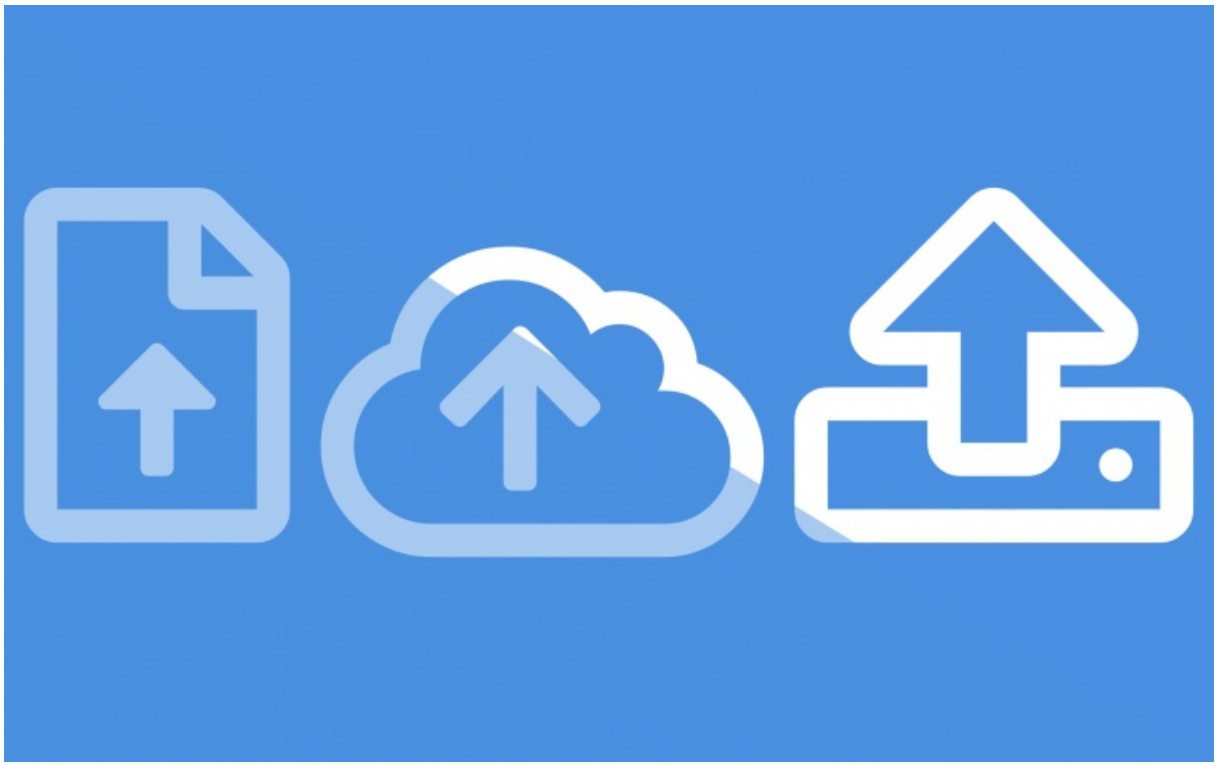
All about Uploading Files in Symfony



With <3 from SymfonyCasts

Chapter 1: Setting up with the Symfony Local Web Server

Yo friends! It's file upload time! Woo! We are going to absolutely *crush* this topic... yea know... because file uploads are a *critical* part of the Internet. Where would we be if we couldn't upload selfies... or videos of Victor's cat... or SPAM our friends with memes!?!?! That's not a world I want to live in.

But... is uploading a file really *that* hard: add a file input to a form, submit, move the file onto your filesystem and... done! Meme unlocked! Well... that's true... until you start thinking about storing files in the cloud, like S3. Oh, and don't forget to add validation to make sure a user can't upload *any* file type - like an executable or PHP script! And you'll need to make sure the filename is unique so it doesn't overwrite other files... but also... it's kind of nice to *keep* the original filename... so it's not just some random hash if the user downloads it later. Oh, and once it's uploaded, we'll need a way to link to that file... except if you need to do a security check before letting the user download the file. Then you'll need to handle things in a totally different way.

Um... so wow! Things got complex! That's awesome! Because we're going to attack *all* of this... and more.

Downloading the Course Code

If you want to upload the *maximum* knowledge into your brain... you should *definitely* download the course code from this page and code along with me. After unzipping the file, you'll find a start/ directory that has the same code you see here. Open the README.md file for all the setup details... and a few extras.

The *last* setup step in our tutorials is *usually* to open a terminal, move into the project and run:

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

to start the built in web server. You *can* totally do this. But, but, but! I want to show you a *new* tool that I'm loving: the Symfony local web server.

Downloading the Symfony Local Web Server

Find your browser and go to https://symfony.com/download. The Symfony local web server - or Symfony "client" - is a single, standalone file that is *full* of superpowers. At the top, you'll see instructions about how to download it. These steps are different depending on your operating system - but it should auto-select the right one.

For me, I'll copy this curl command, find my terminal, paste and enter! This downloaded a single executable file called symfony. To make sure I can type that command from anywhere, I'll move this into a global bin directory. By the way, you only need to do these steps *once* on your computer... so you're done forever!

Unless we've mucked things up, we should *now* be able to run this from anywhere: try it!

```
$ symfony
```

Say hello to the Symfony CLI! It lists the most popular commands, but there are a *lot* more - run:

```
$ symfony help
```

Woh. We'll talk more about this tool in another tutorial. But, to start a local web server, just say:

```
$ symfony serve
```

Ah. The *first* time you run this, you'll get an error about running: symfony server:ca:install. Let's do that:

```
$ symfony server:ca:install
```

You'll probably need to type in your admin password. This command installs a local SSL certificate authority... which is *awesome* because when we run symfony serve, it creates a local web server that supports https! Woh! We get *free* https locally! Sweet!

Find your browser and go to https://127.0.0.1:8000 - or localhost, it's the same thing. Say hello to The SpaceBar! This is the app we've been building in our Symfony 4 series: a news site for space-traveling friends from across the galaxy.

Try logging in with admin1@thespacebar.com and password engage. Then go to /admin/article.

This is the admin section for the articles on the site. Each article has an image... but until now, that image has basically been hardcoded. Click to edit one of the articles. Our first goal is clear: add a file upload field to this form so we can upload the article image, and then render that on the frontend.

But we're going to keep things simple to start... and take a deep and wonderful look into the fundamentals of how files are uploaded on the web and how that looks inside Symfony. Let's go!

Chapter 2: Uploads, multipart/form-data & UploadedFile

This page uses a Symfony form. And we *will* learn how to add a file upload field to a form object. But... let's start simpler - with a good old-fashioned HTML form.

The controller behind this page live at src/Controller/ArticleAdminController.php, and we're on the edit() action. Create a totally new, temporary endpoint: public function temporaryUploadAction(). We're going to create an HTML form in our template, put an input file field inside, and make it submit to this action. Add the @Route() with, how about, /admin/upload/test and name="upload_test". But... don't do anything else yet.

115 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 14
15 class ArticleAdminController extends BaseController
16 {
... lines 17 - 69
70     /**
71      * @Route("/admin/upload/test", name="upload_test")
72      */
73     public function temporaryUploadAction(Request $request)
74     {
... line 75
76     }
... lines 77 - 113
114 }
```

Copy the route name, then open the template for the edit page: templates/article_admin/edit.html.twig. The Symfony form lives inside the _form.html.twig template. So, *above* that form tag, add a new form tag, with method="POST" and action="" set to {{ path('upload_test') }}. Inside, we only need one thing <input type="file">. We need to give this a name so we can reference it on the server: how about name="image".

Finally, add <button type="submit"> and I'll add some classes so that this isn't the *ugliest* button ever. Say: Upload!

24 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 5
6     <form method="POST" action="{{ path('upload_test') }}">
7         <input type="file" name="image">
8
9         <button type="submit" class="btn btn-primary">Upload!</button>
10    </form>
11
12    <hr>
13
14    {{ include('article_admin/_form.html.twig', {
15        button_text: 'Update!'
16    }) }}
... lines 17 - 24
```

That's it! The simplest possible file upload setup: one field, one button.

Fetching the File in the Controller

In some ways, uploading a file is really no different than any other form field: you're always just sending data to the server where each data has a *key* equal to its name attribute. So, the same as any form, to read the submitted data, we'll need the request object. Add a new argument with a Request type-hint - the one from HttpFoundation - $request. Then say: dd() - that's dump & die - $request->files->get('image'). I'm using image because that's the name attribute used on the field.

```
115 lines | src/Controller/ArticleAdminController.php
... lines 1 - 10
11 use Symfony\Component\HttpFoundation\Request;
... lines 12 - 72
73     public function temporaryUploadAction(Request $request)
74     {
75         dd($request->files->get('image'));
76     }
... lines 77 - 115
```

Cool! What do you think this will dump out? A string filename? An array? An object? Let's find out! Choose a file - I'll go into my I <3 Space directory, and select the astronaut photo! Upload!

multipart/form-data

Oh! It's... null!? I did not see that coming. If you're ever uploading a file and it's *totally* not working, you've probably made the same mistake I just did. Go back to the template and add an attribute to the form enctype="multipart/form-data".

```
24 lines | templates/article_admin/edit.html.twig
... lines 1 - 5
6     <form method="POST" action="{{ path('upload_test') }}" enctype="multipart/form-data">
... lines 7 - 9
10    </form>
... lines 11 - 24
```

Yep! Mysteriously, you *never* need this on your forms... *until* you have a file upload field. It basically tells your browser to send the data in a different *format*. We're going to see *exactly* what this means soon cause we are *crushing* the magic behind uploads.

Fortunately, PHP understand this format *and* this format supports file uploads. Refresh the form so the new attribute is rendered. Let's choose the astronaut again. And before hitting Upload, open up your developer tools and go to the Network tab: I want to see what this request looks like. Hit upload!

Nice! This time we get an UploadedFile object *full* of useful data.

But before we dive into that, look down at the network tools and find the POST request we just made. If you look at the request headers... here it is: our browser sent a Content-Type: multipart/form-data header. *This* is because of the enctype attribute. It also added this weird boundary=----WebkitFormBoundary, blah, blah, blah thing.

Ok: this stuff is super-nerdy-cool. *Normally*, when you do *not* have that enctype attribute, when you submit a form, all of the data is sent in the body of the request in a big string full of what looks like query parameters. That's kind of invisible to us, because PHP parses all of that and makes the data available.

But when you add the multipart/form-data attribute, it tells our browser to send the data in a different format. It's actually kind of hard to see what the body of these requests look like - Chrome hides it. No worries! Through the magic of TV... boom! *This* is what the body of that request looks like.

Weird, right! Each field is separated by this mysterious WebkitFormBoundary thing... which is the string that we saw in the Content-Type header! Our form only has one field, but if we had multiple, this separator would be between *every* field. Our browsers invents this string, separates each piece of data with it, then sends this separator up with the request so that the server knows how to parse everything.

Why is this cool? Because we can now send up *multiple* pieces of information about our name="image" field, like the original filename on our system and what type of file it is... which, by the way, can be totally faked by the user. More on that later. After all that, we've got the data itself!

If you look *all* the way at the bottom, it has another WebKitFormBoundary line. If there were more fields on this form, you'd see their data below - all separated by another "boundary".

So... that's it! It literally tells our browser to send the data in a different format - and PHP understands *both* formats just fine. We *need* this format when doing file uploads because a file upload is *more* than just its contents: we also want to send some metadata. And also, due to how the data is encoded, if you *were* able to send binary data on a normal request - without the

multipart/form-data encoding - it would increase the amount of data you need to upload by as much as three times! Not great for uploads!

The UploadedFile Object

Once the data arrives at the server, PHP automatically reads in the file and saves it to a temporary location on your server. Symfony then takes *all* of these details and puts it into a nice, neat UploadedFile object. You can see the originalName: astronaut.jpeg, the mimeType and - *importantly* - the location on the filesystem where the file is temporarily stored.

If we do *nothing* with that file, PHP will automatically delete it at the end of the request. So... our job is clear! We need to move that into a final location and... do a bunch of other things, like make sure it has a unique filename and the correct file *extension*. Let's handle that next.

Chapter 3: Where & How to Store the File

For now, the form is still submitting to this test endpoint. We'll change that soon by moving it into the actual article form. But, to finish a successful file upload, we need *move* the uploaded file from the temporary spot on the filesystem to its final location.

Where to Store Uploads?

So... where *should* we store the uploaded article images? The *first* question to ask is: can these uploaded files be public to everyone? Or do we need to do some sort of security check before a user can view or download them? For article images, they can be public. But we'll talk about private files later.

Ok, so *if* someone needs to be able to view these images, it means they need to live *somewhere* in the public/ directory. Later, we're going to talk about storing files in the cloud! Like S3, which honestly, is an awesome idea. But right now, we're going to keep it simple and store things directly on our server.

So how about storing things in... I don't know... public/uploads? Create that new directory. Then, inside, create an *empty* .gitignore file. The *reason* I'm doing this might be confusing at first. My goal is to *ignore* any files added to this directory from git... because we don't want to commit uploaded files. But I would *also* like to make sure that this directory at least *exists* when I clone the repository.

Find your terminal and add the empty .gitignore file:

```
$ git add public/uploads/.gitignore
```

Next, open up the *real* .gitignore file - the one at the root of your app - and ignore the entire /public/uploads directory. It's a bit weird, but thanks to this, we will ignore *all* files in public/uploads *except* for the .gitignore file we already added.

```
20 lines | .gitignore
... lines 1 - 17
18
19  /public/uploads/
```

Why did we do this? Well, unfortunately, you can't add a *directory* to git. So by adding this .gitignore file, it will guarantee that the public/uploads directory will exist when you clone the repository. Honestly, the file could be named *anything*, it's just sort of a common practice to use an empty .gitignore file for this.

Check it out: create a new file in public/uploads called foo. Then, find your terminal and run:

```
$ git status
```

We see the new public/uploads/.gitignore file but we do *not* see the foo file. That's perfect. Delete that.

Moving the Uploaded File

Let's get to work inside of our controller to move the file. First, set the uploaded file to a new $uploadedFile variable. And, unfortunately, the phpdoc on this get() method is a bit generic... so it doesn't tell our editor that this will be an UploadedFile object. Because I'm *obsessed* with auto-completion, let's add inline doc about this: this *will* be an UploadedFile object - but not the one from Guzzle - the one from HttpFoundation in Symfony.

```
120 lines | src/Controller/ArticleAdminController.php
... lines 1 - 10
11 use Symfony\Component\HttpFoundation\File\UploadedFile;
... lines 12 - 15
16 class ArticleAdminController extends BaseController
17 {
... lines 18 - 73
74     public function temporaryUploadAction(Request $request)
75     {
76         /** @var UploadedFile $uploadedFile */
77         $uploadedFile = $request->files->get('image');
... lines 78 - 79
80     }
... lines 81 - 118
119 }
```

And guess what? This UploadedFile object has a *super* useful method on it: move()! Give *it* the destination directory and it'll take care of the rest. To get that directory, say $destination = and we need to get the path to our uploads/ directory. The best way is to read a parameter: $this->getParameter('kernel.project_dir') - to get the absolute path to the root of the app - then /public/uploads. Then add $uploadedFile->move() and pass it $destination.

Hold Command or Ctrl and click this method. Ah, it returns a File object that represents the new file. Let's see what this looks like: surround this entire call with dd().

```
120 lines | src/Controller/ArticleAdminController.php
... lines 1 - 73
74     public function temporaryUploadAction(Request $request)
75     {
... lines 76 - 77
78         $destination = $this->getParameter('kernel.project_dir').'/public/uploads';
79         dd($uploadedFile->move($destination));
80     }
... lines 81 - 120
```

Alright team! Find your browser, refresh and re-post that upload. I... think it worked! The dumped file object tells me that there *is* a new file in our public/uploads/ directory. Let's go check it out! There it is! Well, I *think* that's it... but sheesh - the filename is *terrible*. Let's check its file size:

```
$ ls -la public/uploads/
```

Yea... that looks correct - it's about 1.8 megabytes. So... we moved the file... but that is a *terrible* filename. Let's fix that next.

Chapter 4: Unique (but not Insane) Filenames

I told the UploadedFile object to move the file into public/uploads. And it *did*... but I kinda get the feeling it wasn't trying very hard. I mean, that is a *horrible* filename. Well, to be fair, this is the temporary filename that PHP decided to use.

Using the Original Filename

Fortunately, the move() method has a second argument: the *name* to give to the file. The *easiest* name to use is: $uploadedFile->getClientOriginalName(). This is the name that the file had on *my* computer: it's one of the pieces of data that is sent up on the request, along with the file contents.

```
123 lines | src/Controller/ArticleAdminController.php
... lines 1 - 15
16 class ArticleAdminController extends BaseController
17 {
... lines 18 - 73
74     public function temporaryUploadAction(Request $request)
75     {
... lines 76 - 78
79         dd($uploadedFile->move(
80             $destination,
81             $uploadedFile->getClientOriginalName()
82         ));
83     }
... lines 84 - 121
122 }
```

Move over and resubmit the form again. There it is: astronaut.jpg!

Security Concerns

But there are a few problems with this. Number one is security. Boo security! I know, I know, if the world were more butterflies and ice cream cones, we wouldn't need to worry about this stuff. But when it comes to file uploads, security concerns are *real*.

Right now, our upload form has *no* validation at *all*. So even though *we* are intending for this to be an image, the user could upload *anything*. And to make things worse, the file will then be publicly accessible. Someone could basically use our site as a private file storage, even storing viruses and trying to trick people into downloading it from our trusted domain. We'll talk about validation a bit later: it is *critical* that you do *not* allow your users to upload *any* file type.

Side note: no matter how you build your app or what safeguards you put it place, you should *always* make sure that your web server will *only* parse your main public/index.php file through PHP. If your server is configured to execute *any* file ending in .php through PHP, that is a *huge* security risk. Ok, back to butterflies and ice cream.

Even after we add validation to guarantee that the uploaded file is *actually* an image, the user could *still* successfully upload an image with a .exe or .php file extension! Even if we validate the file type, allowing fake extensions is weird... and could be risky.

So problem number one is security and we'll tackle part of it in a minute and the other part when we talk about validation.

Problem number two is that the filename is not guaranteed to be unique! If someone else uploads a file called astronaut.jpg, boom! My schweet photo is gone!

Making Filenames Unique

There are a few ways to handle the unique problem - but the easiest one is just to add some sort of unique id to the filename. Set $newFilename to uniqid(), a '-' then $uploadedFile->getClientOriginalName(). Below, use $newFilename.

```
125 lines | src/Controller/ArticleAdminController.php
... lines 1 - 73
    public function temporaryUploadAction(Request $request)
    {
... lines 76 - 78
        $newFilename = uniqid().'-'.$uploadedFile->getClientOriginalName();

        dd($uploadedFile->move(
            $destination,
            $newFilename
        ));
    }
... lines 86 - 125
```

Let's try that! Better. It's kind of an ugly hash on the beginning of the filename, but it *does* solve the unique problem. You can also use a shorter hash or, when we actually save this data to our Article object, you could use the Article id instead of the hash. *Or*, if you *really* want to keep the original filename *exactly* as it was, well... we'll talk about that later when we upload "references" to our Article.

Correcting the File Extension

The other thing I want to solve is the possibility that someone uploads an image with a totally insane file extension - like .potato. We can fix this really nicely. Create a new variable called $originalFilename set to pathinfo() with $uploadedFile->getClientOriginalName() and the constant PATHINFO_FILENAME.

This will give us the original filename - astronaut.jpg - but *without* the file extension: so, just astronaut. Then, for the filename, use $originalFilename, a dash, the uniqid(), a period, and now the *real* extension of the file: $uploadedFile->guessExtension(). Oh, see how there are *two* methods: ->guessClientExtension() and ->guessExtension()? The difference is important: the guessExtension() method looks at the file *contents*, determines the mime type, and returns the file extension for that. But the guessClientExtension() uses the mime type the *user* sent... which can't be trusted.

```
127 lines | src/Controller/ArticleAdminController.php
... lines 1 - 73
    public function temporaryUploadAction(Request $request)
    {
... lines 76 - 79
        $originalFilename = pathinfo($uploadedFile->getClientOriginalName(), PATHINFO_FILENAME);
        $newFilename = $originalFilename.'-'.uniqid().'.'.$uploadedFile->guessExtension();
... lines 82 - 86
    }
... lines 88 - 127
```

So, we're not validating that this is an image file yet, but no matter what they upload, we should now get the correct file extension.

Give it a try! Nice! We've got a .jpeg ending.

Optional: Normalizing Filenames

There's one last thing you might want to do... and it's really optional. Go back to the form. One of my files has uppercase letters and spaces inside. Let's try uploading that. It works! There is *no* problem with storing spaces or... *most* weird characters on a filesystem. But if you want to guarantee cleaner filenames, there's an easy way to do that. I'll use a class called Urlizer: this comes from the gedmo/doctrine-extensions library. It has a nice method called urlize() and we can wrap our $originalFilename in that to make it a bit cleaner.

128 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 8
9  use Gedmo\Sluggable\Util\Urlizer;
... lines 10 - 74
75     public function temporaryUploadAction(Request $request)
76     {
... lines 77 - 81
82         $newFilename = Urlizer::urlize($originalFilename).'-'.uniqid().'.'.$uploadedFile->guessExtension();
... lines 83 - 87
88     }
... lines 89 - 128
```

Try that out. Nice! So now we have a unique, normalized filename that at least *looks* a bit like the original filename. Later, we'll see how we can keep the *exact* original filename in *all* cases... if you care. But unless your users are downloading these files, the exact filenames aren't usually that important.

Next: it's time to put this upload field *properly* into our Symfony form and save the filename to the Article entity.

Chapter 5: File Upload Field in a Form

We're rocking! We know what it looks like to upload a file in Symfony: we can work with the UploadedFile object and we know how to move the file around. That feels good!

It's time to talk about how to work with a file upload field inside a Symfony form. And then, we *also* need to save the *filename* of the uploaded image to our Article entity. Because, ultimately, on the homepage, we need to render each image next to the article.

What your Entity Should Look Like

In the src/Entity directory, let's look at the Article entity. Ok great: the entity is *already* setup! It has an $imageFilename field that is a *string*. This is important: the uploaded file will be stored... *somewhere*: on your server, in the cloud, in your imagination - it doesn't matter. But in the database, the *only* thing you will store is the *string* filename.

Adding the FileType to the Form

The form that handles this page lives at src/Form/ArticleFormType.php. In ArticleAdminController... if you scroll up a little bit... here is the edit() action and you can see it using this ArticleFormType. Right now, this is a nice traditional form: it handles the request and saves the Article to the database. Beautifully... boring!

In ArticleFormType, add a new field with ->add() and call it imageFilename because that's the name of the property inside Article. For the type, use FileType::class.

154 lines | src/Form/ArticleFormType.php

```
... lines 1 - 11
12 use Symfony\Component\Form\Extension\Core\Type\FileType;
... lines 13 - 19
20 class ArticleFormType extends AbstractType
21 {
... lines 22 - 28
29     public function buildForm(FormBuilderInterface $builder, array $options)
30     {
... lines 31 - 34
35         $builder
... lines 36 - 53
54             ->add('imageFilename', FileType::class)
55         ;
... lines 56 - 88
89     }
... lines 90 - 152
153 }
```

But... there's a problem with this. And if you already see it, extra credit points for you! Move over and refresh. Woh.

> The form's view data is expected to be an instance of class File but it is a string.

Um... ok. The problem is not super obvious... but it clearly hates *something* about our new field. Here's the explanation: *we* know that when you upload a file, Symfony gives you an UploadedFile *object*, *not* a string. But, the imageFilename field here on Article... that *is* a string! Connecting the form field *directly* to the string property doesn't make sense. We're missing a layer in the middle: something that can work with the UploadedFile object, move the file, and *then* set the new filename onto the property.

Using an Unmapped Field

How can we do that? Change the field name to just imageFile. There is *no* property on our entity with this name... so this, on its own, will *not* work. Pretty commonly, you'll see people *create* this property on their entity, *just* to make the form work. They

don't persist this property to the database with Doctrine... so the idea *works*, but I don't love it.

Instead, we'll use a trick that we talked a lot about in our forms tutorial: add an option to the field: 'mapped' => false.

156 lines | src/Form/ArticleFormType.php

```
... lines 1 - 19
20 class ArticleFormType extends AbstractType
21 {
... lines 22 - 28
29     public function buildForm(FormBuilderInterface $builder, array $options)
30     {
... lines 31 - 34
35         $builder
... lines 36 - 53
54             ->add('imageFile', FileType::class, [
55                 'mapped' => false
56             ])
57         ;
... lines 58 - 90
91     }
... lines 92 - 154
155 }
```

If you've never seen this before, we'll explain it in a minute. Now that we have a new imageFile field, let's go render it! Open edit.html.twig. Remove the HTML form - we're done with that. The Symfony form lives in _form.html.twig. After the title, add {{ form_row(articleForm.imageFile }}.

24 lines | templates/article_admin/_form.html.twig

```
1 {{ form_start(articleForm) }}
2     {{ form_row(articleForm.title, {
3         label: 'Article title'
4     }) }}
5     {{ form_row(articleForm.imageFile) }}
... lines 6 - 23
24 {{ form_end(articleForm) }}
```

Nothing special here.

This submits back to ArticleAdminController::edit(). Go inside the $form->isValid() block. When you have an unmapped field, the data will *not* be put onto your Article object. So, how can we get it? dd($form['imageFile']->getData()).

130 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 16
17 class ArticleAdminController extends BaseController
18 {
... lines 19 - 48
49     public function edit(Article $article, Request $request, EntityManagerInterface $em)
50     {
... lines 51 - 55
56         if ($form->isSubmitted() && $form->isValid()) {
57             dd($form['imageFile']->getData());
... lines 58 - 66
67         }
... lines 68 - 71
72     }
... lines 73 - 128
129 }
```

Let's try that! Go back to your browser and hit enter on the URL: we need the form to totally re-render. Hey! There's our new field! Select the astronaut again. Um... did that work? Cause... I don't see the filename on my field. Yes: it *did* work - we don't see anything because of a display bug if you're using Symfony's Bootstrap 4 form theme. We'll talk about that later. But, the file *is* attached to the field. Hit Update!

Yes! It's our beloved UploadedFile object! We *totally* know how to work with that! Oh, but before we do: I want to point out something cool. Inspect element and find the form tag. Hey! It has the enctype="multipart/form-data" attribute! We get that for free because we use the {{ form_start() }} function to render the <form> tag. As *soon* as there is even *one* file upload field in the form, Symfony adds this attribute for you. High-five team!

Moving the Uploaded File

Time to finish this. Let's upload a different file - earth.jpeg. And... there's the dump. We have two jobs in our controller: move this file to the final location *and* store the new filename on the $imageFilename property. Back in the controller, scroll down to temporaryUploadAction(), steal all its code, and delete it.

Up in edit(), remove the dd() and set this to an $uploadedFile variable. Add the same inline phpdoc as last time

123 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 16
17 class ArticleAdminController extends BaseController
18 {
... lines 19 - 48
49     public function edit(Article $article, Request $request, EntityManagerInterface $em)
50     {
... lines 51 - 55
56         if ($form->isSubmitted() && $form->isValid()) {
57             /** @var UploadedFile $uploadedFile */
58             $uploadedFile = $form['imageFile']->getData();
59             $destination = $this->getParameter('kernel.project_dir').'/public/uploads';
... lines 60 - 77
78         }
... lines 79 - 82
83     }
... lines 84 - 121
122 }
```

then paste the code. Yep! We'll move the file to public/uploads and give it a unique filename. Take off the dd() around move().

123 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 16
17 class ArticleAdminController extends BaseController
18 {
... lines 19 - 48
49     public function edit(Article $article, Request $request, EntityManagerInterface $em)
50     {
... lines 51 - 55
56         if ($form->isSubmitted() && $form->isValid()) {
57             /** @var UploadedFile $uploadedFile */
58             $uploadedFile = $form['imageFile']->getData();
59             $destination = $this->getParameter('kernel.project_dir').'/public/uploads';
60
61             $originalFilename = pathinfo($uploadedFile->getClientOriginalName(), PATHINFO_FILENAME);
62             $newFilename = Urlizer::urlize($originalFilename).'-'.uniqid().'.'.$uploadedFile->guessExtension();
63
64             $uploadedFile->move(
65                 $destination,
66                 $newFilename
67             );
... lines 68 - 77
78         }
... lines 79 - 82
83     }
... lines 84 - 121
122 }
```

Now, call $article->setImageFilename($newFilename)

```
123 lines | src/Controller/ArticleAdminController.php
... lines 1 - 16
17 class ArticleAdminController extends BaseController
18 {
... lines 19 - 48
49     public function edit(Article $article, Request $request, EntityManagerInterface $em)
50     {
... lines 51 - 55
56         if ($form->isSubmitted() && $form->isValid()) {
57             /** @var UploadedFile $uploadedFile */
58             $uploadedFile = $form['imageFile']->getData();
59             $destination = $this->getParameter('kernel.project_dir').'/public/uploads';
60
61             $originalFilename = pathinfo($uploadedFile->getClientOriginalName(), PATHINFO_FILENAME);
62             $newFilename = Urlizer::urlize($originalFilename).'-'.uniqid().'.'.$uploadedFile->guessExtension();
63
64             $uploadedFile->move(
65                 $destination,
66                 $newFilename
67             );
68             $article->setImageFilename($newFilename);
... lines 69 - 77
78         }
... lines 79 - 82
83     }
... lines 84 - 121
122 }
```

and let Doctrine save the entity, *just* like it already was.

Beautiful! I *do* want to point out that the $newFilename string that we're storing in the database is *just* the filename: it doesn't contain the directory or the word uploads: it's... the filename. Oh, for my personal sanity, let's upload things into an article_image sub-directory: that'll be cleaner when we start uploading multiple types of things. Remove the old files.

```
123 lines | src/Controller/ArticleAdminController.php
... lines 1 - 16
17 class ArticleAdminController extends BaseController
18 {
... lines 19 - 48
49     public function edit(Article $article, Request $request, EntityManagerInterface $em)
50     {
... lines 51 - 55
56         if ($form->isSubmitted() && $form->isValid()) {
57             /** @var UploadedFile $uploadedFile */
58             $uploadedFile = $form['imageFile']->getData();
59             $destination = $this->getParameter('kernel.project_dir').'/public/uploads/article_image';
... lines 60 - 77
78         }
... lines 79 - 82
83     }
... lines 84 - 121
122 }
```

Moment of truth! Find your browser, roll up your sleeves, and refresh! Um... it *probably* worked? In the uploads/ directory... yea! There's our Earth file! Let's see what the database looks like - find your terminal and run:

```
$ php bin/console doctrine:query:sql 'SELECT * FROM article WHERE id = 1'
```

Let's see, the id of this article is 1. Yes! the image_filename column is *totally* set! Fist-pumping time!

Avoid Processing when no Upload

Oh, but there is one tiny thing we need to clean up before moving on. What if we just want to, I don't know, edit the article's title, but we don't need to change the image. No problem - hit Update! Oh... That's HTML5 validation. You might remember from the forms tutorial that this required attribute is added to *every* field... unless you're using form field type guessing. It's annoying - fix it by adding 'required' => false.

157 lines | src/Form/ArticleFormType.php

```
... lines 1 - 19
20 class ArticleFormType extends AbstractType
21 {
... lines 22 - 28
29     public function buildForm(FormBuilderInterface $builder, array $options)
30     {
... lines 31 - 34
35         $builder
... lines 36 - 53
54             ->add('imageFile', FileType::class, [
... line 55
56                 'required' => false,
57             ])
58         ;
... lines 59 - 91
92     }
... lines 93 - 155
156 }
```

Let's try it again. Refresh, change the title, submit and... oof.

> Call to a member function getClientOriginalName on null

Of course! We're not uploading a file! So the $uploadedFile variable is null! That's ok! If the user didn't upload a file, we don't need to do *any* of this logic. In other words, if ($uploadedFile), then do all of that. Otherwise, skip it!

125 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 16
17 class ArticleAdminController extends BaseController
18 {
... lines 19 - 48
49     public function edit(Article $article, Request $request, EntityManagerInterface $em)
50     {
... lines 51 - 55
56         if ($form->isSubmitted() && $form->isValid()) {
57             /** @var UploadedFile $uploadedFile */
58             $uploadedFile = $form['imageFile']->getData();
59             if ($uploadedFile) {
60                 $destination = $this->getParameter('kernel.project_dir').'/public/uploads/article_image';
61
62                 $originalFilename = pathinfo($uploadedFile->getClientOriginalName(), PATHINFO_FILENAME);
63                 $newFilename = Urlizer::urlize($originalFilename).'-'.uniqid().'.'.$uploadedFile->guessExtension();
64
65                 $uploadedFile->move(
66                     $destination,
67                     $newFilename
68                 );
69                 $article->setImageFilename($newFilename);
70             }
... lines 71 - 79
80         }
... lines 81 - 84
85     }
... lines 86 - 123
124 }
```

Refresh now. Got it!

Next: This is looking good! Except that... we need this *exact* same logic in the new() action. To make a *truly* killer upload system, we need to refactor the upload logic into a reusable service.

Chapter 6: Centralizing Upload Logic

We've got a pretty nice system so far: moving the file, unique filenames and putting the filename string into the database. But it *is* kind of a lot of logic to put in the controller... and we *already* need to reuse this code somewhere else: in the new() action.

Creating the Service

That's why I like to isolate my upload logic into a service class. In the Service/ directory - or really anywhere - create a new class: how about UploaderHelper?

23 lines | src/Service/UploaderHelper.php

```
1  <?php
2
3  namespace App\Service;
   ... lines 4 - 6
7
8  class UploaderHelper
9  {
   ... lines 10 - 21
22 }
```

This class will handle *all* things related to uploading files. Create a public function uploadArticleImage(): it will take the UploadedFile as an argument - remember the one from HttpFoundation - and return a string. That will be the string filename that was ultimately saved.

23 lines | src/Service/UploaderHelper.php

```
   ... lines 1 - 5
6  use Symfony\Component\HttpFoundation\File\UploadedFile;
   ... line 7
8  class UploaderHelper
9  {
10     public function uploadArticleImage(UploadedFile $uploadedFile): string
11     {
   ... lines 12 - 20
21     }
22 }
```

Ok! Let's go steal some code for this. In fact, we're going to steal pretty much all the logic here... and paste it in. Make sure to retype the r on Urlizer to get the use statement on top.

```
23 lines | src/Service/UploaderHelper.php
... lines 1 - 4
5  use Gedmo\Sluggable\Util\Urlizer;
... lines 6 - 7
8  class UploaderHelper
9  {
10     public function uploadArticleImage(UploadedFile $uploadedFile): string
11     {
12         $destination = $this->getParameter('kernel.project_dir').'/public/uploads/article_image';
13
14         $originalFilename = pathinfo($uploadedFile->getClientOriginalName(), PATHINFO_FILENAME);
15         $newFilename = Urlizer::urlize($originalFilename).'-'.uniqid().'.'.$uploadedFile->guessExtension();
16
17         $uploadedFile->move(
18             $destination,
19             $newFilename
20         );
21     }
22 }
```

And at the bottom, return $newFilename.

```
32 lines | src/Service/UploaderHelper.php
... lines 1 - 7
8  class UploaderHelper
9  {
... lines 10 - 16
17     public function uploadArticleImage(UploadedFile $uploadedFile): string
18     {
... lines 19 - 28
29         return $newFilename;
30     }
31 }
```

Perfect! Well... not *perfect*, because the $this->getParameter() method is a shortcut that only works in the controller. If you need a parameter - or *any* configuration - from inside a service, you need to add it via dependency injection. Add the public function __construct() with, how about, a string $uploadsPath argument. Instead of just injecting the kernel.project_dir parameter, we'll pass in the *whole* string to where uploads should be stored.

```
32 lines | src/Service/UploaderHelper.php
... lines 1 - 7
8  class UploaderHelper
9  {
10     private $uploadsPath;
11
12     public function __construct(string $uploadsPath)
13     {
14         $this->uploadsPath = $uploadsPath;
15     }
... lines 16 - 30
31 }
```

I'll put my cursor on that argument name, hit Alt + Enter and select initialize fields to create that property and set it. Now, below, we can say $this->uploadsPath and then /article_image.

