {
                if (!$user = $question->getAskedBy()) {
                    return null;
                }

                return sprintf('%s&amp;nbsp;(%s)', $user->getEmail(), $user->getQuestions()->count());
            });
... lines 38 - 39
    }
}
```

HTML IS Allowed in EasyAdmin

Oh, and about that ` `. I added this, in part, to show off the fact that when you render things in EasyAdminyou *can* include HTML in most situations. That's normally *not* how Symfony and Twig work, but since we're never configuring EasyAdmin *based* off of *user* input... and this is all just for an admin interface anyways, EasyAdmin allows embedded HTML in most places.

Ok, let's check things out! Reload and... boom! We get our new "Asked By" format on the index page.

The *real* reason I wanted us to do this was to point out that the formatted value is usedon the index page *and* the detail page... but it is *not* used on the form. The form *always* uses the `__toString()` method from your entity.

Controlling the Autocomplete Query

One of the things we *can* control for these association fields is the query that's used for the results. Right now, our autocomplete field returns *any* user in the database. Let's restrict this to only *enabled* users.

How? Once again, we can call a custom method on `AssociationField` called `->setQueryBuilder()`. Pass this a `function()` with a `QueryBuilder $queryBuilder` argument:

47 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 5
use Doctrine\ORM\QueryBuilder;
... lines 7 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 29
        yield AssociationField::new('askedBy')
... lines 31 - 38
            ->setQueryBuilder(function (QueryBuilder $qb) {
... lines 40 - 41
            });
... lines 43 - 44
    }
}
```

When EasyAdmin generates the list of results, it creates the query builder*for* us, and then we can modify it.Say `$queryBuilder->andWhere()` . The only secret is that you need to know that the entity*alias* in the query is always `entity` . So: `entity.enabled = :enabled` , and then `->setParameter('enabled', true)` :

47 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 29
        yield AssociationField::new('askedBy')
... lines 31 - 38
            ->setQueryBuilder(function (QueryBuilder $qb) {
                $qb->andWhere('entity.enabled = :enabled')
                    ->setParameter('enabled', true);
            });
... lines 43 - 44
    }
}
```

That's it! We don't need to *return* anything because we *modified* the `QueryBuilder` . So let's go see if it worked!

Well, I don't think we'll *notice* any difference because I'm pretty sure every user*is* enabled. But watch this. When I type... here's the AJAX request. Open up the web debug toolbar...hover over the AJAX section and click to open the profiler.

You're now looking at the profiler for the autocomplete AJAX call.Head over to Doctrine section so we can see what that query looks like. Here it is. Click "View formatted query". Cool! It looks on every field to seeif it matches the `%ti%` value *and* it has WHERE `enabled = ?` with a value of 1...which comes from up here. Super cool!

Next: could we use an `AssociationField` to handle a *collection* relationship? Like to edit the collection of*answers* related to a `Question` ? Totally! But we'll need a few Doctrine & form tricks to help us.

Chapter 14: AssociationField for a "Many" Collection

There's one other `AssociationField` that I want to include inside this CRUD section and it's an interesting one: `$answers`. Unlike `$topic` and `$answeredBy`, this is a `Collection`: each `Question` has *many* answers:

208 lines | src/Entity/Question.php

```
... lines 1 - 13
class Question
{
... lines 16 - 41
    #[ORM\OneToMany('question', Answer::class)]
    private Collection $answers;
... lines 44 - 206
}
```

Back in `QuestionCrudController`, `yield AssociationField::new('answers')`:

48 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 42
        yield AssociationField::new('answers');
... lines 44 - 45
    }
}
```

And.... let's just see what happens! Click back to the index page and... Awesome! It recognizes that it's a Collection and prints the number of answers that each `Question` has... which is pretty sweet. And if we go to the form, I'm really starting to like this error! The form is, once again, trying to get a string representation of the entity.

Configuring the choice_label Field Option

We know how to fix this: head over to `Answer.php` and add the `__toString()` method. *But*, there's actually one *other* way to handle this. If you're familiar with the Symfony Form component, then this problem of converting your entity into a string is something that you see all the time with the `EntityType`. The two ways to solve it are either to add the `__toString()` method to your entity, *or* pass your field a `choice_label` option. We can do that here thanks to the `->setFormTypeOption()` method.

Before we fill that in, open up the `AssociationField` class... and scroll down to `new`. Behind the scenes, this uses the `EntityType` for the form. So any options `EntityType` has, *we* have. For example, we can set `choice_label`, which accepts a callback or just the property on the entity that it should use. Let's try `id`:

49 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 42
        yield AssociationField::new('answers')
            ->setFormTypeOption('choice_label', 'id');
... lines 45 - 46
    }
}
```

And now... beautiful! The ID isn't super clear, but we *can* see that it's working.

by_reference => false

Let's... try removing a question! Remove "95", hit "Save and continue editing" and... uh. Absolutely nothing happened? Answer id "95" is still there!

If you're familiar with collections and the Symfony Form component, you might know the fix. Head over and configure one other form type option called `by_reference` set to `false`:

50 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 42
        yield AssociationField::new('answers')
            ->setFormTypeOption('choice_label', 'id')
            ->setFormTypeOption('by_reference', false);
... lines 46 - 47
    }
}
```

I won't go into too much detail, but basically, by setting `by_reference` to `false`, if an answer is *removed* from this question, it will force the system to call the `removeAnswer()` method that I have in `Question`:

208 lines | src/Entity/Question.php

```
... lines 1 - 13
class Question
{
... lines 16 - 163
    public function removeAnswer(Answer $answer): self
    {
        if ($this->answers->removeElement($answer)) {
            // set the owning side to null (unless already changed)
            if ($answer->getQuestion() === $this) {
                $answer->setQuestion(null);
            }
        }

        return $this;
    }
... lines 175 - 206
}
```

That method properly removes the `Answer` from `Question`. But more importantly, it sets `$answer->setQuestion()` to `null`, which is

the *owning* side of this relationship... for you Doctrine geeks out there.

orphanRemoval

Ok, try removing "95" again and saving. Hey! We upgraded to an error!

> An exception occurred ... Not null violation: ... null value in column `question_id` of relation `answer` ...

So what happened? Open `Question.php` back up. When we remove an answer from `Question`, our method sets the `question` property on the `Answer` object to `null`. This makes that `Answer` an *orphan*: its an `Answer` that is no longer related to *any* `Question`.

However, inside `Answer`, we have some code that prevents thisfrom ever happening: `nullable: false`:

94 lines | src/Entity/Answer.php

```
... lines 1 - 10
class Answer
{
... lines 13 - 23
    #[ORM\JoinColumn(nullable: false)]
    private ?Question $question;
... lines 26 - 92
}
```

If we ever try to save an Answer without a Question, our database will stop us.

So we need to decide what should happen when an answer is "orphaned".In some apps, maybe orphaned answers are ok.In that case, change to `nullable: true` and let it save. But in *our* case, if an answer is removed from its question, it should be *deleted*.

In Doctrine, there's a way to force this and say:

> If an `Answer` ever becomes orphaned, please delete it.

It's called "orphan removal". Inside of `Question`, scroll up to find the `$answers` property... here it is. On the end, add `orphanRemoval` set to `true`:

208 lines | src/Entity/Question.php

```
... lines 1 - 13
class Question
{
... lines 16 - 41
    #[ORM\OneToMany('question', Answer::class, orphanRemoval: true)]
    private Collection $answers;
... lines 44 - 206
}
```

Now refresh and... yes! It worked! The "95" is gone! And if you looked in the database, no answer with "ID 95" would exist. Problem solved!

Customizing the AssociationField

The last problem with this answers area is the *same* problem we have with the other ones.If we have many answers in the database, they're *all* going to be loaded onto the page to render the `select`. That's not going to work, so let's add `->autocomplete()`:

51 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 42
        yield AssociationField::new('answers')
            ->autocomplete()
... lines 45 - 48
    }
}
```

When we refresh, uh oh!

> Error resolving `CrudAutocompleteType` : The option `choice_label` does not exist.

Ahhh. When we call `->autocomplete()` , this *changes* the form type behind `AssociationField` . And *that* form type does *not* have a `choice_label` option! Instead, it *always* relies on the `__toString()` method of the entity to display the options, no matter what.

No big deal. Remove that option:

50 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 42
        yield AssociationField::new('answers')
            ->autocomplete()
            ->setFormTypeOption('by_reference', false);
... lines 46 - 47
    }
}
```

You can probably guess what will happen if we refresh.Yup! *Now* it's saying:

> Hey Ryan! Go add that `__toString()` method!

Ok fine! In `Answer` , anywhere in here, add `public function __toString(): string` ... and `return $this->getId()` :

99 lines | src/Entity/Answer.php

```
... lines 1 - 10
class Answer
{
... lines 13 - 93
    public function __toString(): string
    {
        return $this->getId();
    }
}
```

Now... we're back! And if we type... well... the search isn't *great* because it's just numbers, but you get the idea.Hit save and... nice!

Next, let's dig into the powerful Field Configurators systemwhere you can modify something about *every* field in the system from one place. It's also key to understanding how the core of EasyAdmin works.

Chapter 15: The Field Configurator System

Let's finish configuring a few more fields and *then* talk more about a crazy-cool important system that's working behind the scenes: *field configurators*.

Disabling a Form Field only on Edit

One other field that I want to render in the question section is "slug": `yield Field::new('slug')` ... and then `->hideOnIndex()`:

52 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 12
class QuestionCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 24
        yield Field::new('slug')
            ->hideOnIndex();
... lines 27 - 49
    }
}
```

This will just be for the forms.

Now, when we go to Questions... it's not there. If we *edit* a question, it *is* there. Slugs are typically auto-generated... but occasionally it *is* nice to control them. However, once a question has been created and the slug set, it should *never* change.

And so on the *edit* page, I want to *disable* this field. We *could* remove it entirely by adding `->onlyWhenCreating()` ... but pff. That's too easy! Let's *show* it, but disable it.

How? We already know that each field has a form type behind it. And each form type in Symfony has an option called `disabled`. To control this, we can say `->setFormTypeOption()` and pass `disabled`:

57 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 13
class QuestionCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 25
        yield Field::new('slug')
... line 27
            ->setFormTypeOption(
                'disabled',
... line 30
            );
... lines 32 - 54
    }
}
```

But we can't just set this to "true" everywhere...since that would disable it on the new page. *This* is where the `$pageName` argument comes in handy! It'll be a string like `index` or `edit` or `details`. So we can set `disabled` to `true` if `$pageName !==` ... and I'll use the `Crud` class to reference its `PAGE_NEW` constant:

57 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 13
class QuestionCrudController extends AbstractCrudController
{
... lines 16 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 25
        yield Field::new('slug')
... line 27
            ->setFormTypeOption(
                'disabled',
                $pageName !== Crud::PAGE_NEW
            );
... lines 32 - 54
    }
}
```

Let's do this! Over here on the edit page... it's disabled. And if we go back to Questions...and create a new question... we have a *not* disabled slug field!

Ok, enough with the question section!Close `QuestionCrudController` and open `AnswerCrudController`. Uncomment `configureFields()` ... and then I'll paste in some fields. Oh! I just need to retype the end of these classes and hit `Tab` to auto-complete them... to get the missing `use` statements:

34 lines | src/Controller/Admin/AnswerCrudController.php

```
... lines 1 - 6
use EasyCorp\Bundle\EasyAdminBundle\Field\AssociationField;
use EasyCorp\Bundle\EasyAdminBundle\Field\Field;
use EasyCorp\Bundle\EasyAdminBundle\Field\IdField;
use EasyCorp\Bundle\EasyAdminBundle\Field\IntegerField;

class AnswerCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
        yield IdField::new('id')
            ->onlyOnIndex();
        yield Field::new('answer');
        yield IntegerField::new('votes');
        yield AssociationField::new('question')
            ->hideOnIndex();
        yield AssociationField::new('answeredBy');
        yield Field::new('createdAt')
            ->hideOnForm();
        yield Field::new('updatedAt')
            ->onlyOnDetail();
    }
}
```

Perfect There's nothing special here. You might want to add autocomplete to the `question` and `answeredBy` fields, but I'll leave that up to you.

If we refresh... the Answers page looks awesome! And if we edit one, we get our*favorite* error:

> Object of class `Question` could not be converted to string

This comes from the `AssociationField`. The solution is to go into `Question.php` and add `public function __toString(): string` ... and `return $this->name`:

213 lines | src/Entity/Question.php

```
... lines 1 - 13
class Question
{
... lines 16 - 59
    public function __toString()
    {
        return $this->name;
    }
... lines 64 - 211
}
```

And now... that page works too!

Globally Changing a Field

Back on the main Answers page... sometimes this text might be too long to fit nicely in the table. Let's truncate it if it's longer than a certain length. Doing this is... really easy. Head over to the `answer` field, use `TextField` ... and then leverage a custom method `->setMaxLength()` :

```
public function configureFields(string $pageName): iterable
{
    // ...
    yield TextField::new('answer')
        // ...
        ->setMaxLength(50);
}
```

If we set this to 50, that will truncate any text that's longer than 50 characters!

But, I'm going to undo that. Why? Because I want us to do something more interesting!

Right now, I'm using `Field` which tells EasyAdmin to guess the best field type. This is printing as a textarea... so its field type is *really* `TextareaField` ... and we can use that if we want to.

More about Field Configurators

Here's the new goal: I want to set a max length for *every* `TextareaField` across our *entire* app. How can we change the behavior of *many* fields at the same time? With a field configurator.

We talked about these a bit earlier. Scroll down: I already have `/vendor/easycorp/easyadmin-bundle/` opened up. One of the directories is called `Field/` ... and it has a subdirectory called `Configurator/` . After your field is created, it's passed through this configurator system. Any configurator can then make changes to any field. There are two "common" configurators. `CommonPreConfigurator` is called when your field is created, and it does a number of different things to your field, including building the label, setting whether it's required, making it sortable, setting its template path, etc.

There's also a `CommonPostConfigurator` , which runs *after* your field is created.

But mostly, these configurators are specific to one or just a few field *types*. And if you're ever using a field and something "magical" is happening behind the scenes, there's a good chance that it's coming from one of these. For example, the `AssociationConfigurator` is a bit complex... but it sets up all *kinds* of stuff to get that field working.

Knowing about these is important because it's a great way to understand what's going on under the hood, like why some field is behaving in some way or how you can extend it. But it's *also* great because we can create our *own* custom field configurator!

Let's do just that. Up in `src/` ... here we go... create a new directory called `EasyAdmin/` and, inside, a new PHP class called... how about `TruncateLongTextConfigurator` . The only rule for these classes is that they need to implement a `FieldConfiguratorInterface` :

23 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 2
namespace App\EasyAdmin;
... lines 4 - 5
use EasyCorp\Bundle\EasyAdminBundle\Contracts\Field\FieldConfiguratorInterface;
... lines 7 - 10
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 13 - 21
}
```

Go to "Code"->"Generate" or Cmd + N on a Mac, and select "Implement Methods" to implement the two that we need:

23 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 4
use EasyCorp\Bundle\EasyAdminBundle\Context\AdminContext;
... line 6
use EasyCorp\Bundle\EasyAdminBundle\Dto\EntityDto;
use EasyCorp\Bundle\EasyAdminBundle\Dto\FieldDto;
... lines 9 - 10
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
    public function supports(FieldDto $field, EntityDto $entityDto): bool
    {
... line 15
    }

    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
... line 20
    }
}
```

Here's how this works. For *every* field that we return in configureFields() for *any* CRUD section, EasyAdmin will call the supports() method on our new class and basically ask:

> Does this configurator want to operate on this specific field?

These typically return $field->getFieldFqcn() === a specific field type. In our case, we're going to target textarea fields: TextareaField::class :

23 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 8
use EasyCorp\Bundle\EasyAdminBundle\Field\TextareaField;

class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
    public function supports(FieldDto $field, EntityDto $entityDto): bool
    {
        return $field->getFieldFqcn() === TextareaField::class;
    }
... lines 17 - 21
}
```

If the field that's being created is a TextareaField , then we *do* want to modify it. Next, *if* we return true from supports, EasyAdmin calls configure() . Inside, just for now, dd() the $field variable:

23 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 10
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 13 - 17
    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
        dd($field);
    }
}
```

Let's see if it triggers! Find your browser. It doesn't matter where I go, so I'll just go to the index page. And... boom! It hits! This `FieldDto` is *full* of info *and* full of ways to *change* it.

Let's dive into it next, including how this `FieldDto` relates to the `Field` objects that we return from `configureFields()`.

Chapter 16: Field Configurator Logic

We just bootstrapped a *field configurator*: a super-hero-like class where we getto modify *any* field in *any* CRUD section from the comfort of our home. We really *do* live in the future.

At this point in the process, what EasyAdmin gives us is something called a `FieldDto`, which, as you can see, contains *all* the info about this field, like its value, formatted value, form type, template path and much more.

FieldDto vs Field

One thing you might have noticed is that this is a `FieldDto`. But when we're in our CRUD controllers, we're dealing with the `Field` class. Interesting. This is a pattern that EasyAdmin follows a lot. When we're configuring things, we use an easy class like `Field`... where `Field` gives us a lot of nice methods to control everything about it.

But behind the curtain, the *entire* purpose of the `Field` class - or any of the other field classes - is to take *all* of the info we give it and create a `FieldDto`. I'll call `->formatValue()` temporarily and hold `Cmd` or `Ctrl` to jump into that. This moved us into a `FieldTrait` that `Field` uses.

And check it out! When we call `formatValue()`, what that *really* does is say `$this->dto->setFormatValueCallable()`. That Dto is the `FieldDto`. So we call nice methods on `Field`, but in the background, it uses all of that info to craft this `FieldDto`. This means that the `FieldDto` contains the same info as the `Field` objects, but its data, structure and methods are all a bit different.

Truncating the Formatted Value

Ok: back to our goal of truncating long textarea fields. Add a `private const MAX_LENGTH = 25` to keep track of our limit:

32 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 11
12  class TruncateLongTextConfigurator implements FieldConfiguratorInterface
13  {
14      private const MAX_LENGTH = 25;
... lines 15 - 30
31  }
```

Then, below, `if (strlen($field->getFormattedValue()))` is less than or equal to `self::MAX_LENGTH`, then just return:

32 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 11
12  class TruncateLongTextConfigurator implements FieldConfiguratorInterface
13  {
... lines 14 - 20
21      public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
22      {
23          if (strlen($field->getFormattedValue()) <= self::MAX_LENGTH) {
24              return;
25          }
... lines 26 - 29
30      }
31  }
```

And yes, I *totally* forgot about the `<= self::MAX_LENGTH` part. I'll add that later. You should add it now.

Anyways, assuming you wrote this correctly, it says that if the formatted value is already less than 25 characters, don't bother changing it: just let EasyAdmin render like normal.

Below, let's truncate: `$truncatedValue =` ... and I'll use the `u()` function. Hit `Tab` to autocomplete that. Just like with a class, it added a `use` statement on top:

32 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 9
use function Symfony\Component\String\u;
... lines 11 - 32
```

The `u` function gives us a `UnicodeString` object from Symfony's String component.

Pass this `$field->getFormattedValue()` and call `->truncate()` with `self::MAX_LENGTH`, `...` and `false`:

32 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 11
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 14 - 20
    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
        if (strlen($field->getFormattedValue()) <= self::MAX_LENGTH) {
            return;
        }

        $truncatedValue = u($field->getFormattedValue())
            ->truncate(self::MAX_LENGTH, '...', false);
... line 29
    }
}
```

The last argument just makes truncate a little cleaner. Oh, and I forgot a colon right there. That's better. Finally, call `$field->setFormattedValue()` and pass it `$truncatedValue` to override what the formatted value *would* be:

32 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 11
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 14 - 20
    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
        if (strlen($field->getFormattedValue()) <= self::MAX_LENGTH) {
            return;
        }

        $truncatedValue = u($field->getFormattedValue())
            ->truncate(self::MAX_LENGTH, '...', false);
        $field->setFormattedValue($truncatedValue);
    }
}
```

Let's try it! Move over, refresh and... absolutely nothing happens! All of the items in this column *still* have the same length as before. What's happening? It's not the bug in my code... something *else* is going on. But what?

Field Configurator Order

When we create a class and make it implement `FieldConfiguratorInterface`, Symfony's `autoconfigure` feature adds a special tag to our service called `ea.field_configurator`. *That's* the key to getting your field into the configurator system.

At your terminal, run `symfony console debug:container`. And we can actually list all the services with that tag by saying `--tag=ea.field_configurator`:

```
symfony console debug:container --tag=ea.field_configurator
```

Beautiful! This shows, as expected, a *bunch* of services: all the core field configurators plus our configurator.A few of these, like `CommonPreConfigurator` and `CommonPostConfigurator` have a *priority*, which controls the order in which they're called.

If you look closely, our `TruncateLongTextConfigurator` has a priority of 0, like most of these.But, apparently by chance, our `TruncateLongTextConfigurator` is being called before a *different* configurator that is then *overriding* our formatted value! I believe it's `TextConfigurator`. Let's go see if that's the case. Search for `TextConfigurator.php` and make sure to look in "All Places".Here it is!

And... yep! The `TextConfigurator` operates on `TextField` and `TextareaField`. And one of the things it does is set the formatted value! So our class is called first, we set the formatted value...and then a second later, *this* configurator overrides that. Rude!

