API Platform Part 2: Security

With <3 from SymfonyCasts

Chapter 1: Hello API Security + API Docs on Production?

Friends! Welcome to part two of our API Platform series - the one where we're talking all about ice cream, um, security. We're talking about security, not uh, ice cream. Hmm. Um, it's going to be *almost* as awesome as ice cream though, because the topic of security - especially API security - is fascinating! We've got API tokens, session-based authentication, CSRF attacks, dragon attacks and the challenge of securing our API Platform application down to the smallest details, like letting different users see different records *or* even returning or accepting different fields based on the user. Yep, we're going to take this *wonderful* base that API platform has given us and shape it to act *exactly* like we need from a security perspective.

The great thing about API platform is that it's basically just... Symfony security. They didn't reinvent the wheel at all. If you haven't gone through our Symfony Security tutorial yet, *now* would be a good time: you'll feel *much* more dangerous after.

Anyways, let's dive into this! To put the lock down on your API Platform security skills, download the course code from this page and code along with me. After you unzip the file, you'll find a start/ directory that has the same code you see here. Open up the README.md file for all the details on getting the app set up. If you coded through part one of this tutorial, um, you *rock*, but also, I recommend downloading the new course code because I upgraded a few dependencies and added a frontend to the site.

Anyways, one of the last steps in the README will be to open a terminal, move into the project and run:

```
$ yarn encore dev --watch
```

to run Encore and build some JavaScript that'll power a small frontend. To make things more realistic, we'll do a *little* bit of work in JavaScript to see how it interacts with our API from an authentication standpoint. Next, open *another* terminal and start the Symfony local web server:

```
$ symfony serve
```

Once that starts, you should be able to fly back over to your browser and open up https://localhost:8000 to see... Cheese Whiz! The peer-to-peer cheese-selling app we build in part 1. Actually, this is our brand new Vue.js-powered frontend. OOOOooOOO. Go to /api. Ah yes, *this* is the API we built: we have a CheeseListing resource, a User resource, they're related to each other and we've customized a ton of stuff. In a bit, we'll start making this frontend talk to the API.

Hiding the Docs on Production?

But before we get there, head back to /api to see our beautiful, interactive, Swagger documentation. I know some of you are probably thinking: I know, this is great for development, but how can I disable this for production?

The answer is... you shouldn't!

Well first, let me show you how you *can* disable this on production and *then* I'll try to convince you to keep it.

Open up config/packages/api_platform.yaml. API Platform is *highly* configurable. So, reminder time: if you go to an open terminal, you can see all your current configuration - including any default values - by running:

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

You can *also* run:

```
$ php bin/console config:dump api_platform
```

to see an example tree of *all* the possible keys with some explanations. One of the options is called enable_docs. Copy this.

The goal is to disable the docs *only* in production, but let's start by disabling them everywhere to see how it works.

Set enable_docs to false.

19 lines | config/packages/api_platform.yaml

```yaml
api_platform:
    ... lines 2 - 17
    enable_docs: false
```

Ok, go back and refresh /api. Um... there is absolutely *no* change! That's due to a small dev-only bug with a few of these options: when we change those options, they're not causing the routing cache to rebuild automatically. No big problem, clear that cache manually with:

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

Fly back over, refresh and... yep! The documentation is gone. Oh... but instead of a 404, it's a 500 error!

It turns out that going to /api to see the docs was just a convenience. The documentation was *really* stored at /api/docs and this *is* now a 404. You *could* also go to /api/docs.json or /api/docs.jsonld before, but now it's all gone.

One of the unfortunate things about removing the documentation is that it wasn't *just* rendered as HTML. In part 1 of this series, we talked about JSON-LD and Hydra, and how API Platform generates machine-readable documentation to explain your API. If I go to /api/cheeses.jsonld, this advertises that we can go to /api/contexts/cheeses to get more "context", more machine-readable "meaning". Whelp, that doesn't work anymore. The point is: if you disable documentation, realize that you're *also* disabling the machine-friendly documentation.

And if you want to fix the 500 on /api, you also need to disable the "entrypoint". Right now, if we go to /api/index.jsonld, we get a, sort of, "homepage" for our API, which tells us what URLs we could go to next to discover more. When we go to /api, that's the HTML entrypoint and that's... totally broken. To disable that page set enable_entrypoint to false. Rebuild the cache:

20 lines | config/packages/api_platform.yaml

```yaml
api_platform:
    ... lines 2 - 18
    enable_entrypoint: false
```

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

then go refresh. *Now* we get a 404 on this and /api.

So to fully disable your docs without a 500 error, you need *both* of these keys. To make this *only* happen in production, copy them, delete them, and, in the config/packages/prod directory, create a new api_platform.yaml file. Inside, start with api_platform: then paste.

4 lines | config/packages/prod/api_platform.yaml

```yaml
api_platform:
    enable_docs: false
    enable_entrypoint: false
```

If we changed to the prod environment and rebuilt the cache, we'd be done!

But instead... I'm going to comment this out... for two reasons. First, I like the documentation, I like the machine-readable documentation and I like the "entrypoint": the documentation homepage. And second, more importantly, if you want to hide your documentation so that nobody will use your API, that's a bad plan. That's security through obscurity. If your API lives out on the web, you need to assume people will find it and you need to *properly* secure it. Hey! That's the topic of this tutorial!

Rebuild the cache one more time.

At the end of the day, if you're ok keeping the machine-readable docs and the entrypoint stuff, but you *really* don't want to expose the pretty HTML docs, you *could* always create an event subscriber on the kernel.request event - now called the RequestEvent in Symfony 4.3 - and, if the URL is /api *and* the request format is HTML, return a 404.

Tip

You can see an example here: https://bit.ly/2YmR6Uh

Ok, let's get into the *real* action: let's start figuring out how we're going to let users of our API log in.

Chapter 2: API Auth 101: Session? Cookies? Tokens?

How *do* we want our users to log into our API? There are about a million possible answers for this. To figure out *your* answer, don't think about your API, just ask:

> What action will a user take to log into my app?

Most likely, your users will do something super traditional: they'll type an email and password into a form and submit. It doesn't matter if they're typing that into a boring HTML form on your site, a single page application built in Hipster.js or even in a mobile app. In *all* those situations, the user of your API will be *sending* an email & password. By the way, if you do *not* have this situation, that doesn't change much! I'll talk more about why later, but most of what we'll do will transfer to other authentication schemes.

Sessions or Tokens?

It turns out that the *more* important question - more important than what pieces of data your users will send to authenticate - is what *happens* when your API receives that data. How does it respond when you successfully authenticate?

And you might be thinking:

> Hey! We're building a RESTful API... and APIs are supposed to be "stateless"... so that means don't use sessions... and so that means our API will return some sort of an API token.

Yep! That's not... super true. Session-based authentication - the type of login system you've known and loved for *years* - is just a token-based system in disguise! When you perform a traditional login, the server sends back a cookie. This is your "token". Then, every future request sends that token and becomes authenticated.

Here's what's *really* important. If the "user" of your API is you or your company - whether that be *your* JavaScript or a mobile app owned by you, then, on authentication, your API should return an HttpOnly cookie. This type of cookie is automatically sent with each request but is *not* readable in JavaScript, which makes it safe from being stolen by other JavaScript. The *contents* of that cookie, it turns out, are much less important. It could be a session string - like we're used to in PHP, or it could be some encrypted package of information that contains authentication details. If you've heard about JSON web tokens - JWT - that's what those are: strings that actually contain information. In *all* cases, your API will set an HttpOnly cookie, each future request will naturally send that back, and your API will *use* that to authenticate the user. Exactly what that cookie looks like is not really that important.

The great thing about *session-based* authentication with cookies - versus generating a JWT and storing it in a cookie - is that it's super easy to set up. And all HTTP clients - even mobile apps - support cookies.

Listen: later on, we *are* going to dive into some details about token-based authentication systems - the ones where you attach some token string to an Authorization header when you make the request. And we'll talk about when you need this. For example, if *third* parties - like someone *else's* mobile app - need to make requests to your API and be authenticated as users in your system, you would need OAuth.

So here's our first goal: build a super-nice, API-friendly, session-based authentication system where we POST the email and password as a JSON string to an endpoint. Then, instead of returning an API token, that endpoint will start the session and send back the session cookie.

Chapter 3: Login with json_login

If your login system looks similar to the traditional email & password or username & password setup, Symfony has a nice, built-in authentication mechanism to help. In config/packages/security.yaml, under the main firewall, add a new key: json_login. Below that, set check_path to app_login.

```
36 lines | config/packages/security.yaml
security:
... lines 2 - 12
    firewalls:
... lines 14 - 16
        main:
... lines 18 - 19
            json_login:
                check_path: app_login
... lines 22 - 36
```

This is the name of a route that we're going to create in a second - and we'll set its URL to /login. Below this, set username_path to email - because that's what we'll use to log in, and password_path set to password.

```
36 lines | config/packages/security.yaml
security:
... lines 2 - 12
    firewalls:
... lines 14 - 16
        main:
... lines 18 - 19
            json_login:
... line 21
                username_path: email
                password_path: password
... lines 24 - 36
```

With this setup, when we send a POST request to /login, the json_login authenticator will automatically start running, look for JSON in the request, decode it, and use the email and password keys inside to log us in.

How does it know to load the user from the database... and which field to use for that query? The answer is: the providers section. This was added in the last tutorial *for* us by the make:user command. It tells the security system that our User lives in Doctrine and it should query for the user via the email property. If you have a more complex query... or you need to load users from somewhere totally different, you'll need to create a custom user provider or an entirely custom Guard authenticator, instead of using json_login. Basically, json_login works great if you fit into this system. If not, you can throw it in the trash and create your own authenticator.

So, there *may* be some differences between your setup and what we have here. But the *really* important part - what we're going to do on authentication success and failure - will probably be the same.

The SecurityController

To get the json_login system fully working, we need to create that app_login route. In src/Controller create a new PHP class called, how about, SecurityController. Make it extend the normal AbstractController and then create public function login(). Above that, I'll put the @Route annotation and hit tab to auto-complete that and add the use statement. Set the URL to /login, then name="app_login" and also methods={"POST"}: nobody needs to make a GET request to this.

```
21 lines | src/Controller/SecurityController.php
... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
... line 6
use Symfony\Component\Routing\Annotation\Route;
... line 8
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login", methods={"POST"})
     */
    public function login()
    {
... lines 16 - 18
    }
}
```

Initially, you need to have this route here *just* because that's the way Symfony works: you can't POST to /login and have the json_login authenticator do its magic unless you *at least* have a route. If you don't have a route, the request will 404 before json_login can get started.

But also, by default, after we log in successfully, json_login does... nothing! I mean, it will authenticate us, but then it will allow the request to continue and hit our controller. So the easiest way to control what data we return after a successful authentication is to return something from this controller!

But... hmm... I don't really know what we should return yet - I haven't thought about what might be useful. For now, let's return $this->json() with an array, and a user key set to either the authenticated user's id or null.

```
21 lines | src/Controller/SecurityController.php
... lines 1 - 13
    public function login()
    {
        return $this->json([
            'user' => $this->getUser() ? $this->getUser()->getId() : null]
        );
    }
... lines 20 - 21
```

AJAX Login in Vue.js

Let's try this! When we go to https://localhost:8000, we see a small frontend built with Vue.js. Don't worry, you don't need to know Vue.js - I just wanted to use something a bit more realistic. This login form comes from assets/js/components/LoginForm.vue.

It's mostly HTML: the only real functionality is that, when we submit the form, it won't *actually* submit. Instead, Vue will call the handleSubmit() function. Inside, uncomment that big axios block. Axios is a really nice utility for making AJAX requests. This will make a POST request to /login and send up two fields of data email and password. this.email and this.password will be whatever the user entered into those boxes.

```
65 lines | assets/js/components/LoginForm.vue
... lines 1 - 23
<script>
... lines 25 - 41
            axios
                .post('/login', {
                    email: this.email,
                    password: this.password
                })
                .then(response => {
                    console.log(response.data);

                    //this.$emit('user-authenticated', userUri);
                    //this.email = '';
                    //this.password = '';
                }).catch(error => {
                    console.log(error.response.data);
                }).finally(() => {
                    this.isLoading = false;
                })
... lines 58 - 60
</script>
... lines 62 - 65
```

One important detail about axios is that it will automatically encode these two fields as JSON. A lot of AJAX libraries do *not* do this... and it'll make a *big* difference. More on that later.

Anyways, on success, I'm logging the data from the response and on error - that's .catch() - I'm doing the same thing.

Since we haven't even *tried* to add real users to the database yet... let's see what failure feels like! Log in as quesolover@example.com, any password and... huh... nothing happens?

Hmm, if you get this, first check that Webpack Encore is running in the background: otherwise you might still be executing the old, commented-out JavaScript. Mine is running. I'll do a force refresh - I think my browser is messing with me! Let's try that again: queso_lover@example.com, password foo and... yes! We get a 401 status code and it logged error Invalid credentials.. If you look at the response itself... on failure, the json_login system gives us this simple, but perfectly useful API response.

Next, let's hook up our frontend to use this and learn how json_login behaves when we accidentally send it a, let's say, less-well-formed login request.

Chapter 4: Authentication Errors

We just found out that, if we send a bad email & password to the built-in json_login authenticator, it sends back a nicely-formed JSON response: an error key set to what went wrong.

Great! We can totally work with that! But, if you *do* need more control, you can put a key under json_login called failure_handler. Create a class, make it implement AuthenticationFailureHandlerInterface, then use that class name here. With that, you'll have *full* control to return *whatever* response you want on authentication failure.

But this is good! Let's use this to show the error on the frontend. If you're familiar with Vue.js, I have a data key called error which, up here on the login form, I use to display an error message. In other words, all *we* need to do is set this.error to a message and we're in business!

Let's do that! First, *if* error.response.data, then this.error = error.response.data.error. Ah, actually, I messed up here: I *should* be checking for error.response.data.error - I should be checking to make sure the response data has that key. And, I should be printing *just* that key. I'll catch half of my mistake in a minute.

Anyways, if we *don't* see an error key, something weird happened: set the error to Unknown error.

69 lines | assets/js/components/LoginForm.vue

```
... lines 1 - 41
            axios
... lines 43 - 52
              }).catch(error => {
                  if (error.response.data.error) {
                      this.error = error.response.data.error;
                  } else {
                      this.error = 'Unknown error';
                  }
              }).finally(() => {
... lines 60 - 69
```

Move over, refresh... and let's fail login again. Doh! It's printing the *entire* JSON message. *Now* I'll add the missing .error key. But I *should* also include it on the if statement above.

Try it again... when we fail login... that's perfect!

json_login Require a JSON Content-Type

But there's one *other* way we could fail login that we're *not* handling. Axios is smart... or at least, it's *modern*. We pass it these two fields - email and password - and *it* turned that into a JSON string. You can see this in our network tab... down here... Axios set the Content-Type header to application/json and turned the body into JSON.

Most AJAX clients don't do this. Instead, they send the data in a different format that matches what happens when you submit a traditional HTML form. If our AJAX client had done that, what do you think the json_login authenticator would have done? An error?

Let's find out! Temporarily, I'm going to add a *third* argument to .post(). This is an options array and we can use a headers key to set the Content-Type header to application/x-www-form-urlencoded. That's the Content-Type header your browser sends when you submit a form. This will tell Axios *not* to send JSON: it will send the data in a format that's invalid for the json_login authenticator.

73 lines | assets/js/components/LoginForm.vue

```
... lines 1 - 41
            axios
... lines 43 - 45
                }, {
                    headers: {
                        'content-type': 'application/x-www-form-urlencoded'
                    }
                })
... lines 51 - 73
```

Go refresh the Javascript... and fill out the form again. I'm expecting that we'll get *some* sort of error. Submit and... huh. A 200 status code? And the response says user: null.

This is coming from our SecurityController! Instead of intercepting the request and then throwing an error when it saw the malformed data... json_login did nothing! It turns out, the json_login authenticator *only* does its work if the Content-Type header contains the word json. If you make a request without that, json_login does nothing and we end up here in SecurityController.... which is probably not what we want. We probably want to return a response that tells the user what they messed up.

Returning an Error on an Invalid Authentication Request

Simple enough! Inside of the login() controller, we now know that there are two situations when we'll get here: either we hit json_login and were successfully authenticated - we'll see that soon - *or* we sent an invalid request.

Cool: if !$this->isGranted('IS_AUTHENTICATED_FULLY') - so if they're not logged in - return $this->json() and follow the same error format that json_login uses: an error key set to:

> Invalid login request: check that the Content-Type header is "application/json".

And set this to a 400 status code.

27 lines | src/Controller/SecurityController.php

```
... lines 1 - 8
class SecurityController extends AbstractController
{
... lines 11 - 13
    public function login()
    {
        if (!$this->isGranted('IS_AUTHENTICATED_FULLY')) {
            return $this->json([
                'error' => 'Invalid login request: check that the Content-Type header is "application/json".'
            ], 400);
        }
... lines 21 - 24
    }
}
```

I love it! Let's make sure it works. We didn't change any JavaScript, so no refresh needed. Submit and... we got it! The "error" side of our login is bulletproof.

Head back to our JavaScript and... I guess we should remove that extra header so things work again. Now... we're back to "Invalid credentials".

Next... I think we should try putting in some *valid* credentials! We'll hack a user into our database to do this and talk about our session-based authentication.

Chapter 5: Login Success & the Session

Let's see if we can log in for real. But first... we, uh, need to put some users in our database. Head to /api - we'll use our API to do that! Eating our own dog food.

I *do* have a few users in my database already... but you probably don't... and I don't think I set any of these with real passwords anyways. So let's create a brand new shiny user.

But... our API currently has a big shortcoming. I'll close this and open up the POST endpoint. When someone uses our API to create a user, they will eventually send the plain-text password on the password field. But... in the database, this field needs to be set to an *encoded* version of that password. So far, we don't have any mechanism to intercept the plain text password and encode it before it gets to the database.

We'll fix this soon, but we're going to cheat for now. Find your terminal and run:

```
$ php bin/console security:encode
```

This is a fun utility where you can give it a plain-text password - I'll use foo - and it will give us back an *encoded* version of that password. Copy that.

Now we can use our endpoint: I'll use the POST endpoint with email set to quesolover@example.com, password set to the long, encoded password string, username set to quesolover and I'll remove the cheeseListings field: we don't need to create any cheese listings right now. Hit "Execute" and... perfect! A 201 status code. Say hello to quesolover@example.com!

Copy that email address, then go back to our homepage. On the web debug toolbar, you can see that we are *not* logged in: we are anonymous.

Ok, let me open my browser's debugger again... then try to log in: quesolover@example.com, password foo and... nothing updates on our Vue.js app yet... but let's see what happened with that AJAX request.

Yea! It returned a 200 status code with a user key set to 6! It worked! And that response is coming from our SecurityController: we're returning that data.

Wait, Where's my API Token?

But wait, it gets better! If we refresh the homepage, we *are* now logged in as quesolover. And our important job number one is done! Just because we're creating an API doesn't mean that we now need to start thinking about some crazy API token system where the authentication endpoint returns a token string, we store that in JavaScript and then we send that as an Authorization header on all future requests. No, forget that! We're done! Starting now, all future AJAX requests will automatically send the session cookie and we'll be authenticated like normal. It's just that simple.

And yes, we *are* going to talk a bit about API token authentication later. But... there's a good chance you don't need it. And if you don't need it... but try to use it anyways, you'll complicate your app & *may* make it less secure. As a general rule, while you can *use* API tokens in your JavaScript, you should never *store* them anywhere - like local storage of cookies due to security. That makes using API tokens in JavaScript... tricky.

So... if we're not going to return an API token from the authentication endpoint... what *should* we return? Just returning the number 6... probably isn't very useful: our JavaScript won't know the email, username or *any* other information about *who* just logged in. So... what *should* we return? There's not a perfect answer to that question, but I'll show you what I recommend next.

Chapter 6: On Authentication Success

When our AJAX call to authenticate is successful, our app naturally sends back a session cookie, which all future AJAX calls will automatically use to become authenticated. So then... if our API response data doesn't need to contain a token... what *should* it contain?

Return the User as JSON on Success?

One option is to return the authenticated User object as JSON. For example... I think one of my users in the database is id 5 - so if you go to /api/users/5.json, we could return *this* JSON. Or even better we could return the JSON-LD representation of a User.

This has the benefit of being useful: our JavaScript will *then* know some info about who just logged in. But... if you want to get technical about things... this solution isn't RESTful: it sort of turns our authentication endpoint into what looks like a "user" resource. But, don't let that get in your way: if you *do* want to return the User object as JSON, you can serialize it manually and return it. I'll show you how to use the serializer to do this in the next chapter.

But... I have another suggestion.

Returning the User IRI

What if we returned the IRI - /api/users/5 - which is *also* the URL that a client can use to get more info about that user? Let's try that!

At the bottom of the controller, return a new Response() - the one from HttpFoundation - with *no* content: literally pass this null. Returning an empty response is *totally* valid, as long as you use a 204 status code, which means:

> The request was successful... but I have nothing to say to you!

So... where are we putting the IRI? On the Location header! That's a semi-standard way for an API to point to a resource. For the IRI string... hmm... how *can* we generate the URL to /api/users/5? Typically in Symfony, *we* create the route and then *we* can generate a URL to that route by using its internal name. But... now, API Platform is creating the routes for us. Is there a way with API Platform to say:

> Hey! Can you generate the IRI to this exact object?

Yep! And it's a useful trick to know. Add an argument to your controller with the IriConverterInterface type-hint. Now, set the Location header to $iriConverter->getIriFromItem() - which is one of a few useful methods on this class - and pass $this->getUser().