```
32 lines | src/Service/UploaderHelper.php
... lines 1 - 7
8  class UploaderHelper
9  {
... lines 10 - 16
17     public function uploadArticleImage(UploadedFile $uploadedFile): string
18     {
19         $destination = $this->uploadsPath.'/article_image';
20
... lines 21 - 29
30     }
31 }
```

Cool! Let's worry about configuring the $uploadsPath argument to our service in a minute. After all, Symfony's service system is *so* awesome, it'll tell me *exactly* what I need to configure once we try this.

For now, go back into ArticleAdminController and use this. Start by adding another argument: UploaderHelper $uploaderHelper. And celebrate by removing *all* of the logic below and replacing it with $newFilename = $uploaderHelper->uploadArticleImage($uploadedFile).

```
118 lines | src/Controller/ArticleAdminController.php
... lines 1 - 7
8  use App\Service\UploaderHelper;
... lines 9 - 17
18 class ArticleAdminController extends BaseController
19 {
... lines 20 - 49
50     public function edit(Article $article, Request $request, EntityManagerInterface $em, UploaderHelper $uploaderHelper)
51     {
... lines 52 - 56
57         if ($form->isSubmitted() && $form->isValid()) {
... lines 58 - 59
60             if ($uploadedFile) {
61                 $newFilename = $uploaderHelper->uploadArticleImage($uploadedFile);
62                 $article->setImageFilename($newFilename);
63             }
... lines 64 - 72
73         }
... lines 74 - 77
78     }
... lines 79 - 116
117 }
```

Dang - that is nice! There is still a *little* bit of logic here: the form logic and the logic that sets the filename on the Article - but I'm comfortable with that. And we now have this great new method: pass it an UploadedFile object, and it'll move it into the correct directory and give it a unique filename.

Binding the $uploadsPath Argument

Let's take it for a test drive! Go back, refresh the form and... it works! Naw, I'm kidding - we knew this error was coming... but isn't it beautiful?

> Cannot resolve argument $uploadHelper of the edit() method: Cannot autowire service UploadHelper: argument $uploadsPath of method __construct() is type-hinted string, you should configure its value explicitly.

That's programming poetry people! And it makes sense: autowiring doesn't work for scalar arguments. We got this: open config/services.yaml. We *could* configure the *specific* argument for this *specific* service. But if you've watched our Symfony series, you know that *I* like to use the bind feature. The argument name is $uploadsPath. So, below _defaults and bind, add $uploadsPath set to %kernel.project_dir%/public/uploads.

```
47 lines | config/services.yaml
... lines 1 - 9
services:
... line 11
    _defaults:
... lines 13 - 19
        bind:
... lines 21 - 22
            $uploadsPath: '%kernel.project_dir%/public/uploads'
... lines 24 - 47
```

This means: *anywhere* that $uploadsPath is used as an argument for a method that's autowired - usually a controller action or the constructor of a service - pass in this value.

Exceeding upload_max_filesize

Let's go see if that fixed things - reload. *Now* we see the form. To test this fully, let's empty out the article_image/ directory. This time, let's upload the stars photo. Hit update.

Woh! The file "empty string" does not exist!? What the heck! Let's do some digging. When we call guessExtension(), internally, Symfony looks at the contents of the temporary uploaded file to determine what's inside. But... that file is missing! In fact, PHP is telling us that the temporary file name is... an empty string! It's madness!

Why is this happening? I'll give you a clue: the file we just uploaded is 3mb. Go to your terminal and run

```
$ php -i | grep upload
```

There it is: the upload_max_filesize in my php.ini is *2* megabytes, which is PHP's default value. I have a *bunch* of things to say about this. First, make sure you set this value to whatever you *really* want your max to be. You may also need to bump the post_max_size setting - that defaults to 8 mb, and *also* will cause uploads to fail if they're bigger than this.

Second, if you're getting *super* weird results while uploading, this is probably the problem. And *third*, once we add validation to our upload field, we'll get a really nice validation error instead of this crazy fatal error. Symfony has our back.

So let's try a smaller file - our astronaut - it's 1.9 mb. Hit update and... yes! It worked!

Adding the Logic to new() Action

Now that all of our logic is isolated, we can easily repeat this in the new() action. We *do* need to copy these 5 lines or so, but I'm happy with that.

Up in new(), add the argument - UploaderHelper $uploaderHelper - and inside the isValid() block, paste!

126 lines | src/Controller/ArticleAdminController.php

```
... lines 1 - 17
18 class ArticleAdminController extends BaseController
19 {
... lines 20 - 23
24     public function new(EntityManagerInterface $em, Request $request, UploaderHelper $uploaderHelper)
25     {
... lines 26 - 28
29         if ($form->isSubmitted() && $form->isValid()) {
30             /** @var Article $article */
31             $article = $form->getData();
32
33             /** @var UploadedFile $uploadedFile */
34             $uploadedFile = $form['imageFile']->getData();
35
36             if ($uploadedFile) {
37                 $newFilename = $uploaderHelper->uploadArticleImage($uploadedFile);
38                 $article->setImageFilename($newFilename);
39             }
... lines 40 - 46
47         }
... lines 48 - 51
52     }
... lines 53 - 124
125 }
```

This uses the same form, with the same unmapped field, so it'll all just work.

Next: let's talk about validation.

Chapter 7: File Validation

I've ignored it long enough - sorry! We've *gotta* add some validation to the upload field. Because... right now, we can upload *any* file type - it's madness! This is supposed to be an image field people! We need to *only* allow pngs, jpegs, gifs, image stuff.

Validating an Unmapped Field

Normally we add validation to the entity class: we would go into the Article class, find the property, and add some annotations. But... the field we want to validate is an *unmapped* form field - there *is* no imageFile property in Article.

No worries: for unmapped fields, you can add validation directly to the form with the constraints option. And when it comes to file uploads, there are two really important constraints: one called File and an even stronger one called Image. Add new Image() - the one from the Validator\Constraints.

```
161 lines | src/Form/ArticleFormType.php
... lines 1 - 18
19  use Symfony\Component\Validator\Constraints\Image;
20
21  class ArticleFormType extends AbstractType
22  {
... lines 23 - 29
30      public function buildForm(FormBuilderInterface $builder, array $options)
31      {
... lines 32 - 35
36          $builder
... lines 37 - 54
55              ->add('imageFile', FileType::class, [
... lines 56 - 57
58                  'constraints' => [
59                      new Image()
60                  ]
61              ])
62          ;
... lines 63 - 95
96      }
... lines 97 - 159
160 }
```

The Image Constraint

And... that's all we need! That's enough to make sure the user uploads an image. Check it out: find your browser, Google for "Symfony image constraint" and click into the docs.

The Image constraint *extends* the File constraint - so both basically have the same behavior: you can define a maxSize or configure different mimeTypes. The Image constraint just adds... more super-powers. First, it pre-configures the mimeType option to only allow images. And you get a crazy-amount of other image stuff - like minWidth, maxWidth or allowPortrait.

So let's test it! Refresh the page and browse. Oh, the Symfony Best Practices PDF snuck into my directory. Select that, update and... boom! This file is not a valid image.

Validating the File Size

Go back to the docs and click to see the File constraint. The other most common option is maxSize. To see what that looks like, set it to something *tiny*, like 5k.

```
163 lines | src/Form/ArticleFormType.php
... lines 1 - 20
21 class ArticleFormType extends AbstractType
22 {
... lines 23 - 29
30     public function buildForm(FormBuilderInterface $builder, array $options)
31     {
... lines 32 - 35
36         $builder
... lines 37 - 54
55             ->add('imageFile', FileType::class, [
... lines 56 - 57
58                 'constraints' => [
59                     new Image([
60                         'maxSize' => '5k'
61                     ])
62                 ]
63             ])
64         ;
... lines 65 - 97
98     }
... lines 99 - 161
162 }
```

Ok: browse and select *any* of the files. Hit update and... perfect: the file is too large.

Change that back to 5M, or whatever makes sense for you.

```
163 lines | src/Form/ArticleFormType.php
... lines 1 - 20
21 class ArticleFormType extends AbstractType
22 {
... lines 23 - 29
30     public function buildForm(FormBuilderInterface $builder, array $options)
31     {
... lines 32 - 35
36         $builder
... lines 37 - 54
55             ->add('imageFile', FileType::class, [
... lines 56 - 57
58                 'constraints' => [
59                     new Image([
60                         'maxSize' => '5M'
61                     ])
62                 ]
63             ])
64         ;
... lines 65 - 97
98     }
... lines 99 - 161
162 }
```

Validation and upload_max_filesize

Oh, but, remember a few minutes ago when we tried to upload the stars photo? It's 3 megabytes, which is way under the 5 megabytes we just set, but *above* my php.ini upload_max_filesize setting. That caused a really *nasty* error.

Well, try selecting it again and updating. Yes! When you use the File or Image constraint, they *also* catch any PHP-level upload errors and display them quite nicely. You *can* customize this message.

Making the Upload Field Required

And... that's it! Sure, there are a more options and you can control all the messages - but that's easy enough. Except... there *is* one tricky thing: how can we make the upload field required? Like, when someone *creates* an article, they should be required to upload an image before saving it.

Simple, right? Just add a new NotNull() constraint to the imageFile field. Wait, no, that won't work. If we did that, we would need to upload a file even if we were just editing a field on the article: we would literally need to upload an image *every* time we changed anything.

Okay: so we want the imageFile to be required... but *only* if the Article doesn't already have an imageFilename. Start by breaking this onto multiple lines. Then say $imageConstraints =, copy the new Image() stuff and paste it here.

167 lines | src/Form/ArticleFormType.php

```
... lines 1 - 20
21 class ArticleFormType extends AbstractType
22 {
... lines 23 - 29
30     public function buildForm(FormBuilderInterface $builder, array $options)
31     {
... lines 32 - 35
36         $builder
... lines 37 - 45
46             ->add('location', ChoiceType::class, [
... lines 47 - 53
54             ])
55         ;
... line 56
57         $imageConstraints = [
58             new Image([
59                 'maxSize' => '5M'
60             ])
61         ];
62         $builder
63             ->add('imageFile', FileType::class, [
... lines 64 - 66
67             ])
68         ;
... lines 69 - 101
102     }
... lines 103 - 165
166 }
```

Down below, set 'constraints' => $imageConstraints. Oh... and let's spell that correctly.

```
167 lines | src/Form/ArticleFormType.php
... lines 1 - 20
21 class ArticleFormType extends AbstractType
22 {
... lines 23 - 29
30     public function buildForm(FormBuilderInterface $builder, array $options)
31     {
... lines 32 - 61
62         $builder
63             ->add('imageFile', FileType::class, [
... lines 64 - 65
66                 'constraints' => $imageConstraints
67             ])
68         ;
... lines 69 - 101
102     }
... lines 103 - 165
166 }
```

Now we can conditionally add the NotNull() constraint *exactly* when we need it. Scroll up a little. In our forms tutorial, we used the data option to get the Article object that this form is bound to. If this is a "new" form, there may or may not be an Article object - so this will be an Article object or null. I also used that to create an $isEdit variable to figure out if we're on the edit screen or not.

We can leverage that by saying if this is *not* the edit page or if the article doesn't have an image filename, then take $imageConstraints and add new NotNull(). We'll even get fancy and customize the message: Please upload an image.

```
175 lines | src/Form/ArticleFormType.php
... lines 1 - 19
20 use Symfony\Component\Validator\Constraints\NotNull;
21
22 class ArticleFormType extends AbstractType
23 {
... lines 24 - 57
58         $imageConstraints = [
59             new Image([
60                 'maxSize' => '5M'
61             ])
62         ];
63
64         if (!$isEdit || !$article->getImageFilename()) {
65             $imageConstraints[] = new NotNull([
66                 'message' => 'Please upload an image',
67             ]);
68         }
... lines 69 - 109
110     }
... lines 111 - 173
174 }
```

Just saying if !$isEdit is probably enough... but *just* in case, I'm checking to see if, *somehow*, we're on the edit page, but the imageFilename is missing, let's require it.

Cool: testing time! Refresh the entire form, but don't select an upload: we know that this Article *does* have an image already attached. Hit update and... works fine! Now try creating a new Article, fill in a few of the required fields, hit create and... boom! Please upload an image!

Validation, check! Next, let's fix how this renders: we've *gotta* see the filename after selecting a file - seeing nothing is

bummin' me out.

Chapter 8: Upload Field Styling & Bootstrap

If you use the Bootstrap 4 theme with Symfony... things get weird with upload fields! Yea, there *is* a good reason for *why*, but out-of-the-box, it's... just super weird. The problem? Select a file and... get rewarded by seeing absolutely *nothing*! Did the file actually attach? We *should* see the filename somewhere. What happened?

Why Doesn't it Work?

The thing is... styling a file upload field is kinda hard. So, if you *really* want to control how it looks and make it super shiny, Bootstrap allows you to create a "custom" file input structure, which is what Symfony uses by default. Check this out: see the <input type="file"...> field? That's *hidden* by Bootstrap! Try removing the opacity: 0 part and... say hello to the *real* file upload field... *with* the filename that we selected!

Bootstrap hides the input so that it, or *we*, can *completely* control how this *whole* field looks. Everything you *actually* see comes from the label: it takes up the entire width. Even the "Browse" button comes from some :after content.

The *great* thing about this is that styling a label element is easy. The sad panda part is that we don't see the filename when we select a file! We *can* fix that - but it takes a little bit of JavaScript.

Customizing the Text in the Upload Field

Before we do that, we can *also* put a message in the main part of the file field by putting some content in the label element. But... it doesn't work like a normal label.

In the templates/ directory, open article_admin/_form.html.twig. Here's our imageFile field. The second argument to form_row is an array of variables you can use to customize... basically anything. One of the most important ones is called attr: it's how you attach custom HTML attributes to the input field. Pass an attribute called placeholder set to Select an article image.

28 lines | templates/article_admin/_form.html.twig

```
1  {{ form_start(articleForm) }}
   ... lines 2 - 4
5      {{ form_row(articleForm.imageFile, {
6          attr: {
7              'placeholder': 'Select an article image'
8          }
9      }) }}
   ... lines 10 - 27
28 {{ form_end(articleForm) }}
```

This would normally add a placeholder attribute to the input so you can have some text on the field if it's empty. But when you're dealing with a file upload field with the Bootstrap theme, this is used in a different way... but it accomplishes the same thing.

Refresh! Cool! The empty part of the file field now gets this text.

Showing the Selected Filename

But if you select a file... the filename still doesn't show. Let's fix that already. Look at the structure again: Symfony's form theme is using this custom-file-input class on the input. Ok, so what we need to do is this: on *change* of that field, we need to set the HTML of the label to the filename, which *is* something we have access to in JavaScript.

To keep things simple, open base.html.twig: we'll write some JavaScript that will work across the entire site. I'd recommend using Webpack Encore, and putting this code in your main entry file if you want it to be global. But, without Encore, down here works fine.

Use $('.custom-file-input') - that's the class that's on the input field itself, .on('change') and pass this a callback with an event argument. Inside, we need to find the label element: I'll do that by finding the parent of the input and then looking for the custom-file-label class so we can set its HTML.

```
99 lines | templates/base.html.twig
... line 1
2  <html lang="en">
... lines 3 - 15
16     <body>
... lines 17 - 82
83         {% block javascripts %}
... lines 84 - 86
87             <script>
... line 88
89                 $('.custom-file-input').on('change', function(event) {
... lines 90 - 93
94                 });
95             </script>
96         {% endblock %}
97     </body>
98 </html>
```

In the callback, set var inputFile = event.currentTarget - that's the DOM node for the input type="file" element. Next, $(inputFile).parent().find('.custom-file-label').html() and pass this the filename that was just selected: inputFile.files[0].name. The 0 part looks a bit weird, but technically a file upload field can upload *multiple* files. We're not doing that, so we get to take this shortcut.

```
99 lines | templates/base.html.twig
... line 1
2  <html lang="en">
... lines 3 - 15
16     <body>
... lines 17 - 82
83         {% block javascripts %}
... lines 84 - 86
87             <script>
... line 88
89                 $('.custom-file-input').on('change', function(event) {
90                     var inputFile = event.currentTarget;
91                     $(inputFile).parent()
92                         .find('.custom-file-label')
93                         .html(inputFile.files[0].name);
94                 });
95             </script>
96         {% endblock %}
97     </body>
98 </html>
```

Give it a try! Refresh... browse... select rocket.jpg and... yea! Our placeholder gets replaced by the filename. That's what we expect *and* the field is easier to style thanks to this.

Next: the upload side of things is looking good. It's time to start rendering the URL to the upload files... but without letting things get crazy-disorganized. I want to *love* our setup.

Chapter 9: URL to Public Assets

The hardest part of handling uploads... probably isn't the uploading part! For me, it's rendering the URLs to the uploaded files, thumbnailing and creating endpoints to download *private* files. Oh, and we *gotta* keep this organized: I do *not* want a bunch of upload directory names sprinkled over 50 files in my code. It's bad for sanity, I mean, *maintenance*, and will make it hard to move your uploads to the cloud later... which we *are* going to do.

Look back at the homepage: all of these images work except for one. But, *this* is actually the image that we uploaded! Inspect element on that and check its path: /images/astronaut-blah-blah.jpeg. Check out one of the working images. Ah yes: until now, in the fixtures, we set the $imageFilename string to one of the filenames that are hardcoded and committed into the public/images/ directory, like asteroid.jpeg.

These aren't *really* uploaded assets: we were just faking it! Check out the template: templates/article/homepage.html.twig. There it is! We're using the asset()... ah, wrong spot. Here we go: we're saying {{ asset(article.imagePath) }}, which calls getImagePath() inside Article. That just prefixes the filename with images/ and returns it! So if imageFilename is asteroid.jpeg in the database, this returns images/asteroid.jpeg.

Pointing the Path to uploads/

Now that the *true* uploaded assets are stored in a different directory, we can just update this path! In Article, change this to uploads/article_image/ and then $this->getImageFilename().

309 lines | src/Entity/Article.php

```
... lines 1 - 17
18 class Article
19 {
... lines 20 - 184
185     public function getImagePath()
186     {
187         return 'uploads/article_image/'.$this->getImageFilename();
188     }
... lines 189 - 307
308 }
```

Cool! Try it out! It works! We don't care about the broken images from the fixtures: we'll fix them soon. But the *actual* uploaded image *does* render.

Getting Organized

Great first step. Now, let's get organized! One problem is that we have the directory name - article_image - in Article and *also* in UploaderHelper where we move the file around. That's not too bad - but as we start adding more file uploads to the system, we're going to have more duplication. I don't like having these important strings in multiple places.

So, in UploaderHelper, why not create a constant for this? Call it ARTICLE_IMAGE and set it to the directory name: article_image.

34 lines | src/Service/UploaderHelper.php

```
... lines 1 - 7
8 class UploaderHelper
9 {
10     const ARTICLE_IMAGE = 'article_image';
11
... lines 12 - 32
33 }
```

Down below, use that: self::ARTICLE_IMAGE.

```
34 lines | src/Service/UploaderHelper.php
... lines 1 - 7
8  class UploaderHelper
9  {
... lines 10 - 18
19     public function uploadArticleImage(UploadedFile $uploadedFile): string
20     {
21         $destination = $this->uploadsPath.'/'.self::ARTICLE_IMAGE;
22
... lines 23 - 31
32     }
33 }
```

And in Article, do the same thing: UploaderHelper::ARTICLE_IMAGE.

```
310 lines | src/Entity/Article.php
... lines 1 - 5
6  use App\Service\UploaderHelper;
... lines 7 - 18
19 class Article
20 {
... lines 21 - 185
186    public function getImagePath()
187    {
188        return 'uploads/'.UploaderHelper::ARTICLE_IMAGE.'/'.$this->getImageFilename();
189    }
... lines 190 - 308
309 }
```

Small step, and when we refresh, it works fine.

Centralizing the Public Path

Let's keep going! Back in Article, the path starts with uploads... because that's part of the public path to the asset. That's not a huge problem, but I actually *don't* want that uploads string to live here. Why? Well, I kinda don't want my entity to really care *where* or *how* we're storing our uploads. Like, if our site grows and we move our uploads to the cloud, we would need to change this uploads string to a full CDN URL in *all* entities with an upload field. And, that URL might even need to be dynamic - we might use a different CDN locally versus on production! Nope, I don't want my entity to worry about any of these details.

Remove the uploads/ part from the path.

```
310 lines | src/Entity/Article.php
... lines 1 - 18
19 class Article
20 {
... lines 21 - 185
186    public function getImagePath()
187    {
188        return UploaderHelper::ARTICLE_IMAGE.'/'.$this->getImageFilename();
189    }
... lines 190 - 308
309 }
```

Now getImagePath() returns the path to the image relative to wherever our *app* decides to store uploads. In UploaderHelper, add a new public function getPublicPath(). This will take a string $path - that will be something like article_image/astronaut.jpeg - and it will return a string, which will be the *actual* public path to the file. Inside, return 'uploads/'.$path;.

```
39 lines | src/Service/UploaderHelper.php
... lines 1 - 7
8  class UploaderHelper
9  {
... lines 10 - 33
34     public function getPublicPath(string $path): string
35     {
36         return 'uploads/'.$path;
37     }
38 }
```

That may feel like a micro improvement, but it's awesome! Thanks to this, we can call getPublicPath() from anywhere in our app to get the URL to an uploaded asset. If we move to the cloud, we only need to change the URL here! Awesome!

uploaded_asset() Twig Extension

Except... how can we call this from Twig? Because, if we refresh right now... it definitely does *not* work. No worries: let's create a custom Twig function. Open src/Twig/AppExtension - this is the Twig extension we created in our Symfony series. Here's the plan: in the homepage template, instead of using the asset() function, let's use a new function called uploaded_asset(). We'll pass it article.imagePath - and it will ultimately call getPublicPath().

```
65 lines | templates/article/homepage.html.twig
... lines 1 - 20
21                 {% for article in articles %}
... lines 22 - 23
24                     <img class="article-img" src="{{ uploaded_asset(article.imagePath) }}">
... lines 25 - 39
40                 {% endfor %}
... lines 41 - 65
```

In AppExtension, copy getFilters(), paste and rename it to getFunctions(). Return an array, and, inside, add a new TwigFunction() with uploaded_asset and [$this, 'getUploadedAssetPath'].

```
58 lines | src/Twig/AppExtension.php
... lines 1 - 10
11 use Twig\TwigFunction;
... line 12
13 class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
14 {
... lines 15 - 21
22     public function getFunctions(): array
23     {
24         return [
25             new TwigFunction('uploaded_asset', [$this, 'getUploadedAssetPath'])
26         ];
27     }
... lines 28 - 56
57 }
```

Copy that new method name, scroll down and add it: public function getUploadedAssetPath() with a string $path argument. It will also return a string.

```
58 lines | src/Twig/AppExtension.php
... lines 1 - 12
13 class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
14 {
... lines 15 - 42
43     public function getUploadedAssetPath(string $path): string
44     {
... lines 45 - 47
48     }
... lines 49 - 56
57 }
```

Using a Service Subscriber

Inside: we need to get the UploaderHelper service so we can call getPublicPath() on it. Normally we do this by adding it as an argument to the constructor. But, in a few places in Symfony, for performance purposes, we should do something *slightly* different: we use what's called a "service subscriber", because it allows us to fetch the services lazily. If this is a new concept for you, go check out our Symfony Fundamentals course - it's a really cool feature.

The short explanation is that this class has a getSubscribedServices() method where we can choose which services we need. These are then included in the $container object and we can fetch them out by saying $this->container->get().

Add UploaderHelper::class to the array.

```
58 lines | src/Twig/AppExtension.php
... lines 1 - 5
6 use App\Service\UploaderHelper;
... lines 7 - 12
13 class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
14 {
... lines 15 - 49
50     public static function getSubscribedServices()
51     {
52         return [
... line 53
54             UploaderHelper::class,
55         ];
56     }
57 }
```

Then, above, we can return $this->container->get(UploaderHelper::class)->getPublicPath($path).

```
58 lines | src/Twig/AppExtension.php
... lines 1 - 12
13 class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
14 {
... lines 15 - 42
43     public function getUploadedAssetPath(string $path): string
44     {
45         return $this->container
46             ->get(UploaderHelper::class)
47             ->getPublicPath($path);
48     }
... lines 49 - 56
57 }
```

Let's give it a try! Refresh! We got it! That took some work, but I promise you'll be *super* happy you did this.

Next: let's also update the image path in the show page, and learn a bit about what the asset() function does internally and how we can do the same thing automatically in UploaderHelper.

Chapter 10: The asset() Function & assets.context

When we go to the show page... of course, it doesn't work yet! We need to update the template. Copy the uploaded_asset() code, open show.html.twig... here it is, and paste.

86 lines | templates/article/show.html.twig

```
... lines 1 - 4
5  {% block content_body %}
6      <div class="row">
7          <div class="col-sm-12">
8              <img class="show-article-img" src="{{ uploaded_asset(article.imagePath) }}">
... lines 9 - 25
26         </div>
27     </div>
... lines 28 - 78
79 {% endblock %}
... lines 80 - 86
```

Easy! Reload the page now. Oh... it *still* doesn't work. Inspect element on the image. Ah, the path is right, but because there is no / at the beginning, and because the current URL is a sort of sub-directory, it's looking for the image in the wrong place. If you hack in the /... it pops up!

Adding this opening slash is actually one of the jobs of the asset() function. Try this: wrap this entire thing in asset().

86 lines | templates/article/show.html.twig

```
... lines 1 - 4
5  {% block content_body %}
6      <div class="row">
7          <div class="col-sm-12">
8              <img class="show-article-img" src="{{ asset(uploaded_asset(article.imagePath)) }}">
... lines 9 - 25
26         </div>
27     </div>
... lines 28 - 78
79 {% endblock %}
... lines 80 - 86
```

Now refresh. It works! But, wrapping asset() around uploaded_asset() is kind of annoying: can't we just handle this internally in UploaderHelper?

86 lines | templates/article/show.html.twig

```
... lines 1 - 4
5  {% block content_body %}
6      <div class="row">
7          <div class="col-sm-12">
8              <img class="show-article-img" src="{{ uploaded_asset(article.imagePath) }}">
... lines 9 - 25
26         </div>
27     </div>
... lines 28 - 78
79 {% endblock %}
... lines 80 - 86
```

After all, this method is supposed to return the public path to an asset: we shouldn't need to do any other "fixes" on the path after.

The easiest way to fix things would be to add a / at the beginning. That would totally work! But... allow me to nerd-out for a minute and explain an edge-case that the asset() function usually handles for us. Imagine if your site were deployed under a *subdirectory* of a domain. Like, instead of the URL on production being thespacebar.com, it's thegalaxy.org/thespacebar - our app does *not* live at the root of the domain. If you have a situation like this, hardcoding a / at the beginning of the URL won't work! It would need to be /thespacebar/.

The asset() function does this automatically: it detects that subdirectory and... just handles it! To *really* make our getPublicPath() shine, I want to do the same thing here.

Using the RequestStackContext

To do this, we're going to work with a service that you don't see very often in Symfony: it's the service that's used internally by the asset() function to determine the subdirectory. In the constructor, add another argument: RequestStackContext $requestStackContext. I'll hit Alt + Enter and select initialize fields to create that property and set it.

45 lines | src/Service/UploaderHelper.php

```
... lines 1 - 5
6  use Symfony\Component\Asset\Context\RequestStackContext;
... lines 7 - 8
9  class UploaderHelper
10 {
... lines 11 - 16
17     public function __construct(string $uploadsPath, RequestStackContext $requestStackContext)
18     {
... line 19
20         $this->requestStackContext = $requestStackContext;
21     }
... lines 22 - 43
44 }
```

Down in getPublicPath(), return $this->requestStackContext->getBasePath() and *then* '/uploads/'.$path.

45 lines | src/Service/UploaderHelper.php

```
... lines 1 - 8
9  class UploaderHelper
10 {
... lines 11 - 37
38     public function getPublicPath(string $path): string
39     {
40         // needed if you deploy under a subdirectory
41         return $this->requestStackContext
42             ->getBasePath().'/uploads/'.$path;
43     }
44 }
```

If our app lives at the root of the domain - like it does right now - this will just return and empty string. But if it lives at a subdirectory like thespacebar, it'll return /thespacebar.

Try it! Oh... wow - *huge* error! This RequestStackContext service is such a low-level service, that Symfony doesn't make it available to be used for autowiring. Check out the error, it says:

> Yo! You can't autowire the $requestStackContext argument: it's type-hinted with a class called RequestStackContext, but there isn't a service with this id. Maybe you can create a service alias for this class that points to the assets.context service.

This is a bit technical and we talk about this in our Symfony Fundamentals course. Symfony sees that the RequestStackContext type-hint is not autowireable, but it *also* sees that there *is* a service in the container - called

assets.context - that is an *instance* of this class!

Check it out: copy the full class name and then go into config/services.yaml. At the bottom, paste the full class name, go copy the service id they suggested, and say @assets.context.

```
48 lines | config/services.yaml
    ... lines 1 - 9
10  services:
    ... lines 11 - 47
48      Symfony\Component\Asset\Context\RequestStackContext: '@assets.context'
```

This creates a service alias. Basically, there is *now* a new service that lives in the container called Symfony\Component\Asset\Context\RequestStackContext. And if you fetch it, it'll really just give you the assets.context service. The *key* thing is that *this* makes the class autowireable.

To prove it, find your terminal and run:

```
$ php bin/console debug:autowiring request
```

to search for all autowireable classes that contain that string. Hey! There is our RequestStackContext! If we had run this a minute ago, it would *not* have been there.

Refresh the page now. Got it! And if you look at the path, yep! It's /uploads/article_image/astronaut.jpeg. If we lived under a subdirectory, that subdirectory would be there. Small detail, but our site is *still* super portable.

Next, let's create thumbnails of our image so the user doesn't need to download the full size.

Chapter 11: Thumbnailing with LiipImagineBundle

Go back to the homepage. We're rendering these images with a width and height of 100. But the image behind this is *way* bigger! That's wasteful: we don't want the user to wait to download these *gigantic* images, just to see the tiny thumbnail.

Hello LiipImagineBundle

Google for LiipImagineBundle and find its GitHub page. They have a bunch of docs right here... but *most* of the information actually lives over on Symfony.com. Click "Download the Bundle" to get there... and then I'll go back to the homepage - *lots* of good stuff here.

Start back on the Installation page. Copy the composer require line, find your terminal, paste and... go go go!

```
$ composer require liip/imagine-bundle
```

While we're waiting, head back over to the docs. Thanks to Flex, we don't need to enable the bundle or register the routes - that's automatic. Go back to the homepage of the docs... and click the "Filter Sets" link.

This bundle is pretty sweet. You start by creating something called a "filter set" and giving it a name - like my_thumb or whatever you want. Next, you tell the bundle which *filters*, or *transformations*, to apply when you use the my_thumb filter set. And there are a *ton* of them: you can change the size with the thumbnail filter, add a background, add border color, replace the image entirely with a cat gif - pretty much anything you can dream of. We'll just use the thumbnail transformation, but seriously - check out the full list.

Configuring the Filter Set

Let's go check on the install. Excellent! It's done. And the message is right on: it says we need to get to work in the new config file: liip_imagine.yaml. Go open that: config/packages/liip_imagine.yaml. Uncomment the root key to activate the bundle, leave the driver alone - it defaults to gd - and uncomment filter_sets.

42 lines | config/packages/liip_imagine.yaml

```
1  liip_imagine:
2  #    # valid drivers options include "gd" or "gmagick" or "imagick"
3  #    driver: "gd"
4  #
5  #    # define your filter sets under this option
6      filter_sets:
   ... lines 7 - 42
```

Let's create our first filter set called squared_thumbnail_small. We'll use this on the homepage to reduce the images down to 100 by 100. To do that, uncomment the filters key and I'll copy the thumbnail example from below, move it up here, and uncomment it.

42 lines | config/packages/liip_imagine.yaml

```
1   liip_imagine:
    ... lines 2 - 5
6       filter_sets:
    ... lines 7 - 9
10          squared_thumbnail_small:
11              filters:
    ... lines 12 - 42
```

Set the size to 200 by 200 so it looks good on Retina displays. The mode: outbound is *how* the thumbnail is applied - you can also use inbound.

```
42 lines | config/packages/liip_imagine.yaml
liip_imagine:
... lines 2 - 5
    filter_sets:
... lines 7 - 9
        squared_thumbnail_small:
            filters:
                thumbnail:
                    size:        [200, 200]
                    mode:        outbound
                    allow_upscale: true

... lines 17 - 42
```

And... I think we're ready to go! Copy the squared_thumbnail_small name and go into homepage.html.twig. To use this, it's so nice: |imagine_filter() and then the name.

```
65 lines | templates/article/homepage.html.twig
... lines 1 - 2
{% block body %}
... lines 4 - 20
            {% for article in articles %}
            <div class="article-container my-1">
                <a href="{{ path('article_show', {slug: article.slug}) }}">
                    <img class="article-img" src="{{ uploaded_asset(article.imagePath)|imagine_filter('squared_thumbnail_small') }}">
... lines 25 - 37
                </a>
            </div>
            {% endfor %}
... lines 41 - 63
{% endblock %}
```

The Thumbnailing Process

Let's go try it! Watch the image src closely. Refresh! It includes the https://127.0.0.1 part, but that's not important. The path - /media/cache/resolve/squared_thumbnail_small/... blah, blah blah - looks like a path to a physical file, but it's not! This is actually a Symfony route and it's handled by a Symfony controller!

Check it out: at your terminal, run:

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

There it is! The first time we refresh, LiipImagineBundle generates this URL. When our browser tries to download the image, it's handled by a controller from the bundle. That controller opens the original image, applies all the filters - just a thumbnail in our case - and returns the transformed image. That's a *slow* operation: our browser has to wait for all of that to finish.

But, watch what happens when we refresh. Did you see it? The path changed! It *was* /media/cache/resolve - but the resolve part is now gone! This time, the image is *not* handled by a Symfony route. Look at your public/ directory: there is now a media/ directory with cache/squared_thumbnail_small/uploads/article_image/astronaut-...jpeg.

The full process looks like this. The first time we refreshed, LiipImagineBundle noticed that no thumbnail file existed yet. So, it created the URL that pointed to the Symfony route & controller. The page finished rendering, and our browser make a second request to that URL to load the image. That request was handled by the controller from the bundle which thumbnailed the image, *saved* it to the filesystem, and returned it to the user. That's slow.

But when we reloaded the page the *second* time, LiipImagineBundle *noticed* that the filename already existed and generated a URL directly to that *real* file. The request for *that* image was *super* fast.

Oh, also check out the .gitignore file. Thanks to the Flex recipe, we're already ignoring the public/media directory: we do not want to commit this stuff: it'll just regenerate if it's missing.

So, yea - it all kinda works perfectly!

Next, let's add another filter set for the show page *and* add an image preview to the article form.

Chapter 12: Image Preview on the Form

Let's render a thumbnail on the show page too. The size here is restricted to a width of 250. Copy the first filter, paste, and call this one, how about, squared_thumbnail_medium. Set the size to 500 by 500.

```
49 lines | config/packages/liip_imagine.yaml
1  liip_imagine:
   ... lines 2 - 5
6      filter_sets:
   ... lines 7 - 16
17         squared_thumbnail_medium:
18             filters:
19                 thumbnail:
20                     size:        [500, 500]
21                     mode:        outbound
22                     allow_upscale: true
   ... lines 23 - 49
```

Copy the name and this time go into show.html.twig. Add the |imagine_filter() and paste!

