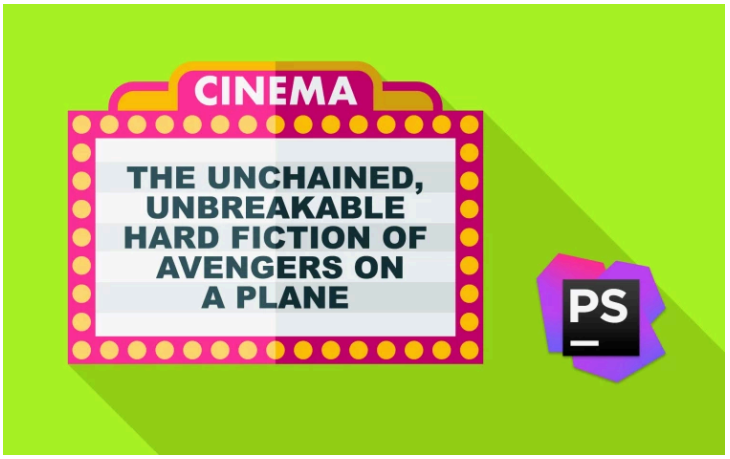
Lean and Mean Dev with PhpStorm (for Symfony)



Chapter 1: Setup

Hey guys! If you've watched even one video on KnpU - please tell me that you have - then you've seen me using PHPStorm.

Now think about how much you use your editor. If you work 40 hours a week I bet you stare deeply into your editor at least 20 of those hours. So over a year, that's... let's see here - carry the one - over 1,000 hours!

So if you could learn something that saves you a few seconds now, that's going to mean you're saving 10's of hours in the future.

Particularly in Symfony development, mastering PHPStorm is going to completely speed up the way you develop and help you catch mistakes you may have missed otherwise.

If you've never used PHPStorm before it does have a free trial period that you can use to see if it's going to work for you.

Go Deeper!

For even *deeper* information about PhpStorm settings and shortcuts, you can also check out Beginner's Guide to PhpStorm - Tips & Tricks.

UI Updates

I'm working with a fresh version of PHPStorm, this is not how I want my development environment to look - all this white is stressing me out. So download your own copy and let's customize this thing together.

If I open any of my files right now you'll see that they look pretty ugly. Let's clean that up first: open up preferences and in Appearance under Appearance & Behavior you can select your theme, change that to Darcula.

(insert Dracular slow laugh here)

This is the theme of the UI which also goes and sets the scheme for the editor. And this is already so much better. Staring into a white screen for 1,000+ hours a year is very similar to just staring at a lightbulb -- not so great on the eyes.

Speaking of eyes, I'd like to see my font size a bit bigger here. Back to preferences, on a mac its shortcut is `⌘`, so I'll start using that. On Windows and Linux, it's the longer `Ctrl`+`Alt`+`S` - you can check out the key mappings under the keymap section, to see and simplify these.

Use this search to look for font. This pulls up a few things, general appearance, colors & fonts. Let's pick font here and there's font size.

I can't immediately change the font size I've got because that would change the Darcula scheme. So first let's click 'save as' so we can customize this, save it as 'darcula mine'. And now I can change the font size to 14 and give it a line spacing of say 1.2. Ahh so much better.

Project Organization

Over here on the left, you can see the `tuts` and `knpu` directories. I use these to help me build this tutorial - but when I'm coding, I don't want to see them. Any time you have a directory that you don't want PHPStorm to look into, or that you just don't want to see, you can hide it. Right click `tuts`, find 'mark directory as' and select 'excluded'. In a second that will hide this for me. Let's also do this on the `knpu` directory.

And now we can change the view in the upper left from 'project' to 'project files'. Let's also exclude this `.idea` directory. If you ever need to see those in the tree again you can just switch back to 'project' and they'll reappear. Then you can even unmark them if you want to.

Plugins

Okay back to the 'project files' view. So PHPStorm is the best editor I've ever seen. But the real power with Symfony comes from an *amazing* plugin. Back in preferences if you search for Symfony it brings you to a plugin spot. Click 'Browse Repositories', here you see the community plugins for Symfony and there's one called the 'Symfony2 Plugin'.

Tip

The "Symfony2 Plugin" was renamed to "Symfony Support" - install this instead

To prove how great it is just look at its 1.2 million downloads. So press the 'Install Plugin' button to get this added to our IDE.

The other plugin that you are definitely going to want to install is this called 'PHP Annotations'. Find that one in the search and install it. Perfect.

Close out of this window, and then press ok and restart PHPStorm.

You only need to install the Symfony plugin once, but each time you start a new project you will need to enable it. Back in preferences, search Symfony and check the 'Enable Plugin for this Project' checkbox. Do that just one time per project - not bad!

The checkbox had a note that it requires a restart, that's mostly not true. But I am going to restart one more time because there are a few small features that will not work until you do this.

Now we're really ready to go! I've already got my my site running on localhost:8000 and we've got a nice looking 404 page since there isn't anything inside of here yet.

Tip

To run the website, execute in your terminal:

```
php app/console server:run
```

And open the homepage at http://localhost:8000/

Just to prove that the Symfony Plugin is installed and running, in the bottom right of PHPStorm you should see a little Symfony menu. Clicking it gives you a quick way to jump into the profiler for any recent requests.

Now that our editor is all setup let's get down to work!

Chapter 2: Annotations

We've got a big project today, a site about movies. But only movies that have Samuel L. Jackson.

> **Tip**
>
> The "Controller" option when you create a new file is no longer available. However, you can use `php bin/console make:controller` from MakerBundle as an even nicer option.

We'll start by creating a `MovieController`. We could right click on the `Controller` directory, go to new, and select file from the menu. But we can be faster! For Mac users, use `⌘+N` to get the "new" menu open. And because we have the Symfony Plugin, by typing 'controller' you'll find an option to create one directly. Let's keep things simple and obvious and call it `MovieController`.

src/AppBundle/Controller/MovieController.php

```php
<?php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class MovieController extends Controller
{
    public function indexAction($name)
    {
        return $this->render('', array('name' => $name));
    }
}
```

Ah, and when you use Git, PHPStorm asks you whether you want to add new files to git. I prefer to just do this myself so I'll click 'no'. Beautiful!

Auto-complete Annotations

We'll need a route to this `MovieController`, so add some annotations above `indexAction()`. Let's change these to `@Route` and let autocomplete do its thing. The one we want is `Sensio\Bundle\FrameworkExtraBundle\Configuration\Route`, select that option and hit tab and notice that it added a `use` statement on line 5:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 4
5  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
↕ // ... lines 6 - 7
8  class MovieController extends Controller
9  {
10     /**
11      * @Route("/movies/new")
12      */
13     public function newAction()
↕ // ... lines 14 - 16
17 }
```

That's really important because when you use annotations you need to have a `use` statement for them. Now instead of spending your time looking it up in the documentation, autocomplete will take care of it for you. Robots FTW!

Let's have our path be `/movies/new` and change `indexAction()` to `newAction()` because this controller will render a "new movie" form. And we'll fix up our render call in a second:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 9
10     /**
11      * @Route("/movies/new")
12      */
13     public function newAction()
14     {
15         return $this->render('', array());
16     }
↕ // ... lines 17 - 18
```

Head over to the browser and check if our new route is hitting our endpoint. Perfect! It can't find the template but that's fine because, there isn't one :)

