FOSUserBundle FTW!

With <3 from SymfonyCasts

Chapter 1: Rock some FOSUserBundle!

The most *popular* bundle in *all* of Symfony is... GifExceptionBundle! Wait... that's not right... but it *should* be. The *actual* most popular bundle is, of course, FOSUserBundle. And it's easy to know why: it gives you *a lot* of crazy-cool stuff, out-of-the-box: registration, reset password, change password, and edit profile pages. Honestly, it feels like stealing.

But guess what? A lot of smart devs don't like FOSUserBundle. How could that be? The bundle *does* give you a lot of stuff. But, it's not a magician: it doesn't know what your design looks like, what fields your registration form should have, the clever text you want in your emails or where to redirect the user after registration.

See, there's a lot of stuff that makes your site *special*. And that means that if you're going to use FOSUserBundle - which *is* awesome - you're going to need to customize a lot of things. And you've come to the right place: once we're done, you're going to be *embarrissingly* good extending this bundle. I mean, your co-workers will gaze at you in amazement as you hook into events, customize text and override templates. It's going to be beautiful thing.

Let's do it!

Code along with me yo!

As always, you should *totally* code along with me... it's probably what the cool kids are doing. Just click the download button on this page and unzip that guy. Inside, you'll find a `start/` directory that has the same code you see here. Open up `README.md` for hilarious text... and setup details.

The last step will be to find your favorite terminal and run:

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

to start the built-in PHP web server. Ok, load this up in your browser: `http://localhost:8000`.

Welcome to AquaNote! This is the same project we've been building in our main Symfony tutorials, but *without* any security logic. Gasp! See that Login link? It's a lie! It goes nowhere! The login link is a lie!

Google for the FOSUserBundle documentation: it lives right on Symfony.com. And make sure you're on the 2.0 version.

Installing & Enabling the Bundle

We'll go through the install details... but in our own order. Of course, first, copy the `composer require` line... but don't worry about the version number. Head over to your terminal and run that:

```
$ composer require friendsofsymfony/user-bundle
```

While we're waiting for Jordi to prepare out delicious FOSUserBundle package, go back and copy the `new FOSUserBundle()` line from the docs, open our `app/AppKernel.php` file and paste it to enable the bundle. Oh, and FOSUserBundle uses `SwiftmailerBundle` to send the password reset and registration confirmation emails. So, uncomment that. You can also create your own custom mailer or tell FOSUserBundle to not send emails.

58 lines | app/AppKernel.php

```php
<?php
... lines 2 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 14
            new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
... lines 16 - 21
            new FOS\UserBundle\FOSUserBundle(),
... lines 23 - 24
        );
... lines 26 - 35
    }
... lines 37 - 56
}
```

Ok, flip back to your terminal. Bah! It exploded! Ok, it *did* install FOSUserBundle. So, let's not panic people. It just went crazy while trying to clear the cache:

> The child node `db_driver` at path `fos_user` must be configured

Ah, so this bundle has some *required* configuration.

Create your User Entity

Before we fill that in, I have a question: what does FOSUserBundle *actually* give us? In reality, just 2 things: a `User` entity and a bunch of routes and controllers for things like registration, edit password, reset password, profile and a login page.

To use the `User` class from the bundle, we need to create our own small `User` class that extends their's.

Inside `src/AppBundle/Entity`, create a new PHP class called `User`. To extend the base class, add a `use` statement for their `User` class with `as BaseUser` to avoid a lame conflict. Then add, `extends BaseUser`.

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

```php
<?php
... line 2
namespace AppBundle\Entity;
... lines 4 - 5
use FOS\UserBundle\Model\User as BaseUser;
... lines 7 - 11
class User extends BaseUser
{
... lines 14 - 24
}
```

There's just one thing we *must* do in this class: add a `protected $id` property. Beyond that, this is just a normal entity class. So I'll go to the Code->Generate menu - or Command+N on a Mac - and choose `ORM Class` to get my fancy `@ORM\Entity` stuff on top. Add ticks around the `user` table name - that's a keyword in some database engines.

```
26 lines | src/AppBundle/Entity/User.php
<?php
... lines 2 - 4
use Doctrine\ORM\Mapping as ORM;
... lines 6 - 7
/**
 * @ORM\Entity
 * @ORM\Table(name="`user`")
 */
class User extends BaseUser
{
... lines 14 - 18
    protected $id;
... lines 20 - 24
}
```

Now, go back to Code->Generate, choose `ORM Annotation` and select the `id` column. Boom! We are annotated! Finally, go back to Code->Generate *one* last time... until we do it more later - and generate the `getId()` method.

```
26 lines | src/AppBundle/Entity/User.php
<?php
... lines 2 - 11
class User extends BaseUser
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    protected $id;

    public function getId()
    {
        return $this->id;
    }
}
```

This class is done!

Tip

Actually, this is unnecessary - the base `User` class already has a `getId()` method.

Adding the Required Configuration

And now we have everything we need to add the required configuration. In the docs, scroll down a little: under the Configure section, copy their example config. Then, back in your editor, open `app/config/config.yml`, and paste this down at the bottom.

Ok, The `db_driver` *is* `orm` and the `firewall_name` - `main` - is also correct. You can see that key in `security.yml`.