```
86 lines | templates/article/show.html.twig
   ... lines 1 - 4
5  {% block content_body %}
6      <div class="row">
7          <div class="col-sm-12">
8              <img class="show-article-img" src="{{ uploaded_asset(article.imagePath)|imagine_filter('squared_thumbnail_medium') }}">
   ... lines 9 - 25
26         </div>
27     </div>
   ... lines 28 - 78
79 {% endblock %}
   ... lines 80 - 86
```

Reload! It works! The first time it has the resolve in the URL and is handled by a Symfony route & controller. The second time, it points directly to the file that was just saved. Awesome!

Adding an Image Preview to the Form

While we're kicking butt, go back to the article admin section and click to edit the article we've been working on. Hmm, it's not obvious that this article has an image attached... or what it looks like. We need a little image thumbnail next to this field.

We got this. Open the form template templates/article_admin/_form.html.twig. Let's think: to render an image, we *could* create a form theme that automatically makes the form_row() function render an image preview for file fields. That's cool. *Or*, we can keep it simple and do it right here.

Create a and another inside to set up a mini grid. Move the file field here. Now add a div with class="col-sm-3": *this* is where we'll render the image... if there is one.

```
39 lines | templates/article_admin/_form.html.twig
1  {{ form_start(articleForm) }}
   ... lines 2 - 5
6      <div class="row">
7          <div class="col-sm-9">
8              {{ form_row(articleForm.imageFile, {
9                  attr: {
10                     'placeholder': 'Select an article image'
11                 }
12             }) }}
13         </div>
14         <div class="col-sm-3">
   ... lines 15 - 17
18         </div>
19     </div>
   ... lines 20 - 38
39 {{ form_end(articleForm) }}
```

To do that, we need the Article object. Copy the image path logic from the homepage and then go find the controller for the admin section: ArticleAdminController. When we render the template - this is in the new() action - we're *only* passing the form variable. In edit(), we're doing the same thing. We *could* add an article variable here - that's a *fine* option. But, we don't *need* to.

Back in the template, we can say {% if articleForm.vars.data %} - *that* will be the Article object - then .imageFilename. If we have an image filename, print <img src="{{ }}"> and paste. Replace article with articleForm.vars.data. And yes, I *should* add an alt attribute - please do that! Set the height to 100, because the actual thumbnail is 200 for quality reasons.

```
39 lines | templates/article_admin/_form.html.twig
1  {{ form_start(articleForm) }}
   ... lines 2 - 5
6      <div class="row">
   ... lines 7 - 13
14         <div class="col-sm-3">
15             {% if articleForm.vars.data.imageFilename %}
16                 <img src="{{ uploaded_asset(articleForm.vars.data.imagePath)|imagine_filter('squared_thumbnail_small') }}" height="100">
17             {% endif %}
18         </div>
19     </div>
   ... lines 20 - 38
39 {{ form_end(articleForm) }}
```

Try it! Refresh and... yes! To make sure we didn't break anything, try creating a new article. Whoops... we broke something!

> Impossible to access attribute imageFilename on a null variable

Ah, we need to be careful: articleForm.vars.data *may* be null on a "new" form - it depends how you set it up. The easiest fix is to add |default. It's kinda weird... when you add |default, it *suppresses* the error and just returns null if there were any problems, which, for the if statement, is the same as false. It looks weird, but works great. Try it. All better.

40 lines | templates/article_admin/_form.html.twig

```
1  {{ form_start(articleForm) }}
   ... lines 2 - 5
6      <div class="row">
   ... lines 7 - 13
14         <div class="col-sm-3">
15             {% if articleForm.vars.data.imageFilename|default %}
16                 <img src="{{ uploaded_asset(articleForm.vars.data.imagePath)|imagine_filter('squared_thumbnail_small') }}" height="100">
17             {% endif %}
18         </div>
19     </div>
   ... lines 20 - 38
39 {{ form_end(articleForm) }}
```

Next, we have a real upload system (yay!) but our article data fixtures are broken: they're just setting imageFilename to a random filename that won't *actually* exist in the uploads/ directory. How can we fix that? By using our file upload system *inside* the fixtures! Well, at least, sort of.

Chapter 13: File Uploads & Data Fixtures

Open up src/DataFixtures/ArticleFixtures.php. Here's how this works: this function creates 10 articles whenever we run bin/console doctrine:fixtures:load. It's a cool helper we created in our Symfony series. But, the setImageFilename() stuff is now a problem. We know that the image filename needs to be the name of a file that lives inside of the uploads/article_image directory - something like astronaut-blah-blah.jpg. Right now, the fixtures use faker to select a random item in $articleImages - this private property. So, it's setting imageFilename to either asteroid.jpeg, mercury.jpeg or lightspeed.png.

This worked before because those images are committed to our repository in the public/images directory and we were pointing to *that* path in our template. When we run doctrine:fixtures:load, it *does* create 10 Article objects and it *does* set the image filename to one of these three filenames. But on the homepage... it doesn't work! There is no upload/article_image/lightspeed.png file. We need to re-think how this works.

Faking the File Upload

How? By *faking* the file upload inside the fixtures. It's kinda...beautiful! Our UploaderHelper service is already really good at moving things into the right spot - why not reuse it here?

Inside ArticleFixtures, create a public function __construct(). Add an UploaderHelper $uploaderHelper argument and I'll hit ALT + Enter and select initialize fields to create that property and set it.

92 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 7
8  use App\Service\UploaderHelper;
... lines 9 - 12
13 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
14 {
... lines 15 - 26
27     private $uploaderHelper;
28
29     public function __construct(UploaderHelper $uploaderHelper)
30     {
31         $this->uploaderHelper = $uploaderHelper;
32     }
... lines 33 - 90
91 }
```

Next, lets "cut" the 3 files in the public/images directory: we're going to move them to a different spot, because they no longer need to be publicly accessible. You'll see what I mean. In the src/DataFixtures directory, create a new folder here called images/ and paste them! Yep! They are no longer in the public/images/ directory.

Because these test images *are* committed to git, I'm going to commit this move - it'll help us in a minute when things... ah... sorta go wrong horribly wrong. Yes! We are planning for disaster!

Here's the idea: we'll use the UploaderHelper down here, point it at one of these 3 files, and have it, sort of, "fake" upload it. Start with $randomImage =, copy the faker code, and paste. This is now one of the three random image filenames.

```
92 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 12
13 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
14 {
... lines 15 - 33
34     protected function loadData(ObjectManager $manager)
35     {
36         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 37 - 63
64             $randomImage = $this->faker->randomElement(self::$articleImages);
... lines 65 - 78
79         });
... lines 80 - 81
82     }
... lines 83 - 90
91 }
```

Next, in UploaderHelper, what I'd *like* to do is call uploadArticleImage() and basically say:

> Hey! *Pretend* like asteroid.jpeg is a file that was just uploaded. And... ya know... do all your normal stuff and move it into the uploads/ directory.

This is easier than you think: in the fixtures class, set $imageFilename to $this->uploaderHelper->uploadArticleImage(). What I *want* to do is now say new UploadedFile() and point it at one of the images. The *problem* is that you can't really create a fake UploadedFile object. Internally, it's *bound* to the PHP uploading process - weird stuff will happen if you try to create one *outside* of that context.

Hello File Object

That's ok! It just means we need to dig deeper! Go back into UploaderHelper. Hold Command or Ctrl and click to open the UploadedFile class. This lives in the Symfony\HttpFoundation\File namespace and *extends* a class called File that lives in the same directory.

The File class is awesome: it simply represents... *any* file on your filesystem, regardless of whether it's an uploaded file or just a normal file. And, if you look closely, the *vast* majority of the methods we've been using come from *this* class - *not* from UploadedFile. And we *can* create a File object outside of an upload context.

So back in ArticleFixtures, instead of creating a new UploadedFile(), say new File() - the one from HttpFoundation. Pass this the path to the random image: __DIR__.'/images/' and then $randomImage, which will be one of these image filenames.

92 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 10
11 use Symfony\Component\HttpFoundation\File\File;
... line 12
13 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
14 {
... lines 15 - 33
34     protected function loadData(ObjectManager $manager)
35     {
36         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 37 - 63
64             $randomImage = $this->faker->randomElement(self::$articleImages);
65             $imageFilename = $this->uploaderHelper
66                 ->uploadArticleImage(new File(__DIR__.'/images/'.$randomImage));
... lines 67 - 78
79         });
... lines 80 - 81
82     }
... lines 83 - 90
91 }
```

Now, take $imageFilename - that'll be whatever the final filename is on the system after moving it, and set that onto the entity.

That's beautiful! In UploaderHelper, we need to make this work *not* with an UploadedFile object, but with the parent File. Change the type-hint to File - again, make sure you get the one from HttpFoundation or you will have *no* fun. To keep things clear, I'll Refactor -> Rename this variable to $file.

50 lines | src/Service/UploaderHelper.php

```
... lines 1 - 5
6 use Symfony\Component\HttpFoundation\File\File;
... lines 7 - 9
10 class UploaderHelper
11 {
... lines 12 - 23
24     public function uploadArticleImage(File $file): string
25     {
... lines 26 - 34
35         $file->move(
36             $destination,
37             $newFilename
38         );
... lines 39 - 40
41     }
... lines 42 - 48
49 }
```

Let's see: everything looks happy, ah - except for getClientOriginalName(): that method does not exist in File - it only exists in UploadedFile. Ok, let's get fancy then: if $file is an instanceof UploadedFile, we can say $originalFilename = $file->getClientOriginalName(). Else, set $originalFilename to $file->getFilename() - that's just the name of the file on the filesytem.

```
50 lines | src/Service/UploaderHelper.php
... lines 1 - 5
6  use Symfony\Component\HttpFoundation\File\File;
... lines 7 - 9
10 class UploaderHelper
11 {
... lines 12 - 23
24     public function uploadArticleImage(File $file): string
25     {
... lines 26 - 27
28         if ($file instanceof UploadedFile) {
29             $originalFilename = $file->getClientOriginalName();
30         } else {
31             $originalFilename = $file->getFilename();
32         }
... lines 33 - 40
41     }
... lines 42 - 48
49 }
```

After this, delete the pathinfo() stuff - we can move that to the next line. Inside urlize(), re-add the pathinfo() and pass the same second argument: PATHINFO_FILENAME.

```
50 lines | src/Service/UploaderHelper.php
... lines 1 - 5
6  use Symfony\Component\HttpFoundation\File\File;
... lines 7 - 9
10 class UploaderHelper
11 {
... lines 12 - 23
24     public function uploadArticleImage(File $file): string
25     {
... lines 26 - 27
28         if ($file instanceof UploadedFile) {
29             $originalFilename = $file->getClientOriginalName();
30         } else {
31             $originalFilename = $file->getFilename();
32         }
33         $newFilename = Urlizer::urlize(pathinfo($originalFilename, PATHINFO_FILENAME)).'-'.uniqid().'.'.$file->guessExtension();
... lines 34 - 40
41     }
... lines 42 - 48
49 }
```

I think that's all we need! Let's completely clear out the uploads/ directory. Now, find your terminal and run:

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

Copying the Files Before Moving

Woh! The file src/DataFixtures/images/asteroid.jpeg does not exist? Hmm. Check this out: it *did* upload two files before going all "explody" on us. Oh, but those original files are missing! Of course! We're using $file->move(). So it *is* working, but instead of copying the files, it's moving them, and the originals are disappearing.

Let's get those files back. Run:

```
$ git status
```

And undelete them with:

```
$ git checkout src/DataFixtures/images
```

Much better. Let's clean out the uploads directory again.

We *do* want to use $file->move() because we *do* want to move the uploaded file in normal circumstances. So, to get around this, in the fixtures, let's copy the original file to a temporary spot. Start with $fs = new Filesystem() - that's a handy object for doing filesystem operations.

96 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 10
11 use Symfony\Component\Filesystem\Filesystem;
... lines 12 - 13
14 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
15 {
... lines 16 - 34
35     protected function loadData(ObjectManager $manager)
36     {
37         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 38 - 64
65             $randomImage = $this->faker->randomElement(self::$articleImages);
66             $fs = new Filesystem();
... lines 67 - 82
83         });
... lines 84 - 85
86     }
... lines 87 - 94
95 }
```

Next, $targetPath = sys_get_temp_dir().'/'.$randomImage. And then use $fs->copy(). We want to copy the original file path into $targetPath.

```
96 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 10
11 use Symfony\Component\Filesystem\Filesystem;
... lines 12 - 13
14 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
15 {
... lines 16 - 34
35     protected function loadData(ObjectManager $manager)
36     {
37         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 38 - 65
66             $fs = new Filesystem();
67             $targetPath = sys_get_temp_dir().'/'.$randomImage;
68             $fs->copy(__DIR__.'/images/'.$randomImage, $targetPath, true);
... lines 69 - 82
83         });
... lines 84 - 85
86     }
... lines 87 - 94
95 }
```

Inside File, pass the temporary path.

```
96 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 10
11 use Symfony\Component\Filesystem\Filesystem;
... lines 12 - 13
14 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
15 {
... lines 16 - 34
35     protected function loadData(ObjectManager $manager)
36     {
37         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 38 - 65
66             $fs = new Filesystem();
67             $targetPath = sys_get_temp_dir().'/'.$randomImage;
68             $fs->copy(__DIR__.'/images/'.$randomImage, $targetPath, true);
69             $imageFilename = $this->uploaderHelper
70                 ->uploadArticleImage(new File($targetPath));
... lines 71 - 82
83         });
... lines 84 - 85
86     }
... lines 87 - 94
95 }
```

Ok, let's try it again!

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

No error, our original files still exist and... we have a directory full of, fake uploaded files. *Now* try the homepage. Beautiful. What I *really* love about this is that we're not doing anything fancy or tricky in our fixtures: we're literally using our upload system.

Cleanup into a Private Method

Though, I don't love having *all* of this logic right in the middle of this already-long function: it's not super obvious what it does. Let's do some cleanup: copy all of this. And at the bottom, create a new private function fakeUploadImage() that will return a string.

102 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 13
14 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
15 {
... lines 16 - 90
91     private function fakeUploadImage(): string
92     {
... lines 93 - 99
100    }
101 }
```

Paste all that logic and return the $this->uploaderHelper line. It selects a random image, uploads it and returns the path.

102 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 13
14 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
15 {
... lines 16 - 90
91     private function fakeUploadImage(): string
92     {
93         $randomImage = $this->faker->randomElement(self::$articleImages);
94         $fs = new Filesystem();
95         $targetPath = sys_get_temp_dir().'/'.$randomImage;
96         $fs->copy(__DIR__.'/images/'.$randomImage, $targetPath, true);
97
98         return $this->uploaderHelper
99             ->uploadArticleImage(new File($targetPath));
100    }
101 }
```

Back up top, delete all this stuff and say $imageFilename = $this->fakeUploadImage().

102 lines | src/DataFixtures/ArticleFixtures.php

```
... lines 1 - 13
14 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
15 {
... lines 16 - 33
34
35     protected function loadData(ObjectManager $manager)
36     {
37         $this->createMany(10, 'main_articles', function($count) use ($manager) {
... lines 38 - 64
65             $imageFilename = $this->fakeUploadImage();
... lines 66 - 77
78         });
... lines 79 - 80
81     }
... lines 82 - 100
101 }
```

Let's run those fixtures one more time!

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

When it finishes... we have some new files... and the homepage is shiny! That's a solid fixture system.

Next: we'll take our first step towards storing uploaded files in the cloud by integrating the gorgeous Flysystem library.

Chapter 14: Flysystem: Filesystem Abstraction

I keep talking about how we're going to eventually move our uploads off of our server and put them onto AWS S3. But right now, our entire upload system is *very* tied to our local filesystem. For example, $file->move()? Yea, that will *always* move things physically on your filesystem.

One of my *favorite* tools to help with this problem is a library called Flysystem. It's written by our friend Frank - who co-authored our React tutorial. He also spoke at SymfonyCon in 2018 about Flysystem and that presentation is available right here on SymfonyCasts.

Flysystem gives you a nice service object that you can use to write or read files. Then, behind the scenes, you can swap out whether you want to use a local filesystem, S3, Dropbox or pretty much anything else. It gives you an easy way to work with the filesystem, but that filesystem could be local or in the cloud.

OneupFlysystemBundle

In Symfony, we have an excellent bundle for this library: Google for OneupFlysystemBundle, find their GitHub page, then click into the docs. Copy the library name, find your terminal and run:

```
$ composer require oneup/flysystem-bundle
```

Adapters & Filesystems

While Jordi is preparing our packages, go back to their docs. Flysystem has two important concepts, which you can see here in the config example. First, we need to set up an "adapter", which is a lower-level object. Give it any name - like my_adapter. Then, this key - local - is the critical part: this says that you want to use the local adapter - an adapter that stores things on the local filesystem. Click the AwsS3 adapter link. If you want to use *this* adapter and store your files in S3, you'll use the key awss3v3. Every adapter also has different options. We're going to start with the local adapter, but move to s3 later.

But the *real* star, is the *filesystem*. Same thing: you give it any nickname, like my_filesystem and then say: this filesystem uses the my_adapter adapter. We'll talk about visibility later. The *filesystem* is the object that we'll work with directly to read, write & delete files.

Ok, go check on Composer. It's done and thanks to the recipe, we have a new config/packages/oneup_flysystem.yaml file with the same config we just saw in the docs.

11 lines | config/packages/oneup_flysystem.yaml

```yaml
# Read the documentation: https://github.com/1up-lab/OneupFlysystemBundle/tree/master/Resources/doc/index.md
oneup_flysystem:
    adapters:
        default_adapter:
            local:
                directory: '%kernel.cache_dir%/flysystem'
    filesystems:
        default_filesystem:
            adapter: default_adapter
            alias: League\Flysystem\Filesystem
```

Configuring the Adapter & Filesystem

Let's create 1 adapter and 1 filesystem for our uploads. Call the adapter, how about, public_uploads_adapter. I'm saying "public uploads" because this will put things into the public/ directory: they will be publicly accessible. We'll talk about private uploads soon - those are files where you need to do some security checks before you allow a user to see them. Change the directory to %kernel.project_dir% and then /public/uploads.

```
10 lines | config/packages/oneup_flysystem.yaml
... line 1
2 oneup_flysystem:
3     adapters:
4         public_uploads_adapter:
5             local:
6                 directory: '%kernel.project_dir%/public/uploads'
... lines 7 - 10
```

That is the *root* of this filesystem: everything will be stored relative to this. Give the filesystem a similar name - public_uploads_filesystem - and set adapter: to public_uploads_adapter.

```
10 lines | config/packages/oneup_flysystem.yaml
... line 1
2 oneup_flysystem:
... lines 3 - 6
7     filesystems:
8         public_uploads_filesystem:
9             adapter: public_uploads_adapter
```

Filesystem Alias?

What about this alias key? Let's see what that does. First, when you configure a filesystem here, it creates a service. Find your terminal and run:

```
$ php bin/console debug:container flysystem
```

There it is: oneup_flysystem.public_uploads_filesystem_filesystem. *That* service was created thanks to our config and we'll use it soon in UploaderHelper. The bundle *also* created another service called: League\Flysystem\Filesystem. Well, actually, it's an *alias*: I'll type 61 to view more info about it. Yep! This points to our public_uploads_filesystem service. The *purpose* of this is that it allows us to type-hint League\Flysystem\Filesystem and Symfony will autowire our filesystem service.

If you only have 1 filesystem, having this alias is great. But if you have multiple, well, you can only autowire *one* of them. I'm going to remove the alias - I'll show you another way to access the filesystem service.

Ok, config done! Next, let's start using this shiny new Filesystem service.

Chapter 15: Using the Filesystem

Config done! Let's get to work in UploaderHelper. Instead of passing the $uploadsPath, which we *were* using to store things, change this to FilesystemInterface - the one from Flysystem - $filesystem. Use that below, and rename the property to $filesystem.

49 lines | src/Service/UploaderHelper.php

```
... lines 1 - 5
6  use League\Flysystem\FilesystemInterface;
... lines 7 - 10
11 class UploaderHelper
12 {
... lines 13 - 14
15     private $filesystem;
... lines 16 - 18
19     public function __construct(FilesystemInterface $filesystem, RequestStackContext $requestStackContext)
20     {
21         $this->filesystem = $filesystem;
22         $this->requestStackContext = $requestStackContext;
23     }
... lines 24 - 47
48 }
```

Now, in the method, instead of $file->move(), we can say $this->filesystem->write(), which is used to create new files. Pass this self::ARTICLE_IMAGE.'/'.$newFilename and then the *contents* of the file: file_get_contents() with $file->getPathname().

49 lines | src/Service/UploaderHelper.php

```
... lines 1 - 10
11 class UploaderHelper
12 {
... lines 13 - 24
25     public function uploadArticleImage(File $file): string
26     {
... lines 27 - 33
34         $this->filesystem->write(
35             self::ARTICLE_IMAGE.'/'.$newFilename,
36             file_get_contents($file->getPathname())
37         );
... lines 38 - 39
40     }
... lines 41 - 47
48 }
```

That's it! This File object has a *ton* of different methods for getting the filename, the full path, the file without the extension and more. Honestly, I get them all confused and have to Google them. getPathname() gives us the absolute file path on the filesystem.

Above, we can get rid of the unused $destination variable. Because the filesystem's root is public/uploads/, the only thing we need to pass to write() is the path *relative* to that: article_image/ and then $newFilename.

I think we're ready! Let's clear out the uploads/ directory again. And then try our fixtures:

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

Oh! It does *not* work!

Binding the Filesystem for Autowiring

> Unused binding $uploadsPath in service UniqueUserValidator.

This is a *bad* error message from Symfony, at least the *second* half of the message. A minute ago, we had an argument here called $uploadsPath. Open up config/services.yaml. Ah, that worked because we have $uploadsPath configured as a global bind. And when you configure a bind, it must be used in at least *one* place in your app. If it's not used anywhere, you get this error. It's kinda nice: Symfony is saying:

> Hey! You configured this bind... but you're not using it - are you maybe... messing something up on accident?

The UniqueUserValidator part of the message is really a bug in the error message, which makes this a bit confusing.

Anyways, remove that bind and try the fixtures again:

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

This is the error I was waiting for.

> Cannot autowire service UploaderHelper argument $filesystem of __construct() references FilesystemInterface but no such service exists.

There are two ways to fix this. First, we could re-add the alias option and point it at this FilesystemInterface. *Or*, we can create a new bind. I'll do the second, because it works better if you have multiple filesystem services, which we will soon. First, rename the argument to be more descriptive, how about $publicUploadFilesystem.

```
49 lines | src/Service/UploaderHelper.php
... lines 1 - 10
11 class UploaderHelper
12 {
... lines 13 - 18
19     public function __construct(FilesystemInterface $publicUploadsFilesystem, RequestStackContext $requestStackContext)
20     {
21         $this->filesystem = $publicUploadsFilesystem;
... line 22
23     }
... lines 24 - 47
48 }
```

Then, under bind, set $publicUploadFilesystem to the filesystem service id - you can see it in the error. It suggests *two* services that implement the FilesystemInterface type-hint - we want the second one. Type @ then paste.

```
48 lines | config/services.yaml
... lines 1 - 9
10 services:
... lines 11 - 19
20         bind:
... lines 21 - 22
23             $publicUploadsFilesystem: '@oneup_flysystem.public_uploads_filesystem_filesystem'
... lines 24 - 48
```

One more time for the fixtures!

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

Ok, no error! Check out the public/uploads/ directory. Yes! We have files! Refresh the homepage. We are good! We still need to tweak a few more details, but our app is now *way* more ready to work locally or in the cloud.

Chapter 16: Flysystem: Streaming & Defensive Coding

There are a few minor problems with our new Flysystem integration. Let's clean them up before they bite us!

Streaming

The first is that using file_get_contents() eats memory: it reads the entire contents of the file into PHP's memory. That's not a huge deal for tiny files, but it *could* be a big deal if you start uploading bigger stuff. And, it's just not necessary.

For that reason, in general, when you use Flysystem, instead of using methods like ->write() or ->update(), you should use ->writeStream() or ->updateStream().

53 lines | src/Service/UploaderHelper.php

```
... lines 1 - 10
11 class UploaderHelper
12 {
... lines 13 - 24
25     public function uploadArticleImage(File $file): string
26     {
... lines 27 - 34
35         $this->filesystem->writeStream(
... lines 36 - 37
38         );
... lines 39 - 43
44     }
... lines 45 - 51
52 }
```

It works the same, except that we need to pass a *stream* instead of the contents. Create the stream with $stream = fopen($file->getPathname()) and, because we just need to *read* the file, use the r flag. Now, pass stream instead of the contents.

53 lines | src/Service/UploaderHelper.php

```
... lines 1 - 10
11 class UploaderHelper
12 {
... lines 13 - 24
25     public function uploadArticleImage(File $file): string
26     {
... lines 27 - 33
34         $stream = fopen($file->getPathname(), 'r');
35         $this->filesystem->writeStream(
36             self::ARTICLE_IMAGE.'/'.$newFilename,
37             $stream
38         );
... lines 39 - 43
44     }
... lines 45 - 51
52 }
```

Yea... that's it! Same thing, but no memory issues. But we *do* need to add one more detail after: if is_resource($stream), then fclose($stream). The "if" is needed because *some* Flysystem adapters close the stream by themselves.

53 lines | src/Service/UploaderHelper.php

```
... lines 1 - 10
11 class UploaderHelper
12 {
... lines 13 - 24
25     public function uploadArticleImage(File $file): string
26     {
... lines 27 - 33
34         $stream = fopen($file->getPathname(), 'r');
35         $this->filesystem->writeStream(
36             self::ARTICLE_IMAGE.'/'.$newFilename,
37             $stream
38         );
39         if (is_resource($stream)) {
40             fclose($stream);
41         }
... lines 42 - 43
44     }
... lines 45 - 51
52 }
```

Deleting the Old File

Ok, for problem number two, go back to /admin/article. Log back in with password engage, edit an article, and go select an image - how about astronaut.jpg. Hit update and... it works! So what's the problem? Well, we just *replaced* an existing image with this new one. Does the old file still exist in our uploads directory? Absolutely! But it probably shouldn't. When an article image is updated, let's delete the old file.

In UploaderHelper, add a second argument - a *nullable* string argument called $existingFilename.

57 lines | src/Service/UploaderHelper.php

```
... lines 1 - 10
11 class UploaderHelper
12 {
... lines 13 - 24
25     public function uploadArticleImage(File $file, ?string $existingFilename): string
26     {
... lines 27 - 47
48     }
... lines 49 - 55
56 }
```

This is nullable because sometimes there may *not* be an existing file to delete. At the bottom, it's beautifully simple: if an $existingFilename was passed, then $this->filesystem->delete() and pass that the full path, which will be self::ARTICLE_IMAGE.'/'.$existingFilename.

```
57 lines | src/Service/UploaderHelper.php
... lines 1 - 10
11 class UploaderHelper
12 {
... lines 13 - 24
25     public function uploadArticleImage(File $file, ?string $existingFilename): string
26     {
... lines 27 - 42
43         if ($existingFilename) {
44             $this->filesystem->delete(self::ARTICLE_IMAGE.'/'.$existingFilename);
45         }
... lines 46 - 47
48     }
... lines 49 - 55
56 }
```

Done! You can see the astronaut file that we're using right now. Oh, but first, head over to ArticleAdminController: we need to pass this new argument. Let's see - this is the edit() action - so pass $article->getImageFilename().

```
126 lines | src/Controller/ArticleAdminController.php
... lines 1 - 17
18 class ArticleAdminController extends BaseController
19 {
... lines 20 - 57
58     public function edit(Article $article, Request $request, EntityManagerInterface $em, UploaderHelper $uploaderHelper)
59     {
... lines 60 - 64
65         if ($form->isSubmitted() && $form->isValid()) {
... lines 66 - 67
68             if ($uploadedFile) {
69                 $newFilename = $uploaderHelper->uploadArticleImage($uploadedFile, $article->getImageFilename());
... line 70
71             }
... lines 72 - 80
81         }
... lines 82 - 85
86     }
... lines 87 - 124
125 }
```

In new(), you can really just pass null - there will *not* be an article image. But I'll pass getImageFilename() to be consistent.

```
126 lines | src/Controller/ArticleAdminController.php
... lines 1 - 17
18 class ArticleAdminController extends BaseController
19 {
... lines 20 - 23
24     public function new(EntityManagerInterface $em, Request $request, UploaderHelper $uploaderHelper)
25     {
... lines 26 - 28
29         if ($form->isSubmitted() && $form->isValid()) {
... lines 30 - 35
36             if ($uploadedFile) {
37                 $newFilename = $uploaderHelper->uploadArticleImage($uploadedFile, $article->getImageFilename());
... line 38
39             }
... lines 40 - 46
47         }
... lines 48 - 51
52     }
... lines 53 - 124
125 }
```

Oh, and there's one other place we need update: ArticleFixtures. Down here, just pass null: we are never updating.

```
102 lines | src/DataFixtures/ArticleFixtures.php
... lines 1 - 13
14 class ArticleFixtures extends BaseFixture implements DependentFixtureInterface
15 {
... lines 16 - 90
91     private function fakeUploadImage(): string
92     {
... lines 93 - 97
98         return $this->uploaderHelper
99             ->uploadArticleImage(new File($targetPath), null);
100    }
101 }
```

Try it! Here is the current astronaut image. Now, move over, upload rocket.jpg this time and update! Back in the directory... there's rocket and astronaut is gone! Love it!

Avoiding Errors

In a *perfect* system, the existing file will *always* exist, right? I mean, how could a filename get set on the entity... without being uploaded? Well, what if we're developing locally... and maybe we clear out the uploads directory to test something - or we clear out the uploads directory in our automated tests. What would happen?

Let's find it! Empty uploads/. Back in our browser, the image preview still shows up because this is rendering a thumbnail file - which we didn't delete - but the original image is totally gone. Select earth.jpeg, update and... it fails! It fails on $this->filesystem->delete().

This *may* be the behavior you want: if something weird happens and the old file is gone, *please* explode so that I know. But, I'm going to propose something slightly less hardcore. If the old file doesn't exist for some reason, I don't want the entire process to fail... it really doesn't need to.

The error from Flysystem is a FileNotFoundException from League\Flysystem. In UploaderHelper wrap that line in a try-catch. Let's catch that FileNotFoundException - the one from League\Flysystem

```
66 lines | src/Service/UploaderHelper.php
... lines 1 - 5
6  use League\Flysystem\FileNotFoundException;
... lines 7 - 12
13 class UploaderHelper
14 {
... lines 15 - 29
30     public function uploadArticleImage(File $file, ?string $existingFilename): string
31     {
... lines 32 - 47
48         if ($existingFilename) {
49             try {
50                 $this->filesystem->delete(self::ARTICLE_IMAGE.'/'.$existingFilename);
51             } catch (FileNotFoundException $e) {
... line 52
53             }
54         }
... lines 55 - 56
57     }
... lines 58 - 64
65 }
```

Logging Problems

That'll fix that problem... but I don't *love* doing this. Honestly, I *hate* silencing errors. One of the benefits of throwing an exception is that we can configure Symfony to notify us of errors via the logger. At SymfonyCasts, we send all errors to a Slack channel so we know if something weird is going on... not that we *ever* have bugs. Pfff.

Here's what I propose: a *soft* failure: we don't fail, but we *do* log that an error happened. Back on the constructor, autowire a new argument: LoggerInterface $logger. I'll hit Alt + Enter and select initialize fields to create that property and set it.

```
66 lines | src/Service/UploaderHelper.php
... lines 1 - 7
8  use Psr\Log\LoggerInterface;
... lines 9 - 12
13 class UploaderHelper
14 {
... lines 15 - 20
21     private $logger;
22
23     public function __construct(FilesystemInterface $publicUploadsFilesystem, RequestStackContext $requestStackContext, LoggerInte
24     {
... lines 25 - 26
27         $this->logger = $logger;
28     }
... lines 29 - 64
65 }
```

Now, down in the catch, say $this->logger->alert() - alert is one of the highest log levels and I usually send all logs that are this level or higher to a Slack channel. Inside, how about: "Old uploaded file %s was missing when trying to delete" - and pass $existingFilename.

```
66 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 29
30     public function uploadArticleImage(File $file, ?string $existingFilename): string
31     {
... lines 32 - 47
48         if ($existingFilename) {
49             try {
50                 $this->filesystem->delete(self::ARTICLE_IMAGE.'/'.$existingFilename);
51             } catch (FileNotFoundException $e) {
52                 $this->logger->alert(sprintf('Old uploaded file "%s" was missing when trying to delete', $existingFilename));
53             }
54         }
... lines 55 - 56
57     }
... lines 58 - 64
65 }
```

Thanks to this, the user gets a smooth experience, but *we* get notified so we can figure out how the heck the old file disappeared.

Move over and re-POST the form. *Now* it works. And to prove the log worked, check out the terminal tab where we're running the Symfony web server: it's streaming all of our logs here. Scroll up and... there it is!

> Old uploaded file "rocket..." was missing when trying to delete

Checking for Filesystem Failure

Ok, there's *one* more thing I want to tighten up. If one of the calls to the Filesystem object fails... what do you think will happen? An exception? Hold Command or Ctrl and click on writeStream(). Check out the docs: we *will* get an exception if we pass an invalid stream or if the file already exists. But for any other type of failure, maybe a network error... instead of an exception, the method just returns false!

Actually, that's not *completely* true - it depends on your adapter. For example, if you're using the S3 adapter and there's a network error, it *may* throw its own type of exception. But the point is this: if any of the Filesystem methods fail, you might *not* get an exception: it might just return false.

For that reason, I like to code defensively. Assign this to a $result variable.

```
75 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 29
30     public function uploadArticleImage(File $file, ?string $existingFilename): string
31     {
... lines 32 - 39
40         $result = $this->filesystem->writeStream(
... lines 41 - 42
43         );
... lines 44 - 65
66     }
... lines 67 - 73
74 }
```

Then say: if ($result === false), let's throw our own exception - I *do* want to know that something failed:

Could not write uploaded file "%s"

and pass $newFilename.

75 lines | src/Service/UploaderHelper.php

```
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 29
30     public function uploadArticleImage(File $file, ?string $existingFilename): string
31     {
... lines 32 - 39
40         $result = $this->filesystem->writeStream(
... lines 41 - 42
43         );
... line 44
45         if ($result === false) {
46             throw new \Exception(sprintf('Could not write uploaded file "%s"', $newFilename));
47         }
... lines 48 - 65
66     }
... lines 67 - 73
74 }
```

Copy that and do the same for delete:

Could not delete old uploaded file "%s"

with $existingFilename.

75 lines | src/Service/UploaderHelper.php

```
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 29
30     public function uploadArticleImage(File $file, ?string $existingFilename): string
31     {
... lines 32 - 52
53         if ($existingFilename) {
54             try {
55                 $result = $this->filesystem->delete(self::ARTICLE_IMAGE.'/'.$existingFilename);
56
57                 if ($result === false) {
58                     throw new \Exception(sprintf('Could not delete old uploaded file "%s"', $existingFilename));
59                 }
60             } catch (FileNotFoundException $e) {
... line 61
62             }
63         }
... lines 64 - 65
66     }
... lines 67 - 73
74 }
```

I'm *throwing* this error instead of just logging something because this would *truly* be an exceptional case - we shouldn't let things continue. But, it's your call.

Let's make sure this all works: move over and select the stars file - or... actually the "Earth from Moon" photo. Update and...

got it!

Next: let's teach LiipImagineBundle to play nice with Flysytem. After all, if we move Flysystem to S3, but LiipImagineBundle is still looking for the source files locally... well... we're not going to have a great time.

Chapter 17: Flysystem <3 LiipImagineBundle

Flysystem is *killing* it for us! But... there's a problem hiding... like, a it-won't-actually-work-in-the-real-world kind of problem. Yikes! In *theory*, we should be able to go into the oneup_flysystem.yaml file right now, change the adapter to S3 and everything would work. In *theory*.

How LiipImagineBundle Finds Images

The problem is LiipImagineBundle. Open up templates/article/homepage.html.twig: we call uploaded_asset(), pass that article.imagePath and *that* value is passed into imagine_filter. So basically, a string like uploads/article_image/something.jpg is passed to the filter.

The problem? By default, LiipImagineBundle *reads* the source image file from the *filesystem*. If we refactored to use S3... well... imagine would be looking in the wrong place!

You can see this by running:

```
$ php bin/console debug:config liip_imagine
```

This is the current config for this bundle, which includes all of its default values. Near the bottom, see that "loaders" section? The "loader" is the piece that's responsible for *reading* the source image. It defaults to using the filesystem and it knows to look in the public/ directory! So when we pass it upload/article_image/ some filename, it finds it perfectly. Well... it works until our files don't live on the server anymore.

