

# Making Fixtures Really Awesome with Alice



With <3 from SymfonyCasts

# Chapter 1: Making Fixtures Awesome with Alice

## Tip

A newer version of HauteLookAliceBundle has been released and portions of this tutorial won't apply to that new version.

Fixtures, those nice little bits of data that you can put into your database so that when you're developing locally you actually have something to look at: like products or users or whatever else you want to play with. I hate writing fixtures because I use this bundle ([DoctrineFixturesBundle](#)) and it's boring and manual and you end up building really big ugly fixture classes:

```
// a boring DataFixtures class
// ...

class MyFixtures implements FixtureInterface
{
    public function load(ObjectManager $manager)
    {
        $user1 = new User();
        $user1->setUsername('foo');
        // ...
        $manager->persist($user1);
        // repeat over and over...

        $manager->flush();
    }
}
```

The reason is that this library just doesn't do anything for you. It's entirely up to you to manage, persist and flush everything that you do.

## [Introducing Alice + Our Video Game Hero App](#)

So I'm going to show you a better way, in fact a much better way with a library called [Alice](#). I prepared a small Symfony application for us which we'll talk about in a second. And I've already started our built-in PHP Web server:

## Tip

Want to code along? Beautiful! Just download the code on this page.



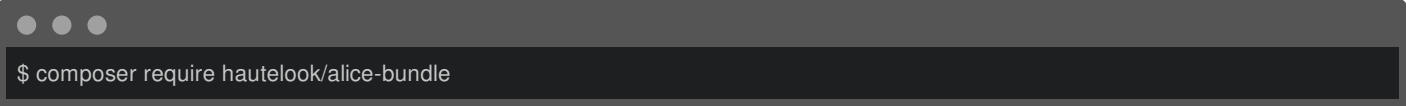
```
$ php app/console server:run
```

And here's our app, it's all about listing our favorite videogame heroes. Right now we're losing because there is nothing in this table. But that's what we're going to fix with Alice.

This table is actually pulling from the Character entity so this is what we actually need to save to the database.

## [Installing Alice via HauteLookAliceBundle](#)

To install Alice we could do it directly but instead I'm going to use an awesome bundle called HauteLookAliceBundle\_. Let's grab the composer require command from it's README and paste that into the terminal:



```
$ composer require hautelook/alice-bundle
```

This bundle is a thin layer around the Alice library, which is something that let's us load fixtures with yml files, and the same DoctrineFixturesBundle that we were talking about before. This is a really nice combination because it's going to mean that

we can still run our normal php app/console doctrine:fixtures:load. But after that, instead of writing raw PHP code, all of our fixtures are going to be in these really nice yml files.

And if that doesn't sound awesome yet, just hang with me. Alice is a lot more than yml files - it contains tons of goodies.

Next let's activate the bundle. In fact if you head back to its documentation you'll see that you need to initialize both this bundle *and* the DoctrineFixturesBundle in our AppKernel. So grab both of those lines, open the AppKernel and let's put it there:

```
// app/AppKernel.php
// ...

if (in_array($this->getEnvironment(), array('dev', 'test'))) {
    // ...

    $bundles[] = new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle();
    $bundles[] = new Hautelook\AliceBundle\HautelookAliceBundle();
}
```

But since I'm only going to be loading my fixtures when I'm developing, I'm going to go ahead and put these inside of the dev environment block. That way, in production I have just a little bit less in my application.