And yea, the `user_class` is also correct. We're crushing it! For the email stuff, it doesn't matter, use `hello@aquanote.com` and `AquaNote Postman`.

```
82 lines | app/config/config.yml
... lines 1 - 74
fos_user:
    db_driver: orm
    firewall_name: main
    user_class: AppBundle\Entity\User
    from_email:
        address: "hello@aquanote.com"
        sender_name: "AquaNote Postman"
```

Generate the Migration

Finally, our app should be un-broken! Try the console:

```
$ php bin/console
```

It's alive! Now, we can generate the migration for our `User` class:

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

Tip

If you get a "Command not Found" error, just install the DoctrineMigrationsBundle.

Yep, that looks about right. Run it:

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

Perfect!

Importing the Routing

At this point, the *only* thing this bundle has given us is the base `User` class... which is nice, but nothing too special, it has a bunch of properties like `username`, `email`, `password`, `lastLogin`, etc.

The *second* thing this bundle gives us is a bunch of free routes and controllers. But to get those, we need to *import* the routes. Back in the documentation, scroll down a bit until you see step 6: Import FOSUserBundle routing files. Ah ha! Copy that routing import.

Find your `app/config/routing.yml` file and paste that on top.

12 lines | app/config/routing.yml

```
... lines 1 - 9
fos_user:
    resource: "@FOSUserBundle/Resources/config/routing/all.xml"
```

As *soon* as you do that, we have new routes! At your terminal, check them out:

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

Awesome! We have `/login`, `/profile/edit`, `/register` and others for resetting and changing your password. If we manually go to `/register` in the browser... yea! A functional registration form. I know, it's *horribly*, *embarrassingly* ugly: we'll fix that.

Oh, and back on the docs, all the way at the bottom, there's a page about Advanced routing configuration. Open that in a new tab. In your app, you may not need *all* of the pages that FOSUserBundle gives you. Maybe you need registration and reset password, but you don't need a profile page. No problem! Instead of importing this `all.xml` file, just import the specific routes you want. Seriously feel free to do this: if some route or controller isn't helping you, kill it.

Enabling the Translator

Oh, and if your registration page doesn't look mine - if it has some weird keys instead of real text, don't worry. In `app/config/config.yml`, just make sure to uncomment the `translator` key under `framework`.

FOSUserBundle uses the translator to translate internal "keys" into English or whatever other language.

A *little* bit of Security

And basically... at this point... we're done installing the bundle! But how is that possible? We haven't touched anything related to security!

Here's the truth: this bundle has almost *nothing* to do with security: it just gives you a `User` class and some routes & controllers! We could already register, reset our password or edit our profile without doing any more setup.

Well, that's *almost* true: we do need a *tiny* bit of security to make registration work. In `security.yml`, add an `encoders` key with `AppBundle\Entity\User` set to `bcrypt`. When we register, FOSUserBundle needs to encode the plain-text password before saving it. This tells it what algorithm to use.

```
32 lines | app/config/security.yml
... lines 1 - 2
security:

    encoders:
        AppBundle\Entity\User: bcrypt

... lines 8 - 32
```

There's one other small bit of security we need right now. In the documentation, under step 4, copy the `providers` key. Paste that over the old `providers` key in `security.yml`.

```
32 lines | app/config/security.yml
... lines 1 - 2
security:

... lines 5 - 7
    # http://symfony.com/doc/current/book/security.html#where-do-users-come-from-user-providers
    providers:
        fos_userbundle:
            id: fos_user.user_provider.username
... lines 12 - 32
```

I'll talk about this more in a few minutes, but it's needed for registration only because FOSUserBundle logs us in after registering... which is really nice!

In the next chapter, we'll do more security setup. But for right now, that's all we need!

So try it out! Refresh `/register`. Let's use `aquanaut1@gmail.com`, `aquanaut1`, and password `turtles`. And boom! We are registered *and* logged in! We even have a role: `ROLE_USER`. More on that later.

With a `User` class, a route import and a *tiny* bit of security, everything in the bundle works: registration, reset password, edit password and edit profile.

The only thing you *can't* do is log out... or login. But that's not FOSUserBundle's fault: setting up security is *our* job. Let's do it next.

Chapter 2: Security Setup

With FOSUserBundle setup, the only things we *can't* do is login and logout. FOSUserBundle *does* give us a `/login` page, but it's just a static HTML form: setting up the actual authentication part is *entirely* up to us. And that's why, if you try to login now, you get this angry message!

To prove how little FOSUserBundle is doing, go to `/login`. If you hover over your web debug toolbar, you can see that the controller behind this page is called `SecurityController`. Cool! In my editor, I'll find that file by filename - Shift+Shift in PHP Storm.

Sweet! See `loginAction`? This renders the login page. And *all* it does is check for any authentication errors stored in the session and render a template. It has *no* logic *whatsoever* for processing the form submit, logging in, or anything else related to security.

Configuring Logout

So let's *finally* add some security goodness, starting with logging out. Right now, if you go to `/logout`, you see an error message. This is coming from that same controller: FOSUserBundle gives us a `/logout` route, but its controller is never supposed to be called. To fix this, in `security.yml`, add `logout: ~`. That's it.

31 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 12
    firewalls:
... lines 14 - 18
        main:
... lines 20 - 22
            logout: ~
... lines 24 - 31
```

Try going to `/logout` again. It works! We are anonymous! By adding the `logout` key, Symfony is now waiting for us to go to `/logout`. When we do, it *intercepts* the request and logs us out. Other than giving us the `/logout` route, FOSUserBundle has nothing to do with this.

Configuring form_login Security

What about logging in? It's the same thing. Under your firewall, add `form_login`. That's actually all you need. But, I'll add a bit more: `csrf_token_generator: security.csrf.token_manager`. That will make sure the CSRF token - which is already added in the FOSUserBundle login template - is verified when we submit.

31 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 12
    firewalls:
... lines 14 - 18
        main:
... lines 20 - 24
            form_login:
                csrf_token_generator: security.csrf.token_manager
... lines 27 - 31
```

As *soon* as we do that, go to `/login` and login with `aquanaut1` password `turtles`. Winning! We are in! FOSUserBundle gives us a login form, but *we* need to take care of the rest... which is pretty easy.