Setting a Configurator Priority

The fix is to get *our* configurator to be called *after* this. To do that, it needs a *negative* priority.

Open up `config/services.yaml`. This is a *rare* moment when we need to configure a service manually.Add `App\EasyAdmin\TruncateLongTextConfigurator:`. We don't need to worry about any potential arguments: those will still be autowired. But we *do* need to add `tags:` with `name: ea.field_configurator` and `priority: -1`:

34 lines | config/services.yaml

```
... lines 1 - 7
services:
... lines 9 - 30
    App\EasyAdmin\TruncateLongTextConfigurator:
        tags:
            - { name: 'ea.field_configurator', priority: -1 }
```

Autoconfiguration *normally* add this tag for us... but with a priority of zero. By setting the tag manually, we can control that.

Whew! Testing time! Refresh and... it *still* doesn't work? Ok, *now* this is my fault. In the configurator, add the missing `< self::MAX_LENGTH`:

32 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 11
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 14 - 20
    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
        if (strlen($field->getFormattedValue()) <= self::MAX_LENGTH) {
... line 24
        }
... lines 26 - 29
    }
}
```

To *fully* test this... and prove the priority was needed, I'll comment out my configurator service.And... yup! The strings *still* aren't truncated. But if I put that back... and try it... yes! It shortened!

Over on the detail page, it *also* truncates here. Could we... truncate on the index page but *not* on the details page? Totally! It's just a matter of figuring out what the current page is from inside the configurator.

The All Powerful AdminContext

One of the arguments passed to us is `AdminContext`:

32 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 4
use EasyCorp\Bundle\EasyAdminBundle\Context\AdminContext;
... lines 6 - 11
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 14 - 20
    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
... lines 23 - 29
    }
}
```

We're going to talk more about this later, but this object holds *all* the information about your admin section. For example, we can say `$crud = $context->getCrud()` to fetch a CRUD object that's the result of the `configureCrud()` method in our CRUD controllers and dashboard. Use this to say: `if ($crud->getCurrentPage() === Crud::PAGE_DETAIL)`, then `return` and do nothing:

37 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 4
use EasyCorp\Bundle\EasyAdminBundle\Config\Crud;
... lines 6 - 12
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 15 - 21
    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
        $crud = $context->getCrud();
        if ($crud?->getCurrentPage() === Crud::PAGE_DETAIL) {
            return;
        }
... lines 28 - 34
    }
}
```

Go refresh. Yes! We get the *full* text on the detail page. Btw, it's not too important, but there *are* some edge cases where `$context->getCrud()` could return null... so I'll code defensively:

37 lines | src/EasyAdmin/TruncateLongTextConfigurator.php

```
... lines 1 - 12
class TruncateLongTextConfigurator implements FieldConfiguratorInterface
{
... lines 15 - 21
    public function configure(FieldDto $field, EntityDto $entityDto, AdminContext $context): void
    {
... line 24
        if ($crud?->getCurrentPage() === Crud::PAGE_DETAIL) {
... line 26
        }
... lines 28 - 34
    }
}
```

If you hold `Cmd` or `Ctrl` to open `getCrud()`, yup! It returns a nullable `CrudDto` ... though in practice, I think this is *always* set as long as you're *on* an admin page.

Next: changing the formatted value for a field is great, but limited. What if you want to render something *totally* different? Including custom markup and logic? To do that, we can override the field template.

Chapter 17: Overriding Field Templates

We know that a field describes both the *form type* that you see on the form and also *how* that field is rendered on the detail and index pages. We also know how easy it is to customize the form type. We can `->setFormType()` to use a completely different type or `->setFormTypeOption()` to *configure* that type.

We can *also* change a lot about how each *renders* on the detail and index pages. For example, let's play with this "Votes" field:

34 lines | src/Controller/Admin/AnswerCrudController.php

```
... lines 1 - 9
use EasyCorp\Bundle\EasyAdminBundle\Field\IntegerField;

class AnswerCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
... lines 21 - 23
        yield IntegerField::new('votes');
... lines 25 - 31
    }
}
```

If I autocomplete the methods on this, we have options like `->setCssClass()`, `->addWebpackEncoreEntries()`, `->addHtmlContentsToBody()`, and `->addHtmlContentsToHead()`. You can even call `->setTemplatePath()` to *completely* override how this field is rendered on the index and detail pages, which we'll do in a moment.

But also notice that there's `->setTemplatePath()` *and* `->setTemplateName()`. What's the difference?

Template "Names" and the Template Registry

To answer that question, I'm going to hit `Shift` + `Shift` and open up a core class from EasyAdmin called `TemplateRegistry.php`. If you don't see it, make sure to "Include non-project items".

Perfect! Internally, EasyAdmin has *many* templates and it maintains this "map" of template names to the template *filename* behind each. So when you call `->setTemplateName()`, what you would pass is some *other* template name. For example, I could pass `crud/field/money` if I wanted to use *that* template instead of the normal one.

But you probably won't override the template *name* very often. Most of the time, if you want to completely control how a field is rendered, you'll call `->setTemplatePath()`.

Here's the plan: when "Votes" is rendered on the index and detail pages, I want to render a *completely* different template. Let's call it `admin/field/votes.html.twig`:

35 lines | src/Controller/Admin/AnswerCrudController.php

```
... lines 1 - 11
class AnswerCrudController extends AbstractCrudController
{
... lines 14 - 18
    public function configureFields(string $pageName): iterable
    {
... lines 21 - 23
        yield IntegerField::new('votes')
            ->setTemplatePath('admin/field/votes.html.twig');
... lines 26 - 32
    }
}
```

Ok! Time to create that. In `templates/`, add 2 new directories `admin/field` ... and a new file: `votes.html.twig`. Inside, I don't really know what to put here yet, so I'll just put "votes!"... and see what happens:

```
templates/admin/field/votes.html.twig
1  votes!
```

When we move over and refresh... there it is! We are now in *complete* control of the votes!

Digging into the Core Templates

But, if you're like me, you're probably wondering what we can *do* inside of here. What variables do we have access to? One important thing to realize (and you can see it here in `TemplateRegistry.php`) is that every single field has a corresponding template. If you need to extend or change how a field is rendered, looking into the *core* template is pretty handy.

For example, `votes` is an `IntegerField`. Whelp, there's a template called `integer.html.twig`. Close this template registry and... let's go find that! Open `vendor/easycorp/easyadmin-bundle/src/` ... close up `Field/` and instead open `Resources/views/crud/field`. *Here* is the list of *all* of the field templates in the system. You can *also* see other templates that are used to render *other* parts of EasyAdmin... and you can override these as well.

Open up `integer.html.twig`. Ok cool. Check out the collection of comments on top. I like this! It helps our editor (and us) to know which variables we have access to. Apparently, we have access to a `field` variable, which is that familiar `FieldDto` object we talked about earlier. All the integer template does is just... print `field.formattedValue`.

Customizing the Template

Copy these three lines and paste them into our `votes.html.twig`:

```
templates/admin/field/votes.html.twig
1  {# @var ea \EasyCorp\Bundle\EasyAdminBundle\Context\AdminContext #}
2  {# @var field \EasyCorp\Bundle\EasyAdminBundle\Dto\FieldDto #}
3  {# @var entity \EasyCorp\Bundle\EasyAdminBundle\Dto\EntityDto #}
   ... lines 4 - 5
```

Then instead of "votes!", say `field.formattedValue` "votes":

```
templates/admin/field/votes.html.twig
1  {# @var ea \EasyCorp\Bundle\EasyAdminBundle\Context\AdminContext #}
2  {# @var field \EasyCorp\Bundle\EasyAdminBundle\Dto\FieldDto #}
3  {# @var entity \EasyCorp\Bundle\EasyAdminBundle\Dto\EntityDto #}
4  {{ field.formattedValue }} votes
```

And when we try this... beautiful! But I bet we can make this fancier! If the votes are negative, let's put a little thumbs down. And if positive, a thumbs up.

Take off the word "votes"... and add `if field.`. Hmm. What we want to get is the *underlying* value - the true `integer`, not necessarily the "formatted" value. We can get that by saying `field.value`.

So `formattedValue` is the *string* that would print on the page, while `value` is the actual underlying (in this case) integer. So `if field.value >= 0`, `else`, and `endif`:

```
templates/admin/field/votes.html.twig
   ... lines 1 - 3
4  {% if field.value >= 0 %}
   ... line 5
6  {% else %}
   ... line 7
8  {% endif %}
9  {{ field.formattedValue }}
```

If it *is* greater than zero, add an icon with `fas fa-thumbs-up text-success`. Copy that... and paste for our thumbs down with `text-danger`:

10 lines | templates/admin/field/votes.html.twig

```
... lines 1 - 3
{% if field.value >= 0 %}
    <i class="fas fa-thumbs-up text-success"></i>
{% else %}
    <i class="fas fa-thumbs-down text-danger"></i>
{% endif %}
... lines 9 - 10
```

And... just like that, we're making this field render *however* we want. It doesn't change how it looks like inside of the *form* (that's entirely handled by the form type), but it *does* control how it's rendered on the index page, *and* the detail page.

But, hmm. We also have a "votes" field inside of the Questions section. While it would be pretty easy to also point *that* votes field to the same new template, instead, I want to create a brand new *custom* field in EasyAdmin. That's next.

Chapter 18: Creating a Custom Field

On the Answers CRUD, we just created this nice custom "Votes" template,which we then *use* by calling `->setTemplatePath()` on the `votes` field. But we *also* have a votes field over in the Questions section...which still renders the *boring* way. I want to use our new template in *both* places.

Technically, doing this is super easy!We could copy `->setTemplatePath()`, go up, open `QuestionCrudController`, find the `votes` field... then paste to use that template path.

But *instead*, let's create a *custom* field.

We know that a field class - like `TextareaField` or `AssociationField` - defines how a field looks on the index and detail pages...as well as how it's rendered on a form. A custom field is a *great* way to encompass a *bunch* of custom field configuration in one place so you can reuse it. And *creating* a custom field is pretty easy.

Creating the Custom Field

Down in the `src/EasyAdmin/` directory, create a new PHP class called, how about, `VotesField`.

The only rule for a field is that it needs to implement `FieldInterface`. This requires us to have two methods: `new` and `getAsDto()`. But what you'll typically do is `use FieldTrait` to make life easier.

17 lines | src/EasyAdmin/VotesField.php

```
... lines 1 - 4
use EasyCorp\Bundle\EasyAdminBundle\Contracts\Field\FieldInterface;
... lines 6 - 7
class VotesField implements FieldInterface
{
    use FieldTrait;
... lines 11 - 15
}
```

Click to open that. Ok, this `FieldTrait` helps manage the `FieldDto` object, with a bunch of useful methodslike `setLabel()`, `setValue()` and `setFormattedValue()` that all fields share.

So *now*, if you go to Code Generate - or "cmd + N" on a Mac -the only thing we need to implement is `new()`. *This* is where we customize all the options for the field.

17 lines | src/EasyAdmin/VotesField.php

```
... lines 1 - 7
class VotesField implements FieldInterface
{
... lines 10 - 11
    public static function new(string $propertyName, ?string $label = null)
    {
        // TODO: Implement new() method.
    }
}
```

Our votes field is *currently* an `IntegerField`. Hold "cmd" or "ctrl" to open that and look at *its* `new()` method... because we want *our* method to look *very* much like this... with a few differences. So copy all of this, close, head to `VotesField` ... and paste. Hit "Ok" to add that use statement on top. I'll also remove `OPTION_NUMBER_FORMAT`. We won't need that... and it relates to a field configurator that I'll show you in a minute.

24 lines | src/EasyAdmin/VotesField.php

```
... lines 1 - 6
use Symfony\Component\Form\Extension\Core\Type\IntegerType;
... line 8
class VotesField implements FieldInterface
{
... lines 11 - 12
    public static function new(string $propertyName, ?string $label = null)
    {
        return (new self())
            ->setProperty($propertyName)
            ->setLabel($label)
            ->setTemplateName('crud/field/integer')
            ->setFormType(IntegerType::class)
            ->addCssClass('field-integer')
            ->setDefaultColumns('col-md-4 col-xxl-3');
    }
}
```

Ok, good start! You may have noticed that `->setDefaultColumns()` is crossed out. That's because it's marked as "internal". That *usually* means it's a function that *we* shouldn't use directly. But in this case, the documentation says that it *is* ok to use from *inside* of a field class... which is where we are!

At this point, we can customize *anything*! Like `->addWebpackEncoreEntries()` to add an extra Webpack Encore entry that will be included when this field is used. What *we* want to do, instead of calling `->setTemplateName()` so that it uses the standard integer field template, is to say `->setTemplatePath()` and pass the same thing we have in `AnswerCrudController`, which is `admin/field/votes.html.twig`.

As a reminder to myself, I'll add some comments about which part controls the index and detail pages... and which part controls the form.

27 lines | src/EasyAdmin/VotesField.php

```
... lines 1 - 12
    public static function new(string $propertyName, ?string $label = null)
    {
        return (new self())
... lines 16 - 17
            // this template is used in 'index' and 'detail' pages
            ->setTemplatePath('admin/field/votes.html.twig')
            // this is used in 'edit' and 'new' pages to edit the field contents
            // you can use your own form types too
            ->setFormType(IntegerType::class)
... lines 23 - 24
    }
}
```

Using the Custom field

Ok, that's it! Let's go use this!

In `AnswerCrudController`, change this to `VotesField` ... and we don't need `->setTemplatePath()` anymore.

38 lines | src/Controller/Admin/AnswerCrudController.php

```
... lines 1 - 12
class AnswerCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 28
        yield VotesField::new('votes', 'Total Votes')
... lines 30 - 35
    }
}
```

Then, in `QuestionCrudController`, do the same thing. Add `VotesField` and... done! If we wanted to, we could even put this `->setTextAlign('right')` *inside* the custom field... *or* remove it.

58 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 4
use App\EasyAdmin\VotesField;
... lines 6 - 14
class QuestionCrudController extends AbstractCrudController
{
... lines 17 - 21
    public function configureFields(string $pageName): iterable
    {
... lines 24 - 35
        yield VotesField::new('votes', 'Total Votes')
... lines 37 - 55
    }
}
```

Testing time! Over in Questions, refresh and... got it! And on the Answers page...it looks great there too!

Watch out for Missing Configurators

But one tiny word of warning. Now that we've changed from `IntegerField` to `VotesField`, *if* there's a field configurator for the `IntegerField`, it will *no* longer be used.

And... there *is* such a configurator. Back down in `vendor/easycorp/easyadmin-bundle/src/Field/Configurator`, you'll find `IntegerConfigurator`. This operates *only* when the field you're using is an `IntegerField`. And so, this configurator *was* being used until a second ago... but not anymore.

If you look inside, it does some work with a custom number format, which allows you to control the *format* that the number is printed. We don't really need this, but don't forget about the "field configurator" system... and how a *custom* field won't be processed in the same way.

Next, let's learn how to configure a bit more of the CRUD itself, like how a CRUD section is sorted by default, pagination settings, and more.

Chapter 19: configureCrud()

If you look at the index page of any of the crud sections, it doesn't sort by defaultIt just... lists things in whatever order they come out of the database. It would nice to *change* that: to sort by a specific column whenever we go to a section.

So far, we've talked about configuring assets, fields and actions.But "how a crud section sorts"... doesn't fall into any of these categories. Nope, for the first time, we need to configure something on the "crud section" as a whole.

Go to one of your crud controllers and open its base class.We know that there are a number of methods that we can override to control things... and we've already done that for many of these.But we have *not* yet used `configureCrud()`. As its name suggests, this is all about controlling settings on the entire crud *section*.

And *just* like with most of the other methods, we can override this in our dashboard controllerto make changes to *all* crud sections, or override it in one specific crud controller.

Sorting All Crud Sections

Let's see if we can set the default sortingacross our entire *admin* to sort by id descending.To do that, open `DashboardController` and, anywhere inside, go to Code -> Generate -or command+N on a Mac - select override methods and choose `configureCrud()`.

79 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 21
class DashboardController extends AbstractDashboardController
{
... lines 24 - 58
    public function configureCrud(): Crud
    {
... lines 61 - 64
    }
... lines 66 - 77
}
```

The `Crud` object has a bunch of methods on it...including one called `setDefaultSort()`. That sounds handy! Pass that the array of the fields we want to sort by. So, `id` set to `DESC`.

79 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 58
    public function configureCrud(): Crud
    {
        return parent::configureCrud()
            ->setDefaultSort([
                'id' => 'DESC',
            ]);
    }
... lines 66 - 79
```

Back over at the browser, when we click on "Questions"...beautiful! By default, it now sorts by id.Really, *all* sections now sort by id.

Sorting Differently in One Section

And what if we want to sort Questions in a different way than the default? I bet you already know how we would do that.So let's make things more interesting. Every `Question` is owned by a `User`. What if we wanted to show the questions whose users are enabled first? Can we do that?

First, since we want to only apply this to the questions section,we need to make this change inside of `QuestionCrudController`. Go to the bottom and...same thing: override `configureCrud()` ... and call the exact same method as before: `setDefaultSort()`. Now

we can say, `askedBy` - that's the property on `Question` that is a relation to `User` - `askedBy.enabled`. Set this to `DESC`.

And then, after sorting by enabled first, sort the rest by `createdAt` `DESC`.

67 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 14
class QuestionCrudController extends AbstractCrudController
{
... lines 17 - 21
    public function configureCrud(Crud $crud): Crud
    {
        return parent::configureCrud($crud)
            ->setDefaultSort([
                'askedBy.enabled' => 'DESC',
                'createdAt' => 'DESC',
            ]);
    }
... lines 30 - 65
}
```

Let's check it! Click "Questions". Because we're sorting across multiple fields, you don't see any header highlighted as the currently-sorted column. But... it looks right, at least based on the "Created At" dates.

To know for sure, click the database link on the web debug toolbar. Then... search this page for "ORDER BY". There it is! Click "View formatted query". And...yes! It's ordering by `user.enabled` and then `createdAt`. Pretty cool.

Disabling Sorting on a Field

So we now have default sorting... though the user can, of course, still click any header to sort by whichever column they want. But sometimes, sorting by a specific field doesn't make sense! You can see that it's already disabled for "answers".

And if we go over to... let's see... the Users section, there's also no sort for the avatar field, which *also* seems reasonable.

If you want to control this a bit further, you *can*. Like, let's pretend that we don't want people sorting by the name of the question. This is something we can configure on the field itself.

In `QuestionCrudController`, for the name field, call `setSortable(false)`.

68 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 30
    public function configureFields(string $pageName): iterable
    {
... lines 33 - 41
        yield Field::new('name')
            ->setSortable(false);
... lines 44 - 65
    }
... lines 67 - 68
```

And... just like that, the arrow is gone.

Inlining Controls for an Admin Section

Before we move on - because there isn't *a ton* that we need to control on the CRUD-level, let me show you one more thing. Head to the Topics section. This entity is really *simple*... so we have plenty of space here to show these fields.

Normally, the "actions" on the index page are hidden under this dropdown. But, we *can* render them inline.

To do that, head to `TopicCrudController` ... go down... and override `configureCrud()`. On the `Crud` object, call `->showEntityActionsInlined()`.

41 lines | src/Controller/Admin/TopicCrudController.php

```
... lines 1 - 10
class TopicCrudController extends AbstractCrudController
{
... lines 13 - 17
    public function configureCrud(Crud $crud): Crud
    {
        return parent::configureCrud($crud)
            ->showEntityActionsInlined();
    }
... lines 23 - 39
}
```

That's it. Now... yea! That looks better.

Next: I want to talk about using a what-you-see-is-what-you-get editor.There's actually a simple one built *into* Easy Admin. But we're going to go further and install our own. Doing *that* will require some custom JavaScript.

Chapter 20: Custom Field JavaScript

Go to the Question edit page. Ok: the question itself is in a `textarea`, which is nice. But it would be even *better* if we could have a *fancy* editor that helps with our markup.

Hello TextEditorField

Fortunately EasyAdmin has something *just* for this. In `QuestionCrudController`, for the `question` field, instead of a textarea, change to `TextEditorField`.

69 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 13
use EasyCorp\Bundle\EasyAdminBundle\Field\TextEditorField;
... line 15
class QuestionCrudController extends AbstractCrudController
{
... lines 18 - 31
    public function configureFields(string $pageName): iterable
    {
... lines 34 - 44
        yield TextEditorField::new('question')
... lines 46 - 66
    }
}
```

Refresh the page and... we have a cute lil' editor for free! Nice!

If you look inside of `TextEditorField` ... you can see a bit about how this works. Most importantly, it calls `addCssFiles()` and `addJsFiles()`. Easy Admin comes with extra JavaScript and CSS that adds this special editor functionality. And by leveraging these two methods, that CSS and Javascript is *included* on the page whenever this field is rendered.

Adding JavaScript to our Admin Encore Entry

So this is nice... except that... our `question` field isn't meant to hold HTML. It's meant to hold markdown... so this editor doesn't make a lot of sense.

Let's go back to using the `TextareaField`.

68 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 30
    public function configureFields(string $pageName): iterable
    {
... lines 33 - 43
        yield TextareaField::new('question')
... lines 45 - 65
    }
... lines 67 - 68
```

So we don't need a fancy field... but it *would* be really cool if, as we type inside of here, a preview of the final HTML were rendered right below this.

Let's do that! For this to happen, we're going to write some custom JavaScript that will render the markdown. We could also make an Ajax call to render the markdown... it doesn't matter. Either way, we need to write custom JavaScript!