Annotation Options

Annotations have a lot of options and one of them here is called `name`, which auto-completes for us. Hey thanks. Fill that in with `movies_new`:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 9
10     /**
11      * @Route("/movies/new", name="movies_new")
12      */
13     public function newAction()
↕ // ... lines 14 - 18
```

If you're the curious sort and are wondering what other options you have, just hold the control key and hit the space bar: that will bring up all the options that you have in that spot. In fact, "control+space" can be used pretty much anywhere to show you your auto-complete option - really handy.

This autocomplete works for two reasons. First, an annotation represents a real class. Hold command, or control if you're in windows, and click `@Route` to open up the class that fuels it.

vendor/sensio/framework-extra-bundle/Configuration/Route.php

```
↕ // ... lines 1 - 11
12 namespace Sensio\Bundle\FrameworkExtraBundle\Configuration;
↕ // ... lines 13 - 19
20 class Route extends BaseRoute
21 {
↕ // ... lines 22 - 43
44 }
```

If you hold command again and go into its base class, you'll see that all of those options are represented as properties inside of that class:

vendor/symfony/symfony/src/Symfony/Component/Routing/Annotation/Route.php

```
↕ // ... lines 1 - 11
12 namespace Symfony\Component\Routing\Annotation;
↕ // ... lines 13 - 21
22 class Route
23 {
24     private $path;
25     private $name;
26     private $requirements = array();
27     private $options = array();
28     private $defaults = array();
29     private $host;
30     private $methods = array();
31     private $schemes = array();
32     private $condition;
↕ // ... lines 33 - 181
182 }
```

To take this further if you go back to `MovieController` you can hold command and click on the `name` option and have it take you right to that property. You may not need this very often, but sometimes it's useful to see how an annotation option is used to figure out what value you want to set for it.

Now thanks to the annotations plugin, annotations act a lot more like normal code, with fancy auto-completion and other goodies.

Chapter 3: Twig

The Symfony Plugin has all kinds of awesome setup for Twig integration. So let's render a template here, `movie/new.html.twig`:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 9
10      /**
11       * @Route("/movies/new", name="movies_new")
12       */
13      public function newAction()
14      {
15          return $this->render('movie/new.html.twig', array());
16      }
↕ // ... lines 17 - 18
```

PHPStorm highlights this immediately because it's missing its template, how thoughtful!

Let's make that template, in `app/Resources/views` we'll create a directory called `movie` and a file called `new.html.twig`. Like always, start by extending the base template, but notice we get autocomplete on that `extends` tag which is *fantastic*! And we even get autocomplete on the template name. Down here, we get more autocomplete on `block` and `endblock` -- it's like Christmas morning!

app/Resources/views/movie/new.html.twig

```
1 {% extends '::base.html.twig' %}
2
3 {% block body %}
4     <h1>New Samuel L Jackson Movie</h1>
5 {% endblock %}
```

For our `h1` tag we'll say 'New Samuel L Jackson Movie'. And because the plugin gives us intelligence on the Twig file names, you can hover over any template name, hold command, and click it to go there. You can even do the same thing on a block: holding command and clicking `block body` takes us to the `block body` in `base.html.twig` because it knows where that's coming from.

I'll close these two templates and go back to our controller. When we refresh we've got our functional page. Now, to the left of `newAction`, we have a cute little icon that appeared. When you click it, it shows you related files to this controller. So we see here we've got the controller itself and the template, and this list will get longer as we start referencing more things like repositories and services.

Like most templates, we're going to want to use some variables. Let's start with a quote variable that we'll display in the form for inspiration. Eventually we'll have this generated randomly but for now we'll just hardcode it in:

src/AppBundle/Controller/MovieController.php

```
// ... lines 1 - 12
    public function newAction()
    {
        return $this->render('movie/new.html.twig', array(
            'quote' => 'If my answers frighten you then you should cease asking scary questions. (Pulp Fiction)'
        ));
    }
// ... lines 19 - 20
```

"If my answers frighten you then you should cease asking scary questions". Oh Sam you're too much!

I could just go to my tree and double click `new.html.twig` to open it again, but instead I'm going to hold command and click the name in the controller to get us there. In the template we'll use the normal `` `{{ }}` `` inside a p tag, but watch this: we get autocomplete on our quote variable. That is huge!

app/Resources/views/movie/new.html.twig

```
// ... lines 1 - 2
{% block body %}
// ... lines 4 - 5
    <h3>
        {{ quote }}
    </h3>
{% endblock %}
```

In my experience, it doesn't work 100% of the time, but it does work most of the time. It will even look inside your controller to see what kind of object it is and give you autocomplete on its methods.

And remember, anywhere that you get autocomplete, you can hit command+space to see all of your options. Here it shows us the variable and all of the function names and filters that come from everywhere inside of Symfony and Twig. For example, if you start using the `upper` filter, you'll get that autocompleted. Same goes for the `date` filter, and as you're typing `date` you'll see the arguments you'll need. The first is the value to the left and the second here - `format` - is what we'll use on the filter itself: it's a nice reminder we need to pass something like `Y-m-d`. To make this a bit clearer I'll add 'Today is:' before the date is actually rendered and we'll put our quote in an h3 tag.

app/Resources/views/movie/new.html.twig

```
↕ // ... lines 1 - 2
3 {% block body %}
↕ // ... line 4
5     <p>Today: {{ 'now'|date('Y-m-d') }}</p>
↕ // ... lines 6 - 9
10 {% endblock %}
```

If you really can't remember how the `date` filter works, you can hold command now and click into it. That's going to take you to whatever class provides that filter. In this case there's two. If you ever see something like this: where one is in the `cache` directory, just ignore that and click the other option.

The Twig environment is passed first and after that you have the date, the format and the timezone and you can just see how things work. This is wonderful for debugging.

Eventually we're going to have a form so I'll go ahead and create a new template called `_form.html.twig` and put my fake form tag stuff right there. Heck, go crazy and throw a button in too:

app/Resources/views/movie/_form.html.twig

```
1 <form action="" method="POST">
2     <button type="submit" class="btn btn-primary">Save</button>
3 </form>
```

You're seeing another trick with PHPStorm, the mini-code generation comes from a feature called live templates. I just type `form`, hit tab, and it builds the tag for me. The same works with `button` then tab, `textarea` then tab and a lot of other things. We'll create our own live templates in a bit.

Back in `new.html.twig`, you know how this works. We use the `include` function and not only do you get autocomplete on that function name, but you also get it on the template name.

Don't worry about the "movie" part of the name, just start typing `_form.html.twig` and you're all set. And there are two formats here: both valid, but I like the first one, it's cleaner.

app/Resources/views/movie/new.html.twig

```
↕ // ... lines 1 - 2
3 {% block body %}
↕ // ... lines 4 - 6
7     {{ include('movie/_form.html.twig') }}
↕ // ... lines 8 - 11
12 {% endblock %}
```

Go back and refresh, beautiful! Everything looks awesome, except I guess my button is ugly. So I'll throw in some twitter bootstrap classes to make this a bit prettier.

Ahh there we go! And that's some of what you can expect now with Twig!

Chapter 4: Namespaces

Since we have this new movie form, we're going to want to save the data to the database. To do this let's create a `Movie` entity. I'm not going to generate it, let's just do this by creating a good ole fashioned PHP Class called `Movie`. But when I do this, notice the namespace box is empty. I could type `AppBundle\Entity` here, but I'm trying to do as little work as possible people!