Adding Remember Me Functionality

Oh, and on the login form, we also have a remember me checkbox. If you want this to work, you'll need to add one more setting: `remember_me:` with `secret: '%secret%'` to use the secret from `parameters.yml`.

34 lines | app/config/security.yml

```
... lines 1 - 2
security:
... lines 4 - 12
    firewalls:
... lines 14 - 18
        main:
... lines 20 - 24
            remember_me:
                secret: '%secret%'
... lines 27 - 34
```

Ok, so about 5 lines to get our entire security system working. That kicks butt! And now, we can hook up the login link for real. Open `app/Resources/views/base.html.twig` and find the static link. Add an if statement: `if is_granted('ROLE_USER')`, then `else` and `endif`. FOSUserBundle guarantees that every user always at least has `ROLE_USER`. So it's safe to use this to figure out whether or not the user is logged in.

53 lines | app/Resources/views/base.html.twig

```
<!DOCTYPE html>
<html>
... lines 3 - 13
    <body>
... lines 15 - 19
        <header class="header">
... lines 21 - 22
            <ul class="navi">
... line 24
                {% if is_granted('ROLE_USER') %}
... line 26
                {% else %}
... line 28
                {% endif %}
            </ul>
        </header>
... lines 32 - 50
    </body>
</html>
```

For the logout link, use the route `fos_user_security_logout`, then we'll say "Logout". Oh, put all of this stuff inside an `li` tag. If you run:

53 lines | app/Resources/views/base.html.twig

```
... lines 1 - 19
        <header class="header">
... lines 21 - 22
            <ul class="navi">
... line 24
                {% if is_granted('ROLE_USER') %}
                    <li><a href="{{ path('fos_user_security_logout') }}">Logout</a></li>
... lines 27 - 28
                {% endif %}
            </ul>
        </header>
... lines 32 - 53
```

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

you can see that this is one of the routes we imported. Use a similar one for login: just copy the logout line, and change it to login.

```
53 lines | app/Resources/views/base.html.twig
... lines 1 - 19
        <header class="header">
... lines 21 - 22
            <ul class="navi">
... line 24
                {% if is_granted('ROLE_USER') %}
... line 26
                {% else %}
                    <li><a href="{{ path('fos_user_security_login') }}">Login</a></li>
                {% endif %}
            </ul>
        </header>
... lines 32 - 53
```

Nice! Go back, and refresh. Hit Logout! Woohoo!

Next, let's see what this looks like in the database, and talk about roles.

Chapter 3: Dynamic Roles and Canonical Fields

Let's see what this all looks like in the database! To query the user table, we can actually use the console:

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

Nice! We inherited a bunch of columns from the base `User` class, like `username`, `email` and `enabled`. It even tracks our last login. Thanks!

Canonical Fields!?

But there are two weird fields: `username_canonical` and `email_canonical`. What the heck? These are one of the more controversial things about FOSUserBundle. Before we explore them, first, just know that when you set `username` or `email`, the corresponding canonical field is automatically set for you. So, these canonical fields are not something you normally need to worry or think about.

So why do they exist? Suppose that when you registered, you used some capital letters in your username. The `username` column will be exactly as you typed it: *with* the capital letters. But `username_canonical` will be lowercased. Then, when you login, FOSUserBundle lowercases the submitted username and queries via the `username_canonical` column.

Why? Because some databases - like Postgresql - are case *sensitive*. The canonical fields allow a user to login with a case *insensitive* username - `aquanaut` in all lowercase, uppercase or any combination.

But mostly... this is just a detail you shouldn't think about. It's all handled for you and other than being ugly in the database, it doesn't hurt anything.

Logging in with Username or Email

And by the way, right now you can *only* login with your `username`. If you want to be able to login with `username` or `email`, no problem! The documentation has a section about this. Just change your user provider to `fos_user.user_provider.username_email`.

What does this do? When you submit your login form, the `provider` section is responsible for taking what you entered and finding the correct `User` record. Our current user provider finds the `User` by the `username_canonical` field. This other one looks up a `User` by username *or* email. And you're 100% free to create your *own* user provider, if you need to login with some other, weird logic. FOSUserBundle won't notice or care.

Dynamic Roles

Check out the database result again and look at `roles`. I know, it's strange: this is an `array` field. I'll hold command and click to open the base `User` class from FOSUserBundle. See, `roles` holds an array. When you save, it automatically serializes to a string in the database. This is done with the Doctrine `array` field type.

Notice that even though it's empty in the database, when we login, our user has `ROLE_USER`. This is thanks to the base `User` class from FOSUserBundle: it makes sure the `User` has whatever roles are stored in the database *plus* `ROLE_USER`.

Creating an Admin User

Let's try an example of a `User` that has a different role. Run the console:

```
$ php bin/console
```

Ah, so the bundle comes with a few handy console commands, for activating, creating, promoting and demoting users. Let's create a new one:

```
$ php bin/console fos:user:create
```

How about `admin`, `admin@aquanote.com` and password `admin`.