Open up the `webpack.config.js` file. We *do* have a custom `admin` CSS file. Now we're *also* going to need a custom `admin.js` file. So up in the `assets/` directory, right next to the main `app.js` that's included on the frontend, create a new `admin.js` file.

Inside, we're going to import two things. First, import `./styles/admin.css` to bring in our admin styles. And second, import

./bootstrap .

3 lines | assets/admin.js

```
1  import './styles/admin.css';
2  import './bootstrap';
```

This file is also imported by app.js . Its purpose is to start the Stimulus application and load anythingin our controllers/ directory as a Stimulus controller.

If you haven't used Stimulus before, it's not required to do custom JavaScript...it's just the way that *I* like to write custom JavaScript... and I think it's awesome. We have a big tutorial all about it if you want to jump in.

So the admin.js file imports the CSS file and it also initializes the Stimulus controllersNow over in webpack.config.js , we can change this to be a *normal* entrypoint... and point it at ./assets/admin.js .

77 lines | webpack.config.js

```
    ... lines 1 - 8
9   Encore
    ... lines 10 - 23
24      .addEntry('admin', './assets/admin.js')
    ... lines 25 - 73
74  ;
    ... lines 75 - 77
```

The end result is that Encore will now output a built admin.js file and a built admin.css file... since we're import CSS from our JavaScript.

And because we just made a change to the Webpack config file,find the terminal tab that's running Encore,stop it with "control+C" and restart it:

```
yarn watch
```

Perfect! It says that the "admin" entrypoint is outputting an admin.css file *and* an admin.js file. It also splits some of the code into a few other files for performance.

Thanks to this change, if you go refresh any page...and view the page source, yup!We still have a link tag for admin.css but now the admin *JavaScript* is also being included,which is all of this stuff right here.We now have the ability to add*custom* JavaScript.

The Stimulus Controller

So here's the plan. We're going to install a JavaScript markdown parser called snarkdown.Then, as we type into this box, in real time, we'll use it to render an HTML preview below this.And to hook all of this up, we're going to write a Stimulus controller.

Let's start by installing that library.Over in the main terminal tab, run:

```
yarn add snarkdown --dev
```

Excellent! Next, up in assets/controllers/ , create a new file called snarkdown_controller.js . And because this tutorial is*not* a Stimulus tutorial, I'll paste in some contents.

27 lines | assets/controllers/snarkdown_controller.js

```
import { Controller } from '@hotwired/stimulus';
import snarkdown from 'snarkdown';
const document = window.document;

export default class extends Controller {
    static targets = ['input'];

    outputElement = null;

    initialize() {
        this.outputElement = document.createElement('div');
        this.outputElement.className = 'markdown-preview';
        this.outputElement.textContent = 'MARKDOWN WILL BE RENDERED HERE';

        this.element.append(this.outputElement);
    }

    connect() {
        this.render();
    }

    render() {
        const markdownContent = this.inputTarget.value;
        this.outputElement.innerHTML = snarkdown(markdownContent);
    }
}
```

What's inside of here... isn't that important. But to get it to work, we're going to need some custom attributes that will *attach* this controller to the form field. Let's do that next *and* use a performance trick so that our new controller isn't unnecessarily downloaded by frontend users.

Chapter 21: Custom Stimulus JavaScript Controller

We just created a Stimulus controller. *Now* we need to *apply* this controller to the "row" that's around each field. Let me make things a bit smaller. So we're going to apply the controller to this row. The code in the controller will *watch* the textarea for changes and render a preview.

The whole flow looks like this. When that row first appears on the page, the `initialize()` method will add a preview div. Then, whenever we type into the field, Stimulus will call `render()` ... which will render the HTML preview. We're not going to talk more about the Stimulus code, but if you have any questions, let us know in the comments.

Thanks to the fact that `admin.js` is importing `bootstrap.js`, which initializes all of the controllers in the `controllers/` directory, our new `snarkdown_controller` *is* already available in the admin section. So, we can get to work!

On the field, call `setFormTypeOptions()` and pass this an array. We need to set a few attributes. The first is `row_attr`: the attributes that you want to add to the form "row". This is not an Easy Admin thing... it's a normal option inside Symfony's form system. Add a `data-controller` attribute set to `snarkdown`. I *did* just typo that, which is going to *totally* confuse future me.

Next pass an `attr` option: the attributes that should be added the textarea itself. Add one called `data-snarkdown-target` set to `input`. In Stimulus language, this makes the textarea a "target"...so that it's easy for us to find. Also add `data-action` set to `snarkdown#render`.

77 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 14
15  class QuestionCrudController extends AbstractCrudController
16  {
... lines 17 - 30
31      public function configureFields(string $pageName): iterable
32      {
... lines 33 - 43
44          yield TextareaField::new('question')
45              ->hideOnIndex()
46              ->setFormTypeOptions([
47                  'row_attr' => [
48                      'data-controller' => 'snarkdown',
49                  ],
50                  'attr' => [
51                      'data-snarkdown-target' => 'input',
52                      'data-action' => 'snarkdown#render',
53                  ],
54              ]);
... lines 55 - 74
75      }
76  }
```

This says: whenever the textarea changes, call the `render()` method on our `snarkdown` controller.

Let's try this! Move over and refresh... and type a little... hmm. No preview. And no errors in the console either. Debugging time! Inspect the element. Bah! A typo on the controller name... so the controller was never initialized.

Fix that - `snarkdown` - and now when we refresh, there it is! It starts with a preview... and when we type... it instantly updates to show that as bold. Awesome!

Though, we could style this a bit better... and fortunately we know how to add CSS to our admin area. In `admin.css`, add a `.markdown-preview` selector. This is the class that the preview div has when we add it. Let's give this some margin, a border and some padding.

12 lines | assets/styles/admin.css

```
... lines 1 - 6
.markdown-preview {
    margin-top: 10px;
    border: 2px dashed #da3735;
    padding: 5px;
}
```

And now... neato! And to make this *even* cooler, in `QuestionCrudController`, on the field, call `->setHelp('Preview')`.

78 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 14
class QuestionCrudController extends AbstractCrudController
{
... lines 17 - 30
    public function configureFields(string $pageName): iterable
    {
... lines 33 - 43
        yield TextareaField::new('question')
... lines 45 - 54
            ->setHelp('Preview:');
... lines 56 - 75
    }
}
```

Help messages render below the field... so... ah. This gives the preview a little header.

Making Admin Controllers Lazy

So with the combination of Stimulus and an `admin.js` file that imports `bootstrap.js`, we can add custom JavaScript to our admin section simply by dropping a new controller into the `controllers/` directory.

This *does* create one small problem. *Every* file in the `controllers/` directory is *also* registered and packaged into the built `app.js` file for the frontend. This means that users that visit our frontend are downloading `snarkdown_controller` *and* snarkdown itself. That's probably not a security problem... but it *is* wasteful and will slow down the frontend experience.

My favorite way to fix this is to go into the controller and add a superpower that's special to Stimulus inside of Symfony. Put a comment directly above the controller with `stimulusFetch` colon then inside single quotes `lazy`.

28 lines | assets/controllers/snarkdown_controller.js

```
... lines 1 - 4
/* stimulusFetch: 'lazy' */
export default class extends Controller {
... lines 7 - 26
}
```

What does that do? It tells Encore to *not* download this controller code - *or* anything it imports - until the moment that an element appears on the page that matches this controller. In other words, the code *won't* be downloaded immediately. But then, the *moment* a `data-controller="snarkdown"` element appears on the page, it'll be downloaded via Ajax and executed. Pretty perfect for admin stuff.

Check it out. On your browser, go back to the admin section. Pull up your network tools and go to the Questions section. I'll make the tools bigger... then go edit a question. On the network tools filter, click "JS".

Check out this last entry: `assets_controllers_snarkdown_controller_js.js`. *That* is the file that contains our `snarkdown_controller` code. And notice the "initiator" is "load_script". That's a Webpack function that tells me that this was downloaded *after* the page was loaded. Specifically, once the textarea appeared on the page.

And if we visit any *different* page... yep! That file was *not* downloaded at all because there is *no* `data-controller="snarkdown"` element on the page.

Next, it's finally time to do something with our dashboard! Let's render a chart and talk about what other things you can do with

your admin section's landing page.

Chapter 22: The Dashboard Page

We know that, on a technical level, the dashboard is the key to everything. All Crud controllers run in the context of the dashboard that link to them, which allows us to control things on a global level by adding methods to the dashboard controller.

But the dashboard is also... just a page! A page with a controller that's the homepage of our admin. And so, we can - and should - *do* something with that page!

The simplest option is just to redirect to a specific CRUD section...so that when the user goes to `/admin`, they're immediately redirected to, for example, the question admin. In a little while, we'll learn how to generate URLs to specific Crud controllers.

Or to be a little more fun, we can render something *real* on this page. Let's do that: let's render some stats and a chart.

To get the stats that I want to show, we need to query the database. Specifically, we need to query from `QuestionRepository`. `DashboardController` is a normal controller... which means that it's *also* a service. And so, when a service needs access to *other* services, we use dependency injection!

Add a constructor... then autowire `QuestionRepository $questionRepository`. I'll hit Alt+Enter and go to initialize properties to create that property and set it.

95 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 8
use App\Repository\QuestionRepository;
... lines 10 - 22
class DashboardController extends AbstractDashboardController
{
    private QuestionRepository $questionRepository;

    public function __construct(QuestionRepository $questionRepository)
    {
        $this->questionRepository = $questionRepository;
    }
... lines 31 - 93
}
```

If you're wondering why I'm not using *action* injection - where we add the argument to the method - I'll explain why in a few minutes. But it *is* possible.

Before we render a template, let's prepare a few variables: `$latestQuestions` equals `$this->questionRepository->findLatest()`. That's a custom method I added before we started. Also set `$topVoted` to `$this->questionRepository->findTopVoted()`: *another* custom method.

95 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 33
    public function index(): Response
    {
        $latestQuestions = $this->questionRepository
            ->findLatest();
        $topVoted = $this->questionRepository
            ->findTopVoted();
... lines 40 - 44
    }
... lines 46 - 95
```

Finally, at the bottom, like almost *any* other controller, return `$this->render()` to render, how about, `admin/index.html.twig`. Pass in the two variables: `latestQuestions` and `topVoted`.

95 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 33
    public function index(): Response
    {
... lines 36 - 40
        return $this->render('admin/index.html.twig', [
            'latestQuestions' => $latestQuestions,
            'topVoted' => $topVoted,
        ]);
    }
... lines 46 - 95
```

Awesome! Let's go add that! In `templates/admin/`, create a new `index.html.twig` ... and I'll paste in the contents.

32 lines | templates/admin/index.html.twig

```
{% extends '@EasyAdmin/page/content.html.twig' %}

{% block page_title %}
    Cauldron Overflow Dashboard
{% endblock %}

{% block main %}
    <div class="row">
        <div class="col-6">
            <h3>Latest Questions</h3>
            <ol>
                {% for question in latestQuestions %}
                    <li>
                        <a href="{{ path('app_question_show', {'slug': question.slug}) }}">{{ question.name }}</a>
                        <br>- {{ question.createdAt|date }}
                    </li>
                {% endfor %}
            </ol>
        </div>
        <div class="col-6">
            <h3>Top Voted</h3>
            <ol>
                {% for question in topVoted %}
                    <li>
                        <a href="{{ path('app_question_show', {'slug': question.slug}) }}">{{ question.name }}</a> ({{ question.votes }})
                    </li>
                {% endfor %}
            </ol>
        </div>
    </div>
{% endblock %}
```

But there's nothing tricky here. I *am* extending `@EasyAdmin/page/content.html.twig`. If you ever need to render a custom page... but one that still *looks* like it lives inside the admin area, this is the template you want.

If you open it up... hmm, there's not much here! But check out the extends: `ea.templatePath('layout')`. If you look in the `views/` directory of the bundle itself, this is a fancy way of extending `layout.html.twig`. And *this* is a great way to discover all of the different blocks that you can override.

Back in *our* template, the `main` block holds the content, we loop over the latest questions... and the top voted. Very straightforward. And if you refresh the page, instead of the EasyAdmin welcome message, we see our stuff!

Adding a Chart!

Let's have some fun and render a chart on this page. To do this, we'll use a Symfony UX library. At your terminal, run:

```
composer require symfony/ux-chartjs
```

While that's installing, I'll go to the GitHub page for this library and load up its documentation.These days, the docs live on symfony.com and you'll find a link there from here.

Ok, so after installing the library, we need to run:

```
yarn install --force
```

And then... sweet! Just like that, we have a new Stimulus controllerthat has the ability to render a chart via Chart.js.

But I don't want talk too much about this chart library.Instead, we're going to steal the example code from the docs.Notice that we need a service in order to build a chart called `ChartBuilderInterface`. Add that as a second argument to the controller: `ChartBuilderInterface $chartBuilder`. I'll hit Alt+Enter and go to initialize properties to create that property and set it.

127 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 21
22  use Symfony\UX\Chartjs\Builder\ChartBuilderInterface;
... lines 23 - 24
25  class DashboardController extends AbstractDashboardController
26  {
... line 27
28      private ChartBuilderInterface $chartBuilder;
... line 29
30      public function __construct(QuestionRepository $questionRepository, ChartBuilderInterface $chartBuilder)
31      {
... line 32
33          $this->chartBuilder = $chartBuilder;
34      }
... lines 35 - 125
126 }
```

Then, all the way at the bottom...just to keep things clean... create a new private function called `createChart()` ... that will return a `Chart` object. *Now* steal the example code from the docs -everything except for the render - paste it into the method...and, at the bottom `return $chart`.

Oh, and `$chartBuilder` needs to be `$this->chartBuilder`. I'm not going to bother making any of this dynamic:I just want to see that the chart *does* render.

127 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 99
    private function createChart(): Chart
    {
        $chart = $this->chartBuilder->createChart(Chart::TYPE_LINE);
        $chart->setData([
            'labels' => ['January', 'February', 'March', 'April', 'May', 'June', 'July'],
            'datasets' => [
                [
                    'label' => 'My First dataset',
                    'backgroundColor' => 'rgb(255, 99, 132)',
                    'borderColor' => 'rgb(255, 99, 132)',
                    'data' => [0, 10, 5, 2, 20, 30, 45],
                ],
            ],
        ]);

        $chart->setOptions([
            'scales' => [
                'y' => [
                    'suggestedMin' => 0,
                    'suggestedMax' => 100,
                ],
            ],
        ]);

        return $chart;
    }
... lines 126 - 127
```

Back up in the `index()` method, pass a new `chart` variable to the template set to `$this->createChart()`.

127 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 37
    public function index(): Response
    {
... lines 40 - 44
        return $this->render('admin/index.html.twig', [
... lines 46 - 47
            'chart' => $this->createChart(),
        ]);
    }
... lines 51 - 127
```

Finally, to render this, over in `index.html.twig`, add one more `div` with `class="col-12"` ... and, inside, `render_chart(chart)` ... where `render_chart()` is a custom function that comes from the library that we just installed.

35 lines | templates/admin/index.html.twig

```
... lines 1 - 6
{% block main %}
    <div class="row">
... lines 9 - 29
        <div class="col-12">
            {{ render_chart(chart) }}
        </div>
    </div>
{% endblock %}
```

And... that should be it! Find your browser, refresh and... nothing! Um, force refresh? Still nothing. In the console... a big error.

Ok, over in the terminal tab that holds Encore, it wants me to run `yarn install --force` ... which I already did. Hit Ctrl+C to stop Encore... then restart it so that it sees the new files from the UX library:

```
yarn watch
```

And... yes! Build successful. And in the browser... we have a chart!

Next: let's do the *shortest* chapter ever where we talk about the pros, cons and limitationsof injecting services into the *action* methods of your admin controllers versus through the constructor.

Chapter 23: Service Action Injection

You may have noticed that I seem to be *avoiding* "action" injection. For both `QuestionRepository` and `ChartBuilderInterface`, normally, when I'm in a controller, I'll like to be lazy and autowire them directly into the controller *method*.

The Problem with Action Injection

Let's actually try that, at least for `ChartBuilderInterface`. Remove `ChartBuilderInterface` from the constructor... and, instead add it to the method: `ChartBuilderInterface $chartBuilder`.

127 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
class DashboardController extends AbstractDashboardController
{
... lines 27 - 28
    public function __construct(QuestionRepository $questionRepository)
    {
        $this->questionRepository = $questionRepository;
    }
... lines 33 - 35
    public function index(ChartBuilderInterface $chartBuilder = null): Response
    {
... lines 38 - 49
    }
... lines 51 - 125
}
```

And now... I need to pass `$chartBuilder` into `createChart()` ... because, down here we can't reference the property anymore. So add `ChartBuilderInterface $chartBuilder` ... and use that argument.

127 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 35
    public function index(ChartBuilderInterface $chartBuilder = null): Response
    {
... lines 38 - 44
        return $this->render('admin/index.html.twig', [
... lines 46 - 47
            'chart' => $this->createChart($chartBuilder),
        ]);
    }
... lines 51 - 99
    private function createChart(ChartBuilderInterface $chartBuilder): Chart
    {
        $chart = $chartBuilder->createChart(Chart::TYPE_LINE);
... lines 103 - 124
    }
... lines 126 - 127
```

Cool. So in theory, this should work... because this is a normal controller and... this is how action injection works! But you might already notice that PhpStorm is pretty mad. And, it's right! If we refresh, huge error!

> `DashboardController::index` must be compatible with `AbstractDashboardController::index`.

The problem is that our parent class - `AbstractDashboardController` - has an `index()` method with no arguments. So it's not *legal* for us to override that and add a *required* argument.

The Workaround

But if you *do* want action injection to work, there *is* a workaround: allow the argument to be optional. So add `= null`.

That makes PHP happy *and*, in practice, even though it's optional, Symfony *will* pass the chart builder service. So this *will* work... but to code defensively just in case, I'm going to add a little `assert()` function.

This may or may not be a function you're familiar with. It comes from PHP itself. You put an expression inside like `null !== $chartBuilder` - and if that expression is *false*, an exception will be thrown.

127 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 35
36      public function index(ChartBuilderInterface $chartBuilder = null): Response
37      {
38          assert(null !== $chartBuilder);
... lines 39 - 49
50      }
... lines 51 - 127
```

So now we can *confidently* know that *if* our code gets this far, we *do* have a `ChartBuilderInterface` object.

Refresh now and... got it! So action injection *does* still work... but it's not *as* awesome as it normally is. Though, it *does* have one concrete advantage over constructor injection: the `ChartBuilderInterface` service won't be instantiated unless the `index()` method is called. So if you were in a normal Crud controller with multiple actions, action injection allows you to make sure that a service is *only* instantiated for the action that *needs* it, instead of in *all* situations.

Next: let's learn how to override templates, like EasyAdmin's layout template, or how an `IdField` is rendered across our entire admin area.

Chapter 24: Override all the Templates!

Everything you see in EasyAdmin, from the layout of this table,to even how each individual field is rendered, is controlled by a template. EasyAdmin has a *bunch* of templates and we can override *all* of them!

We looked at these a bit earlier. Let's dive back into `vendor/easycorp/easyadmin-bundle/src/Resources/views/`. There's a *lot* of good stuff here, like `layout.html.twig`. This is the base layout file for *every* page. A few minutes ago, we also saw `content.html.twig`. This is a nice layout template that you can extend if you're creating a custom page inside of EasyAdmin.

In `crud/`, we see the templates for the individual pages and in `field/`, there's a template that controls how *every* field type is rendered.

In many of these templates, you'll see things like `ea.templatePath('layout')`. To understand that more deeply, hit "shift+shift" and open up a core class that we explored earlier called `TemplateRegistry`.

Internally, EasyAdmin maintains a map from a template "name" to an actual template path.So when you see something like `ea.templatePath('layout')`, that's going to use `TemplateRegistry` to figure out to load `@EasyAdmin/layout`, where `@EasyAdmin` is a Twig alias that points to this `views/` directory.

Overriding a Core Template

With that in mind, here's our goal: I want to add a footer to the bottom of *every* page. Look again at `layout.html.twig`. Near the bottom, let's see... search for "footer". There we go! This has a block called `content_footer`. So if you *define* a `content_footer`, then it will get dumped right here. Let's override the `layout.html.twig` template and do that!

There are two ways to override a template. The first is to use Symfony's normal system for overriding templates that live inside of a bundle. You do that by creating a file in a very specific path.Inside of `templates/`, create a directory called `bundles/` ... and inside of that, another directory with the name of the bundle: `EasyAdminBundle`. Now, match whatever path from the bundle that you want to override. Since we want to override `layout.html.twig`, create a new file called `layout.html.twig`.

But, hmm. I don't really want to override this *entirely*. I want to *extend* it. And, we can do that! Add `{% extends %}`, with `@EasyAdmin/layout.html.twig`. The only problem is that, by creating a `templates/bundles/EasyAdminBundle/layout.html.twig` file, when Symfony looks for the `@EasyAdmin/layout.html.twig` template, it will now use *our* file! In other words, we're extending *ourselves*!

To tell Symfony that we want to use the *original* template, add an exclamation point in front.

6 lines | templates/bundles/EasyAdminBundle/layout.html.twig

```
1  {% extends '@!EasyAdmin/layout.html.twig' %}
   ... lines 2 - 6
```

Perfect! Below, add `{% block content_footer %}` and `{% endblock %}` ... with a nice little message inside.

6 lines | templates/bundles/EasyAdminBundle/layout.html.twig

```
   ... lines 1 - 2
3  {% block content_footer %}
4      Made with  by the guys and gals at SymfonyCasts
5  {% endblock %}
```

Let's try it! Refresh *any* page and... hello footer!

Overriding a Field Template

So that's the *first* way to override a template. The second is even *more* powerful because it allows us to control exactly *when* we want our overridden templates to be used. Like, you can override a template across your entire admin *or* just for one crud section. Let me close a few files.

So for our next trick, let's override the template that's used to render id fields.We're going to add a little key icon next to the ID.

Open up `IdField.php`. Ok, it sets its template name to `crud/field/id`. In the template registry... here it is... that corresponds to this template. So the template that renders `IdField` is `Resources/views/crud/field/id.html.twig`.

Instead of using the *first* method to override this - which *would* work - let's do something different.

Copy this template and create our override template... which could live *anywhere*. How about in `templates/admin/field/` ... and call it `id_with_icon.html.twig`. Paste the contents... and I'll put the icon right before the id.

6 lines | templates/admin/field/id_with_icon.html.twig

```
{# @var ea \EasyCorp\Bundle\EasyAdminBundle\Context\AdminContext #}
{# @var field \EasyCorp\Bundle\EasyAdminBundle\Dto\FieldDto #}
{# @var entity \EasyCorp\Bundle\EasyAdminBundle\Dto\EntityDto #}
{# this template is used to display Doctrine entity primary keys #}
<span class="fa fa-key"></span> {{ field.formattedValue }}
```

At this moment, this will not, *yet* be used. To activate it globally, go to `DashboardController`: you can configure template override paths down in `configureCrud()`. Check it out: `->overrideTemplate()` where the first argument is the *name* of the template: that's the thing you see inside `TemplateRegistry` or `IdField`. So whenever EasyAdmin renders `crud/field/id`, we'll now have it point to `admin/crud/field/id_with_icon.html.twig`.

128 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
class DashboardController extends AbstractDashboardController
{
... lines 27 - 79
    public function configureCrud(): Crud
    {
        return parent::configureCrud()
... lines 83 - 85
            ->overrideTemplate('crud/field/id', 'admin/field/id_with_icon.html.twig');
    }
... lines 88 - 126
}
```

How cool is that? Let's try it! Refresh and... whoops... let me get rid of my extra `crud/` path. *Now* let's try it! And... yes! Awesome! We see the key icon across the entire system.