30 lines | src/Controller/SecurityController.php

```
... lines 1 - 4
use ApiPlatform\Core\Api\IriConverterInterface;
... lines 6 - 8
use Symfony\Component\HttpFoundation\Response;
... lines 10 - 11
class SecurityController extends AbstractController
{
... lines 14 - 16
    public function login(IriConverterInterface $iriConverter)
    {
... lines 19 - 24
        return new Response(null, 204, [
            'Location' => $iriConverter->getIriFromItem($this->getUser())
        ]);
    }
}
```

Cool! Let's see what this look like! Go back to LoginForm.vue. Right now, on success, we're logging response.data. Change that to response.headers so we can see what the headers look like.

```
69 lines | assets/js/components/LoginForm.vue
... lines 1 - 41
            axios
... lines 43 - 46
                .then(response => {
                    console.log(response.headers);
... lines 49 - 52
                }).catch(error => {
... lines 54 - 69
```

Back on our browser, refresh the homepage. By the way, you can see that the Vue.js app is reporting that we are *not* currently authenticated... even though the web debug toolbar says that we *are*. That's because our backend app & JavaScript aren't working together on page load to share this information. We'll fix that really soon.

When we log in this time... we get a 204 status code! Yes! And the logs contain a *big* array of headers with... location: "/api/users/6".

Using the IRI in JavaScript

This gives our JavaScript *everything* it needs: we can make a second request to this URL if we want to know info about the user. We're going to do *exactly* that.

Back in PhpStorm, open up CheeseWhizApp.vue. This is the main Vue file that's responsible for rendering the *entire* page - you can see the CheeseWhiz header stuff on top. And... further below, it embeds the LoginForm.vue component.

This *also* holds the logic that prints whether or not we're authenticated... via a user variable. We're not going to get too much into the details of Vue.js, but when we render the LoginForm component, we pass it a callback via the v-on attribute.

```
74 lines | assets/js/components/CheeseWhizApp.vue
... lines 1 - 25
            <div class="col-xs-12 col-md-6 px-5" style="background-color: #7FB7D7; padding-bottom: 150px;">
... line 27
                <loginForm
                    v-on:user-authenticated="onUserAuthenticated"
                ></loginForm>
            </div>
... lines 32 - 74
```

This basically means that, inside of LoginForm.vue, once the user is authenticated, we should dispatch an event called user-authenticated. When we do that, Vue will execute this onUserAuthenticated method. *That* accepts a userUri argument, which we then use to make an AJAX request for that user's data. On success, it updates the user property, which *should* cause the message on the page to change and say that we're logged in.

Phew! Let me show you what this looks inside LoginForm.vue. Uncomment the last three lines in the callback. This dispatches the user-authenticated event and passes it the user IRI that it needs. The userUri variable doesn't exist, but we know how to get that: response.headers.location. I'll take out my console.log().

```
67 lines | assets/js/components/LoginForm.vue
... lines 1 - 41
            axios
... lines 43 - 46
              .then(response => {
                this.$emit('user-authenticated', response.headers.location);
                this.email = '';
                this.password = '';
              }).catch(error => {
... lines 52 - 67
```

Let's do this! Move over, refresh, then login as quesolover@example.com, password foo. And... oh boo:

> TypeError: Cannot read property 'substr' of undefined.

My bad! I forgot that all headers are normalized and lowercased. Make sure the location header has a *lowercase* "L". Refresh the whole page one more time, put in the email, password and... watch the left side. Boom! It says:

> You are currently authenticated as quesolover Log out.

At this point we're using session-based authentication, which is the *best* solution in *many* cases. And because we're relying on cookies for authentication, our authentication endpoint can really return... whatever is useful! Note that this *also* avoids the need for the very un-RESTful /me endpoint that some API's like Facebook expose as a, sort of "cheating" way for a client to get information about who you are currently logged in as.

Next - if we refreshed right now, our JavaScript would forget that we're logged in. Silly JavaScript! Let's leverage the serializer to communicate who is logged in from the server to JavaScript on page load.

Chapter 7: Logout & Passing API Data to JS on Page Load

Hey! We can log in and our JavaScript even *knows* when we log in, and who we are: it prints our username & a log out link that goes to /logout... which doesn't actually work yet... cause we haven't enabled that in Symfony.

Adding Logout

Wait... but what does "logging out" even *mean* in an API context? Whelp, like everything, it depends on *how* you authenticate. Because we're using a session cookie, logging out basically means removing the user information from the session. If you were using some sort of API token, it would mean *invalidating* that token on your authentication server - like, removing it from some database table for tokens... again, it depends on your setup. We'll talk more about that type of authentication a bit later - on a special security part 2 of this tutorial.

Anyways, no surprise that Symfony has built-in support for logging the user out of the session. In config/packages/security.yaml, under our firewall add logout: then, below, say path: app_logout. Just like app_login, this is the name of a route that we're going to create next. When a user accesses this route, they'll be logged out.

39 lines | config/packages/security.yaml

```
security:
... lines 2 - 12
    firewalls:
... lines 14 - 16
        main:
... lines 18 - 24
            logout:
                path: app_logout
... lines 27 - 39
```

To create that, open src/Controller/SecurityController.php and add public function logout() with @Route() above. Set the URL to /logout and name it app_logout.

Just like with the app_login route, the route just... *needs* to exist... otherwise the user will see a 404 when they go to /logout. As long as it *does* exist, when the user goes to /logout, the logout mechanism will intercept the request, remove the user from the session, then redirect them to the homepage... which is configurable.

This means that, unless we've messed something up, the controller will *never* be reached. Let's *scream* in case it somehow *is* executed: Throw an exception with:

> should not be reached

38 lines | src/Controller/SecurityController.php

```
... lines 1 - 11
class SecurityController extends AbstractController
{
... lines 14 - 29
    /**
     * @Route("/logout", name="app_logout")
     */
    public function logout()
    {
        throw new \Exception('should not be reached');
    }
}
```

Let's try the flow: move over, hit log out and... before it loads, you *can* see that we're currently logged in. And now... gone! We are anonymous.

Passing Data to JavaScript on Page Load

Before we keep going with all this API & security goodness, our app has a bug. If we log in... as soon as the AJAX call finishes, we've made our Vue.js frontend smart enough to update and say that we're logged in. But when we refresh, that's gone! Gasp! Our web debug toolbar knows we're logged in... but our JavaScript does not. And... it makes sense: when we first load the page... if you look at the HTML source - you can ignore all the web debug toolbar stuff down here... the entire application looks like this. There's no HTML or JavaScript data that hints to Vue that we're authenticated.

How can we fix that? There are basically two options. First, as soon as the page loads, we could make an AJAX request to some endpoint and say:

> Hey! Who am I logged in as? Cause... I forgot?

To make that happen, we would need some sort of a /me endpoint: something that would return information about *who* we are. A useful, but not-so-RESTful endpoint.

The *other* option is quite nice: send the user data from your HTML *into* Vue on page load.

Open up the template for this page: templates/frontend/homepage.html.twig. Yep, nothing here but some small HTML to bootstrap the Vue app... though I *am* doing one interesting thing: I'm passing a prop to Vue called entrypoint. I'm not using this anywhere... but it's a cool example: entrypoint is the URL to our documentation "homepage". In theory, we could use that dynamic URL in our Vue app to figure out what *other* URLs we could call... we would use our API like a browser: surfing through links. Anyways, this shows a nice way to pass simple data into Vue.

And.. we *could* pass the current user's IRI as another prop. Inside Vue, we would then make an AJAX call to *that* URL to get the user data... so, no need for the /me endpoint. That's really the *simplest* option, though it does have a minor downside: there will be a *slight* delay on page load before our app knows who's logged in.

Serializing Data Directly to JavaScript

To avoid that AJAX call, we can dump that data *directly* into Vue. Check it out: start in src/Controller/FrontendController.php. This is the controller behind the homepage.html.twig template.. and it's not *super* impressive. Add a SerializerInterface $serializer argument... and then pass a new variable to the template called user. I want this to be the JSON-LD version of our user - the *exact* same thing we would get from making an AJAX request. Set this to $serializer->serialize() passing it $this->getUser() and jsonld for the format. If the user is *not* logged in, the new user variable will be null... but if they *are* logged in, we'll get our big, nice JSON-LD structure.

24 lines | src/Controller/FrontendController.php

```
... lines 1 - 6
use Symfony\Component\Serializer\SerializerInterface;
... lines 8 - 11
class FrontendController extends AbstractController
{
... lines 14 - 16
    public function homepage(SerializerInterface $serializer)
    {
        return $this->render('frontend/homepage.html.twig', [
            'user' => $serializer->serialize($this->getUser(), 'jsonld')
        ]);
    }
}
```

Now that we have this, create a script tag and set the data on a global variable... how about: window.user = {{ user|raw }}.

```
14 lines | templates/frontend/homepage.html.twig
... lines 1 - 2
{% block body %}
    <script>
        window.user = {{ user|raw }};
    </script>
... lines 7 - 12
{% endblock %}
```

Hey! Our user data is accessible to JavaScript!

Head over to CheeseWhizApp to use it. Generally speaking, I try not to use or reference global variables from my JavaScript. As a compromise, I like to *only* reference global variables from my *top* level component. If a deeper component needs it, I'll pass it down as a prop.

Create a new mounted() function - Vue will automatically call this after the component is "mounted" to the page - and if window.user, so, if it's *not* null, then this.user = window.user.

```
79 lines | assets/js/components/CheeseWhizApp.vue
... lines 1 - 41
<script>
... lines 43 - 45
    export default {
... lines 47 - 62
        mounted() {
            if (window.user) {
                this.user = window.user;
            }
        }
    }
</script>
... lines 70 - 79
```

It's that simple! Sneak over and refresh your browser. And... our JavaScript *instantly* knows we're logged in. If we log out... yep! Our app doesn't explode. Woo! Yea... this is kinda fun!

More Complex Authentication

Ok! Authentication... including logging out and the frontend is done! If you're feeling great about this approach for you app, awesome! But if you're screaming:

> Ryan! My app is more complex... I have multiple APIs that talk to each other... or... my API will be exposed publicly... or I have some other reason that prevents me from this simple session-based authentication!

Then... don't worry. I've already gotten *so* many questions & feedback from the start of this tutorial that we're planning to create a separate, part 2 of this security tutorial where we'll talk about other common API use-cases and the right way to authenticate within those. We'll also talk about OAuth.

But in general, if you need a more custom authentication system - perhaps you can't use json_login because your login is more complex than just handling an email & password... or you're already planning some sort of token-based authentication - you can build that system by creating a custom Guard authenticator, which is something we talk about in our security tutorial.

Next, before we dive deep into denying and granting access to our API for different users, we need to talk about CSRF attacks and SameSite cookies. It turns out, if you use cookie-based authentication like we are, you may be vulnerable to CSRF attacks. Fortunately, there's a new, beautiful way to mitigate that.

Chapter 8: SameSite Cookies & CSRF Attacks

Before we go further into API platform, we need to have a quick heart-to-heart about CSRF attacks. This is a complex topic... so I'll try to hit the highlights. If you're consuming your API from JavaScript, you have two basic options for authentication. First, you can use HttpOnly cookies, which is how sessions work. Or second, you could return some sort of "access token" on login and store that in JavaScript. Generally speaking, storing access tokens in JavaScript is a dangerous practice because it can be stolen if some bad JavaScript somehow runs on your page. If the access token have a short lifetime, that helps... but then they're less useful... because your user will need to constantly log in. Sheesh, this stuff is complex!

Anyways, *that* is the principle reason why I recommend using HttpOnly cookie-based authentication - like a session - for your JavaScript frontend. And, like I mentioned earlier, if you need this for your JavaScript front-end, it can also be used in other situations, like for authenticating a mobile app that you're building.

But... using an HttpOnly cookie - whether that's a session cookie or something clever like a JWT - is vulnerable to its *own* type of attack: a CSRF attack. Boo! The simplest example of a CSRF attack is this: an attacker puts an HTML form on *their* site but makes the action attribute submit to *our* site. Then, when someone who is logged into our site visits the bad site, the attacker tricks that user into filling out this form - making it look like they're ordering free ice cream, when in fact that endpoint on our site sends the *attacker* ice cream instead. The user has been tricked into taking some authenticated action on our site that they didn't intend.

Avoiding CSRF Attacks

This problem has traditionally been solved with a CSRF token - an extra field that must be sent on that form submit that proves that the request originated from the *real* site - not from somewhere else. Symfony's form component adds CSRF tokens automatically.

And if you Google for "dunglas csrf", you'll find a bundle called DunglasAngularCsrfBundle which helps generate and use CSRF tokens in your API. Yea, the name says "Angular", but it works with anything.

The *downside* is that using CSRF tokens in an API is... annoying: you need to manage CSRF tokens and send that field manually from your JavaScript on *every* request. If you're using cookie-based authentication and need to 100% prevent a CSRF attack for an endpoint, this is the time-tested way to do that.

Hello SameSite Cookies

But... there is a *new* way to prevent CSRF attacks that is emerging... a solution that is implemented inside browsers themselves. It's called a "SameSite" cookie... which you can read *all* about on the Internet.

The basic reason that CSRF attacks are possible is that when a user submits the form that lives on the "bad" site, any cookies that *our* domain set are sent with that request to our app... even though the request isn't "originating" from our domain. For most cookies that... should probably not happen. Instead, we should be able to say:

> Hey browsers! See this session cookie that my Symfony app is setting? I want you to *only* send that back to my app if the request *originates* from my domain.

That *is* now possible by setting a special "attribute" when you add a cookie called "SameSite".

So... because Symfony is responsible for creating the session cookie... how can we tell it to use this cool SameSite attribute? It already is. Open up config/packages/framework.yaml. If you've started a Symfony project any time recently - like we did for this tutorial - then you probably already have this key: framework.session.cookie_samesite set to lax. Yep, our session cookie is *already* setting SameSite to lax.

18 lines | config/packages/framework.yaml

```yaml
framework:
... lines 2 - 8
    session:
... lines 10 - 11
        cookie_samesite: lax
... lines 13 - 18
```

What does lax mean? Well, the other possible setting is strict. If SameSite is set to strict, then the cookie will *never* be sent when a third-party initiates a request to our site... even if they literally click a link to visit our site... meaning... they wouldn't be logged in when they arrive. The lax setting is different because the cookie *will* be sent for "top-level" GET requests... meaning GET requests where the address bar changes. That "more or less" means... when the user clicks a link to your site - though there are a few other, less visible ways that another site could covertly make a GET request to your site and have the cookie sent.

Caveats with SameSite

Anyways, lax is probably what you need and it *should* protect your API from CSRF attacks... as long as you're aware of two things. First, you need to make sure that your GET endpoints never *do* anything - they should just return data. If you, for example, allowed users to send ice cream via a GET request, then that endpoint is vulnerable to CSRF attacks. It's ok to *return* information on a GET request because, while third parties can *initiate* GET requests, they can't read the returned data, unless you go out of your way to make this possible.

And second... most... but not *all* browsers support SameSite cookies. If a user visits your site in a browser that does *not* support SameSite cookies, it will treat it like a normal cookie. That means your site will work fine, but *that* user *would* be vulnerable to a CSRF attack.

If that's a problem, you'll either need to implement CSRF tokens *or* block old browsers from using your site... since they're basically using a vulnerable browser. As you can see from caniuse, the browsers that don't support it include Opera Mini, Blackberry browser and a few other minor ones. IE 11 *does* support it on Windows 10.

I know... it's crazy! If you use API tokens in JavaScript, they can be stolen by other JavaScript. If you use the safer HttpOnly cookies, then you need to worry about CSRF protection... unless you use SameSite cookies... which protects *almost* every browser... or you could use CSRF tokens to be safest... but it complicates your life.

Oh... the world of API authentication. The typical recommendation is, regardless of what you choose to do, be aware of what you're protected against and what you're not.

Now it's time to turn to *authorization*, which answers questions like: how can we lock down certain resources or operations? How can we hide fields from some users or allow only some types of users to update specific fields? Ah... we're going to dream up *every* way of customizing access to our API.

Chapter 9: ApiResource access_control

There are two big parts to security in any app. First, how does your user authenticate? How do they log in? Honestly, *that* is the *trickiest* part... and it has really nothing to do with API Platform. We're authenticating via the json_login authenticator and a session cookie. That's a great solution for many applications. But in the bonus part 2 of the security tutorial, we'll talk about other types of applications and solutions.

Regardless of how your users authenticate, step two of security - *authorization* - will look the same. Authorization is all about denying access to read or perform different operations... and this is enforced in a way that's *independent* of how you log in. So even if the way clients of your API authenticate is *much* different than what we're doing, all this authorization stuff will still be relevant.

Denying access with access_control in security.yaml

When a user logs in - no matter *how* they authenticate or where your user data is stored - your login mechanism assigns that user a set of roles. In our app, those roles are stored in the database and we'll eventually let admin users modify them via our API. The simplest way to prevent access to an endpoint is by making sure the user has some role. And the *easiest* way to do that is via access_control in security.yaml.

We could, for example, say that every URL that matches the ^/api/cheeses regular expression - so anything that *starts* with /api/cheeses - requires ROLE_ADMIN. This is normal, boring Symfony security... and I love it!

Using access_control on your ApiResource

access_control is great for some situations, but most of the time you'll need more flexibility. In a traditional Symfony app, I typically add security to my controllers. But... in API Platform... um... we don't have any controllers! Ok, so instead of thinking about protecting each controller, we'll think about protecting each API *operation*. Maybe we want this collection GET operation to be accessible anonymously but we want to require a user to be authenticated in order to POST and create a new CheeseListing.

Open up that entity: src/Entity/CheeseListing.php. We already have an itemOperations key, which we used to remove the delete operation and also to customize the normalization groups of get. We can do something similar with a collectionOperations option. Start by setting this to get and post.

207 lines | src/Entity/CheeseListing.php

```
... lines 1 - 16
/**
 * @ApiResource(
... lines 19 - 24
 *     collectionOperations={
 *          "get",
 *          "post"
 *     },
... lines 29 - 35
 * )
... lines 37 - 46
 */
class CheeseListing
... lines 49 - 207
```

If we stopped here, this would change *nothing*. Oh... except that I have a syntax error! Silly comma! Anyways, API Platform adds two collection operations by default - get and post - so we're simply repeating what it was already doing. But *now* we can customize these operations.

For the post operation - that's how we create new cheese listings - we really need the user to be authenticated to do this. Set post to {} with a new access_control option inside. We're going to set this to a mini-expression: is_granted() passing that, inside single quotes ROLE_USER - that's the role that our app gives to *every* user.

```
207 lines | src/Entity/CheeseListing.php
... lines 1 - 24
 *     collectionOperations={
... line 26
 *         "post"={"access_control"="is_granted('ROLE_USER')"}
 *     },
... lines 29 - 207
```

Tip

access_control option was replaced by security in api-platform/core v2.5.0. If you're on a newer version - use it instead:

```
/**
 * ...
 * @ApiResource(
 *     collectionOperations={
 *         "get",
 *         "post"={"security"="is_granted('ROLE_USER')"}
 *     },
 *     ...
 * )
 * ...
 */
class CheeseListing
{
    // ...
}
```

Let's try that! The web debug toolbar tells me that I'm *not* logged in right now. Let's make a POST request... set the owner to /api/users/6 - that's my user id... though the value doesn't matter yet... nor do any of the others fields. Hit Execute and... perfect! A 401 error:

> Full authentication is required to access this resource.

If we logged in, this would work.

Let's tighten up our itemOperations too. The PUT operations - editing a CheeseListing - is an interesting case... we definitely need the user to be logged in... but we *probably* also *only* want the "owner" of a CheeseListing to be able to edit it. Let's just handle the first part now. I'll copy my whole access_control config and paste. While we're here, let's also re-add the delete operation... but maybe only admin users can do this. Check for some ROLE_ADMIN role.

```
208 lines | src/Entity/CheeseListing.php
... lines 1 - 18
 *     itemOperations={
... lines 20 - 22
 *         "put"={"access_control"="is_granted('ROLE_USER')"},
 *         "delete"={"access_control"="is_granted('ROLE_ADMIN')"}
 *     },
... lines 26 - 208
```

Go refresh the docs... yes! The DELETE operation is back! Notice that the docs are... basically "static" as far as security is concerned: it documents the *whole* API, including operations that you might not have access to: it doesn't magically read your roles and hide or show different operations. That's done on purpose, but I wanted to point it out.

Try the PUT endpoint... I think I have a CheeseListing with id 1 and... just send the title field. Another 401!

Next, our security setup is about to get smarter and more complex. The goal is to make sure that only the "owner" of a CheeseListing can update that CheeseListing... and maybe also admin users. To *really* know that things are working, I think it's time to bootstrap a basic system to functionally test our API.

Chapter 10: Bootstrapping a Test Suite

Our security requirements are about to get... pretty complicated. And so, instead of testing all of that manually and hoping we don't break anything, I think it's time to take a few minutes to get a nice functional testing system set up.

Spin over to your terminal and run:

```
$ composer require test --dev
```

This will download Symfony's test-pack, which most importantly comes with the PHPUnit bridge: a small Symfony component that wraps around PHPUnit.

When that finishes... yep - we'll put our tests in the tests/ directory and *run* PHPUnit via php bin/phpunit. That's a bit different than if you installed PHPUnit directly and you'll see why in a minute. That file - bin/phpunit was added by the recipe. And... if you run git status, you'll see that the recipe also added a phpunit.xml.dist file, which holds sensible default config for PHPUnit itself.

Configuring the PHPUnit Version

Open that file up. One of the keys here is called SYMFONY_PHPUNIT_VERSION, which at the time I recorded this is set to 7.5 in the recipe. You can leave this or change it to the latest version of PHPUnit, which for me is 8.2

Tip

PHPUnit 8.2 requires PHP 7.2. If you're using PHP 7.1, stay with PHPUnit 7.5.