And now promote it!

```
$ php bin/console fos:user:promote
```

Hmm, let's give `admin`, `ROLE_ADMIN`. Ok, try the query again:

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

Booya! Our new user has `ROLE_ADMIN`! Quick, go login! Well, logout first, then go login! Use `admin` and `admin`. Woohoo! We have *both* `ROLE_USER` and `ROLE_ADMIN`.

In your app, if you want to give different roles to your users, you have 2 options. First, via the command line by using `fos:user:promote`. If you only have a few users that need special permissions, this is a great option. Or, you can create a user admin area and use the `ChoiceType` with the `'multiple' => true` and `'expanded' => true` to select the roles as checkboxes.

Ok, time to squash the ugly and make the FOSUserBundle pages use our layout!

Chapter 4: Layout and Template Customization

Everything *works*, but if you go to `/register`... it looks *awful*. Well, of *course* it looks awful! FOSUserBundle has *no* idea how the page should be styled. But don't worry: we can get this looking *much* better, very quickly.

First, on the web debug toolbar, find the template icon and click it. This will show you all the templates used to render this page... which is a *beautiful* cheat sheet for knowing what templates you can override!

Correcting the Base Layout

The one I'm interested in is `layout.html.twig`, which lives in FOSUserBundle.

In my editor, I'll press Shift+Shift to open that file. Ok, *every* Twig template in FOSUserBundle extends *this* `layout.html.twig` file. For example, see the "Logged in as" text? That's coming from here.

But, we want all of FOSUserBundle's templates to instead extend *our* `base.html.twig` template. How can we do that?

Overriding the Layout

By overriding `layout.html.twig`. Let's see how. First, to override *any* template from a bundle, just go to `app/Resources`, then create a directory with the same name as the bundle: `FOSUserBundle`. Inside, create one more directory: `views`.

Tip

The location where templates should live to override bundle templates has changed in Symfony 4. But, the idea is still the same. For details, see: https://symfony.com/blog/new-in-symfony-3-4-improved-the-overriding-of-templates.

In this case, the `layout.html.twig` template lives right the root of the `views/` directory in the bundle. So that's where we need to create our's. Inside, extend the normal `base.html.twig`.

6 lines | app/Resources/FOSUserBundle/views/layout.html.twig

```
{% extends 'base.html.twig' %}
... lines 2 - 6
```

Here's the magic part. Hit Shift+Shift again and open `register.html.twig`: this is the template for the registration page. Notice that it overrides a block called `fos_user_content`. In `layout.html.twig`, this is printed in the middle.

So check this out: inside of our version of `layout.html.twig`, add `{% block body %}`: that's the name of the block in our `base.html.twig`. Then, include `{% block fos_user_content %}` and `{% endblock %}`.

6 lines | app/Resources/FOSUserBundle/views/layout.html.twig

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

{% block body %}
    {% block fos_user_content %}{% endblock %}
{% endblock %}
```

We're effectively *transferring* the content from the `fos_user_content` block to the correct block: `body`.

These 5 lines of code are *huge*. Refresh the page! Ha! So much better! Not perfect, but every page now lives in our layout. If you want, you can even add a little more markup around the block.

```
12 lines | app/Resources/FOSUserBundle/views/layout.html.twig
... lines 1 - 2
{% block body %}
    <div class="container">
        <div class="row">
            <div class="col-xs-12">
                {% block fos_user_content %}{% endblock %}
            </div>
        </div>
    </div>
{% endblock %}
```

Overriding Individual Templates

Overriding the base layout is step one. But, each individual page still won't look quite right. On this page, we *at least* need a "Register" `h1`, and I'd like to make that button look better.

So in addition to overriding `layout.html.twig`, you really need to override every template from FOSUserBundle that you use - like registration, reset password, login and a few others.

Once again, click the templates link in the web debug toolbar. The template behind this page is `register.html.twig`, which we already have open. But notice, it immediately includes `register_content.html.twig`. This is a really common pattern in this bundle.

Let me show you: I'll click the `views` link to move my tree into FOSUserBundle. In `Registration`, we have `register.html.twig` and `register_content.html.twig`. In `Profile` there's the same for edit and show.

In most cases, you'll want to override the `_content.html.twig` template. Why? Well, it doesn't really matter: by overriding the `_content.html.twig` template, you don't need to worry about extending anything: you can just focus on the content.

Copy the contents of `register_content.html.twig`. Then, back in `app/Resources/views`, create a `Registration` directory. I'm doing that because this template lives in a `Registration` directory. Finally, create `register_content.html.twig` and paste in the content. Let's add a couple of classes to the button and an h1 that says: "Register Aquanauts!"