Let's make the robots do this for you. Go into preferences, search for 'directories' and here you can see the three directories that we excluded earlier. Click on `src` and click the blue sources button above. Now we can see this on the right and clicking the little 'p' allows us to edit the properties. So, if we were using PSR-4 inside of this directory we could actually put our namespace prefix in this box - like `KnpU`. But all we needed to do was mark it as a source route, so we're done!

When we head back, oh look the `Entity` directory disappeared! Where did we misplace that to? Well, sometimes PHPStorm does that to empty directories. Temporarily I'll switch back to project view and now create a new PHP Class. Because of the sources change, it guesses the namespace part perfect. Call the class `Movie`:

src/AppBundle/Entity/Movie.php

```
1  <?php
2
3  namespace AppBundle\Entity;
4
5  class Movie
6  {
7
8  }
```

Okay switch back to Project Files and there it is. Excellent! In a second we'll set this up with some annotations, but right now I want to use it inside of my `MovieController`. We'll add a form here soon, and when we do, we'll need a new `Movie` object. So we'll say `$movie = new`... and I could end this with `AppBundle\Entity\Movie` -- but you really don't want to do that. Always go directly to the last part: in our case `Movie`, and let that autocomplete:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 4
5 use AppBundle\Entity\Movie;
↕ // ... lines 6 - 8
9 class MovieController extends Controller
10 {
↕ // ... lines 11 - 13
14     public function newAction()
15     {
16         $movie = new Movie();
↕ // ... lines 17 - 20
21     }
22 }
```

Because, when you do that, it adds the `use` statement on line 5 which is HUGE. If you are not using the autocomplete functionality for `use` statements, then you're doing it wrong.

Now, if something went wrong and you didn't autocomplete or you end up with `Movie` here but no `use` statement, you can import it if you want. Just hit `alt + enter`. `Alt enter` is one of the most important shortcut we are going to see. It's called the "actions" shortcut. Often when you're inside some code. you can `alt + enter` and get a list of actions to do. I'll select 'import class' from this list and it will add that `use` statement for us.

To take this a bit further let's add a `public function listAction()` that will list those movies from the database. I don't want to build this out now, I just want it to return a simple `Response` that says "todo". So let's add `return new Response` and select the Response we want, which is the one from HttpFoundation. Hit tab and the `use` statement politely puts itself on top of our file. Finish with 'TODO'. Now let's add a route above this, `@Route("/movies", name="movies_list")`:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\HttpFoundation\Response;
↕ // ... line 9
10 class MovieController extends Controller
11 {
↕ // ... lines 12 - 23
24     /**
25      * @Route("/movies", name="movies_list")
26      */
27     public function listAction()
28     {
29         return new Response('TODO!');
30     }
31 }
```

Oh and now that we have this second endpoint, back in `new.html.twig`, if we want to create a link to this page, we can. And we can use another live template. Type `a`, hit tab, and use `{{ path() }}` and - amazingly - we even get auto-complete on the route name. For the link text, say "back to list':

app/Resources/views/movie/new.html.twig

```
↕ // ... lines 1 - 5
6      <a href="{{ path('movies_list') }}">Back to list</a>
↕ // ... lines 7 - 16
```

Refresh that, and there's our link right back to our list. Awesome!

So, don't. write. `use` statements. ever.

Chapter 5: Doctrine

We have a `Movie` class, but its missing all of its annotations! Yikes! I *really* don't feel like typing all of that.

Let me introduce you to another really important shortcut: generate. You can get to it with `command+n`. Or just click the code menu at the top, and select generate. Either way in this class, go down to "Generate ORM Class": and just like *all* menus in PhpStorm, you can just start typing to search for it.

Sweet - this automatically added all the `Entity` annotation stuff on top for us.

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 4
5  /**
6   * @Doctrine\ORM\Mapping\Entity
7   * @Doctrine\ORM\Mapping\Table(name="movie")
8   */
9  class Movie
10 {
11
12 }
```

But it looks different: usually we'd just have the `@ORM\Entity` stuff on top, and it added it this way because we don't have the `use` statement yet. There's a simple solution to that. Just copy the full `Entity` annotation, say `use`, paste that and then delete the last part and instead stay `as ORM;`:

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 4
5  use Doctrine\ORM\Mapping as ORM;
↕ // ... lines 6 - 15
```

That's the standard `use` statement that you see at the top of all Doctrine entities.

Let's get rid of these lines here, then get back inside of the class because context] is important with `command+n`. Hit command n and search for 'ORM' and select "Generate ORM Class":

src/AppBundle/Entity/Movie.php

```
// ... lines 1 - 4
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="movie")
 */
class Movie
{

}
```

Perfect!

Let's add our properties, `private $id;`, `private $title;` for the movie title, `private $samsCharacterName` for Sam's character, `private $rating` to give the movie appearance a rating, `private $isMainCharacter` so we can see if Sam was the lead in that movie, and lastly `private $releasedAt` which will tell us exactly when the movie came out.

src/AppBundle/Entity/Movie.php

```
// ... lines 1 - 10
class Movie
{
    private $id;

    private $title;

    private $samsCharacterName;

    private $rating;

    private $isMainCharacter;

    private $releasedAt;


}
```

Easy!

The warning here is that each of these is an unused private field, which is true so far. But good to know, in the future that could just be extra code.

Generate ORM Annotations

Now that we have these we can go back to command n, search ORM and select "Generate ORM annotations". Select all of the fields that are in the menu and hit ok:

src/AppBundle/Entity/Movie.php

```
// ... lines 1 - 10
class Movie
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string")
     */
    private $title;

    /**
     * @ORM\Column(type="string")
     */
    private $samsCharacterName;

    /**
     * @ORM\Column(type="string")
     */
    private $rating;

    /**
     * @ORM\Column(type="boolean")
     */
    private $isMainCharacter;

    /**
     * @ORM\Column(type="datetime")
     */
    private $releasedAt;
}
```

Awesome! Each field has all its annotations! Even better than that, it recognizes that `id` is our primary key so it set that up and it noticed that `$isMainCharacter` is probably a boolean, because of the 'is' in the beginning. And it also saw that `$releasedAt` is a `datetime`. This

isn't perfect, I would prefer `$releasedAt` to just be a date, and rating up here is not a string, but an integer:

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 10
11 class Movie
12 {
↕ // ... lines 13 - 29
30     /**
31      * @ORM\Column(type="integer")
32      */
33     private $rating;
↕ // ... lines 34 - 39
40     /**
41      * @ORM\Column(type="date", nullable=true)
42      */
43     private $releasedAt;
44 }
```

But I do love getting autocomplete on all of those different types. Pressing `control+space` gives you a list of all the different types you have access to.

Ok, let's give `$isMainCharacter` a default value just in case it's ever not set. We'll make `$releasedAt` optional since a movie might not be released yet: set `nullable=true`: more autocomplete:

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 10
11 class Movie
12 {
↕ // ... lines 13 - 34
35     /**
36      * @ORM\Column(type="boolean")
37      */
38     private $isMainCharacter = false;
39
40     /**
41      * @ORM\Column(type="date", nullable=true)
42      */
43     private $releasedAt;
44 }
```

Up here for `$samsCharacterName`, well this probably won't be too long so we can give it a length of 100 instead of the default 255:

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 10
11 class Movie
12 {
↕ // ... lines 13 - 24
25     /**
26      * @ORM\Column(type="string", length=100)
27      */
28     private $samsCharacterName;
↕ // ... lines 29 - 43
44 }
```

Alright, this is all looking really nice.

Generating Getters and Setters

At this point we just have private properties so we need our getters and setters. Back to generate! Use our favorite shortcut, command+n, select getters and then `$id`:

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 10
11 class Movie
12 {
↕ // ... lines 13 - 47
48     public function getId()
49     {
50         return $this->id;
51     }
52 }
```

Then back to generate and select getters and setters and select everything else. Before I finish this I want to pause and say that you don't necessarily need a getter and setter for every field in Doctrine. Sometimes you might want to wait to add the getters and setters until you actually need them. Then when the need arises you have these awesome shortcuts available.