Re-Using the Parent Template

But we can make this even better. The id template is super simple... since it just prints the formatted value. But *sometimes* the original template might do more complex stuff. Instead of repeating all of that in *our* template, we can *include* the original template. So quite literally `include()` ... and I'll start typing `id.html.twig` ... and let that autocomplete.

7 lines | templates/admin/field/id_with_icon.html.twig

```
... lines 1 - 4
<span class="fa fa-key"></span>
{{ include('@EasyAdmin/crud/field/id.html.twig') }}
```

At the browser... we get the same result.

Next, let's talk about permissions: How we can deny access to entire CRUD controllers or specific actions based on the user's role.

Chapter 25: Permissions

In `config/packages/security.yaml`, thanks to the `access_control` that we added *way* back at the start of the tutorial, you can only get to the admin section if you have `ROLE_ADMIN`. As far as security goes.... that's all we have so far. If you have `ROLE_ADMIN`, you get access to *everything* inside of the admin area. Lucky you!

But in this app, there needs to be *three* different admin user types and each will have access to different *parts* of the admin section.

You can see these described up under `role_hierarchy`. We have `ROLE_ADMIN`, which is the lowest level. Then we `ROLE_MODERATOR` above that, which *includes* `ROLE_ADMIN`, but we're going to give this some *extra* access, like the ability to moderate questions. And finally, there's `ROLE_SUPER_ADMIN`, which is the highest level of permissions and will be allowed to do everything.

Hiding a Menu Link by Role

Here's the first goal: only users with `ROLE_MODERATOR` should be allowed to go to the Questions CRUD section. Right now, if I hover over the security part of the web debug toolbar... yup! I *only* have `ROLE_ADMIN` ... so I should *not* be able to go here.

Fixing this is two steps. First, we need hide this link unless the user has `ROLE_MODERATOR`. Open up `DashboardController` ... and find `configureMenuItems()`: this is where we configure those links. On the `Questions` link, add `->setPermission()` and then pass the role that's required: `ROLE_MODERATOR`.

129 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
25  class DashboardController extends AbstractDashboardController
26  {
... lines 27 - 57
58      public function configureMenuItems(): iterable
59      {
... line 60
61          yield MenuItem::linkToCrud('Questions', 'fa fa-question-circle', Question::class)
62              ->setPermission('ROLE_MODERATOR');
... lines 63 - 66
67      }
... lines 68 - 127
128 }
```

Since the user *I'm* logged in as does *not* have this role... when we refresh, the link disappears.

The ?signature In the URL

> Starting in EasyAdmin v4.1.0, URLs will *no longer* contain these "signatures". You don't need to make any changes, but you can read more about the decision at: https://github.com/EasyCorp/EasyAdminBundle/issues/5018

But, of course, I still *technically* have access to this section! The link is gone, but if someone sent me this URL, then I *could* still access this. So that *is* still a problem. Though, at the very least, a user wouldn't be able to *guess* the URL, because EasyAdmin generates a signature. That's this `signature=''` part. What that does is prevent anyone from messing with a URL and trying to access something else. For example, if I tried to change "QuestionCrudController"to "AnswerCrudController" to be sneaky and gain access to another section, I see:

> The signature of the URL is not valid.

So without the link to Questions, there won't be a way for me to somehow *guess* the URL. *But* if somebody just sends me the link, I do still technically have access. We'll fix that in a second.

By the way, if you want to disable that signature feature in your admin section,that can be done in `configureDashboard()` by calling `->disableUrlSignatures()`. Just be *extra* careful that you have your security configured correctly.

Restricting a Crud Section By Role

Anyways, to *truly* restrict access to this CRUD section, go to `QuestionCrudController`. In EasyAdmin language, what we need to do is set a permission on the *action* or *actions* that should require that role. We don't have a `configureActions()` method yet, so I'll go to "Override Methods" to add it.

What we've been doing so far is adding and disabling actions on certain pages.We can *also* call `->setPermission()` and pass an action name - like `Action::INDEX` and the role you need to have: `ROLE_MODERATOR`.

86 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 8
use EasyCorp\Bundle\EasyAdminBundle\Config\Actions;
... lines 10 - 16
class QuestionCrudController extends AbstractCrudController
... lines 18 - 32
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
            ->setPermission(Action::INDEX, 'ROLE_MODERATOR');
    }
... lines 38 - 84
}
```

If I refresh the index page now... it fails!

> You don't have enough permissions to run the "index" action

Now go to the Homepage... log out... and log *back* in as "moderatoradmin@example.com" with password "adminpass".Cool. This user has `ROLE_MODERATOR`. Head back to the Admin section...and *now* we *do* see the Questions link... and we can access the Questions section. Sweet!

However, we only restricted access to the*index* action. So the same problem applies to the other actions: if someone sent me the URL to the "new" or "edit" pages, then I *will* be able to access those... as long as I have the minimum `ROLE_ADMIN`.

So, let's lock down a couple more actions: the `DETAIL` action for `ROLE_MODERATOR` and also the `EDIT` action for `ROLE_MODERATOR`. In a few minutes, we'll learn how to restrict access to an*entire* CRUD controller. What we're doing should *only* be needed if you need to restrict things differently on an action-by-action basis.

88 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 32
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
            ->setPermission(Action::INDEX, 'ROLE_MODERATOR')
            ->setPermission(Action::DETAIL, 'ROLE_MODERATOR')
            ->setPermission(Action::EDIT, 'ROLE_MODERATOR');
    }
... lines 40 - 88
```

Ok, let's think. The only two actions that we haven't listed yet are `NEW` and `DELETE`. Those are pretty sensitive, so I only want to allow super admins to access those. Copy this, paste, and say `Action::NEW` restricted to `ROLE_SUPER_ADMIN`. Paste again and say `Action::DELETE` *also* restricted to `ROLE_SUPER_ADMIN`.

90 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 32
33      public function configureActions(Actions $actions): Actions
34      {
35          return parent::configureActions($actions)
... lines 36 - 38
39              ->setPermission(Action::NEW, 'ROLE_SUPER_ADMIN')
40              ->setPermission(Action::DELETE, 'ROLE_SUPER_ADMIN');
41      }
... lines 42 - 90
```

Thanks to these changes, when we refresh...yes! It hides the delete link correctly. And even if I were able to guess the URL to that action, I wouldn't be able to get there. Oh, but EasyAdmin has a really nice "batch delete"...and that *is* still allowed. Let's lock that down as well.

Paste another line, change this to `BATCH_DELETE` with `ROLE_SUPER_ADMIN`. Now when we refresh, the check boxes are gone! I have no batch actions that I can do on this page.

91 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 32
33      public function configureActions(Actions $actions): Actions
34      {
35          return parent::configureActions($actions)
... lines 36 - 40
41              ->setPermission(Action::BATCH_DELETE, 'ROLE_SUPER_ADMIN');
42      }
... lines 43 - 91
```

Next, sometimes permissions are... *not* this complex! Let's learn how we can restrict access to an *entire* crud section with one line of code.

Chapter 26: Restricting Access to an Entire Crud Section

So... great! We can now restrict things on an action-by-action basis. But *sometimes*... it's not that complicated! *Sometimes* you just want to say:

> I want to require `ROLE_MODERATOR` to be able to access *any* part of a CRUD section as a whole.

In that case, instead of trying to set permissions on *every* action like this, you can be lazy and use *normal* security.

For example, head to the top of `QuestionCrudController`. Above the class, leverage the `#[IsGranted]` attribute from SensioFrameworkExtraBundle. Just for a minute, let's pretend that we're going to require `ROLE_SUPER_ADMIN` to use *any* part of this section.

93 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 15
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

#[IsGranted('ROLE_SUPER_ADMIN')]
class QuestionCrudController extends AbstractCrudController
... lines 20 - 93
```

If we move over now and refresh..."Access Denied"! Yea, since these controllers are *real* controllers, just about everything that works in a normal controller also works inside of these CRUD controllers.

Let's undo that. *Or*, if you want, we can put `ROLE_MODERATOR` up there to make sure that if we missed any actions, users will *at least* need to have `ROLE_MODERATOR`. Since we're already logged in with that user, now...we're good!

93 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 17
#[IsGranted('ROLE_MODERATOR')]
class QuestionCrudController extends AbstractCrudController
... lines 20 - 93
```

Make sure Permissions Match Link Permissions

One thing I *do* want to point out is that, when you link to something, you *do* need to keep the permissions on that link "in sync" with the permissions for the controller you're linking to.

For example, let's temporarily remove the link permission for the menu item.

129 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
class DashboardController extends AbstractDashboardController
{
... lines 27 - 57
    public function configureMenuItems(): iterable
    {
... line 60
        yield MenuItem::linkToCrud('Questions', 'fa fa-question-circle', Question::class);
            //->setPermission('ROLE_MODERATOR');
... lines 63 - 66
    }
... lines 68 - 127
}
```

Then, in `QuestionCrudController`, down on index, temporarily require `ROLE_SUPER_ADMIN`. This means that we should *not* have

access.

93 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 18
class QuestionCrudController extends AbstractCrudController
{
... lines 21 - 34
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
            ->setPermission(Action::INDEX, 'ROLE_SUPER_ADMIN')
... lines 39 - 43
    }
... lines 45 - 91
}
```

And if we move over and refresh...that's true! We're denied access! But go back to `/admin`. Uh oh: the Questions link *does* show up. EasyAdmin isn't smart enough to realize that if we clicked this, we wouldn't have access! It's *our* responsibility to make sure that the permissions on our link are set up correctly.

Go change this back to `ROLE_MODERATOR` ... and over here, we'll restore that permission. Now we're good. Our question section requires `ROLE_MODERATOR` and specific actions inside of it, like `DELETE`, require `ROLE_SUPER_ADMIN`.

Nice work team!

But security can go even further! Next let's hide individual fields based on permissions and even hide specific entity *results* based on which admin user is logged in. Whoa...

Chapter 27: Entity & Field Permissions

Most of the time, securing your admin will probably mean denying access to entire sections *or* specific actions based on a role. *But* we can go a lot further.

Hiding a Field for some Admins

Let's imagine that, for some reason, the number of votes is a sensitive number that should *only* be displayed and modified by super admins. Head over to `QuestionCrudController`. This is something we can control on each field, so find `VotesField`. Here it is. Add `->setPermission()` and then pass `ROLE_SUPER_ADMIN`.

94 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 18
class QuestionCrudController extends AbstractCrudController
{
... lines 21 - 45
    public function configureFields(string $pageName): iterable
    {
... lines 48 - 70
        yield VotesField::new('votes', 'Total Votes')
... line 72
            ->setPermission('ROLE_SUPER_ADMIN');
... lines 74 - 91
    }
}
```

I'm currently logged in as "moderatoradmin", so I'm *not* a super admin. And so, when I refresh, it's as simple as that! The votes field disappears, both on the list page *and* on the edit page. Super cool!

Hiding some Results for some Admins

Ok, let's try something different. What if we want to show only *some* items in an admin section based on the user? Maybe, for some reason, my user can only see *certain* questions.

Or, here's a better example. I'm currently logged in as a moderator, whose job is to *approve* questions. If we click the Users section, a moderator *probably* shouldn't be able to see and edit *other* user accounts. We could hide the section entirely for moderators, *or* we could add some security so that only their *own* user account is visible to them. This is called "entity permissions". It answers the question of whether or not to show a specific *row* in an admin section based on the current user. And we control this on the CRUD level: we set an entity permission for an *entire* CRUD section.

Head over to `UserCrudController` and, at the bottom, override the `configureCrud()` method. And now, for this entire CRUD, we can say `->setEntityPermission()` and pass `ADMIN_USER_EDIT`.

65 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 5
use EasyCorp\Bundle\EasyAdminBundle\Config\Crud;
... lines 7 - 17
class UserCrudController extends AbstractCrudController
{
... lines 20 - 24
    public function configureCrud(Crud $crud): Crud
    {
        return parent::configureCrud($crud)
            ->setEntityPermission('ADMIN_USER_EDIT');
    }
... lines 30 - 63
}
```

Notice this is *not* a role. EasyAdmin calls the security system for *each* entity that it's about to display and passes this `ADMIN_USER_EDIT` string *into* the security system. If we used a role here - like `ROLE_SUPER_ADMIN` - that would return true or false for *every* item. It would end up showing either *all* the items or *none* of them.

Nope, a role won't work here. So, instead, I'm passing this `ADMIN_USER_EDIT` string, which is something I *totally* just invented. In a few minutes, we're going to create a custom voter to handle that.

But since we *haven't* created that voter yet, this will return false in the security system in *all* cases. In other words, if this is working correctly, we won't see *any* items in this list.

Entity Permissions and formatValue()

Let's try it! Refresh and... okay. We don't see any items in the list, but it's because we have a gigantic error. It's coming from `UserCrudController`: the `formatValue()` callback on the avatar field:

> Argument #2 ($user) must be of type `App\Entity\User`, null given

This error originates in a configurator. Go look at that field. Let's see... avatar... here it is. You might remember that `formatValue()` is the way we control how a value is *rendered* on the index and detail pages. And it's simple: it passes us the current `User` object - since we're in the `UserCrudController` and rendering users - and then we return whatever value we want.

But, when you use entity permissions, it's possible that this `User` object will be `null` because this is a row that *won't* be displayed. I'm not sure exactly *why* EasyAdmin calls our callback... even though the row is about to be hidden, but it *does*. So it means that we need to allow this to be `null`. I'll add a question mark to make it nullable.

And then, because we're using PHP 8, we can be super trendy by using a new syntax: `$user?->getAvatarUrl()`. That says that *if* there is a user, call `->getAvatarUrl()` and return it. Else, just return `null`.

65 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 30
31      public function configureFields(string $pageName): iterable
32      {
... lines 33 - 34
35          yield AvatarField::new('avatar')
36              ->formatValue(static function ($value, ?User $user) {
37                  return $user?->getAvatarUrl();
38              })
... lines 39 - 62
63      }
... lines 64 - 65
```

There's one other place that we need to do this. It's in `QuestionCrudController`, down here on the `askedBy` field. Add a question mark, and then another question mark right in the middle of `$question?->getAskedBy()`.

94 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 18
class QuestionCrudController extends AbstractCrudController
{
... lines 21 - 45
    public function configureFields(string $pageName): iterable
    {
... lines 48 - 73
        yield AssociationField::new('askedBy')
... line 75
            ->formatValue(static function ($value, ?Question $question): ?string {
                if (!$user = $question?->getAskedBy()) {
                    return null;
                }
... lines 80 - 81
            })
... lines 83 - 91
    }
}
```

Go refresh again and... beautiful! No results are showing, and we get this nice message:

> Some results can't be displayed because you don't have enough permissions.

Woo! And of course, if we tried to search for something, that would also take into account our permissions.

Next, let's create the voter so that we can deny access *exactly* when we want to and ultimately show only *our* user record when a moderator is in the Users section.

Chapter 28: Security Voter & Entity Permissions

Thanks to `->setEntityPermission()`, EasyAdmin now runs *every* entity in this list through the security system, passing `ADMIN_USER_EDIT` for each. If we were running this security check manually in a normal Symfony app, it would be the equivalent of `$this->isGranted('ADMIN_USER_EDIT')`, where you pass the actual entity object - the `$user` object - as the second argument.

Right now, when we do that, security always returns false because...I just invented this `ADMIN_USER_EDIT` string. To run our custom security logic, we need a voter.

Creating The Voter

Find your terminal and run:

```
symfony console make:voter
```

I'll call it "AdminUserVoter". Perfect! Spin over and open this: `src/Security/Voter/AdminUserVoter.php`. I'm not going to talk *too* deeply about how voters work: we talk about those in our Symfony Security tutorial. But basically, the `supports()` method will be called *every* time the security system is called. The first argument will be something like `ROLE_ADMIN` or, in our case, `ADMIN_USER_EDIT`. And *also*, in our case, `$subject` will be the `User` object. Our job is to return true in that situation.

42 lines | src/Security/Voter/AdminUserVoter.php

```
... lines 1 - 4
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\User\UserInterface;

class AdminUserVoter extends Voter
{
    protected function supports(string $attribute, $subject): bool
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['POST_EDIT', 'POST_VIEW'])
            && $subject instanceof \App\Entity\AdminUser;
    }
... line 18
    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
    {
... lines 21 - 39
    }
}
```

So let's check to see if the attribute is in an array with just `ADMIN_USER_EDIT`. I don't really need `in_array()` anymore, but I'll keep it in case I add more attributes later. Also check to make sure that `$subject` is an `instanceof User`.

40 lines | src/Security/Voter/AdminUserVoter.php

```
... lines 1 - 7
use Symfony\Component\Security\Core\User\UserInterface;
... line 9
class AdminUserVoter extends Voter
{
    protected function supports(string $attribute, $subject): bool
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['ADMIN_USER_EDIT'])
            && $subject instanceof User;
    }
... lines 19 - 38
}
```

That's it! Now, when the security system calls `supports()`, *if* we return true, then Symfony will call `voteOnAttribute()`. Our job there is simply to return true or false based on whether or not the current user should have access to this `User` object in the admin.

Once again, we're passed the `$attribute`, which will be `ADMIN_USER_EDIT`, and `$subject`, which will be the `User` object. To help my editor, add an extra "if" statement: `if (!$subject instanceof User)`, then throw a new `LogicException('Subject is not an instance of User?')`.

40 lines | src/Security/Voter/AdminUserVoter.php

```
... lines 1 - 19
    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
    {
... lines 22 - 26
        if (!$subject instanceof User) {
            throw new \LogicException('Subject is not an instance of User?');
        }
... lines 30 - 37
    }
... lines 39 - 40
```

This should never happen, but that'll help my editor or static analysis. Finally, down in the `switch` (we only have one `case` right now), if that attribute is equal to `ADMIN_USER_EDIT`, then we want to allow access if `$user === $subject`. So if the currently-authenticated `User` object - that's what this is here - is equal to the `User` object that we're asking about for security, then grant access. Otherwise, deny access.

40 lines | src/Security/Voter/AdminUserVoter.php

```
... lines 1 - 19
    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
    {
... lines 22 - 30
        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case 'ADMIN_USER_EDIT':
                return $user === $subject;
        }

        return false;
    }
... lines 39 - 40
```

Symfony will instantly know to use our voter thanks to auto configuration. So when we refresh... got it! We *just* see our one user and the message:

> Some results can't be displayed because you don't have enough permissions.

Awesome! If you go down to the web debug toolbar, click the security icon, and then click "Access Decision", this shows you

all the security decisions that were made during that request. It looks like `ADMIN_USER_EDIT` was called multiple times for the multiple rows on the page. With this user object - access was denied...and with this other user object - that's us - access was granted.

Entity permissions are also enforced when you go to the detail, edit, or delete pages. Again, if you go down to the web debug toolbar and click "Access Decision", at the bottom... you can see it checked for `ADMIN_USER_EDIT`.

Granting Access to ROLE_SUPER_ADMIN

This is great! Except that super admins should be able to see *all* users. Right now, no matter who I log in as, we're only going to show *my* user. To solve this, down in our logic, we can check to see if the user has `ROLE_SUPER_ADMIN`. But to do *that*, we need a service.

Add `public function __construct()`, and inject the `Security` service from Symfony (I'll call it `$security`). Hit "alt" + "enter", and go to "Initialize properties" to create that property and set it. Then, down here, return true if `$user === $subject` *or* if `$this->security->isGranted('ROLE_SUPER_ADMIN')`.

48 lines | src/Security/Voter/AdminUserVoter.php

```
... lines 1 - 7
use Symfony\Component\Security\Core\Security;
... lines 9 - 10
class AdminUserVoter extends Voter
{
    private Security $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }
... lines 19 - 27
    protected function voteOnAttribute(string $attribute, $subject, TokenInterface $token): bool
    {
... lines 30 - 39
        switch ($attribute) {
            case 'ADMIN_USER_EDIT':
                return $user === $subject || $this->security->isGranted('ROLE_SUPER_ADMIN');;
        }
... lines 44 - 45
    }
}
```

Cool! I won't bother logging in as a super admin to try this. But if we *did*, we would now see *every* user.

Adding Permissions Logic to the Query

So there's just one *tiny* problem with our setup. Imagine that we have a lot of users - like *thousands* - which is pretty realistic. And *our* user is ID 500. In that case, you would actually see *many* pages of results here. And our user *might* be on page 200. So you'd see no results on page one... or two... or three... until finally, on page 200, you'd find our *one* result. So it can get a little weird if you have *many* items in an admin section, and *many* of them are hidden.

To fix this, we can modify the query that's made for the index page to *only* return the users we want. This is totally optional, but can make for a better user experience.

So far, we've been letting EasyAdmin query for *every* user or *every* question. But we *do* have control over that query. Open up `UserCrudController` and, anywhere, I'll go near the top, override a method from the base controller called `createIndexQueryBuilder()`.

75 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 5
use Doctrine\ORM\QueryBuilder;
... lines 7 - 22
class UserCrudController extends AbstractCrudController
{
... lines 25 - 35
    public function createIndexQueryBuilder(SearchDto $searchDto, EntityDto $entityDto, FieldCollection $fields, FilterCollection $filters): Que
    {
        return parent::createIndexQueryBuilder($searchDto, $entityDto, $fields, $filters);
    }
... lines 40 - 73
}
```

Here's how this works: the parent method starts the query builder for us. And it already takes into account things like the Search on top or "filters", which we'll talk about in a few minutes.