You *do* need one fixture class, but we can just copy it from the documentation and put it into our application. I'll create the DataFixtures/ORM directory. By the way this stuff *does* work with the ODM or other doctrine libraries. And I'll create a file called AppFixtures. Copy the contents in there and don't forget to update your namespace and rename the class:

```
// src/AppBundle/DataFixtures/ORM/AppFixtures.php
namespace AppBundle\DataFixtures\ORM;

use Hautelook\AliceBundle\Alice\DataFixtureLoader;
use Nelmio\Alice\Fixtures;

class AppFixtures extends DataFixtureLoader
{
    /**
     * {@inheritdoc}
     */
    protected function getFixtures()
    {
        return array(
            __DIR__ . '/test.yml',
        );
    }
}
```

The fixtures class is special because it's already wired up to load yml files. Let's call ours characters.yml and then go ahead and create that file:

```
// src/AppBundle/DataFixtures/ORM/AppFixtures.php
// ...

protected function getFixtures()
{
    return array(
        __DIR__ . '/characters.yml',
    );
}
```

## [Your First Alice yml File](#)

Now, here is how Alice works. Inside the yml file this is now pure Alice code. You start with the full entity namespace. This tells Alice what type of object it's going to create. Below that, we just start inventing keys. These aren't important yet but they *will* be later when we start linking two entities together. Under that we just give each property a value. Let's create Yoshi:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
  character1:
    name: Yoshi
    realName: T. Yoshisaur Munchakoopas
    highScore: 99999
    email: yoshi@nintendo.com
    tagLine: Yoshi!
```

Let's cheat and look back at the Character entity to see what other fields we want to fill in. We now have a fully functional and armed single-file fixture. So let's try it out.

## [Loading your Fixtures](#)

As I mentioned earlier, this is a wrapper around the Doctrine fixtures library so we use the same php app/console doctrine:fixtures:load command to run everything. No errors is good so let's try refreshing the page. Yoshi!


## [Loading A LOT of Test Data \(Ranges\)](#)

If this is all that Alice gave us I wouldn't be telling you about it. It actually gives us a ton more. So usually in fixtures you want a lot of things. Like five characters or ten characters or 50 blog posts or something like that.

One of the most powerful features of Alice is this range syntax:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
  character1:
    name: Yoshi
    realName: T. Yoshisaur Munchakoopas
    highScore: 99999
    email: yoshi@nintendo.com
    tagLine: Yoshi!
  character{2..10}:
    name: Mario
    realName: Homo Nintendonus
    highScore: 50000
    email: mario@nintendo.com
    tagLine: Let's a go!
```

So, in this case we're going to be creating characters two through 10. Behind the scenes you can see how this is basically a for loop but the syntax is a lot cleaner. To test that out let's reload our fixtures:



```
$ php app/console doctrine:fixtures:load
```

And now Mario is taking over our database!

So we have 10 characters now but since nine of them are identical they're not very realistic. But this is where Alice gets really interesting. It has this special <> syntax which allows you to call functions that are special to Alice.

For example, when you're inside of a range you can use this syntax to call the <current()> function that's going to give us whatever index were at in that moment:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
  # ...
  character{2..10}:
    name: Mario<current()>
    realName: Homo Nintendonus
  # ...
```

So let's reload our fixtures again and now we have Mario2, Mario3, Mario4.

## [Introducing Faker: For all your Fake Data Needs](#)

So this is better but still not very realistic. Behind the scenes Alice hooks up with another library called Faker\_. And as it's

name sounds it's all about creating fake data. Fake names, fake company names, fake addresses, fake e-mails - it supports a ton of stuff. To use Faker we just use that same syntax we saw and use one of the many built-in functions.

For example, one of the functions is called `firstName()`. Since this is going to return us some pretty normal names, let's put the word Super in front of it so at least it sounds like a superhero:


```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
# ...
character{2..10}:
  name: Super <firstName()>
  realName: Homo Nintendonus
# ...
```

Then we're going to use a few others like `name()`, `numberBetween()`, `email()` and `sentence` which gives us one random sentence:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
# ...
character{2..10}:
  name: Super <firstName()>
  realName: <name()>
  highScore: <numberBetween(100, 99999)>
  email: <email()>
  tagLine: <sentence()>
```

These functions are pretty self-explanatory but if you Google for "Faker PHP" and scroll down on the README just a little bit, they have a huge list\_ of all the functions that they support. They're actually called formatters but a lot of them take arguments.