```
11 lines | app/Resources/FOSUserBundle/views/Registration/register_content.html.twig
{% trans_default_domain 'FOSUserBundle' %}

<h1>Register Aquanaut!</h1>

{{ form_start(form, {'method': 'post', 'action': path('fos_user_registration_register'), 'attr': {'class': 'fos_user_registration_regist
    {{ form_widget(form) }}
    <div>
        <input class="btn btn-primary" type="submit" value="{{ 'registration.submit'|trans }}" />
    </div>
{{ form_end(form) }}
```

Ok, refresh! Love it! In your app, make sure to do this for all of the different pages from the bundle that you're using. And remember, if you don't need a page - like the edit profile page - save yourself some time by *not* importing its route or overriding its template.

Chapter 5: Customizing Text via Translations

Create a new user: `aquanaut2@gmail.com`, `aquanaut2`, `turtles`, `turtles`. Hey, a pretty flash message!

> The user has been created successfully

Tip

If you don't see this, make sure you're rendering `success` flash messages in your base template: http://knpuniversity.com/screencast/symfony-forms/save-redirect-set-flash#rendering-the-flash-message

Well, that message lacks some pizzaz! How can we give it some personality? I mean, it's probably being set deep in some PHP file somewhere. Do we need to override that file?

Nope! Every string you see from this bundle is being passed through the translator. So to change text, you just need to translate it!

Back in PHPStorm, I'll close a few tabs, then press Shift+Shift and look for `FOSUserBundle.en.yml`. Whenever FOSUserBundle translates something, it translates it through a *domain* called `FOSUserBundle`... which just means that when you translate its strings, they'll live in a file called `FOSUserBundle.{language}.yml`.

Search for "The user has been created successfully". There it is! Under `registration`, `flash`, `user_created`.

Overriding the Translation

Let's breathe some life into this translation! To do that, inside `app/Resources/translations`, create a new file: `FOSUserBundle.en.yml`. Then, add the same keys: `registration`, `flash`, and `user_created` with the message:

> Welcome! Now let's do some science!

4 lines | app/Resources/translations/FOSUserBundle.en.yml

```yaml
registration:
    flash:
        user_created: "Welcome! Now let's do some science!"
```

Since this is a brand new translation file, you'll need to clear you cache for Symfony to see it:

```
$ php bin/console cache:clear
```

I know, you should almost *never* need to clear cache in the dev environment: this is one of those really rare things. When you *update* this translation file, you will *not* need to clear the cache.

Cool! Go back to `/register` and repeat: `aquanaut3`, `aquanaut3@gmail.com`, turtles, turtles... and yes! Let's do some science! Great idea!

Except, instead of science, let's customize some forms!

Chapter 6: Customizing the Forms

What if we wanted to add our own fields to the registration form? Like, what if we needed to add a "First Name" field? No problem!

Adding a firstName field to User

Start in our `User` class. This is a normal entity... so we can add whatever fields we want, like `private $firstName`. I'll go to Code->Generate, or Command+N on a Mac, then select "ORM Annoations" to annotate `firstName`. Then I'll go back to Code->Generate to add the getter and setter methods.

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

```php
<?php
... lines 2 - 11
class User extends BaseUser
{
... lines 14 - 20
    /**
     * @ORM\Column(type="string")
     */
    private $firstName;
... lines 25 - 30
    public function getFirstName()
    {
        return $this->firstName;
    }

    public function setFirstName($firstName)
    {
        $this->firstName = $firstName;
    }
}
```

Notice, right now, `firstName` is *not* nullable... meaning it's *required* in the database... and that's fine! If `firstName` is optional in your app, you can of course add `nullable=true`. I'm mentioning this for one reason: if you add *any* required fields to your `User`, the `fos:user:create` command will *no* longer work... because it will create a new `User` but leave those fields blank. I never use that command in production anyways, but, you've been warned!

Move over to your terminal to generate the migration:

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

That looks right! Run it:

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

New field added! Now, how can we add it to the form? Simple! We can create our *own* new form class and tell FOSUserBundle to use it instead.

Creating the Override Form

In `src/AppBundle`, create a new `Form` directory and a new class called `RegistrationFormType`. Extend the normal `AbstractType`. Then, I'll use Code->Generate Menu or Command+N to override the `buildForm()` method. Inside, just say `$builder->add('firstName')`.

```
22 lines | src/AppBundle/Form/RegistrationFormType.php
<?php
... line 2
namespace AppBundle\Form;
... lines 4 - 8
class RegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('firstName');
    }
... lines 16 - 20
}
```

In a minute, we'll tell FOSUserBundle to use *this* form instead of its normal registration form. But... instead of completely *replacing* the default form, what I *really* want to do is just *add* one field to it. Is there a way to extend the existing form?

Extending the Core Form

Totally! And once again, the web debug toolbar can help us out. Mouse over the form icon and click that. This tells us what form is used on this page: it's called `RegistrationFormType` - the same as our form class!

To build on *top* of that form, you don't actually extend it. Instead, override a method called `getParent()`. Inside, we'll return the class that we want to extend. At the top, add `use`, autocomplete `RegistrationFormType` from the bundle and put `as BaseRegistrationFormType` to avoid conflicting.

Now in `getParent()`, we can say `return BaseRegistrationFormType::class`.

```
22 lines | src/AppBundle/Form/RegistrationFormType.php
<?php
... lines 2 - 6
use FOS\UserBundle\Form\Type\RegistrationFormType as BaseRegistrationFormType;
... line 8
class RegistrationFormType extends AbstractType
{
... lines 11 - 16
    public function getParent()
    {
        return BaseRegistrationFormType::class;
    }
}
```

And that is it! This form will have the existing fields *plus* `firstName`.

Registering the Form with FOSUserBundle

To tell FOSUserBundle about *our* form, we need to do two things. First, register this as a service. In my `app/config/services.yml`, add `app.form.registration` with `class` set to the `RegistrationFormType`. It also needs to be tagged with `name: form.type`.