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 10
11 class Movie
12 {
↕ // ... lines 13 - 49
50     public function getTitle()
51     {
52         return $this->title;
53     }
54
55     public function setTitle($title)
56     {
57         $this->title = $title;
58     }
59
60     public function getSamsCharacterName()
61     {
62         return $this->samsCharacterName;
63     }
64
65     public function setSamsCharacterName($samsCharacterName)
66     {
67         $this->samsCharacterName = $samsCharacterName;
68     }
↕ // ... lines 69 - 98
99 }
```

If I press `command+,` to get into preferences, you'll find that the templates that generate these methods are editable. Search for "templates", and you'll see the area where you can modify the getter and setter templates. I've already done this: usually they generate with some PHPDoc, which to me is kind of meaningless, so I've already removed it for nice clean rendering.

Generating the Repository

One last thing here! This entity needs a repository. Back to the action shortcut, which is... `alt+enter`! This opens up a menu to add the doctrine repository, which as you may have guessed, adds a repository class here:

src/AppBundle/Entity/MovieRepository.php

```
↕ // ... lines 1 - 2
3 namespace AppBundle\Entity;
4
5 use Doctrine\ORM\EntityRepository;
↕ // ... lines 6 - 12
13 class MovieRepository extends EntityRepository
14 {
15 }
```

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 6
7 /**
8  * @ORM\Entity(repositoryClass="MovieRepository")
↕ // ... line 9
10  */
11 class Movie
12 {
↕ // ... lines 13 - 98
99 }
```

In my case I prefer to have these in a `Repository` directory. So that's nice that it helped me create that, but I'll move it manually.

A quick copy and paste will do that, then update the namespace to end with `Repository`:

src/AppBundle/Repository/MovieRepository.php

```
↕ // ... lines 1 - 2
3 namespace AppBundle\Repository;
4
5 use Doctrine\ORM\EntityRepository;
↕ // ... lines 6 - 12
13 class MovieRepository extends EntityRepository
14 {
15 }
```

Even though I had to move that manually, what's really cool is that it's highlighting and saying "Yo! Your repository Class is messed up! You can't use the short class name because now it's in a different namespace." I can delete what's there and type `MovieRepository` and I get autocomplete on the entire name:

src/AppBundle/Entity/Movie.php

```
↕ // ... lines 1 - 6
7  /**
8   * @ORM\Entity(repositoryClass="AppBundle\Repository\MovieRepository")
↕ // ... line 9
10  */
11 class Movie
12 {
↕ // ... lines 13 - 98
99 }
```

And there's more autocompleting goodness I won't show here for when you're building relationships and using the Query Builder.

Chapter 6: Forms

> **Tip**
>
> This screencast shows the *old*, 2.7 and earlier form syntax. But, the code blocks below have been updated to show the new syntax!

Now that we've got our entity let's create a form! Click on AppBundle, press our handy shortcut `command+n`, and if you search form you'll find that option in the menu! It's all coming together!

We'll call this `MovieType`:

src/AppBundle/Form/MovieType.php

```php
<?php

namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

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

    }

    public function configureOptions(OptionsResolver $resolver)
    {

    }
}
```

Notice it was actually smart enough to put that inside of a `Form` directory *and* build out the whole structure that we'll need. All I need next is for it to pour me a cup of coffee.

The first thing we always do inside of here is call `$resolver->setDefaults(array())` and pass the `data_class` option so that it binds it to our `Movie` entity. Conveniently, this gives us more autocomplete: we can just type `Movie` and it adds the rest:

src/AppBundle/Form/MovieType.php

```
// ... lines 1 - 8
class MovieType extends AbstractType
{
// ... lines 11 - 15
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Movie'
        ));
    }
}
```

Configuring Form Fields

This will even help us build our fields. If we type `$builder->add('')` here, because this is bound to our `Movie` entity, it knows all the properties we have there. So let's plug in our property of `title` which should be a `text` field, `samsCharacterName` which is probably a text field as well and `isMainCharacter` which will be a checkbox. We'll want to make sure that we prevent html5 validation on that. A third argument of `'required' => false` will take care of that for us and even that has autocomplete. It's madness!

src/AppBundle/Form/MovieType.php

```
↕ // ... lines 1 - 5
6  use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
↕ // ... lines 7 - 8
9  use Symfony\Component\Form\Extension\Core\Type\TextType;
↕ // ... lines 10 - 12
13 class MovieType extends AbstractType
14 {
15     public function buildForm(FormBuilderInterface $builder, array
   $options)
16     {
17         $builder->add('title', TextType::class)
18             ->add('samsCharacterName', TextType::class)
19             ->add('isMainCharacter', CheckboxType::class, array(
20                 'required' => false,
21             ))
↕ // ... lines 22 - 25
26     }
↕ // ... lines 27 - 33
34 }
```

Let's also include `rating` as an integer field and lastly, `releasedAt` which is a date field:

src/AppBundle/Form/MovieType.php

```
↕ // ... lines 1 - 6
7  use Symfony\Component\Form\Extension\Core\Type\DateType;
8  use Symfony\Component\Form\Extension\Core\Type\IntegerType;
9  use Symfony\Component\Form\Extension\Core\Type\TextType;
↕ // ... lines 10 - 12
13 class MovieType extends AbstractType
14 {
15     public function buildForm(FormBuilderInterface $builder, array
   $options)
16     {
17         $builder->add('title', TextType::class)
↕ // ... lines 18 - 21
22             ->add('rating', IntegerType::class)
23             ->add('releasedAt', DateType::class, array(
↕ // ... line 24
25             ));
26     }
↕ // ... lines 27 - 33
34 }
```

Digging into Options and Fields

We can control how this date field renders: by default with Symfony it would be 3 select boxes. I'll set a `widget` option, except I don't remember what value to set that to. No worries, I'll just hold the command key over the `widget` option and it will take me straight to where that is setup inside of the core code:

vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/DateType

```
↕ // ... lines 1 - 11
12 namespace Symfony\Component\Form\Extension\Core\Type;
↕ // ... lines 13 - 26
27 class DateType extends AbstractType
28 {
29     const DEFAULT_FORMAT = \IntlDateFormatter::MEDIUM;
↕ // ... lines 30 - 167
168     public function configureOptions(OptionsResolver $resolver)
169     {
↕ // ... lines 170 - 202
203         $resolver->setDefaults(array(
↕ // ... lines 204 - 206
207             'widget' => 'choice',
↕ // ... lines 208 - 224
225         ));
↕ // ... lines 226 - 235
236         $resolver->setAllowedValues('widget', array(
237             'single_text',
238             'text',
239             'choice',
240         ));
↕ // ... lines 241 - 245
246     }
↕ // ... lines 247 - 321
322 }
```