The solution? We need this to use Flysystem.

Flysystem Loader

Let's go to back to the LiipImagineBundle documentation: find their GitHub page and then click down here on the "Download the Bundle" link as an easy way to get into their full docs. Now, go back to the main page and... down here near the bottom, it talks about different "data loaders". The default is "File System", we want Flysystem.

Let's see... yea, we've already installed the bundle. Copy this loaders section - we already have our Flysystem config all set up. Then, open our liip_imagine.yaml file and, really, anywhere, paste!

This creates a loader called profile_photos - that name can be anything. Let's use flysystem_loader. The critical part is the key flysystem: that says to use the "Flysystem" loader that comes with the bundle. The only thing *it* needs to know, is the service id of the filesystem that we want to use.

```
57 lines | config/packages/liip_imagine.yaml
1  liip_imagine:
   ... lines 2 - 5
6      loaders:
7          flysystem_loader:
8              flysystem:
   ... lines 9 - 57
```

For that, go back to config/services.yaml and copy the *long* service id from the bind section. Back in liip_imagine.yaml, paste!

```
57 lines | config/packages/liip_imagine.yaml
1  liip_imagine:
   ... lines 2 - 5
6      loaders:
7          flysystem_loader:
8              flysystem:
9                  filesystem_service: oneup_flysystem.public_uploads_filesystem_filesystem
   ... lines 10 - 57
```

We now have a "loader" called flysystem_loader, and a "loader's" job is to... ya know, "load" the source file. You can *technically* have multiple loaders, though I've never had to do that. To *always* have LiipImagineBundle load the files via Flysystem, below, add data_loader set to the loader's name: flysystem_loader. I'll add a comment:

default loader to use for all filter sets

```
57 lines | config/packages/liip_imagine.yaml
1  liip_imagine:
   ... lines 2 - 5
6      loaders:
7          flysystem_loader:
8              flysystem:
9                  filesystem_service: oneup_flysystem.public_uploads_filesystem_filesystem
10
11     # default loader to use for all filter sets
12     data_loader: flysystem_loader
   ... lines 13 - 57
```

Because, you can technically specify which loader you want to use on each filterset. Again, I've never had to do that: we always want to use flysystem.

Cool! Let's try it! Go into the public/ directory... let me find it... and delete all the existing thumbnails - let's delete media/cache/ entirely. By doing this, the bundle will use the data loader to get the contents of each image so that it can recreate the thumbnails.

Correcting the Path to LiipImagineBundle

Testing time! Let's go back to, how about, the homepage. And... it doesn't work. Drat! Inspect element. Hmm, it *does* start with the media/cache/resolve part. Then, the path at the end is - uploads/article_image/lightspeed...png. That's the path that we're passing to the filter.

Go back to the homepage template. The problem *now* - and it's *really* cool - is that we told LiipImagineBundle to use Flysystem to load files... but the *root* of our filesystem is the public/uploads directory. In other words, if you want to read a file from our filesystem, the path needs to be *relative* to this directory. In other words, it should *not* contain the uploads/ part

The fix? Remove the uploaded_asset() function: we can just pass article.imagePath, which will be article_image/ the filename.

```
65 lines | templates/article/homepage.html.twig
... lines 1 - 2
3  {% block body %}
4      <div class="container">
5          <div class="row">
... lines 6 - 8
9              <div class="col-sm-12 col-md-8">
... lines 10 - 21
22                 <div class="article-container my-1">
23                     <a href="{{ path('article_show', {slug: article.slug}) }}">
24                         <img class="article-img" src="{{ article.imagePath|imagine_filter('squared_thumbnail_small') }}">
... lines 25 - 37
38                     </a>
39                 </div>
... line 40
41             </div>
... lines 42 - 61
62         </div>
63     </div>
64 {% endblock %}
```

I love this! Need to thumbnail something? Just pass it the Flysystem path: you don't need the word uploads or anything like that. The uploaded_asset() function *will* still be useful if you want the public path to an asset *without* thumbnailing, but if you're using imagine_filter, passing the short, relative path is all you need.

Try it! Refresh! It still doesn't work? Oh yea! A few minutes ago, we deleted all of the original images from the fixtures. But we *did* re-upload a few of them. So if you scroll down... here we go - here's the Earth image we uploaded. So, it *is* now working perfectly.

Let's reload our fixtures to make sure:

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

Now the homepage... yes - everything is here. Let's make the same change in the other two places we're thumbnailing. Click onto the show page. This lives in templates/article/show.html.twig: remove uploaded_asset there. Refresh... good!

```
86 lines | templates/article/show.html.twig
... lines 1 - 4
5  {% block content_body %}
6      <div class="row">
7          <div class="col-sm-12">
8              <img class="show-article-img" src="{{ article.imagePath|imagine_filter('squared_thumbnail_medium') }}">
... lines 9 - 25
26         </div>
27     </div>
... lines 28 - 78
79 {% endblock %}
... lines 80 - 86
```

For the other one, go back to the admin article section - log back in with password "engage", because we reloaded the database. When we're editing an image, yep, also broken.

Find this in templates/article_admin/_form.html.twig: take off uploaded_asset().

```
40 lines | templates/article_admin/_form.html.twig
1  {{ form_start(articleForm) }}
   ... lines 2 - 5
6      <div class="row">
   ... lines 7 - 13
14         <div class="col-sm-3">
15             {% if articleForm.vars.data.imageFilename %}
16                 <img src="{{ articleForm.vars.data.imagePath|imagine_filter('squared_thumbnail_small') }}" height="100">
17             {% endif %}
18         </div>
19     </div>
   ... lines 20 - 38
39 {{ form_end(articleForm) }}
```

And... got it!

The Resolver: Saving the Images to Flysystem

So, the "data loader" is responsible for reading the *original* image. But, there's *another* important concept from LiipImagineBundle called "resolvers". Click down on the "Flysystem Resolver" in their docs. The resolver is responsible for *saving* the thumbnail image back to the filesystem after all of the transformations. By default, no surprise, LiipImagineBundle writes things directly to the filesystem. So even if we moved Flysystem to s3, LiipImagineBundle would *still* be writing the thumbnail files back to our server - into the public/media directory.

Tip

You can also completely offload the processing and storage of your files to a cloud service like rokka.io by leveraging LiipRokkaImagineBundle.

Let's change that! In the docs, copy the resolvers section. Back in our liip_imagine.yaml file, paste that. It's pretty much the same as before: we'll call it flysystem_resolver and tell it to *save* the images using the same filesystem service. Remove visibility - that sets the Flysystem visibility, which is a concept we'll talk about soon. True is the default value anyways, which basically means these files will be publicly accessible.

```
67 lines | config/packages/liip_imagine.yaml
1  liip_imagine:
   ... lines 2 - 13
14     resolvers:
15         flysystem_resolver:
16             flysystem:
17                 filesystem_service: oneup_flysystem.public_uploads_filesystem_filesystem
18                 cache_prefix: media/cache
19                 root_url: /uploads
20
   ... lines 21 - 67
```

cache_prefix is the subdirectory within the filesystem where the files should be stored and root_url is the URL that all the paths will be prefixed with when the image paths are rendered. Right now, it needs to be /uploads.

For example, if LiipImagineBundle stores a file called media/cache/foo.jpg into Flysystem, we know that the public path to this will be /uploads/media/cache/foo.jpg. We'll talk more about this setting later when we move to s3.

Ok, delete the media/ directory entirely. Oh, and I almost forgot the last step: add cache set to flysystem_resolver - let's put an "r" on that.

67 lines | config/packages/liip_imagine.yaml

```yaml
liip_imagine:
... lines 2 - 13
    resolvers:
        flysystem_resolver:
            flysystem:
                filesystem_service: oneup_flysystem.public_uploads_filesystem_filesystem
                cache_prefix: media/cache
                root_url: /uploads

    # default cache resolver for saving thumbnails
    cache: flysystem_resolver
... lines 23 - 67
```

This tells the bundle to *always* use this resolver. I'm not sure why it's called "cache" - the bundle seems to use "resolver" and "cache" to describe this one concept.

Ok! Moment of truth! Refresh. Ha! It works! Go check out where the thumbnails are stored: there is *no* media/ directory anymore! The Flysystem filesystem points to the public/uploads directory, so the media/cache directory lives *there*. And thanks to the /uploads root_url, when it renders the path, it knows to start with /uploads and then the path in Flysystem.

I love this! It's a bit tricky to get these two libraries to play together perfectly. But now we are *much* more prepared to switch between local uploads and S3.

Next: we can generate public URLs to thumbnailed files or the original files. But, what if you need to force all the URLs to include the domain name? This is something you don't think about until you need to generate a PDF or send an email from a console command or worker. Then... it can be a nightmare. Let's add this to our asset system in a way that we love.

Chapter 18: Absolute Asset Paths

One of the things I've noticed is that this word uploads - the directory where uploads are being stored - is starting to show up in a few places. We have it here in our liip_imagine config file, the oneup_flysystem.yaml file and in UploaderHelper: it's used in getPublicPath().

Centralizing the uploads/ Path

It's not a *huge* problem, but repetition is a bummer and this will cause some issues when moving to S3: we'll need to hunt down all of these paths and change them to point to the S3 domain.

Let's tighten this up. In services.yaml, create two new parameters: The first will be uploads_dir_name set to uploads - this is the name of the directory where we are storing uploaded files. Call the second one uploads_base_url and set this to almost the same thing: / and then %uploads_dir_name%. This represents the base URL to the uploaded assets.

51 lines | config/services.yaml

```
... lines 1 - 5
6  parameters:
... lines 7 - 8
9      uploads_dir_name: 'uploads'
10     uploads_base_url: '/%uploads_dir_name%'
... lines 11 - 51
```

Thanks to these, we can do some cleanup! In liip_imagine.yaml, we need the URL. Copy uploads_base_url and then use %uploads_base_url%.

67 lines | config/packages/liip_imagine.yaml

```
1  liip_imagine:
... lines 2 - 13
14     resolvers:
15         flysystem_resolver:
16             flysystem:
... lines 17 - 18
19                 root_url: '%uploads_base_url%'
... lines 20 - 67
```

Next, in oneup_flysystem.yaml, we need the directory name. Copy the other parameter: %uploads_dir_name%.

10 lines | config/packages/oneup_flysystem.yaml

```
... line 1
2  oneup_flysystem:
3      adapters:
4          public_uploads_adapter:
5              local:
6                  directory: '%kernel.project_dir%/public/%uploads_dir_name%'
... lines 7 - 10
```

The last place is in UploaderHelper. The getBasePath() call will give us the directory where the site is installed - usually an empty string. Then we need to pass in the uploads_base_url parameter.

Add a new argument to the constructor: string $uploadedAssetsBaseUrl. I'll create the property by hand and give it a slightly different name: $publicAssetBaseUrl, not for any particular reason. Set that in the constructor:

```
78 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 22
23     private $publicAssetBaseUrl;
24
25     public function __construct(FilesystemInterface $publicUploadsFilesystem, RequestStackContext $requestStackContext, LoggerInte
26     {
... lines 27 - 29
30         $this->publicAssetBaseUrl = $uploadedAssetsBaseUrl;
31     }
... lines 32 - 78
```

Back in getPublicPath(), use this: getBasePath() then $this->publicAssetsBaseUrl, which will contain the / at the beginning.

```
78 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 70
71     public function getPublicPath(string $path): string
72     {
73         // needed if you deploy under a subdirectory
74         return $this->requestStackContext
75             ->getBasePath().$this->publicAssetBaseUrl.'/'.$path;
76     }
77 }
```

Cool! But, Symfony will not be able to autowire this string argument. You can see the error if you try to reload any page. Yep!

We know how to fix that: back in services.yaml, add a bind: $uploadedAssetsBaseUrl set to %uploads_base_url%. Now... it works!

```
51 lines | config/services.yaml
... lines 1 - 11
12 services:
13     # default configuration for services in *this* file
14     _defaults:
... lines 15 - 21
22         bind:
... lines 23 - 25
26             $uploadedAssetsBaseUrl: '%uploads_base_url%'
... lines 27 - 51
```

Linking to the Full Image

Small step, but with all this config in one spot, we can do something kinda cool... with almost no effort. But first, I want to *triple* check that all this public path stuff is setup correctly. Our getPublicPath() method is currently used in one spot: by the uploaded_asset() Twig function. But, we're not actually *using* this Twig function anywhere at the moment.

So try this: in the form, we're showing the thumbnail. It might be useful to allow the user to click this and see the *original* image. That's pretty easy: add <a href=""> and use uploaded_asset(articleForm.vars.data.imagePath).

That's it! Wrap this around the img tag and let's also add target="_blank".

```
42 lines | templates/article_admin/_form.html.twig
1  {{ form_start(articleForm) }}
   ... lines 2 - 5
6      <div class="row">
   ... lines 7 - 13
14         <div class="col-sm-3">
15             {% if articleForm.vars.data.imageFilename %}
16                 <a href="{{ uploaded_asset(articleForm.vars.data.imagePath) }}" target="_blank">
17                     <img src="{{ articleForm.vars.data.imagePath|imagine_filter('squared_thumbnail_small') }}" height="100">
18                 </a>
19             {% endif %}
20         </div>
21     </div>
   ... lines 22 - 40
41 {{ form_end(articleForm) }}
```

Cool. Test that - refresh and... click. Nice! This sends us directly to the *source* image.

Absolute URLs

Thanks to our setup, we can now solve a really annoying problem. Inspect element on the image: notice that both the href and the image src paths do *not* contain the domain name. That's not a problem at *all* in a normal web context. But if you ever try to render a page into a PDF with something like wkhtmltopdf or create a console command to send an email that references an uploaded file, well... suddenly, those paths will start to break! In those contexts, you *need* the URLs to be absolute.

There are a few ways to solve this... and honestly, I went back and forth on the best approach. I finally settled on something that we've used here on SymfonyCasts for years. Open your .env file. We're going to create a brand new, custom environment variable called SITE_BASE_URL. Set the default value to https://localhost:8000.

```
35 lines | .env
   ... lines 1 - 33
34 SITE_BASE_URL=https://localhost:8000
```

Remember: this file *is* committed to git, so this is the *default* value. You can create a .env.local file to override this value locally or on production. Or, of course, if it's easy, you can override this by setting a real environment variable.

Next, go back to services.yaml. And for the uploads_base_url, use %env()% and inside, SITE_BASE_URL: that's the syntax for referencing an environment variable.

```
51 lines | config/services.yaml
   ... lines 1 - 5
6  parameters:
   ... lines 7 - 9
10     uploads_base_url: '%env(SITE_BASE_URL)%/%uploads_dir_name%'
   ... lines 11 - 51
```

And... just like that - *every* single path to every single uploaded asset will now be absolute. Seriously! Test it out! Boom! Both the link href and the image src contain the https://localhost:8000 part.

And, sure, you could add some config so that you could turn this on only when you need it... but I don't really see the point. I'll keep absolute URLs always.

Next: let's start uploading *private* assets: stuff that can't be put into the public/ directory because we need to check security before we let a user download it.

Chapter 19: Setup for Uploading Private Article References

New challenge folks! Our alien authors are *begging* for a new feature: they want to be able to upload "supporting" files and attach them to the article - like PDFs that they're referencing, images... text notes... really anything. But these files will *only* be visible to anyone that can *edit* an article. I'll call these "article references" and every article will be able to have zero to many references, which is where things start to get interesting.

Creating the ArticleReference Entity

Let's create the new entity:

```
$ php bin/console make:entity
```

Call it ArticleReference and give it an article property. This will be a relation back to the Article class. This will be a ManyToOne relation: each Article can have many ArticleReferences. Then, this will be not null in the database: every ArticleReference *must* be related to an Article. Say yes to map the other side of the relationship - it's convenient to be able to say $article->getArticleReferences(). And no to orphan removal - we won't be using that feature.

Nice! Ok, this needs a few more fields: filename a string that will hold the filename on the filesystem, originalFilename, a string that will hold the *original* filename that was on the user's system - more on that later - and mimeType - we'll use that to store what *type* of file it is - which will come in handy later.

And... done! Next run:

```
$ php bin/console make:migration
```

Let's go make sure the migration file doesn't contain any surprises... yep!

> CREATE TABLE article_reference

... with a foreign key back to article. Run that with:

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

Removing Extra Adder/Remover

Before we get back to work, open the Article entity. The command *did* create the $articleReferences property that allows us to say $article->getArticleReferences(). That's super convenient. It also added addArticleReference() and removeArticleReference(). I'm going to delete these: I'm just not going to need them: I'll read the references from the article, but never set them from this direction.

324 lines | src/Entity/Article.php

```
... lines 1 - 18
class Article
{
... lines 21 - 89
    /**
     * @ORM\OneToMany(targetEntity="App\Entity\ArticleReference", mappedBy="article")
     */
    private $articleReferences;
... line 94
    public function __construct()
    {
... lines 97 - 98
        $this->articleReferences = new ArrayCollection();
    }
... lines 101 - 315
    /**
     * @return Collection|ArticleReference[]
     */
    public function getArticleReferences(): Collection
    {
        return $this->articleReferences;
    }
}
```

Form CollectionType

Ok team: let's think about how we want this to work. The user needs to be able to upload *multiple* reference files to each article. A lot of you *may* be expecting me to use Symfony's CollectionType: that's a special field that allows you to embed a *collection* of fields into a form - like multiple upload fields.

Well... sorry. We are definitely *not* going to use CollectionType. That field is hard *enough* to work with if you want to be able to add or delete rows. Adding uploading to that? Oof, that's crazy talk.

We're going to do something different. And it's going to be a *much* better user experience anyways! We're going to leave the main form alone and build a separate "article reference upload", sort of, "widget", next to it that'll eventually upload via AJAX, allow deleting, editing and re-ordering. It's gonna be schweet!

Adding the HTML Form

Open the edit template: templates/article_admin/edit.html.twig. Everything we're going to do will be inside of this template, *not* the new template. The reason is simple: trying to upload files to a *new* entity - something that hasn't been saved to the database - is super hard! You need to store files in a temporary spot, keep track of them, and assign them to the entity when your user *does* finally save - if they ever do that. So, totally possible - but complex. If you can, have your user fill in some basic data, *save* your new entity to the database, then show the upload fields.

Anyways, let's add an <hr> and set up a bit of structure: div class="row" and div class="col-sm-8". Say "Details" here and move the entire form inside. Now add a div class="col-sm-4" and say "References".

26 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 2
{% block content_body %}
    <h1>Edit the Article! ?</h1>

    <hr>

    <div class="row">
        <div class="col-sm-8">
            <h3>Details</h3>
            {{ include('article_admin/_form.html.twig', {
                button_text: 'Update!'
            }) }}
        </div>
        <div class="col-sm-4">
            <h3>References</h3>
        </div>
    </div>
{% endblock %}
... lines 20 - 26
```

Let's see how this looks... nice! Form on the left, upload widget thingy on the right.

Here's the plan: add a <form> tag with the normal method="POST" and enctype="multipart/form-data". Inside, add a single upload field: <input type="file" name="">, how about reference. Then, <button type="submit">, some classes to make it not ugly, and "Upload".

31 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 2
{% block content_body %}
... lines 4 - 7
    <div class="row">
... lines 9 - 14
        <div class="col-sm-4">
            <h3>References</h3>

            <form action="" method="POST" enctype="multipart/form-data">
                <input type="file" name="reference">
                <button type="submit" class="btn btn-sm btn-primary">Upload</button>
            </form>
        </div>
    </div>
{% endblock %}
... lines 25 - 31
```

Cool! Yes, we *are* going to talk about allowing the user to upload *multiple* files at once. Don't worry, things are going to get *much* fancier.

Next, let's get the endpoint setup for this upload and store everything in the database, including a few pieces of information about the file that we did *not* store for the article images.

Chapter 20: Uploading References

Unlike the main form on this page, this form will submit to a *different* endpoint. And instead of continuing to put more things into ArticleAdminController, let's create a new controller for everything related to article references: ArticleReferenceAdminController. Extend BaseController - that's just a small base controller we created in our Symfony series: it extends the normal AbstractController. So nothing magic happening there.

21 lines | src/Controller/ArticleReferenceAdminController.php

```
1  <?php
2
3  namespace App\Controller;
   ... lines 4 - 9
10 class ArticleReferenceAdminController extends BaseController
11 {
   ... lines 12 - 19
20 }
```

The Upload Endpoint

Back in the new class, create public function uploadArticleReference() and, above, @Route: make sure to get the one from Symfony/Component. Set the URL to, how about, /admin/article/{id}/references - where the {id} is the Article id that we want to attach the reference to. Add name="admin_article_add_reference". Oh, and let's also set methods={"POST"}.

21 lines | src/Controller/ArticleReferenceAdminController.php

```
   ... lines 1 - 4
5  use App\Entity\Article;
   ... line 6
7  use Symfony\Component\HttpFoundation\Request;
8  use Symfony\Component\Routing\Annotation\Route;
9
10 class ArticleReferenceAdminController extends BaseController
11 {
12     /**
13      * @Route("/admin/article/{id}/references", name="admin_article_add_reference", methods={"POST"})
   ... line 14
15      */
16     public function uploadArticleReference(Article $article, Request $request)
17     {
   ... line 18
19     }
20 }
```

That's optional, but it'll let us create *another* endpoint later with the same URL that can be used to *fetch* all the references for a single article.

Let's keep going! Because the article {id} is in the URL, add an Article $article argument. Oh, and we need security! You can only upload a file if you have access to *edit* this article. In our app, we check that with this @IsGranted("MANAGE", subject="article") annotation, which leverages a custom voter that we created in our Symfony series. It basically makes sure that you are the *author* of this article or a super admin.

21 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 5
6  use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
... lines 7 - 9
10 class ArticleReferenceAdminController extends BaseController
11 {
12     /**
13      * @Route("/admin/article/{id}/references", name="admin_article_add_reference", methods={"POST"})
14      * @IsGranted("MANAGE", subject="article")
15      */
16     public function uploadArticleReference(Article $article, Request $request)
17     {
... line 18
19     }
20 }
```

Finally, we're ready to fetch the file: add the Request argument - the one from HttpFoundation - and let's dd($request->files->get()) and then the name from the input field: reference.

21 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 9
10 class ArticleReferenceAdminController extends BaseController
11 {
12     /**
13      * @Route("/admin/article/{id}/references", name="admin_article_add_reference", methods={"POST"})
14      * @IsGranted("MANAGE", subject="article")
15      */
16     public function uploadArticleReference(Article $article, Request $request)
17     {
18         dd($request->files->get('reference'));
19     }
20 }
```

Solid start. Copy the route name and head back to the template. Set the action attribute to {{ path() }}, the route name, and for the placeholder part, I'll use multiple lines and pass id set to article.id. Oh wait... we don't have an article variable inside this template. We do have the articleForm variable, and we *could* get the Article from that... but to shorten things, let's properly pass it in.

33 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 2
3  {% block content_body %}
... lines 4 - 7
8      <div class="row">
... lines 9 - 14
15         <div class="col-sm-4">
... lines 16 - 17
18             <form action="{{ path('admin_article_add_reference', {
19                 id: article.id
20             }) }}" method="POST" enctype="multipart/form-data">
... lines 21 - 22
23             </form>
24         </div>
25     </div>
26 {% endblock %}
... lines 27 - 33
```

Find the edit() action of ArticleAdminController and pass an article variable. *Now* we can say article.id.

```
127 lines | src/Controller/ArticleAdminController.php
... lines 1 - 17
18 class ArticleAdminController extends BaseController
19 {
... lines 20 - 57
58     public function edit(Article $article, Request $request, EntityManagerInterface $em, UploaderHelper $uploaderHelper)
59     {
... lines 60 - 82
83         return $this->render('article_admin/edit.html.twig', [
... line 84
85             'article' => $article,
86         ]);
87     }
... lines 88 - 125
126 }
```

Phew! Ok, let's check this out: refresh and inspect element on the form. Yep, the URL looks right and the enctype attribute is there. Ok, try it: select the Symfony Best Practices doc and... upload! Yes! It's our favorite UploadedFile object!

These article references are special because we need to keep them private: they should *only* be accessible to the author or a super admin. The process for uploading & downloading private files is, a bit different.

Setting up UploaderHelper

But, we'll start in very similar way: by opening our *favorite* service, and all-around nice class, UploaderHelper. Down here, add a new public function uploadArticleReference() that will have a File argument and return a string... pretty much the same as the other method, except that we won't need an $existingFilename because we won't let ArticleReference objects be updated. If you want to upload a modified file - cool! Delete the old ArticleReference and upload a new one. You'll see what I mean as we keep building this out.

```
83 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 70
71     public function uploadArticleReference(File $file): string
72     {
... line 73
74     }
... lines 75 - 81
82 }
```

To get started, just dd($file).

```
83 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 70
71     public function uploadArticleReference(File $file): string
72     {
73         dd($file);
74     }
... lines 75 - 81
82 }
```

Back in the controller, let's finish this *whole* darn thing. Set the file to an $uploadedFile object and I'll add the same inline documentation that says that this is an UploadedFile object - the one from HttpFoundation.

```
40 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 6
7  use App\Service\UploaderHelper;
8  use Doctrine\ORM\EntityManagerInterface;
... line 9
10 use Symfony\Component\HttpFoundation\File\UploadedFile;
... lines 11 - 13
14 class ArticleReferenceAdminController extends BaseController
15 {
... lines 16 - 19
20     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
21     {
22         /** @var UploadedFile $uploadedFile */
23         $uploadedFile = $request->files->get('reference');
... lines 24 - 37
38     }
39 }
```

Then say $filename =... oh - we don't have the UploaderHelper service yet! Add that argument: UploaderHelper $uploaderHelper. Then $filename = $uploaderHelper->uploadArticleReference($uploadedFile).

```
40 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 13
14 class ArticleReferenceAdminController extends BaseController
15 {
... lines 16 - 19
20     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
21     {
... lines 22 - 24
25         $filename = $uploaderHelper->uploadArticleReference($uploadedFile);
... lines 26 - 37
38     }
39 }
```

We know that won't work yet... but if we use our *imagination*, we know that... someday, it should return the new filename that was stored on the filesystem. To put this value into the database, we need to create a new ArticleReference object and persist it.

Tightening Up ArticleReference

Oh, but before we do - go open that class. This is a *very* traditional entity: it has some properties and everything has a getter and a setter. That's great, but because every ArticleReference *needs* to have its Article property set... and because an ArticleReference will never *change* articles, find the setArticle() method and... obliterate it!

Instead, add a public function __construct() with a required Article argument. Set *that* onto the article property. This is an optional step - but it's always nice to think critically about your entities: what methods do you *not* need?

```
91 lines | src/Entity/ArticleReference.php
... lines 1 - 9
10 class ArticleReference
11 {
... lines 12 - 39
40     public function __construct(Article $article)
41     {
42         $this->article = $article;
43     }
... lines 44 - 89
90 }
```

Saving ArticleReference & the Original Filename

Back up in our controller, say $articleReference = new ArticleReference() and pass $article. Call $article->setFilename($filename) to store the unique filename where this file was stored on the filesystem.

```
40 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 5
6 use App\Entity\ArticleReference;
... lines 7 - 13
14 class ArticleReferenceAdminController extends BaseController
15 {
... lines 16 - 19
20     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
21     {
... lines 22 - 26
27         $articleReference = new ArticleReference($article);
28         $articleReference->setFilename($filename);
... lines 29 - 37
38     }
39 }
```

But remember! There are a couple of *new* pieces of info that we can set on ArticleReference - like the *original* filename. Set that to $uploadedFile->getClientOriginalName(). Now, *technically* this method can return null, though, I'm not actually sure if that's something that can happen in any realistic scenario. But, just in case, add ?? $filename. So, if the client original name is missing for some reason, fall back to $filename.

```
40 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 5
6 use App\Entity\ArticleReference;
... lines 7 - 13
14 class ArticleReferenceAdminController extends BaseController
15 {
... lines 16 - 19
20     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
21     {
... lines 22 - 26
27         $articleReference = new ArticleReference($article);
28         $articleReference->setFilename($filename);
29         $articleReference->setOriginalFilename($uploadedFile->getClientOriginalName() ?? $filename);
... lines 30 - 37
38     }
39 }
```

Finally, *just* in case we ever want to know what *type* of file this is, we'll store the file's mime type. Set this to $uploadedFile->getMimeType(). This can *also* return null - so default it to application/octet-stream, which is sort of a common way to say "I have no idea what this file is".

40 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 13
14 class ArticleReferenceAdminController extends BaseController
15 {
... lines 16 - 19
20     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
21     {
... lines 22 - 26
27         $articleReference = new ArticleReference($article);
28         $articleReference->setFilename($filename);
29         $articleReference->setOriginalFilename($uploadedFile->getClientOriginalName() ?? $filename);
30         $articleReference->setMimeType($uploadedFile->getMimeType() ?? 'application/octet-stream');
... lines 31 - 37
38     }
39 }
```

With that done, save this: add the EntityManagerInterface $entityManager argument, then $entityManager->persist($articleReference) and $entityManager->flush().

40 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 13
14 class ArticleReferenceAdminController extends BaseController
15 {
... lines 16 - 19
20     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
21     {
... lines 22 - 31
32         $entityManager->persist($articleReference);
33         $entityManager->flush();
... lines 34 - 37
38     }
39 }
```

Finish with return redirectToRoute() and send the user back to the edit page: admin_article_edit passing this id set to $article->getId().

40 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 13
14 class ArticleReferenceAdminController extends BaseController
15 {
... lines 16 - 19
20     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
21     {
... lines 22 - 34
35         return $this->redirectToRoute('admin_article_edit', [
36             'id' => $article->getId(),
37         ]);
38     }
39 }
```

Yep - that's the route on the edit endpoint.

Alright! With any luck, it should hit our dd() statement. Go back to your browser: I already have the Symfony Best Practices PDF selected. Hit update... yea! UploadedFile coming from UploaderHelper.

Next: let's move the uploaded file... except that... we can't move it using the filesystem service object we have now... because we can't store these private files in the public/ directory. Hmm...

Chapter 21: Storing Private Files

Here's the tricky part: we can't just go into UploaderHelper and use the Flysystem filesystem like we did before to save the uploaded file... because *that* writes everything into the public/uploads/ directory. If we need to check security before letting a user download a file, then it *can't* live in the public/ directory.

And *that* means we need a *second* Flysystem filesystem: one that can store things somewhere *outside* the public/ directory. Side note: it *is* possible to solve the "private" uploads problem with just *one* filesystem using signed URLs, and we'll talk about it later when we move to S3.

Creating a Private Filesystem

But for now, a great solution is to create a *private* filesystem. Open the config/packages/oneup_flysystem.yaml file. Copy the public_uploads_adapter, paste and call it private_uploads_adapter. You can store the files *anywhere*, as long as it's not in public/. But, the var/ directory is sort of meant for this type of thing. So let's say: var/uploads. Oh, and I could re-use my uploads_dir_name parameter here - but it won't give us any benefit. That parameter is really meant to keep the upload directory and *public* path to assets in sync. But these files won't have a public path anyways... we'll make them downloadable in an entirely different way.

17 lines | config/packages/oneup_flysystem.yaml

```
... line 1
2 oneup_flysystem:
3     adapters:
... lines 4 - 7
8         private_uploads_adapter:
9             local:
10                directory: '%kernel.project_dir%/var/uploads'
... lines 11 - 17
```

Next, for filesystems, do the same thing: make a private_uploads_filesystem that will use the private_uploads_adapter.

17 lines | config/packages/oneup_flysystem.yaml

```
... line 1
2 oneup_flysystem:
... lines 3 - 11
12    filesystems:
... lines 13 - 14
15        private_uploads_filesystem:
16            adapter: private_uploads_adapter
```

Cool! Next, in UploaderHelper, we're already passing the $publicUploadFilesystem as an argument. We will *also* need the private one. Before we add it here, go into services.yaml. Remember, under _defaults, we're binding the $publicUploadFilesystem argument to the public fileystem service. Let's do the same for the private one. Call it $privateUploadFilesystem and change the service id to point to the "private" one.

```
52 lines | config/services.yaml
... lines 1 - 11
12 services:
... line 13
14     _defaults:
... lines 15 - 21
22         bind:
... lines 23 - 25
26             $privateUploadsFilesystem: '@oneup_flysystem.private_uploads_filesystem_filesystem'
... lines 27 - 52
```

Now, copy that argument name and, in UploaderHelper, add a second argument: FilesystemInterface $privateUploadFilesystem. Create a new property on top called $privateFilesystem and set it below: $this->privateFilesystem = $privateUploadFilesystem

```
110 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 18
19     private $privateFilesystem;
... lines 20 - 26
27     public function __construct(FilesystemInterface $publicUploadsFilesystem, FilesystemInterface $privateUploadsFilesystem, Reques
28     {
... line 29
30         $this->privateFilesystem = $privateUploadsFilesystem;
... lines 31 - 33
34     }
... lines 35 - 110
```

Re-using the Upload Logic

Ok, we're ready! Most of the logic in uploadArticleImage() *should* be reusable: we're basically going to do the same thing... just through the *private* filesystem: we need to figure out the filename and stream it through Flysystem. The only part of this method that we *don't* need is the $existingFilename. We don't need to delete an *existing* file... because we're not going to allow files to be "updated" for a specific ArticleReference - we'll just have the user delete them and re-upload the new file.

Refactoring time! Copy all of this code down through the fclose() and, at the bottom, create a new private function called uploadFile(). This will take in the File object that we're uploading... and we also need to pass the directory name - you'll see what that is in a moment. Then add a bool $isPublic flag so that this method knows whether to store things in the public or private filesystem.

```
110 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 85
86     private function uploadFile(File $file, string $directory, bool $isPublic)
87     {
... lines 88 - 107
108     }
109 }
```

To start, paste that exact logic

110 lines | src/Service/UploaderHelper.php

```
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 85
86     private function uploadFile(File $file, string $directory, bool $isPublic)
87     {
88         if ($file instanceof UploadedFile) {
89             $originalFilename = $file->getClientOriginalName();
90         } else {
91             $originalFilename = $file->getFilename();
92         }
93         $newFilename = Urlizer::urlize(pathinfo($originalFilename, PATHINFO_FILENAME)).'-'.uniqid().'.'.$file->guessExtension();
94
95         $stream = fopen($file->getPathname(), 'r');
96         $result = $this->filesystem->writeStream(
97             self::ARTICLE_IMAGE.'/'.$newFilename,
98             $stream
99         );
100
101        if ($result === false) {
102            throw new \Exception(sprintf('Could not write uploaded file "%s"', $newFilename));
103        }
104
105        if (is_resource($stream)) {
106            fclose($stream);
107        }
108    }
109 }
```

and, at the bottom, return $newFilename. Oh, and I should also probably add a return type.

96 lines | src/Service/UploaderHelper.php

```
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 67
68     private function uploadFile(File $file, string $directory, bool $isPublic): string
69     {
... lines 70 - 92
93         return $newFilename;
94     }
95 }
```

Let's see... the first thing we need to do is handle this $isPublic argument. So Let's say $filesystem = $isPublic ? and, if it *is* public, use $this->filesystem, otherwise use $this->privateFilesystem. Below, replace $this->filesystem with $filesystem.

```
96 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 67
68     private function uploadFile(File $file, string $directory, bool $isPublic): string
69     {
... lines 70 - 76
77         $filesystem = $isPublic ? $this->filesystem : $this->privateFilesystem;
... lines 78 - 79
80         $result = $filesystem->writeStream(
... lines 81 - 82
83         );
... lines 84 - 93
94     }
95 }
```

The other thing we need to update is the directory: it's hardcoded to ARTICLE_IMAGE. Replace that with $directory: this is the directory inside the filesystem where the file will be stored.

```
96 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 67
68     private function uploadFile(File $file, string $directory, bool $isPublic): string
69     {
... lines 70 - 79
80         $result = $filesystem->writeStream(
81             $directory.'/'.$newFilename,
82             $stream
83         );
... lines 84 - 93
94     }
95 }
```

All done! Back up in uploadArticleImage(), re-select *all* that code we just copied, delete it, do a happy dance and replace it with $newFilename = $this->uploadFile() passing the $file, the directory - self::ARTICLE_IMAGE - and whether or not this file should be public, which is true.

```
96 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 36
37     public function uploadArticleImage(File $file, ?string $existingFilename): string
38     {
39         $newFilename = $this->uploadFile($file, self::ARTICLE_IMAGE, true);
... lines 40 - 53
54     }
... lines 55 - 94
95 }
```

Now we can do the same thing down in uploadArticleReference. Oh, but first, we need to create another constant for the directory const ARTICLE_REFERENCE = 'article_reference.

```
96 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... line 15
16     const ARTICLE_REFERENCE = 'article_reference';
... lines 17 - 94
95 }
```

Back down, all we need is return $this->uploadFile(), with $file, self::ARTICLE_REFERENCE and false so that it uses the *private* filesystem.

```
96 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 55
56     public function uploadArticleReference(File $file): string
57     {
58         return $this->uploadFile($file, self::ARTICLE_REFERENCE, false);
59     }
... lines 60 - 94
95 }
```

I think that's it! Let's test this puppy out! Move over and refresh to re-POST the form. No error... but I have no idea if that worked... because we're not rendering anything yet. Check out the var/ directory... var/uploads/article_reference/symfony-best-practices..., we got it!