```
22 lines | app/config/services.yml
... lines 1 - 5
services:
... lines 7 - 17
    app.form.registration:
        class: AppBundle\Form\RegistrationFormType
        tags:
            - { name: form.type }
```

Finally, copy the class name and go into `app/config/config.yml`. This bundle has a *lot* of configuration. And at the bottom of the documentation page, you can find a reference called `FOSUserBundle Configuration Reference`. I'll open it in a new tab.

This is *pretty* awesome: a *full* dump of all of the configuration options. Some of these are explained in more detail in other places in the docs, but I love seeing everything right in front of me. And we can see what we're looking for under `registration.form.type`.

Go back to your editor and add those: `registration`, `form` and `type`. Paste our class name!

85 lines | app/config/config.yml

```
... lines 1 - 74
fos_user:
... lines 76 - 81
    registration:
        form:
            type: AppBundle\Form\RegistrationFormType
```

And... we're done! Go back to registration and refresh. We got it! And that *will* save when we submit.

Customizing the Form Order

Why is `firstName` at the bottom? Well, remember inside of `register_content.html.twig`, we're using `form_widget(form)`... which just dumps out the fields in whatever order they were added. Need more control? Cool: remove that and instead use `form_row(form.email)`, `form_row(form.username)`, `form_row(form.firstName)` and `form_row(form.plainPassword)`.

14 lines | app/Resources/FOSUserBundle/views/Registration/register_content.html.twig

```
... lines 1 - 2
<h1>Register Aquanaut!</h1>
... line 4
{{ form_start(form, {'method': 'post', 'action': path('fos_user_registration_register'), 'attr': {'class': 'fos_user_registration_regist
    {{ form_row(form.email) }}
    {{ form_row(form.username) }}
    {{ form_row(form.firstName) }}
    {{ form_row(form.plainPassword) }}
... lines 10 - 12
{{ form_end(form) }}
```

If you're not sure what the field names are called, again, use your web debug toolbar for the form: it shows you everything.

Refresh that page! Yes! First Name is *exactly* where we want it to be. Next, what about the username? Could we remove that if our app only needs an email?

Chapter 7: My Users don't have a Username!

Okay: a challenge! In some apps... you don't even *need* a username - you register and login entirely with your email.

But... with FOSUserBundle, we *always* have a username field. So, are we stuck?

Nope! It *is* true that you can't remove the `username` field from your `user` table. But, we *can* remove it from everywhere else.

Auto-Setting the Username Field

Start in the `User` class. I'll go to the Code->Generate menu - or Command+N on a Mac - go to "Override Methods" and choose `setEmail()`. Before the parent call, add `$this->setUsername($email)`.

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

```
<?php
... lines 2 - 11
class User extends BaseUser
{
... lines 14 - 40
    /**
     * Overridden so that username is now optional
     *
     * @param string $email
     * @return User
     */
    public function setEmail($email)
    {
        $this->setUsername($email);

        return parent::setEmail($email);
    }
}
```

This is big! Thanks to this, we no longer need to worry about *ever* setting the `username` field. In the database, `username` will always equal the `email`... which is definitely redundant and unnecessary, but in practice, it works fine.

Removing the Username Field from the Form

The only other thing we need to do is remove the `username` field from the registration form. How? Easy: in `RegistrationFormType`, add `->remove('username')`.

23 lines | src/AppBundle/Form/RegistrationFormType.php

```
<?php
... lines 2 - 8
class RegistrationFormType extends AbstractType
... line 10
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('firstName')
            ->remove('username');
    }
... lines 17 - 21
}
```

Then, in the template, remove its `form_row()`.

13 lines | app/Resources/FOSUserBundle/views/Registration/register_content.html.twig

```
... lines 1 - 2
<h1>Register Aquanaut!</h1>
... line 4
{{ form_start(form, {'method': 'post', 'action': path('fos_user_registration_register'), 'attr': {'class': 'fos_user_registration_regist
    {{ form_row(form.email) }}
    {{ form_row(form.firstName) }}
    {{ form_row(form.plainPassword) }}
... lines 9 - 11
{{ form_end(form) }}
```

And yea, that's it! Refresh! No more username! Register as `aquanaut4@gmail.com` and submit. Check it out in the database:

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

There it is! The `username` is now equal to the email.

Oh, and if you're using the profile edit page from the bundle, you'll also want to remove the `username` field from the `ProfileFormType` form. It's done in exactly the same way.

Chapter 8: Customize everything with Events

We can override templates. We can override translations. We can override forms. But there's *more* than that. For example, after we finish registration, we're redirected to this registration confirmation page. You know what? I'd rather do something else: I'd rather redirect to the homepage and skip this page entirely. How can we do that?

Well, let's do a little bit of digging. If you hover over the route name in the web debug toolbar, you can see that this is page rendered by `RegistrationController`. Back in my editor I'll press Shift+Shift and look for `RegistrationController` in the bundle. Specifically, `registerAction()` is responsible for both rendering the registration page *and* processing the form submit.

And check this out: after the form is valid, it redirects to the confirmation page.

Events to the Rescue!

So at first, it seems like we need to override the controller itself. But not so fast! The controller - in fact *every* controller in FOSUserBundle is *littered* with events: `REGISTRATION_INITIALIZE`, `REGISTRATION_SUCCESS`, `REGISTRATION_COMPLETED` and `REGISTRATION_FAILURE`. Each of these represents a *hook* point where we can add custom logic.

In this case, if you look closely, you can see that after it dispatches an event called `REGISTRATION_SUCCESS`, below, it checks to see if the `$event` has a response set on it. If it does not, it redirects to the confirmation page. But if it *does*, it uses that response.

That's the key! If we can add a listener to `REGISTRATION_SUCCESS`, we can create our own `RedirectResponse` and set that on the event so that this controller uses it. Let's go!

Creating the Event Subscriber

Inside of AppBundle, create a new directory called `EventListener`. And in there, a new PHP class: how about `RedirectAfterRegistrationSubscriber`. Make this implement `EventSubscriberInterface`: the interface that all event subscribers must have. I'll use our favorite Code->Generate menu, or Command+N on a Mac, go to "Implement Methods" and select `getSubscribedEvents`.

12 lines | src/AppBundle/EventListener/RedirectAfterRegistrationSubscriber.php