Why is that awesome you ask? Because I can search for what I need inside of here and boom `setAllowedValues` `single_text`, `text` and `choice`. So let's paste `single_text` back into our file.

src/AppBundle/Form/MovieType.php

```
↕ // ... lines 1 - 6
7 use Symfony\Component\Form\Extension\Core\Type\DateType;
↕ // ... line 8
9 use Symfony\Component\Form\Extension\Core\Type\TextType;
↕ // ... lines 10 - 12
13 class MovieType extends AbstractType
14 {
15     public function buildForm(FormBuilderInterface $builder, array
   $options)
16     {
17         $builder->add('title', TextType::class)
↕ // ... lines 18 - 22
23             ->add('releasedAt', DateType::class, array(
24                 'widget' => 'single_text'
25             ));
26     }
↕ // ... lines 27 - 33
34 }
```

That trick of holding command and clicking works for the field types too: command+click `integer` and suddenly you're inside the class that provides this!

vendor/symfony/symfony/src/Symfony/Component/Form/Extension/Core/Type/IntegerT

```
↕ // ... lines 1 - 11
12 namespace Symfony\Component\Form\Extension\Core\Type;
↕ // ... lines 13 - 19
20 class IntegerType extends AbstractType
21 {
↕ // ... lines 22 - 37
38     public function configureOptions(OptionsResolver $resolver)
39     {
↕ // ... lines 40 - 47
48         $resolver->setDefaults(array(
49             // deprecated as of Symfony 2.7, to be removed in Symfony 3.0.
50             'precision' => null,
51             // default scale is locale specific (usually around 3)
52             'scale' => $scale,
53             'grouping' => false,
54             // Integer cast rounds towards 0, so do the same when
   displaying fractions
55             'rounding_mode' =>
   IntegerToLocalizedStringTransformer::ROUND_DOWN,
56             'compound' => false,
57         ));
↕ // ... lines 58 - 69
70     }
↕ // ... lines 71 - 78
79 }
```

It's like being teleported but, you know, without any risk to your atoms. You can even use this to take you to the property inside of `Movie` for that specific field.

Form Rendering

Our form is setup so let's go ahead and create this inside of our controller, `$form = $this->createForm(new MovieType(), $movie);`. Like always, we need to pass our form back into our template with `$form->createView()`:

src/AppBundle/Controller/MovieController.php

```
// ... lines 1 - 5
use AppBundle\Form\MovieType;
// ... lines 7 - 10
class MovieController extends Controller
{
// ... lines 13 - 15
    public function newAction()
    {
        $movie = new Movie();

        $form = $this->createForm(MovieType::class, $movie);

        return $this->render('movie/new.html.twig', array(
            'quote' => 'If my answers frighten you then you should cease asking scary questions. (Pulp Fiction)',
            'form' => $form->createView()
        ));
    }
// ... lines 27 - 34
}
```

Time to render this! Click into the `new.html.twig` template, ah that's right my form is actually going to be over here in `_form.html.twig`. It shouldn't surprise you that you'll get autocomplete here on things like `form_start` and `form_end`:

app/Resources/views/movie/_form.html.twig

```
{{ form_start(form) }}
// ... lines 2 - 8
{{ form_end(form) }}
```

Want more autocomplete awesomeness you say? You're mad! But ok: type `{{ form_row(form) }}` and it'll auto-complete the `title` field for you. So we'll plug in all of our fields here:

app/Resources/views/movie/_form.html.twig

```
{{ form_start(form) }}
    {{ form_row(form.title) }}
    {{ form_row(form.samsCharacterName) }}
    {{ form_row(form.isMainCharacter) }}
// ... lines 5 - 8
{{ form_end(form) }}
```

And if I forget one of them, I can hit `control+space` to bring up all of my options. This will also show you some other methods that exist on that `FormView` object. Ah ok so we still have

`rating` and `releasedAt` to add here. Clean up a bit of indentation here and perfect!

app/Resources/views/movie/_form.html.twig

```
{{ form_start(form) }}
    {{ form_row(form.title) }}
    {{ form_row(form.samsCharacterName) }}
    {{ form_row(form.isMainCharacter) }}
    {{ form_row(form.rating) }}
    {{ form_row(form.releasedAt) }}

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

Time to try this out: back to our new movies page, refresh and there we go! It renders with no problems other than the fact that this is a form only it's developer could love. And well, maybe not even that: I'm going to make it prettier. In `config.yml`, down in the `twig` key add `form_themes`:

app/config/config.yml

```
// ... lines 1 - 34
twig:
// ... lines 36 - 37
    form_themes:
// ... lines 39 - 83
```

Now this should autocomplete, but for whatever reason this one key is not doing that. But for the most part, you will see autocompletion inside of your configuration files.

Form Theming

Right here let's plug in the bootstrap form theme: and the plugin isn't perfect because we don't get autocomplete on this either. But I do know that there is a file inside the project for bootstrap, so I'll go to find, file and start typing in bootstrap. We want the one called `bootstrap_3_layout.html.twig`. To cheat, I'll just copy that file name and paste that in here:

app/config/config.yml

```
↕ // ... lines 1 - 34
35 twig:
↕ // ... lines 36 - 37
38     form_themes:
39         - bootstrap_3_layout.html.twig
↕ // ... lines 40 - 83
```

Refresh with our new form theme and .... Awesome!

Chapter 7: Live Templates

Now that we have our form rendering, we want to be able to submit it! Crazy idea, I know but let's see what we can do. We'll use our standard code for this `$form->handleRequest($request);`. Now we don't have the `$request` object yet, so let's put our type hint in for it. And like always, make sure you have the right one and let it autocomplete to get the `use` statement for you. And real quick, plug request in here too:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\HttpFoundation\Request;
↕ // ... line 11
12 class MovieController extends Controller
13 {
↕ // ... lines 14 - 16
17     public function newAction(Request $request)
18     {
↕ // ... lines 19 - 35
36     }
↕ // ... lines 37 - 44
45 }
```

Below, let's add `if ($form->isValid())` with a TODO to remind us to actually start saving stuff. Next, there are two things we typically do here: first add a flash message, like "Sam would be proud". And I truly think he would. And second, we redirect with `return $this->redirectToRoute()` and let's send our users to the `movies_list` route, which of course we get a nice autocomplete on:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 16
17      public function newAction(Request $request)
18      {
↕ // ... lines 19 - 22
23          $form->handleRequest($request);
24          if ($form->isValid()) {
25              // todo do some work, like save ...
26
27              $this->addFlash('success', 'Sam would be pleased');
28
29              return $this->redirectToRoute('movies_list');
30          }
↕ // ... lines 31 - 35
36      }
↕ // ... lines 37 - 46
```

Creating a Live Template

That was pretty easy, but we'll be writing those 8 lines of code over and over and over again inside of our project. There must be a better way! To fix that we are going to use something called a "Live Template". That's what lets us just type `a` in a twig template, hit tab and get a full `a` tag printed out. We want to do the same type of thing and have it generate all this form handling boiler plate code for us.

To make this, first select all the code that you want included. Then click on the Tools dropdown menu, and select "Save as Live Template". Let's give this a name, like `formhandle` with a description of "Adds controller form handling code". Inside the template text tweak the indentation but other than that, let's just leave this alone for now. Hit 'ok'.

Now that we have this, we can delete our hard work. Replace it by typing `formhandle`, hitting tab, and then... boom!

src/AppBundle/Controller/MovieController.php

```
// ... lines 1 - 16
    public function newAction(Request $request)
    {
// ... lines 19 - 22
        $form->handleRequest($request);
        if ($form->isValid()) {
            // todo do some work, like save ...