Of course, there's absolutely no way for *anyone* to access this file... but we'll fix that up soon enough.

Next: unless we really, *really*, trust our authors, we probably shouldn't let them upload *any* file type. Let's tighten up validation.

Chapter 22: Mime Type Validation

Unless the authors that can upload these files are super, super trusted, like, you invited them to your wedding and they babysit your dog when you're on vacation level of trusted... we need some validation. Right now, an author could upload literally *any* file type to the system.

No problem: find the controller. Hmm, there's no form here. In ArticleAdminController, we put the validation on the form. Then we could check $form->isValid() and any errors rendered automatically.

Manually Validating

But because we're *not* inside a form, we need to validate directly... which is totally fine! Add another argument: ValidatorInterface $validator. This is the service that the form system uses internally for validation.

53 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 13
14 use Symfony\Component\Validator\Validator\ValidatorInterface;
... line 15
16 class ArticleReferenceAdminController extends BaseController
17 {
... lines 18 - 21
22     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
23     {
... lines 24 - 50
51     }
52 }
```

Then, before we do *anything* with that uploaded file, say $violations = $validator->validate(). Pass this the object that you want to validate. For us, it's the $uploadedFile object itself. If we stopped here, it would read any validation annotations off of that class and apply those rules... which would be *zero* rules! This is a core class! There's no validation rules, and we can't just open up that file and add them. No worries: pass a second argument: the constraint to validate against.

53 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 15
16 class ArticleReferenceAdminController extends BaseController
17 {
... lines 18 - 21
22     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
23     {
... lines 24 - 26
27         $violations = $validator->validate(
28             $uploadedFile,
... lines 29 - 31
32         );
... lines 33 - 50
51     }
52 }
```

Remember: there are two main constraints for uploads: the Image constraint that we used before and the more generic File constraint, which we need here because the user can upload more than just images. Say new File() - the one from the Validator component.

53 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 12
13 use Symfony\Component\Validator\Constraints\File;
... lines 14 - 15
16 class ArticleReferenceAdminController extends BaseController
17 {
... lines 18 - 21
22     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
23     {
... lines 24 - 26
27         $violations = $validator->validate(
28             $uploadedFile,
29             new File([
... line 30
31             ])
32         );
... lines 33 - 50
51     }
52 }
```

This constraint has two main options. The first is maxSize. Set it to 1k... just so we can see the error.

53 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 15
16 class ArticleReferenceAdminController extends BaseController
17 {
... lines 18 - 21
22     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
23     {
... lines 24 - 26
27         $violations = $validator->validate(
28             $uploadedFile,
29             new File([
30                 'maxSize' => '1k'
31             ])
32         );
... lines 33 - 50
51     }
52 }
```

This $violations variable is *basically* an array of errors... except it's not *actually* an array - it's an object that holds errors. To check if *anything* failed validation, we can say if $violations->count() is greater than 0. For now, let's just dd($violations) so we can see what it looks like.

```
53 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 15
16 class ArticleReferenceAdminController extends BaseController
17 {
... lines 18 - 21
22     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
23     {
... lines 24 - 26
27         $violations = $validator->validate(
... lines 28 - 31
32         );
33
34         if ($violations->count() > 0) {
35             dd($violations);
36         }
... lines 37 - 50
51     }
52 }
```

Cool! Move over, select the Best Practices PDF - that's definitely more than 1kb - and upload! Say hello to the ConstraintViolationList: a glorified array of ConstraintViolation error objects. And there's the message: the file is too large. If you want, you can customize that message by passing the maxSizeMessage option... cause it *is* kind of a nerdy message.

Displaying the Validation Errors

So, in theory, you can have multiple validation rules and multiple errors. To keep things simple, let's show the first error if there is one. Use $violation = $violations[0] to get it. The ConstraintViolationList class implements ArrayAccess, which is why we can use this syntax. Oh, and let's help out my editor by telling it that this is a ConstraintViolation object.

```
60 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 13
14 use Symfony\Component\Validator\ConstraintViolation;
... lines 15 - 16
17 class ArticleReferenceAdminController extends BaseController
18 {
... lines 19 - 22
23     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
24     {
... lines 25 - 34
35         if ($violations->count() > 0) {
36             /** @var ConstraintViolation $violation */
37             $violation = $violations[0];
... lines 38 - 42
43         }
... lines 44 - 57
58     }
59 }
```

And now... hmm... how *should* we show this error to the user? This controller will *eventually* turn into an AJAX, or API endpoint that communicates via JSON. But because this is still a normal form submit, the easiest option is to put the error into a flash message and display it on the next page. Say $this->addFlash(), pass it an "error" type, and then $violation->getMessage().

```
60 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 16
17 class ArticleReferenceAdminController extends BaseController
18 {
... lines 19 - 22
23     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
24     {
... lines 25 - 34
35         if ($violations->count() > 0) {
36             /** @var ConstraintViolation $violation */
37             $violation = $violations[0];
38             $this->addFlash('error', $violation->getMessage());
... lines 39 - 42
43         }
... lines 44 - 57
58     }
59 }
```

Finish by stealing the redirect code from the bottom to send us back to the edit page.

```
60 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 16
17 class ArticleReferenceAdminController extends BaseController
18 {
... lines 19 - 22
23     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
24     {
... lines 25 - 34
35         if ($violations->count() > 0) {
36             /** @var ConstraintViolation $violation */
37             $violation = $violations[0];
38             $this->addFlash('error', $violation->getMessage());
39
40             return $this->redirectToRoute('admin_article_edit', [
41                 'id' => $article->getId(),
42             ]);
43         }
... lines 44 - 57
58     }
59 }
```

To *render* that flash message, open templates/base.html.twig and scroll down... I'm looking for the flash message logic we added in our Symfony series. There it is! We're rendering success messages, but we don't have anything to render error messages. Copy this, paste, and loop over error. Make it look *scary* with alert-danger.

105 lines | templates/base.html.twig

```
... line 1
2  <html lang="en">
... lines 3 - 15
16     <body>
... lines 17 - 73
74         {% for message in app.flashes('error') %}
75             <div class="alert alert-danger">
76                 {{ message }}
77             </div>
78         {% endfor %}
... lines 79 - 102
103    </body>
104 </html>
```

Cool! Test it out - refresh! And... nice! It redirects and *there* is our error.

Validating the Mime Types

This is great... but what we *really* want to do is control the *types* of files that are uploaded. Change the max size to 5m and add a mimeTypes option set to an array.

69 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 16
17 class ArticleReferenceAdminController extends BaseController
18 {
... lines 19 - 22
23     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
24     {
... lines 25 - 27
28         $violations = $validator->validate(
29             $uploadedFile,
30             new File([
31                 'maxSize' => '5M',
32                 'mimeTypes' => [
... lines 33 - 39
40                 ]
41             ])
42         );
... lines 43 - 66
67     }
68 }
```

Let's see... what *do* we want to allow? Well, probably *any* image is ok - so we can use image/* and definitely we should allow application/pdf.

69 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 16
17 class ArticleReferenceAdminController extends BaseController
18 {
... lines 19 - 22
23     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
24     {
... lines 25 - 27
28         $violations = $validator->validate(
29             $uploadedFile,
30             new File([
31                 'maxSize' => '5M',
32                 'mimeTypes' => [
33                     'image/*',
34                     'application/pdf',
... lines 35 - 39
40                 ]
41             ])
42         );
... lines 43 - 66
67     }
68 }
```

But... what else? It's tricky: there are a lot of mime types out there. A nice way to cheat is to press Shift+Shift and look for a core class called MimeTypeExtensionGuesser.

This is a pretty neat class: it's what Symfony uses behind the scenes to "guess" the correct file extension based on the mime type of a file. It's useful right *now* because it has a *huge* list of mime types and their extensions. Check it out: search for 'doc'. There it is: application/msword. And if you keep digging for other things like docx or xls, you can get a pretty good list of stuff you might want to accept.

Close this file and go back to the option: I'll paste in a few mime types. This covers a lot your standard "document" stuff. Oh, I forgot one! Add application/vnd.ms-excel.

```
70 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 16
17 class ArticleReferenceAdminController extends BaseController
18 {
... lines 19 - 22
23     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
24     {
... lines 25 - 27
28         $violations = $validator->validate(
29             $uploadedFile,
30             new File([
31                 'maxSize' => '5M',
32                 'mimeTypes' => [
33                     'image/*',
34                     'application/pdf',
35                     'application/msword',
36                     'application/vnd.ms-excel',
37                     'application/vnd.openxmlformats-officedocument.wordprocessingml.document',
38                     'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet',
39                     'application/vnd.openxmlformats-officedocument.presentationml.presentation',
40                     'text/plain'
41                 ]
42             ])
43         );
... lines 44 - 67
68     }
69 }
```

Let's try it out! Go back, select the Best Practices PDF, Upload and... no error! Try it again - but with this earth.zip file - that's a zip of two photos. Submit and... error! But *wow* is that a wordy error. You an change that message with the mimeTypesMessage option.

Requiring the File

Oh! There's *one* last case we need to validate for. Hit enter on the URL to refresh the form. Do *nothing* and hit upload. Ah!!! Whoops! Everything explodes inside UploaderHelper... because there *is* no uploaded file! The horror!

Back in the controller, the second argument to validate() can accept an *array* of validation constraints. Put the new File into an array. Then add: new NotBlank() with a custom message: please select a file to upload.

73 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 13
14 use Symfony\Component\Validator\Constraints\NotBlank;
... lines 15 - 17
18 class ArticleReferenceAdminController extends BaseController
19 {
... lines 20 - 23
24     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
25     {
... lines 26 - 28
29         $violations = $validator->validate(
30             $uploadedFile,
31             [
32                 new NotBlank(),
33                 new File([
... lines 34 - 43
44                 ])
45             ]
46         );
... lines 47 - 70
71     }
72 }
```

Refresh one more time. The huge error is replaced by a *much* more pleasant validation message.

Next: the author can *upload* a file reference... but it is literally impossible for them to *download* it. How can we make these private files accessible, but still check security first?

Chapter 23: Endpoint for Downloading Private Files

When we upload an article reference file, it *successfully* gets moved into the var/uploads/article_reference/ directory. That's great. And *that* means those files are not publicly accessible to anyone... which is what we wanted.

Listing the Uploaded References

Except... how can we allow *authors* to access them? As a first step, let's at least *list* the files on the page. In edit.html.twig, add a <ul> with some Bootstrap classes.

```
43 lines | templates/article_admin/edit.html.twig
... lines 1 - 2
3  {% block content_body %}
... lines 4 - 7
8      <div class="row">
... lines 9 - 14
15         <div class="col-sm-4">
... lines 16 - 17
18             <ul class="list-group small">
... lines 19 - 23
24             </ul>
... lines 25 - 33
34         </div>
35     </div>
36 {% endblock %}
... lines 37 - 43
```

Then loop with {% for reference in article.articleReferences %}. Inside, add an <li>, a *bunch* of classes to make it look fancy, and then print, how about, reference.originalFilename.

```
43 lines | templates/article_admin/edit.html.twig
... lines 1 - 2
3  {% block content_body %}
... lines 4 - 7
8      <div class="row">
... lines 9 - 14
15         <div class="col-sm-4">
... lines 16 - 17
18             <ul class="list-group small">
19                 {% for reference in article.articleReferences %}
20                     <li class="list-group-item d-flex justify-content-between align-items-center">
21                         {{ reference.originalFilename }}
22                     </li>
23                 {% endfor %}
24             </ul>
... lines 25 - 33
34         </div>
35     </div>
36 {% endblock %}
... lines 37 - 43
```

This is pretty cool: when we move the files onto the server, we give them a weird filename. But because we saved the *original* filename, we can show that here: the author has *no* idea we're naming their files crazy things internally.

Let's see how this looks. Nice! 2 uploaded PDF's.

The Download Controller

To add a download link, we know that we can't just link to the file directly: it's not public. Instead, we're going to link to a Symfony route and controller and that *controller* will check security and return the file to the user. Let's do this in ArticleReferenceAdminController. Add a new public function, how about, downloadArticleReference().

84 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 17
18 class ArticleReferenceAdminController extends BaseController
19 {
... lines 20 - 78
79     public function downloadArticleReference(ArticleReference $reference)
80     {
... line 81
82     }
83 }
```

Add the @Route() above this with /admin/article/references/{id}/download - where the {id} this time is the id of the ArticleReference object. Then, name="admin_article_download_reference" and methods={"GET"}, just to be extra cool.

84 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 17
18 class ArticleReferenceAdminController extends BaseController
19 {
... lines 20 - 75
76     /**
77      * @Route("/admin/article/references/{id}/download", name="admin_article_download_reference", methods={"GET"})
78      */
79     public function downloadArticleReference(ArticleReference $reference)
80     {
... line 81
82     }
83 }
```

Because the {id} is the id of the ArticleReference, we can add that as an argument: ArticleReference $reference. Just dd($reference) so we can see if this is working.

84 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 17
18 class ArticleReferenceAdminController extends BaseController
19 {
... lines 20 - 75
76     /**
77      * @Route("/admin/article/references/{id}/download", name="admin_article_download_reference", methods={"GET"})
78      */
79     public function downloadArticleReference(ArticleReference $reference)
80     {
81         dd($reference);
82     }
83 }
```

Love it! Copy the route name and head back into the template. Add a here for styling and an anchor with href="{{ path() }}", the route name, and id: reference.id. For the text, I'll use the Font Awesome download icon.

```
49 lines | templates/article_admin/edit.html.twig
... lines 1 - 2
{% block content_body %}
... lines 4 - 7
    <div class="row">
... lines 9 - 14
        <div class="col-sm-4">
... lines 16 - 17
            <ul class="list-group small">
                {% for reference in article.articleReferences %}
                    <li class="list-group-item d-flex justify-content-between align-items-center">
... lines 21 - 22
                        <span>
                            <a href="{{ path('admin_article_download_reference', {
                                id: reference.id
                            }) }}"><span class="fa fa-download"></span></a>
                        </span>
                    </li>
                {% endfor %}
            </ul>
... lines 31 - 39
        </div>
    </div>
{% endblock %}
... lines 43 - 49
```

Try it out! Refresh and... download! So far so good.

Creating a Read File Stream

In some ways, our job in the controller is really simple: read the contents of the file and send it to the user. But... we don't *actually* want to read the contents of the file into a string and then put it in a Response. Because if it's a *large* file, that will eat up PHP memory.

This is already why, in UploaderHelper, we're using a *stream* to write the file. And now, we'll use a stream to *read* it. To keep all this streaming logic centralized in this class, add a new public function readStream() with a string $path argument and bool $isPublic so we know which of these two filesystems to read from.

```
112 lines | src/Service/UploaderHelper.php
... lines 1 - 12
class UploaderHelper
{
... lines 15 - 70
    public function readStream(string $path, bool $isPublic)
    {
... lines 73 - 81
    }
... lines 83 - 110
}
```

Above the method, advertise that this will return a resource - PHP doesn't have a resource return type yet. Inside, step 1 is to get the right filesystem using the $isPublic argument.

```
112 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13  class UploaderHelper
14  {
... lines 15 - 67
68      /**
69       * @return resource
70       */
71      public function readStream(string $path, bool $isPublic)
72      {
73          $filesystem = $isPublic ? $this->filesystem : $this->privateFilesystem;
... lines 74 - 81
82      }
... lines 83 - 110
111 }
```

Then, $resource = $filesystem->readStream($path).

```
112 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13  class UploaderHelper
14  {
... lines 15 - 67
68      /**
69       * @return resource
70       */
71      public function readStream(string $path, bool $isPublic)
72      {
73          $filesystem = $isPublic ? $this->filesystem : $this->privateFilesystem;
74
75          $resource = $filesystem->readStream($path);
... lines 76 - 81
82      }
... lines 83 - 110
111 }
```

That's... pretty much it! But hold Cmd or Ctrl and click to see the readStream() method. Ah yes, if this fails, Flysystem will return false. So let's code defensively: if ($resource === false), throw a new \Exception() with a nice message:

> Error opening stream for %s

and pass $path. At the bottom, return $resource.

```
112 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 67
68     /**
69      * @return resource
70      */
71     public function readStream(string $path, bool $isPublic)
72     {
73         $filesystem = $isPublic ? $this->filesystem : $this->privateFilesystem;
74
75         $resource = $filesystem->readStream($path);
76
77         if ($resource === false) {
78             throw new \Exception(sprintf('Error opening stream for "%s"', $path));
79         }
80
81         return $resource;
82     }
... lines 83 - 110
111 }
```

This is great! We now have an easy way to get a stream to *read* any file in our filesystems... which will work if the file is stored locally or somewhere else.

Checking Security

In the controller add the UploaderHelper argument. Oh, but before we use this, I forgot to check security! That was the whole point! The goal is to allow these files to be downloaded by anyone who has access to *edit* the article. We've been checking that via the @IsGranted('MANAGE') annotation - which leverages a custom voter we created in the Symfony series. We can use this annotation here because the article in the annotation refers to the $article argument to the controller.

But in this new controller, we *don't* have an article argument, so we can't use the annotation in the same way. No problem: add $article = $reference->getArticle() and then run the security check manually: $this->denyAccessUnlessGranted() with that same 'MANAGE' string and $article.

```
95 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
81     {
82         $article = $reference->getArticle();
83         $this->denyAccessUnlessGranted('MANAGE', $article);
... lines 84 - 92
93     }
94 }
```

Refresh to try it. We *still* have access because we're logged in as an admin.

Next, let's take our file stream and send it to the user! We'll also learn how to control the filename and force the user's browser to download it.

Chapter 24: Streaming the File Download

We have a method that will allow us to open a *stream* of the file's contents. But... how can we send that to the user? We're used to returning a Response object or a JsonResponse object where we already have the response as a string or array. But if you want to *stream* something to the user without reading it all into memory, you need a special class called StreamedResponse.

Add $response = new StreamedResponse(). This takes one argument - a *callback*. At the bottom, return this.

95 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 11
12 use Symfony\Component\HttpFoundation\StreamedResponse;
... lines 13 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
81     {
... lines 82 - 84
85         $response = new StreamedResponse(function() use ($reference, $uploaderHelper) {
... lines 86 - 89
90         });
91
92         return $response;
93     }
... lines 94 - 95
```

Here's the idea: we can't just start streaming the response or echo'ing content right now inside the controller: Symfony's just not ready for that yet, it has more work to do, more headers to set, etc. That's why we *normally* create a Response object and *later*, when it's ready, Symfony echo's the response's content for us.

With a StreamedResponse, when Symfony is ready to finally send the data, it executes our callback and then we can do *whatever* we want. Heck, we can echo 'foo' and that's what the user would see.

Add a use statement and bring $reference and $uploaderHelper into the callback's scope so we can use them. To send a file stream to the user, it looks a little strange. Start with $outputStream set to fopen('php://output') and wb.

95 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 11
12 use Symfony\Component\HttpFoundation\StreamedResponse;
... lines 13 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
81     {
... lines 82 - 84
85         $response = new StreamedResponse(function() use ($reference, $uploaderHelper) {
86             $outputStream = fopen('php://output', 'wb');
... lines 87 - 89
90         });
91
92         return $response;
93     }
... lines 94 - 95
```

We *usually* use fopen to write to a file. But this special php://output allows us to write to the "output" stream - a fancy way of saying that anything we write to this stream will just get "echo'ed" out. Next, set $fileStream to $uploaderHelper->readStream() and pass this the path to the file - something like article_reference/symfony-best-practices-blah-blah.pdf.

Oh, except, we don't have an easy way to do that yet! In our Article entity, we added a nice getImagePath() method that read the constant from UploaderHelper and added the filename. I like that.

Let's copy that and go do the exact same thing in ArticleReference. At the bottom, paste and rename this to getFilePath(). Let's add a return type too - I probably should have done that in Article. Then, re-type the r on UploaderHelper to get the use statement, change the constant to ARTICLE_REFERENCE and update the method call to getFilename().

97 lines | src/Entity/ArticleReference.php

```
... lines 1 - 4
5 use App\Service\UploaderHelper;
... lines 6 - 10
11 class ArticleReference
12 {
... lines 13 - 91
92     public function getFilePath(): string
93     {
94         return UploaderHelper::ARTICLE_REFERENCE.'/'.$this->getFilename();
95     }
96 }
```

Great! Back in the controller, pass $reference->getFilePath() and then false for the $isPublic argument.

```
95 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
81     {
... lines 82 - 84
85         $response = new StreamedResponse(function() use ($reference, $uploaderHelper) {
86             $outputStream = fopen('php://output', 'wb');
87             $fileStream = $uploaderHelper->readStream($reference->getFilePath(), false);
... lines 88 - 89
90         });
91
92         return $response;
93     }
... lines 94 - 95
```

Finally, now that we have a "write" stream and a "read" stream, we can use a function called stream_copy_to_stream() to... do exactly that! Copy $fileStream to $outputStream.

```
95 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
81     {
... lines 82 - 84
85         $response = new StreamedResponse(function() use ($reference, $uploaderHelper) {
86             $outputStream = fopen('php://output', 'wb');
87             $fileStream = $uploaderHelper->readStream($reference->getFilePath(), false);
88
89             stream_copy_to_stream($fileStream, $outputStream);
90         });
91
92         return $response;
93     }
... lines 94 - 95
```

There ya go! The fanciest way of echo'ing content that you've probably ever seen, but it *avoids* eating memory.

Setting the Content-Type

Try it out! Refresh and... it works... sort of. We *are* sending the file contents... but the browser is *clearly* not handling it well. The reasons is that we haven't told the browser what *type* of file this is, so it's just treating it like the world's ugliest web page.

And... hey! Remember when we stored the $mimeType of the file in the database? Whelp, that's about to come in handy... big time! Add $response->headers->set() with Content-Type set to $reference->getMimeType().

```
96 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
81     {
... lines 82 - 90
91         $response->headers->set('Content-Type', $reference->getMimeType());
... lines 92 - 93
94     }
95 }
```

Try it again. Hello PDF!

Content-Disposition: Forcing Download

Another thing you might want to do is *force* the browser to download the file. It's really up to you. By default, based on the Content-Type, the browser may try to open the file - like it is here - or have the user download it. To force the browser to *always* download the file, we can leverage a header called Content-Disposition.

This header has a very specific format, so Symfony comes with a helper to create it. Say $disposition = HeaderUtils::makeDisposition(). For the first argument, we'll tell it whether we want the user to download the file, or open it in the browser by passing HeaderUtils::DISPOSITION_ATTACHMENT or DISPOSITION_INLINE.

```
102 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 10
11 use Symfony\Component\HttpFoundation\HeaderUtils;
... lines 12 - 80
81     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
82     {
... lines 83 - 92
93         $disposition = HeaderUtils::makeDisposition(
94             HeaderUtils::DISPOSITION_ATTACHMENT,
... line 95
96         );
... lines 97 - 99
100     }
101 }
```

Next, pass it the *filename*.

This is *especially* cool because, without this, the browser would probably try to call the file... just... "download" - because that's the last part of the URL. Now it will use $reference->getOriginalFilename().

Tip

If your original filename is not in ASCII characters, add a 3rd argument to HeaderUtils::makeDisposition to provide a "fallback" filename.

```
102 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 10
11 use Symfony\Component\HttpFoundation\HeaderUtils;
... lines 12 - 80
81     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
82     {
... lines 83 - 92
93         $disposition = HeaderUtils::makeDisposition(
94             HeaderUtils::DISPOSITION_ATTACHMENT,
95             $reference->getOriginalFilename()
96         );
... lines 97 - 99
100    }
101 }
```

Before we set this header, I just want you to see what it looks like. So, dd($disposition)

```
102 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 10
11 use Symfony\Component\HttpFoundation\HeaderUtils;
... lines 12 - 80
81     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
82     {
... lines 83 - 92
93         $disposition = HeaderUtils::makeDisposition(
94             HeaderUtils::DISPOSITION_ATTACHMENT,
95             $reference->getOriginalFilename()
96         );
97         dd($disposition);
... lines 98 - 99
100    }
101 }
```

move over, refresh and... there it is. It's just a string, like any other header - but it has this specific format, which is why Symfony has a helper method.

Set this on the actual response with $response->headers->set('Content-Disposition', $disposition).

```
102 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 19
20 class ArticleReferenceAdminController extends BaseController
21 {
... lines 22 - 80
81     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
82     {
... lines 83 - 96
97         $response->headers->set('Content-Disposition', $disposition);
... lines 98 - 99
100    }
101 }
```

Try it one more time. Yes! It downloads *and* uses the original filename.

Next: let's make this all *way* cooler by uploading instantly via AJAX.

Chapter 25: Dropzone: AJAX Upload

When I started creating this tutorial, I got a lot of requests for things to talk about... which, by the way - thank you! That was awesome! Your requests *absolutely* helped drive this tutorial. One request that I heard over and over again was: handling multiple file uploads at a time.

It makes sense: instead of uploading files one-by-one, an author should be able to select a *bunch* at a time! This is something that's *totally* supported by the web: if you add a multiple attribute to a file input, boom! Your browser will allow you to select multiple files. In Symfony, we would then be handling an *array* of UploadedFile objects, instead of one.

But, I'm *not* going to show how to do that. Mostly... because I don't like the user experience! What if I select 10 files, wait for *all* of them to upload, then one is too big and fails validation? If you're not inside a form, you could probably save 9 of them and send back an error. But if you're inside a form, good luck: unless you do some serious work, *none* of them will be saved because the entire form was invalid!

I also want my files to start uploading as soon as I select them *and* I want a progress bar. Basically... I want to handle uploads via JavaScript. In fact, over the next few videos, we're going to create a pretty awesome little widget for uploading multiple files, deleting them, editing their filenames and even re-ordering them.

Installing Dropzone

First: the upload part. Google for a library called Dropzone: it's probably the most popular JavaScript library for handling file uploads. It creates a little... "drop zone"... and when you drop a file here or select a file, it starts uploading. Super nice!

Search for a Dropzone CDN. I normally use Webpack Encore, and so, whenever I need a third-party library, I install it via yarn and import it when I need to use it. If you're using Encore, you *can* do this - and I recommend it. But in this tutorial, to keep things simple, we're *not* using Encore. And so, in our edit template, we're including a normal JavaScript file that lives in the public/js/ directory: admin_article_form.js, which holds some pretty traditional JavaScript.

To get Dropzone rocking, copy the minified JavaScript file and go to the template Actually, copy the whole script tag with SRI - that'll include the nice integrity attribute.

54 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 47
48 {% block javascripts %}
... lines 49 - 50
51     <script src="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.5.1/min/dropzone.min.js" integrity="sha256-cs4thShDfjkqFGk5s2Lxj35
... line 52
53 {% endblock %}
```

Grab the minified link tag too. We don't have a stylesheets block yet, so we need to add one: {% block stylesheets %}{% endblock %}, call {{ parent() }} and paste the link tag.

54 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 41
42 {% block stylesheets %}
43     {{ parent() }}
44
45     <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/dropzone/5.5.1/min/dropzone.min.css" integrity="sha256-e47xOkX
46 {% endblock %}
... lines 47 - 54
```

Dropzone basically "takes over" your form tag. You don't need a button anymore... or even the file input. The form tag *does* need a dropzone class... but that's it!

54 lines | templates/article_admin/edit.html.twig

```
... lines 1 - 2
{% block content_body %}
... lines 4 - 7
    <div class="row">
... lines 9 - 14
        <div class="col-sm-4">
... lines 16 - 33
            <form action="{{ path('admin_article_add_reference', {
                id: article.id
            }) }}" method="POST" enctype="multipart/form-data" class="dropzone">
            </form>
        </div>
    </div>
{% endblock %}
... lines 41 - 54
```

Try it! Refresh and... hello Dropzone!

How Dropzone Uploads

When you select a file with Dropzone, it's smart enough to upload to the action URL on our form. So... in theory... it should just... sort of work.

Back in the controller, scroll up to the upload endpoint and dump($uploadedFile). I'm not using dd() - dump and die - because this will submit via AJAX - and by using dump() without die'ing, we'll be able to see it in the profiler.

103 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 19
class ArticleReferenceAdminController extends BaseController
{
... lines 22 - 25
    public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
    {
        /** @var UploadedFile $uploadedFile */
        $uploadedFile = $request->files->get('reference');
        dump($uploadedFile);
... lines 31 - 76
    }
... lines 78 - 101
}
```

Ok: select a file. The *first* cool thing is that the file upload AJAX request showed up down on the web debug toolbar! I'll click the hash and open that up in a new tab.

This is awesome! We're now looking at *all* the profiler data for that AJAX request! Actually... hmm... that's not true. Look closely: it says that we were redirected from a POST request to the admin_article_add_reference route. We're looking at the profiler for the article edit page!

This is a bit confusing. Click the "Last 10" link to see a list of the last 10 requests made into our app. Now it's more obvious: Dropzone made a POST request to /admin/article/41/references - that's our upload endpoint. But, for some reason, that redirected us to the *edit* page. Click the token link to see the profiler for the POST request.

Check out the Debug tab. There it is: *this* is the dump from our controller... and it's null. Where's our upload? The problem is that, by default, Dropzone uploads a field called file. But in the controller, we're expecting it to be called reference.

Customizing Dropzone

We *could* fix this in the controller... but we can also configure Dropzone to use the reference key. We're going to do that because, in general, as *cool* as it is that we can just add a "dropzone" class to our form and it mostly works, to *really* get this system working, we're going to need to customize a *bunch* of things on Dropzone.

Open up admin_article_form.js. First, at the very top, add Dropzone.autoDiscover = false. That tells Dropzone to *not* automatically configure itself on any form that has the dropzone class: we're going to do it manually.

42 lines | public/js/admin_article_form.js

```
1  Dropzone.autoDiscover = false;
2
   ... lines 3 - 42
```

Try it out - close the extra tab and refresh. Hmm... still there? Maybe a force refresh? *Now* it's gone. The dropzone class still gives us some styling, but it's not functional anymore.

To get it working again, inside the document.ready(), call a new initializeDropzone() function.

42 lines | public/js/admin_article_form.js

```
   ... lines 1 - 2
3  $(document).ready(function() {
4      initializeDropzone();
   ... lines 5 - 29
30 });
   ... lines 31 - 42
```

Copy that name, and, below, add it: function initializeDropzone(). If I were using Webpack Encore, I'd probably organize this function into its own file and import it.

42 lines | public/js/admin_article_form.js

```
   ... lines 1 - 31
32 function initializeDropzone() {
   ... lines 33 - 40
41 }
```

The goal here is to find the form element and initialize Dropzone on it. To do that, let's add another class on the form: js-reference-dropzone.

54 lines | templates/article_admin/edit.html.twig

```
   ... lines 1 - 2
3  {% block content_body %}
   ... lines 4 - 7
8      <div class="row">
   ... lines 9 - 14
15         <div class="col-sm-4">
   ... lines 16 - 33
34             <form action="{{ path('admin_article_add_reference', {
35                 id: article.id
36             }) }}" method="POST" enctype="multipart/form-data" class="dropzone js-reference-dropzone">
37             </form>
38         </div>
39     </div>
40 {% endblock %}
   ... lines 41 - 54
```

Copy that, and back inside our JavaScript, say var formElement = document.querySelector() with .js-reference-dropzone.

```
42 lines | public/js/admin_article_form.js
... lines 1 - 31
32 function initializeDropzone() {
33     var formElement = document.querySelector('.js-reference-dropzone');
... lines 34 - 40
41 }
```

Yes, yes, I'm using straight JavaScript here instead of jQuery to be a bit more hipster - no big reason for that. There's also a jQuery plugin for Dropzone. Next, to avoid an error on the "new" form that doesn't have this element, if !formElement, return.

```
42 lines | public/js/admin_article_form.js
... lines 1 - 31
32 function initializeDropzone() {
33     var formElement = document.querySelector('.js-reference-dropzone');
34     if (!formElement) {
35         return;
36     }
... lines 37 - 40
41 }
```

Finally, initialize things with var dropzone = new Dropzone(formElement). And *now* we can pass an array of options. The one we need now is paramName. Set it to reference.

```
42 lines | public/js/admin_article_form.js
... lines 1 - 31
32 function initializeDropzone() {
33     var formElement = document.querySelector('.js-reference-dropzone');
34     if (!formElement) {
35         return;
36     }
37
38     var dropzone = new Dropzone(formElement, {
39         paramName: 'reference'
40     });
41 }
```

That should do it! Head over and select another file - how about earth.jpeg. And... cool! It looks like it worked. Click to open the profiler for the AJAX request.

Oh... careful - once again, we got redirected! So this is the profiler for the edit page. Click the link to go back to the profiler for the POST request and go back to the Debug tab. Yes! *Now* we're getting the normal UploadedFile object.

Close this and refresh. Look at the list! There is earth.jpeg! It worked! Of course, it's a little weird that it redirected after success... and if there were a validation error... that would *also* cause a redirect... and so it would look successful to Dropzone. The problem is that our endpoint isn't set up to be an API endpoint. Let's fix that next and make Dropzone read our validation errors.

Chapter 26: API Endpoint & Errors with Dropzone

The AJAX upload finishes successfully... but the response is a redirect... which doesn't break anything technically... but it's weird. Our endpoint isn't setup to be an API endpoint - it's 100% traditional: we're redirecting on error *and* success.

But now that we *are* using this as an API endpoint, let's fix that! And... this kinda simplifies things. For the validation error, we can say return $this->json($violations, 400).

93 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 24
25     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
26     {
... lines 27 - 51
52         if ($violations->count() > 0) {
53             return $this->json($violations, 400);
54         }
... lines 55 - 66
67     }
... lines 68 - 91
92 }
```

How nice is that? And at the bottom, we don't *really* need to return anything yet, but it's pretty standard to return the JSON of a resource after creating it. So, return $this->json($articleReference).

93 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 24
25     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
26     {
... lines 27 - 64
65
66         return $this->json($articleReference);
67     }
... lines 68 - 91
92 }
```

Let's try it! Move over, refresh... even though we don't need to... and select astronaut.jpg. This time... it fails! Let's see what the error looks like. Hmm, actually, better: click to open the profiler - you can always see the error there. Oh:

> A circular reference has been detected when serializing object of class Article.

This is a *super* common problem with the serializer, and we saw it earlier. We're serializing ArticleReference. And, by default, that will serialize all the properties that have getter methods... including the article property. Then when it serializes the Article, it finds the $articleReferences property and tries to serialize the ArticleReference objects... in an endless loop.

The easiest way to fix this is to define a serialization group. In ArticleReference, above the id property, add @Groups and let's invent one called main. Put this above all the fields that we actually want to serialize, how about $id, $filename, $originalFilename and $mimeType. We're not actually *using* the JSON response yet so it doesn't matter - but we *will* use it in

a few minutes.

102 lines | src/Entity/ArticleReference.php

```
... lines 1 - 6
7  use Symfony\Component\Serializer\Annotation\Groups;
... lines 8 - 11
12 class ArticleReference
13 {
14     /**
... lines 15 - 17
18      * @Groups("main")
19      */
20     private $id;
... lines 21 - 27
28     /**
... line 29
30      * @Groups("main")
31      */
32     private $filename;
... line 33
34     /**
... line 35
36      * @Groups("main")
37      */
38     private $originalFilename;
... line 39
40     /**
... line 41
42      * @Groups("main")
43      */
44     private $mimeType;
... lines 45 - 100
101 }
```

Back in the controller, let's break this onto multiple lines. The second argument is the status code and we should actually use 201 - that's the proper status code when you've *created* a resource. Next is headers - we don't need anything custom, and, for context, add an array with groups set to ['main'].