For example you can see our `numberBetween`, `sentence` and even some things for creating random names where you can choose which gender you want. So let's check this out. Reload your fixtures, scroll back over refresh the page.



```
$ php app/console doctrine:fixtures:load
```

Now we have ten super friends and no identical data.

### [Making a Field \(sometimes\) Blank](#)

If you want to make one of these fields sometimes empty you can do that as well. For example, if `tagLine` is optional then you may want to see what your set looks like when some of the characters don't have one. To do that create a percentage put a ? after it and then list what value you want:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
# ...
character{2..10}:
# ...
tagLine: 80%? <sentence()>
```

So in this case 80% of the time we're going to get a random sentence and 20% of the time we're going to get nothing. So reload the fixtures, and this time you see that about 20% of our characters are missing their tag line.

### [Creating your Own Faker Formatter \(Function\)](#)

So I love the random data, I love how easy this is. But one thing I don't like is that our names just aren't that realistic. We're dealing with video game heroes here and none of our names are actually of real video game heroes.

To fix this let's create our own formatter called `characterName`:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
# ...
character{2..10}:
  name: <characterName()>
  realName: <name()>
  highScore: <numberBetween(100, 99999)>
  email: <email()>
  tagLine: <sentence()>
```

Now if you try this out you are going to get the error that the formatter is missing:

```
Unknown formatter "characterName"
```

So how do we create it? With the bundle it's super easy. Just go back to your fixtures class, AppFixtures and create a function called characterName. And in this function we just need to return a character name. I'll paste in a few of my favorites and then at the bottom we'll use the `:phpfunction:array_rand` function to return a random character each time Alice calls this:


```
// src/AppBundle/DataFixtures/ORM/AppFixtures.php
// ...

class AppFixtures extends DataFixtureLoader
{
    // ...

    public function characterName()
    {
        $names = array(
            'Mario',
            'Luigi',
            'Sonic',
            'Pikachu',
            'Link',
            'Lara Croft',
            'Trogdor',
            'Pac-Man',
        );

        return $names[array_rand($names)];
    }
}
```

I love when things are this simple!



```
$ php app/console doctrine:fixtures:load
```

Flip back to the browser and when you refresh this time, real video game heroes!

## [True Love with Relationships](#)

So there's one more complication that I want to introduce, and that's relationships. I have an entity called Universe as in "Nintendo Universe" or "Sega Universe".

First, let's go into our yml file and create a few of these. We'll start just like before by putting the namespace and creating a few entries under that. So I'll have one for Nintendo, one for Sega and one for classic arcade:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
  # ...

AppBundle\Entity\Universe:
  universe_nintendo:
    name: Nintendo
  universe_sega:
    name: Sega
  universe_arcade:
    name: Classic Arcade
```

The Character entity already has a ManyToOne\_ relationship to universe on a universe property:

```
// src/AppBundle/Entity/Character.php
// ...

class Character
{
  // ...

  /**
   * @var Universe
   * @ORM\ManyToOne(targetEntity="Universe")
   */
  private $universe;
}
```


So our goal is to take these Universe objects and set them on the charcter property.

To reference another object, just use the @ symbol and then the internal key to that object. So we'll link Mario to the Nintendo universe and everyone else, for now, to the Sega Universe:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
  character1:
    name: Yoshi
    # ...
    universe: @universe_nintendo

  character{2..10}:
    name: <characterName()>
    # ...
    universe: @universe_sega

AppBundle\Entity\Universe:
  universe_nintendo:
    name: Nintendo
  universe_sega:
    name: Sega
  universe_arcade:
    name: Classic Arcade
```



```
$ php app/console doctrine:fixtures:load
```