34 lines | phpunit.xml.dist

```
... lines 1 - 3
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... lines 5 - 8
>
    <php>
        <ini name="error_reporting" value="-1" />
        <server name="APP_ENV" value="test" force="true" />
        <server name="SHELL_VERBOSITY" value="-1" />
        <server name="SYMFONY_PHPUNIT_REMOVE" value="" />
        <server name="SYMFONY_PHPUNIT_VERSION" value="8.2" />
    </php>
... lines 17 - 32
</phpunit>
```

But wait... why are we controlling the version of PHPUnit via some variable in an XML file? Isn't PHPUnit a dependency in our composer.json file? Actually... no! When you use Symfony's PHPUnit bridge, you *don't* require PHPUnit directly. Instead, you tell *it* which version you want, and *it* downloads it in the background.

Check it out. Run:

```
$ php bin/phpunit
```

This downloads PHPUnit in the background and installs its dependencies. That's not really an important detail... but it might surprise you the first time you run this. Where *did* it download PHPUnit? Again, that's not an important detail... but I *do* like to know what's going on. In your bin/ directory, you'll *now* find a .phpunit/ directory.

Customizing the test Database

After downloading PHPUnit, that bin/phpunit script *does* then execute our tests... except that we don't have any yet. Before we write our first one, let's tweak a few other things.

The recipe added another interesting file: .env.test. Thanks to its name, this file will *only* be loaded in the test environment... which allows *us* to create test-specific settings. For example, I like to use a different database for my tests so that running them doesn't mess with my development database.

6 lines | .env.test

```
# define your env variables for the test env here
KERNEL_CLASS='App\Kernel'
APP_SECRET='$ecretf0rt3st'
SYMFONY_DEPRECATIONS_HELPER=999999
... lines 5 - 6
```

Inside .env, copy the DATABASE_URL key, paste it in .env.test and add a little _test at the end.

6 lines | .env.test

```
... lines 1 - 4
DATABASE_URL=mysql://root:@127.0.0.1:3306/cheese_whiz_test
```

Cool, right? One of the gotchas of the .env system is that the .env.local file is *normally* loaded last and allows us to override settings for our local computer. That happens in *every* environment... except for test. In the test environment, .env & .env.test are read, but *not* .env.local. That little inconsistency exists so that everyone's test environment behaves the same way.

Anyways, now that our test environment knows to use a different database, let's create that database! Run bin/console doctrine:database:create. But since we want this command to execute in the test environment, also add --env=test:

```
$ php bin/console doctrine:database:create --env=test
```

Repeat that same thing with the doctrine:schema:create command:

```
$ php bin/console doctrine:schema:create --env=test
```

Perfect! Next, Api Platform has some *really* nice testing tools which... aren't released yet. Boo! Well... because I'm impatient, let's backport those new tools into our app and get our first test running.

Chapter 11: Backport the API Platform 2.5 Test Tools

To test our API, we're going to use PHPUnit and some other tools to make a request into our API, get back a response, and assert different things about it, like the status code, that the response is JSON and that the JSON has the right keys.

API platform recently introduced some really nice tools for doing all this. In fact it was *so* recent that... they're not released yet! When I run:

```
$ composer show api-platform/core
```

...you can see that I'm using Api Platform 2.4.5. These new features will come out in version 2.5. So if you're already on version 2.5, you can skip to the end of this video where we bootstrap our first test.

But if you're with me on 2.4.5... welcome! We're going to do some hacking and backport those features manually into our app. If you downloaded the course code, you should have a tutorial/ directory inside. If you don't have this, try downloading the code again to get it.

Copying the new Test Classes

See this Test/ directory? This contains a copy of all of those new testing classes. The only difference is that I've changed their namespaces to use App\ApiPlatform\Test instead of what they look like inside of ApiPlatform: ApiPlatform\Core\Bridge\Symfony\Bundle\Test.

That's because we're going to move *all* of this directly into our src/ directory. Start by creating a new ApiPlatform/ directory... and then copy Test/ and paste it there.

I'm also going to right click on the tutorial/ directory and say "Mark as Excluded". That tells PhpStorm to *ignore* those files... which is important so it doesn't get confused by seeing the same classes in two places.

Registering the new Test Client Service

Beyond the new classes, we need to make one other change. Open up config/ and create a new file: services_test.yaml. Thanks to its name, this file will *only* be loaded in the test environment. Inside, add services: and create a new service called test.api_platform.client: with class: App\ApiPlatform\Test\Client and arguments: ['@test.client']. Also add public: true.

8 lines | config/services_test.yaml

```yaml
services:
  # added so we can use the new API Platform test tools before
  # they are released. In API Platform 2.5, this won't be needed.
  test.api_platform.client:
    class: App\ApiPlatform\Test\Client
    arguments: ['@test.client']
    public: true
```

These two steps completely replicate what ApiPlatform will give you in version 2.5. Well... unless they change something. I don't normally show unreleased features... because they might change... but these tools are *so* useful, I just *had* to include them. When 2.5 *does* come out, there could be a few differences.

Bootstrapping the First Test

Ok, let's create our first test! Inside the tests/ directory, create a functional/ directory and then a new class: CheeseListingResourceTest.

This isn't an official convention, but because we're *functionally* testing our API... and because everything in API Platform is based on the API resource, it makes sense to create a test class for each resource: we test the CheeseListing resource via CheeseListingResourceTest.

Make this extend ApiTestCase - that's one of the new classes we just moved into our app. If you're using API Platform 2.5, the namespace will be totally different - it'll start with ApiPlatform\Core.

```
14 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 2
namespace App\Tests\Functional;

use App\ApiPlatform\Test\ApiTestCase;

class CheeseListingResourceTest extends ApiTestCase
... lines 8 - 14
```

The first thing I want to test is the POST operation - the operation that creates a new CheeseListing. A few minutes ago, under collectionOperations, we added an access control that made it so that you *must* be logged into use this. Oh... and I duplicated the collectionOperations key too! The second one is overriding the first... so let's remove the extra one.

Anyways, for our first test, I want to make sure this security *is* working. Add public function testCreateCheeseListing(). And inside, make sure this all isn't an elaborate dream with $this->assertEquals(42, 42).

```
14 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 8
    public function testCreateCheeseListing()
    {
        $this->assertEquals(42, 42);
    }
... lines 13 - 14
```

Ok! Run that with:

```
$ php bin/phpunit
```

We're alive! Next, let's turn this into a *true* functional test against our API! We'll also take care of some other details to make our tests shine.

Chapter 12: Api Tests & Assertions

Time to test our API! When someone uses our API for real, they'll use some sort of HTTP client - whether it be in JavaScript, PHP, Python, whatever. So, no surprise that to *test* our API, we'll do the exact same thing. Create a client object with $client = self::createClient().

17 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 6
class CheeseListingResourceTest extends ApiTestCase
{
    public function testCreateCheeseListing()
    {
        $client = self::createClient();
... lines 12 - 14
    }
}
```

This creates a, sort of, "fake" client, which is another feature that comes from the API Platform test classes. I say "fake" client because instead of making *real* HTTP requests to our domain, it makes them directly into our Symfony app via PHP... which just makes life a bit easier. And, side note, this $client object has the same interface as Symfony's new http-client component. So if you like how this works, next time you need to make real HTTP requests in PHP, try installing symfony/http-client instead of Guzzle.

Making Requests

Let's do this! Make a request with $client->request(): make a POST request to /api/cheeses.

How nice is that? We're going to focus our tests *mostly* on asserting security stuff. Because we haven't logged in, this request will *not* be authenticated... and so our access control rules *should* block access. Since we're *anonymous*, that should result in a 401 status code. Let's assert that! $this->assertResponseStatusCodeSame(401).

17 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 8
    public function testCreateCheeseListing()
    {
... lines 11 - 12
        $client->request('POST', '/api/cheeses');
        $this->assertResponseStatusCodeSame(401);
    }
... lines 16 - 17
```

That assertion is *not* part of PHPUnit: we get that - and a bunch of other nice assertions - from API Platform's test classes.

Let's try this! Run the test:

```
$ php bin/phpunit
```

Deprecation Warnings?

Oh, interesting. At the bottom, we see deprecation warnings! This is a feature of the PHPUnit bridge: if our tests cause deprecated code to be executed, it prints those details after running the tests. These deprecations are coming from API Platform itself. They're already fixed in the next version of API Platform... so it's nothing we need to worry about. The warnings *are* a bit annoying... but we'll ignore them.

Missing symfony/http-client

Above all this stuff... oh... interesting. It died with

> Call to undefined method: Client::prepareRequest()

What's going on here? Well... we're missing a dependency. Run

```
$ composer require symfony/http-client
```

API Platform's testing tools depend on this library. That "undefined" method is a pretty terrible error...it wasn't obvious at all how we should fix this. But there's already an issue on API Platform's issue tracker to throw a more *clear* error in this situation. It should say:

> Hey! If you want to use the testing tools, please run composer require symfony/http-client

That's what we did! I also could have added the --dev flag... since we *only* need this for our tests... but because I might need to use the http-client component later inside my actual app, I chose to leave it off.

Ok, let's try those tests again:

```
$ php bin/phpunit
```

Content-Type Header

Oooh, it failed! The response contains an error! Oh...cool - we automatically get a nice view of that failed response. We're getting back a

> 406 Not acceptable

In the body... reading the error in JSON... is not so easy... but... let's see, here it is:

> The content-type application/x-www-form-urlencoded is not supported.

We talked about this earlier! When we used the Axios library in JavaScript, I mentioned that when you POST data, there are two "main" ways to format the data in the request. The first way, and the way that *most* HTTP clients use by default, is to send in a format called application/x-www-form-urlencoded. Your browser sends data in this format when you submit a form. The second format - and the one that Axios uses by default - is to send the data as JSON.

Right now... well... we're not actually sending *any* data with this request. But if we *did* send some data, by default, this client object would format that data as application/x-www-form-urlencoded. And... looking at our API docs, our API expects data as JSON.

So even though we're not sending any data yet, the client is already sending a Content-Type header set to application/x-www-form-urlencoded. API Platform reads this and says:

> Woh, woh woh! You're trying to send me data in the wrong format! 406 status code to you!

The most straightforward way to fix this is to *change* that header. Add a third argument - an options array - with a headers option to another array, and Content-Type set to application/json.

18 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 8
    public function testCreateCheeseListing()
    {
... line 11
        $client->request('POST', '/api/cheeses', [
            'headers' => ['Content-Type' => 'application/json']
        ]);
... line 15
    }
... lines 17 - 18
```

Ok, try the tests again:

```
$ php bin/phpunit
```

This time... 400 Bad Request. Progress! Down below... we see there was a syntax error coming from some JsonDecode class. Of course! We're *saying* that we're sending JSON data... but we're actually sending *no* data. Any empty string is technically invalid JSON.

Add another key to the options array: json set to an empty array.

19 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 8
    public function testCreateCheeseListing()
    {
... line 11
        $client->request('POST', '/api/cheeses', [
... line 13
            'json' => [],
        ]);
... line 16
    }
... lines 18 - 19
```

This is a really nice option: we pass it an array, and then the client will automatically json_encode that for us and send that as the *body* of the request. It gives us behavior similar to Axios. We're not sending any data yet... because we shouldn't have to: we should be denied access *before* validation is executed.

Let's try that next! We'll also talk about a security "gotcha" then finish this test by creating a user and logging in.

Chapter 13: Logging in Inside the Test

We've added two options when we make the request: a Content-Type header, to tell API Platform that any data we send will be JSON-formatted, and a json option set to an empty array which is enough to *at least* send some valid JSON in the body of the request.

Because, before we added the json option, we got this error: a 400 Bad Request with... some details in the body that indicate we sent *invalid* JSON.

Deserialization Before Security

But... wait a second. *Assuming* our access_control security is set up correctly... shouldn't we get denied access *before* API Platform tries to deserialize the JSON data we're sending to the endpoint?

This is a little bit of a security gotcha... and even as I'm recording this, it looks like API Platform may change how this works in their next version. Progress!

When you send data to an endpoint API platform does the following things in this order. First, it deserializes the JSON into whatever resource object we're working with - like a CheeseListing object. Second, it applies the security access controls. And *third* it applies our validation rules.

Do you see the problem? It's subtle. If API Platform has any problems deserializing the JSON into the object, the user of our API will see an error about that... even if that user doesn't have access to perform the operation. The JSON syntax error is one example of this. But there are other examples, like if you send a badly-formatted date string to a date field, you'll get a normalization error about this... even if you don't have access to that operation.

This is probably not a *huge* deal in most cases, but it *is* possible for a user to get *some* details about how your endpoints work... even if that user don't have access to them. Of course... they still can't *do* anything with those endpoints... but I *do* want you to be aware of this.

But... at this *very* moment, there's a pull request open on API Platform to rename access_control to something else - probably security - and to change the behavior so that security runs *before* deserialization. In other words, if this *does* concern you, it's likely to not behave like this in the future.

Ok, but now that we *are* sending valid JSON, let's see if the test passes! Run:

```
$ php bin/phpunit
```

And... we've got it! Green!

Creating the User in the Database

We've proven that you *do* need to log in to execute this operation. So now... let's log in and make sure it works!

To do that, we *first* need to put a user in the database. Cool! We got this: $user = new User() and then fill in the email setEmail() and username with setUsername(). The only other field that's required on the user is the password. Remember, that field is the *encoded* password. For now, let's cheat and generate an encoded password manually. Find your terminal and, once again, run:

```
$ php bin/console security:encode-password
```

Let's pass this foo and... it gives me this giant, encoded password string. Copy that, and paste it into setPassword().

```
38 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 7
class CheeseListingResourceTest extends ApiTestCase
{
    public function testCreateCheeseListing()
    {
... lines 12 - 18
        $user = new User();
        $user->setEmail('cheeseplease@example.com');
        $user->setUsername('cheeseplease');
        $user->setPassword('$argon2id$v=19$m=65536,t=6,p=1$AIC3IESQ64NgHfpVQZqviw$1c7M56xyiaQFBjlUBc7T0s53/PzZCjV56lt
... lines 23 - 35
    }
}
```

The User object is ready! To save this to the database, it's the same as being inside our code: we need to get the entity manager, then call persist and flush on it. But, *normally*, to get the entity manager - or any service - we use autowiring. Tests are the *one* place where autowiring doesn't work... because you're, sort of, "outside" of your application.

Instead, we'll fetch the services from the container by their *ids*. Try this: $em = self::$container - a parent class sets the container on this nice property - ->get() and the service id. Use doctrine then say ->getManager().

You can *also* use the type-hint you use for autowiring as the service id. In other words, self::$container->get(EntityManagerInterface::class) would work super well. And actually... it's probably a bit simpler than what I did.

Anyways, now that we have the entity manager, use the famous: $em->persist($user) and $em->flush().

```
38 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 9
    public function testCreateCheeseListing()
    {
... lines 12 - 23
        $em = self::$container->get('doctrine')->getManager();
        $em->persist($user);
        $em->flush();
... lines 27 - 35
    }
... lines 37 - 38
```

POST to Login

Hey! We've got a user in the database! To test if an *authenticated* user can create a cheese listing... um... how can we authenticate as this user? Well, because we're using traditional session-based authentication... we just need to log in! Make a POST request to /login. I'll keep the header, but this time we *will* send some JSON data: email set to cheeseplease@example.com and password => 'foo'.

38 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 9
    public function testCreateCheeseListing()
    {
... lines 12 - 27
        $client->request('POST', '/login', [
            'headers' => ['Content-Type' => 'application/json'],
            'json' => [
                'email' => 'cheeseplease@example.com',
                'password' => 'foo'
            ],
        ]);
... line 35
    }
... lines 37 - 38
```

And we should probably assert that this worked. Copy the response status code assertion, paste it down here, and check that this returns 204... because 204 is what we decided to return from SecurityController.

38 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 34
        $this->assertResponseStatusCodeSame(204);
... lines 36 - 38
```

We're not *quite* yet making an authenticated request to create a new CheeseListing... but let's check our progress! Find your terminal and run:

```
$ php bin/phpunit
```

Got it! Woo! We're now logged in and ready to start making *authenticated* requests.

Except... if you've done functional tests before... you might see a problem. Try running the tests again:

```
$ php bin/phpunit
```

Explosion!

> Duplicate entry cheeseplease@example.com

coming from the database. *The* most annoying thing about functional tests is that you need to control what's in the database... including what might be "left over" in the database from a previous test. This is nothing specific to API Platform... though the API Platform team *does* have some tools to help with this.

Next, let's guarantee that the database is in a clean state before each test is executed.

Chapter 14: Resetting the Database Between Tests

One of the trickiest things about functional tests is controlling the database. In my opinion, a *perfect* world is one where the database is completely empty at the start of each test. That would give us a completely predictable state where we could create whatever User or CheeseListing objects we want and know *exactly* what's in the database.

Some people prefer to go a step further and load a predictable set of "fixtures" data before each test - like a few users and some cheese listings. That's fine, but it's not the approach I prefer. Why? Because if we can make the database empty, then each test is forced to create whatever data - like users or cheese listings - that it needs. That might sound bad at first... because... that's more work! But the end result is that each test reads like a complete story: we can see that we get a 401 status code, then we create a user, then we log in with the user we just created. A nice, complete story.

Installing Alice

So... how can we empty the database before each test? There are a few answers, but one common one you'll see in the API Platform world is called Alice. Find your terminal and install it with:

```
$ composer require alice --dev
```

This will install hautelook/alice-bundle. What does that bundle actually do? I've talked about it a few times in the past on SymfonyCasts: it allows you to specify fixtures data via YAML. It's really fun actually and has some nice features for quickly creating a set of objects, using random data and linking objects to each other. It was the inspiration behind a fixture class that we created and used in our Symfony Doctrine tutorial. The recipe creates a fixtures/ directory for the YAML files and a new command for loading that data.

The ReloadDatabaseTrait

But... what does any of that have to do with testing? Nothing! It's just a way to load fixture data and you can use it... or not use it. But AliceBundle has an *extra*, unrelated, feature that helps manage your database in the test environment.

Back in our test class... once PHPStorm finishes reindexing we're going to use a new trait: use ReloadDatabaseTrait.

41 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 6
use Hautelook\AliceBundle\PhpUnit\ReloadDatabaseTrait;
... line 8
class CheeseListingResourceTest extends ApiTestCase
{
    use ReloadDatabaseTrait;
... lines 12 - 39
}
```

That's it! *Just* by having this, before each test method is called, the trait will handle emptying our database tables and reloading our Alice YAML fixtures... which of course we don't have. So, it'll *just* empty the database.

Try it!

```
$ php bin/phpunit
```

We see even *more* deprecation warnings now - the AliceBundle has a few deprecations it needs to take care of - but it works! We can run it over and over again... it *always* passes because the database *always* starts empty. This is a *huge* step towards having dependable, readable tests.

Removing the Logging Output

While we're here, we're getting this odd log output at the top of our tests. I can tell that this is coming from Symfony... it's almost like each time an "error" log is triggered in our code, it's being printed here. That's... ok... but why? Symfony *normally* stores log files in var/logs/test.log, for the test environment.

The answer is... because we never installed a logger! Internally, Symfony ships with its *own* logger service so that if *any* other services or bundles want to log something, it's available! But that logger is *super* simple... on purpose: it's just meant as a fallback logger if nothing better is installed. Instead of writing to a file, it logs errors to stderr... which basically means they get printed to the screen from the command line.

Let's install a *real* logger:

```
$ composer require logger
```

This installs Monolog. When it finishes... try running the tests again:

```
$ php bin/phpunit
```

And... no more output! *Now* the logs are stored in var/logs/test.log. If you tail that file...

```
$ tail var/logs/test.log
```

there it is!

Next, I want to make *one* more improvement to our test suite before we get back to talking about API Platform security. I want to create a base test class with some helper methods that will enable us to move fast and write clean code in our tests.

Chapter 15: Base Test Class full of Goodies

I *love* using functional tests for my API. Even with the nice Swagger frontend, it's *almost* faster to write a test than it is to try things manually... over and over again. To make writing tests *even* nicer, I have a few ideas.

Inside src/, create a new directory called Test/. Then, add a new class: CustomApiTestCase. Make this extend the ApiTestCase that our test classes have been using so far. If you're using API platform 2.5, the namespace will start with the ApiPlatform\Core namespace.

36 lines | src/Test/CustomApiTestCase.php

```
... lines 1 - 2
namespace App\Test;
... line 4
use App\ApiPlatform\Test\ApiTestCase;
... lines 6 - 8
class CustomApiTestCase extends ApiTestCase
... lines 10 - 36
```

Base Class with createUser() and logIn()

We're creating a new base class that all our functional tests will extend. Why? Shortcut methods! There are a lot of tasks that we're going to do over and over again, like creating users in the database & logging in. To save time... and honestly, to make each test more readable, we can create reusable shortcut methods right here.

Start with protected function createUser(). We'll need to pass this the $email we want, the $password we want... and it will return the User object after it saves it to the database.

Go steal the logic for all of this from our test class: grab everything from $user = new User() to the flush call. Paste this and, at the bottom, return $user. Oh, and we need to make a couple of things dynamic: use the $email variable and the $password variable. This will temporarily *still* be the *encoded* password... but we're going to improve that in a minute. For the username, let's be clever! Grab the substring of the email from the zero position to wherever the @ symbol is located. Basically, use everything *before* the @ symbol.

36 lines | src/Test/CustomApiTestCase.php

```
... lines 1 - 8
class CustomApiTestCase extends ApiTestCase
{
    protected function createUser(string $email, string $password): User
    {
        $user = new User();
        $user->setEmail($email);
        $user->setUsername(substr($email, 0, strpos($email, '@')));
        $user->setPassword($password);

        $em = self::$container->get('doctrine')->getManager();
        $em->persist($user);
        $em->flush();

        return $user;
    }
... lines 24 - 35
}
```

We're also going to need to log in from... basically every test. Add a protected function logIn(). To accomplish this, we'll need to make a request... which means we need the Client object. Add that as the first argument followed by the same string $email and string $password, except that *this* time $password will be the plain text password. We shouldn't need to return anything.

Let's sharpen our code-stealing skills once again by going back to our test, copying these last two lines, pasting them, and making $email and $password dynamic.