```
100 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 24
25     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterface
26     {
... lines 27 - 65
66         return $this->json(
67             $articleReference,
68             201,
69             [],
70             [
71                 'groups' => ['main']
72             ]
73         );
74     }
... lines 75 - 98
99 }
```

Let's see if that fixed things. Close the profiler and select "stars". Duh - I totally forgot - the stars file is too big - you can see it failed. But when you hover over it... object Object? That's not a great error message... We'll fix that in a minute.

Select Earth from the Moon.jpg and... nice! It works and the JSON response looks awesome!

Displaying Errors Correctly

Ok, let's look back at what happened with stars. This failed validation and so the server returned a 400 status code. Dropzone *did* notice that - it knows it failed. But, by default, Dropzone expects the Response to be just a string with the error message, not a nice JSON structure with a detail key like we have.

No worries: we just need a little extra JavaScript to help this along. Back in admin_article_form.js, add another option called init and set that to a function.

```
49 lines | public/js/admin_article_form.js
... lines 1 - 31
32 function initializeDropzone() {
... lines 33 - 37
38     var dropzone = new Dropzone(formElement, {
... line 39
40         init: function() {
... lines 41 - 45
46         }
47     });
48 }
```

Dropzone calls this when it's setting itself up, and it's a great place to add extra behavior via events. For example, want to do something whenever there's an error? Call this.on('error') and pass that a callback with two arguments: a file object that holds details about the file that was uploaded and data - the data sent back from the server.

49 lines | public/js/admin_article_form.js

```
... lines 1 - 31
32 function initializeDropzone() {
... lines 33 - 37
38     var dropzone = new Dropzone(formElement, {
... line 39
40         init: function() {
41             this.on('error', function(file, data) {
... lines 42 - 44
45             });
46         }
47     });
48 }
```

Because the real validation message lives on the detail key, we can say: if data.detail, this.emit('error') passing file and the actual error message string: data.detail.

49 lines | public/js/admin_article_form.js

```
... lines 1 - 31
32 function initializeDropzone() {
... lines 33 - 37
38     var dropzone = new Dropzone(formElement, {
... line 39
40         init: function() {
41             this.on('error', function(file, data) {
42                 if (data.detail) {
43                     this.emit('error', file, data.detail);
44                 }
45             });
46         }
47     });
48 }
```

That's it! Refresh the *whole* thing... and upload the stars file again. It failed... but when we hover on it! Nice! There's our validation error.

Next: now that our files are automatically uploaded via AJAX, the reference list should *also* automatically update when each upload finishes. Let's render that whole section with JavaScript.

Chapter 27: Rendering the File List Client Side

Here's the plan. Since we're using Dropzone to upload things via Ajax, I want to transform this entire section into a fully JavaScript-driven dynamic widget. Some of this stuff we're going to talk about isn't strictly related to handling uploads, but I got a lot of requests to show a full upload "gallery" where you can upload, edit, delete and re-order files. So... let's do that!

Select another file to upload, like rocket.jpeg. It uploads... but you don't see it on the list until we refresh. Lame! Instead of rendering this list inside Twig, let's render it via JavaScript. Once we've done that, updating it dynamically will be easy!

Article References Collection Endpoint

To power the frontend, we need a new API endpoint that will return all of the references for a specific Article. We got this: go into ArticleReferenceAdminController and create a new public function called getArticleReferences().

109 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function getArticleReferences(Article $article)
81     {
... line 82
83     }
... lines 84 - 107
108 }
```

Add the @Route() above this with /admin/article/{id}/references.

This time, the id is the article id. URLs aren't technically important, but this is on purpose: in an API, /admin/article/{id} would be the URL to get info about a specific article. Adding /references onto that is a nice way to read its references.

Now add the methods="GET" - yes you *can* leave off the curly braces when there's just one method - and name="admin_article_list_references".

109 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 75
76     /**
77      * @Route("/admin/article/{id}/references", methods="GET", name="admin_article_list_references")
... line 78
79      */
80     public function getArticleReferences(Article $article)
81     {
... line 82
83     }
... lines 84 - 107
108 }
```

Down in the method, add the Article argument and don't forget the security check: @IsGranted("MANAGE", subject="article"). We can use the annotation this time because we *do* have an article argument. Then, oh, it's beautiful: return $this->json($article->getArticleReferences());.

```
109 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 75
76     /**
77      * @Route("/admin/article/{id}/references", methods="GET", name="admin_article_list_references")
78      * @IsGranted("MANAGE", subject="article")
79      */
80     public function getArticleReferences(Article $article)
81     {
82         return $this->json($article->getArticleReferences());
83     }
... lines 84 - 107
108 }
```

How nice is it!? Let's check it out: in the browser, take off the /edit and replace it with /references. And... oh boy, it explodes!

> Semantical error: Couldn't find constant article... make sure annotations are installed and enabled.

Well, they are - this is a *total* rookie mistake I made with my annotations. On the @IsGranted annotation, it should be subject="article". Try it again. *Here* we go - that's the error I was expecting: our favorite circular reference has been detected.

This is the *exact* same thing we saw a second ago when we tried to serialize a single ArticleReference. And the fix is the same: we need to use the main serialization group.

Pass 200 as the status code, no custom headers, but one custom groups option set to main.

```
116 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 79
80     public function getArticleReferences(Article $article)
81     {
82         return $this->json(
83             $article->getArticleReferences(),
84             200,
85             [],
86             [
87                 'groups' => ['main']
88             ]
89         );
90     }
... lines 91 - 114
115 }
```

Try it again. Gorgeous! That contains *everything* we need to render the list in JavaScript.

JavaScript Rendering

To do that, we're not going to use Vue.js or React. Those are both *wonderful* options, and if you're serious about building some high-quality front-end apps, you need to give them a serious look. But, to keep the concepts understandable, I'm going to stick to jQuery and a few modern JavaScript techniques.

Start in edit.html.twig. Find the list and completely empty it: we'll fill this in via JavaScript. But add a new class so we can find it: js-reference-list. Let's also add a data-url attribute: I want to print the URL to our new endpoint to make it easy for JavaScript to fetch the references. Copy the new route name, paste it into path and add pass the id route wildcard set to article.id.

```
42 lines | templates/article_admin/edit.html.twig
... lines 1 - 2
3  {% block content_body %}
... lines 4 - 7
8      <div class="row">
... lines 9 - 14
15         <div class="col-sm-4">
... lines 16 - 17
18             <ul class="list-group small js-reference-list" data-url="{{ path('admin_article_list_references', {id: article.id}) }}"></ul>
... lines 19 - 25
26         </div>
27     </div>
28 {% endblock %}
... lines 29 - 42
```

The ReferenceList JavaScript Class

Next, in admin_article_form.js, I'm going to paste in a class that I've started: you can copy this from the code block on this page. This uses the newer "class" syntax from JavaScript... which is compatible with *most* browsers, but not all of them. That's why I've added this note to use Webpack Encore, which will rewrite the new syntax so that it's compatible with whatever browsers you need.

84 lines | public/js/admin_article_form.js

```
... lines 1 - 33
// todo - use Webpack Encore so ES6 syntax is transpiled to ES5
class ReferenceList
{
    constructor($element) {
        this.$element = $element;
        this.references = [];
        this.render();

        $.ajax({
            url: this.$element.data('url')
        }).then(data => {
            this.references = data;
            this.render();
        })
    }

    render() {
        const itemsHtml = this.references.map(reference => {
            return `
<li class="list-group-item d-flex justify-content-between align-items-center">
    ${reference.originalFilename}

    <span>
        <a href="/admin/article/references/${reference.id}/download"><span class="fa fa-download"></span></a>
    </span>
</li>
`
        });

        this.$element.html(itemsHtml.join(''));
    }
}
... lines 66 - 84
```

Before we dive into this class, let's start using it up on our document.ready() function. Say var referenceList = new ReferenceList() and pass it $('.js-reference-list') - that's the element we just added the attribute to.

84 lines | public/js/admin_article_form.js

```
... lines 1 - 2
$(document).ready(function() {
    var referenceList = new ReferenceList($('.js-reference-list'));
... lines 5 - 31
});
... lines 33 - 84
```

And... yea! The class mostly takes care of the rest! In the constructor(), we take in the jQuery element and store it on this.$element. It also keeps track of all the *references* that it has, which starts empty and calls this.render(), whose job is to completely fill the ul element.

this.references.map is a fancy way to loop over the references array, which is empty at the start, but won't be forever. For each reference, it creates a string of HTML that is basically a copy of what we had in our template before. This uses a feature called template literals that allows us to create a multi-line string with variables inside - like reference.originalFilename and referenced.id. The data from the references will ultimately come from our new endpoint, so I'm using the same keys that our JSON has.

I *did* hardcode the URL to the download endpoint instead of doing something fancier. You could generate that with FOSJsRoutingBundle if you want, but hardcoding it is also not a huge deal.

Finally, at the bottom, we take all that HTML and stick it into the element. This is a bit similar to what React does, but *definitely* less powerful.

Back up in the constructor, the references array *starts* empty, but we immediately make an Ajax call by reading the data-url attribute off of our element. When it finishes, we set this.references to its data and once again call this.render().

Phew! Let's see if it actually works! Refresh and... yes! If you watched closely, it was empty for a *moment*, then filled in once the AJAX call finished.

Dynamically Adding the Row

Now that we're rendering this in JavaScript, we have a clean way to add a *new* row whenever a file finishes uploading. Back inside the init function for Dropzone, add another event listener: this.on('success') and pass a callback with the same file and data arguments. To start, just console.log(data) so we can see what it looks like.

88 lines | public/js/admin_article_form.js

```
... lines 1 - 66
67 function initializeDropzone() {
... lines 68 - 72
73     var dropzone = new Dropzone(formElement, {
... line 74
75         init: function() {
76             this.on('success', function(file, data) {
77                 console.log(file, data);
78             });
... lines 79 - 84
85         }
86     });
87 }
```

Ok, refresh, select any file and... in the console... nice! We *already* did the work of returning the new ArticleReference JSON on success... even though we didn't need it before. Thanks past us!

And *now*, we're dangerous. If we can somehow take that data, put it into the references property in our class and re-render, we'll be good!

To help that, add a new function called addReference(). This will take in a new reference and then push it onto this.references. Then call this.render().

96 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 49
50     addReference(reference) {
51         this.references.push(reference);
52         this.render();
53     }
... lines 54 - 69
70 }
... lines 71 - 96
```

For people that are used to React, I *do* want to mention two things. First, we're *mutating*, um, changing the this.references property when we say this.references.push(). Changing "state", which is basically what this is, is a big "no no" in React. But in our simpler system, it's fine. Second, each time we call this.render(), it is *completely* emptying the ul and re-adding all the HTML from scratch. Front-end frameworks like React or Vue are *way* smarter than this and are able to update *just* the pieces that changed.

Anyways, inside of initializeDropzone(), add a referenceList argument: we're going to force this to get passed to us. I'll even document that this will be an instance of the ReferenceList class.

96 lines | public/js/admin_article_form.js

```
... lines 1 - 71
/**
 * @param {ReferenceList} referenceList
 */
function initializeDropzone(referenceList) {
... lines 76 - 94
}
```

Back on top, pass in the object - referenceList.

96 lines | public/js/admin_article_form.js

```
... lines 1 - 2
$(document).ready(function() {
... lines 4 - 5
    initializeDropzone(referenceList);
... lines 7 - 31
});
... lines 33 - 96
```

And *now* inside success, instead of console.log(), we'll say referenceList.addReference(data).

96 lines | public/js/admin_article_form.js

```
... lines 1 - 74
function initializeDropzone(referenceList) {
... lines 76 - 80
    var dropzone = new Dropzone(formElement, {
... line 82
        init: function() {
            this.on('success', function(file, data) {
                referenceList.addReference(data);
            });
... lines 87 - 92
        }
    });
}
```

Cool! Give your page a nice refresh. And... let's see: astronaut.jpg is the last file on the list currently. So let's upload Earth from the Moon.jpeg. It uploads and... boom! So fast! We can even instantly downloaded it.

Next: let's keep leveling up: authors need a way to *delete* existing file references.

Chapter 28: Deleting Files

The next thing our file gallery needs is the ability to delete files. I know this tutorial is all about uploading... but in these chapters, we're sorta, *accidentally* creating a nice API for our Article references. We already have the ability to get all references for a specific article, create a new reference and download a reference's file. Now we need an endpoint to delete a reference.

Add a new function at the bottom called deleteArticleReference(). Put the @Route() above this with /admin/article/references/{id}, name="admin_article_delete_reference" and - this will be important - methods={"DELETE"}. We do *not* want to make it possible to make a GET request to this endpoint. First, because that's crazy-dangerous. And second, because if we kept building out the API, we would want to have a different endpoint for making a GET request to /admin/article/references/{id} that would return the JSON for that one reference.

125 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 115
116     /**
117      * @Route("/admin/article/references/{id}", name="admin_article_delete_reference", methods={"DELETE"})
118      */
119     public function deleteArticleReference(ArticleReference $reference)
120     {
... lines 121 - 122
123     }
124 }
```

Inside, add the ArticleReference $reference argument and then we'll add our normal security check. In fact, copy it from above and put it here.

125 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 18
19 class ArticleReferenceAdminController extends BaseController
20 {
... lines 21 - 115
116     /**
117      * @Route("/admin/article/references/{id}", name="admin_article_delete_reference", methods={"DELETE"})
118      */
119     public function deleteArticleReference(ArticleReference $reference)
120     {
121         $article = $reference->getArticle();
122         $this->denyAccessUnlessGranted('MANAGE', $article);
123     }
124 }
```

The deleteFile() Service Method

Ok: how can we delete a file? Through the magic of Flysystem of course! And the best place for that logic to live is probably UploaderHelper. We already have functions for uploading two types of files, getting the public path and reading a stream. Copy the readStream() function declaration, paste, rename it to deleteFile() and remove the return type.

```
123 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 83
84     public function deleteFile(string $path, bool $isPublic)
85     {
... lines 86 - 92
93     }
... lines 94 - 121
122 }
```

We'll start the same way: by grabbing whichever filesystem we need.

```
123 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 83
84     public function deleteFile(string $path, bool $isPublic)
85     {
86         $filesystem = $isPublic ? $this->filesystem : $this->privateFilesystem;
... lines 87 - 92
93     }
... lines 94 - 121
122 }
```

Next say $result = $filesystem->delete() and pass that $path.

```
123 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 83
84     public function deleteFile(string $path, bool $isPublic)
85     {
86         $filesystem = $isPublic ? $this->filesystem : $this->privateFilesystem;
87
88         $result = $filesystem->delete($path);
... lines 89 - 92
93     }
... lines 94 - 121
122 }
```

Finally, code defensively: if $result === false, throw a new exception with Error deleting "%s" and $path.

```
123 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 83
84     public function deleteFile(string $path, bool $isPublic)
85     {
86         $filesystem = $isPublic ? $this->filesystem : $this->privateFilesystem;
87
88         $result = $filesystem->delete($path);
89
90         if ($result === false) {
91             throw new \Exception(sprintf('Error deleting "%s"', $path));
92         }
93     }
... lines 94 - 121
122 }
```

The DELETE Endpoint

That's nice! Back in the controller, add an UploaderHelper argument, oh and we're also going to need the EntityManagerInterface service as well. Remove the reference from the database with $entityManager->remove($reference) and $entityManager->flush(). Then $uploaderHelper->deleteFile() passing that $reference->getFilePath() and false so it uses the private filesystem.

```
133 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 19
20 class ArticleReferenceAdminController extends BaseController
21 {
... lines 22 - 119
120     public function deleteArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper, EntityManagerInterface $en
121     {
... lines 122 - 124
125         $entityManager->remove($reference);
126         $entityManager->flush();
127
128         $uploaderHelper->deleteFile($reference->getFilePath(), false);
... lines 129 - 130
131     }
132 }
```

Quick note: in the real world, if there was a problem deleting the file from Flysystem - which is *definitely* possible when you're storing in the cloud - then you could end up with a situation where the *row* is deleted in the database, but the file still exists! If you changed the order, you'd have the opposite problem: the file might get deleted, but then the row stays because of a temporary connection error to the database.

If you're worried about this, use a Doctrine transaction to wrap *all* of this logic. If the file *was* successfully deleted, commit the transaction. If not, roll it back so both the file and row stay.

Anyways, what should this endpoint return? Well... how about... nothing! Return a new Response() - the one from HttpFoundation - with null as the content and a 204 status code. 204 means: the operation was successful but I have nothing else to say!

```
133 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 12
13  use Symfony\Component\HttpFoundation\Response;
... lines 14 - 19
20  class ArticleReferenceAdminController extends BaseController
21  {
... lines 22 - 119
120     public function deleteArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper, EntityManagerInterface $en
121     {
... lines 122 - 129
130         return new Response(null, 204);
131     }
132 }
```

Hooking up the JavaScript

That's it! That is a *nice* endpoint! Head back to our JavaScript so we can put this all together. First, down in the render() function, add a little trash icon next to the download link. I'll make this a button... just because semantically, it requires a DELETE request, so it's not something the user can click without JavaScript. Give it a js-reference-delete class so we can find it, some styling classes and, inside, we'll use FontAwesome for the icon.

```
117 lines | public/js/admin_article_form.js
... lines 1 - 34
35  class ReferenceList
36  {
... lines 37 - 74
75      render() {
76          const itemsHtml = this.references.map(reference => {
77              return `
78  <li class="list-group-item d-flex justify-content-between align-items-center" data-id="${reference.id}">
... lines 79 - 80
81      <span>
... line 82
83          <button class="js-reference-delete btn btn-link"><span class="fa fa-trash"></span></button>
84      </span>
85  </li>
86  `
87          });
... lines 88 - 89
90      }
91  }
... lines 92 - 117
```

Copy that class name and go back up to the constructor. Here say this.$element.on('click') and then pass .js-reference-delete. This is called a delegate event handler. It's handy because it allows us to attach a listener to any .js-reference-delete elements, even if they're added to the HTML *after* this line is executed. For the callback, I'll pass an ES6 arrow function so that the this variable inside is still my ReferenceList object. Call a new method: this.handleReferenceDelete() and pass it the event object.

```
117 lines | public/js/admin_article_form.js
... lines 1 - 34
class ReferenceList
{
    constructor($element) {
... lines 38 - 41
        this.$element.on('click', '.js-reference-delete', (event) => {
            this.handleReferenceDelete(event);
        });
... lines 45 - 51
    }
... lines 53 - 90
}
... lines 92 - 117
```

Copy that name, head down, and paste to create that. Inside, we need to do two things: make the AJAX request to delete the item from the server *and* remove the reference from the references array and call this.render() so it disappears.

Start with const $li =. I'm going to use the button that was just clicked to find the <li> element that's around everything - you'll see why in a second. So, const $li = $(event.currentTarget) to get the button that was clicked, then .closest('.list-group-item').

```
117 lines | public/js/admin_article_form.js
... lines 1 - 34
class ReferenceList
{
... lines 37 - 58
    handleReferenceDelete(event) {
        const $li = $(event.currentTarget).closest('.list-group-item');
... lines 61 - 72
    }
... lines 74 - 90
}
... lines 92 - 117
```

To create the URL for the DELETE request, I need the id of this specific article reference. To get that, add a new data-id attribute on the li set to ${reference.id}. I'm adding this here instead of directly on the button so that we could re-use it for other behaviors.

Now we can say const id = $li.data('id') and $li.addClass('disabled') to make it look like we're doing something during the AJAX call.

```
117 lines | public/js/admin_article_form.js
... lines 1 - 34
class ReferenceList
{
... lines 37 - 58
    handleReferenceDelete(event) {
        const $li = $(event.currentTarget).closest('.list-group-item');
        const id = $li.data('id');
        $li.addClass('disabled');
... lines 63 - 72
    }
... lines 74 - 90
}
... lines 92 - 117
```

Make that with $.ajax() with url set to '/admin/article/references/'+id and method: 'DELETE'.

```
117 lines | public/js/admin_article_form.js
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 58
59     handleReferenceDelete(event) {
60         const $li = $(event.currentTarget).closest('.list-group-item');
61         const id = $li.data('id');
62         $li.addClass('disabled');
... line 63
64         $.ajax({
65             url: '/admin/article/references/'+id,
66             method: 'DELETE'
... lines 67 - 71
72         });
73     }
... lines 74 - 90
91 }
... lines 92 - 117
```

To handle success, chain a .then() on this with another arrow function.

```
117 lines | public/js/admin_article_form.js
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 58
59     handleReferenceDelete(event) {
60         const $li = $(event.currentTarget).closest('.list-group-item');
61         const id = $li.data('id');
62         $li.addClass('disabled');
... line 63
64         $.ajax({
65             url: '/admin/article/references/'+id,
66             method: 'DELETE'
67         }).then(() => {
... lines 68 - 71
72         });
73     }
... lines 74 - 90
91 }
... lines 92 - 117
```

Now that the article reference has been deleted from the server, let's remove it from this.references. A nice way to do that is by saying: this.references = this.references.filter() and passing this an arrow function with return reference.id !== id.

117 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 58
59     handleReferenceDelete(event) {
60         const $li = $(event.currentTarget).closest('.list-group-item');
61         const id = $li.data('id');
62         $li.addClass('disabled');
... line 63
64         $.ajax({
65             url: '/admin/article/references/'+id,
66             method: 'DELETE'
67         }).then(() => {
68             this.references = this.references.filter(reference => {
69                 return reference.id !== id;
70             });
... line 71
72         });
73     }
... lines 74 - 90
91 }
... lines 92 - 117
```

This callback function will be called once for each item in the array. If the function returns true, that item will be put into the new references variable. If it returns false, it won't be. The end effect is that we get an identical array, except *without* the reference that was just deleted.

After this, call this.render().

117 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 58
59     handleReferenceDelete(event) {
60         const $li = $(event.currentTarget).closest('.list-group-item');
61         const id = $li.data('id');
62         $li.addClass('disabled');
63
64         $.ajax({
65             url: '/admin/article/references/'+id,
66             method: 'DELETE'
67         }).then(() => {
68             this.references = this.references.filter(reference => {
69                 return reference.id !== id;
70             });
71             this.render();
72         });
73     }
... lines 74 - 90
91 }
... lines 92 - 117
```

Let's try it! Refresh and... cool! There's our delete icon - it looks a little weird, but we'll fix that in a minute. Let's see, in var/uploads we have a rocket.jpeg file. Let's delete that one. Ha! It disappeared! The 204 status code looks good and... the

file is gone!

It's strange when things work on the first try!

Alignment Tweak

While we're here, let's fix this alignment issue - it's weirding me out. Down in the render() function, add a few Bootstrap classes to the download link and make the delete button smaller.

Try that. Better... but it's still just a *touch* off. Add vertical-align: middle to the download icon. It's subtle but... yep - the buttons are lined up now.

117 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 74
75     render() {
76         const itemsHtml = this.references.map(reference => {
77             return `
78 <li class="list-group-item d-flex justify-content-between align-items-center" data-id="${reference.id}">
... lines 79 - 80
81     <span>
82         <a href="/admin/article/references/${reference.id}/download" class="btn btn-link btn-sm"><span class="fa fa-download" style="ve
83         <button class="js-reference-delete btn btn-link btn-sm"><span class="fa fa-trash"></span></button>
84     </span>
85 </li>
86 `
87         });
... lines 88 - 89
90     }
91 }
... lines 92 - 117
```

Next: our users are *begging* for another feature: the ability to rename the file after it's been uploaded.

Chapter 29: Edit Endpoint & Deserialization

I want more fancy! Seriously, we're going to add pretty much *everything* we can think of to make this a sweet, flexible, sort of, file "gallery". What about allowing the user to *update* a file reference?

Okay, well, we're not going to allow the user to update the *actual* attached file, there's just no point. Want to upload a newer version of a file? Just delete the old one and upload the new one. Feature, done!

But we *could* allow them to change the filename. Remember: this is the *original* filename. And, yea, if they uploaded a file called astronaut.jpeg, it would be totally cool to let them change that to something else after. Let's do it!

The Update API Endpoint

Let's keep thinking about our ArticleReference routes as a set of nice, RESTful API endpoints. We already have an endpoint to create and delete an ArticleReference. This will be an endpoint to *edit* a reference... except that the only field the user will be allowed to edit will be the originalFilename.

Copy the beginning of our delete endpoint, paste, close it up and we'll call this updateArticleReference(). Keep the same URL, but change the route name to admin_article_update_reference - it should be *reference*, not *references*, let's fix that in both places - I don't think I'm referencing that route name anywhere. And instead of methods={"DELETE"}, use methods={"PUT"}.

142 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 19
20 class ArticleReferenceAdminController extends BaseController
21 {
... lines 22 - 132
133     /**
134      * @Route("/admin/article/references/{id}", name="admin_article_update_reference", methods={"PUT"})
135      */
136     public function updateArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper, EntityManagerInterface $e
137     {
138         $article = $reference->getArticle();
139         $this->denyAccessUnlessGranted('MANAGE', $article);
140     }
141 }
```

Cool! Let's think about how we want this endpoint to work. First, our JavaScript will send a request with a JSON body that contains the data that should be updated on the ArticleReference. In this case, the data will have only one field: originalFilename.

Deserializing JSON

So far, we've been using $this->json() to turn an object or multiple objects into JSON. This uses Symfony's serializer behind the scenes. Now we're going to use the serializer to do the opposite: to turn JSON *back* into an ArticleReference object. That's called deserialization and... it's... pretty freakin' awesome!

Let's add a few more arguments: SerializerInterface $serializer and Request - the one from HttpFoundation - so we can read the raw JSON body.

```
165 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 15
16 use Symfony\Component\Serializer\SerializerInterface;
... lines 17 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 162
163     }
164 }
```

To automagically turn the JSON into an ArticleReference object, say $serializer->deserialize(). The serializer only has these two methods: serialize() and deserialize().

```
165 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 141
142         $serializer->deserialize(
... lines 143 - 149
150         );
... lines 151 - 162
163     }
164 }
```

This method needs the raw JSON from the request - that's $request->getContent(), what *type* of object to turn this into - ArticleReference::class - and the *format* of the data: json, because the serializer can also handle XML or any crazy format you dream up.

```
165 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 141
142         $serializer->deserialize(
143             $request->getContent(),
144             ArticleReference::class,
145             'json',
... lines 146 - 149
150         );
... lines 151 - 162
163     }
164 }
```

Finally, we can pass some options - called "context". By default, deserialize() will always create a *new* object... but we want it

to update an *existing* object. To do that, pass an option called object_to_populate set to $reference.

```
165 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 141
142         $serializer->deserialize(
143             $request->getContent(),
144             ArticleReference::class,
145             'json',
146             [
147                 'object_to_populate' => $reference,
... line 148
149             ]
150         );
... lines 151 - 162
163     }
164 }
```

Oh, and when we've been *serializing*, we've been passing a groups option, which tells the serializer to put the properties from the "main" group into the JSON. We can do the same thing here: we don't want a clever user to be able to update the internal filename or the id: we need to restrict their power to changing the originalFilename.

Above $originalFilename, turn the groups value into an array and give it a second group: input.

```
102 lines | src/Entity/ArticleReference.php
... lines 1 - 11
12 class ArticleReference
13 {
... lines 14 - 33
34     /**
... line 35
36      * @Groups({"main", "input"})
37      */
38     private $originalFilename;
... lines 39 - 100
101 }
```

In the controller, *way* back down here, set groups to input. So if any other fields or passed, they'll just be ignored.

```
165 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 141
142         $serializer->deserialize(
143             $request->getContent(),
144             ArticleReference::class,
145             'json',
146             [
147                 'object_to_populate' => $reference,
148                 'groups' => ['input']
149             ]
150         );
... lines 151 - 162
163     }
164 }
```

And... yea, that's it! We *do* need to think about validation - but, pff, we'll handle that later - like in 2 minutes. Right now we can celebrate with $entityManager->persist($reference)... which we technically don't need because this isn't a new object, but I usually add it, and $entityManager->flush().

```
165 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 151
152         $entityManager->persist($reference);
153         $entityManager->flush();
... lines 154 - 162
163     }
164 }
```

What should we return? Typically after you edit a resource in an API, we return that resource again. Scroll all the way up to our upload endpoint and steal the JSON logic. We could also refactor this into a private method if we wanted to avoid duplication. Back down in *our* method, paste, rename the variable to $reference and use 200 as the status code: we're not *creating* a resource in this case.

165 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 154
155         return $this->json(
156             $reference,
157             200,
158             [],
159             [
160                 'groups' => ['main']
161             ]
162         );
163     }
164 }
```

Ok, that endpoint should be good! Or at least, we're ready to hook up our JavaScript so we can find out if it explodes when we use it! That's next.

Chapter 30: JavaScript for Editing a Reference

To make this all work, but to avoid going *totally* insane and coding JavaScript for the next 30 minutes, we're going to turn the printed string into an input text body and, on "blur" - so when we click *away* from it, we'll make an AJAX request to save the new filename.

Let's copy the original filename code and replace it with <input type="text" and value=" that original filename stuff. Let's also add two classes: one from Bootstrap to make things look nice and another - js-edit-filename - so that we can *find* this field in JavaScript. Oh, one more detail: add a style attribute with width: auto - just another styling thing.

136 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 92
93
94     render() {
95         const itemsHtml = this.references.map(reference => {
96             return `
97 <li class="list-group-item d-flex justify-content-between align-items-center" data-id="${reference.id}">
98     <input type="text" value="${reference.originalFilename}" class="form-control js-edit-filename" style="width: auto;">
... lines 99 - 103
104 </li>
105 `
106         });
... lines 107 - 108
109     }
110 }
... lines 111 - 136
```

Next: copy the js- class name and head back up to the constructor. We're going to do the same thing we did with our delete link: this.$element.on('blur'), this time with .js-edit-filename and then our arrow function. Inside that, call a new function: this.handleReferenceEditFilename() and pass that the event.

136 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
37     constructor($element) {
... lines 38 - 45
46         this.$element.on('blur', '.js-edit-filename', (event) => {
47             this.handleReferenceEditFilename(event);
48         });
... lines 49 - 55
56     }
... lines 57 - 109
110 }
... lines 111 - 136
```

Keep going: copy the method name, scroll down a bit, and create that function, which will accept an event object. Let's also steal the first two lines from handleReferenceDelete(): we're going to start the exact same way.

136 lines | public/js/admin_article_form.js

```
... lines 1 - 34
class ReferenceList
{
... lines 37 - 78
    handleReferenceEditFilename(event) {
        const $li = $(event.currentTarget).closest('.list-group-item');
        const id = $li.data('id');
... lines 82 - 91
    }
... lines 93 - 109
}
... lines 111 - 136
```

Heck, we're going to make an AJAX request to the same URL! Just with the PUT method insteadof DELETE.

When we send that AJAX request, we're only going to send one piece of data: the originalFilename that's in the text box. But I want you to pretend that we're allowing *multiple* fields to be updated on the reference. So, more abstractly, what we were *really* want to do is find the reference that's being updated from inside this.references, change the originalFilename data on it, JSON-encode that *entire* object, and send it to the endpoint.

If that doesn't make sense yet, don't worry. To find the reference object that's being updated right now, say const reference = this.references.find() and pass this an arrow function with a reference argument. Inside, return reference.id === id.

136 lines | public/js/admin_article_form.js

```
... lines 1 - 34
class ReferenceList
{
... lines 37 - 78
    handleReferenceEditFilename(event) {
        const $li = $(event.currentTarget).closest('.list-group-item');
        const id = $li.data('id');
        const reference = this.references.find(reference => {
            return reference.id === id;
        });
... lines 85 - 91
    }
... lines 93 - 109
}
... lines 111 - 136
```

This loops over all the references and returns the first one it finds that matches the id... which *should* only be one. Now change the originalFilename property to $(event.currentTarget) - that will give us the input element - .val().

```
136 lines | public/js/admin_article_form.js
... lines 1 - 34
class ReferenceList
{
... lines 37 - 78
    handleReferenceEditFilename(event) {
        const $li = $(event.currentTarget).closest('.list-group-item');
        const id = $li.data('id');
        const reference = this.references.find(reference => {
            return reference.id === id;
        });
        reference.originalFilename = $(event.currentTarget).val();
... lines 86 - 91
    }
... lines 93 - 109
}
... lines 111 - 136
```

Ok! We're ready to send the AJAX request! Copy the first-half of the AJAX call from the delete function, remove the .then() stuff, change the method to PUT and, for the data, just pass reference.

```
136 lines | public/js/admin_article_form.js
... lines 1 - 34
class ReferenceList
{
... lines 37 - 78
    handleReferenceEditFilename(event) {
        const $li = $(event.currentTarget).closest('.list-group-item');
        const id = $li.data('id');
        const reference = this.references.find(reference => {
            return reference.id === id;
        });
        reference.originalFilename = $(event.currentTarget).val();

        $.ajax({
            url: '/admin/article/references/'+id,
            method: 'PUT',
            data: reference
        });
    }
... lines 93 - 109
}
... lines 111 - 136
```

There *is* a small problem with this - so if you see it, hang on! But, the idea is cool: we're sending up *all* of the reference data. And yes, this *will* send more fields than we need, but that's ok! The deserializer just ignores that extra stuff.

Testing time! Refresh the whole page. Oh wow - we have an extra < sign! As cool as that looks, let's scroll down to render and... there it is - remove that.

Refresh again. Let's tweak the filename and then click off to trigger the "blur". Uh oh!

> Cannot set property originalFilename of undefined.

Hmm. Look back at our code: for some reason it's not finding our reference. Oh, duh: return referenced.id === id.

Ok, let's see if I've *finally* got everything right. Refresh, add a dash to the filename, click off and... 500 error! That's progress! Open the profiler for that request in a new tab. Ok: a "Syntax Error" coming from a JsonDecode class. Oh, and look at the

data that's passed to the deserialize() function! That's not JSON!

Silly mistake. When we set the data key to the reference object, jQuery doesn't send up that data as JSON, it uses the standard "form submit" format. *We* want JSON.stringify(reference).

```
136 lines | public/js/admin_article_form.js
... lines 1 - 34
35 class ReferenceList
36 {
... lines 37 - 78
79     handleReferenceEditFilename(event) {
... lines 80 - 86
87         $.ajax({
... lines 88 - 89
90             data: JSON.stringify(reference)
91         });
92     }
... lines 93 - 109
110 }
... lines 111 - 136
```

I think we've got it this time. Refresh, tweak the filename, click off and... no errors! Check out the network tab. Yeah 200! The response returns the updated originalFilename and, if you scroll down to the request body... cool! You can see the raw JSON that was sent up.

Validation

The *last* thing we need to do is... add validation. I know, it's always that annoying last detail once you've got the "happy" path working perfectly. But, right now, we could leave the filename *completely* blank and our system would be ok with that. Well ya know what? I am totally *not* ok with that!

Ultimately, our endpoint modifies the ArticleReference object and *that* is what we should validate. Above the originalFilename field, add @NotBlank() and let's also use @Length(). The length can be 255 in the database, but let's use max=100.

```
105 lines | src/Entity/ArticleReference.php
... lines 1 - 7
8 use Symfony\Component\Validator\Constraints as Assert;
... lines 9 - 12
13 class ArticleReference
14 {
... lines 15 - 34
35     /**
... lines 36 - 37
38      * @Assert\NotBlank()
39      * @Assert\Length(max=100)
40      */
41     private $originalFilename;
... lines 42 - 103
104 }
```

Then, inside our endpoint, there's no form here, but that's fine. Add the ValidatorInterface $validator argument. And right after we update the object with the serializer, add $violations = $validator->validate() and pass it the $reference object. Then if $violations->count() > 0, return $this->json($violations, 400).