When we check it out now, sure enough we see Nintendo on top followed by 9 Segas. So I know you're thinking, "can we somehow randomly assign random universes to the characters?" And absolutely! In fact, the syntax is ridiculously straight forward. Just get rid of the sega part and put a star:

```
# src/AppBundle/DataFixtures/ORM/characters.yml
AppBundle\Entity\Character:
# ...

character{2..10}:
# ...
  universe: @universe_*

AppBundle\Entity\Universe:
  universe_nintendo:
    name: Nintendo
  universe_sega:
    name: Sega
  universe_arcade:
    name: Classic Arcade
```

Now, Alice is going to find any keys that start with `universe_` and randomly assign them to the characters. Reload things again and now we have a nice assortment of universes:

```
$ php app/console doctrine:fixtures:load
```

## Using Multiple yml Files

Because our project is pretty small I've kept everything in a single file, which I recommend that you do until it gets just too big. Once it does, feel free to separate into multiple yml files.

In our case I'll create a `universe.yml` file and put the universe stuff in it:

```
# src/AppBundle/DataFixtures/ORM/universe.yml
# these have been removed from characters.yml
AppBundle\Entity\Universe:
  universe_nintendo:
    name: Nintendo
  universe_sega:
    name: Sega
  universe_arcade:
    name: Classic Arcade
```

Of course when you do this it's not going to work because it's only loading the `characters.yml` file right now. So we get a missing reference error:

```
Reference universe_nintendo is not defined
```

There are actually a few ways to load the two yml files but the easiest is to go back into your `AppFixtures` class and just add it to the array:

```
// src/AppBundle/DataFixtures/ORM/AppFixtures.php
// ...

protected function getFixtures()
{
    return array(
        __DIR__ . '/universe.yml',
        __DIR__ . '/characters.yml',
    );
}
```

Unfortunately, order *is* important here. So since we're referencing the universes from within the `characters.yml` we need to load the `universe.yml` file first. Let's reload things to make sure they're working.

```
$ php app/console doctrine:fixtures:load
```

And they are!



## Joyful Fixtures

To back up, after we installed the bundle we only really touched two things. The AppFixtures class, which has almost nothing in it, and our yml files which are very very small and straight forward. This is awesome! This puts the joy back into writing fixtures files for me and I absolutely love it.

There are a few topics that we haven't talked about like processors and templates but I'll cover those in a future lesson.

See you guys!

# Chapter 2: Processors: Do Custom Stuff While Loading

## Tip

A newer version of HauteLookAliceBundle has been released and portions of this tutorial won't apply to that new version.

I don't want to brag, but these are probably the nicest super-hero fixtures ever. But we've neglected a column! The avatar.

Check out the Character entity - we have a column for this called avatarFilename:

```
141 lines | src/AppBundle/Entity/Character.php
... lines 1 - 10
11 class Character
12 {
... lines 13 - 54
55 /**
56  * @ORM\Column(nullable=true)
57  */
58 private $avatarFilename;
... lines 59 - 139
140 }
```

It's going to hold *just* the filename part of an image, like trogdor.png or pac-man-is-the-man.jpg. This makes sense if you look in the template for the homepage. *If* there's an avatarFilename, we print an img tag and expect the image to be in some uploads/avatars directory, relative to web/:

```
45 lines | app/Resources/views/Homepage/homepage.html.twig
... lines 1 - 16
17 {% for character in characters %}
18     <tr>
... lines 19 - 24
25     <td>
... lines 26 - 31
32         {% if character.avatarFilename %}
33             
34         {% endif %}
35     </td>
36 </tr>
... lines 37 - 45
```

Oh boy, so this means the avatar is a bit harder. Yea, we have to set the value in the database, but we *also* need to make sure to put a corresponding image file into this directory. I don't want a bunch of broken images!