            $this->addFlash('success', 'Sam would be pleased');

            return $this->redirectToRoute('movies_list', array());
        }
// ... lines 31 - 35
    }
// ... lines 37 - 46
```

Enjoy that autocompletion.

Live Template Variables

Of course now we need to make parts of this dynamic. We'll need to edit our live template...how do we do that? Well, it's probably in the preferences menu and if we search for it, it is!

We'll assume that the variable is always called `$form` because it usually is, but we do want to change our success message here. Let's swap it out for `'$SUCCESSMESSAGE$'` with dollar signs on both sides and surrounded in quotes. Let's do the same thing down here in the redirect, we'll change that to `'$ROUTENAME$'`. In the event that the route has parameters add in array next to it for convenience. Save this.

Get rid of the code and again type in `formhandle` and tab. Type in 'Sam would be proud', hit tab, type `movie_list` for our redirect, then tab one more time. Now that is awesome and so much faster than typing it all out again and again and again.

Another great spot to use Live Templates is inside of this form row stuff since we'll be typing this all the time. Copy one of our rows, go to Tools and "Save as Live Template". We'll call this one `formrow` with a description of "Renders form_row in Twig". Below, there's a note saying its applicable for HTML, HTML text and HTML Twig files - so that's right. Let's update this line of code to be `form.$FIELD$`. Click ok and now we can try this out.

Type `formrow`, hit tab and now we can just start typing in the field property name and select it from the list that pops up. Now, as great as `formrow` is, sometimes you can't use it. womp womp. That's when you'll end up rendering a field manually, which will probably be a `div` and the three parts: `{{ form_label(form.releasedAt) }}`. Copy that and update for `form_widget`, and `form_errors`:

app/Resources/views/movie/_form.html.twig

```
{{ form_start(form) }}
// ... lines 2 - 6
    <div class="form-group">
        {{ form_label(form.releasedAt) }}
        {{ form_widget(form.releasedAt) }}
        {{ form_errors(form.releasedAt) }}
    </div>
// ... lines 12 - 13
{{ form_end(form) }}
```

Clearly this is also begging to be turned into a Live Template.

You know the drill! Select the full div, go to Tools, Save as Live Template. Let's name this one `formrowfull`, and it "Renders widget/label/errors". What's cool here is that there's just one variable that's duplicated 3 times. So we can just say `$FIELD$` and repeat that each time. Just a quick indentation fix there and click ok. Awesome!

Let's get rid of the div we had, type `formrowfull`, hit the tab key, and as we start typing the property name we get the autocomplete and it's filling instantly on each of the three lines. I'll fix the spacing here to wrap it up!

So, live templates: big win.

Chapter 8: Fast Navigating

I just hate seeing developers click manually into their directory tree to search for things. Most projects are too big to be opening files like this. Really, you want to have your hands off your mouse as often as possible.

Time to introduce you to some new best friend shortcuts for navigating! In fact, PHPStorm thinks they're so important that they even have some of these listed front and center when you don't have any files open.

Navigating to Class and File

Click the navigate menu at the top. The first option in the list is "Navigate to Class". Its shortcut is `command+O`. Knowing that I want to get into the `MovieType` class, I'll hit `command+O`, start typing the name and boom, I've got that open with just my keyboard.

Need to open a file, like config.yml? Well you're in luck, just use `command+shift+O`, start typing the file name in the search box and there it is. We could even look up this `bootstrap_3_layout` thing here. Use `command+shift+O` again, type the file name, and we can see where it is in the project.

Great! In this case we can see how deep that structure is for this particular file, it's way down in Symfony. If I saw you navigating here manually, I'd be tempted send Samuel L Jackson to your office to tip over your chair. He'd arrive before you navigated there.

Navigating Deep Directories

And what's *really* cool - other than *not* being tipped over by Sam - is that we can use the navigation tree on top to actually click and look around inside of these deep directories.If we really wanted to, we could move the tree on the left all the way to that specific spot by double clicking that directory name at the top. And now we're there and can look around and see what other interesting stuff is hiding here.

Navigating to Function Names

The next shortcut on the list up there is "Symbol", which is `command+alt+O`. Symbol is referring basically to function names. So here we can look for something like the `buildForm` method from inside of my `MovieType`. And now it's showing me all of the instances in the project where `buildForm` is used. I'll pick the top one and it takes me straight there.

Inside of my controller I call `createForm`. So back to `command+alt+O`, type `createForm`, and there is the one inside of the base `Controller` class we extend. This is the function I am calling when I say `$this->createForm()` from inside `MovieController`.

Clicking into Functions

In this case, if I wanted to know exactly where this `createForm()` function lives, I would just hold command and click into it -- so don't forget about that shortcut as well. We can do the same thing here for `handleRequest` if we needed to know where that is. Clicking that will take us right into this core Symfony spot and again we can look around inside of here and double click the tree to move us here.

Recent Files

Most of the time, you might be interested in the shortcut `command+E`: this will pop up a list of your recent files. Just start typing the name of the file you're looking for until it appears in the results.

Navigating Tabs

Once you have some tabs open, you'll see me do this a lot: my cursor will stay in place while I hold `command+shift+[` and `command+shift+]` to move around the open files. I use this constantly to jump around from one file to the next.

Searching ALL THE THINGS!

Okay, the last shortcut I'm going to show you for finding things in your project is easy to remember: `shift+shift`. Hitting shift twice will open up a feature called 'search everywhere'. This is a bunch of searches put together, where you can start typing `MovieController` and find that, or `buildForm` and it will find that for you. Or you can even find CSS tags. In one of the core Symfony files, there's a CSS snippet called `.block-exception`. And there it is in our list of results. And clicking on it takes you straight to that file deep inside of Symfony.

There are a lot of ways to move around your project, but the biggest ones to remember are `command+O` and `command+shift+O`. Or put them together, with `shift+shift`.

Ya! We're working way faster.

Chapter 9: Refactoring

Ok back to work, let's finish up this `MovieController`. `Command+O`, `MovieController`, and now we've got that file open.

We need to save the POST'ed movie to the database. Start with getting our entity manager with `$em = $this->getDoctrine()->getManager();`. Then `$em->persist($movie);`, and `$em->flush();`:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 11
12 class MovieController extends Controller
13 {
↕ // ... lines 14 - 16
17     public function newAction(Request $request)
18     {
↕ // ... lines 19 - 23
24         if ($form->isValid()) {
25             $em = $this->getDoctrine()->getManager();
26             $em->persist($movie);
27             $em->flush();
↕ // ... lines 28 - 32
33         }
↕ // ... lines 34 - 38
39     }
↕ // ... lines 40 - 53
54 }
```

Nice and simple!