Instead of returning this query builder, set it to `$queryBuilder`. Then, because super admins should be able see *everything*, if `$this->isGranted('ROLE_SUPER_ADMIN')`, then just return the unmodified `$queryBuilder` so that *all* results are shown.

85 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 35
    public function createIndexQueryBuilder(SearchDto $searchDto, EntityDto $entityDto, FieldCollection $fields, FilterCollection $filters): Que
    {
        $queryBuilder = parent::createIndexQueryBuilder($searchDto, $entityDto, $fields, $filters);

        if ($this->isGranted('ROLE_SUPER_ADMIN')) {
            return $queryBuilder;
        }
... lines 43 - 48
    }
... lines 50 - 85
```

But if we *don't* have `ROLE_SUPER_ADMIN`, that's where we want to change things. Add `$queryBuilder->andWhere()`. Inside the query, the alias for the entity will always be called "entity". So we can say `entity.id = :id` and `->setParameter('id', $this->getUser()->getId())`. I don't get the auto complete on this because it thinks my user is just a `UserInterface`, but we know this *will* be our `User` entity which *does* have a `getId()` method. At the bottom, `return $queryBuilder`. And... I guess I could have just returned that right here... so let's do that.

85 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 35
    public function createIndexQueryBuilder(SearchDto $searchDto, EntityDto $entityDto, FieldCollection $fields, FilterCollection $filters): Que
    {
... lines 38 - 43
        $queryBuilder
            ->andWhere('entity.id = :id')
            ->setParameter('id', $this->getUser()->getId());

        return $queryBuilder;
    }
... lines 50 - 85
```

I love it! Let's try it! Spin over and... nice! Just our *one* result. And you don't see that message about results being hidden due to security... because, *technically*, *none* of them were hidden due to security. They were hidden due to our query. But regardless, permissions are *still* being enforced. If a user somehow got the edit URL to a User that they're not supposed to be able to access, the entity permissions will *still* deny that.

Next, each CRUD section has a nice search box on top. Yay! But EasyAdmin *also* has a great filter system where you can add more ways to slice and dice the data in each section. Let's explore those.

Chapter 29: The Filter System

Let's go log out... and then log *back* in as our "super admin" user: "superadmin@example.com"... with "adminpass". Now head back to the admin area, find the Users list and... perfect! As promised, we can see *every* user in the system.

Our user list is pretty short right now, but it's going to get longer and longer as people realize... just how amazing our site is. It would be *great* if we could filter the records in this index section by some criteria... for example, to only show users that are enabled or *not* enabled. Fortunately, EasyAdmin has a system for this called, well, filters!

Hello configureFilters()

Over in `UserCrudController`, I'll go to the bottom and override yet another method called `configureFilters()`.

91 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 9
use EasyCorp\Bundle\EasyAdminBundle\Config\Filters;
... lines 11 - 23
class UserCrudController extends AbstractCrudController
{
... lines 26 - 85
    public function configureFilters(Filters $filters): Filters
    {
        return parent::configureFilters($filters);
    }
}
```

This looks and feels a lot like `configureFields()`: we can call `->add()` and then put the name of a field like `enabled`.

92 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 85
    public function configureFilters(Filters $filters): Filters
    {
        return parent::configureFilters($filters)
            ->add('enabled');
    }
... lines 91 - 92
```

And... that's all we need! If we refresh the page, watch this section around the top. We have a new "Filters" button! That opens up a modal where we can filter by whichever fields are available. Let's say "Enabled", "No" and... *all* of these are gone because *all* of our users *are* enabled.

We can go and change that... or clear the filter entirely.

Filter Types

Ok: notice that `enabled` in our entity is a boolean field... and EasyAdmin detected that. It knew to make this as a "Yes" or "No" checkbox. Just like with the *field* system, there are also many different types of *filters*. And if you just add a filter by saying `->add()` and then the property name, EasyAdmin tries to guess the correct filter type to use.

But, you *can* be explicit. What we have now is, in practice, identical to saying `->add(BooleanFilter::new('enabled'))`.

93 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 86
    public function configureFilters(Filters $filters): Filters
    {
        return parent::configureFilters($filters)
            ->add(BooleanFilter::new('enabled'));
    }
... lines 92 - 93
```

When we refresh now... and check the filters... that makes no difference because that was already the filter type it was guessing.

Each filter class controls how that filter looks in the form up here, and *also* how it modifies the *query* for the page. Hold cmd or ctrl and open the `BooleanFilter` class. It has a `new()` method just like fields, and this sets some basic information: the most important being the form type and any form type options.

The `apply()` method is the method that will be called when the filter is *applied*: it's where the filter *modifies* the query.

Filter Form Type Options

Back in `new()`, this uses a form field called `BooleanFilterType`. Hold cmd or ctrl to open *that*. Like all form types, this exposes a bunch of options that allow us to control its behavior. Apparently there's an `expanded` option, which is the reason that we're seeing this field as *expanded* radio buttons.

Just to see if we can, let's try changing that. Close that file... and after the filter, add `->setFormTypeOption('expanded', false)`.

93 lines | src/Controller/Admin/UserCrudController.php

```
... lines 1 - 86
    public function configureFilters(Filters $filters): Filters
    {
        return parent::configureFilters($filters)
            ->add(BooleanFilter::new('enabled')->setFormTypeOption('expanded', false));
    }
... lines 92 - 93
```

Try it now: refresh... head to the filters and... awesome! The non-expanded version means it's rendered as a dropdown.

The Many Filter Type Classes

Let's add some filters to the Questions section. Open `QuestionCrudController` and, near the bottom, override `configureFilters()`. Start with an entity relation. Each question has a `ManyToOne` relationship to `Topic`, so let's `->add('topic')`.

101 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 10
use EasyCorp\Bundle\EasyAdminBundle\Config\Filters;
... lines 12 - 19
class QuestionCrudController extends AbstractCrudController
{
... lines 22 - 94
    public function configureFilters(Filters $filters): Filters
    {
        return parent::configureFilters($filters)
            ->add('topic');
    }
}
```

Go refresh. We get the new filter section... and "Topic" is... this cool dropdown list where we can select whatever topic we want!

To know how you can control this - or any - filter, you need to know what *type* it is. Just like with fields, if you click on the filter class, you can see there's a `src/Filter/` directory deep in the bundle. So `vendor/easycorp/easyadmin-bundle/src/Filter/` ... and here is the full list of all possible filters.

I bet `EntityFilter` is the filter that's being used for the relationship. By opening this up, we can learn about any methods it might have that will let us configure it *or* how the query logic is done behind the scenes.

Let's add a few more filters, like `createdAt` ... `votes` ... and `name` .

104 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 94
    public function configureFilters(Filters $filters): Filters
    {
        return parent::configureFilters($filters)
            ->add('topic')
            ->add('createdAt')
            ->add('votes')
            ->add('name');
    }
... lines 103 - 104
```

And... no surprise, those all show up! The coolest thing is what they look like. The `createdAt` field has a really easy way to choose dates, or even filter *between* two dates. For Votes, you can choose "is equal", "is greater than", "is less than", etc. And Name has different types of fuzzy searches that you can apply. *Super* powerful.

We can also create our *own* custom filter class. That's as easy as creating a custom class, making it implement `FilterInterface` , and using this `FilterTrait` . *Then* all you need to do is implement the `new()` method where you set the form type and then the `apply()` method where you modify the query.

Ok, right now, we have one "crud controller" per entity. But it's *totally* legal to have *multiple* CRUD controllers for the *same* entity: you may have a situation where each section shows a different filtered list. But even if you don't have this use-case, adding a second CRUD controller for an entity will help us dive deeper into how EasyAdmin works. That's next.

Chapter 30: Multiple Cruds for a Single Entity?

Right now, we have one CRUD controller per entity. But we *can* create *more* than one CRUD controller for the *same* entity. Why would this be useful? Well, for example, we're going to create a separate "Pending Approval" questions sectionthat *only* lists questions that need to be approved.

Ok, so, we need a new CRUD controller. Instead of generating it this time, let's create it by hand. Call the class `QuestionPendingApprovalCrudController`. We're making this by hand because, instead of extending the normal base classfor a CRUD controller, we'll extend `QuestionCrudController`. That way, it inherits *all* the normal `QuestionCrudController` config and logic.

8 lines | src/Controller/Admin/QuestionPendingApprovalCrudController.php

```
... lines 1 - 2
namespace App\Controller\Admin;
... line 4
class QuestionPendingApprovalCrudController extends QuestionCrudController
{
}
```

Linking to the Controller and setController()

Done! Step two: whenever we add a new CRUD controller,we need to link to it from our dashboard. Open `DashboardController` ... duplicate the question menu item... say "Pending Approval"... and I'll tweak the icon.

132 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
class DashboardController extends AbstractDashboardController
{
... lines 27 - 57
    public function configureMenuItems(): iterable
    {
... lines 60 - 62
        yield MenuItem::linkToCrud('Pending Approval', 'far fa-question-circle', Question::class)
            ->setPermission('ROLE_MODERATOR')
... lines 65 - 69
    }
... lines 71 - 130
}
```

If we stopped now, you might be thinking:

> Wait a second! Both of these menu items simply point to the `Question` entity. How will EasyAdmin know which controller to go to?

This definitely *is* a problem. The truth is that, when we have multiple CRUD controllersfor the same entity, EasyAdmin *guesses* which to use. To tell it explicitly, add `->setController()` and then pass it `QuestionPendingApprovalCrudController::class`.

132 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 57
58      public function configureMenuItems(): iterable
59      {
... lines 60 - 62
63          yield MenuItem::linkToCrud('Pending Approval', 'far fa-question-circle', Question::class)
... line 64
65              ->setController(QuestionPendingApprovalCrudController::class);
... lines 66 - 69
70      }
... lines 71 - 132
```

Do we need to set the controller on the other link to be safe? Absolutely. And we'll do that in a few minutes.

But let's try this. Refresh. We get two links... and each section looks absolutely identical, which makes sense. Let's modify the query for the new section to only show *non-approved* questions. And... we already know how to do that!

Over in the new controller, override the method called `createIndexQueryBuilder()`. Then we'll just modify this: `->andWhere()` and we know that our entity alias is always `entity`. So `entity.isApproved` (that's the field on our `Question` entity) `= :approved` ... and then `->setParameter('approved', false)`.

20 lines | src/Controller/Admin/QuestionPendingApprovalCrudController.php

```
... lines 1 - 4
5   use Doctrine\ORM\QueryBuilder;
6   use EasyCorp\Bundle\EasyAdminBundle\Collection\FieldCollection;
7   use EasyCorp\Bundle\EasyAdminBundle\Collection\FilterCollection;
8   use EasyCorp\Bundle\EasyAdminBundle\Dto\EntityDto;
9   use EasyCorp\Bundle\EasyAdminBundle\Dto\SearchDto;
10
11  class QuestionPendingApprovalCrudController extends QuestionCrudController
12  {
13      public function createIndexQueryBuilder(SearchDto $searchDto, EntityDto $entityDto, FieldCollection $fields, FilterCollection $filters): Que
14      {
15          return parent::createIndexQueryBuilder($searchDto, $entityDto, $fields, $filters)
16              ->andWhere('entity.isApproved = :approved')
17              ->setParameter('approved', false);
18      }
19  }
```

Let's try it! We go from a *bunch* question to... just *five*. It works! Except that if you go to the original Question section... that *also* only shows five!

Yup, it's guessing the *wrong* CRUD controller. So in practice, as soon as you have multiple CRUD controllers for an entity, you should *always* specify the controller when you link to it. For this one, use `QuestionCrudController::class`.

133 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
25  class DashboardController extends AbstractDashboardController
26  {
... lines 27 - 57
58      public function configureMenuItems(): iterable
59      {
... line 60
61          yield MenuItem::linkToCrud('Questions', 'fa fa-question-circle', Question::class)
62              ->setController(QuestionCrudController::class)
... lines 63 - 70
71      }
... lines 72 - 131
132 }
```

If we head over and refresh this page... there's no difference! That's because we modified the *link*... but we're already *on* the

page for the *new* CRUD controller. So click the link and... much better!

Including Entity Data in the Page Title

Let's tweak a few things on our new CRUD controller. Override `configureCrud()`. Most importantly, we should `->setPageTitle()` to set the title for `Crud::PAGE_INDEX` to "Questions Pending Approval".

27 lines | src/Controller/Admin/QuestionPendingApprovalCrudController.php

```
... lines 1 - 7
use EasyCorp\Bundle\EasyAdminBundle\Config\Crud;
... lines 9 - 11
class QuestionPendingApprovalCrudController extends QuestionCrudController
{
    public function configureCrud(Crud $crud): Crud
    {
        return parent::configureCrud($crud)
            ->setPageTitle(Crud::PAGE_INDEX, 'Questions pending approval');
    }
... lines 19 - 25
}
```

Now... it's *much* more obvious which page we're on.

Oh, and when we set the page title, we can actually pass a *callback* if we want to use the `Question` object itself in the name... assuming you're setting the page title for the detail or edit pages where you're working with a *single* entity.

Check it out: call `->setPageTitle()` again, and set *this* one for `Crud::PAGE_DETAIL`. Then, instead of a string, pass a callback: a `static function` that will receive the `Question` object as the first argument. Inside, we can return whatever we want: how about `return sprintf()` with `#%s %s` ... passing `$question->getId()` and `$question->getName()` as the wildcards.

31 lines | src/Controller/Admin/QuestionPendingApprovalCrudController.php

```
... lines 1 - 14
    public function configureCrud(Crud $crud): Crud
    {
        return parent::configureCrud($crud)
... line 18
            ->setPageTitle(Crud::PAGE_DETAIL, static function (Question $question) {
                return sprintf('#%s %s', $question->getId(), $question->getName());
            });
    }
... lines 23 - 31
```

Let's check it! Head over to the detail page for one of these questions and...awesome! Dynamic data in the title.

And while we're here, I also want to add a "help" message to the index page:

> Questions are not published to users until approved by a moderator

32 lines | src/Controller/Admin/QuestionPendingApprovalCrudController.php

```
... lines 1 - 14
    public function configureCrud(Crud $crud): Crud
    {
        return parent::configureCrud($crud)
... lines 18 - 21
            ->setHelp(Crud::PAGE_INDEX, 'Questions are not published to users until approved by a moderator');
    }
... lines 24 - 32
```

When we refresh... our message shows up right next to the title!

Autocomplete() and Multiple CRUD Controllers

Okay, there's one more subtle problem that having two CRUD controllers has just created.To see it, jump into `AnswerCrudController`. Find the `AssociationField` for `question` ... and add `->autocomplete()` ... which it needs because there's going to be *a lot* of questions in our database.

39 lines | src/Controller/Admin/AnswerCrudController.php

```
... lines 1 - 12
class AnswerCrudController extends AbstractCrudController
{
... lines 15 - 19
    public function configureFields(string $pageName): iterable
    {
... lines 22 - 26
        yield AssociationField::new('question')
            ->autocomplete()
... lines 29 - 36
    }
}
```

If we look at our main Questions page...this first question is *probably* an approved question - since most are - so I'll copy part of its name. Now go to Answers, edit an answer... and go down to the Question field. This uses autocomplete, which is cool! But if I paste the string, it says "No results found"?

The reason is subtle. Go down to the web debug toolbar and open the profiler for one of those autocomplete AJAX requests. Look at the URL closely... part of it says "crudController = QuestionPendingApprovalCrudController"!

When an autocomplete AJAX request is made for an entity (in this case,it's trying to autocomplete Question), that AJAX request is done *by* a CRUD controller. If you jump into `AbstractCrudController` ... there's actually an `autocomplete()` action. This is the action that's called to create the autocomplete response. It's done this way so that the autocomplete results can reuse your index query builder. Unfortunately, just like with our dashboard links, the autocomplete system is *guessing* which of our two CRUD controllers to use for Question... and it's guessing wrong.

To fix this, once again, we just need to be explicit.Add `->setCrudController(QuestionCrudController::class)`.

41 lines | src/Controller/Admin/AnswerCrudController.php

```
... lines 1 - 20
    public function configureFields(string $pageName): iterable
    {
... lines 23 - 27
        yield AssociationField::new('question')
            ->autocomplete()
            ->setCrudController(QuestionController::class)
... lines 31 - 38
    }
... lines 40 - 41
```

This time, I'll refresh... go down to the Question field, search for the string and...it finds it!

Next, what if we want to run some code before or after an entity is updated, created, or deleted? EasyAdmin has two solutions: Events and controller methods.

Chapter 31: Extending with Events

So far, we've added behavior to our code by overriding methods in our controllers. And that is a *great* approach, and will be what you should use in most cases. But there *is* another possibility: events.

Over in `QuestionCrudController`, up in `configureFields()` ... let's return one more field: `yield AssociationField::new('updatedBy')`.

This field - that lives on the `Question` entity - is a `ManyToOne` to `User`. The idea is that, whenever someone updates a `Question`, this field will be set to the `User` object that just updated it. Let's make this *only* show up on the detail page: `->onlyOnDetail()`.