## Filling in avatarFilename Data

But, we'll worry about that later. First, let's get some values into the avatarFilename field. Open up characters.yml and start to set the avatarFilename.

```
18 lines | src/AppBundle/DataFixtures/ORM/characters.yml
```

```
1  AppBundle\Entity\Character:
  ... lines 2 - 9
10  character{2..10}:
11    name: <characterName()>
  ... lines 12 - 16
17    avatarFilename:
```

For *any* of this to work, we're going to need some *real* image files handy. Fortunately, I got some for us! They live in a resources directory at the root of the project:

```
resources/
  kitten1.jpg
  kitten2.jpg
  kitten3.jpg
  kitten4.jpg
```

But since I want to avoid any trademark legal battles with Nintendo, I've decided that instead of Mario and Yoshi, we'll use readily-available images of kittens. Thank you Internet.

So we need our value to be *one* of these. Let's setup a custom Faker formatter like we did before. Call this one avatar():

```
18 lines | src/AppBundle/DataFixtures/ORM/characters.yml
```

```
1  AppBundle\Entity\Character:
  ... lines 2 - 9
10  character{2..10}:
11    name: <characterName()>
  ... lines 12 - 16
17    avatarFilename: <avatar()>
```

Try reloading the fixtures now:

```
php app/console doctrine:fixtures:load
```

Ah, there's our error!

```
Unknown formatter "avatar"
```

Time to fix that! Open AppFixtures and create a new public function called avatar(). To keep things lazy, let's copy the guts of characterName() and update the options to be kitten1.jpg, then 2, 3 and 4. Sweet!

```
49 lines | src/AppBundle/DataFixtures/ORM/AppFixtures.php
```

```
... lines 1 - 7
8  class AppFixtures extends DataFixtureLoader
9  {
  ... lines 10 - 36
37  public function avatar()
38  {
39      $filenames = array(
40          'kitten1.jpg',
41          'kitten2.jpg',
42          'kitten3.jpg',
43          'kitten4.jpg',
44      );
45
46      return $filenames[array_rand($filenames)];
47  }
48  }
```

Reload reload! ... the fixtures:

```
php app/console doctrine:fixtures:load
```

Great, and now reload our page. Ah, broken images! Yay! The img tags are printing out beautifully, but there isn't *actually* a kitten3.jpg file inside the uploads/avatars directory. We've got work to do!

## Creating the Processor

This is where Processors come in. Whenever you need to do something *other* than just setting simple data, you'll use a Processor, which is like a hook that's called before and after each object is saved.

Step1! Create a new class. It doesn't matter where it goes, so put it inside ORM/ and call it AvatarProcessor. The only rule of a processor is that it needs to implement ProcessorInterface. And that means we have to have two methods: postProcess() and preProcess().

Each is passed whatever object is being saved right now, so let's just dump the class of the object:

```
29 lines | src/AppBundle/DataFixtures/ORM/AvatarProcessor.php
1  <?php
2
3  namespace AppBundle\DataFixtures\ORM;
4
5  use Nelmio\Alice\ProcessorInterface;
6
7  class AvatarProcessor implements ProcessorInterface
8  {
9      /**
10       * Processes an object before it is persisted to DB
11       *
12       * @param object $object instance to process
13       */
14     public function preProcess($object)
15     {
16         var_dump(get_class($object));
17     }
18
19     /**
20      * Processes an object before it is persisted to DB
21      *
22      * @param object $object instance to process
23      */
24     public function postProcess($object)
25     {
26         // TODO: Implement postProcess() method.
27     }
28 }
```

Cool new processor class, check! To hook it up, go back into AppFixtures. The parent DataFixturesLoader class has an empty getProcessors() method that we need to override. Because it's empty, we don't need to call the parent. Just return an array with a new AvatarProcessor object in it:

56 lines | [src/AppBundle/DataFixtures/ORM/AppFixtures.php](#)

... lines 1 - 7

```
8 class AppFixtures extends DataFixtureLoader
9 {
    ... lines 10 - 48
49     protected function getProcessors()
50     {
51         return array(
52             new AvatarProcessor()
53         );
54     }
55 }
```

Let's reload the fixtures to see what happens!

```
php app/console doctrine:fixtures:load
```

Cool! It calls preProcessor for every object - whether it's a Universe or a Character.