While we're here, I'll copy the `getDoctrine()` line and paste it into `listAction()` so we can make that work. Here we'll query for the movies with `$em ->getRepository()` and the first cool thing here is that... you guessed it! We get autocomplete on this. Type the entity name, select the right one and hit tab:

src/AppBundle/Controller/MovieController.php

```
// ... lines 1 - 11
class MovieController extends Controller
{
// ... lines 14 - 43
    public function listAction()
    {
        $em = $this->getDoctrine()->getManager();
        $movies = $em->getRepository('AppBundle:Movie')
// ... lines 48 - 52
    }
}
```

The second cool thing is that it autocompletes the methods on the repository, and this includes any custom methods you have inside of there. We'll use the normal `findAll();`. Down below, we'll `return $this->render('movies/list.html.twig');` which doesn't exist yet, with an array and we'll pass it movies:

src/AppBundle/Controller/MovieController.php

```
// ... lines 1 - 11
class MovieController extends Controller
{
// ... lines 14 - 43
    public function listAction()
    {
        $em = $this->getDoctrine()->getManager();
        $movies = $em->getRepository('AppBundle:Movie')
            ->findAll();

        return $this->render('movie/list.html.twig', array(
            'movies' => $movies,
        ));
    }
}
```

Awesome!

Let's go create that file: `list.html.twig`. And to save some time, I'll just paste in some code for this template: it's really straight forward:

app/Resources/views/movie/list.html.twig

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

{% block body %}
    <table class="table">
        <thead>
            <tr>
                <th>Movie name</th>
                <th>Character</th>
            </tr>
        </thead>
        <tbody>
        {% for movie in movies %}
            <tr>
                <td>{{ movie.title }}</td>
                <td>{{ movie.samsCharacterName }}</td>
            </tr>
        {% endfor %}
        </tbody>
    </table>
{% endblock %}
```

The only interesting thing here is that you do get autocomplete on the different properties for `movie`, Which really is pretty amazing. The reason that works is because PHPStorm knows that the `MovieRepository` returns `Movie` objects, so that cascades all the way down into Twig.

So as long as you have good PHPDoc that says what types of objects functions return, that should connect down to how things auto-complete in Twig.

Let's try this whole thing out. But before we do that we need to setup our database. Head over to `parameters.yml` by using our shortcut `command+shift+O` to open it up. Everything seems to be in order here, so let's just change our database to be `phpstorm`.

And just when you thought I couldn't get any lazier with my shortcuts I'm going to use the built in terminal from PHPStorm. This drops me right into the correct directory so I can run:

```
./app/console doctrine:database:create
./app/console doctrine:schema:create
```

Hide this little terminal, head back and refresh our browser. And we see... oops we have a small rendering problem, which is probably in our form template.

Back to PHPstorm, `command+shift+O` and search `_form.html.twig`. And yep there it is, change `form-control` to `form-group` - that's the *correct* Bootstrap class. Refresh again. That looks way better.

Now we can fill out the form. Let's see, which one of our favorite Samuel L Jackson movies should we choose first - so many choices. Let's just go with the obvious best: "Snakes on a Plane", with our character, Neville Flynn. Neville *is* the main character, and clearly this film was a 10 out of 10. I mean come on, this is Sam at his best! And I think everyone remembers the fateful day this film was released: August 18, 2006. Let's save this small piece of history.

Refactoring to a Method

Beautiful! Now for a little refactoring. First, we have `$em = $this->getDoctrine()->getManager();` in a couple of places. So I'm going to refactor that: select that line and we'll see another important shortcut, `control+t`. This is the refactor shortcut. If you ever forget it, just go to the Refactor menu at the top and select "Refactor This" from the list.

We've got our line selected, press `control+t`. There's a lot of options in here, but we want the one called method. In the "Extract Method" form, add `getEm` as the name, make sure that the private radio button is selected and refactor!

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 11
12 class MovieController extends Controller
13 {
↕ // ... lines 14 - 43
44     public function listAction()
45     {
46         $em = $this->getEm();
↕ // ... lines 47 - 52
53     }
↕ // ... lines 54 - 57
58     private function getEm()
59     {
60         $em = $this->getDoctrine()->getManager();
61         return $em;
62     }
63 }
```

Boom! It puts this on the bottom and it changes the original code to `$this->getEm();`. Copy this line, pressing `control+c` will select the whole line and we'll paste it right up here:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 11
12 class MovieController extends Controller
13 {
↕ // ... lines 14 - 16
17     public function newAction(Request $request)
18     {
↕ // ... lines 19 - 24
25             $em = $this->getEm();
↕ // ... lines 26 - 38
39     }
↕ // ... lines 40 - 62
63 }
```

Refactoring to a BaseController

This shortcut is going to be useful in more than just this controller, so this is the perfect time to create a `BaseController` for my whole project. Click the `Controller` directory at the top of your IDE, and from there press `command+n`, select `Controller` from the menu and name it `BaseController`. To start, remove that public function and add `abstract` before the class:

src/AppBundle/Controller/BaseController.php

```
1 <?php
2
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6
7 class BaseController extends Controller
8 {
9 }
```

This is now the spot for our own shortcuts. In `MovieController` update it to extend `BaseController`. This `use` statement is showing up in a darker gray to indicate that it isn't being used any more and the same for the `Response` `use` statement. Delete both of those:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 10
11 class MovieController extends BaseController
12 {
↕ // ... lines 13 - 61
62 }
```

Tip

Or you can go "Code" -> "Optimize Imports" instead (`Control` + `Alt` + `O` on a Mac) to remove unused imports, add missing imports, and organize import statements in the current file.

With the `BaseController` in action, let's refactor `getEm()` into the `BaseController`. Select the method, one cool way to do that is hit `option+up` which will select larger and larger contexts. Now hit `control+t`. Select "Pull Members Up" from the menu, which will pull them up to the parent class. It already recognizes that `BaseController` is the destination, and at the bottom it's warning us that the access will be changed from private to protected, which is awesome! Hit refactor and we can see it's gone from this file, because now it lives inside of `BaseController`:

src/AppBundle/Controller/BaseController.php

```
↕ // ... lines 1 - 6
7 class BaseController extends Controller
8 {
9
10     /**
11      * @return \Doctrine\Common\Persistence\ObjectManager|object
12      */
13     protected function getEm()
14     {
15         $em = $this->getDoctrine()->getManager();
16         return $em;
17     }
18 }
```

That there is a really fast way to refactor things!

Refactoring to Rename a Method

Now that I have this here I'm realizing that `getEm()` is not that great of a name choice. So back to `command+t`, select rename from the menu and change it to `getEntityManager` to make my code a little clearer. When we do that I get a summary down here of all the spots in our project where PHPStorm sees `getEm()`. Click the "Do Refactor" button to rename this and all the other spots in the code that need updating:

src/AppBundle/Controller/BaseController.php

```
↕ // ... lines 1 - 6
7  class BaseController extends Controller
8  {
9
10     /**
11      * @return \Doctrine\Common\Persistence\ObjectManager|object
12      */
13     protected function getEntityManager()
14     {
15         $em = $this->getDoctrine()->getManager();
16         return $em;
17     }
18 }
```

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 10
11 class MovieController extends BaseController
12 {
↕ // ... lines 13 - 15
16     public function newAction(Request $request)
17     {
↕ // ... lines 18 - 23
24             $em = $this->getEntityManager();
↕ // ... lines 25 - 37
38     }
↕ // ... lines 39 - 42
43     public function listAction()
44     {
45         $em = $this->getEntityManager();
↕ // ... lines 46 - 51
52     }
53 }
```

Sah-weet!

There are a lot of other things you can do with the refactor feature in PHPStorm. For example, we can extract this out to a variable. This could add a level of clarity to what things are.

Or, say you get into a spot where you messed up your formatting in some terrible way. Oof that looks just awful. Make it stop! At any point, you can go up to the code menu at the top and select reformat code, which is also `command+option+L` and it will tidy that back up for you.

So let's hit `command+A` to select all the code I want to reformat, then `command+option+L` and now everything is back into place based on our coding standards, which you can of course control.

Chapter 10: Services

Alright, remember this hardcoded quote? Let's create a new class that can replace this with a random Samuel L Jackson quote instead.

In `AppBundle` I'll create a new directory called `Service`. Inside of our new `Service` directory let's add a php class called `QuoteGenerator`, and look how nicely it added that namespace for us!

src/AppBundle/Service/QuoteGenerator.php