```
36 lines | src/Test/CustomApiTestCase.php
... lines 1 - 24
    protected function logIn(Client $client, string $email, string $password)
    {
        $client->request('POST', '/login', [
            'headers' => ['Content-Type' => 'application/json'],
            'json' => [
                'email' => $email,
                'password' => $password
            ],
        ]);
        $this->assertResponseStatusCodeSame(204);
    }
```

Woo! Time to shorten our code! Change the test class to extend our shiny new CustomApiTestCase. Below, replace *all* the user stuff with $this->createUser('cheeseplease@example.com') and... aw... dang! I should have copied that long password. Through the power of PhpStorm... undo.... copy... redo... and paste as the second argument.

Replace the login stuff too with $this->login(), passing $client that same cheeseplease@example.com and the plain text password: foo.

```
26 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 5
use App\Test\CustomApiTestCase;
... lines 7 - 8
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 11 - 12
    public function testCreateCheeseListing()
    {
... lines 15 - 21
        $this->createUser('cheeseplease@example.com', '$argon2id$v=19$m=65536,t=6,p=1$AIC3IESQ64NgHfpVQZqviw$1c7M56xyia
        $this->logIn($client, 'cheeseplease@example.com', 'foo');
    }
}
```

Let's check things! Go tests go!

```
$ php bin/phpunit
```

If we ignore those deprecation warnings... it passed!

Encoding the User Password

Ok, this feels good. What else can we do? The weirdest thing *now* is probably that we're passing this long encoded password. It's not obvious that this is an encoded version of the password foo and... it's annoying! Heck, I had to undo 2 minutes of work earlier so I could copy it!

Let's do this properly. Replace that huge, encoded password string with just foo.

Now, inside the base test class, remove the $password variable and replace it with $encoded = and... hmm. We need to get the service out of the container that's responsible for encoding passwords. We can get that with self::$container->get('security.password_encoder'). We also could have used UserPasswordEncoderInterface::class as the service id - that's the type-hint we use normally for autowiring. Now say ->encodePassword() and pass the $user object and then the plain text password: $password. Finish this with $user->setPassword($encoded).

39 lines | src/Test/CustomApiTestCase.php

```
... lines 1 - 8
class CustomApiTestCase extends ApiTestCase
{
    protected function createUser(string $email, string $password): User
    {
... lines 13 - 16
        $encoded = self::$container->get('security.password_encoder')
            ->encodePassword($user, $password);
        $user->setPassword($encoded);
... lines 20 - 25
    }
... lines 27 - 38
}
```

Beautiful!

And this point... I'm happy! Heck, I'm thrilled! But... I *do* have *one* more shortcut idea. It'll be *pretty* common for us to want to create a user and then log in immediately. Let's make that easy! Add a protected function createUserAndLogIn()... which needs the same three arguments as the function above: $client, $email and $password... and we'll return the User. Inside say $user = $this->createUser() with $email and $password, then $this->logIn() with $client, $email, $password. At the bottom, return $user.

48 lines | src/Test/CustomApiTestCase.php

```
... lines 1 - 39
    protected function createUserAndLogIn(Client $client, string $email, string $password): User
    {
        $user = $this->createUser($email, $password);

        $this->logIn($client, $email, $password);

        return $user;
    }
```

Nice! Now we can shorten things a *little* bit more: $this->createUserAndLogIn().

25 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 8
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 11 - 12
    public function testCreateCheeseListing()
    {
... lines 15 - 21
        $this->createUserAndLogIn($client, 'cheeseplease@example.com', 'foo');
    }
}
```

Let's try it! Run:

```
$ php bin/phpunit
```

All green!

Looking back at our test, the purpose of this test was really two things. First, to make sure that if an anonymous users tries to use this endpoint, they'll get a 401 status code. And second, that an *authenticated* user *should* have access. Let's add that second part!

Make the *exact* same request as before... except that *this* time we *should* have access. Assert that we get back a 400 status code. Wait, why 400 and not 200 or 201? Well 400 because we're not actually passing real data... and so this will fail validation: a 400 error. If you wanted to make this a bit more useful, you could pass *real* data here - like a title, description, etc - and test that we get back a 201 successful status code.

```
31 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 12
    public function testCreateCheeseListing()
    {
... lines 15 - 23
        $client->request('POST', '/api/cheeses', [
            'headers' => ['Content-Type' => 'application/json'],
            'json' => [],
        ]);
        $this->assertResponseStatusCodeSame(400);
    }
... lines 30 - 31
```

Let's try this!

```
$ php bin/phpunit
```

It works! Oh, but one last, *tiny* bit of cleanup. See this headers key? We can remove that... and we have one more in CustomApiTestCase that we can also remove.

```
29 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 12
    public function testCreateCheeseListing()
    {
... line 15
        $client->request('POST', '/api/cheeses', [
            'json' => [],
        ]);
... lines 19 - 22
        $client->request('POST', '/api/cheeses', [
            'json' => [],
        ]);
... line 26
    }
... lines 28 - 29
```

But wait... didn't we need this so that API Platform knows we're sending data in the right format? Absolutely. But... when you pass the json option, the Client automatically sets the Content-Type header for us. To prove it, run the tests one last time:

```
$ php bin/phpunit
```

Everything works perfectly!

Hey! This is a great setup! So let's get back to API Platform security stuff! Right now, to edit a cheese listing, you simply need to be logged in. We need to make that smarter: you should only be able to edit a cheese listing if you are the *owner* of that cheese listing... and maybe also admin users can edit any cheese listing.

Let's do that next and *prove* it works via a test.

Chapter 16: ACL: Only Owners can PUT a CheeseListing

Back to security! We need to make sure that you can *only* make a PUT request to update a CheeseListing if you are the *owner* of that CheeseListing. As a reminder, each CheeseListing is related to one User via an $owner property. Only *that* User should be able to update this CheeseListing.

Let's start by writing a test. In the test class, add public function testUpdateCheeseListing() with the normal $client = self::createClient() and $this->createUser() passing cheeseplease@example.com and password foo. Wait, I only want to use createUser() - we'll log in manually a bit later.

51 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 29
    public function testUpdateCheeseListing()
    {
        $client = self::createClient();
        $user = $this->createUser('cheeseplease@example.com', 'foo');
... lines 34 - 48
    }
}
```

Always Start with self::createClient()

Notice that the *first* line of my test is $client = self::createClient()... even though we haven't needed to *use* that $client variable yet. It turns out, making this the first line of *every* test method is important. Yes, this of course creates a $client object that will help us make requests into our API. But it *also* boots Symfony's container, which is what gives us access to the entity manager and all other services. If we swapped these two lines and put $this->createUser() first... it would totally *not* work! The container wouldn't be available yet. The moral of the story is: always start with self::createClient().

Testing PUT /api/cheeses/{id}

Ok, let's think about this: in order to test updating a CheeseListing, we *first* need to make sure there's a CheeseListing in the database to update! Cool! $cheeseListing = new CheeseListing() and we can pass a title right here: "Block of Cheddar". Next say $cheeseListing->setOwner() and make sure this CheeseListing is *owned* by the user we just created. Now fill in the last required fields: setPrice() to $10 and setDescription().

51 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 34
        $cheeseListing = new CheeseListing('Block of cheddar');
        $cheeseListing->setOwner($user);
        $cheeseListing->setPrice(1000);
        $cheeseListing->setDescription('mmmm');
... lines 39 - 48
    }
... lines 50 - 51
```

To save, we need the entity manager! Go back to CustomApiTestCase... and copy the code we used to get the entity manager. Needing the entity manager is *so* common, let's create another shortcut for it: protected function getEntityManager() that will return EntityManagerInterface. Inside, return self::$container->get('doctrine')->getManager().

```
53 lines | src/Test/CustomApiTestCase.php
... lines 1 - 9
class CustomApiTestCase extends ApiTestCase
{
... lines 12 - 48
    protected function getEntityManager(): EntityManagerInterface
    {
        return self::$container->get('doctrine')->getManager();
    }
}
```

Let's use that: $em = $this->getEntityManager(), $em->persist($cheeseListing) and $em->batman(). Kidding. But wouldn't that be awesome? $em->flush().

```
51 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 39
        $em = $this->getEntityManager();
        $em->persist($cheeseListing);
        $em->flush();
... lines 43 - 48
    }
}
```

Great setup! Now... to the *real* work. Let's test the "happy" case first: let's test that *if* we log in with this user and try to make a PUT request to update a cheese listing, we'll get a 200 status code.

Easy peasy: $this->logIn() passing $client, the email and password. Now that we're authenticated, use $client->request() to make a PUT request to /api/cheeses/ and then the id of that CheeseListing: $cheeseListing->getId().

```
51 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 43
        $this->logIn($client, 'cheeseplease@example.com', 'foo');
        $client->request('PUT', '/api/cheeses/'.$cheeseListing->getId(), [
... line 46
        ]);
... line 48
    }
... lines 50 - 51
```

For the options, most of the time, the only thing you'll need here is the json option set to the data you need to send. Let's *just* send a title field set to updated. That's enough data for a valid PUT request.

```
51 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 44
        $client->request('PUT', '/api/cheeses/'.$cheeseListing->getId(), [
            'json' => ['title' => 'updated']
        ]);
... line 48
    }
... lines 50 - 51
```

What status code will we get back on success? You don't have to guess. Down on the docs... it tells us: 200 on success.

Assert that $this->assertResponseStatusCodeSame(200).

```
51 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 47
        $this->assertResponseStatusCodeSame(200);
    }
... lines 50 - 51
```

Perfect start! Copy the method name so we can execute *just* this test. At your terminal, run:

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

And... above those deprecation warnings... yes! It works.

But.. that's no surprise! We haven't *really* tested the security case we're worried about. What we *really* want to test is what happens if I login and try to edit a CheeseListing that I do *not* own. Ooooo.

Rename this $user variable to $user1, change the email to user1@example.com and update the email below on the logIn() call. That'll keep things easier to read... because *now* I'm going to create a *second* user: $user2 = $this->createUser() with user2@example.com and the same password.

```
58 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 29
    public function testUpdateCheeseListing()
    {
... line 32
        $user1 = $this->createUser('user1@example.com', 'foo');
        $user2 = $this->createUser('user2@example.com', 'foo');
... lines 35 - 36
        $cheeseListing->setOwner($user1);
... lines 38 - 50
        $this->logIn($client, 'user1@example.com', 'foo');
... lines 52 - 55
    }
... lines 57 - 58
```

Now, copy the *entire* login, request, assert-response-status-code stuff and paste it right *above* here: before we test the "happy" case where the *owner* tries to edit their *own* CheeseListing, let's first see what happens when a *non-owner* tries this.

Log in this time as user2@example.com. We're going to make the exact same request, but *this* time we're expecting a 403 status code, which means we *are* logged in, but we do *not* have access to perform this operation.

```
58 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 44
        $this->logIn($client, 'user2@example.com', 'foo');
        $client->request('PUT', '/api/cheeses/'.$cheeseListing->getId(), [
            'json' => ['title' => 'updated']
        ]);
        $this->assertResponseStatusCodeSame(403, 'only author can updated');
... lines 50 - 55
    }
... lines 57 - 58
```

I *love* it! With any luck, this should *fail*: our access_control is *not* smart enough to prevent this yet. Try the test:

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

And... yes! We expected a 403 status code but got back 200.

Using object.owner in access_control

Ok, let's fix this!

The access_control option - which will probably be renamed to security in API Platform 2.5 - allows you to write an "expression" inside using Symfony's expression language. This is_granted() thing is a *function* that's available in that, sort of, Twig-like expression language.

We can make this expression more interesting by saying and to add *more* logic. API Platform gives us a few *variables* to work with inside the expression, including one that represents the *object* we're working with on this operation... in other words, the CheeseListing object. That variable is called... object! Another is user, which is the currently-authenticated User or null if the user is anonymous.

Knowing that, we can say and object.getOwner() == user.

```
208 lines | src/Entity/CheeseListing.php
... lines 1 - 16
/**
 * @ApiResource(
 *     itemOperations={
... lines 20 - 22
 *         "put"={"access_control"="is_granted('ROLE_USER') and object.getOwner() == user"},
... line 24
 *     },
... lines 26 - 36
 * )
... lines 38 - 47
 */
class CheeseListing
... lines 50 - 208
```

Yea... that's it! Try the test again and...

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

It passes! I told you the security part of this was going to be easy! *Most* of the work was the test, but I *love* that I can *prove* this works.

access_control_message

While we're here, there's one other related option called access_control_message. Set this to:

> only the creator can edit a cheese listing

... and make sure you have a comma after the previous line.

211 lines | src/Entity/CheeseListing.php

```
... lines 1 - 16
/**
 * @ApiResource(
 *     itemOperations={
... lines 20 - 22
 *         "put"={
 *             "access_control"="is_granted('ROLE_USER') and object.getOwner() == user",
 *             "access_control_message"="Only the creator can edit a cheese listing"
 *         },
... line 27
 *     },
... lines 29 - 39
 * )
... lines 41 - 50
 */
class CheeseListing
... lines 53 - 211
```

If you run the test... this makes *no* difference. But this option *did* just change the message the user sees. Check it out: after the 403 status code, var_dump() $client->getResponse()->getContent() and pass that false. Normally, if you call getContent() on an "error" response - a 400 or 500 level response - it throws an exception. This tells it *not* to, which will let us see that response's content. Try the test:

59 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 49
        var_dump($client->getResponse()->getContent(true));
... lines 51 - 56
    }
}
```

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

The hydra:title says "An error occurred" but the hydra:description says:

> only the creator can edit a cheese listing.

So, the access_control_message is a nice way to improve the error your user sees. By the way, in API Platform 2.5, it'll probably be renamed to security_message.

Remove the var_dump(). Next, there's a bug in our security! Ahh!!! It's subtle. Let's find it and squash it!

Chapter 17: ACL & previousObject

Via the access_control on the PUT operation, we were able to make sure that only the *owner* of this CheeseListing can edit it. If you aren't the owner, access denied! We assert that in our test.

Now... I'm going to *trick* the security system! We're logged in as user2@example.com but the CheeseListing we're trying to update is owned by user1@example.com... which is why we're getting the 403 status code.

Right now, we've configured the serialization groups to allow for the owner field to be *updated* via the PUT request. That might sound odd, but it could be useful for admin users to be able to do this. But... this complicates things *beautifully*! Let's try changing the owner field to /api/users/ then $user2->getId().

59 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 29
    public function testUpdateCheeseListing()
    {
... lines 32 - 45
        $client->request('PUT', '/api/cheeses/'.$cheeseListing->getId(), [
            // try to trick security by reassigning to this user
            'json' => ['title' => 'updated', 'owner' => '/api/users/'.$user2->getId()]
        ]);
... lines 50 - 56
    }
}
```

Clearly, this should *not* be allowed: the user that *doesn't* own this CheeseListing is trying to edit it and... make themselves the owner! Naughty!

But... try the test:

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

It fails! We expected a 403 status code but got 200! What?

I mentioned earlier that when a request comes in, API Platform goes through three steps in a specific order. First it deserializes the JSON and updates the CheeseListing object. Second it applies our access_control security and *third* it executes our validation rules.

See the problem? By the time API Platform processes our access_control, this object has been updated! Its owner has *already* been changed! I mean, it hasn't been updated in the database yet, but the object in memory has the *new* owner. This causes access to be granted. Gasp!

Hello previous_object

There are two solutions to this depending on your API Platform version.

In API Platform 2.4 - that's our version - instead of object, use previous_object. Very simply: previous_object is the CheeseListing *before* the JSON is processed and object is the CheeseListing *after* the JSON has been deserialized.

In API Platform 2.5, you'll do something different: use the new security option instead of access_control. It's just that simple: security and access_control work identically, except that security runs *before* the object is updated from the posted data. There's also another option called security_post_denormalize if you want to run a security check *after* deserialization. In that case, the object variable is the *updated* object.

Phew! For us on API Platform 2.4, as soon as we change to previous_object... it should work! Try the test:

```
211 lines | src/Entity/CheeseListing.php
... lines 1 - 16
/**
 * @ApiResource(
 *     itemOperations={
... lines 20 - 22
 *         "put"={
 *             "access_control"="is_granted('ROLE_USER') and previous_object.getOwner() == user",
... line 25
 *         },
... line 27
 *     },
... lines 29 - 39
 * )
... lines 41 - 50
 */
class CheeseListing
... lines 53 - 211
```

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

Scroll up... all better!

access_control on User

Now that we've got a rock-solid set of access_control for CheeseListing, let's repeat this for User... because we don't have *any* access control stuff here now.

Start by saying itemOperations={}. For the get operation... let's steal an access_control from CheeseListing. Let's see... to be able to fetch a single User, let's say that you need to at *least* be logged in. So, ROLE_USER.

```
199 lines | src/Entity/User.php
... lines 1 - 15
/**
 * @ApiResource(
... lines 18 - 21
 *     itemOperations={
 *         "get"={"access_control"="is_granted('ROLE_USER')"},
... lines 24 - 25
 *     }
... lines 27 - 33
 */
class User implements UserInterface
... lines 36 - 199
```

For the put operation, you're probably going to need to be logged in and... you should probably only be able to update your *own* record. Use is_granted('ROLE_USER') and object == user.

```
199 lines | src/Entity/User.php
... lines 1 - 21
 *     itemOperations={
... line 23
 *         "put"={"access_control"="is_granted('ROLE_USER') and object == user"},
... line 25
 *     }
... lines 27 - 199
```

In this case, because we're not checking a specific property, we can safely use object instead of previous_object: you can send data to change a specific property... but not the entire object.

Finally, for delete, let's say that you can only delete a User if you're an admin: access_control looking for ROLE_ADMIN.

```
199 lines | src/Entity/User.php
... lines 1 - 21
 *     itemOperations={
... lines 23 - 24
 *         "delete"={"access_control"="is_granted('ROLE_ADMIN')"}
 *     }
... lines 27 - 199
```

Cool! Next, collectionOperations! For get, let's say that you need to be logged in... and for post, for *creating* a User... hey, that's registration! Put nothing here: this must be available to anonymous users.

```
199 lines | src/Entity/User.php
... lines 1 - 17
 *     collectionOperations={
 *         "get"={"access_control"="is_granted('ROLE_USER')"},
 *         "post"
 *     },
... lines 22 - 199
```

Very nice! We could create some tests for this, but now that we're getting comfortable... and because these access rules are still *fairly* simple, I'll skip it and test once manually.

Go refresh the docs to do that. And... syntax error! Wow, I'm *super* lazy with my commas. Try it again. My web debug toolbar tells me that I am *not* logged in. So if we try the GET collection operation... 401 status code. Perfect!

Top (Resource) Level accessControl

Until now, we've been adding the access control rules on an operation-by-operation basis. But you can also add rules at the *resource* level. Add accessControl... this time with a *capital* C - the top-level options are camel case. A few of our operations require ROLE_USER... so, if we want to, we could say accessControl="is_granted('ROLE_USER')".

```
200 lines | src/Entity/User.php
... lines 1 - 15
/**
 * @ApiResource(
 *     accessControl="is_granted('ROLE_USER')",
... lines 19 - 29
 * )
... lines 31 - 34
 */
... lines 36 - 200
```

This becomes the *default* access control that will be used for all operations *unless* an operation overrides this with their *own* access_control. This means that we don't need to repeat access_control on the get collection or get item operations. But! We *do* now need to set access_control on the post operation to look for IS_AUTHENTICATED_ANONYMOUSLY. We're overriding the default access control and making sure that *anyone* can access this operation.

```
200 lines | src/Entity/User.php
... lines 1 - 15
/**
 * @ApiResource(
... line 18
 *     collectionOperations={
 *          "get",
 *          "post"={"access_control"="is_granted('IS_AUTHENTICATED_ANONYMOUSLY')"},
 *     },
 *     itemOperations={
 *          "get",
... lines 25 - 26
 *     },
... lines 28 - 29
 * )
... lines 31 - 34
 */
... lines 36 - 200
```

Using the resource-level versus operation-level access control is a matter of taste... and resource-level controls fit better on some resources than others.

Let's make sure this works... open the POST operation, send an empty body and 500 error? Let's see... bah! Another annotation mistake. I like annotations... but I'll admit, they *can* get a bit big with API Platform... and apparently my comma key is broken today.

Let's execute that operation again and... got it! A 400 error: this value should not be blank.

Next, let's *also* making it possible for an admin user to be able to edit *any* CheeseListing. We *could* push our access_control logic further... but it's probably time to talk about voters.

Chapter 18: Access Control & Voters

The access control system in API Platform instantly gives you a lot of power: you can check for a simple role *or* write more complex logic and... it works!

But... it's also ugly. And... it can get even uglier! What if I said that a user should be able to update a CheeseListing if they are the owner of the CheeseListing *or* they are an admin user. We could... *maybe* add an or to the expression... and then we might need parentheses... No, that's not something I want to hack into my annotation expression. Instead, let's use a voter!

Generating the Voter

Voters technically have nothing to do with API Platform... but they *do* work *super* well as a way to keep your API Platform access controls clean and predictable. Find your terminal and run:

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

Call it CheeseListingVoter. I commonly have one voter for each entity or "resource" that has complex access rules. This creates src/Security/Voter/CheeseListingVoter.php.

```
42 lines | src/Security/Voter/CheeseListingVoter.php
... lines 1 - 4
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\User\UserInterface;

class CheeseListingVoter extends Voter
{
    protected function supports($attribute, $subject)
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['POST_EDIT', 'POST_VIEW'])
            && $subject instanceof \App\Entity\BlogPost;
    }
... lines 18 - 40
}
```

Updating access_control

Before we dive into the new class, go to CheeseListing. Instead of saying is_granted('ROLE_USER') and previous_object.getOwner() == user, simplify to is_granted('EDIT', previous_object).

```
211 lines | src/Entity/CheeseListing.php
... lines 1 - 16
/**
 * @ApiResource(
 *     itemOperations={
... lines 20 - 22
 *         "put"={
 *             "access_control"="is_granted('EDIT', previous_object)",
... line 25
 *         },
... line 27
 *     },
... lines 29 - 39
 * )
... lines 41 - 50
 */
class CheeseListing
... lines 53 - 211
```

This... deserves some explanation. The word EDIT... well... I just invented that. We could use EDIT or MANAGE or CHEESE_LISTING_EDIT... it's *any* word that describes the "intention" of the "access" you want to check: I want to check to see if the current user can "edit" this CheeseListing. This string will be passed to the voter and, in a minute, we'll see how to use it. We're also passing previous_object as the second argument. Thanks to this, the voter will *also* receive the CheeseListing that we're deciding access on.