## Moving Images Around

Ok, let's copy some images. First, we only want to do work if the object that's passed to us is a Character. So, if we're *not* an instance of Character, just return:

46 lines | [src/AppBundle/DataFixtures/ORM/AvatarProcessor.php](#)

... lines 1 - 15

```
16     public function preProcess($object)
17     {
18         if (!$object instanceof Character) {
19             return;
20         }
    ... lines 21 - 33
34     }
    ... lines 35 - 46
```

Next, some Character's don't have an avatar, so if this doesn't have an avatarFilename, we'll just return - we don't need to move any files around:

46 lines | [src/AppBundle/DataFixtures/ORM/AvatarProcessor.php](#)

... lines 1 - 15

```
16     public function preProcess($object)
17     {
    ... lines 18 - 21
22         if (!$object->getAvatarFilename()) {
23             return;
24         }
    ... lines 25 - 33
34     }
    ... lines 35 - 46
```

Now we *know* there's an avatarFilename. We also know that the originals live in this resources/ directory, so we just need to copy those into the web/uploads/avatars directory.

First, create a variable that points to the root directory of our project. This will get me all the way back to the root - there are other ways to do this, but this is simple.

To do the copying, let's use Symfony's Filesystem object - it does nice things like create the directory if it doesn't exist. And hey, that's nice! My editor just added the use statement for me. Now, call copy(). The original file is \$projectRoot, resources,

then the `avatarFilename`. The destination is `$projectRoot` again, then to `web/uploads/avatars` then the object's `avatarFilename`:

```
46 lines | src/AppBundle/DataFixtures/ORM/AvatarProcessor.php
... lines 1 - 15
16     public function preProcess($object)
17     {
... lines 18 - 25
26         $projectRoot = __DIR__.'../../..';
27
28         $fs = new Filesystem();
29         $fs->copy(
30             $projectRoot.'/resources/'.$object->getAvatarFilename(),
31             $projectRoot.'/web/uploads/avatars/'.$object->getAvatarFilename(),
32             true
33         );
34     }
... lines 35 - 46
```

We're using this directory because that's what my app is expecting in the template. The third argument is whether to override an existing file. And that should get the job done! Reload those fixtures!

```
php app/console doctrine:fixtures:load
```

Now refresh! Ok, super-hero kittens! And if you want to know how to get access to the container in a Processor, keep watching.

## Chapter 3: Creating Unique Files

### Tip

A newer version of HauteLookAliceBundle has been released and portions of this tutorial won't apply to that new version.

There's a little issue. These two kittens are the *exact* same filename. The first is kitten2.jpg and so is the second. That's fine for us, but imagine if we could delete characters, and if doing that deleted the image. If we deleted *this* character, it would delete the image for the second one too. To be more realistic, each character needs a unique image.

NO problem. Setup a new \$targetFilename instead of using the original filename. Set it to fixtures\_ then mt\_rand() and .jpg:

```
49 lines | src/AppBundle/DataFixtures/ORM/AvatarProcessor.php
... lines 1 - 15
16     public function preProcess($object)
17     {
... lines 18 - 26
27         $targetFilename = 'fixtures_'.mt_rand(0, 100000).'.jpg';
... lines 28 - 36
37     }
... lines 38 - 49
```

Copy the file to *this* filename. And make sure that the avatarFilename is our new, random thing:

```
49 lines | src/AppBundle/DataFixtures/ORM/AvatarProcessor.php
... lines 1 - 15
16     public function preProcess($object)
17     {
... lines 18 - 26
27         $targetFilename = 'fixtures_'.mt_rand(0, 100000).'.jpg';
... lines 28 - 29
30         $fs->copy(
31             $projectRoot.'/resources/'.$object->getAvatarFilename(),
32             $projectRoot.'/web/uploads/avatars/'.$targetFilename,
33             true
34         );
35
36         $object->setAvatarFilename($targetFilename);
37     }
... lines 38 - 49
```

Time to reload those fixtures:

```
php app/console doctrine:fixtures:load
```

In web/uploads/avatars, we see a bunch of random filenames. And when we refresh, they're all using different filenames.