106 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 19
20  class QuestionCrudController extends AbstractCrudController
21  {
... lines 22 - 46
47      public function configureFields(string $pageName): iterable
48      {
... lines 49 - 92
93          yield AssociationField::new('updatedBy')
94              ->onlyOnDetail();
95      }
... lines 96 - 104
105 }
```

Right now, in our fixtures, we are *not* setting that field. So if we go to any question... it says "Updated By", "Null". Our goal is to set that field automatically when a question is updated.

A *great* solution for this would be to use the doctrine extensions library and its "blameable" feature. Then, no matter *where* this entity is updated - inside the admin or not - the field would automatically be set to whoever is logged in.

Discovering the Events

But let's see if we can achieve this *just* inside our EasyAdmin section via events. EasyAdmin has a bunch of events that it dispatches and the best way to find them is to go into the source code. In EasyAdmin's vendor code, open the `src/Event/` directory. Most of these are... pretty self explanatory! `BeforeCrudAction` is dispatched at the start when *any* CRUD action is executed, "after" would be at the end of that action.. and we also have a bunch of things related to entities, like `BeforeEntityUpdatedEvent` or `BeforeEntityPersistedEvent`, where "persisted" means "created".

For our case, the one I'm looking at is `BeforeEntityUpdatedEvent`. If we could run code *before* an entity is updated, we could set this `updatedBy` field and then let it save naturally. Let's do that.

Creating The Event Subscriber

Open up `BeforeEntityUpdatedEvent` and copy its namespace. Then, over on our terminal, run:

```
symfony console make:subscriber
```

Let's call it `BlameableSubscriber`. It then asks us which event we want to listen to, and it suggests a bunch from the core of Symfony. The one from EasyAdminBundle won't be here, so, instead, I'll paste its namespace, *then* go grab its class name... and paste that too.

And... perfect! We have a new `BlameableSubscriber` class! Go open that up: `src/EventSubscriber/BlameableSubscriber.php`.

22 lines | src/EventSubscriber/BlameableSubscriber.php

```
... lines 1 - 4
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use EasyCorp\Bundle\EasyAdminBundle\Event\BeforeEntityUpdatedEvent;

class BlameableSubscriber implements EventSubscriberInterface
{
    public function onBeforeEntityUpdatedEvent(BeforeEntityUpdatedEvent $event)
    {
        // ...
    }

    public static function getSubscribedEvents()
    {
        return [
            BeforeEntityUpdatedEvent::class => 'onBeforeEntityUpdatedEvent',
        ];
    }
}
```

This is a normal Symfony event subscriber and, thanks to auto configuration,Symfony will instantly see this and start using it. In other words, whenever EasyAdmin dispatches `BeforeEntityUpdatedEvent`, it will call our method.

This `$event` object is *packed* with useful info. For example, if I just say `$event->`, one method is called `getEntityInstance()`, which is *exactly* what we want.

To be able to set the `updatedBy` property on our question, we're going to need the current user object, which we get via the security service. Let's autowire that: add `public function __construct()` - with a `Security $security` argument. Hit "alt" + "enter" and go "Initialize properties" to create that property and set it.

30 lines | src/EventSubscriber/BlameableSubscriber.php

```
... lines 1 - 6
use Symfony\Component\Security\Core\Security;
... line 8
class BlameableSubscriber implements EventSubscriberInterface
{
    private Security $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }
... lines 17 - 28
}
```

Love it. Below, start with `$question = $event->getEntityInstance()`. And then `if (!$question instanceof Question)`, just `return` ... because this is going to be called when *every* entity is saved across our entire system. Next, `$user = $this->security->getUser()` and `if (!$user instanceof User)`, let's throw a new: `LogicException()` ... the exception class doesn't matter. This is a situation that will never *actually* happen: we only have one `User` class in our app. So if you're logged in, you will *definitely* have this `User` instance. *But*, this helps our editor and static analysis tools.

43 lines | src/EventSubscriber/BlameableSubscriber.php

```
... lines 1 - 19
    public function onBeforeEntityUpdatedEvent(BeforeEntityUpdatedEvent $event)
    {
        $question = $event->getEntityInstance();
        if (!$question instanceof Question) {
            return;
        }

        $user = $this->security->getUser();
        // We always should have a User object in EA
        if (!$user instanceof User) {
            throw new \LogicException('Currently logged in user is not an instance of User?!');
        }
... lines 32 - 33
    }
... lines 35 - 43
```

Down here... we can now say `$question->setUpdatedBy()`, and pass `$user`.

43 lines | src/EventSubscriber/BlameableSubscriber.php

```
... lines 1 - 19
    public function onBeforeEntityUpdatedEvent(BeforeEntityUpdatedEvent $event)
    {
... lines 22 - 32
        $question->setUpdatedBy($user);
    }
... lines 35 - 43
```

Let's try it. This question's "Updated By" is "Null". Edit something (make sure you actually make a change so it saves), hit "Save changes" and... got it! "Updated By" is populated! And *that* is my current user. Sweet!

Alternative: Overriding a Method

So events are a powerful concept in EasyAdmin. However, they're a little bit less important in EasyAdmin 3 and 4 than they used to be. And that's because most of our configuration is now written in PHP in our controller. So instead of leveraging events, there's often an easier way: we can just override a method in our controller.

Event subscribers *still* have their place, because they are a great way to do an operation on *multiple* entities in your system. But if you only need to do something on *one* entity... it's easier to override a method inside that entity's controller.

Let's try it. I'll got to the bottom of my controller class and override yet *another* method. The methods that we can override are almost a README of all the different ways that you can extend things. There's a `createEntity()` method, a `createEditForm()` method, and the one we want is called `updateEntity()`. This is the method that actually updates and saves the entity. *Before* that happens, we want to set the property.

115 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 20
class QuestionCrudController extends AbstractCrudController
{
... lines 23 - 106
    /**
     * @param Question $entityInstance
     */
    public function updateEntity(EntityManagerInterface $entityManager, $entityInstance): void
    {
        parent::updateEntity($entityManager, $entityInstance);
    }
}
```

Go steal the code from our subscriber... close that event class... paste that in... and hit "OK" to add that use statement. And now we'll tweak some code: `$user = $this->getUser()` ... and then `$question` is actually going to be `$entityInstance`. So we can say

`$entityInstance->setUpdatedBy()` .

122 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 110
    public function updateEntity(EntityManagerInterface $entityManager, $entityInstance): void
    {
        $user = $this->getUser();
        if (!$user instanceof User) {
            throw new \LogicException('Currently logged in user is not an instance of User?!');
        }
        $entityInstance->setUpdatedBy($user);

        parent::updateEntity($entityManager, $entityInstance);
    }
... lines 121 - 122
```

If you want to code defensively, since there's no type-hint on `$entityInstance` , we could do another check where we say `if (!$entityInstance instanceof Question)` then throw an exception. But in practice, this *will* always be a `Question` object.

Ok: let's see if this works. Go into `BlameableSubscriber` ... and comment out the listener. The subscriber is still here, but it won't *do* anything anymore. Then go back to Questions... and edit a different question. Actually, before I do that, go look at the details to make sure there's no "Updated By". Perfect! Now edit, make a change, save your changes, and...it still works!

Next, let's do a little bit more with our admin menu, like adding sections to make this whole thing better organized.

Chapter 32: Having Fun with the Menu

Our menu on the left is getting a *little* long and... *kind of* confusing... since we now have *two* question links. To make this more user-friendly, let's divide this into a sub-menu. We do that inside of `DashboardController` ... because that's, of course, where we configure the menu items.

Adding a Sub Menu

To create a sub-menu, say `yield MenuItem::subMenu()` and then give that a name - like `Questions` - and an icon... just like we do with normal menu items.

To populate the *items* in this menu, say `->setSubItems()` , pass this an array, and then we'll wrap our other two menu item objects *inside* of this. Of course, now we need to indent, remove the `yield` , and... replace the semicolons with commas.

Perfect! Now change `Questions` to... how about `All` ... and let's play with the icons. Change the first to `fa fa-list` ... and the second to `fa fa-warning` .

136 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
class DashboardController extends AbstractDashboardController
{
... lines 27 - 57
    public function configureMenuItems(): iterable
    {
... line 60
        yield MenuItem::subMenu('Questions', 'fa fa-question-circle')
            ->setSubItems([
                MenuItem::linkToCrud('All', 'fa fa-list', Question::class)
                    ->setController(QuestionCrudController::class)
                    ->setPermission('ROLE_MODERATOR'),
                MenuItem::linkToCrud('Pending Approval', 'fa fa-warning', Question::class)
                    ->setPermission('ROLE_MODERATOR')
                    ->setController(QuestionPendingApprovalCrudController::class),
            ]);
... lines 70 - 73
    }
... lines 75 - 134
}
```

Let's try that. Move over... refresh and... ahhh, much cleaner!

Menu Sections

But wait, there's *more* we can do with the menu...like adding separators... technically called "sections". Right after `linkToDashboard()` , add `yield MenuItem::section()` and pass it `Content` .

136 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 57
    public function configureMenuItems(): iterable
    {
... line 60
        yield MenuItem::section('Content');
... lines 62 - 75
    }
... lines 77 - 138
```

Let's put one more down here - `yield MenuItem::section()` ... but this time leave the label blank. So unlike sub-menus, which *wrap* menu items, you can just pop a section anywhere that you want a separator.

138 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 57
58      public function configureMenuItems(): iterable
59      {
... lines 60 - 72
73          yield MenuItem::linkToCrud('Users', 'fas fa-users', User::class);
74          yield MenuItem::section();
75          yield MenuItem::linkToUrl('Homepage', 'fas fa-home', $this->generateUrl('app_homepage'));
76      }
... lines 77 - 138
```

Let's go check it out. Refresh and... very nice! Separator one says "Content"... and separator two gives us a little gap without any text.

External Links

We saw earlier that you can add menu links to point to a dashboard, other CRUD sections..or just *any* URL you want, like the Homepage. So, not surprisingly, you can *also* link to external sites. For instance, let's say that I love StackOverflow *so* much, that I want to link to it. We can tweak the icons, and for the URL, pass whatever you want, like https://stackoverflow.com.

139 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 57
58      public function configureMenuItems(): iterable
59      {
... lines 60 - 75
76          yield MenuItem::linkToUrl('StackOverflow', 'fab fa-stack-overflow', 'https://stackoverflow.com');
77      }
... lines 78 - 139
```

Oh, but let me fix my icon name. Great! Now when we refresh... no surprise, that works fine.

More Menu Item Options

If you look closer at these menu items, you'll see that they have a *lot* of options on them! We know we have things like `setPermission()` and `setController()`, but we *also* have methods like `setLinkTarget()`, `setLinkRel()`, `setCssClass()`, or `setQueryParameter()`. For this case, let's `->setLinkTarget('_blank')` ... so that *now* if I click "StackOverflow", it pops up in a new tab.

140 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 57
58      public function configureMenuItems(): iterable
59      {
... lines 60 - 75
76          yield MenuItem::linkToUrl('StackOverflow', 'fab fa-stack-overflow', 'https://stackoverflow.com')
77              ->setLinkTarget('_blank');
78      }
... lines 79 - 140
```

Next: what if we need to disable an action on an entity-by-entity basis? Like, we want only want to allow questions to be deleted if they are *not* approved. Let's dive into that.

Chapter 33: Conditionally Disabling an Action

Okay, new goal. This page lists *all* of the questions on our site..while the Pending Approval page lists only *not* approved questions. So ID 24 is *not* approved. We can see this same item on the main Questions page...and at the end of each row, there's a link to delete that question.

I want to change this so that only *non-approved* questions can be deleted. For instance, we *should* be able to delete question 24, but *not* question 13 because it's an *approved* question. How can we do that?

Since we're talking about questions, let's go to `QuestionCrudController`. The most obvious place is `configureActions()`. After all, this is where we configure which actions our CRUD has, which action links appear on which page, and what permissions each has. We can even call `->disable()` and pass an action name to completely disable an action for this CRUD.

Actions and Action Objects

But, that's *not* what we want to do here. We don't want to disable the "delete" action *everywhere*, we just want to disable it for *some* of our questions. To figure out how to do that, we need to talk more about the `Actions` and `Action` classes.

The `Actions` class is basically a container that says which actions should be on which page. So it knows that on our index page, we want to have a "show" or "detail" action, "Edit" action, and "Delete" action.

This `Actions` object is actually created in `DashboardController`. It *also* has a `configureActions()` method. And if we jump into the parent method, yup! *This* is where it creates the `Actions` object and sets up all the *default* actions for each page. So `PAGE_INDEX` will have `NEW`, `EDIT`, and `DELETE` actions... and `PAGE_DETAIL` will have `EDIT`, `INDEX`, and `DELETE`. *We* also added the `DETAIL` action to `PAGE_INDEX`.

Notice that when we use the `->add()` method - or when our parent controller uses it - we pass a *string* for the action name. `Action::EDIT` is a just constant that resolves to the string "edit".

But, behind the scenes, EasyAdmin creates an `Action` *object* to represent this. And that `Action` object knows *everything* about how that action should look, including its label, CSS classes, and other stuff. So really, this `Actions` object is a collection of the `Action` *objects* that should be displayed on each page.

And if you *did* find yourself with an `Action` object - I'll jump into that class - there would be all kinds of things that you could configure on it, like its label, icon, and more. It even has a method called `displayIf()` where we can dynamically control whether or not this action is displayed.

So... great! We could use that to conditionally hide or show the delete link! Yep! Except that... inside of `configureActions()`, to do that, we need a way to *get* the `Action` object for a specific action... like "give me the `Action` object for the "delete" action on the "index" page. Then we could call `->displayIf()` on that.

But... this doesn't work. There's no way for us to access the `Action` object that represents the `DELETE` action on the `PAGE_INDEX`. So... does this mean that the built-in actions added by `DashboardController` can't be changed?

Thankfully, no! We *can* tweak these `Action` objects thanks to a nice function called `->update()`. Say `->update(Crud::PAGE_INDEX, Action::DELETE)`, and then pass a callback that will receive an `Action` argument.

Using Actions::displayIf()

Perfect! This now means that, after the `DELETE` action object is created for `PAGE_INDEX`, it will be passed to *us* so we can make changes. For now, just `dd($action)`.

125 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 21
class QuestionCrudController extends AbstractCrudController
{
... lines 24 - 37
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
            ->update(Crud::PAGE_INDEX, Action::DELETE, static function(Action $action) {
                dd($action);
            })
... lines 44 - 49
    }
... lines 51 - 123
}
```

If we refresh... yup! It dumped the `Action` object, as expected... which has an `ActionDto` object inside... where all the data is really held.

Back in the callback, add `$action->displayIf()` and pass this another callback: a `static function()` that will receive a `Question $question` argument. Now, *each* time the `DELETE` action is about to be displayed on the index page -like for the first, second then third question, etc - it will call our function and pass us that `Question`. Then, we can decide whether or not the delete action link should be shown. Let's show the delete link if `!$question->getIsApproved()`.

127 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
            ->update(Crud::PAGE_INDEX, Action::DELETE, static function(Action $action) {
                $action->displayIf(static function (Question $question) {
                    return !$question->getIsApproved();
                });
            })
... lines 46 - 51
    }
... lines 53 - 127
```

Sweet! Let's see what happens. Refresh and... error!

```
Call to a member function getAsDto() on null
```

Boo Ryan. I always do that. Inside `update()`, you need to *return* the action. There we go, much better!

And now... if we check the menu... look! The "Delete" action is gone! But if you go down to ID 24 - which is *not* approved - it's there! That's awesome!

Forbidding Deletes Dynamically

But, this isn't *quite* good enough. We're hiding the link on this *one* page only. And so, we should repeat this for the `DELETE` action on the *detail* page. And... you may need to disable the delete batch action entirely.

But even *that* wouldn't be enough... because if an admin somehow got the "Delete" URL for an approved question,the delete action *would* still work. The action *itself* isn't secure.

To give us that extra layer of security, right before an entity is deleted,let's check to see if it's approved. And if it *is*, we'll throw an exception.

To test this, temporarily comment-out this logic and `return true` ... so that the delete link *always* shows. Back to the Questions page... got it!

129 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
    public function configureActions(Actions $actions): Actions
    {
        return parent::configureActions($actions)
            ->update(Crud::PAGE_INDEX, Action::DELETE, static function(Action $action) {
                $action->displayIf(static function (Question $question) {
                    // always display, so we can try via the subscriber instead
                    return true;
                    //return !$question->getIsApproved();
                });
            })
... lines 48 - 53
    }
... lines 55 - 129
```

Now go to the bottom of `QuestionCrudController`. Earlier we overrode `updateEntity()`. This time we're going to override `deleteEntity()` ... which will allow us to call code right *before* an entity is deleted. To help my editor, I'll document that the entity is going to be an instance of `Question`.

141 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 128
    /**
     * @param Question $entityInstance
     */
    public function deleteEntity(EntityManagerInterface $entityManager, $entityInstance): void
    {
... lines 134 - 138
    }
... lines 140 - 141
```

Now, `if ($entityInstance->getIsApproved())`, throw a new `\Exception('Deleting approved questions is forbidden')`. This is going to look like a 500 Error to the user... so we could also throw an "access denied exception".Either way, this isn't a situation that anyone should have... unless we have a bug in our code or a user is trying to do something they shouldn'tBad admin user!

141 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 131
    public function deleteEntity(EntityManagerInterface $entityManager, $entityInstance): void
    {
        if ($entityInstance->getIsApproved()) {
            throw new \Exception('Deleting approved questions is forbidden!');
        }

        parent::deleteEntity($entityManager, $entityInstance);
    }
... lines 140 - 141
```

I won't try this, but I'm pretty sure it would work.However, this *is* all a bit tricky! You need to secure the actual *action*... and also make sure that you remember to hide *all* the links to this action with the correct logic.

Life would be a lot easier if we could, instead, *truly* disable the `DELETE` action conditionally, on an entity-by-entity basis. If we could do that, EasyAdmin would hide or show the "Delete" links automatically... and even handle securing the action if someone guessed the URL.

Is that possible? Yes! We're going to need an event listener and some EasyAdmin internals.That's next.

Chapter 34: Dynamic Disable an Action & AdminContext

We've done a good job of hiding the `DELETE` action conditionally and disallowing deletes using that same condition. But it would be *much* simpler if we could truly *disable* the `DELETE` action on an entity-by-entity basis. Then EasyAdmin would naturally just... hide the "Delete" link.

The AdminContext Object

To figure out how to do this, let's click into our base class - `AbstractCrudController` - and go down to where the controller methods are. Check this out: in every controller method - like `index()`, `detail()`, or `delete()` - we're passed something called an `AdminContext`. This is a configuration object that holds *everything* about your admin section, *including* information about which EasyAdmin *actions* should be enabled. So, by the time our controller method has been called, our EasyAdmin *actions* config has already been used to populate details inside of this `AdminContext`.

And look what happens immediately inside the method: it dispatches an event! I wonder if we could hook into this event and *change* the action config - like conditionally disabling the `DELETE` action - before the rest of the method runs and the template renders.

Creating the Event Subscriber

Let's try that! Scroll up to `BeforeCrudActionEvent` - let me search for that... there we go - and copy it. Spin over to your terminal and run:

```
symfony console make:subscriber
```

Let's call it `HideActionSubscriber` ... and then paste the long event class. Beautiful! Let's go see what that subscriber looks like.

21 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 7
class HideActionSubscriber implements EventSubscriberInterface
{
    public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
    {
    }

    public static function getSubscribedEvents()
    {
        return [
            BeforeCrudActionEvent::class => 'onBeforeCrudActionEvent',
        ];
    }
}
```

It looks... pretty familiar! Let's `dd($event)` to get started.

22 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 9
    public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
    {
        dd($event);
    }
... lines 14 - 22
```

When we refresh... it immediately hits that because this event is dispatched *before* every single CRUD action.

Working with AdminContext

The hardest part of figuring out how to dynamically disable the action is just...figuring out where all the data is. As you can see, we have the `AdminContext`. *Inside* the `AdminContext`, among other things, is something called a `CrudDto`. Inside the `CrudDto`, we have an `ActionConfigDto`. *This* holds information about all the actions, including "index" (the current page name), and all the action config. This shows us, for each page, which array of action objects should be enabled. So for the "edit" page, we have these two `ActionDto` objects, and each `ActionDto` object contains all the information about what that action should look like. Whew...

So now the trick is to use this information (and there's *a lot* of it) to modify this config and disable the `DELETE` action in the right situation. Back over in our listener, the first thing we need to do is get that `AdminContext`. Set a variable and do an if statement all at once: `if (!$adminContext = $event->getAdminContext())`, then `return`.

I'm coding defensively. It's probably not necessary... but technically the `getAdminContext()` method might not return an `AdminContext`. I'm not even sure if that's possible, but better safe than sorry. Now get the `CrudDto` the same way: `if (!$crudDto = $adminContext->getCrud())`, then also `return`. Once again, this is *theoretically* possible... but not going to happen (as far as I know) in any real situation.

31 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 10
11      public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
12      {
13          if (!$adminContext = $event->getAdminContext()) {
14              return;
15          }
16          if (!$crudDto = $adminContext->getCrud()) {
17              return;
18          }
... lines 19 - 21
22      }
... lines 23 - 31
```

Next, remember that we only want to perform our change when we're dealing with the `Question` class. The `CrudDto` has a way for us to check which entity we're dealing with. Say `if ($crudDto->getEntityFqcn() !== Question::class)`, then `return`.

31 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 10
11      public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
12      {
... lines 13 - 18
19          if ($crudDto->getEntityFqcn() !== Question::class) {
20              return;
21          }
22      }
... lines 23 - 31
```

So... this is *relatively* straightforward, but, to be honest, it took me some digging to find *just* the right way to get this info.

Disabling the Action

Now we can get to the core of things. The first thing we want to do is *disable* the delete action entirely if a question is approved. We can get the entity instance by saying `$question = $adminContext->getEntity()->getInstance()`. The `getEntity()` gives us an `EntityDto` object... and then you can get the instance from that.

38 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 11
12      public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
13      {
... lines 14 - 24
25          $question = $adminContext->getEntity()->getInstance();
... lines 26 - 28
29      }
... lines 30 - 38
```

Below, we're going to do something a *little* weird at first. Say `if ($question instanceof Question)` (I'll explain why I'm doing that in a second) `&& $question->getIsApproved()`, then disable the action by saying `$crudDto->getActionsConfig()` - which gives us an `ActionsDto` object - then `->disableActions()` with `[Action::DELETE]`.