```
... lines 1 - 2
namespace AppBundle\EventListener;
... lines 4 - 6
class RedirectAfterRegistrationSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
    }
}
```

We want to attach a listener to `FOSUserEvents::REGISTRATION_SUCCESS`, which, by the way, is just a constant that equals some string event name.

In `getSubscribedEvents()`, add `FOSUserEvents::REGISTRATION_SUCCESS` assigned to `onRegistrationSuccess`. This means that when the `REGISTRATION_SUCCESS` event is fired, the `onRegistrationSuccess` method should be called. Create that above: `public function onRegistrationSuccess()`.

22 lines | src/AppBundle/EventListener/RedirectAfterRegistrationSubscriber.php

```
<?php
... lines 2 - 8
class RedirectAfterRegistrationSubscriber implements EventSubscriberInterface
{
    public function onRegistrationSuccess(FormEvent $event)
    {

    }

    public static function getSubscribedEvents()
    {
        return [
            FOSUserEvents::REGISTRATION_SUCCESS => 'onRegistrationSuccess'
        ];
    }
}
```

Oh, and notice that when this event is dispatched, the bundle passes a `FormEvent` object. That will be the first argument to our listener method: `FormEvent $event`. That's what we need to set the response onto.

Investigating all the Events

Before we go any further, I'll hold command and click into the `FOSUserEvents` class... cause it's awesome! It holds a list of *every* event dispatched by FOSUserBundle, what its purpose is, and what event object you will receive. This is gold.

Creating the RedirectResponse

Back in `onRegistrationSuccess`, we need to create a `RedirectResponse` and set it on the event. But to redirect to the homepage, we'll need the router. At the top of the class, create `public function __construct()` with a `RouterInterface $router` argument. Next, I'll hit Option+Enter, select "Initialize Fields" and choose `router`.

33 lines | src/AppBundle/EventListener/RedirectAfterRegistrationSubscriber.php

```
<?php
... lines 2 - 10
class RedirectAfterRegistrationSubscriber implements EventSubscriberInterface
{
    private $router;

    public function __construct(RouterInterface $router)
    {
        $this->router = $router;
    }
... lines 19 - 32
}
```

That was just a shortcut to create the `private $router` property and set it in the constructor: nothing fancy.

Now, in `onRegistrationSuccess()` we can say `$url = $this->router->generate('homepage')`, and `$response = new RedirectResponse($url)`. You may or may not be familiar with `RedirectResponse`. In a controller, to redirect, you use `$this->redirectToRoute()`. In reality, that's just a shortcut for these two lines!

Finally, add `$event->setResponse($response)`.

```
33 lines | src/AppBundle/EventListener/RedirectAfterRegistrationSubscriber.php
<?php
... lines 2 - 10
class RedirectAfterRegistrationSubscriber implements EventSubscriberInterface
{
... lines 13 - 19
    public function onRegistrationSuccess(FormEvent $event)
    {
        $url = $this->router->generate('homepage');
        $response = new RedirectResponse($url);
        $event->setResponse($response);
    }
... lines 26 - 32
}
```

Ok, this class is *perfect*! To tell Symfony about the event subscriber, head to `app/config/services.yml`. At the bottom, add `app.redirect_after_registration_subscriber`, set the class, and add `autowire: true`. By doing that, thanks to the `RouterInterface` type-hint, Symfony will automatically know to pass us the router.

Finally, add a tag on the bottom: `name: kernel.event_subscriber`. And we are done!

```
28 lines | app/config/services.yml
... lines 1 - 5
services:
... lines 7 - 22
    app.redirect_after_registration_subscriber:
        class: AppBundle\EventListener\RedirectAfterRegistrationSubscriber
        autowire: true
        tags:
            - { name: kernel.event_subscriber }
```

Try it out! Go back to `/register` and signup as `aquanaut5@gmail.com`. Fill out the rest of the fields and submit!

Boom! Back to our homepage! You can customize just about *anything* with events. So don't override the controller. Instead, hook into an event!

Chapter 9: TargetPathTrait: Redirect to Previous Page

We now have control over where the user goes after registering. But... it's not as awesome as it could be. Let me show you why.

First, look at my `app/config/security.yml` file. In order to access any URL that start with `/admin`, you need to be logged in. For example, if I go to `/admin/genus`, it sends me to the login page.

Thanks to Symfony's `form_login` system, if we logged in, it would automatically redirect us *back* to `/admin/genus`... which is awesome! That's clearly where the user wants to go.

But what if I instead clicked a link to register and submitted that form? Shouldn't that *also* redirect me back to `/admin/genus` after? Yea, that would be *way* better: I want to keep the user's experience really smooth.

How can we do this?

Using TargetPathTrait

Guess what? It's almost *effortless*, thanks to a trait that was added in Symfony 3.1: `TargetPathTrait`. In your subscriber, `use TargetPathTrait`. Then, down in `onRegistrationSuccess`, add `$url = $this->getTargetPath()` - a method provided by that trait. Pass this `$event->getRequest()->getSession()` and for the "provider key" argument, pass `main`. Provider key is a fancy term for your firewall's name.