### [Accessing the container in a Processor](#)

You can do whatever you want inside a Processor, but with a glaring limitation so far: you don't have access to the container or any of your services.

Let's try to log the random filenames being used for each Character. That means we'll need the logger service, and right now we don't have access to anything. To get it, we'll treat AvatarProcessor like any other service and use dependency injection.

Create a `__construct()` function, and type-hint the argument with `LoggerInterface` from PSR. That'll add my use statement. Now, set that on a logger property:

```
63 lines | src/AppBundle/DataFixtures/ORM/AvatarProcessor.php
... lines 1 - 6
7   use Psr\Log\LoggerInterface;
... lines 8 - 9
10  class AvatarProcessor implements ProcessorInterface
11  {
12      private $logger;
13
14      public function __construct(LoggerInterface $logger)
15      {
16          $this->logger = $logger;
17      }
... lines 18 - 61
62 }
```

Before worrying about how we'll pass in the logger, go down below and log a debug message. Fill in the placeholders with the object's name, the `$targetFilename` and then the original `avatarFilename`:

```
63 lines | src/AppBundle/DataFixtures/ORM/AvatarProcessor.php
... lines 1 - 9
10  class AvatarProcessor implements ProcessorInterface
11  {
12      private $logger;
13
14      public function __construct(LoggerInterface $logger)
15      {
16          $this->logger = $logger;
17      }
... lines 18 - 23
24  public function preProcess($object)
25  {
... lines 26 - 43
44      $this->logger->debug(sprintf(
45          'Character %s using filename %s from %s',
46          $object->getName(),
47          $targetFilename,
48          $object->getAvatarFilename()
49      ));
50      $object->setAvatarFilename($targetFilename);
51  }
... lines 52 - 61
62 }
```

This class is *not* registered as a service - we just create it manually in AppFixtures:



```
56 lines | src/AppBundle/DataFixtures/ORM/AppFixtures.php
```

```
... lines 1 - 7
```

```
8 class AppFixtures extends DataFixtureLoader
9 {
  ... lines 10 - 48
49 protected function getProcessors()
50 {
51     return array(
52         new AvatarProcessor()
53     );
54 }
55 }
```

Passing the logger in is simple. The base `DataFixturesLoader` class *has* the container and puts it on a `$container` property, just like a Controller. So we can say `$this->container->get('logger')`:

```
56 lines | src/AppBundle/DataFixtures/ORM/AppFixtures.php
```

```
... lines 1 - 7
```

```
8 class AppFixtures extends DataFixtureLoader
9 {
  ... lines 10 - 48
49 protected function getProcessors()
50 {
51     return array(
52         new AvatarProcessor($this->container->get('logger'))
53     );
54 }
55 }
```

To test this out, open up a new tab and let's tail the `app/logs/dev.log` directory, because `app/console` runs in the dev environment by default. And let's grep it for the word `Character`:

```
tail -f app/logs/dev.log | grep "Character"
```

Now reload the fixtures!

```
php app/console doctrine:fixtures:load
```

No errors, AND we get our log messages. Btw, you can also see log messages directly when running a command by passing the `-vvv` option:

```
php app/console doctrine:fixtures:load -vvv
```

This can be pretty handy.

This means that there's *nothing* you can't do with a Processor. Need a service? Just use normal dependency injection, pass it in, do awesome things with your fixtures, then celebrate.

Cheers!