```
1  <?php
2
3  namespace AppBundle\Service;
4
5  class QuoteGenerator
6  {
↕  // ... lines 7 - 20
21 }
```

Let's get to work by adding `public function getRandomQuote()`. I'll paste in some quotes, then we can use `$key = array_rand($quotes);` to get one of those quotes and return it:

src/AppBundle/Service/QuoteGenerator.php

```
// ... lines 1 - 4
class QuoteGenerator
{
    public function getRandomQuote()
    {
        $quotes = array(
            'If my answers frighten you then you should cease asking scary questions. (Pulp Fiction)',
            'Now that we know who you are, I know who I am. (Unbreakable)',
            'Enough is enough! I have had it with these m*****f*****g snakes on this m*****f*****g plane! (Snakes on a Plane)',
            'Say "what" again. SAY "WHAT" AGAIN! I dare you, I double dare you (Pulp Fiction)',
        );

        $key = array_rand($quotes);
        $quote = $quotes[$key];

        return $quote;
    }
}
```

Registering a Service

Next, I want to register this as a service and use it inside of my controller. So hit `Shift`+`Command`+`O`, search for `services.yml`, delete the comments under the services key, and put our service name there instead. I'll give it a nickname of `quote_generator`:

app/config/services.yml

```
// ... lines 1 - 5
services:
    quote_generator:
// ... lines 8 - 10
```

Tip

If you're using Symfony 3.3, your `app/config/services.yml` contains some extra code that may break things when following this tutorial! To keep things working - and learn about what this code does - see https://knpuniversity.com/symfony-3.3-changes

Notice that PHPStorm is autocompleting my tabs wrong, I want to hit tab and have it give me four spaces. So let's fix that real quick in preferences by first hitting `command+,`, then searching for "tab". In the left tree find yml under "Code Style" and update the indent from 2 to 4. Click apply, then ok and that should do it!

Yep, that looks perfect. As I was saying: we'll call the service, `quote_generator`, but this name really doesn't matter. And of course we need the `class` key, and we have autocomplete here too. If you hit `Control`+`Space` you'll get a list of all the different keys you can use to determine how a service is created, which is pretty incredible.

So we'll do `class`, but don't type the whole long name! Like everywhere else, just type the last part of the class: here `Quote` and hit tab to get the full line. Now add an empty `arguments` line: we don't have any of those yet:

app/config/services.yml

```
↕ // ... lines 1 - 5
6 services:
7     quote_generator:
8         class: AppBundle\Service\QuoteGenerator
9         arguments: []
```

This is now ready to be used in `MovieController`.

Use `Command`+`Shift`+`]` to move over to that tab. And here instead of this quote, we'll say `$this->get('')` and the plugin is already smart enough to know that the `quote_generator` service is there. And it even knows that there is a method on it called `getRandomQuote()`:

src/AppBundle/Controller/MovieController.php

```
↕ // ... lines 1 - 10
11 class MovieController extends BaseController
12 {
↕ // ... lines 13 - 15
16     public function newAction(Request $request)
17     {
↕ // ... lines 18 - 33
34         return $this->render('movie/new.html.twig', array(
35             'quote' => $this->get('quote_generator')->getRandomQuote(),
↕ // ... line 36
37         ));
38     }
↕ // ... lines 39 - 52
53 }
```

This is one my *favorite* features of the Symfony plugin.

Save that, head back to the form, refresh and now we see a new random quote at the bottom of the page.

Generating the Constructor

Now in `QuoteGenerator`, let's pretend like we need access to some service - like maybe we want to log something inside of here. The normal way of doing that is with dependency injection, where we pass the logger through via the `constructor`. So let's do exactly that, but with as little work as possible.

I could type `public function __construct`, but instead I'm going to use the generate command. Hit `Command`+`N` and pick the "Constructor" option from the menu here. I don't think this constructor comment is all that helpful, so go back into preferences with `Command`+`,`, search for "templates", and under "file and code templates", we have one called "PHP Constructors". I'll just go in here and delete the comment from the template.

Ok, let's try adding the constructor again. Much cleaner:

src/AppBundle/Service/QuoteGenerator.php

```
↕ // ... lines 1 - 4
5  class QuoteGenerator
6  {
7
8      public function __construct()
9      {
10     }
↕ // ... lines 11 - 25
26 }
```

Generating Constructor Properties

At this point we need the logger, so add the argument `LoggerInterface $logger`:

src/AppBundle/Service/QuoteGenerator.php

```
↕ // ... lines 1 - 4
5  use Psr\Log\LoggerInterface;
6
7  class QuoteGenerator
8  {
↕ // ... lines 9 - 14
15     public function __construct(LoggerInterface $logger)
16     {
17         $this->logger = $logger;
18     }
↕ // ... lines 19 - 34
35 }
```

This is the point where we would normally create the `private $logger` property above, and set it down in the constructor with `$this->logger = $logger;`. This is a really common task, so if we can find a faster way to do this that would be awesome.

Time to go back to the actions shortcut, `Option`+`Enter`, select "Initialize fields", then choose `logger`, and *it* adds all that code for you:

src/AppBundle/Service/QuoteGenerator.php

```
↕ // ... lines 1 - 6
7  class QuoteGenerator
8  {
9
10     /**
11      * @var LoggerInterface
12      */
13     private $logger;
14
15     public function __construct(LoggerInterface $logger)
16     {
17         $this->logger = $logger;
18     }
↕ // ... lines 19 - 34
35 }
```

We don't even have to feed it!

Farther down, it's really easy to use, `$this->logger->info('Selected quote: '.$quote);`:

src/AppBundle/Service/QuoteGenerator.php

```
↕ // ... lines 1 - 6
7  class QuoteGenerator
8  {
↕ // ... lines 9 - 19
20     public function getRandomQuote()
21     {
↕ // ... lines 22 - 29
30         $quote = $quotes[$key];
31         $this->logger->info('Selected quote: '.$quote);
↕ // ... lines 32 - 33
34     }
35 }
```

We've added the argument here, so now we need to go to `services.yml`, which I'll move over to. And notice it's highlighting `quote_generator` with a missing argument message because it knows that this service has one argument. So we can say `@logger`, or even `Command`+`O` and then use autocomplete to help us:

app/config/services.yml

```
// ... lines 1 - 5
services:
    quote_generator:
        class: AppBundle\Service\QuoteGenerator
        arguments: ['@logger']
```

Head back, refresh, it still works and I could go down here to click into my profiler to check out the logs. Or, in PhpStorm, we can go to the bottom right Symfony menu, click it and use the shortcut to get into the Profiler. Click that, go to logs, and there's our quote right there.

The autocompletion of the services and the ability to generate your properties is probably one of the most important features that you need to master with PHPStorm because it's going to help you fly when you develop in Symfony.

With <3 from SymfonyCasts