How Voters Work

Here's how the voter system works: whenever you call is_granted(), Symfony loops through all of the "voters" in the system and asks each one:

> Hey! Lovely request we're having, isn't it? Do you happen to know how to decide whether or not the current user has EDIT access to this CheeseListing object?

Symfony itself comes with basically two core voters. The first knows how to decide access when you call is_granted() and pass it ROLE_ something, like ROLE_USER or ROLE_ADMIN. It determines that by looking at the roles on the authenticated user. The second voter knows how to decide access if you call is_granted() and pass it one of the IS_AUTHENTICATED_ strings: IS_AUTHENTICATED_FULLY, IS_AUTHENTICATED_REMEMBERED or IS_AUTHENTICATED_ANONYMOUSLY.

Now that we've created a class and made it extend Symfony's Voter base class, our app has a *third* voter. This means that, whenever someone calls is_granted(), Symfony will call the supports() method and pass it the $attribute - that's the string EDIT, or ROLE_USER - and the $subject, which will be the CheeseListing object in our case.

Coding the Voter

Our job here is to answer the question: do we know how to decide access for this $attribute and $subject combination? Or should another voter handle this?

We're going to design our voter to decide access if the $attribute is EDIT - and we may support other strings later... like maybe DELETE - *and* if $subject is an instanceof CheeseListing.

43 lines | src/Security/Voter/CheeseListingVoter.php

```
... lines 1 - 9
class CheeseListingVoter extends Voter
{
    protected function supports($attribute, $subject)
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, ['EDIT'])
            && $subject instanceof CheeseListing;
    }
... lines 19 - 41
}
```

If anything else is passed - like ROLE_ADMIN - supports() will return false and Symfony will know to ask a different voter.

But if we return true from supports(), Symfony will call voteOnAttribute() and pass us the same $attribute string - EDIT - the same $subject - CheeseListing object - and a $token, which contains the authenticated User object. Our job in this method is clear: return true if the user *should* have access or false if they should *not*.

Let's start by helping my editor: add @var CheeseListing $subject to hint to it that $subject will *definitely* be a CheeseListing.

After this, the generated code has a switch-case statement - a nice example for a voter that handles *two* different attributes for the same object. I'll delete the second case, but leave the switch-case statement in case we *do* want to support another attribute later.

So, if $attribute is equal to EDIT, let's put our security business logic. If $subject->getOwner() === $user, return true! Access granted. Otherwise, return false.

43 lines | src/Security/Voter/CheeseListingVoter.php

```
... lines 1 - 19
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
... lines 22 - 27
        /** @var CheeseListing $subject */
... line 29
        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case 'EDIT':
                if ($subject->getOwner() === $user) {
                    return true;
                }

                return false;
        }

        return false;
    }
... lines 42 - 43
```

That's it! Oh, in case we make a typo and pass some *other* attribute to is_granted(), the end of this function always return false to deny access. That's cool, but let's make this mistake *super* obvious. Throw a big exception:

    Unhandled attribute "%s"

and pass that $attribute.

```
43 lines | src/Security/Voter/CheeseListingVoter.php
... lines 1 - 19
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
... lines 22 - 39
        throw new \Exception(sprintf('Unhandled attribute "%s"', $attribute));
    }
... lines 42 - 43
```

I love it! Our access_control is simple: is_granted('EDIT', previous_object). If we've done our job, this will call our voter and everything will work just like before. And hey! We can check that by running out test!

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

Scroll up... all green!

Also allowing Admin Access

But... I had a different motivation originally for refactoring this into a voter: I want to *also* allow "admin" users to be able to edit *any* CheeseListing. For that, we'll check to see if the user has some ROLE_ADMIN role.

To check if a user has a role from inside a voter, we *could* call the getRoles() method on the User object... but that won't work if you're using the role hierarchy feature in security.yaml. A more robust option - and my preferred way of doing this - is to use the Security service.

Add public function __construct() with one argument: Security $security. I'll hit Alt + Enter -> Initialize Fields to create that property and set it

```
55 lines | src/Security/Voter/CheeseListingVoter.php
... lines 1 - 7
use Symfony\Component\Security\Core\Security;
... lines 9 - 10
class CheeseListingVoter extends Voter
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }
... lines 19 - 53
}
```

Inside voteOnAttribute, for the EDIT attribute, if $this->security->isGranted('ROLE_ADMIN'), return true.

55 lines | src/Security/Voter/CheeseListingVoter.php

```
... lines 1 - 27
    protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
    {
... lines 30 - 38
        switch ($attribute) {
            case 'EDIT':
... lines 41 - 44
                if ($this->security->isGranted('ROLE_ADMIN')) {
                    return true;
                }
... lines 48 - 49
        }
... lines 51 - 52
    }
... lines 54 - 55
```

That's *lovely*. I don't have a test for this... but you *could* add one in CheeseListingResourceTest by creating a *third* user, giving them ROLE_ADMIN, logging in and trying to edit the CheeseListing. Or you could *unit* test the voter itself if your logic is getting pretty crazy.

Let's at *least* make sure we didn't break anything. Go tests go!

```
$ php bin/phpunit --filter=testUpdateCheeseListing
```

All good.

I *love* voters, and this is the way I handle access controls in API Platform. Sure, if you're just checking for a role, no problem: use is_granted('ROLE_ADMIN'). But if your logic gets *any* more complex, use a voter.

Next, our API *still* requires an API client to POST an *encoded* version of a user's password when creating a new User resource. That's crazy! Let's learn how to "hook" into the "saving" process so we can intercept the plain text password and encode it.

Chapter 19: Adding the plainPassword Field

As a few of you have *already*, and *correctly* noticed... our POST operation for /api/users... doesn't really work yet! I mean, it *works*... but, for the password field, we can't POST the *plain* text password, we have to pass an encoded version of the password... which makes no sense. We are *not* expecting the users of our API to *actually* do this.

Great. So, how can we fix this? We know that the deserialization process sees these email, password and username fields and then calls the setter methods for each: setPassword(), setUsername() and setEmail(). That creates a challenge because we need to use a *service* to encode the plain-text password. And we can't access services from inside an entity.

Nope, we need some way to *intercept* the process, we need to be able to run code *after* the JSON is deserialized into a User object, but *before* it's saved to the database. One way to do this is via a Doctrine event listener or entity listener, which are more or less the same thing. That's a fine option... though things can get tricky when a user is updating their password. We talk about that on an older Symfony 3 Security Tutorial.

We're going to try a different approach - an approach that's more specific to API Platform.

Testing the POST User Endpoint

Before we get there, let's write a test to make sure this works. In the test/Functional/ directory, create a new UserResourceTest class. Make this extend our nice CustomApiTestCase and use the ReloadDatabaseTrait so the database gets emptied before each test.

27 lines | tests/Functional/UserResourceTest.php

```
    ... lines 1 - 4
use App\Test\CustomApiTestCase;
use Hautelook\AliceBundle\PhpUnit\ReloadDatabaseTrait;
    ... line 7
class UserResourceTest extends CustomApiTestCase
{
    use ReloadDatabaseTrait;
    ... lines 11 - 26
}
```

Because we're testing the POST endpoint, add public function testCreateUser() with our usual start: $client = self::createClient().

27 lines | tests/Functional/UserResourceTest.php

```
    ... lines 1 - 11
    public function testCreateUser()
    {
        $client = self::createClient();
    ... lines 15 - 25
    }
```

In this case... we don't need to put anything into the database before we start... so we can jump *straight* to the request: $client->request() to make a POST request to /api/users. And we of course need to send some data via the json key. If we look at our docs... the three fields we need are email, password and username. Ok: email set to cheeseplease@example.com, username set to cheeseplease and, here's the big change, password set *not* to some crazy encoded password... but to the plain text password. How about: brie. At the end, toast to our success by asserting that we get this 201 success status code: $this->assertResponseStatusCodeSame(201).

```
27 lines | tests/Functional/UserResourceTest.php
... lines 1 - 11
    public function testCreateUser()
    {
... lines 14 - 15
        $client->request('POST', '/api/users', [
            'json' => [
                'email' => 'cheeseplease@example.com',
                'username' => 'cheeseplease',
                'password' => 'brie'
            ]
        ]);
        $this->assertResponseStatusCodeSame(201);
... lines 24 - 25
    }
```

But... this won't be enough to make sure that the password was *correctly* encoded. Nope, to know for sure, let's try to login: $this->logIn() passing the $client, the email and the password: brie.

```
27 lines | tests/Functional/UserResourceTest.php
... lines 1 - 11
    public function testCreateUser()
    {
... lines 14 - 24
        $this->logIn($client, 'cheeseplease@example.com', 'brie');
    }
```

That's all we need! The logIn() method has a built-in assertion. So if the password is *not* correctly encoded, we'll know with a big, giant test failure.

Copy the testCreateUser() method name and let's go try it!

```
$ php bin/phpunit --filter=testCreateUser
```

Failure! Yay! The login fails with:

> Invalid credentials.

Because the password is *not* being encoded yet.

Adding plainPassword

Let's get to work. According to our test, we want the user to be able to POST a field called password. But... the password property on our User is meant to hold the *encoded* password... not the plain text password. We could, sort of, use it for both: have API Platform temporarily store the plain text password on the password field... then encoded it before the user is saved to the database.

But don't do that. First, it's just a bit dirty: using that *one* property for two purposes. And second, I really, *really* want to *avoid* storing plain text passwords in the database... which could happen if, for some reason, we introduced a bug that caused our system to "forget" to encode that field before saving.

A better option is to create a new property below this called $plainPassword. But this field will *not* be persisted to Doctrine: it exists *just* as temporary storage. Make this *writable* with @Groups({"user:write"})... then *stop* exposing the password field itself.

215 lines | src/Entity/User.php

```
... lines 1 - 35
class User implements UserInterface
{
... lines 38 - 78
    /**
     * @Groups("user:write")
     */
    private $plainPassword;
... lines 83 - 213
}
```

So, yes, this will temporarily mean that the POSTed field needs to be called *plainPassword* - but we'll fix that in a few minutes with @SerializedName.

Ok, go to the Code -> Generate menu - or Command+N on a Mac - and generate the getter and setter for this field. Oh... except I don't want those up here! I want them *all* the way at the bottom. And... we can tighten this up a bit: this will return a nullable string, the argument on the setter will be a string and all of my setters return self - they all have return $this at the end.

217 lines | src/Entity/User.php

```
... lines 1 - 35
class User implements UserInterface
{
... lines 38 - 204
    public function getPlainPassword(): ?string
    {
        return $this->plainPassword;
    }

    public function setPlainPassword(string $plainPassword): self
    {
        $this->plainPassword = $plainPassword;

        return $this;
    }
}
```

eraseCredentials

Great! The new $plainPassword field is now a *writable* field in our API instead of $password. The docs show this... the POST operation... yep! It advertises plainPassword.

Before we talk about *how* we can intercept this POST request, read the plainPassword field, encode it, and set it back on the password property, there's one *teenie*, *tiny* security detail we should handle. If you scroll down in User... eventually you'll find an eraseCredentials() method. This is something that UserInterface forces us to have. After a successful authentication, Symfony calls this method... and the idea is that we're supposed to "clear" any sensitive data that may be stored on the User - like a plain-text password - *just* to be safe. It's not *that* important, but as soon as you're storing a plain-text password on User, even though it will *never* be saved to the database, it's a good idea to clear that field here.

If we stopped now... yay! We haven't... really... done anything: we added this new plainPassword property... but nothing is using it! So, the request would ultimately explode in the database because our $password field will be null.

Next, we need to *hook* into the request-handling process: we need to run some code *after* deserialization but *before* persisting. We'll do that with a data persister.

Chapter 20: Data Persister: Encoding the Plain Password

When an API client makes a POST request to /api/users, we need to be able to run some code *after* API Platform deserializes the JSON into a User object, but *before* it gets saved to Doctrine. That code will encode the plainPassword and set it on the password property.

Introducing Data Persisters

How can we do that? One great answer is a custom "data persister". OooooOOOo. API Platform comes with only one data persister out-of-the-box, at least, only one that we care about for now: the Doctrine data persister. After deserializing the data into a User object, running security checks and executing validation, API Platform finally says:

> It's time to save this resource!

To figure out *how* to save the object, it loops over *all* of its data persisters... so... really... just one at this point... and asks:

> Hi data persister! Do you know how to "save" this object?

Because our two API resources - User and CheeseListing are both Doctrine entities, the Doctrine data persister says:

> Oh yea, I totally do know how to save that!

And then it happily calls persist() and flush() on the entity manager.

This... is awesome. Why? Because if you want to hook into the "saving" process... or if you ever create an API Resource class that is *not* stored in Doctrine, you can do that *beautifully* with a custom data persister.

Check it out: in the src/ directory - it doesn't matter where - but let's create a DataPersister/ directory with a new class inside: UserDataPersister.

This class will be responsible for "persisting" User objects. Make it implement DataPersisterInterface. You could also use ContextAwareDataPersisterInterface... which is the same, except that all 3 methods are passed the "context", in case you need the $context to help your logic.

An ASP developer stole our code block. Quick - chase them!

Anyways I'll go to the Code -> Generate menu - or Command+N on a Mac - and select "Implement Methods" to generate the three methods this interface requires.

24 lines | src/DataPersister/UserDataPersister.php

```
... lines 1 - 8
    public function supports($data): bool
    {
        // TODO: Implement supports() method.
    }

    public function persist($data)
    {
        // TODO: Implement persist() method.
    }

    public function remove($data)
    {
        // TODO: Implement remove() method.
    }
... lines 23 - 24
```

And... we're... ready! As *soon* as you create a class that implements DataPersisterInterface, API Platform will immediately start using that. This means that, *whenever* an object is saved - or removed - it will *now* call supports() on our data persister to see if we know how to handle it.

In our case, if data is a User object, we *do* support saving this object. Say that with: return $data instanceof User.

```
35 lines | src/DataPersister/UserDataPersister.php
... lines 1 - 17
    public function supports($data): bool
    {
        return $data instanceof User;
    }
... lines 22 - 35
```

As *soon* as API Platform finds *one* data persister whose supports() returns true, it calls persist() on that data persister and does *not* call any other data persisters. The core "Doctrine" data persister we talked about earlier has a really low "priority" in this system and so its supports() method is always called last. That means that our custom data persister is now *solely* responsible for saving User objects, but the core Doctrine data persister will still handle *all* other Doctrine entities.

Saving in the Data Persister

Ok, forget about encoding the password for a minute. Now that our class is *completely* responsible for saving users... we need to... yea know... make sure we save the user! We need to call persist and flush on the entity manager.

Add public function __construct() with the EntityManagerInterface $entityManager argument to autowire that into our class. I'll hit my favorite Alt + Enter and select "Initialize fields" to create that property and set it.

```
35 lines | src/DataPersister/UserDataPersister.php
... lines 1 - 6
use Doctrine\ORM\EntityManagerInterface;
... line 8
class UserDataPersister implements DataPersisterInterface
{
    private $entityManager;

    public function __construct(EntityManagerInterface $entityManager)
    {
        $this->entityManager = $entityManager;
    }
... lines 17 - 33
}
```

Down in persist(), it's pretty simple: $this->entityManager->persist($data) and $this->entityManager->flush(). Data persisters are also called when an object is being deleted. In remove(), we need $this->entityManager->remove($data) and $this->entityManager->flush().

35 lines | src/DataPersister/UserDataPersister.php

```
... lines 1 - 22
    public function persist($data)
    {
        $this->entityManager->persist($data);
        $this->entityManager->flush();
    }

    public function remove($data)
    {
        $this->entityManager->remove($data);
        $this->entityManager->flush();
    }
... lines 34 - 35
```

Congrats! We now have a data persister that... does *exactly* the same thing as the core Doctrine data persister! But... oh yea... now, we're dangerous. *Now* we can encode the plain password.

Encoding the Plain Password

To do that, we need to autowire the service responsible for encoding passwords. If you can't remember the right type-hint, find your terminal and run:

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

And... there it is: UserPasswordEncoderInterface. Add the argument - UserPasswordEncoderInterface $userPasswordEncoder - hit "Alt + Enter" again and select "Initialize fields" to create that property and set it.

48 lines | src/DataPersister/UserDataPersister.php

```
... lines 1 - 7
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
... line 9
class UserDataPersister implements DataPersisterInterface
{
... line 12
    private $userPasswordEncoder;
... line 14
    public function __construct(EntityManagerInterface $entityManager, UserPasswordEncoderInterface $userPasswordEncoder)
    {
... line 17
        $this->userPasswordEncoder = $userPasswordEncoder;
    }
... lines 20 - 46
}
```

Now, down in persist(), we know that $data will always be an instance of User. ... because that's the only time our supports() method returns true. I'm going to add a little PHPdoc above this to help my editor.

> Hey PhpStorm! $data is a User! Ok?,

48 lines | src/DataPersister/UserDataPersister.php

```
... lines 1 - 5
use App\Entity\User;
... lines 7 - 9
class UserDataPersister implements DataPersisterInterface
{
... lines 12 - 25
    /**
     * @param User $data
     */
    public function persist($data)
... lines 30 - 46
}
```

Let's think. This endpoint will be called both when *creating* a user, but also when it's being updated. And... when someone *updates* a User record, they may or may *not* send the plainPassword field in the PUT data. They would probably only send this *if* they wanted to *update* the password.

This means that the plainPassword field *might* be blank here. And if it is, we should do nothing. So, if $data->getPlainPassword(), then $data->setPassword() to $this->userPasswordEncoder->encodePassword() passing the User object - that's $data - and the plain password: $data->getPlainPassword().

That's it friends! Well, to be extra cool, let's call $data->eraseCredentials()... *just* to make sure the plain password doesn't stick around any longer than it needs to. Again, this is probably not *needed* because this field isn't saved to the database anyways... but it might avoid the plainPassword from being serialized to the session via the security system.

48 lines | src/DataPersister/UserDataPersister.php

```
... lines 1 - 28
    public function persist($data)
    {
        if ($data->getPlainPassword()) {
            $data->setPassword(
                $this->userPasswordEncoder->encodePassword($data, $data->getPlainPassword())
            );
            $data->eraseCredentials();
        }
... lines 37 - 39
    }
... lines 41 - 48
```

And... done! Aren't data persisters positively lovely?

Oh, well, we're not *quite* finished yet. The field in our API is still called plainPassword... but we wrote our test expecting that it would be called just password... which I kinda like better.

No problem. Inside User, find the plainPassword property and give it a new identity: @SerializedName("password").

218 lines | src/Entity/User.php

```
... lines 1 - 13
use Symfony\Component\Serializer\Annotation\SerializedName;
... lines 15 - 36
class User implements UserInterface
{
... lines 39 - 78
    /**
... line 80
     * @SerializedName("password")
     */
    private $plainPassword;
... lines 84 - 216
}
```

Let's check that on the docs... under the POST operation... perfect!

So... how can we see if this all works? Oh... I don't know... maybe we can run our awesome test!

```
$ php bin/phpunit --filter=testCreateUser
```

Above all the noise.. we got it!

Next, our validation rules around the plainPassword field... aren't quite right yet. And it's trickier than it looks at first: plainPassword should be required when *creating* a User, but not when *updating* it. Duh, duh, duh!

Chapter 21: Validation Groups

We're missing some validation related to the new password setup. If we send an empty POST request to /api/users, I get a 400 error because we're missing the email and username fields. But what I *don't* see is a validation error for the missing password!

No problem. We know that the password field in our API is actually the plainPassword property in User. Above this, add @Assert\NotBlank().

```
219 lines | src/Entity/User.php
... lines 1 - 36
class User implements UserInterface
{
... lines 39 - 78
    /**
... lines 80 - 81
     * @Assert\NotBlank()
     */
    private $plainPassword;
... lines 85 - 217
}
```

We're good! If we try that operation again... password *is* now required.

Sigh. But like many things in programming, fixing one problem... creates a *new* problem. This will also make the password field required when *editing* a user. Think about it: since the plainPassword field isn't persisted to the database, at the beginning of each request, after API Platform queries the database for the User, plainPassword will always be null. If an API client only sends the username field... because that's all they want to update... the plainPassword property will *remain* null and we'll get the validation error.

Testing the User Update

Before we fix this, let's add a quick test. In UserResourceTest, add a new public function testUpdateUser() with the usual $client = self::createClient() start. Then, create a user and login at the same time with $this->createUserAndLogin(). Pass that the $client and the normal cheeseplease@example.com with password foo.

```
43 lines | tests/Functional/UserResourceTest.php
... lines 1 - 7
class UserResourceTest extends CustomApiTestCase
{
... lines 10 - 27
    public function testUpdateUser()
    {
        $client = self::createClient();
        $user = $this->createUserAndLogIn($client, 'cheeseplease@example.com', 'foo');
... lines 32 - 41
    }
}
```

Great! Let's see if we can update *just* the username: use $client->request() to make a PUT request to /api/users/ $user->getId(). For the json data, pass only username set to newusername.

```
43 lines | tests/Functional/UserResourceTest.php
... lines 1 - 27
    public function testUpdateUser()
    {
... lines 30 - 32
        $client->request('PUT', '/api/users/'.$user->getId(), [
            'json' => [
                'username' => 'newusername'
            ]
        ]);
... lines 38 - 41
    }
```

This should be a totally valid PUT request. To make sure it works, use $this->assertResponseIsSuccessful()... which is a nice assertion to make sure the response is *any* 200 level status code, like 200, 201, 204 or whatever.