```
170 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 136
137     public function updateArticleReference(ArticleReference $reference, EntityManagerInterface $entityManager, SerializerInterface $
138     {
... lines 139 - 151
152         $violations = $validator->validate($reference);
153         if ($violations->count() > 0) {
154             return $this->json($violations, 400);
155         }
... lines 156 - 167
168     }
169 }
```

We're actually *not* going to handle that in JavaScript - I'll leave rendering the errors up to you - you could highlight the element in red and print the error below... whatever you want.

But let's at *least* make sure it works. Clear out the filename, hit tab to blur and... there it is! A 400 error with our beautiful error response. To handle this in JavaScript, you'll chain a .catch() onto the end of the AJAX call and then do whatever you want.

Ok, what else can we add to our upload widget? How about the ability to reorder the list. That's next.

Chapter 31: Reordering the Files

What else do you want to add to our file gallery widget? How about allowing them to be reordered? Yea, that isn't *really* related to uploading either, but a lot of people asked for it... so, let's do it!

Adding the position Field

To start, the ArticleReference entity needs a field that can store its order in the list. Find your terminal and run:

```
$ php bin/console make:entity
```

Update ArticleReference and add one new field position. This is an integer and make it not nullable. Cool!

Go find the property... there it is. Make it default to 0: until the user decides to reorder stuff, setting them all to 0 is fine.

```
122 lines | src/Entity/ArticleReference.php
    ... lines 1 - 12
13  class ArticleReference
14  {
    ... lines 15 - 48
49      /**
50       * @ORM\Column(type="integer")
51       */
52      private $position = 0;
    ... lines 53 - 120
121 }
```

Create the migration with the usual:

```
$ php bin/console make:migration
```

and go to the src/Migrations directory so we can make sure it doesn't contain any surprises. Looks perfect! Close that and run:

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

Adding the Sortable Library

Ok, the database is ready! For the frontend, there are a *ton* of libraries that can help you sort and reorder stuff. I'm going to use one called Sortable. It's got a lot of support and *tons* of options. We'll need a few of them.

If you're using Webpack Encore, I'd recommend installing this via yarn and then importing the library when you need it. Because we're not, I'll Google for "sortablejs cdn". It's this one, from jsdelivr - the first is a different library. It turns out "Sortable"... is a pretty generic name.

Click to copy the HTML+SRI script tag, then go find the edit template. Scroll down to the JavaScript block and... paste!

```
43 lines | templates/article_admin/edit.html.twig
... lines 1 - 35
36 {% block javascripts %}
... lines 37 - 39
40     <script src="https://cdn.jsdelivr.net/npm/sortablejs@1.8.3/Sortable.min.js" integrity="sha256-uNITVqEk9HNQeW6mAAm2PJwFX2gN
... line 41
42 {% endblock %}
```

Hey! We *now* have a global Sortable variable.

Integrating Sortable

Next, open admin_article_form.js and scroll up to the constructor so we can start using this. Here's the plan: we're going to make each element - each "row" - sortable. And when we finish dragging, we'll send an AJAX request to save the new positions.

Add this.sortable = Sortable.create(). We're storing the *instance* of our new sortable object onto a property because we'll need it later. Pass this the *parent* of the elements that should be sortable. So... hmm... in our case, we want to attach sortable to the <ul> element that's around everything. Fortunately, that's *exactly* what this.$element represents! So we can say this.$element, and, this actually wants a raw HTMLElement, not a jQuery object, so add [0].

```
137 lines | public/js/admin_article_form.js
... lines 1 - 34
35 class ReferenceList
36 {
37     constructor($element) {
... line 38
39         this.sortable = Sortable.create(this.$element[0]);
... lines 40 - 56
57     }
... lines 58 - 110
111 }
... lines 112 - 137
```

Give it a test! Refresh... and grab... sweet! When we finish ordering, nothing *saves* yet, but we'll get there.

Making it Nicer!

Before we do, I think we can make this a bit nicer. Pass a second argument to create(): an array of options. Pass one called handle set to .drag-handle.

```
141 lines | public/js/admin_article_form.js
... lines 1 - 34
35 class ReferenceList
36 {
37     constructor($element) {
... line 38
39         this.sortable = Sortable.create(this.$element[0], {
40             handle: '.drag-handle',
... line 41
42         });
... lines 43 - 59
60     }
... lines 61 - 114
115 }
... lines 116 - 141
```

With this, instead of being able to grab *anywhere* to start sorting, we'll only be able to grab elements with this class. Down in

render, how about, *before* the text field, add , and fa and fa-reorder.

141 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35  class ReferenceList
36  {
... lines 37 - 97
98      render() {
99          const itemsHtml = this.references.map(reference => {
100             return `
101 <li class="list-group-item d-flex justify-content-between align-items-center" data-id="${reference.id}">
102     <span class="drag-handle fa fa-reorder"></span>
... lines 103 - 108
109 </li>
110 `
111         });
... lines 112 - 113
114     }
115 }
... lines 116 - 141
```

Oh, and *while* we're making this fancy, add animation: 150... it just makes it look cooler. Try it! There's our drag handle and... nice - it's a bit smoother.

141 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35  class ReferenceList
36  {
37      constructor($element) {
... line 38
39          this.sortable = Sortable.create(this.$element[0], {
40              handle: '.drag-handle',
41              animation: 150,
42          });
... lines 43 - 59
60      }
... lines 61 - 114
115 }
... lines 116 - 141
```

This library doesn't require *any* CSS, which is cool... but we *can* make it a little nicer by adding some. In the public/css/ directory, open styles.css. This is a nice, boring, normal CSS file that's included on every page.

Add .sortable-ghost. When you're dragging, Sortable adds this class to *where* the element will be added if you stop sorting at that moment. Give this a background color. Oh, and also, give the drag-handle a cursor: grab.

258 lines | public/css/styles.css

```
... lines 1 - 251
252 /* Sortable */
253 .sortable-ghost {
254     background-color: lightblue;
255 }
256 .drag-handle {
257     cursor: grab;
258 }
```

Try it one more time - do a force refresh if it doesn't show up at first. And... there's the blue background!

Ok, the database is setup and the frontend is ready. Next, let's add an API endpoint to save the positions and make sure they're rendered in the right order.

Chapter 32: Reordering Endpoint & AJAX

Let's upload *all* of these files. How nice is that? One fails because it's the wrong type and another fails because it's too big. But we get nice errors and all the rest worked. *And* this gives us a *lot* more to play with for reordering!

Getting the Sorted Ids

To make an AJAX call when we finishing dragging, add a new option: onEnd set to an arrow function. Inside console.log(this.sortable) - that's the sortable object we stored earlier .toArray().

144 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
37     constructor($element) {
... line 38
39         this.sortable = Sortable.create(this.$element[0], {
... lines 40 - 41
42             onEnd: () => {
43                 console.log(this.sortable.toArray());
44             }
45         });
... lines 46 - 62
63     }
... lines 64 - 117
118 }
... lines 119 - 144
```

Check it out: refresh the page, drag one of these... and go look at the console. Woh! Those are the reference ids... in the right order! Try it again: move this one up and... yep! The id 11 just moved up a few spots.

But... how the heck is this working? How does sortable know what the ids are? Well, honestly... we got lucky. It knows thanks to the data-id attribute that we put on each li! We added that for our *own* JavaScript... but the Sortable library *also* knows to read that!

The Reorder Endpoint

This is amazing! This is the *exact* data we need to send to the server! Open up ArticleReferenceAdminController and find downloadArticleReference(). If you look closely, about half of the methods in this controller have an {id} route wildcard where the id is for an ArticleReference. Those endpoints are actions that operating on a single *item*. The other half of the endpoints, the ones on top, *also* have an {id} wildcard, but these are for the Article.

What about *our* new endpoint? We'll be reordering *all* of the references for one article... so it's a bit more like these ones on top. Copy this entire action for getting article references, change the name to reorderArticleReferences and put /reorder on the URL. Make this a method="POST" and name it admin_article_reorder_references.

```
186 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 93
94     /**
95      * @Route("/admin/article/{id}/references/reorder", methods="POST", name="admin_article_reorder_references")
96      * @IsGranted("MANAGE", subject="article")
97      */
98     public function reorderArticleReferences(Article $article)
99     {
100         return $this->json(
101             $article->getArticleReferences(),
102             200,
103             [],
104             [
105                 'groups' => ['main']
106             ]
107         );
108     }
... lines 109 - 184
185 }
```

If you're wondering about the URL or the method POST, well, this endpoint isn't very RESTful.. it doesn't fit into the nice create-read-update-delete model... and that's ok. Usually when I have a weird endpoint like this, I use POST.

Inside the method, here's the plan: our JavaScript will send a JSON body containing an array of the ids in the right order. This array exactly. Add the Request argument so we can get read that data and the EntityManagerInterface so we can save stuff.

```
200 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 97
98     public function reorderArticleReferences(Article $article, Request $request, EntityManagerInterface $entityManager)
99     {
... lines 100 - 121
122     }
... lines 123 - 198
199 }
```

To decode the JSON *this* time, it's so simple! I'm going to skip using Symfony's serializer. Say $orderedIds = json_decode() passing that $request->getContent() and true so it gives us an associative array.

```
200 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 97
98     public function reorderArticleReferences(Article $article, Request $request, EntityManagerInterface $entityManager)
99     {
100         $orderedIds = json_decode($request->getContent(), true);
... lines 101 - 121
122     }
... lines 123 - 198
199 }
```

Then, if orderedIds === false, something went wrong. Let's return this->json() and, to at least *somewhat* match the validation responses we've had so far, let's set a detail key to, how about, Invalid body with 400 for the status code.

200 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 97
98     public function reorderArticleReferences(Article $article, Request $request, EntityManagerInterface $entityManager)
99     {
100         $orderedIds = json_decode($request->getContent(), true);
101
102         if ($orderedIds === null) {
103             return $this->json(['detail' => 'Invalid body'], 400);
104         }
... lines 105 - 121
122     }
... lines 123 - 198
199 }
```

Using the Ordered Ids to Update the Database

Ok, cool: we've got the array of ids in the *new* order we want. Use this to say $orderedIds = array_flip($orderedIds). This deserves some explanation. The original array is a map from the position to the id - the keys are 0, 1, 2, 3 and so on. After the flip, we have a *very* handy array: the key is the *id* and the value is its new position.

200 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 97
98     public function reorderArticleReferences(Article $article, Request $request, EntityManagerInterface $entityManager)
99     {
100         $orderedIds = json_decode($request->getContent(), true);
101
102         if ($orderedIds === null) {
103             return $this->json(['detail' => 'Invalid body'], 400);
104         }
105
106         // from (position)=>(id) to (id)=>(position)
107         $orderedIds = array_flip($orderedIds);
... lines 108 - 121
122     }
... lines 123 - 198
199 }
```

To use this, foreach over $article->getArticleReferences() as $reference. And inside, $reference->setPosition() passing this $orderedIds[$reference->getId()] to look up the new position.

```
200 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 97
98     public function reorderArticleReferences(Article $article, Request $request, EntityManagerInterface $entityManager)
99     {
100         $orderedIds = json_decode($request->getContent(), true);
101
102         if ($orderedIds === null) {
103             return $this->json(['detail' => 'Invalid body'], 400);
104         }
105
106         // from (position)=>(id) to (id)=>(position)
107         $orderedIds = array_flip($orderedIds);
108         foreach ($article->getArticleReferences() as $reference) {
109             $reference->setPosition($orderedIds[$reference->getId()]);
110         }
... lines 111 - 121
122     }
... lines 123 - 198
199 }
```

And yes, we *could* code more defensively - like checking to make sure each array key was actually sent. And I *would* do that if this were a public API that other people used, or if invalid data could cause some harm.

Anyways, at the bottom, save: $entityManager->flush().

```
200 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21 class ArticleReferenceAdminController extends BaseController
22 {
... lines 23 - 97
98     public function reorderArticleReferences(Article $article, Request $request, EntityManagerInterface $entityManager)
99     {
100         $orderedIds = json_decode($request->getContent(), true);
101
102         if ($orderedIds === null) {
103             return $this->json(['detail' => 'Invalid body'], 400);
104         }
105
106         // from (position)=>(id) to (id)=>(position)
107         $orderedIds = array_flip($orderedIds);
108         foreach ($article->getArticleReferences() as $reference) {
109             $reference->setPosition($orderedIds[$reference->getId()]);
110         }
111
112         $entityManager->flush();
... lines 113 - 121
122     }
... lines 123 - 198
199 }
```

Sending the AJAX Request

Ok, let's hook up the JavaScript! Back in admin_article_form.js, scroll up... let's see - find the onEnd() of sortable. Say $.ajax()

and give this the url key. For the URL, remember, the ul element has a data-url attribute, which is the path to the admin_article_list_references route, so /admin/article/{id}/references. Not by accident, the URL that *we* want is that plus /reorder.

148 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
37     constructor($element) {
... lines 38 - 41
42             onEnd: () => {
43                 $.ajax({
... lines 44 - 46
47                 });
48             }
49         });
... lines 50 - 66
67     }
... lines 68 - 121
122 }
... lines 123 - 148
```

So let's do a *little* bit of code re-use... and a little bit of hardcoding: in general, I don't worry *too* much about hardcoding URLs in JavaScript. Copy this.$element.data('url') from below, paste, and add /reorder. Then, method set to POST and data set to JSON.stringify(this.sortable.toArray()).

148 lines | public/js/admin_article_form.js

```
... lines 1 - 34
35 class ReferenceList
36 {
37     constructor($element) {
... lines 38 - 41
42             onEnd: () => {
43                 $.ajax({
44                     url: this.$element.data('url')+'/reorder',
45                     method: 'POST',
46                     data: JSON.stringify(this.sortable.toArray())
47                 });
48             }
49         });
... lines 50 - 66
67     }
... lines 68 - 121
122 }
... lines 123 - 148
```

Ok, let's do this! Move over and refresh. No errors yet... Move "astronaut-1.jpg" down two spots and... hey! A 200 status code on that AJAX request! That's a good sign. Refresh and... aw! It's right back up on top!

Changing the Endpoint Order

Oh wait... the problem is that we're not *rendering* the list correctly! This list loads by making an Ajax request. In the controller... here's the endpoint: getArticleReferences(). And it gets the data from $article->getArticleReferences(). The *problem* is that this method doesn't know that it should order the reference's by position.

Open up the Article entity and, above $articleReferences, add @ORM\OrderBy({"position"="ASC"}).

325 lines | src/Entity/Article.php

```
... lines 1 - 18
19 class Article
20 {
... lines 21 - 89
90     /**
91      * @ORM\OneToMany(targetEntity="App\Entity\ArticleReference", mappedBy="article")
92      * @ORM\OrderBy({"position"="ASC"})
93      */
94     private $articleReferences;
... lines 95 - 323
324 }
```

Let's go check out the endpoint: I'll click to open the URL in a new tab. Woohoo! astronaut-1.jpg is *third*! Refresh the main page. Boom! The astronaut is right were we sorted it. Let's move it down a bit further... move the Symfony Best Practices up from the bottom and refresh. The sorting sticks. Awesome!

Next, instead of saving the uploaded files locally, let's upload them to AWS S3.

Chapter 33: Configuring S3 Bucket & IAM User

Friends, I think it's finally time to store the uploaded files up... in the cloud. We're going to use Amazon S3. But thanks to Flysystem, we could easily use a different service - they have a *bunch* of adapters. Google again for OneupFlysystemBundle... and click into their docs so we can see how to implement the s3 adapter. Search for S3 and... there it is.

Configuring the AWS S3 Adapter

The first thing we need is this aws/aws-sdk-php package. Copy that, move over to your terminal and run:

```
$ composer require aws/aws-sdk-php
```

Creating the S3 Bucket

While we're waiting for that, let's create the S3 bucket that will store our stuff! I'm already logged into the S3 section of AWS. Click "Create bucket" and let's call it sfcasts-spacebar. Choose whatever region makes sense for you - but remember that, because you'll need it later.

On the next screen, if you need encryption or logging or any of these things, check them. But we'll just click next again to get to permissions. There *are* a few things we need to do here. First, uncheck the two top boxes for "Block new public ACLs" and "Remove public access granted through public ACLs". By unchecking these boxes, we can now have private files *and* public files all in the same bucket. Click "Next" again and then "Create bucket".

IAM Permissions

Awesome! Bucket done! To be able to actually *access* this bucket... I'm going to open an new tab for the IAM service. Click "Users" and add a new user. Let's call it: sfcasts-spacebar-s3-access.

Okay. Check yes for "programmatic access", but don't check console access. This user will exist *solely* so we can use its credentials in our app to talk to S3.

For permissions, this is *always* the tricky part, at least for me. There are a lot of existing "policies" that can grant different permissions to different services... I'm going to open another tab to IAM and click to create a new policy.

There's a builder to help create the policy... or you can click the JSON tab to do it yourself. So... what do we put here? Fortunately, Flysystem has our back. In its docs for AWS S3, scroll down and... nice! It gives us the IAM permissions we need! Copy that, go back, and paste. Tweak the bucket name to be *our* bucket name. Let's see... it's sfcasts-spacebar. Back on the policy, paste that in both spots.

This policy basically gives the new user full access to this specific bucket. Click "Review policy" and give it a name, how about sfcasts-spacebar-full-s3-bucket-access. Ok, create policy!

With that done, close that tab and go back to the original IAM tab where we're creating our new user. Click the little refresh button and search for sfcasts. The second policy was from me testing this earlier. Check the first box and hit "Next". Skip the tags... looks good... and create user!

Congrats! The hardest part is over! This gives us two things we need: a key and a secret. Next: let's set these as environment variables in our app and configure Flysystem to talk to S3!

Chapter 34: Flysystem & S3

With our key & secret in hand, and this *unescapable* feeling of power that they're giving us, let's hook up Flysystem to use an S3 adapter. Oh, first, go check on that library we were installing. Done! This is a PHP library for interacting with any AWS service, and it has *nothing* to do with Symfony or Flysystem. Copy the example configuration. Our *first* job is to register a service for this S3Client class that comes from that library.

Registering the S3Client Service

Let's close *all* these tabs so we can concentrate. Open config/services.yaml and, at the bottom, paste that config! But I'm going to simplify this: copy the class name, remove it, and paste *that* as the service id. Why? First, because, when possible, it's just easier to use the class name as the service id instead of inventing a string id. And second, this will allow us to *autowire* the S3Client service into any of our services or controllers. We won't need that for what we're doing, but it's nice.

```
62 lines | config/services.yaml
... lines 1 - 53
54      Aws\S3\S3Client:
55          arguments:
56              -
57                  version: '2006-03-01' # or 'latest'
58                  region: "region-id" # 'eu-central-1' for example
59                  credentials:
60                      key: "s3-key"
61                      secret: "s3-secret"
```

This takes just one argument: a big array of config. This *old* looking API version is actually still the most recent. For region, this depends on what region you chose for your bucket. Mine is us-east-1 because I selected Virginia. If you selected a different region, it won't work. Kidding! Just do some Googling to find the right region id.

What about the key and secret? *These* are the values IAM gave us after creating the user. But, we probably don't want to put their values right here and commit them to the repository. Instead, open the .env file and, inside of the custom vars section we created in a previous tutorial, let's invent two new environment variables AWS_S3_ACCESS_ID and AWS_S3_ACCESS_SECRET.

```
38 lines | .env
... lines 1 - 32
33  AWS_S3_ACCESS_ID=
34  AWS_S3_ACCESS_SECRET=
... lines 35 - 38
```

If you want, you *could* copy the values and put them directly into this file. But remember, the .env file *is* committed to your git repository... and you really *don't* want any secret value to be committed. Instead, create a new file at the root of your app called .env.local. This file is *also* read by Symfony and any values will *override* the ones in .env. It's also *ignored* from git via our .gitignore file.

Copy the two keys from .env and paste them here. And *now* we can grab the real values. Copy the id, paste, then show the secret, copy, and paste that.

Environment variables, set! To use them, head back to services.yaml. Replace the key with the special environment variable syntax: %env()% and inside, AW... go copy the name - AWS_S3_ACCESS_ID. Re-use that syntax for the secret: AWS_S3_ACCESS_SECRET.

```
62 lines | config/services.yaml
... lines 1 - 53
    Aws\S3\S3Client:
        arguments:
            -
                version: '2006-03-01'
                region: 'us-east-1'
                credentials:
                    key: '%env(AWS_S3_ACCESS_ID)%'
                    secret: '%env(AWS_S3_ACCESS_SECRET)%'
```

If you forget about Flysystem for a minute, we now have a *fully* functional S3Client service that we an autowire and use to do anything with our new bucket! The question *now* is: how can we tell Flysystem to use this?

The Flysystem AWS-S3-V3 Adapter

Go back to the OneupFlysystemBundle docs. Ok, so once the service is set up, we apparently need to go into the actual config for *this* bundle and change to a new adapter: awss3v3.

But to use *that*... hmm... it's not too obvious on this page. Go back to the Flysystem docs about S3 and scroll up. Here we go: the Flysystem S3 adapter is its own separate package. Copy this line, find your terminal and paste:

```
$ composer require league/flysystem-aws-s3-v3
```

Once that finishes... there. *Now* we can use this awss3v3 adapter. Open up config/packages/oneup_flysystem.yaml. Remove *all* that local config. Replace it with awss3v3:. The first sub-key this needs is: client, which points to the *service* id for the S3Client.

```
19 lines | config/packages/oneup_flysystem.yaml
... line 1
2 oneup_flysystem:
3     adapters:
4         public_uploads_adapter:
5             awss3v3:
... lines 6 - 19
```

Add client:, copy the service id, and paste.

```
19 lines | config/packages/oneup_flysystem.yaml
... line 1
2 oneup_flysystem:
3     adapters:
4         public_uploads_adapter:
5             awss3v3:
6                 client: Aws\S3\S3Client
... lines 7 - 19
```

The adapter also needs to know what S3 bucket it should be talking to. This is *also* something that you might *not* want to commit to your repository, because production will probably use a different bucket than when you're developing locally. So, back in our trusty .env file, add a third environment variable AWS_S3_ACCESS_BUCKET... well, I could just call this AWS_S3_BUCKET... I didn't *really* mean to keep that ACCESS part in there. But, no problem.

```
39 lines | .env
... lines 1 - 34
35 AWS_S3_BUCKET_NAME=
... lines 36 - 39
```

Just like before, copy that, duplicate it in .env.local and give it a real value, which... if you go back to S3, is sfcasts-spacebar. Paste that.

Finally, copy the new variable's name, open oneup_flysystem.yaml, and set bucket to %env(AWS_S3_ACCESS_BUCKET)%.

```
19 lines | config/packages/oneup_flysystem.yaml
... line 1
2  oneup_flysystem:
3      adapters:
4          public_uploads_adapter:
5              awss3v3:
6                  client: Aws\S3\S3Client
7                  bucket: '%env(AWS_S3_BUCKET_NAME)%'
... lines 8 - 19
```

That's it! What about the private_uploads_adapter? Well, temporarily, copy the config from the public adapter and paste it *exactly* down there. We're actually *not* going to need two filesystems anymore... but we'll talk about that soon.

```
19 lines | config/packages/oneup_flysystem.yaml
... line 1
2  oneup_flysystem:
3      adapters:
4          public_uploads_adapter:
5              awss3v3:
6                  client: Aws\S3\S3Client
7                  bucket: '%env(AWS_S3_BUCKET_NAME)%'
... lines 8 - 19
```

Oh, and don't forget the % sign at the end of the %env()% syntax! I *did* do that correctly in services.yaml.

Ok, I think we're ready! Both filesystems will use an awss3v3 adapter and each of *those* knows to us the S3Client service that's reading our key and secret. So... it should... just kinda work! The easiest way to find out is to reload the fixtures:

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

And yes, I *do* recommend using S3 when developing locally if that's what you're using on production. You *could* change the adapter to be the local adapter, but the less differences you have between your local environment & production, the better.

Fixtures done! Go and refresh the S3 page. Hey! We have an article_image directory and it's *full* of images! I think it worked! Go the homepage and... nothing works. That's because our paths are all still pointing at the *local* server - not at S3. Let's fix that next!

Chapter 35: S3 Asset Paths

Hey! Flysystem is *now* talking to S3! We know this because we can see the article_image directory and all the files inside of it. But when we went back to the homepage and refreshed, nothing worked!

Check out the image src URL: this is *definitely* wrong, because this *now* needs to point to S3 directly. But! Things get even *more* interesting if you go back to the S3 page and refresh. We have a media/ directory! And if you dig, there are the thumbnails! Woh!

This means that this thumbnail request *did* successfully get processed by a Symfony route and controller and it *did* correctly grab the *source* file from S3, thumbnail it and write it *back* to S3. That's freaking cool! And it worked because we already made LiipImagineBundle play nicely with Flysystem. We told the "loader" to use Flysystem - that's the thing that downloads the source image when it needs to thumbnail it - *and* the resolver to use Flysystem, which is the thing that actually saves the final image.

Correcting our Base URL

So if our system is working so awesomely... why don't the images show up? It's because of the *hostname* in front of the images: it's pointing at our local server, but it *should* be pointing at S3.

Click any of the images on S3. Here it is: every object in S3 has its own, public URL. Well actually, every object has a URL, but whether or not anyone can *access* that URL is another story. More on that later. I'm going to copy the very beginning of that, and then go open services.yaml. Earlier, we created a parameter called uploads_base_url. LiipImagineBundle uses this to prefix every URL that it renders. The current value includes 127.0.0.1:8000 because that's our SITE_BASE_URL environment variable value. That worked fine when things were stored locally... but not anymore!

Change this to https://s3.amazonaws.com/ and then our bucket name, which is already available as an environment variable: %env()%, then go copy AWS_S3_ACCESS_BUCKET, and paste.

61 lines | config/services.yaml

```
... lines 1 - 5
6  parameters:
... lines 7 - 8
9      uploads_base_url: 'https://s3.amazonaws.com/%env(AWS_S3_BUCKET_NAME)%'
... lines 10 - 61
```

This is our new base URL. What about the uploads_dir_name parameter? We're not using that at *all* anymore! Trash it.

Ok, let's try it! Refresh and... it actually works! I mean... of course, it works!

Correcting the Absolute URLs

There's one other path we need to fix: the absolute path to uploaded assets that are *not* thumbnailed. Open up src/Service/UploaderHelper.php and find the getPublicPath() method... there it is. *This* is a super-handy method: it allows us to get the full, public path to any uploaded file. This $publicAssetBaseUrl property... if you look on top, it comes from an argument called $uploadedAssetsBaseUrl. And in services.yaml, *that* is bound to the uploads_base_url parameter... that we just set!

There are a few layers, but it means that, in UploaderHelper the $publicAssetBaseUrl property is *now* the long S3 URL, which is *perfect*!

Head back to down getPublicPath(). *Even* before we changed uploads_base_url to point to S3, we were *already* setting it to the absolute URL to our domain... which means that *this* method already had a subtle bug!

Check it out: the original purpose of this code was to use $this->requestStackContext->getBasePath() to "correct" our paths in case our site was deployed under a sub-directory of a domain - like https://space.org/thespacebar. In that case, getBasePath() would equal thespacebar and would automatically prefix all of our URLs.

But ever since we started including the full domain in $publicAssetBaseUrl, this would create a broken URL! We could

remove this. Or, to make it *still* work if $publicAssetsBaseUrl happens to *not* include the domain, above this, set $fullPath = , copy the path part, replace that with $fullPath, and paste.

```
129 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 60
61     public function getPublicPath(string $path): string
62     {
63         $fullPath = $this->publicAssetBaseUrl.'/'.$path;
... lines 64 - 69
70         return $this->requestStackContext
71             ->getBasePath().$fullPath;
72     }
... lines 73 - 127
128 }
```

Then, if strpos($fullPath, '://') !== false, we know that $fullpath is already absolute. In that case, return it! That's what our code is doing. But if it's *not* absolute, we can keep prefixing the sub-directory.

```
129 lines | src/Service/UploaderHelper.php
... lines 1 - 12
13 class UploaderHelper
14 {
... lines 15 - 60
61     public function getPublicPath(string $path): string
62     {
63         $fullPath = $this->publicAssetBaseUrl.'/'.$path;
64         // if it's already absolute, just return
65         if (strpos($fullPath, '://') !== false) {
66             return $fullPath;
67         }
68
69         // needed if you deploy under a subdirectory
70         return $this->requestStackContext
71             ->getBasePath().$fullPath;
72     }
... lines 73 - 127
128 }
```

Hey! The files are uploading to S3 and our public paths are pointing to the new URLs *perfectly*. Next, we can simplify! Remember how we have one public filesystem and one private filesystem? With S3, we only need one.

Chapter 36: S3 & Private Object via ACLs

Head to /admin/article and log back in since we cleared our database recently: admin1@thespacebar.com, password engage. Edit any of the articles. Everything *should* work just fine: I'll select a few references to upload and... it works nicely. It *is* a bit slower now that the server is sending the files to S3 in the background, though that should be less noticeable once we're on production, especially if our server is also hosted on AWS.

So... can we download these? Try it! Yea, it works great! Open up ArticleReferenceAdminController and search for "download". Here it is: the download is handled by downloadArticleReference: we open a file stream from Flysystem - which is now from S3 - and stream that back to the user. By planning ahead and using Flysystem, when we switched to S3, *nothing* had to change!

But, there is *one* tiny problem. Back on the page, click the image. Access denied!? This *should* show us the full-size, original image. Hmm, the URL *looks* right. And, indeed! The problem isn't the path, the problem is with that file's *permissions* on S3.

Each file, or "object" on S3 can be set to be publicly accessible *or* private. File are *private* by default. In fact, the only reason that we can see the thumbnails, which are *also* stored in S3... is that LiipImagineBundle is smart enough to make sure that when it saves the files to S3, it saves them as *public*.

When an author uploads an article image, we need to do the same thing: we *do* want the original images to be public.

Giving the Images Public ACL

Head over to UploaderHelper and find uploadFile(). So far, we've been using the $isPublic argument to choose between the public and private filesystem objects. But when we changed to S3, I temporarily made these two filesystems *identical*. That wasn't on accident: with S3, we don't need two filesystems anymore! We can use the same one for both public and private files, and control the visibility on a file-by-file basis.

Check it out: remove the $filesystem = part and always use $this->filesystem.

131 lines | src/Service/UploaderHelper.php

```
... lines 1 - 13
14 class UploaderHelper
15 {
... lines 16 - 108
109         $newFilename = Urlizer::urlize(pathinfo($originalFilename, PATHINFO_FILENAME)).'-'.uniqid().'.'.$file->guessExtension();
110
111         $stream = fopen($file->getPathname(), 'r');
112         $result = $this->filesystem->writeStream(
... lines 113 - 117
118         );
... lines 119 - 129
130 }
```

To tell Flysystem that a file should be public or private, add a *third* argument to writeStream(): an array of options. The option we want is visibility. If $isPublic is true, use AdapterInterface - the one from Flysystem - ::VISIBILITY_PUBLIC. Otherwise, AdapterInterface::VISIBILITY_PRIVATE.

131 lines | src/Service/UploaderHelper.php

```
... lines 1 - 5
use League\Flysystem\AdapterInterface;
... lines 7 - 13
class UploaderHelper
{
... lines 16 - 108
        $newFilename = Urlizer::urlize(pathinfo($originalFilename, PATHINFO_FILENAME)).'-'.uniqid().'.'.$file->guessExtension();

        $stream = fopen($file->getPathname(), 'r');
        $result = $this->filesystem->writeStream(
            $directory.'/'.$newFilename,
            $stream,
            [
                'visibility' => $isPublic ? AdapterInterface::VISIBILITY_PUBLIC : AdapterInterface::VISIBILITY_PRIVATE
            ]
        );
... lines 119 - 129
}
```

Cool, right? That won't instantly change the permissions on the files we've already uploaded. So let's go upload a new one. Close the tab, select a new file, how about rocket.jpg and... update! The thumbnail still works and if you click it, yes! The original file is public!

By the way, you can see this setting when you're looking at the individual files in S3. Click back to the root of the bucket, find the rocket.jpg file and click it. Under "Permissions", here we go. *My* account has all permissions, of course, and under "Public Access", *Everyone* has "Read object" access.

Remove that Extra Private Filesystem!

Hey! This is awesome! Thanks to the object-by-object permissions super-power of S3, we don't need an extra "private" filesystem at all! We can do some serious cleanup! Start in config/packages/oneup_flysystem.yaml: remove the private_uploads_adapter and filesystem.

12 lines | config/packages/oneup_flysystem.yaml

```
# Read the documentation: https://github.com/1up-lab/OneupFlysystemBundle/tree/master/Resources/doc/index.md
oneup_flysystem:
    adapters:
        public_uploads_adapter:
            awss3v3:
                client: Aws\S3\S3Client
                bucket: '%env(AWS_S3_BUCKET_NAME)%'

    filesystems:
        public_uploads_filesystem:
            adapter: public_uploads_adapter
```

Next, in services.yaml, because there's no private_upload_filesystem anymore, remove that bind.

```
60 lines | config/services.yaml
... lines 1 - 10
services:
... line 12
    _defaults:
... lines 14 - 20
        bind:
            $markdownLogger: '@monolog.logger.markdown'
            $isDebug: '%kernel.debug%'
            $publicUploadsFilesystem: '@oneup_flysystem.public_uploads_filesystem_filesystem'
            $uploadedAssetsBaseUrl: '%uploads_base_url%'
... lines 26 - 60
```

That will break UploaderHelper because we're using that bind on top. But... we don't need it anymore! Remove the $privateFilesystem property and the $privateUploadFilesystem argument.

```
129 lines | src/Service/UploaderHelper.php
... lines 1 - 13
class UploaderHelper
{
... lines 16 - 18
    private $filesystem;


    private $requestStackContext;
... lines 23 - 27
    public function __construct(FilesystemInterface $publicUploadsFilesystem, RequestStackContext $requestStackContext, LoggerInt
    {
        $this->filesystem = $publicUploadsFilesystem;
        $this->requestStackContext = $requestStackContext;
        $this->logger = $logger;
        $this->publicAssetBaseUrl = $uploadedAssetsBaseUrl;
    }
... lines 35 - 127
}
```

But, we're still using that property in two places... the first is down in readStream. Now that everything is stored in *one* filesystem, delete that old code, remove the unused argument and always use $this->filesystem. Reading a stream is the same for public and private files.

```
125 lines | src/Service/UploaderHelper.php
... lines 1 - 13
class UploaderHelper
{
... lines 16 - 75
    public function readStream(string $path)
    {
        $resource = $this->filesystem->readStream($path);
... lines 79 - 84
    }
... lines 86 - 123
}
```

Repeat that in deleteFile(): delete the extra logic & argument, and use $this->filesystem *always*.

```
125 lines | src/Service/UploaderHelper.php
... lines 1 - 13
14  class UploaderHelper
15  {
... lines 16 - 86
87      public function deleteFile(string $path)
88      {
89          $result = $this->filesystem->delete($path);
... lines 90 - 93
94      }
... lines 95 - 123
124 }
```

Let's see... these two methods are called from ArticleReferenceAdminController. Take off that second argument for readStream().

```
200 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21  class ArticleReferenceAdminController extends BaseController
22  {
... lines 23 - 126
127     public function downloadArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper)
128     {
... lines 129 - 131
132         $response = new StreamedResponse(function() use ($reference, $uploaderHelper) {
133             $outputStream = fopen('php://output', 'wb');
134             $fileStream = $uploaderHelper->readStream($reference->getFilePath());
... lines 135 - 136
137         });
... lines 138 - 145
146     }
... lines 147 - 198
199 }
```

Then, search for "delete", and remove the second argument from deleteFile() as well.

```
200 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 20
21  class ArticleReferenceAdminController extends BaseController
22  {
... lines 23 - 150
151     public function deleteArticleReference(ArticleReference $reference, UploaderHelper $uploaderHelper, EntityManagerInterface $en
152     {
... lines 153 - 158
159         $uploaderHelper->deleteFile($reference->getFilePath());
... lines 160 - 161
162     }
... lines 163 - 198
199 }
```

That felt great! There's one more piece of cleanup we can do, it's optional, but nice. Using the word "public" in the adapter and filesystem isn't accurate anymore! Let's use uploads_adapter and uploads_filesystem.

```
12 lines | config/packages/oneup_flysystem.yaml
... line 1
2  oneup_flysystem:
3      adapters:
4          uploads_adapter:
... lines 5 - 8
9      filesystems:
10         uploads_filesystem:
11             adapter: uploads_adapter
```

We reference this in a few spots. In liip_imagine.yaml, take out the public_ in these two spots.