38 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 11
12      public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
13      {
... lines 14 - 23
24          // disable action entirely delete, detail, edit
25          $question = $adminContext->getEntity()->getInstance();
26          if ($question instanceof Question && $question->getIsApproved()) {
27              $crudDto->getActionsConfig()->disableActions([Action::DELETE]);
28          }
29      }
... lines 30 - 38
```

There are a few things I want to explain. The first is that this event is going to be called at the beginning of *every* CRUD page. If you're on a CRUD page like `EDIT`, `DELETE`, or `DETAIL`, then `$question` *is* going to be a `Question` instance. *But*, if you're on the index page... that page does *not* operate on a *single* entity. In that case, `$question` will be null. By checking for `$question` being an `instanceof Question`, we're basically checking to make sure that `Question` isn't null. It also helps my editor know, over here, that I can call the `->getIsApproved()` method.

The other thing I want to mention is that, at this point, when you're working with EasyAdmin, you're working with a lot of DTO objects. We talked about these earlier. Inside of our controller, we deal with these nice objects like `Actions` or `Filters`. But behind the scenes, these are just helper objects that ultimately configure DTO objects. So in the case of `Actions`, internally, it's *really* configuring an `ActionConfigDto`. Any time we call a method on `Actions` ... it's actually... if I jump around... making changes to the DTO.

And if we looked down here on the `Filters` class, we'd see the same thing. So by the time you get to *this* part of EasyAdmin, you're dealing with those DTO objects. They hold all of the same data as the objects we're used to working with, but with different methods for interacting with them. In this case, if you dig a bit, `getActionsConfig()` gives you that `ActionConfigDto` object... and it has a method on it called `->disabledActions()`. I'll put a comment above this that says:

```
// disable action entirely for delete, detail & edit pages
```

Yup, if we're on the detail, edit, or delete pages, then we're going to have a `Question` instance... and we can disable the `DELETE` action entirely.

But this *isn't* going to disable the links on the index page. Watch: if we refresh that page... all of these are approved, so I should *not* be able to delete them. If I *click* "Delete" on ID 19... yay! It *does* prevent us:

```
You don't have enough permissions to run the "delete"
action [...] or the "delete" action has been disabled.
```

That's thanks to us disabling it right here. And also, if we go to the detail page, you'll notice that the "Delete" action is gone. But if we click a Question down here, like ID 24 that is *not* approved, it *does* have a "Delete" button.

Ok, let's finish by hiding the "Delete" link on the index page. To do that, add `$actions = $crudDto->getActionConfig()`, just like we

did before, and then `->getActions()`. *This* will give us an *array* of the `ActionDto` objects that will be enabled for this page. So if this is the index page, for example, then it will have a "Delete" action in that array. I'm going to check for that: `if (!$deleteAction = $actions[Action::DELETE])` ... and then add `??  null` in case that key isn't set. If there is *no* delete action for some reason, just `return`.

50 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 12
    public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
    {
... lines 15 - 30
        // This gives you the "configuration for all the actions".
        // Calling ->getActions() returns the array of actual actions that will be
        // enabled for the current page... so then we can modify the one for "delete"
        $actions = $crudDto->getActionsConfig()->getActions();
        if (!$deleteAction = $actions[Action::DELETE] ?? null) {
            return;
        }
... lines 38 - 40
    }
... lines 42 - 50
```

But if we *do* have a `$deleteAction`, then say `$deleteAction->setDisplayCallable()`.

This is a great example of the difference between how code looks on these DTO objects and how it looks with the objects in the controllers. There, on the `Action` object, we can call `$action->displayIf()`. In the event listener, with this `ActionDto`, you can do the same thing, but it's called `->setDisplayCallable()`. Pass this a `function()` with a `Question $question` argument... then we'll say: please display this action link if `!$question->getIsApproved()`.

50 lines | src/EventSubscriber/HideActionSubscriber.php

```
... lines 1 - 12
    public function onBeforeCrudActionEvent(BeforeCrudActionEvent $event)
    {
... lines 15 - 37
        $deleteAction->setDisplayCallable(function(Question $question) {
            return !$question->getIsApproved();
        });
    }
... lines 42 - 50
```

Phew! Let's try that! We're looking to see that this "Delete" action link is hidden from the index page. And now... it *is*! It's gone for all of them, *except*... if I go down and find one with a higher ID...which is *not* approved... yes! It *does* have a "Delete" link.

> To prevent admin users from using the checkboxes next to each question to "batch delete" approved questions, in `configureActions()`, call `->disable(Action::BATCH_DELETE)`.

Next, let's add a custom action! We're going to start simple: a custom action link that takes us to the frontend of the site. Then we'll get *more* complicated.

Chapter 35: Simple Custom GET Action

Let's add a totally *custom* action. What if, when we're on the detail page for a question, we add a new action called "view"that takes us to the frontend for that question? Sounds good! Start in `QuestionCrudController`. To add a new action... we'll probably need to do some work inside of `configureActions()`. We already know how to add actions to different pages: with the `->add()` method. Let's try adding, to `Crud::PAGE_DETAIL`, a new action called `view`.

142 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 21
22  class QuestionCrudController extends AbstractCrudController
23  {
... lines 24 - 37
38      public function configureActions(Actions $actions): Actions
39      {
40          return parent::configureActions($actions)
... lines 41 - 53
54              ->add(Crud::PAGE_DETAIL, 'view');
55      }
... lines 56 - 140
141 }
```

Adding the Custom Action in configureActions()

There are a bunch of built-in action names - like `index` or `delete` - and we usually reference those via their `Action` constant. But in this case, we're making a *new* action... so let's just "invent" a string called `view` ... and see what happens.

Refresh and... what happened was... an error!

> The "view" action is not a built-in action, so you can't
> add or configure it via its name. Either refer to one of
> the built-in actions or create a custom action called "view".

In the last chapters, we talked about how, behind-the-scenes,each action is actually an `Action` object. We don't really think about that most of the time... but when we create a*custom* action, we need to deal with this object directly.

Above the `return`, create an `Action` object with `$viewAction = Action::new()` ... and pass this the action name that we just invented: `view`. Then, below, instead of the string, this argument accepts an `$actionNameOrObject`. Pass in that new `$viewAction` variable.

144 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
38      public function configureActions(Actions $actions): Actions
39      {
40          $viewAction = Action::new('view');
... line 41
42          return parent::configureActions($actions)
... lines 43 - 55
56              ->add(Crud::PAGE_DETAIL, $viewAction);
57      }
... lines 58 - 144
```

Setting the Action to redirect

Refresh again to see... *another* error:

> Actions must link to either a route, a CRUD action, or a URL.

And then it gives us three different methods we can use to set that up. That's a pretty great error message. It sounds like `linkToRoute()` or `linkToUrl()` is what we need.

So, up here, let's modify our action. We *could* use `->linkToRoute()` ... but as we learned earlier, that would generate a URL *through* the admin section, complete with all the admin query parameters. Not what we want. Instead, use `->linkToUrl()`.

But, hmm. We can't use `$this->generateUrl()` yet... because we need to know *which* `Question` we're generating the URL for. And we don't have that! Fortunately, the argument accepts a string or *callable*. Let's try that: pass a `function()` ... and then to see what arguments this receives, let's use a trick: `dd(func_get_args())`.

147 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
38      public function configureActions(Actions $actions): Actions
39      {
40          $viewAction = Action::new('view')
41              ->linkToUrl(function() {
42                  dd(func_get_args());
43              });
... lines 44 - 59
60      }
... lines 61 - 147
```

Back in the browser... awesome! We are apparently passed *one* argument, which is the `Question` object. We're dangerous! Use that: `return $this->generateUrl()`, passing the frontend route name: which is `app_question_show`. This route has a `slug` route wildcard... so add the `Question $question` argument to the function and set `slug` to `$question->getSlug()`.

149 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
38      public function configureActions(Actions $actions): Actions
39      {
40          $viewAction = Action::new('view')
41              ->linkToUrl(function(Question $question) {
42                  return $this->generateUrl('app_question_show', [
43                      'slug' => $question->getSlug(),
44                  ]);
45              });
... lines 46 - 61
62      }
... lines 63 - 149
```

Testing time! And now... yes! We have a "View" button. If we click it... it works!

Customizing How the Action Looks

And just like any other action, we can modify how this looks. Let's `->addCssClass('btn btn-success')`, `->setIcon('fa fa-eye')`, and `->setLabel('View on site')`: all things that we've done before for other actions.

152 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
38      public function configureActions(Actions $actions): Actions
39      {
40          $viewAction = Action::new('view')
... lines 41 - 45
46              ->addCssClass('btn btn-success')
47              ->setIcon('fa fa-eye')
48              ->setLabel('View on site');
... lines 49 - 64
65      }
... lines 66 - 152
```

Refresh and... that looks great! If we want to include this action on other pages, we can. Because, if you go to the index page, there's no "view on frontend" action. Thankfully, we created this nice `$viewAction` variable, so, at the bottom, we can reuse it:

`->add(Crud::PAGE_INDEX, $viewAction)` .

153 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
38      public function configureActions(Actions $actions): Actions
39      {
... lines 40 - 49
50          return parent::configureActions($actions)
... lines 51 - 63
64              ->add(Crud::PAGE_DETAIL, $viewAction)
65              ->add(Crud::PAGE_INDEX, $viewAction);
66      }
... lines 67 - 153
```

Refresh and... got it! Though... you can see the `btn` styling doesn't really work well here. I won't do it, but you could clone the `Action` object and then customize each one.

I was wrong! Cloning will *not* work, due to the fact that "clones" are shallow in PHP... and the data inside an "action" object is stored in the internal `ActionDto` . Anyways, try this solution instead:

```
$viewAction = function() {
    return Action::new('view')
        ->linkToUrl(function(Question $question) {
            return $this->generateUrl('app_question_show', [
                'slug' => $question->getSlug(),
            ]);
        })
        ->setIcon('fa fa-eye')
        ->setLabel('View on site');
};

// ...

return parent::configureActions($actions)
    // ...
    ->add(Crud::PAGE_DETAIL, $viewAction()->addCssClass('btn btn-success'))
    ->add(Crud::PAGE_INDEX, $viewAction());
```

Okay, so creating an action that links somewhere is cool. But what about a *true* custom action that connects to a custom controller with custom logic... that does custom... stuff? Let's add a custom action that allows moderators to approve questions, next.

Chapter 36: True Custom Action

The whole point of this "Pending Approval" section is to allow moderators to approve or delete questions. We can *delete* questions... but there's no way to approve them. Sure, we could add a little "Is Approved" checkbox to the form. But a *true* "approve" action with a button on the detail or index pages would be *a lot* nicer. It would also allow us to run custom code on approval if we need to. So let's create another custom action.

Adding the Action as a Button

Over in `QuestionCrudController`, say `$approveAction = Action::new()` ... and I'll make up the word `approve`. Down at the bottom, add that to the detail page: `->add(Crud::PAGE_DETAIL, $approveAction)`.

155 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 21
22  class QuestionCrudController extends AbstractCrudController
23  {
... lines 24 - 37
38      public function configureActions(Actions $actions): Actions
39      {
... lines 40 - 48
49          $approveAction = Action::new('approve');
... line 50
51          return parent::configureActions($actions)
... lines 52 - 66
67              ->add(Crud::PAGE_DETAIL, $approveAction);
68      }
... lines 69 - 155
```

Before we try that, call `->addCssClass('btn btn-success')` and `->setIcon('fa fa-check-circle')`. Also add `->displayAsButton()`.

158 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
38      public function configureActions(Actions $actions): Actions
39      {
... lines 40 - 48
49          $approveAction = Action::new('approve')
50              ->addCssClass('btn btn-success')
51              ->setIcon('fa fa-check-circle')
52              ->displayAsButton();
... lines 53 - 70
71      }
... lines 72 - 158
```

By default, an action renders as a *link*... where the URL is wherever you want it to go. But in this case, we don't want approval to be done with a simple link that makes a "GET" request. Approving something will modify data on the server... and so it should really be a "POST" request. This will cause the action to render as a *button* instead of a link. We'll see how that works in a minute.

Linking to a CRUD Action

Ok, we *have* now created the action... but we need to link it to a URL or to a CRUD action. In this case, we need a CRUD action where we can write the approve logic. So say `linkToCrudAction()` passing the name of a method that we're going to create later. Let's call it `approve`.

159 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 37
38      public function configureActions(Actions $actions): Actions
39      {
... lines 40 - 48
49          $approveAction = Action::new('approve')
50              ->linkToCrudAction('approve')
... lines 51 - 71
72      }
... lines 73 - 159
```

Sweet! Refresh and... duh! The button won't be here... but if we go to the detail page...got it! "Approve"!

Overriding the Template to Add a Form

Inspect element and check out the source code. Yup! This literally rendered as a button... and that's it. There's no form around this... and no JavaScript magic to make it submit. We can click this all day long and absolutely *nothing* happens. To make it work, we need to wrap it in a form so that, on click, it submits a POST request to the new action.

How can we do that? By leveraging a custom template. We know that EasyAdmin has *lots* of templates. Inside EasyAdmin... in its `Resources/views/crud/` directory, there's an `action.html.twig` file. *This* is the template that's responsible for rendering *every* action. You can see that it's either an `a` tag or a `button` based on our config.

Copy the three lines on top that document the variables we have...and let's go create our *own* custom template. Inside `templates/admin/`, add a new file called `approve_action.html.twig`. Paste in the comments... and then... just to *further* help us know what's going on, dump that `action` variable: `dump(action)`.

5 lines | templates/admin/approve_action.html.twig

```
{# @var ea \EasyCorp\Bundle\EasyAdminBundle\Context\AdminContext #}
{# @var action \EasyCorp\Bundle\EasyAdminBundle\Dto\ActionDto #}
{# @var entity \EasyCorp\Bundle\EasyAdminBundle\Dto\EntityDto #}
{{ dump(action) }}
```

To *use* this template, over in `QuestionCrudController` ... right on the action, add `->setTemplatePath('admin/approve_action.html.twig')`.

160 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 21
22  class QuestionCrudController extends AbstractCrudController
23  {
... lines 24 - 37
38      public function configureActions(Actions $actions): Actions
39      {
... lines 40 - 48
49          $approveAction = Action::new('approve')
50              ->setTemplatePath('admin/approve_action.html.twig')
... lines 51 - 72
73      }
... lines 74 - 158
159 }
```

Let's try it. Refresh and... cool! We see the dump and *all* the data on that `ActionDto` object. The most important thing for *us* is `linkURL`. This contains the URL we can use to execute the `approve()` action that we'll create in a minute.

And because this new template is *only* being used by our *one* action... we're free to do whatever we want! All the other actions are still using the core `action.html.twig` template. Add a form... with `action="{{ action.linkUrl }}"` ... and then `method="POST"`. Inside, we need the button. We *could* create it ourselves... or we can be lazy and `{{ include('@EasyAdmin/crud/action.html.twig') }}`.

7 lines | templates/admin/approve_action.html.twig

```
... lines 1 - 3
<form action="{{ action.linkUrl }}" method="POST">
    {{ include('@EasyAdmin/crud/action.html.twig') }}
</form>
```

That's all we need! Reload the page... and inspect that element to see... exactly what we want: a `form` with the correct action... and our button inside. Though, we *do* need to fix the styling a little bit. Add `class="me-2"`.

7 lines | templates/admin/approve_action.html.twig

```
... lines 1 - 3
<form action="{{ action.linkUrl }}" method="POST" class="me-2">
... lines 5 - 7
```

Refresh and... looks better!

Try clicking this. We get... a *giant error*! Progress!

> The controller for URI "/admin" is not callable: Expected method "approve" on [our class].

Let's add that custom controller method next, and learn how to generate URLs to other EasyAdmin pages from inside PHP.

Chapter 37: Custom Controller & Generating Admin URLs

The final step to building our custom EasyAdmin action is to... *write* the controller method! In `QuestionCrudController`, all the way down at the bottom, this will be a *normal* Symfony action. You can pretend like you're writing this in a non-EasyAdmin controller class with a route above it. Say `public function approve()`. When the user gets here, the `id` of the entity will be in the URL. To help read that, autowire `AdminContext $adminContext`.

164 lines | src/Controller/Admin/QuestionCrudController.php

```
<?php

... lines 3 - 159
    public function approve()
    {
    }
}
```

Why are we allowed to add that argument? Because first, `AdminContext` is a service... just like the entity manager or the router. And second, the `approve()` method is a completely normal Symfony controller... so we're autowiring this service just like we would do with anything else.

Get the question with `$question = $adminContext->getEntity()->getInstance()`. And yes, sometimes, finding the data you need in `AdminContext` requires a little digging. Let's add a sanity check... (mostly for my editor): `if (!$question instanceof Question)`, throw a new `\LogicException('Entity is missing or not a Question')`. Now, we can very easily say `$question->setIsApproved(true)`.

170 lines | src/Controller/Admin/QuestionCrudController.php

```
<?php

... lines 3 - 159

    public function approve(AdminContext $adminContext)
    {
        $question = $adminContext->getEntity()->getInstance();
        if (!$question instanceof Question) {
            throw new \LogicException('Entity is missing or not a Question');
        }
        $question->setIsApproved(true);
    }
}
```

The last step is to save this entity... which looks completely normal! Autowire `EntityManagerInterface $entityManager` ... and then add `$entityManager->flush()`.

172 lines | src/Controller/Admin/QuestionCrudController.php

```php
<?php

... lines 3 - 159

    public function approve(AdminContext $adminContext, EntityManagerInterface $entityManager)
    {
        $question = $adminContext->getEntity()->getInstance();
        if (!$question instanceof Question) {
            throw new \LogicException('Entity is missing or not a Question');
        }
        $question->setIsApproved(true);

        $entityManager->flush();
    }
}
```

Rendering a Template

Sweet! Ok... but... what should we *do* after that? Well, we *could* render a template. Sometimes you'll create a custom action that is literally a new page in your admin section... and you would do that by rendering a template in a completely normal way. We already have an example of that inside `DashboardController`. The `index()` method is *really* a regular action... where we render a template. So if you wanted to render a template in a custom action,it would look pretty much exactly like this.

Generating an Admin Url

But in *our* situation, we want to redirect. And, we know how to do that from inside of a controller.But hmm, I want to redirect back to the "detail" page in the admin. In order to generate a URL to somewhere inside EasyAdmin,we need a special admin URL generator service that can help add the query parameters.

Let's autowire this: `AdminUrlGenerator $adminUrlGenerator`. Then `$targetUrl =` ... and build the URL by saying `$adminUrlGenerator`, `->setController(self::class)` - because we're going to link back to ourself - `->setAction(Crud::PAGE_DETAIL)`, `->setEntityId($question->getId())` ... and then *finally*, `->generateUrl()`.

There are a number of other methods you can call on this builder...but these are the most important. At the bottom `return $this->redirect($targetUrl)`.

181 lines | src/Controller/Admin/QuestionCrudController.php

```php
<?php

... lines 3 - 22
#[IsGranted('ROLE_MODERATOR')]
class QuestionCrudController extends AbstractCrudController
{
... lines 26 - 161
    public function approve(AdminContext $adminContext, EntityManagerInterface $entityManager, AdminUrlGenerator $adminUrlGenerato
    {
        $question = $adminContext->getEntity()->getInstance();
        if (!$question instanceof Question) {
            throw new \LogicException('Entity is missing or not a Question');
        }
        $question->setIsApproved(true);

        $entityManager->flush();

        $targetUrl = $adminUrlGenerator
            ->setController(self::class)
            ->setAction(Crud::PAGE_DETAIL)
            ->setEntityId($question->getId())
            ->generateUrl();

        return $this->redirect($targetUrl);
    }
}
```

Ok team, let's give this a try. Refresh and... got it! We're back on the detail page! And if we look for "Alice thought she might...", it's not on our "Pending Approval" page anymore!

Let's try one more to be sure: approve ID 23. Go to Show, click "Approve", and...it's *gone*. This is working!

Hiding Approve for Approved Question

The only weird thing now, which you *probably* saw, is that when you go to the detail page on an *already-approved* question... you still see the "Approve" button. Clicking on that doesn't hurt anything... but it's confusing! Fortunately, we know how to fix this.

Find your custom action... and add `->displayIf()`. Pass that a `static function()`, which will receive the `Question $question` argument... and return a `bool`. I've been a little lazy on my return types, but you can put that if you want. Finally, `return !$question->getIsApproved()`.

184 lines | src/Controller/Admin/QuestionCrudController.php

```
<?php

... lines 3 - 22
#[IsGranted('ROLE_MODERATOR')]
class QuestionCrudController extends AbstractCrudController
{
... lines 26 - 39
    public function configureActions(Actions $actions): Actions
    {
... lines 42 - 50
        $approveAction = Action::new('approve')
            ->setTemplatePath('admin/approve_action.html.twig')
            ->linkToCrudAction('approve')
            ->addCssClass('btn btn-success')
            ->setIcon('fa fa-check-circle')
            ->displayAsButton()
            ->displayIf(static function (Question $question): bool {
                return !$question->getIsApproved();
            });
... lines 60 - 77
    }
... lines 79 - 182
}
```

Move over now... refresh and... beautiful! The "Approve" button is *gone*. But when we go back to a question that *does* need to be approved, it's *still* there.

Custom Action JavaScript

If we wanted to, we could go further and write some JavaScript to make this fancier. For example, in our custom template, we could use the `stimulus_controller` function to reference a custom Stimulus controller. Then, when we click this button, we could, for example, open a modal that says:

> Are you sure you want to approve this question?

The point is, *we* control what this action, link, button, etc. look like. If you want to attach some custom JavaScript, do it.

Next, let's add a *global* action. A "global action" is something that applies to *all* of the items inside of a section. We're going to create a global *export* action that exports questions to CSV.

Chapter 38: A Global "Export" Action

There are actually *three* different types of actions in EasyAdmin. The first consists of the normal actions, like Add, Edit, Delete, and Detail. These operate on a single entity. The second type is *batch* actions, which operate on a *selection* of entities. For example, we can click two of these check boxes and use the Delete button up here. That is the *batch* Delete, and it's the only built-in batch action.