And... to be extra cool, let's assert that the response *does* contain the updated username: we'll test that the field *did* update. For that, there's a really nice assertion: $this->assertJsonContains(). You can pass this any subset of fields you want to check. We want to assert that the json *contains* a username field set to newusername.

```
43 lines | tests/Functional/UserResourceTest.php
... lines 1 - 27
    public function testUpdateUser()
    {
... lines 30 - 37
        $this->assertResponseIsSuccessful();
        $this->assertJsonContains([
            'username' => 'newusername'
        ]);
    }
```

It's gorgeous! Copy the method name, find your terminal, and run:

```
$ php bin/phpunit --filter=testUpdateUser
```

And... it fails! 400 bad request because of the validation error on password.

Validation Groups

So... how *do* we fix this? We want this field to be required for the POST operation... but *not* for the PUT operation. The answer is validation groups. Check this out: *every* constraint has an option called groups. These are kinda like normalization groups: you just make up a name. Let's put this into a... I don't know... group called create.

```
222 lines | src/Entity/User.php
... lines 1 - 39
class User implements UserInterface
... lines 41 - 81
    /**
... lines 83 - 84
     * @Assert\NotBlank(groups={"create"})
     */
    private $plainPassword;
... lines 88 - 220
}
```

If you *don't* specify groups on a constraint, the validator automatically puts that constraint into a group called Default. And...

by... default... the validator only executes constraints that are in this Default group.

We can see this. If you rerun the test now:

```
$ php bin/phpunit --filter=testUpdateUser
```

It passes! The NotBlank constraint above plainPassword is now *only* in a group called create. And because the validator only executes constraints in the Default group, it's not included. The NotBlank constraint is now *never* used.

Which... is not exactly what we want. We don't want it to be included on the PUT operation but we *do* want it to be included on the POST operation. Fortunately, we can specify validation groups on an operation-by-operation basis.

Let's break this access_control onto the next line for readability. Add a comma then say "validation_groups"={}. Inside, put Default then create.

```
222 lines | src/Entity/User.php
... lines 1 - 16
/**
 * @ApiResource(
... line 19
 *     collectionOperations={
... line 21
 *         "post"={
... line 23
 *             "validation_groups"={"Default", "create"}
 *         },
 *     },
... lines 27 - 33
 * )
... lines 35 - 38
 */
class User implements UserInterface
... lines 41 - 222
```

The POST operation should execute all validation constraints in *both* the Default and create groups.

Find your terminal and, this time, run *all* the user tests:

```
$ php bin/phpunit test/Functional/UserResourceTest.php
```

Green!

Next, sometimes, based on who is logged in, you might need to show additional fields or *hide* some fields. The same is true when creating or updating a resource: an admin user might have access to *write* a field that normal users can't.

Let's start getting this all set up!

Chapter 22: Conditional Field Setup

So far, we've been talking about granting or denying access to something entirely. In CheeseListing, the most complex case was when we used a voter to deny access to the PUT operation unless you are the owner of this CheeseListing or an admin user.

But there are several other ways that you might need to customize access to your API. For example, what if we have a field that we only want *readable*, or maybe *writable* by certain *types* of users? A great example of this is in User: the $roles field. Right now, the $roles field is *not* part of our API at all: *nobody* can change this field via the API.

That certainly make sense... for *most* API users. But what if we create an admin section and we *do* need this field to be editable by admin users? How can we do that?

We'll get there. For now, let's add this to the API for *all* users by adding @Groups("user:write"). This creates a *huge* security hole... so we'll come back to this in a few minutes and make sure that *only* admin users can write to this field.

223 lines | src/Entity/User.php

```
... lines 1 - 39
class User implements UserInterface
{
... lines 42 - 56
    /**
... line 58
     * @Groups({"user:write"})
     */
    private $roles = [];
... lines 62 - 221
}
```

Adding a phoneNumber Field

Let me give you another example: suppose each User has a phoneNumber field. We want that field to be *writeable* by anyone that can *write* to that User. But, for privacy reasons, we *only* want this field to be *readable* by admin users: we don't want to expose this info when a normal API client fetches data for a user.

Let's get this set up, then talk about how to make the field *conditionally* part of our API depending on who is authenticated. To add the field, run:

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

Update the User entity, add a new phoneNumber field that's a string, length, how about 50 and say "yes" to nullable: the field will be optional.

```
240 lines | src/Entity/User.php
... lines 1 - 39
class User implements UserInterface
{
... lines 42 - 89
    /**
     * @ORM\Column(type="string", length=50, nullable=true)
     */
    private $phoneNumber;
... lines 94 - 227
    public function getPhoneNumber(): ?string
    {
        return $this->phoneNumber;
    }

    public function setPhoneNumber(?string $phoneNumber): self
    {
        $this->phoneNumber = $phoneNumber;

        return $this;
    }
}
```

Cool! We now have a nice new phoneNumber property in User. Generate the migration with:

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

Let's double-check that migration file... and... it looks good: it adds the phone_number field but nothing else.

```
36 lines | src/Migrations/Version20190515191421.php
... lines 1 - 12
final class Version20190515191421 extends AbstractMigration
{
... lines 15 - 19
    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mys

        $this->addSql('ALTER TABLE user ADD phone_number VARCHAR(50) DEFAULT NULL');
    }
... lines 27 - 34
}
```

Run it with:

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

That updates our *normal* database. But because our test environment uses a *different* database, we *also* need to update that too. Instead of worrying about migrations on the test database, update it with:

```
$ php bin/console doctrine:schema:update --force --env=test
```

Now that the field is in the database, let's expose it to the API. I'll steal the @Groups() from above and put this in user:read and user:write.

241 lines | src/Entity/User.php

```
... lines 1 - 39
class User implements UserInterface
{
... lines 42 - 89
    /**
... line 91
     * @Groups({"user:read", "user:write"})
     */
    private $phoneNumber;
... lines 95 - 239
}
```

Ok! This is a perfectly boring field that is readable and writable by everyone who has access to these operations.

Testing the Conditional Behavior

Before we jump into making that field more dynamic, let's write a test for the behavior we want. Add a new public function testGetUser() and start with the normal $client = self::createClient(). Create a user & log in with $user = $this->createUserAndLogin(), email cheeseplease@example.com, password foo and... I forgot the first argument: $client.

73 lines | tests/Functional/UserResourceTest.php

```
... lines 1 - 8
class UserResourceTest extends CustomApiTestCase
{
... lines 11 - 44
    public function testGetUser()
    {
        $client = self::createClient();
        $user = $this->createUserAndLogIn($client, 'cheeseplease@example.com', 'foo');
... lines 49 - 71
    }
}
```

That method creates a *super* simple user: with just the username, email and password fields filled in. But this time, we *also* want to set the phoneNumber. We can do that manually with $user->setPhoneNumber('555.123.4567'), and then saving it to the database. Set the entity manager to an $em variable - we'll need it a few times - and then, because we're *updating* the User, all we need is $em->flush().

73 lines | tests/Functional/UserResourceTest.php

```
... lines 1 - 44
    public function testGetUser()
    {
... lines 47 - 49
        $user->setPhoneNumber('555.123.4567');
        $em = $this->getEntityManager();
        $em->flush();
... lines 53 - 71
    }
```

In this test, we're *not* logged in as an admin user: we're logged in by the user that we're *fetching*. Our goal is for the API to return the phoneNumber field *only* to admin users. It's a little weird, but, for now, I don't even want users to be able to see their *own* phone number.

Let's make a request and assert that: $client->request() to make a GET request to /api/users/ and then $user->getId(). To start, let's do a sanity check: $this->assertJsonContains() to make sure that the response contains the the username field set to cheeseplease.

73 lines | tests/Functional/UserResourceTest.php

```
... lines 1 - 44
    public function testGetUser()
    {
... lines 47 - 53
        $client->request('GET', '/api/users/'.$user->getId());
        $this->assertJsonContains([
            'username' => 'cheeseplease'
        ]);
... lines 58 - 71
    }
```

But what we *really* want assert is that the phoneNumber field is *not* in the response. There's no fancy assert for this so... we'll do it by hand. Start with $data = $client->getResponse()->toArray().

73 lines | tests/Functional/UserResourceTest.php

```
... lines 1 - 44
    public function testGetUser()
    {
... lines 47 - 58
        $data = $client->getResponse()->toArray();
... lines 60 - 71
    }
```

This handy function will see that the response is JSON and automatically json_decode() it into an array... or throw an exception if something went wrong. Now we can use $this->assertArrayNotHasKey('phoneNumber', $data).

73 lines | tests/Functional/UserResourceTest.php

```
... lines 1 - 44
    public function testGetUser()
    {
... lines 47 - 59
        $this->assertArrayNotHasKey('phoneNumber', $data);
... lines 61 - 71
    }
```

Boom! That's enough to make the test fail... because that field *should* be in the response right now. Copy the testGetUser method name and... try it:

```
$ php bin/phpunit --filter=testGetUser
```

Yay! Failure!

> Failed asserting that array does not have the key phoneNumber.

Next, let's finish the *second* half of the test - asserting that an admin user *can* see this field. Then, we'll discuss the strategy for making the phoneNumber field *conditionally* available in our API.

Chapter 23: Testing, Updating Roles & Refreshing Data

This test is failing... which is great! It proves that our end goal - *only* returning the phoneNumber field to users authenticated as an admin - is totally *not* working yet. Before we get it working, let's finish the test: let's make the user an admin and assert that the phoneNumber field *is* returned.

First, "promote" this user to be an admin: call $user->setRoles() and pass this an array with ROLE_ADMIN inside.

Services are Recreated between Requests

Easy! But... we need to be careful with what we do next.

Let me explain: in the "real" world - when you're not inside a test - each request is handled by a separate PHP process where all the service objects - like the entity manager - are instantiated *fresh*. That's just... how PHP works: objects are created during the request and then trashed at the end of the request.

But in our test environment, that's *not* the case: we're *really* making *many*, sort of, "fake" requests into our app all within the *same* PHP process. This means that, in theory, if we made a request that, for some reason, changed a property on a service object, when we make the *next* request... that property would *still* be changed. In the test environment, one request can *affect* the next requests. That's *not* what we want because it's *not* how the real world works.

Not to worry: Symfony handles this automatically for us. Before each request with the client, the client "resets" the container and recreates it from scratch. That gives each request an "isolated" environment because each request will create *new* service objects.

But it *also* affects the service objects that we use *inside* our test... and it can be confusing. Stick with me through the explanation, and then I'll give you some rules to live by at the end.

Check out the entity manager here on top. Internally, the entity manager keeps track of all of the objects that it's fetched or persisted. This is called the identity map. Then, when we call flush(), it loops over all of those objects, finds the ones that have changed and runs any queries it needs.

Up here, this entity manager *does* have this User object in its identity map, because the createUserAndLogIn() method just *used* that entity manager to persist it. But when we make a request, two things happen. First, the *old* container - the one we've been working with inside the test class so far, is "reset". That means that the "state" of a lot of services is reset back to its initial state, back when the container was originally created. And second, a totally new container is created for the request itself.

This has two side effects. First, the "identity map" on this entity manager was just "reset" back to empty. It means that it has *no* idea that we ever persisted this User object. And second, if you called $this->getEnityManager() now, it would give you the EntityManager that was just used for the last request, which would be a different object than the $em variable. That detail is less important.

On a high level, you basically want to think of everything *above* $client->request() as code that runs on one page-load and everything *after* $client->request() as code that runs on a totally different page load. If the two parts of this method really *were* being executed by two different requests, I wouldn't expect the entity manager down here to be aware that I persisted this User object on some previous request.

Ok, I get it, I lost you. What's going on behind the scenes is technical - it confuses me too sometimes. But, here's what you need to know. After calling $user->setRoles(), if we *just* said $em->flush(), nothing would save. The $em variable was "reset" and so it doesn't know it's supposed to be "managing" this User object.

Here's the rule to live by: after each request, if you need to work with an entity - whether to read data or update data - do *not* re-use an entity object you were working with from *before* that request. Nope, query for a new one:
$user = $em->getRepository(User::class)->find($user->getId()).

We would need to do the *same* thing if we wanted to *read* data. If we made a PUT request up here to edit the user and wanted to assert that a field *was* updated in the database, we should query for a *new* User object down here. If we used the *old* $user variable, it would hold the old data, even though the database *was* successfully updated.

Logging in as Admin

So I'll put a comment about this: we're refreshing the user and elevating them to an admin. Saying ->flush() is enough for this to save because we've just queried for this object.

```
73 lines | tests/Functional/UserResourceTest.php
... lines 1 - 8
class UserResourceTest extends CustomApiTestCase
{
... lines 11 - 44
    public function testGetUser()
    {
... lines 47 - 61
        // refresh the user & elevate
        $user = $em->getRepository(User::class)->find($user->getId());
        $user->setRoles(['ROLE_ADMIN']);
        $em->flush();
... lines 66 - 71
    }
}
```

Below, say $this->logIn() and pass this $client and... the same two arguments as before: the email & password.

```
73 lines | tests/Functional/UserResourceTest.php
... lines 1 - 44
    public function testGetUser()
    {
... lines 47 - 65
        $this->logIn($client, 'cheeseplease@example.com', 'foo');
... lines 67 - 71
    }
```

Wait... why do we need to log in again? Weren't we already logged in... and didn't we just change this user's roles in the database? Yep! Unrelated to the test environment, in order for Symfony's security system to "notice" that a user's roles were updated in the database, that user needs to log back in. It's a quirk of the security system and hopefully one we'll fix soon. Heck, I *personally* have a two year old pull request open to do this! I gotta finish that!

Anyways, *that's* why we're logging back in: so that the security system sees the updated roles.

Finally, down here, we can do the same $client->request() as before. In fact, let's copy it from above, including the assertJsonContains() part. But this time, assert that there *should* be a phoneNumber field set to 555.123.4567.

```
73 lines | tests/Functional/UserResourceTest.php
... lines 1 - 44
    public function testGetUser()
    {
... lines 47 - 67
        $client->request('GET', '/api/users/'.$user->getId());
        $this->assertJsonContains([
            'phoneNumber' => '555.123.4567'
        ]);
    }
```

Phew! Ok, we already know this will fail: when we make a GET request for a User, it *is* currently returning the phoneNumber field: the test is failing on assertArrayNotHasKey().

Dynamic Fields

So... now that we have this big, fancy test... how *are* we going to handle this? How *can* we make some fields *conditionally* available?

Via... dynamic normalization groups.

When you make a request for an operation, the normalization groups are determined on an operation-by-operation basis. In the case of a User, API Platform is using user:read for normalization and user:write for denormalization. In CheeseListing, we're customizing it even deeper: when you get a single CheeseListing, it uses the cheese_listing:read and cheese_listing:item:get groups.

That's great. But all of these groups are still static: we can't change them on a user-by-user basis... or via *any* dynamic information. You can't say:

> Oh! This is an admin user! So when we normalize this resource, I want to include an *extra* normalization group.

But... doing this *is* possible. On User, above $phoneNumber, we're going to leave the user:write group so it's *writable* by anyone with access to a write operation. But instead of user:read, change this to admin:read.

241 lines | src/Entity/User.php

```
... lines 1 - 39
class User implements UserInterface
{
... lines 42 - 89
    /**
... line 91
     * @Groups({"admin:read", "user:write"})
     */
    private $phoneNumber;
... lines 95 - 239
}
```

That's a *new* group name that I... just invented. Nothing uses this group, so if we try the test now:

```
$ php bin/phpunit --filter=testGetUser
```

It fails... but gets further! It fails on UserResourceTest line 68... it's failing down here. We successfully made phoneNumber *not* return when we fetch a user.

Next, we're going to create something called a "context builder", which will allow us to *dynamically* add this admin:read group when an "admin" is trying to normalize a User resource.

Chapter 24: Context Builder & Service Decoration

When you make a GET request for a collection of users or a single user, API Platform will use the same normalization group: user:read. This means the response will *always* contain the email, username and cheeseListings fields.

Now we need to do something smarter: we need to be able to *also* normalize using another group - admin:read - but *only* if the authenticated user has ROLE_ADMIN. The key to doing this is something called a "context builder".

Remember: when API Platform, or really, when Symfony's serializer goes through its normalization or denormalization process, it has something called a "context", which is a fancy word for "options that are passed to the serializer". The most common "option", or "context" is groups. The context is normally hardcoded via annotations but we can *also* tweak it dynamically.

Creating the Context Builder

Google for "API Platform context builder" and find the serialization page. If we scroll down a bit... it talks about changing the serialization context dynamically and gives an example. Steal this BookContextBuilder code. This class can live anywhere in src/, but to follow the docs, let's create a Serializer/ directory and a new PHP class inside of that called AdminGroupsContextBuilder... because the purpose of this context builder will be to add an admin:read group or an admin:write group to *every* resource if the authenticated user is an admin.

Paste the code and rename the class to AdminGroupsContextBuilder.

32 lines | src/Serializer/AdminGroupsContextBuilder.php

```
... lines 1 - 2
namespace App\Serializer;

use ApiPlatform\Core\Serializer\SerializerContextBuilderInterface;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Authorization\AuthorizationCheckerInterface;

final class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
    private $decorated;
    private $authorizationChecker;

    public function __construct(SerializerContextBuilderInterface $decorated, AuthorizationCheckerInterface $authorizationChecker)
    {
        $this->decorated = $decorated;
        $this->authorizationChecker = $authorizationChecker;
    }

    public function createFromRequest(Request $request, bool $normalization, ?array $extractedAttributes = null): array
    {
        $context = $this->decorated->createFromRequest($request, $normalization, $extractedAttributes);
        $resourceClass = $context['resource_class'] ?? null;

        if ($resourceClass === Book::class && isset($context['groups']) && $this->authorizationChecker->isGranted('ROLE_ADMIN') && f
            $context['groups'][] = 'admin:input';
        }

        return $context;
    }
}
```

Service Declaration & Decoration

A lot of times in Symfony, when you're "hooking" into some existing process, like hooking into the "context building" process, all you need to do is create a class, make it implement some interface or extend some base class and... boom! Symfony or API Platform magically sees it and uses it. That happened earlier when we created the voter: no config was needed beyond the class itself.

But, that's *not* true for a context builder: this needs some service config... some interesting service config. Open config/services.yaml and go the bottom. Our new class *is* already registered as a service... that happens automatically. But we need to override that service definition to add some extra config. Start with the class name... which is also the service id: App\Serializer\AdminGroupsContextBuilder. Then, I'll look back at the docs, copy these three lines... and paste. Change the BookContextBuilder part of the argument to AdminGroupsContextBuilder.