```
42 lines | src/AppBundle/EventListener/RedirectAfterRegistrationSubscriber.php
<?php
... lines 2 - 11
class RedirectAfterRegistrationSubscriber implements EventSubscriberInterface
{
    use TargetPathTrait;
... lines 15 - 22
    public function onRegistrationSuccess(FormEvent $event)
    {
        // main is your firewall's name
        $url = $this->getTargetPath($event->getRequest()->getSession(), 'main');
... lines 27 - 33
    }
... lines 35 - 41
}
```

What's going on here? Well, whenever you try to access a secured page anonymously, Symfony stores that URL somewhere in the session before redirecting you to the login page. Then `form_login` uses that to redirect you after you authenticate. The `TargetPathTrait` is just a shortcut for *us* to read that *same* key from the session.

If `$url` is empty - it means the user went directly to the registration page. No worries! Just send them to the homepage.

42 lines | src/AppBundle/EventListener/RedirectAfterRegistrationSubscriber.php

```
<?php
... lines 2 - 11
class RedirectAfterRegistrationSubscriber implements EventSubscriberInterface
{
... lines 14 - 22
    public function onRegistrationSuccess(FormEvent $event)
    {
... lines 25 - 27
        if (!$url) {
            $url = $this->router->generate('homepage');
        }
... lines 31 - 33
    }
... lines 35 - 41
}
```

Let's try the *entire* flow. I'll go back to `/admin/genus`: it redirects me to the login page and sets that session key behind the scenes. Then, I'll manually type in `/register` - but pretend like we clicked a link. Register as `aquanaut6`, password `turtles`.

Booya! Logged in *and* on the `/admin/genus` page. That's a kick butt registration form.

Chapter 10: FOSUserBundle <3's Guard Authenticators

We *now* understand that FOSUserBundle *just* gives us a nice `User` class and some routes & controllers for registration, reset password, edit profile and a few other things. The bundle does *not* provide any authentication. Open `app/config/security.yml`. The `form_login` authentication mechanism we're using is core to Symfony itself, not this bundle.

So, one of the questions we get a lot is: how can I use Guard authentication with FOSUserBundle? It turns out, it's simple! Guard authentication and FOSUserBundle solve different problems, and they work together beautifully. Teamwork makes the dream work!

But, why would you want to use Guard authentication with FOSUserBundle? Well, as easy as `form_login` is, it's a pain to customize. Guard is more work up front, but gives you a lot more control. You can also use Guard to add some sort of API authentication on top of `form_login`.

Creating the Authenticator

Let's replace `form_login` with a more flexible Guard authenticator. At the root of our project, you should have `tutorial/` directory with a file called `LoginFormAuthenticator.php`. In `src/AppBundle`, create a new directory called `Security` and paste that file here.

103 lines | src/AppBundle/Security/LoginFormAuthenticator.php

```php
<?php
... line 2
namespace AppBundle\Security;
... lines 4 - 19
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
... lines 22 - 101
}
```

This `LoginFormAuthenticator` is almost an exact copy of the authenticator we created in our Symfony Security tutorial. I've just added CSRF token checking - since our HTML login form has a CSRF token in it - and made a few other minor tweaks. For example at the bottom, I updated the login route name to use the one from FOSUserBundle.

The authenticator is very straightforward: It looks for the submitted `_username` and `_password` fields from the login form. It doesn't care if you built that login form yourself, or if it comes from FOSUserBundle. Then, it queries for your `User` object by email only and checks to see if the password is valid. Obviously you can write your authenticator to do anything.

Registering the Authenticator

To get this to work, like all authenticators, we need to register it as a service. I'll add `app.security.login_form_authenticator`, set the class to `LoginFormAuthenticator` and use `autowire: true`.

32 lines | app/config/services.yml

```yaml
... lines 1 - 5
services:
... lines 7 - 28
    app.security.login_form_authenticator:
        class: AppBundle\Security\LoginFormAuthenticator
        autowire: true
```

Copy that service ID. Then open `app/config/security.yml`. Ok, let's comment-out `form_login` entirely. And instead, add `guard`, `authenticators`, then paste the service ID.

```
38 lines | app/config/security.yml
... lines 1 - 2
security:
... lines 4 - 12
    firewalls:
... lines 14 - 18
        main:
... lines 20 - 27
            guard:
                authenticators:
                    - app.security.login_form_authenticator

#           form_login:
#               csrf_token_generator: security.csrf.token_manager
... lines 34 - 38
```

That's it! FOSUserBundle doesn't care who or what is processing the login form submit.

Let's try it! Click log out, click login and login with admin@aquanote.com. Yea, this *does* still say "Username", but we know that our authenticator actually logs us in via email. So, we'll want to tweak that language. Use the password `admin` and... boom!

Congrats! You just used a Guard authenticator with FOSUserBundle. Wasn't that nice? You should feel empowered to use FOSUserBundle because you want things like a registration page or reset password system. But, you can *still* take control of your actual login mechanism and do whatever the heck you want.

The last part of this bundle that you'll need to customize are the emails: the reset password email and the registration confirmation email, if you want to send that one. The docs are good on this topic, and it's mostly a matter of overriding templates... which we already mastered.

All right guys, go use FOSUserBundle to quickly bootstrap your site! As long as you understand what it does... and does *not* give you, it's awesome. Seeya next time!