Side note: to make sure approved questions aren't deleted - which is work we just finished, you should also remove the batch Delete action for the Question crud. Otherwise, people might *try* to batch Delete questions. That won't work... thanks to some code we wrote, but they'll get a very-unfriendly 500 error.

Anyways, the third type of action is called a "global action", which operates on *all* the entities in a section. There are *no* built-in global actions, but we're going to add one: a button to "export" the entire questions list to a CSV file.

Creating the Global Action

For the most part, creating a global action... isn't much different than creating a normal custom action. It starts the same. Over in the actions config, create a new `$exportAction = Action::new()` and call it `export`. Below, we'll `->linkToCrudAction()` and *also* call it `export`. Then, add some CSS classes... and an icon. Cool. We're ready to add this to the index page: `->add(Crud::PAGE_INDEX, $exportAction)` to get that button on the main list page.

189 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 23
class QuestionCrudController extends AbstractCrudController
{
... lines 26 - 39
    public function configureActions(Actions $actions): Actions
    {
... lines 42 - 59
        $exportAction = Action::new('export')
            ->linkToCrudAction('export')
            ->addCssClass('btn btn-success')
            ->setIcon('fa fa-download');
... line 64
        return parent::configureActions($actions)
... lines 66 - 81
            ->add(Crud::PAGE_INDEX, $exportAction);
    }
... lines 84 - 187
}
```

If we stopped now, this would be a *normal* action. When we refresh... yup! It shows up next to each item in the list. *Not* what we wanted. To make it a *global* action, back in the action config, call `->createAsGlobalAction()`. You can also see how you would create a batch action.

190 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 39
    public function configureActions(Actions $actions): Actions
    {
... lines 42 - 59
        $exportAction = Action::new('export')
... lines 61 - 63
            ->createAsGlobalAction();
... lines 65 - 83
    }
... lines 85 - 190
```

Now refresh and... awesome!

Coding up the Custom Action

If we click the new button, we get a familiar error..because we haven't created that action yet. To help build the CSV file, we're going to install a third party library. At your terminal, say:

```
composer require handcraftedinthealps/goodby-csv
```

How's that for a great name? The "goodby-csv" library is a well-known CSV package... but it hasn't been updated for a while. So "handcraftedinthealps" forked it and made it work with modern versions of PHP. *Super* helpful!

If you downloaded the course code, you *should* have a `tutorial/` directory with a `CsvExporter.php` file inside. Copy that... and then, in your `src/Service/` directory, paste. This will handle the heavy lifting of creating the CSV.

65 lines | src/Service/CsvExporter.php

```
... lines 1 - 2
3  namespace App\Service;
... lines 4 - 12
13 class CsvExporter
14 {
15     public function createResponseFromQueryBuilder(QueryBuilder $queryBuilder, FieldCollection $fields, string $filename): Response
16     {
17         $result = $queryBuilder->getQuery()->getArrayResult();
18
19         // Convert DateTime objects into strings
20         $data = [];
21         foreach ($result as $index => $row) {
22             foreach ($row as $columnKey => $columnValue) {
23                 $data[$index][$columnKey] = $columnValue instanceof \DateTimeInterface
24                     ? $columnValue->format('Y-m-d H:i:s')
25                     : $columnValue;
26             }
27         }
... lines 28 - 62
63     }
64 }
```

At the bottom, this returns a `StreamedResponse` (that's a Symfony response)... that contains the file download with the CSV data inside. I won't go into the specifics of how this works...it's all related to the package we installed.

To call this method, we need to pass it three things: the `QueryBuilder` that should be used to query for the results, the `FieldCollection` (this comes from EasyAdmin and holds the fieldsto include), and also the filename that we want to use for the download. In `QuestionCrudController`, create that `export()` action: `public function export()`.

194 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 23
24 class QuestionCrudController extends AbstractCrudController
25 {
... lines 26 - 189
190     public function export()
191     {
192     }
193 }
```

Reusing the List Query Builder

Ok, step 1 is to create a QueryBuilder. We could simply inject the `QuestionRepository`, make a QueryBuilder... and pass that to `CsvExporter`. But we're going to do something a bit more interesting...and powerful.

When we click the "Export" button, I want to export *exactly* what we see in this list, including the current *order* of the items *and* any search parameter we've used to filter the results. To do that, we need to steal some code from our parent class. Scroll up to the top of the controller... and then hold "cmd" or "ctrl" to open `AbstractCrudController`. Inside, search for "index". There it is.

So `index()` is the action that renders the list page. And we can see some logic about how it makes its query. We want to replicate that. Specifically, we need these three variables: this is where it figures out which fields to show, which filters are being applied, and ultimately, where it creates the `QueryBuilder`. Copy those... go back to our `export()` action, and paste. I'll say "Okay" to add a few `use` statements.

To get this to work, we need a `$context`. That's the `AdminContext` which, as you probably remember, is something we can autowire as a service into our methods. Say `AdminContext` ... but this time, call it `$context`. Awesome!

201 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 9
use EasyCorp\Bundle\EasyAdminBundle\Collection\FieldCollection;
... lines 11 - 14
use EasyCorp\Bundle\EasyAdminBundle\Context\AdminContext;
... lines 16 - 25
class QuestionCrudController extends AbstractCrudController
{
... lines 28 - 193
    public function export(AdminContext $context)
    {
        $fields = FieldCollection::new($this->configureFields(Crud::PAGE_INDEX));
        $filters = $this->container->get(FilterFactory::class)->create($context->getCrud()->getFiltersConfig(), $fields, $context->getEntity());
        $queryBuilder = $this->createIndexQueryBuilder($context->getSearch(), $context->getEntity(), $fields, $filters);
    }
... lines 200 - 201
```

At this point, we have both the `QueryBuilder` and the `FieldCollection` that we need to call `CsvExporter`. So... let's do it! Autowire `CsvExporter $csvExporter` ... then, at the bottom, it's as simple as `return $csvExporter->createResponseFromQueryBuilder()` passing `$queryBuilder`, `$fields`, and then the filename. How about, `questions.csv`

202 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 7
use App\Service\CsvExporter;
... lines 9 - 26
class QuestionCrudController extends AbstractCrudController
{
... lines 29 - 192
    public function export(AdminContext $context, CsvExporter $csvExporter)
    {
... lines 195 - 198
        return $csvExporter->createResponseFromQueryBuilder($queryBuilder, $fields, 'questions.csv');
    }
}
```

Let's try it! Refresh... hit "Export" and... I think it worked! Let me open that up. Beautiful! We have a CSV of *all* of our data!

Forwarding Ordering & Filtering Query Params to the Action

But... there *is* one hidden problem. Notice the *ordering* of these items. In the CSV file... it seems like they're in a random order. But if we look at the list in the browser, these are ordered by ID! Try searching for something. Cool. 7 results. But if we export again... and open it... uh oh! We get the *same* long list of results! So the Search in the CSV export isn't working either!

The problem is this: the search term and any ordering that's currently applied is reflected in the URL via query parameters. But when we press the "Export" button, we only get the *basic* query parameters, like which CRUD controller or action is being called. We do *not* also get the filter, search, or order query parameters. So then, when we get the `$queryBuilder` and `$filter`, the parameters aren't there to tell EasyAdmin what filtering and ordering to do!

How can we fix this? By generating a *smarter* URL that *does* include those query parameters.

Up in `configureActions()`, instead of `->linkToCrudAction()`, let's `->linkToUrl()` and *completely* take control. Pass this a callback function. Inside, let's create the URL manually.

218 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 52
53      public function configureActions(Actions $actions): Actions
54      {
... lines 55 - 72
73          $exportAction = Action::new('export')
74              ->linkToUrl(function () {
... lines 75 - 79
80              })
... lines 81 - 102
103     }
... lines 104 - 218
```

You might remember that, to generate URLs to EasyAdmin,we need the `AdminUrlGenerator` service. Unfortunately, `configureActions()` isn't a real action - it's just a random methodin our controller - and so we can't autowire services into it.But no problem: let's autowire what we need into the constructor.

Add `public function __construct()` ... and then autowire `AdminUrlGenerator $adminUrlGenerator` and also `RequestStack $requestStack`. We're going to need that in a minute to get the `Request` object. Hit "alt" + "enter" and go to "Initialize properties" to create both of those properties and set them.

218 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 22
23  use EasyCorp\Bundle\EasyAdminBundle\Router\AdminUrlGenerator;
24  use Symfony\Component\HttpFoundation\RequestStack;
... lines 25 - 27
28  class QuestionCrudController extends AbstractCrudController
29  {
30      private AdminUrlGenerator $adminUrlGenerator;
31      private RequestStack $requestStack;
32
33      public function __construct(AdminUrlGenerator $adminUrlGenerator, RequestStack $requestStack)
34      {
35          $this->adminUrlGenerator = $adminUrlGenerator;
36          $this->requestStack = $requestStack;
37      }
... lines 38 - 216
217 }
```

Back down in `configureActions()` ... here we go... inside `->linkToUrl()`, get the request: `$request = $this->requestStack->getCurrentRequest()`. Then, for the URL, create it from scratch: `$this->adminUrlGenerator`, then `->setAll($request->query->all()`. This starts by generating a URL that has *all* of the same query parameters as the current request. Now, override the action - `->setAction('export')` and then `->generateUrl()`.

218 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 52
53      public function configureActions(Actions $actions): Actions
54      {
... lines 55 - 72
73          $exportAction = Action::new('export')
74              ->linkToUrl(function () {
75                  $request = $this->requestStack->getCurrentRequest();
76
77                  return $this->adminUrlGenerator->setAll($request->query->all())
78                      ->setAction('export')
79                      ->generateUrl();
80              })
... lines 81 - 102
103     }
... lines 104 - 218
```

Basically, this says:

> Generate the same URL that I have now... but change the action to point to `export`.

Testing time! Refresh the page. We *should* have 7 results. Export, open that up and... yes! Got it! It shows the *same* results... and in the *same* order as what we saw on the screen!

Next, let's learn to re-order the actions themselves and generate a URLfrom our frontend show page so that we can have an "edit" button right here for admin users.

Chapter 39: Linking to EasyAdmin from Twig

Let's go look at the "Show" page for a question that is*not* approved yet. We have a lot of buttons up here...which is fine. But what if we don't like their order? Like, Delete might make more sense as the last button instead of in the middle.

Ordering Actions

No problem! We can control this inside of `configureActions()`. At the bottom, after we've set up the actions, call another method - `->reorder()` - and pass this the page that we want to reorder them on. In this case, it's `Crud::PAGE_DETAIL`. Then, very simply, add the names of the actions. We have quite a few... let's start with `approve`, `view` ... and then the three built-in actions: `Action::EDIT`, `Action::INDEX`, and `Action::DELETE`. These are the five actions that correspond to these five buttons.

225 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 27
class QuestionCrudController extends AbstractCrudController
{
... lines 30 - 52
    public function configureActions(Actions $actions): Actions
    {
... lines 55 - 84
        return parent::configureActions($actions)
... lines 86 - 102
            ->reorder(Crud::PAGE_DETAIL, [
                'approve',
                'view',
                Action::EDIT,
                Action::INDEX,
                Action::DELETE,
            ]);
    }
... lines 111 - 223
}
```

Adding Action Icons to the Entire Admin

Now when we refresh... very nice! Though... I'm noticing that it looks a bit odd that some of these have icons and others don't. Let's see if we can add an icon to the Edit and Index actions across our*entire* admin.

If we want to modify something for all of our admin,we need to do it inside of `DashboardController`. As we saw earlier, to modify a *built-in* action, we can call the `->update()` function. Pass this the page - `Crud::PAGE_DETAIL` - the action - `Action::EDIT` - and then a `static function` with an `Action $action` argument. Inside, modify and return the `Action` at the same time: `return $action->setIcon('fa fa-edit')`.

146 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 24
class DashboardController extends AbstractDashboardController
{
... lines 27 - 100
    public function configureActions(): Actions
    {
        return parent::configureActions()
... line 104
            ->update(Crud::PAGE_DETAIL, Action::EDIT, static function (Action $action) {
                return $action->setIcon('fa fa-edit');
            })
... lines 108 - 110
    }
... lines 112 - 144
}
```

Let's do the same thing one more time for the index action button: use `Action::PAGE_INDEX` ... and we'll give this `fa fa-list` .

146 lines | src/Controller/Admin/DashboardController.php

```
... lines 1 - 100
    public function configureActions(): Actions
    {
        return parent::configureActions()
... lines 104 - 107
            ->update(Crud::PAGE_DETAIL, Action::INDEX, static function (Action $action) {
                return $action->setIcon('fa fa-list');
            });
    }
... lines 112 - 146
```

Refresh now and... I love it! We see the icons here... and if we go anywhere else - like to an Answer's detail page - the icons are here too.

Adding a Link to the Admin From Twig

At this point, we know how to generate a link to any EasyAdminBundle page. If I scroll up a bit... the key is to get the `AdminUrlGenerator` , and then set whatever you need on it, like the action and CRUD controller.

Go to the Homepage... and click into a question. To make life easier for admin users, I want to put an "Edit" button that takes us directly to the edit action for this specific question. So... how do we generate URLs to EasyAdmin from *Twig*?

Open the template for this page - `templates/question/show.html.twig` - and find the `<h1>` . Here it is. For organization, I'll wrap this in a `<div>` with `class="d-flex justify-content-between"` .

95 lines | templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 37
                    <div class="col">
                        <div class="d-flex justify-content-between">
                            <h1 class="q-title-show">{{ question.name }}</h1>
... lines 41 - 46
                        </div>
... lines 48 - 52
                    </div>
... lines 54 - 93
{% endblock %}
```

After the `h1` , add the link... but only for admin users. So `{% if is_granted('ROLE_ADMIN') %}` ... and `{% endif %}` . Inside `<a href="">` - I'll leave the `href` empty for a moment - with `class="text-white"` . And inside of *that*, a `<span class="fa fa-edit">` .

95 lines | templates/question/show.html.twig

```
... lines 1 - 4
{% block body %}
... lines 6 - 38
                    <div class="d-flex justify-content-between">
... lines 40 - 41
                        {% if is_granted('ROLE_ADMIN') %}
                            <a class="text-white" href="">
                                <span class="fa fa-edit"></span>
                            </a>
                        {% endif %}
                    </div>
... lines 48 - 93
{% endblock %}
```

Back in our browser, try that. And... hello edit link!

To generate the URL, we need to tell EasyAdminwhich CRUD controller, action, and entity ID to use...all stuff we've done in PHP. In Twig, it's *nearly* the same thing thanks to a shortcut function called `ea_url()` .

This gives us the `AdminUrlGenerator` object. And so, we can just...call the normal methods, like `.setController()` ... passing the *long* controller class name. We have to use double slashes so that they don't get escaped, since we're inside of a stringNow add `.setAction('edit')` and `.setEntityId(question.id)` .

99 lines | templates/question/show.html.twig

```
... lines 1 - 41
                        {% if is_granted('ROLE_ADMIN') %}
                        <a class="text-white" href="{{ ea_url()
                            .setController('App\\Controller\\Admin\\QuestionCrudController')
                            .setAction('edit')
                            .setEntityId(question.id)
                        }}">
                                <span class="fa fa-edit"></span>
                            </a>
                        {% endif %}
... lines 51 - 99
```

It's a little weird to write this kind of code in Twig, but that's how it's doneBack over here, refresh... and try the button. Got it!

Ok team, last topic! Let's talk about how we can leverage layout panels and other goodies to organize our forminto groups, rows, or even tabs on this form page.

Chapter 40: Form Panels

Last topic! We made it! And our admin is getting *super* customized. For this final trick, I want to look closer at the form. Almost all of this is controlled by the `Field` configuration. Each field corresponds to a Symfony form type... and then EasyAdmin renders those fields through the form system. It really *is* that simple.

Custom Form Theme

EasyAdmin comes with a custom form theme. So if you wanted to, for example, make a text type field look different in EasyAdmin, you could create a *custom* form theme template. This theme can be added to the `$crud` object in `configureCrud()`. Down here, for example, we could say `->addFormTheme()` to add our form theme template to just *one* CRUD controller... *or* you could put this in the dashboard to apply everywhere.

Form Panel

But, apart from a custom form theme, there *are* a few *other* ways that EasyAdmin allows us to control what this page looks like... which, right now, is just a long list of fields.

Over in `QuestionCrudController`, up in `configureFields()` ... here we go... right before the `askedBy` field, add `yield FormField::`. So we're starting like normal, but instead of saying `new`, say `addPanel('Details')`.

227 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 28
class QuestionCrudController extends AbstractCrudController
{
... lines 31 - 112
    public function configureFields(string $pageName): iterable
    {
... lines 115 - 140
        yield FormField::addPanel('Details');
... lines 142 - 161
    }
... lines 163 - 225
}
```

Watch what this does! Refresh and... cool! "Asked By" and "Answers" appear under this "Details" header. That's because, as you can see, `askedBy` and `answers` are the two fields that appear *after* the `addPanel()` call. And because the rest of these fields are *not* under a panel, they just... kind of appear at the bottom, which *works*, but doesn't look the greatest.

So, when I use `addPanel()`, I put *everything* under a panel. Right after `IdField`, which *isn't* going to appear on the form, say `FormField::addPanel('Basic Data')`. Oh! And let me make sure I don't forget to `yield` that.

228 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 112
    public function configureFields(string $pageName): iterable
    {
... lines 115 - 116
        yield FormField::addPanel('Basic Data');
... lines 118 - 162
    }
... lines 164 - 228
```

Thanks to this... awesome! We have a "Basic Data" panel, all of the fields below that, then the second panel down here.

Customizing the Panels

These panels have a few methods on them. One of the most useful is `->collapsible()`. Make this panel collapsible... and the other as well.

230 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 112
    public function configureFields(string $pageName): iterable
    {
... lines 115 - 116
        yield FormField::addPanel('Basic Data')
            ->collapsible();
... lines 119 - 142
        yield FormField::addPanel('Details')
            ->collapsible();
... lines 145 - 164
    }
... lines 166 - 230
```

I bet you can guess what this does. Yep! We get a nice way to collapse each section.

What else can we tweak? How about `->setIcon('fa fa-info')` ... or `->setHelp('Additional Details)` ?

Oh, I actually meant to put this down on the other panel, so let me grab this..find that other panel... here we go... and paste.

232 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 112
    public function configureFields(string $pageName): iterable
    {
... lines 115 - 142
        yield FormField::addPanel('Details')
            ->collapsible()
            ->setIcon('fa fa-info')
            ->setHelp('Additional Details');
... lines 147 - 166
    }
... lines 168 - 232
```

Let's check it out! Nice! The second panel has an icon and some sub-text.

By the way, the changes we're making not only affect the form page, but also the Detail pageGo check out the Detail page for one of these. Yup! The same organization is happening here, which is nice.

Form Tabs

If you want to organize things even a bit *more*, instead of panels, you can use tabs. Change `addPanel()` to `addTab()` . And... repeat that below: `addTab()` .

232 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 112
    public function configureFields(string $pageName): iterable
    {
... lines 115 - 116
        yield FormField::addTab('Basic Data')
... lines 118 - 142
        yield FormField::addTab('Details')
... lines 144 - 166
    }
... lines 168 - 232
```

When we refresh now... yup! Each shows up as a separate tab. But the `->collapsible()` doesn't really make sense anymore. It *is* still being called, but it doesn't do anything. So, remove that.

Fixing the Icon on the Tab

Oh, and we also lost our icon! We added an `fa fa-info` icon... but it's not showing! Or *is* it? If you look closely, there's some extra space. Inspect element on that. There *is* an icon! But... it looks... weird. It has an extra `fa-fa` for some reason.

We can fix this by changing the icon to, simply, `info`. This is... sort of a bug. Or, it's at least inconsistent. When we use tabs, EasyAdmin adds the `fa-` for us. So all we need is `info`. Watch: when I refresh... there! `fa-info` ... and *now* the icon shows up!

231 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 112
113     public function configureFields(string $pageName): iterable
114     {
... lines 115 - 142
143         yield FormField::addTab('Details')
144             ->setIcon('info')
... lines 145 - 165
166     }
... lines 167 - 231
```

Form Columns

The *last* thing we can do, instead of having this long list of fields,is to put the fields *next* to each other. We do this by controlling the *columns* on this page.

To show this off, move the `name` field above `slug`. Yup, got it. And now let's see if we can put these fields*next* to each other. We're using bootstrap, which means there are 12 invisible columns on each page.So, on `name`, say `->setColumns(5)` ... and on `slug`, do the same thing: `->setColumns(5)`.

233 lines | src/Controller/Admin/QuestionCrudController.php

```
... lines 1 - 112
113     public function configureFields(string $pageName): iterable
114     {
... lines 115 - 119
120         yield Field::new('name')
... line 121
122             ->setColumns(5);
123         yield Field::new('slug')
... lines 124 - 128
129             ->setColumns(5);
... lines 130 - 167
168     }
... lines 169 - 233
```

We could use `6` to take up *all* of the space, but I'll stick with `5` and give it some room. Refresh now and... very nice! The fields float next to each other. This is a great way to help this page...make a bit more sense.

And... that's it, friends! We are *done*! This was fun! We should do it again sometime. I *love* EasyAdmin, and we here at SymfonyCasts are *super* proud of the admin section we built with it...which includes *a lot* of custom stuff. Let us know what you're building! And as always, we're here for you down in the Comments section with any questions, ideas,or delicious recipes that you might have.

All right friends, see you next time!