33 lines | config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 29
    App\Serializer\AdminGroupsContextBuilder:
        decorates: 'api_platform.serializer.context_builder'
        arguments: [ '@App\Serializer\AdminGroupsContextBuilder.inner' ]
```

If you've never seen this decorates option before, welcome to service decoration! It's a slightly advanced feature of Symfony's container but it is *incredibly* powerful... and API Platform uses it in several places.

Internally, API Platform already has a "context builder": it has a *single* service that it calls that's responsible for building the "context" in every situation. That service is what reads our normalizationContext annotation config.

But now, *we* want to hook into that process. But... we don't want to *replace* the core functionality. No, we want to *add* to it. We do this via service decoration. The *id* of the *core* "context builder" service is api_platform.serializer.context_builder.

Tip

GraphQL comes with its own services and you would need to use api_platform.graphql.serializer.context_builder service instead.

So our config says:

> Please register a new service called App\Serializer\AdminGroupsContextBuilder and make it *replace* the core api_platform.serializer.context_builder service.

Yep, this means that, whenever API Platform needs to build the "context", it will now use *our* service *instead* of the original, *core* service. If we *only* did this, our class would *replace* the core class - not something we want. Fortunately, the decoration feature allows us to pass the *original* service as an argument, by using the same id as our service plus .inner. Yep, this weird string is a magic way to reference the original, core context builder service.

If you look in AdminGroupsContextBuilder, it implements SerializerContextBuilderInterface. That's the interface we must implement to be a "context builder". The first constructor argument *also* implements SerializerContextBuilderInterface and is called $decorated. *This* is the original, core API Platform context builder service.

The only method this interface requires is createFromRequest(), which API Platform calls when it's building the context. Check out that first line: $context = $this->decorated->createFromRequest().

Yep, we're calling the *core* context builder, passing it all the arguments, and letting *it* do its normal logic, like reading the normalizationContext and denormalizationContext config off of our annotations. *Then*, below this, we can *extend* the context with our own logic.

Phew! This may look complicated the first time you see it, but I *love* this feature. On an object-oriented level, this is the "decorator" pattern: the recommended way to "extend" the functionality of a class. The config in services.yaml is Symfony's way of letting you "decorate" any core service.

At this point, *our* service *is* being used as the context builder. Next, let's fill in our custom logic to add the dynamic groups.

Chapter 25: Context Builder: Dynamic Fields/Groups

Here's the goal: add logic to our new context builder so that, if the currently- authenticated user has ROLE_ADMIN, an extra admin:output group is *always* added during normalization.

Delete the existing code... though it's *pretty* close to what we're going to do. Then say $isAdmin =. When we copied this class, in addition to the "decorated" context builder, it came with a *second* argument: AuthorizationCheckerInterface. This is a service that allows us to check whether or not a user has a role.

But wait... when we needed to do that in our voter, we autowired a different service via the Security type-hint. Well... these are both ways to do the *exact* same thing: use whichever you like. Yep, we can say $isAdmin = $this->authorizationChecker->isGranted('ROLE_ADMIN'). Then, if $context['groups'] and $isAdmin... we should add the extra group!

33 lines | src/Serializer/AdminGroupsContextBuilder.php

```
... lines 1 - 8
final class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
... lines 11 - 19
    public function createFromRequest(Request $request, bool $normalization, ?array $extractedAttributes = null): array
    {
... lines 22 - 23
        $isAdmin = $this->authorizationChecker->isGranted('ROLE_ADMIN');

        if (isset($context['groups']) && $isAdmin) {
... line 27
        }
... lines 29 - 30
    }
}
```

But... why am I checking *if* $context['groups']? Well, first, I should probably be checking if isset($context['groups']). And second... it doesn't really matter for us. In *theory*, if you had a resource with *no* groups configured, it would mean the serializer should serialize *all* fields. In that situation, we wouldn't want to add the admin:output group because it would actually cause *less* fields to be serialized. But because I like to *always* specify normalization and denormalization groups, this isn't a real situation: $context['groups'] will *always* have something in it at this point.

Add the new group with $context['groups'][] = 'admin:read'. Right? Well, it's not *that* simple. This createFromRequest() method is called both when the object is being serialized to JSON - so when it's being "normalized" - *and* when the JSON is being *deserialized* to the object - when it's being "denormalized". That's what this normalization flag here is telling us.

Cool! We can say, *if* the object is being normalized, add admin:read, else, add admin:write.

33 lines | src/Serializer/AdminGroupsContextBuilder.php

```
... lines 1 - 19
    public function createFromRequest(Request $request, bool $normalization, ?array $extractedAttributes = null): array
    {
... lines 22 - 25
        if (isset($context['groups']) && $isAdmin) {
            $context['groups'][] = $normalization ? 'admin:read' : 'admin:write';
        }
... lines 29 - 30
    }
... lines 32 - 33
```

We're done! I'll even remove this $resourceClass thing. That tells us the *class* of the object that's being serialized or deserialized... which we don't need because we're adding these groups to *every* resource.

Side Note: You can Decorate Multiple Times

Side note: we've only created *one* context builder so far, but it's legal to create as *many* as you want. In the last chapter, I talked about how our new service "decorates" and "replaces" the core context builder. But you could repeat this 3 times - each time "decorating" that same core service. You can do this because Symfony is smart: it will create 3 layers of decoration. This $decorated property *might* be the "core" service... or it could just be the *next* decorated service... which itself would call the *next*, until the core one is eventually called.

If that didn't make sense, don't sweat it. My point is: if you want to control the context in multiple ways, you could smash all that logic into one context builder or create *several*. The service config for each will look identical to the one we created.

Let's head over and run our tests:

```
$ php bin/phpunit --filter=testGetUser
```

This time... it passes! That test proves that a normal user will *not* get the phoneNumber field.... but as soon as we give that user ROLE_ADMIN and make *another* request, phoneNumber *is* returned! Mission accomplished!

Making "roles" Writeable by only an Admin

And now we can fix the *huge* security problem I created a few minutes ago: instead of allowing *anyone* to set the $roles property on User, only *admin* users should be able to do this.

Before we make that change, let's tweak our test to check for this. In testUpdateUser, let's *also* try to pass a roles field set to ROLE_ADMIN. Because we're *not* logged in as an admin user, once we've finished our work, the roles field should simply be ignored. It won't cause a validation error... it just won't be processed at all.

79 lines | tests/Functional/UserResourceTest.php

```
... lines 1 - 8
class UserResourceTest extends CustomApiTestCase
{
... lines 11 - 28
    public function testUpdateUser()
    {
... lines 31 - 33
        $client->request('PUT', '/api/users/'.$user->getId(), [
            'json' => [
... line 36
                'roles' => ['ROLE_ADMIN'] // will be ignored
            ]
        ]);
... lines 40 - 48
    }
... lines 50 - 78
}
```

To make sure it's ignored, at the bottom, say $em = $this->getEntityManager() and then query for a fresh user from the database: $user = $em->getRepository(User:class)->find($user->getId()). I'll put some PHPDoc above this to tell PhpStorm that this will be a User object. Finish with $this->assertEquals() that we expect ['ROLE_USER'] to be returned from $user->getRoles().

79 lines | tests/Functional/UserResourceTest.php

```
... lines 1 - 28
    public function testUpdateUser()
    {
... lines 31 - 44
        $em = $this->getEntityManager();
        /** @var User $user */
        $user = $em->getRepository(User::class)->find($user->getId());
        $this->assertEquals(['ROLE_USER'], $user->getRoles());
    }
... lines 50 - 79
```

Why ROLE_USER? Because even if the roles property is empty in the database... the getRoles() method *always* returns *at least* ROLE_USER.

Let's make sure this fails. Copy testUpdateUser and run that test:

```
$ php bin/phpunit --filter=testUpdateUser
```

It... yes - fails! Our API *does* let us write to the roles property. How do we fix this? You probably already know... and it's *gorgeously* simple. Change the group to admin:write.

241 lines | src/Entity/User.php

```
... lines 1 - 39
class User implements UserInterface
{
... lines 42 - 56
    /**
... line 58
     * @Groups({"admin:write"})
     */
    private $roles = [];
... lines 62 - 239
}
```

Run the test again:

```
$ php bin/phpunit --filter=testUpdateUser
```

This time... it passes! Context builders are *awesome*.

Context Builders & Documentation

Though... they do have one downside: the dynamic groups are *not* reflected anywhere in the documentation. Refresh the docs... then open the GET operation for a single User. The docs say that this will return a User model with email, username and cheeseListings fields. It does *not* say that phoneNumber will be returned. And even if we logged in as an admin user, this won't change - there would *still* be no mention of phoneNumber. The docs are *static*. If an admin user makes this request, that JSON *will* contain a phoneNumber field, but it won't say that in the docs.

Next, we're going to do a crazy experiment. Because we're following tight naming conventions for our groups - like cheese_listing:output, cheese_listing:input and even operation-specific groups like cheese_listing:item:get, could we use a context builder to *automatically* set these groups... so we don't need to manually manage them via annotations? The answer is... yes. But things get *really* interesting if you want your docs to reflect this.

Chapter 26: Automatic Serialization Groups

I want to show you something... kind of experimental. We've been following a strict naming convention inside our API resource classes for the normalization and denormalization groups. For normalization, we're using cheese_listing:read and for denormalization, cheese_listing:write. When we need even *more* control, we're adding an operation-specific group like cheese_listing:item:get.

If you have a lot of different behaviors for each operation, you may end up with a *lot* of these normalization_context and denormalization_context options... which is a bit ugly... but also error prone. When it comes to controlling which fields are and are *not* exposed to our API, this stuff is important!

So here's my idea: in AdminGroupsContextBuilder, we have the ability to dynamically add groups. Could we detect that we're *normalizing* a CheeseListing *item* and automatically add the cheese_listing:read and cheese_listing:item:get groups? The answer is... of course! But the final solution may not look *quite* like you expect.

Adding the Automatic Groups

Let's start in AdminGroupsContextBuilder. At the bottom, I'm going to paste in a new method: private function addDefaultGroups(). You can copy the method from the code block on this page. This looks at which entity we're working with, whether it's being normalized or denormalized and the exact *operation* that's currently being executed. It uses this information to always add *three* groups. The first is easy: {class}:{read/write}. So user:read, cheese_listing:read or cheese_listing:write. That matches the main groups we've been using.

65 lines | src/Serializer/AdminGroupsContextBuilder.php

```
... lines 1 - 8
final class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
... lines 11 - 37
    private function addDefaultGroups(array $context, bool $normalization)
    {
        $resourceClass = $context['resource_class'] ?? null;

        if (!$resourceClass) {
            return;
        }

        $shortName = (new \ReflectionClass($resourceClass))->getShortName();
        $classAlias = strtolower(preg_replace('/[A-Z]/', '_\\0', lcfirst($shortName)));
        $readOrWrite = $normalization ? 'read' : 'write';
        $itemOrCollection = $context['operation_type'];
        $operationName = $itemOrCollection === 'item' ? $context['item_operation_name'] : $context['collection_operation_name'];

        return [
            // {class}:{read/write}
            // e.g. user:read
            sprintf('%s:%s', $classAlias, $readOrWrite),
            // {class}:{item/collection}:{read/write}
            // e.g. user:collection:read
            sprintf('%s:%s:%s', $classAlias, $itemOrCollection, $readOrWrite),
            // {class}:{item/collection}:{operationName}
            // e.g. user:collection:get
            sprintf('%s:%s:%s', $classAlias, $itemOrCollection, $operationName),
        ];
    }
}
```

The next is more specific: the class name, then item or collection, which is whether this is an "item operation" or a "collection operation" - then read or write. If we're making a GET request to /api/users, this would add user:collection:read.

The last is the *most* specific... and is kind of redundant unless you create some custom operations. Instead of read or write, the last part is the operation name, like user:collection:get.

To use this method, back up top, add $context['groups'] = $context['groups'] ?? [];. That will make sure that if the groups key does *not* exist, it will be added and set to an empty array. Now say $context['groups'] = array_merge() of $context['groups'] and $this->addDefaultGroups(), which needs the $context and whether or not the object is being normalized. So, the $normalization argument.

65 lines | src/Serializer/AdminGroupsContextBuilder.php

```
... lines 1 - 19
    public function createFromRequest(Request $request, bool $normalization, ?array $extractedAttributes = null): array
    {
... lines 22 - 23
        $context['groups'] = $context['groups'] ?? [];
        $context['groups'] = array_merge($context['groups'], $this->addDefaultGroups($context, $normalization));
... lines 26 - 35
    }
... lines 37 - 65
```

We can remove the $context['groups'] check in the if statement because it will *definitely* be set already.

```
65 lines | src/Serializer/AdminGroupsContextBuilder.php
... lines 1 - 19
    public function createFromRequest(Request $request, bool $normalization, ?array $extractedAttributes = null): array
    {
... lines 22 - 28
        if ($isAdmin) {
            $context['groups'][] = $normalization ? 'admin:read' : 'admin:write';
        }
... lines 32 - 35
    }
... lines 37 - 65
```

Oh, and just to clean things up, let's remove any possible duplications: $context['groups'] = array_unique($context['groups']).

```
65 lines | src/Serializer/AdminGroupsContextBuilder.php
... lines 1 - 19
    public function createFromRequest(Request $request, bool $normalization, ?array $extractedAttributes = null): array
    {
... lines 22 - 32
        $context['groups'] = array_unique($context['groups']);
... lines 34 - 35
    }
... lines 37 - 65
```

That's it! We can *now* go into CheeseListing, for example, and remove the normalization and denormalization context options.

```
207 lines | src/Entity/CheeseListing.php
... lines 1 - 16
/**
 * @ApiResource(
 *     itemOperations={
... lines 20 - 25
 *     },
 *     collectionOperations={
... lines 28 - 29
 *     },
 *     shortName="cheeses",
 *     attributes={
... lines 33 - 34
 *     }
 * )
... lines 37 - 46
 */
class CheeseListing
... lines 49 - 207
```

In fact, let's prove everything still works by running the tests:

```
$ php bin/phpunit
```

Even though we just *drastically* changed how the groups are added, everything *still* works!

Ah! My Documentation

So... that was easy, right? Well... remember a few minutes ago when we discovered that the documentation does *not* see any groups that you add via a context builder? Yep, now that we've removed the normalizationContext and denormalizationContext options... our docs are going to start falling apart.

Refresh the docs... and go look at the GET operation for a single CheeseListing item. This... actually... *still* shows the correct fields. That's because we're *still* manually - and now *redundantly* - setting the normalization_context for that one operation.

But if you look at the *collection* GET operation... it says it will return *everything*: id, title, description, shortDescription, price, createdAt, createdAtAgo, isPublished *and* owner. Spoiler alert: it will *not* actually return all of those fields.

If you try the operation... and hit Execute... it *only* returns the fields we expect. So... we've added these "automatic" groups... which is kinda nice. But we've positively *destroyed* our documentation. Can we have both automatic groups *and* good documentation? Yes! By leveraging something called a resource metadata factory: a wild, low-level, advanced feature of API Platform.

Let's dig into that next.

Chapter 27: Resource Metadata Factory: Dynamic ApiResource Options

Using a context builder to dynamically add groups is a great option when the groups you're adding are *contextual* to who is authenticated... like we add admin:read *only* if the user is an admin. That's because the context builder isn't taken into account when your documentation is built. For these "extra" admin fields... that may not be a huge deal. But the more you put into the context builder, the less perfect your docs become.

However, if you're using a context builder to do something crazy like what *we're* trying now - adding a bunch of groups in *all* situations - then things really start to fall apart. Our docs are now *very* inaccurate for *all* users.

How *can* we customize the normalization and denormalization groups *and* have the docs notice the changes? The answer is with a "resource metadata factory"... which is... at least at first... as dark and scary as the name sounds.

Creating the Resource Metadata Factory

Inside the ApiPlatform/ directory, create a new class called AutoGroupResourceMetadataFactory. Make this implement ResourceMetadataFactoryInterface and then take a break... cause we just created one *seriously* scary-looking class declaration line.

```
24 lines | src/ApiPlatform/AutoGroupResourceMetadataFactory.php
... lines 1 - 2
namespace App\ApiPlatform;
... line 4
use ApiPlatform\Core\Metadata\Resource\Factory\ResourceMetadataFactoryInterface;
... lines 6 - 7
class AutoGroupResourceMetadataFactory implements ResourceMetadataFactoryInterface
{
... lines 10 - 22
}
```

Next, go to Code -> Generate - or Command+N on a Mac - and select "Implement Methods". This interface only requires one method.

```
24 lines | src/ApiPlatform/AutoGroupResourceMetadataFactory.php
... lines 1 - 5
use ApiPlatform\Core\Metadata\Resource\ResourceMetadata;
... line 7
class AutoGroupResourceMetadataFactory implements ResourceMetadataFactoryInterface
{
... lines 10 - 16
    public function create(string $resourceClass): ResourceMetadata
    {
... lines 19 - 21
    }
}
```

So... what the heck does this class do? It's job is pretty simple: given an API Resource class - like App\Entity\User - its job is to read all the API Platform *metadata* for that class - usually via annotations - and return it as a ResourceMetadata object. Yep, this ResourceMetadata object contains *all* of the configuration from our ApiResource annotation... which API Platform *then* uses to power... pretty much *everything*.

Service Decoration

Just like with the context builder, API Platform only has *one* core resource metadata factory. This means that instead of, sort of, *adding* this as some *additional* resource metadata factory, we need to completely *replace* the core resource metadata factory with our own. Yep, it's service decoration to the rescue!

The first step to decoration has... *nothing* to do with Symfony: it's the implementation of the decorator pattern. That sounds fancy. Create a public function __construct() where the first argument will be the "decorated", "core" object. This means that it will have the same interface as this class: ResourceMetadataFactoryInterface $decorated. Hit Alt + Enter and go to "Initialize Fields" to create that property and set it.

24 lines | src/ApiPlatform/AutoGroupResourceMetadataFactory.php

```
... lines 1 - 7
class AutoGroupResourceMetadataFactory implements ResourceMetadataFactoryInterface
{
    private $decorated;
... line 11
    public function __construct(ResourceMetadataFactoryInterface $decorated)
    {
        $this->decorated = $decorated;
    }
... lines 16 - 22
}
```

Inside the method, call the decorated class so it can do all the heavy-lifting: $resourceMetadata = $this->decorated->create($resourceClass). Then, return this at the bottom: we won't make any modifications yet.

24 lines | src/ApiPlatform/AutoGroupResourceMetadataFactory.php

```
... lines 1 - 16
    public function create(string $resourceClass): ResourceMetadata
    {
        $resourceMetadata = $this->decorated->create($resourceClass);

        return $resourceMetadata;
    }
... lines 23 - 24
```

The *second* step to decoration is *all* about Symfony: we need to tell it to use *our* class as the core "resource metadata factory" instead of the normal one... but to pass us the normal one as our first argument. Open up config/services.yaml. We've done all this before with the context builder: override the App\ApiPlatform\AutoGroupResourceMetadataFactory service... then I'll copy the first two options from above... and paste here. We actually don't need this autoconfigure option - that's a mistake in the documentation. It doesn't hurt... but we don't need it.

Ok, for decoration to work, we need to know what the core service id is that we're replacing. To find this, you'll need to read the docs... or maybe even dig a bit deeper if it's not documented. What we're doing is *so* advanced that you *won't* find it on the docs. The service we're decorating is api_platform.metadata.resource.metadata_factory. And for the "inner" thing, copy our service id and paste below to make: @App\ApiPlatform\AutoGroupResourceMetadataFactory.inner.

37 lines | config/services.yaml

```
... lines 1 - 8
services:
... lines 10 - 33
    App\ApiPlatform\AutoGroupResourceMetadataFactory:
        decorates: 'api_platform.metadata.resource.metadata_factory'
        arguments: ['@App\ApiPlatform\AutoGroupResourceMetadataFactory.inner']
```

Cool! Since our resource metadata factory isn't *doing* anything yet... everything should still work *exactly* like before. Let's see if that's true! Find your terminal and run the tests:

```
$ php bin/phpunit
```

And... huh... nothing broke! I, uh... didn't mean to sound so surprised.

Pasting in the Groups Logic

For the guts of this class, I'm going to paste two private functions on the bottom. These are low-level, boring functions that will do the hard work for us: updateContextOnOperations() and getDefaultGroups(), which is nearly identical to the method we copied into our context builder. You can copy both of these from the code block on this page.

77 lines | src/ApiPlatform/AutoGroupResourceMetadataFactory.php

```php
// ... lines 1 - 7
class AutoGroupResourceMetadataFactory implements ResourceMetadataFactoryInterface
{
// ... lines 10 - 33
    private function updateContextOnOperations(array $operations, string $shortName, bool $isItem)
    {
        foreach ($operations as $operationName => $operationOptions) {
            $operationOptions['normalization_context'] = $operationOptions['normalization_context'] ?? [];
            $operationOptions['normalization_context']['groups'] = $operationOptions['normalization_context']['groups'] ?? [];
            $operationOptions['normalization_context']['groups'] = array_unique(array_merge(
                $operationOptions['normalization_context']['groups'],
                $this->getDefaultGroups($shortName, true, $isItem, $operationName)
            ));

            $operationOptions['denormalization_context'] = $operationOptions['denormalization_context'] ?? [];
            $operationOptions['denormalization_context']['groups'] = $operationOptions['denormalization_context']['groups'] ?? [];
            $operationOptions['denormalization_context']['groups'] = array_unique(array_merge(
                $operationOptions['denormalization_context']['groups'],
                $this->getDefaultGroups($shortName, false, $isItem, $operationName)
            ));


            $operations[$operationName] = $operationOptions;
        }

        return $operations;
    }

    private function getDefaultGroups(string $shortName, bool $normalization, bool $isItem, string $operationName)
    {
        $shortName = strtolower($shortName);
        $readOrWrite = $normalization ? 'read' : 'write';
        $itemOrCollection = $isItem ? 'item' : 'collection';

        return [
            // {shortName}:{read/write}
            // e.g. user:read
            sprintf('%s:%s', $shortName, $readOrWrite),
            // {shortName}:{item/collection}:{read/write}
            // e.g. user:collection:read
            sprintf('%s:%s:%s', $shortName, $itemOrCollection, $readOrWrite),
            // {shortName}:{item/collection}:{operationName}
            // e.g. user:collection:get
            sprintf('%s:%s:%s', $shortName, $itemOrCollection, $operationName),
        ];
    }
}
```

Next, up in create(), I'll paste in a bit more code.

```
77 lines | src/ApiPlatform/AutoGroupResourceMetadataFactory.php
... lines 1 - 16
    public function create(string $resourceClass): ResourceMetadata
    {
... lines 19 - 20
        $itemOperations = $resourceMetadata->getItemOperations();
        $resourceMetadata = $resourceMetadata->withItemOperations(
            $this->updateContextOnOperations($itemOperations, $resourceMetadata->getShortName(), true)
        );

        $collectionOperations = $resourceMetadata->getCollectionOperations();
        $resourceMetadata = $resourceMetadata->withCollectionOperations(
            $this->updateContextOnOperations($collectionOperations, $resourceMetadata->getShortName(), false)
        );
... lines 30 - 31
    }
... lines 33 - 77
```

This is *way* more code than I normally like to paste in magically... but adding all the groups requires some pretty ugly & boring code. We start by getting the ResourceMetadata object from the core, decorated resource metadata factory. That ResourceMetadata object has a method on it called getItemOperations(), which returns an array of configuration that matches the itemOperations for whatever resource we're working on. Next, I call the updateContextOnOperations() method down here, which contains *all* the big, hairy code to loop over the different operations and make sure the normalization_context has our "automatic groups"... and that the denormalization_context *also* has the automatic groups.

The end result is that, by the bottom of this function, the ResourceMetadata object contains *all* the "automatic" groups we want for *all* the operations. Honestly, this whole idea is... kind of an experiment... and there might even be some subtle bug in my logic. But... it *should* work.

And *thanks* to this new stuff, the code in AdminGroupsContextBuilder is redundant: remove the private function on the bottom... and the line on top that called it.