```
67 lines | config/packages/liip_imagine.yaml
1  liip_imagine:
... lines 2 - 5
6      loaders:
7          flysystem_loader:
8              flysystem:
9                  filesystem_service: oneup_flysystem.uploads_filesystem_filesystem
... lines 10 - 13
14     resolvers:
15         flysystem_resolver:
16             flysystem:
17                 filesystem_service: oneup_flysystem.uploads_filesystem_filesystem
... lines 18 - 67
```

And in services.yaml, update the "bind" in the same way. Hmm, and I think I'll change the argument name it's binding to: just $uploadFilesystem.

```
60 lines | config/services.yaml
... lines 1 - 10
11 services:
... line 12
13     _defaults:
... lines 14 - 20
21         bind:
... lines 22 - 23
24             $uploadsFilesystem: '@oneup_flysystem.uploads_filesystem_filesystem'
... lines 25 - 60
```

That *will* break UploaderHelper: we need to rename the argument there. But, let's just see what happens if we... "forget" to do that. Refresh the page:

> Unused binding $uploadFilesystem in S3Client.

This is that generic... and somewhat "inaccurate" error that says that we've configured a bind that's never used! The error is even better if we temporarily delete the bind entirely. Ah, here it is:

> Cannot autowire UploaderHelper: argument $publicUploadFilesystem references an interface, but that interface cannot be autowired.

This is saying: Hey! I don't know what you want me to send for this argument! Put the bind back, then, in UploaderHelper... here it is. Change the argument to match the bind: $uploadFilesystem.

125 lines | src/Service/UploaderHelper.php

```
... lines 1 - 13
14 class UploaderHelper
15 {
... lines 16 - 27
28     public function __construct(FilesystemInterface $uploadsFilesystem, RequestStackContext $requestStackContext, LoggerInterface
29     {
30         $this->filesystem = $uploadsFilesystem;
... lines 31 - 33
34     }
... lines 35 - 123
124 }
```

Oh, and there's one more thing we can get rid of! Do we need the public/uploads directory anymore? No! Delete it! And inside .gitignore, we can remove the custom public/uploads/ line we added.

So by putting things in S3... it simplifies things!

Next: now that I've been complimenting our S3 setup and saying how awesome it, I have a... confession to make! We've just introduced a hidden performance bug. Let's crush it!

Chapter 37: Cached S3 Filesystem For Thumbnails

Check this out: I'm going to turn off my Wifi! Gasp! What do you think will happen? I mean, other than I'm gonna miss all my Tweets and Instagrams! What will happen when I refresh? The page will load, but all the images will be broken, right?

In the name of science, I command us to try it!

Woh! An error!?

> Error executing ListObjects on https://sf-casts-spacebar ... Could not contact DNS servers.

What? Why is our Symfony app trying to connect to S3?

Here's the deal: on *every* request... for *every* thumbnail image that will be rendered, our Symfony app makes an API request to S3 to figure out if the image has already been thumbnailed or if it still needs to be. Specifically, LiipImagineBundle is doing this.

This bundle has two key concepts: the resolver and the loader. But there are actually *three* things that happen behind the scenes. First, every single time that we use |imagine_filter(), the resolver takes in that path and has to ask:

> Has this image already been thumbnailed?

And if you think about it, the *only* way for the resolver to figure this out is by making an API request to S3 to ask:

> Yo S3! Does this thumbnail file already exist?

If it *does* exist, LiipImagineBundle renders a URL that points directly to that image on S3. If not, it renders a URL to the Symfony route and controller that will use the loader to download the file and the resolver to save it back to S3.

Phew! The point is: on page load, our app is making one request to S3 *per* thumbnail file that the page renders. Those network requests are *super* wasteful!

The Cached Filesystem

What's the solution? Cache it! Go back to OneupFlysystemBundle and find the main page of their docs. Oh! Apparently I need Wifi for that! There we go. Go back to their docs homepage and search for "cache". You'll eventually find a link about "Caching your filesystem".

This is a *super* neat feature of Flysystem where you can say:

> Hey Flysystem! When you check some file metadata, like whether or not a file exists, cache that so that we don't need to ask S3 every time!

Actually, it's even more interesting & useful. LiipImagineBundle calls the exists() method on the Filesystem object to see if the thumbnail file already exists. If that returns *false*, the cached filesystem does *not* cache that. But if it returns true, it *does* cache it. The result is this: the first time LiipImagineBundle asks if a thumbnail image exists, Flysystem will return false, and Liip will know to generate it. The *second* time it asks, because the "false" value wasn't cached, Flysystem *will* still talk to S3, which will *now* say:

> Yea! That file *does* exist.

And because the cached adapter *does* cache this, the *third* time LiipImagineBundle calls exists, Flysystem will immediately return true without talking to S3.

To get this rocking, copy the composer require line, find your terminal and paste to download this "cached" Flysystem adapter.

```
$ composer require league/flysystem-cached-adapter
```

While we're waiting, go check out the docs. Here's the "gist" of how this works, it's 3 parts. First, you have some existing

filesystem - like my_filesystem. *Second*, via this cache key, you register a *new* "cached" adapter and tell it how you want things to be cached. And third, you tell your existing filesystem to process its logic through that cached adapter. If that doesn't totally make sense yet, no worries.

For *how* you want the cached adapter to cache things, there are a *bunch* of options. We're going to use the one called PSR6. You may or may not already know that Symfony has a *wonderful* cache system built right into it. Anytime you need to cache *anything*, you can just use *it*!

Configuring Symfony's Cache Pool

Start by going to config/packages/cache.yaml. *This* is where you can configure anything related to Symfony's cache system, and we talked a bit about it in our Symfony Fundamentals course. The app key determines how the cache.app service caches things, which is a general-purpose cache service you can use for anything, including this! *Or*, to be fancier - I like being fancy - you can create a cache "pool" *based* on this.

Check it out. Uncomment pools and create a new cache pool below this called cache.flysystem.psr6. The name can be anything. Below, set adapter to cache.app.

21 lines | config/packages/cache.yaml

```
1  framework:
2      cache:
   ... lines 3 - 17
18         pools:
19             cache.flysystem.psr6:
20                 adapter: cache.app
```

That's it! This creates a *new* cache service called cache.flysystem.psr6 that, really... just uses cache.app behind the scenes to cache everything. The *advantage* is that this new service will automatically use a cache "namespace" so that its keys won't collide with other keys from other parts of your app that *also* use cache.app.

In your terminal, run:

```
$ php bin/console debug:container psr6
```

There it is! A new fancy cache.flysystem.psr6 service.

Back in oneup_flysystem.yaml, let's use this! On top... though it doesn't matter where, add cache: and put one new cached adapter below it: psr6_app_cache. The name here *also* doesn't matter - but we'll reference it in a minute.

21 lines | config/packages/oneup_flysystem.yaml

```
   ... line 1
2  oneup_flysystem:
3      cache:
4          psr6_app_cache:
   ... lines 5 - 21
```

And below *that* add psr6:. That exact *key* is the important part: it tells the bundle that we're going to pass it a PSR6-style caching object that the adapter should use internally. Finally, set service to what we created in cache.yaml: cache.flysystem.psr6.

21 lines | config/packages/oneup_flysystem.yaml

```
   ... line 1
2  oneup_flysystem:
3      cache:
4          psr6_app_cache:
5              psr6:
6                  service: cache.flysystem.psr6
   ... lines 7 - 21
```

At this point, we have a new Flysystem *cache* adapter... but nobody is using it. To fix that, duplicate uploads_filesystem and create a second one called cached_uploads_filesystem. Make it use the same adapter as before, but with an extra key: cache: set to the adapter name we used above: psr6_app_cache.

21 lines | config/packages/oneup_flysystem.yaml

```
... line 1
2  oneup_flysystem:
... lines 3 - 13
14     filesystems:
... lines 15 - 17
18         cached_uploads_filesystem:
19             adapter: uploads_adapter
20             cache: psr6_app_cache
```

Thanks to this, all Filesystem calls will *first* go through the cached adapter. If something is cached, it will return it immediately. Everything else will get forwarded to the S3 adapter and work like normal. This is *classic* object decoration.

After all of this work, we should have one new service in the container. Run:

```
$ php bin/console debug:container cached_uploads
```

There it is: oneup_flysystem.cached_uploads_filesystem_filesystem. *Finally*, go back to liip_imagine.yaml. For the loader, we don't really need caching: this downloads the source file, which should only happen one time anyways. Let's leave it.

But for the resolver, we *do* want to cache this. Add the cached_ to the service id. The resolver is responsible for checking if the thumbnail file exists - something we *do* want to cache - *and* for *saving* the cached file. But, "save" operations are never cached - so it won't affect that.

69 lines | config/packages/liip_imagine.yaml

```
1  liip_imagine:
... lines 2 - 13
14     resolvers:
15         flysystem_resolver:
16             flysystem:
17                 # use the cached version so we're not checking to
18                 # see if the thumbnailed file lives on S3 on every request
19                 filesystem_service: oneup_flysystem.cached_uploads_filesystem_filesystem
... lines 20 - 69
```

Let's try this! Refresh the page. Ok, everything seems to work fine. Now, check your tweets, like some Instagram photos, then turn off your Wifi again. Moment of truth: do a force refresh to *fully* make sure we're reloading. Awesome! Yea, the page looks *terrible* - a bunch of things fail. But our server did *not* fail: we are *no* longer talking to S3 on every request. *Big* win.

Next, let's use a *super* cool feature of S3 - *signed* URLs - to see an alternate way of allowing users to download private files, which, for large stuff, is more performant.

Chapter 38: Private Downloads & Signed URLs

I have *one* more performance enhancement I want to do. If you click download, it works great! But if these files were bigger, you'd start to notice that the downloads would be kinda slow! Open up ArticleReferenceAdminController and search for download. Remember: we're reading a stream from S3 and sending that directly to the user. That's cool... but it also means that there's a middleman in the process: our server! That slows things down. Couldn't we somehow give the user direct access to the file on S3?

Go back to our bucket, head to its root directory, then click into article_reference. If you click any of these files, each *does* have a URL. But if you try to go to it, it's not public. That's *great* because these files are *meant* to be private... but it sorta ruins our idea of pointing users directly to this URL.

Well, good news! We *can* have our cake and eat it too... as we say... for some reason in English. Um, we *can* have the best of both worlds with... signed URLs.

Hello Signed URLs

Signed URLs are *not* something that we can create with Flysystem - it's specific to S3. So, instead of using our Filesystem object, we'll deal with S3 directly, which turns out to be pretty awesome!

Google for "S3 PHP client signed url" to find their docs about this. Signed URLs let *us* say:

> Hey S3! I want to create a public URL to download this file... but I only want the link to be valid for, like, 20 minutes.

Cool, right! Because the link is temporary, it's ok to let users use it.

We'll do this by interacting with the S3Client object directly... which is super *awesome* because, a few minutes ago, we registered an S3Client service so we could use it with Flysystem. Half our job is already done!

The other thing we'll need is the bucket name.

Creating the Signed URL

Head back to downloadArticleReference(). Remove the UploaderHelper argument - we won't need that anymore - and add S3Client $s3client. Also add string $s3BucketName.

194 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 7
8   use Aws\S3\S3Client;
... lines 9 - 21
22  class ArticleReferenceAdminController extends BaseController
23  {
... lines 24 - 127
128     public function downloadArticleReference(ArticleReference $reference, S3Client $s3Client, string $s3BucketName)
129     {
... lines 130 - 139
140     }
... lines 141 - 192
193 }
```

That won't autowire, so copy the argument name, open up services.yaml and add a bind for this $s3BucketName:. For the value, copy the environment variable bucket syntax from before and... paste.

```
61 lines | config/services.yaml
... lines 1 - 10
services:
... line 12
    _defaults:
... lines 14 - 20
        bind:
... lines 22 - 25
            $s3BucketName: '%env(AWS_S3_BUCKET_NAME)%'
... lines 27 - 61
```

Cool! Back in the controller, copy the $disposition line - we're going to put this back in a minute. Then, delete *everything* after the security check, paste the $disposition line, but comment it out for now.

Ok, let's go steal some code from the docs! We already have the S3Client object, so just grab the rest. Paste that then... let's see... replace my-bucket with the $s3BucketName variable. For Key, that's the *file* path: $reference->getFilePath(). And, for $request = $s3Client->createPresignedRequest(), you can use whatever lifetime you want. These files are pretty small, so we don't need too much time - but let's make the URLs live for 30 minutes.

```
194 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 7
use Aws\S3\S3Client;
... lines 9 - 21
class ArticleReferenceAdminController extends BaseController
{
... lines 24 - 127
    public function downloadArticleReference(ArticleReference $reference, S3Client $s3Client, string $s3BucketName)
    {
        $article = $reference->getArticle();
        $this->denyAccessUnlessGranted('MANAGE', $article);

        $command = $s3Client->getCommand('GetObject', [
            'Bucket' => $s3BucketName,
            'Key' => $reference->getFilePath()
        ]);
        $request = $s3Client->createPresignedRequest($command, '+30 minutes');
... lines 138 - 139
    }
... lines 141 - 192
}
```

Now that we have this "request" thing... how can we get the URL? Back on their docs, scroll down... here it is: $request->getUri().

When the user hits our endpoint, what *we* want to do is *redirect* them to the URL. Do that with return new RedirectResponse(), (string) - they mentioned that in the docs, it turns the URI into a string - then $request->getUri().

194 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 12
use Symfony\Component\HttpFoundation\RedirectResponse;
... lines 14 - 21
class ArticleReferenceAdminController extends BaseController
{
... lines 24 - 127
    public function downloadArticleReference(ArticleReference $reference, S3Client $s3Client, string $s3BucketName)
    {
        $article = $reference->getArticle();
        $this->denyAccessUnlessGranted('MANAGE', $article);

        $command = $s3Client->getCommand('GetObject', [
            'Bucket' => $s3BucketName,
            'Key' => $reference->getFilePath()
        ]);
        $request = $s3Client->createPresignedRequest($command, '+30 minutes');

        return new RedirectResponse((string) $request->getUri());
    }
... lines 141 - 192
}
```

Let's try it! Refresh! And... download! Ha! It works! We're loading this directly from S3. This long URL contains a signature that proves to S3 that the request was pre-authenticated and should last for 30 minutes.

Forcing S3 Response Headers

But we *did* lose one thing: our Content-Disposition header! This gave us two nice things: it forced the user to download the file instead of loading it "inline", *and* it controlled the download filename.

Hmm, this is tricky. Now that the user is no longer downloading the file directly from us, we don't really have a way to set custom *headers* on the response. Well, actually, that's a big ol' lie! There are *two* ways to do that. First, you can set custom headers on each object in S3. *Or* you can *hint* to S3 that you want *it* to set custom headers on your behalf when the user goes to the signed URL.

How? Add another option to getCommand(): ResponseContentType set to $reference->getMimeType(). That'll hint to S3 that we want it to set a Content-Type header on the download response.

201 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 21
class ArticleReferenceAdminController extends BaseController
{
... lines 24 - 127
    public function downloadArticleReference(ArticleReference $reference, S3Client $s3Client, string $s3BucketName)
    {
... lines 130 - 137
        $command = $s3Client->getCommand('GetObject', [
            'Bucket' => $s3BucketName,
            'Key' => $reference->getFilePath(),
            'ResponseContentType' => $reference->getMimeType(),
... line 142
        ]);
... lines 144 - 146
    }
... lines 148 - 201
```

And ResponseContentDisposition. Move the $disposition code up above, then use that value down here.

201 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 21
22 class ArticleReferenceAdminController extends BaseController
23 {
... lines 24 - 127
128     public function downloadArticleReference(ArticleReference $reference, S3Client $s3Client, string $s3BucketName)
129     {
... lines 130 - 132
133         $disposition = HeaderUtils::makeDisposition(
134             ResponseHeaderBag::DISPOSITION_ATTACHMENT,
135             $reference->getOriginalFilename()
136         );
137
138         $command = $s3Client->getCommand('GetObject', [
139             'Bucket' => $s3BucketName,
140             'Key' => $reference->getFilePath(),
141             'ResponseContentType' => $reference->getMimeType(),
142             'ResponseContentDisposition' => $disposition,
143         ]);
... lines 144 - 146
147     }
... lines 148 - 201
```

Cool, right? Go download the file one more time. Ha! It downloads *and* uses the original filename. This is probably the best way to allow users to download private files. Oh, and if you need even *faster* downloads... cause S3 isn't *that* fast for large files, you can do the same thing with Cloudfront. Cloudfront is another service that gives users faster access to S3 files, and has a similar process for creating signed URLs.

Ok friends, only *one* thing left, and it's a fun one! Let's talk about how our file upload endpoint *might* look different if we were building a pure API.

Chapter 39: API-Style Uploads

How does a file upload work if you're building an API? Well, you have two options. First, you can make your API endpoint look *exactly* like what we already built in uploadArticleReference().

Using our Current Endpoint with an API Client

Let me show you what I mean. I'm going to use Postman to interact with our endpoint as if it were truly meant to be an API endpoint used by API clients. For the URL, copy the URL in the browser, paste, and change /edit to /references. Yep, that'll hit our controller. Make this a POST request.

What about the *body* of the request? What should that look like? Well, because we wrote our endpoint to basically handle a traditional form-submit, the format will be form-data. For the key, remember that we're expecting the file data on a field called reference. Change the field type to "file" and select earth.jpeg.

That's it! Before trying this, our site is being served over https thanks to the Symfony local web server and some certificate magic it does behind the scenes. But Postman doesn't *know* to use that magic, so the certificate won't work. In the Postman preferences - I've already done it - turn SSL verification off. Or you can run the Symfony web server with the --allow-http flag if you want to avoid this.

Ok, send the request! Oh... what's this? Check out the preview. The login page, of course! Uploading requires a valid user. Just to play around, let's remove the @IsGranted() temporarily.

200 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 21
22 class ArticleReferenceAdminController extends BaseController
23 {
24     /**
25      * @Route("/admin/article/{id}/references", name="admin_article_add_reference", methods={"POST"})
26      */
27     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
28     {
... lines 29 - 75
76     }
... lines 77 - 198
199 }
```

Try it again. Beautiful! It works!

So, the *first* way to build an upload endpoint for an API is... like this! An endpoint that requires the multipart form data format that we checked out at the beginning of this tutorial. Any API client will be able to work with this and a lot of API's are built this way.

Pure API Endpoint with JSON: base64_decode

But, there's another way. And if you're building an API, this might *feel* a little bit more natural. To see it, change the body to "raw", or actually, to JSON so we can set the request body manually, instead of Postman building it for us from the nice form-data GUI.

When we change to use a JSON body, Postman helpfully auto-sets the Content-Type header to application/json, which depending on your API, you may or may not need. But it's always a good practice.

Ok, let's think about this from the perspective of a *user* of our API: if I want to send a file reference to a server, usually I'd expect the body to look something like this {"filename": "space.txt"} with, maybe a bunch of other fields. Because... in an API, the request usually contains *JSON*! Not the weird form-data format.

Of course, space.txt isn't the *content* of a file, but we *would* still probably want to be able to send the original filename. For the

data, hmm, I'm just making this up, what if we create a data key and put the binary data right here? That's great! Oh, except... you *can't* put binary data in JSON: it's just *not* supported.

API's work around this fact by expecting the client to base64 *encode* the data. Search for "base64 encode online" to find a site that can base64 encode some stuff for us really easily. Let's type in some text that we want to encode and... oops! We're on the *decode* side. Switch to encode and... there we go! We get this simple, encoded string. By the way, the main downside to this approach is that base64 encoded data is slightly bigger than the original data. On small or medium files, this makes very little difference. But if you're uploading *huge* files, using the base64 encoded data will slow things down, because more data needs to be transferred.

Anyways, paste *that* on the data key. We know this won't work... because our controller is *totally* not set up to receive JSON, but pff. Let's try it anyways. Hit send and... validation error!

> Please select a file to upload

Deserializer & A Model Class

Love it! Let's get to work. Back in our controller, to see what it looks like, let's make this endpoint capable of handling *both* ways of uploading files: form-data *and* JSON.

We can figure out which situation we're in by looking at the Content-Type header. So, if $request->headers->get('Content-Type') === 'application/json', we'll do our *new* thing, else, run the normal code. And... this is pretty cool... the *only* part that'll *really* be different is the $uploadedFile part. Move that into the else.

216 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 22
23 class ArticleReferenceAdminController extends BaseController
24 {
... lines 25 - 27
28     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
29     {
30         if ($request->headers->get('Content-Type') === 'application/json') {
... lines 31 - 42
43         } else {
44             /** @var UploadedFile $uploadedFile */
45             $uploadedFile = $request->files->get('reference');
46         }
... lines 47 - 214
215 }
```

In the first part of the if, just like a normal API endpoint, we need to decode the JSON request content into something useful. To do that, let's use the serializer! Search for "deser", there it is. Earlier, we used deserialize() to turn the JSON into an ArticleReference object. That worked because the keys in that JSON matched the property names in that class.

But in this case, look at the fields: filename and data. We *do* have an originalFilename field, and we *could* rename the filename key to that... but we definitely do *not* have... and do not *want* a data property on ArticleReference that's equal to a base64 encoded version of our file. That makes no sense.

This is a *classic* case where the data of an endpoint doesn't match the structure of our entity. And that's cool! Instead of using the entity, we can create a new *model* class.

Inside src/, let's create a new Api/ directory - just for organization - and inside, a new class: how about ArticleReferenceUploadApiModel. The *whole* point of this class is to help us deal with the data for this endpoint. So, its properties should match the data. Add public $filename and public $data.

19 lines | src/Api/ArticleReferenceUploadApiModel.php

```
1  <?php
2
3  namespace App\Api;
   ... lines 4 - 6
7  class ArticleReferenceUploadApiModel
8  {
   ... lines 9 - 11
12     public $filename;
   ... lines 13 - 16
17     public $data;
18 }
```

Yes! Gasp! They're public! Because this class will only be used for this *one*, *narrow*, purpose, it's ok to make life a bit easier with public properties. If this makes you want to scream and tackle me, I get it! Just make them private and add the getter & setter methods. That will work perfectly.

While we're here, don't forget about validation: add @Assert\NotBlank above both of these.

19 lines | src/Api/ArticleReferenceUploadApiModel.php

```
1  <?php
2
3  namespace App\Api;
4
5  use Symfony\Component\Validator\Constraints as Assert;
6
7  class ArticleReferenceUploadApiModel
8  {
9      /**
10      * @Assert\NotBlank()
11      */
12     public $filename;
13
14     /**
15      * @Assert\NotBlank()
16      */
17     public $data;
18 }
```

We're ready! Back in the controller add a new argument at the end: SerializerInterface $serializer. Then, it's beautiful, really $uploadApiModel = $serializer->deserialize(). This takes three arguments: the raw JSON - $request->getContent() - the *type* of object it should be turned into - ArticleReferenceUploadApiModel::class - and the input format, json.

```
216 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 4
5  use App\Api\ArticleReferenceUploadApiModel;
... lines 6 - 22
23 class ArticleReferenceAdminController extends BaseController
24 {
... lines 25 - 27
28     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
29     {
30         if ($request->headers->get('Content-Type') === 'application/json') {
31             $uploadApiModel = $serializer->deserialize(
32                 $request->getContent(),
33                 ArticleReferenceUploadApiModel::class,
34                 'json'
35             );
... lines 36 - 42
43         } else {
... lines 44 - 45
46         }
... lines 47 - 214
215 }
```

We don't need a context this time, because we're not deserializing into an existing object and we don't need to use groups.

And because this object has some constraints, we'll need to check validation up here: $violations = $validator->validate($uploadApiModel). And if $violations->count() > 0, return the normal, $this->json($violations, 400).

```
216 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 4
5  use App\Api\ArticleReferenceUploadApiModel;
... lines 6 - 22
23 class ArticleReferenceAdminController extends BaseController
24 {
... lines 25 - 27
28     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
29     {
30         if ($request->headers->get('Content-Type') === 'application/json') {
... lines 31 - 36
37             $violations = $validator->validate($uploadApiModel);
38             if ($violations->count() > 0) {
39                 return $this->json($violations, 400);
40             }
... lines 41 - 42
43         } else {
... lines 44 - 45
46         }
... lines 47 - 214
215 }
```

At the bottom, let's dd($uploadApiModel) so we can see if this crazy idea is working.

216 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 4
5  use App\Api\ArticleReferenceUploadApiModel;
... lines 6 - 22
23 class ArticleReferenceAdminController extends BaseController
24 {
... lines 25 - 27
28     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
29     {
30         if ($request->headers->get('Content-Type') === 'application/json') {
... lines 31 - 36
37             $violations = $validator->validate($uploadApiModel);
38             if ($violations->count() > 0) {
39                 return $this->json($violations, 400);
40             }
41
42             dd($uploadApiModel);
43         } else {
... lines 44 - 45
46         }
... lines 47 - 214
215 }
```

You ready to try this? Spin back over to Postman, high-five someone near you and... send! Hey! Check out that *beautiful* dump! The text is still encoded, but that's a *killer* first step. Leave the filename blank to check validation. Looks great.

Let's finish this next: we still need to base64 *decode* that data and push it into our normal file upload system. Let's do that in a clean way that we can love.

Chapter 40: Coding the API Upload Endpoint

Our controller is reading this JSON and decoding it into a nice ArticleReferenceUploadApiModel object. But the data property on that is still *base64* encoded.

base64_decode from the Model Class

Decoding is easy enough. But let's make our new model class a bit smarter to help with this. First, change the data property to be *private*. If we *only* did this, the serializer would *no* longer be able to set that onto our object.

27 lines | src/Api/ArticleReferenceUploadApiModel.php

```
... lines 1 - 6
7  class ArticleReferenceUploadApiModel
8  {
... lines 9 - 16
17     private $data;
... lines 18 - 25
26 }
```

Hit "Send" to see this. Yep! the data key is ignored: it's not a field the client can send, because there's no setter for it and it's not public. Then, validation fails because that field is still empty.

So, because I've mysteriously said that we should set the property to private, add a public function setData() with a nullable string argument... because the user could forget to send that field. Inside, $this->data = $data.

27 lines | src/Api/ArticleReferenceUploadApiModel.php

```
... lines 1 - 6
7  class ArticleReferenceUploadApiModel
8  {
... lines 9 - 16
17     private $data;
... lines 18 - 20
21     public function setData(?string $data)
22     {
23         $this->data = $data;
... line 24
25     }
26 }
```

Now, create another property: private $decodedData. And inside the setter, $this->decodedData = base64_decode($data). And because this is private and does *not* have a setter method, if a smart user tried to send a decodedData key on the JSON, it would be ignored. The only valid fields are filename - because it's public - and data - because it has a setter.

```
27 lines | src/Api/ArticleReferenceUploadApiModel.php
... lines 1 - 6
7  class ArticleReferenceUploadApiModel
8  {
... lines 9 - 16
17     private $data;
18
19     private $decodedData;
20
21     public function setData(?string $data)
22     {
23         $this->data = $data;
24         $this->decodedData = base64_decode($data);
25     }
26 }
```

Try it again. It's working *and* the decoded data is ready! It's a simple string in our case, but this would work equally well if you base64 encoded a PDF, for example.

Saving a Temporary File

Let's look at the controller. We know the "else" part, that's the "traditional" upload part, is working by simply setting an $uploadedFile object and letting the rest of the controller do its magic. So, if we can create an UploadedFile object up here, we're in business! It should go through validation... and process.

If you remember from our fixtures, we can't *actually* create UploadedFile objects - it's tied to the PHP upload process. But we *can* create File objects. Open up ArticleFixtures. At the bottom, yep! We create a new File() - that's the *parent* class of UploadedFile and pass it $targetPath, which is the path to a file on the filesystem. UploaderHelper can already handle this.

In the controller, we can do the same thing. Start by setting $tmpPath to sys_get_temp_dir() plus '/sf_upload'.uniqueid() to guarantee a unique, temporary file path. Yep, we're literally going to *save* the file to disk so our upload system can process it. We *could* also enhance UploaderHelper to be able to handle the content as a *string*, but this way will re-use more logic.

```
221 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 23
24  class ArticleReferenceAdminController extends BaseController
25  {
... lines 26 - 28
29      public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
30      {
31          if ($request->headers->get('Content-Type') === 'application/json') {
... lines 32 - 43
44              $tmpPath = sys_get_temp_dir().'/sf_upload'.uniqid();
... lines 45 - 47
48          } else {
... lines 49 - 50
51          }
... lines 52 - 96
97      }
... lines 98 - 219
220 }
```

To get the raw content, go back to the model class. We need a getter. Add public function getDecodedData() with a nullable string return type. Then, return $this->decodedData.

```
32 lines | src/Api/ArticleReferenceUploadApiModel.php
... lines 1 - 6
7  class ArticleReferenceUploadApiModel
8  {
... lines 9 - 26
27     public function getDecodedData(): ?string
28     {
29         return $this->decodedData;
30     }
31 }
```

Now we can say: file_put_contents($tmpPath, $uploadedApiModel->getDecodedData()). Oh, I'm not getting any auto-completion on that because PhpStorm doesn't know what the $uploadedApiModel object is. Add some inline doc to help it. Now, $this->, got it - getDecodedData().

```
221 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 23
24  class ArticleReferenceAdminController extends BaseController
25  {
... lines 26 - 28
29     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
30     {
31         if ($request->headers->get('Content-Type') === 'application/json') {
32             /** @var ArticleReferenceUploadApiModel $uploadApiModel */
33             $uploadApiModel = $serializer->deserialize(
... lines 34 - 36
37             );
... lines 38 - 43
44             $tmpPath = sys_get_temp_dir().'/sf_upload'.uniqid();
45             file_put_contents($tmpPath, $uploadApiModel->getDecodedData());
... lines 46 - 47
48         } else {
... lines 49 - 50
51         }
... lines 52 - 96
97     }
... lines 98 - 219
220 }
```

Finally, set $uploadedFile to a new File() - the one from HttpFoundation. Woh! That was weird - it put the full, long class name here. Technically, that's fine... but why? Undo that, then go check out the use statements. Ah: this is one of those rare cases where we already have *another* class imported with the same name: File. Let's add our use statement manually, then alias is to, how about, FileObject. I know, a bit ugly, but necessary.

Below, new FileObject() and pass it the temporary path. Let's dd() that.

```
221 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 11
12 use Symfony\Component\HttpFoundation\File\File as FileObject;
... lines 13 - 23
24 class ArticleReferenceAdminController extends BaseController
25 {
... lines 26 - 28
29     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
30     {
31         if ($request->headers->get('Content-Type') === 'application/json') {
... lines 32 - 43
44             $tmpPath = sys_get_temp_dir().'/sf_upload'.uniqid();
45             file_put_contents($tmpPath, $uploadApiModel->getDecodedData());
46             $uploadedFile = new FileObject($tmpPath);
47             dd($uploadedFile);
48         } else {
... lines 49 - 50
51         }
... lines 52 - 96
97     }
... lines 98 - 219
220 }
```

Phew! Back on Postman, hit send. Hey! That looks great! Copy that filename, then, wait! That was just the directory - copy the *actual* filename - called pathname, find your terminal and I'll open that in vim.

Getting the "Client Original Name"

Yes! The contents are *perfect*! So... are we done? Let's find out! Take off the dd(), move over and... this is our moment of glory... send! Oh, boo! No glory, just errors. Life of a programmer.

> Undefined method getClientOriginalName() on File.

This comes from down here on line 84. Ah yes, the UploadedFile object has a few methods that its parent File does not. Notably getClientOriginalName().

No problem, back up, create an $originalName variable on both sides of the if. For the API style, set it to $uploadApiModel->filename: the API client will send this manually. For the else, set $originalName to $uploadedFile->getClientOriginalName(). Now, copy $originalName, head back down to setOriginalFilename() and paste! And if for some reason it's not set, we can still use $filename as a backup. But that's definitely impossible for our API-style thanks to the validation rules.

```
222 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 23
24 class ArticleReferenceAdminController extends BaseController
25 {
... lines 26 - 28
29     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
30     {
31         if ($request->headers->get('Content-Type') === 'application/json') {
... lines 32 - 46
47             $originalFilename = $uploadApiModel->filename;
48         } else {
... lines 49 - 50
51             $originalFilename = $uploadedFile->getClientOriginalName();
52         }
... lines 53 - 83
84         $articleReference->setOriginalFilename($originalFilename ?? $filename);
... lines 85 - 97
98     }
... lines 99 - 220
221 }
```

Deep breath. Let's try it again. Woh! Did that just work? It looks right. Go refresh the browser. Ha! We have a space.txt file! And we can even download it! Go check out S3 - the article_reference directory.

Oh, interesting! The files are prefixed with sf-uploads - that's the temporary filename we created on the server. That's because UploaderHelper uses that to create the unique filename. And really, that's fine! These filenames are 100% internal. But if it bothers you, you could use the original filename to help make the temporary file.

Anyways... we did it! A fully JSON-driven API upload endpoint. Fun, right?

Removing the Temporary File

Before we finish... and ride off into the sunset, as champions of uploading in Symfony, let's make sure we delete that temporary file after we finish.

All the way down here, before persist, but *after* we've tried to read the mime type from the file, add, if is_file($uploadedFile->getPathname()), then delete it: unlink($uploadedFile->getPathname()).

227 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 23
24 class ArticleReferenceAdminController extends BaseController
25 {
... lines 26 - 28
29     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
30     {
... lines 31 - 84
85         $articleReference->setMimeType($uploadedFile->getMimeType() ?? 'application/octet-stream');
86
87         if (is_file($uploadedFile->getPathname())) {
88             unlink($uploadedFile->getPathname());
... line 89
90         }
91
92         $entityManager->persist($articleReference);
... lines 93 - 102
103     }
... lines 104 - 225
226 }
```

The if is sorta unnecessary, but I like it. To double-check that this works, let's dd($uploadedFile->getPathname()), go find Postman and send. Copy the path, find your terminal, and try to open that file. It's gone!

227 lines | src/Controller/ArticleReferenceAdminController.php

```
... lines 1 - 23
24 class ArticleReferenceAdminController extends BaseController
25 {
... lines 26 - 28
29     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
30     {
... lines 31 - 84
85         $articleReference->setMimeType($uploadedFile->getMimeType() ?? 'application/octet-stream');
86
87         if (is_file($uploadedFile->getPathname())) {
88             unlink($uploadedFile->getPathname());
89             dd($uploadedFile->getPathname());
90         }
91
92         $entityManager->persist($articleReference);
... lines 93 - 102
103     }
... lines 104 - 225
226 }
```

Celebrate by removing that dd() and sending one last time. I'm *so* happy.

```
226 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 23
24 class ArticleReferenceAdminController extends BaseController
25 {
... lines 26 - 28
29     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
30     {
... lines 31 - 86
87         if (is_file($uploadedFile->getPathname())) {
88             unlink($uploadedFile->getPathname());
89         }
... lines 90 - 101
102     }
... lines 103 - 224
225 }
```

Oh, and don't forget to put security back: @IsGranted("MANAGE", subject="article"). In a real project, wherever I test my API endpoints - like Postman or via functional tests, I would actually *authenticate* myself properly so they worked, instead of temporarily hacking out security. Generally speaking, removing security is, uh, not a *great* idea.

```
227 lines | src/Controller/ArticleReferenceAdminController.php
... lines 1 - 23
24 class ArticleReferenceAdminController extends BaseController
25 {
26     /**
27      * @Route("/admin/article/{id}/references", name="admin_article_add_reference", methods={"POST"})
28      * @IsGranted("MANAGE", subject="article")
29      */
30     public function uploadArticleReference(Article $article, Request $request, UploaderHelper $uploaderHelper, EntityManagerInterfac
31     {
... lines 32 - 102
103     }
... lines 104 - 225
226 }
```

Hey! That's it! We did it! Woh! I had a *ton* of a fun making this tutorial - we got to play with uploads, a bunch of cool libraries and... the *cloud*. Uploading is *fairly* simple, but there *can* be a lot of layers to keep track of, like Flysystem and LiipImagineBundle.

As always, let us know what you're building and if you have questions, ask them in the comments. Alright friends, seeya next time!