```
37 lines | src/Serializer/AdminGroupsContextBuilder.php
... lines 1 - 8
final class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
... lines 11 - 19
    public function createFromRequest(Request $request, bool $normalization, ?array $extractedAttributes = null): array
    {
        $context = $this->decorated->createFromRequest($request, $normalization, $extractedAttributes);

        $context['groups'] = $context['groups'] ?? [];

        $isAdmin = $this->authorizationChecker->isGranted('ROLE_ADMIN');

        if ($isAdmin) {
            $context['groups'][] = $normalization ? 'admin:read' : 'admin:write';
        }

        $context['groups'] = array_unique($context['groups']);

        return $context;
    }
}
```

Ok... let's see what happens! Refresh the docs. The *first* thing you'll notice is on the bottom: there are now *tons* of models! This is the downside of this approach: it's *total* overkill for the models: Swagger shows *every* possible combination of the groups... even if none of our operations uses them.

Let's look at a specific operation - like GETing the collection of cheeses. Oh... actually - that's *not* a good example - the CheeseListing resource is temporarily broken - I'll show you why in a few minutes. Let's check out a User operation instead. Yep! It shows us *exactly* what we're going to get back.

So... we did it! We added dynamic groups that our API documentation knows about. Except... there are a few problems. It's possible that when you refreshed your docs, this did *not* work for you... due to *caching*. Let's talk more about that next and fix the CheeseListing resource.

Chapter 28: Dynamic Groups without Caching

Our resource metadata factory is now automatically adding a specific set of normalization and denormalization groups to each operation of each resource. That means that we can customize which fields are readable and writable for each operation just by adding specific groups to every property. And the *true* bonus is that... our documentation is aware of these dynamic groups! It *correctly* tells us which fields are readable and writable.

But... if you're coding along... it's possible that your docs did *not* update. If that happened, the fix is to run:

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

Here's the deal: the results of AutoGroupResourceMetadataFactory are cached... which makes sense: the ApiResource options should only need to be loaded one time... then cached. Unfortunately, for right now, this means that each time you make *any* change to this class, you need to manually rebuild your cache.

Changing CheeseListing shortName & Groups

But before we worry about that... all of our CheeseListing operations are... wait for it... broken! Yay! Check out the GET operation for the collection of cheese listings. It says that it will return an array of... nothing! And in fact... if you tried it, it *would* indeed return an array where each CheeseListing contains *no* fields!

This is a small detail related to how our resource metadata factory names the groups: it uses the API resource "short name" for each group - like user:read. What *is* this shortName thing? For CheeseListing, it comes from the shortName option inside the annotation. Or, if you don't have this option - API Platform guesses a shortName based on the class name.

The shortName most importantly becomes part of the URL. Check this out: execute a GET request to /api/cheeses. Then, use the web debug toolbar to open the profiler for that request... and go to the API Platform section. This shows you the "Resource Metadata" for CheeseListing. Hey, "Resource Metadata" - that's the name of the class that our resource metadata factory is creating!

Look at the normalization_context of the item GET operation: it still has cheese_listing:read and cheese_listing:item:get... because we still have those groups manually on the annotation... which we really should remove now. Then our resource metadata factory added 3 new groups: cheeses:read, cheeses:item:read and cheeses:item:get.

Basically, the group names in the new system - like cheeses:read - don't *quite* match the group names that we've been using so far - like cheese_listing:read. No worries, we just need to update our code to use the new group names.

But first, we added the shortName option in part 1 of this tutorial to change the URLs from /api/cheese_listings to /api/cheeses. Now, change the shortName to just cheese.

207 lines | src/Entity/CheeseListing.php

```
... lines 1 - 16
/**
 * @ApiResource(
... lines 19 - 30
 *     shortName="cheese",
... lines 32 - 35
 * )
... lines 37 - 46
 */
class CheeseListing
... lines 49 - 207
```

If you refresh the docs... surprise! All of the URLs are *still* /api/cheeses: API Platform automatically takes the shortName and makes it *plural* when creating the URLs. So, this change... didn't really.. change anything! I did it just so we could keep all of our group names singular. I'll do a find and replace to change cheese_listing: to cheese:.

207 lines | src/Entity/CheeseListing.php

```
... lines 1 - 47
class CheeseListing
{
... lines 50 - 56
    /**
... line 58
     * @Groups({"cheese:read", "cheese:write", "user:read", "user:write"})
... lines 60 - 65
     */
    private $title;
... line 68
    /**
... line 70
     * @Groups({"cheese:read"})
... line 72
     */
    private $description;
... line 75
    /**
... lines 77 - 79
     * @Groups({"cheese:read", "cheese:write", "user:read", "user:write"})
... line 81
     */
    private $price;
... lines 84 - 94
    /**
... lines 96 - 97
     * @Groups({"cheese:read", "cheese:write"})
... line 99
     */
    private $owner;
... lines 102 - 123
    /**
     * @Groups("cheese:read")
     */
    public function getShortDescription(): ?string
... lines 128 - 142
    /**
... lines 144 - 145
     * @Groups({"cheese:write", "user:write"})
... line 147
     */
    public function setTextDescription(string $description): self
... lines 150 - 172
    /**
... lines 174 - 175
     * @Groups("cheese:read")
     */
    public function getCreatedAtAgo(): string
... lines 179 - 205
}
```

Then, in User, there's *one* other spot we need to change. Above username, change this to cheese:item:get. Don't forget to *also* change cheese_listing:write to cheese:write. I'll catch that mistake a bit later.

```
239 lines | src/Entity/User.php
... lines 1 - 37
class User implements UserInterface
{
... lines 40 - 66
    /**
... line 68
     * @Groups({"user:read", "user:write", "cheese:item:get", "cheese:write"})
... line 70
     */
    private $username;
... lines 73 - 237
}
```

Phew! Ok, go refresh the docs... and open the GET operation for /api/cheeses. Yay! It properly advertises that it will return an array of the correct fields. And when we try it... hey! It even works!

Making our Resource Metadata Factory Not Cached

This whole resource metadata factory thing is *super* low-level in API Platform... but it *is* a nice way to add dynamic groups and have your docs reflect those changes. The only problem is the one I mentioned a few minutes ago: the results are cached... even in the dev environment. So if you tweak any logic inside this class, you'll need to manually clear your cache after *every* change. It's not the end of the world... but it *is* annoying.

And... ya know what? It might be *more* than simply "annoying". What if you wanted to add a dynamic normalization group based on who is logged in *and* you wanted the documentation to automatically update when the user is logged in to reflect that dynamic group? We already know that a context builder can add a dynamic group... but the docs won't update. But... because our resource metadata factory is cached... we can't put the logic there either: it would load the metadata just once, then use the same, cached metadata for everyone. It would *not* change after the user logged in.

But what if our resource metadata factory... *wasn't* cached? Duh, duh, duh!

Service Decoration Priority

Check this out: in config/services.yaml, add a new option to the service: decoration_priority set to -20.

```
41 lines | config/services.yaml
... lines 1 - 8
services:
... lines 10 - 33
    App\ApiPlatform\AutoGroupResourceMetadataFactory:
... line 35
        # causes this to decorate around the cached factory so that
        # our service is never cached (which, of course, can have performance
        # implications!
        decoration_priority: -20
... lines 40 - 41
```

Wow... yea... we just took an already-advanced concept and... went even deeper. When we decorate a core service, we might not be the *only* service decorating it: there may be *multiple* levels of decoration. And... that's fine! Symfony handles all of that behind the scenes.

In the case of the resource metadata factory, API Platform *itself* decorates that service multiple times... each "layer" adding a bit more functionality. Normally, when *we* decorate a service from our application, our object becomes the *outermost* object in the chain. One of the *other* services that decorates the core service and is part of that chain is called CachedResourceMetadataFactory. You can probably guess what it does: it calls the core resource metadata factory, gets the result, then caches it.

So... why is this a problem? If we are the *outermost* resource metadata factory, then... even if the CachedResourceMetadataFactory caches the core metadata, our function would still *always* be called... and *our* changes should *never* be cached. But... that is *not* what's happening!

Why? Because that CachedResourceMetadataFactory has a decoration_priority of -10... and the default is 0. Before we added the decoration_priority option, this meant that Symfony made CachedResourceMetadataFactory the *first* object in the decoration chain and *our* class the second. And *that* caused our class's results to be cached. By setting our decoration_priority to -20, our object is moved *before* CachedResourceMetadataFactory... and suddenly, our results are no longer cached.

Crazy, right? We can *now* put whatever dynamic logic we want into our custom resource metadata factory. Refresh the docs... and look down on the models. Yep, no surprises. *Now* go into our class and add FOOO to the end of one of the groups. If we had made this tweak a minute ago and refreshed without clearing the cache, we would have seen *no* changes. But now... it's there instantly! All the *core* logic that reads the annotations *is* still cached, but our class is *not*.

Just... be careful with this: the reason the logic is normally cached is that API Platform calls this function *many* times during a request. So, any logic you add here needs to be lightning quick. You may even decide to add a private $resourceMetadata array property where you store the ResourceMetadata object for each class as you calculate it. Then, if create() is called on the same request for the same $resourceClass, you can return it from this array instead of running our logic over and over again.

Ok team, I hope you enjoyed this crazy dive into custom resource metadata factories. Next, we know how to hide or show a field based on who is logged in - like returning the phoneNumber field *only* if the user has ROLE_ADMIN. But what if we *also* need to hide or show a field based on which *object* is being serialized? What if we *also* want to return the phoneNumber field when an authenticated user is fetching their *own* User data?

Chapter 29: Custom Normalizer: Object-by-Object Dynamic Fields

We now know how to add dynamic groups: we added admin:read above phoneNumber and then, via our context builder, we're dynamically adding that group to the serialization context *if* the authenticated user has ROLE_ADMIN.

So... we're pretty cool! We can easily run around and expose input our output fields only to admin users by using these two groups.

But... the context builder - and also the more advanced resource metadata factory - has a tragic flaw! We can only change the context *globally*. What I mean is, we're deciding which groups should be used for normalizing or denormalizing a specific *class*... no matter how many different objects we might be working with. It does *not* allow us to change the groups on an object-by-object basis.

Let me give you a concrete example: in addition to making the $phoneNumber readable by admin users, I *now* want a user to *also* be able to read their *own* phoneNumber: if I make a request and the response will contain data for my *own* User object, it *should* include the phoneNumber field.

You might think:

> Ok, let's put phoneNumber in some new group, like owner:read... and add that group dynamically in the context builder.

That's great thinking! But... look in the context builder, look at what's passed to the createFromRequest() method... or really, what's *not* passed: it does *not* pass us the specific *object* that's being serialized. Nope, this method is called just *once* per request.

Creating a Normalizer

Ok, no worries. Context builders are a *great* way to add or remove groups on a global or class-by-class basis. But they are *not* the way to dynamically add or remove groups on an object-by-object basis. Nope, for that we need a custom normalizer. Let's convince MakerBundle to create one for us. Run:

```
$ php bin/console make:serializer:normalizer
```

Call this UserNormalizer. When an object is being transformed into JSON, XML or any format, it goes through two steps. First, a "normalizer" transforms the object into an array. And second, an "encoder" transforms that array into whatever format you want - like JSON or XML.

When a User object is serialized, it's already going through a core normalizer that looks at our normalization groups & reads the data via the getter methods. We're now going to hook *into* that process so that we can change the normalization groups *before* that core normalizer does its job.

Go check out the new class: src/Serializer/Normalizer/UserNormalizer.php.

```
37 lines | src/Serializer/Normalizer/UserNormalizer.php
... lines 1 - 8
class UserNormalizer implements NormalizerInterface, CacheableSupportsMethodInterface
{
    private $normalizer;

    public function __construct(ObjectNormalizer $normalizer)
    {
        $this->normalizer = $normalizer;
    }

    public function normalize($object, $format = null, array $context = array()): array
    {
        $data = $this->normalizer->normalize($object, $format, $context);

        // Here: add, edit, or delete some data

        return $data;
    }

    public function supportsNormalization($data, $format = null): bool
    {
        return $data instanceof \App\Entity\BlogPost;
    }

    public function hasCacheableSupportsMethod(): bool
    {
        return true;
    }
}
```

This works a bit differently than the context builder - it works more like the voter system. The serializer doesn't have just *one* normalizer, it has *many* normalizers. Each time it needs to normalize something, it loops over *all* the normalizers, calls supportsNormalization() and passes us the data that it needs to normalize. If we return true from supportsNormalization(), it means that *we* know how to normalize this data. And so, the serializer will call *our* normalize() method. Our normalizer is then the *only* normalizer that will be called for this data: we are 100% responsible for transforming the object into an array.

Normalizer Logic

Of course... we don't *really* want to *completely* take over the normalization process. What we *really* want to do is change the normalization groups... and then call the *core* normalizer so it can do its normal work. That's why the class was generated with a constructor where we're autowiring a class called ObjectNormalizer. This is the main, core, normalizer for objects: it's the one that's responsible for reading the data via our getter methods. So... cool! Our custom normalizer is basically... just offloading all the work to the *core* normalizer!

Let's start customizing this! For supportsNormalization(), return $data instanceof User. So if the thing that's being normalized is a User object, we handle that.

50 lines | src/Serializer/Normalizer/UserNormalizer.php

```
... lines 1 - 9
class UserNormalizer implements NormalizerInterface, CacheableSupportsMethodInterface
{
... lines 12 - 34
    public function supportsNormalization($data, $format = null): bool
    {
        return $data instanceof User;
    }
... lines 39 - 48
}
```

Now we know that normalize() will *only* be called if $object is a User. Let's add some PHPDoc above this to help my editor.

50 lines | src/Serializer/Normalizer/UserNormalizer.php

```
... lines 1 - 18
    /**
     * @param User $object
     */
    public function normalize($object, $format = null, array $context = array()): array
... lines 23 - 50
```

The goal here is to check to see if the User object that's being normalized is the *same* as the currently-authenticated User. If it is, we'll add that owner:read group. Add that check on top: if $this->userIsOwner($object) - that's a method we'll create in a minute - then, add the group. The $context is passed as the third argument... and we're passing *it* to the core normalizer below. Let's modify it first! Use $context['groups'][] = 'owner:read.

That's lovely! A normalizer is only used for... um... *normalizing* an object to an array - it is *not* used for *denormalizing* an array back into an object. That's why we're always adding owner:read here. If you wanted to create this *same* feature for denormalization... and add an owner:write group.. you'll need to create a separate *denormalizer* class. There's no MakerBundle command to generate it, but the logic will be almost identical to this... and you can even make your *one* normalizer class implement both NormalizerInterface *and* DenormalizerInterface.

Oh, also, we don't need to check for the existence of a groups key on the array because, in our system, we are *always* setting at least one group.

Let's add that missing method: private function userIsOwner(). This will take a User object and return a bool. For now, fake it: return rand(0, 10) > 5.

46 lines | src/Serializer/Normalizer/UserNormalizer.php

```
... lines 1 - 35
    private function userIsOwner(User $user): bool
    {
        return mt_rand(0, 10) > 5;
    }
... lines 40 - 46
```

And... I think that's it! Like with voters, this is a situation where we *don't* need to add any configuration: as soon as we create a class and make it implement NormalizerInterface, the serializer will see it and start using it.

So... let's take this for a test drive! Back on the docs, I'm currently *not* logged in. Let's refresh the page... and create a new user. How about email goudadude@example.com, password foo, same username, no cheeseListings, but *with* a phoneNumber. Execute and... perfect! A 201 status code. Copy that email... go back to the homepage.. and log in: goudadude@example.com, password foo and... go!

Cool! Now that we're authenticated, head back to /api. Yep, the web debug toolbar *confirms* that I'm a "gouda dude". Let's try the GET operation to fetch a collection of users. Because of our random logic, I'd expect *some* results to show the phoneNumber and some not. Execute and... hey! The first user has a phoneNumber field! It's null... because apparently we didn't set a phoneNumber for that user, but the field *is* there. And, thanks to the randomness, there is *no* phoneNumber for the second and third users. If you try the operation again... yes! This time the first *and* second users have that field, but not the third. Hey! We're now dynamically adding the owner:read group on an object-by-object basis! Normalizers rock!

But... wait a second. Something is wrong! We're missing the JSON-LD fields for these users. Well, ok, we have them on the top-level for the collection itself... and even the embedded CheeseListing data has them... but each user is missing @id and @type. Something in our new normalizer killed the JSON-LD stuff!

Next, let's figure out what's going on, find this bug, then *crush* it!

Chapter 30: Diving into the Normalizer Internals

When a User object is normalized - whether it's a single User object or a collection of User objects - our new UserNormalizer class is now 100% responsible for that process. Whenever the serializer needs to normalizer a User object, it loops over all of the normalizers and stops when it finds the *first* one whose supportsNormalization() method returns true. That's now *our* normalizer.

And... what we're doing is simple: adding a custom group... then, because we autowired the ObjectNormalizer - that's the "main" normalizer Symfony uses to normalize objects - we're calling it and making *it* do all of the heavy lifting. But we discovered in the last chapter that this isn't *quite* working like we want: each User record is missing the JSON-LD fields. The embedded cheeseListings data has them... but each user does not.

What's going on?

The Many Core Normalizers

When you work with API Platform, the Symfony serializer has *many* normalizers. I'll hit Shift+Shift and search for a class called ItemNormalizer. There are a bunch of these - open the one in the JsonLd directory. This is one of those normalizers. And, if you followed its parent class, you would find that it eventually extends ObjectNormalizer. Basically, this class uses the *normal* functionality of the parent ObjectNormalizer - that's the one that reads data off of our getter methods - but then *adds* the JSON-LD fields - like @context and @id.

Actually, let's back up even further... and pretend like our custom UserNormalizer doesn't exist. Out-of-the-box, when API Platform normalizes a single User object, *many* normalizers are used. First, it loops over all of the normalizers and finds the *one* normalizer that can turn a User object into an array of data. That's normally handled by the JsonLd ItemNormalizer, which reads all the data from the User object *and* adds a few more JSON-LD specific fields. Once this finishes, the serializer *then* loops over the individual pieces of data - like the phoneNumber string or the cheeseListings collection - and sends each of *these* through the normalization process again, asking each normalizer if they support this piece of data until it finds one that does. Or... if none do, it just uses the value as the final value - that happens for simple scalar fields like phoneNumber.

This creates a *super* powerful system. For example, another normalizer - the DateTimeNormalizer - is responsible for normalizing any DateTime objects... like if you had a createdAt property. It normalizes DateTime objects into a string that can used in JSON or XML.

So each *piece* of data - from the top-level User down to each property... and even further for related objects - is normalized by exactly *one* normalizer.

Why is the JSON-LD Info Gone?

Cool! So then... back to our question: why are the JSON-LD fields missing? Well, when we autowired the ObjectNormalizer and then called it... we're not *really* calling the correct, core normalizer. Nope, instead of asking the serializer to loop over *all* of the normalizers to find the correct one to use, we accidentally autowired just one *specific* normalizer and are using it. Basically, instead of using the ItemNormalizer that does all the ObjectNormalizer goodness and then adds the JSON-LD fields, we're using the ObjectNormalizer directly. Hence... we lost those fields!

This is a *long* way of saying that what we *really* want to do is modify the $context and then send the User object back through the entire normalization chain again so it can find the core normalizer that's *usually* responsible for normalizing objects.

How do we do that? It's easy! Um, and... kinda tricky. It involves recursion... well... hopefully *avoiding* recursion. Now that we understand what's going on, let's fix it next.

Chapter 31: A "Normalizer Aware" Normalizer

When a User object is being normalized, our UserNormalizer is called. After adding a dynamic group to $context, we want to send the User object *back* through the full normalizer system: we want the *original*, "core" normalizer - whatever class that might be - to do its magic. Right now, we're not *quite* accomplishing that because we're directly calling a *specific* normalizer - ObjectNormalizer.

So... how can we call the "normalizer chain" instead of this specific normalizer? By implementing a new interface: NormalizerAwareInterface.

46 lines | src/Serializer/Normalizer/UserNormalizer.php

```
// ... lines 1 - 6
use Symfony\Component\Serializer\Normalizer\NormalizerAwareInterface;
// ... lines 8 - 10
class UserNormalizer implements NormalizerInterface, CacheableSupportsMethodInterface, NormalizerAwareInterface
// ... lines 12 - 46
```

This requires us to have a single new method setNormalizer(). When the serializer system see that a normalizer has this interface, *before* it uses that normalizer, it will call setNormalizer() and pass a normalizer object... that *really* holds the "chain" of *all* normalizers. It, sort of, passes us the top-level normalizer - the one that is responsible for looping over all of the normalizers to find the correct *one* to use for this data.

So, we *won't* autowire ObjectNormalizer anymore: remove the constructor and that property. Instead, use NormalizerAwareTrait. That trait is just a shortcut: it has a setNormalizer() method and it stores the normalizer on a protected property.

46 lines | src/Serializer/Normalizer/UserNormalizer.php

```
// ... lines 1 - 7
use Symfony\Component\Serializer\Normalizer\NormalizerAwareTrait;
// ... lines 9 - 10
class UserNormalizer implements NormalizerInterface, CacheableSupportsMethodInterface, NormalizerAwareInterface
{
    use NormalizerAwareTrait;
// ... lines 14 - 44
}
```

The end result is that $this->normalizer is *now* the main, normalizer chain. We add the group, then pass User back through the original process.

Tip

Please, make sure you have Xdebug PHP extension installed and enabled, otherwise the page may just hang if you try to execute this code and you will need to restart your built-in web server or PHP-FPM.

And... some of you *may* already see a problem with this. When we hit Execute to try this... it runs for a few seconds, then... error!

> Maximum function nesting level of 256 reached, aborting!

Yay! Recursion! We're changing the $context and then calling normalize() on the original normalizer chain. Well... guess what that does? It once again calls supportsNormalization() on this object, we return true, and it calls normalize() on us again. We're basically calling *ourselves* over and over and over again.

Wah, wah. Hmm. We only want to be called *once* per object being normalized. What we need is some sort of a flag... something that says that we've *already* been called so we can avoid calling ourselves again.

Avoiding Recursion with a $context Flag

The way to do this is by adding a flag to the $context itself. Then, down in supportsNormalization(), if we see that flag, it means that we've already been called and we can return false.

But wait... the $context isn't passed to supportsNormalization()... so that's a problem. Well... not a big problem - we can get that by tweaking our interface. Remove NormalizerInterface and replace it with ContextAwareNormalizerInterface. We can do that because it extends NormalizerInterface. The only difference is that this interface requires our method to have one extra argument: array $context.

55 lines | src/Serializer/Normalizer/UserNormalizer.php

```
... lines 1 - 6
use Symfony\Component\Serializer\Normalizer\ContextAwareNormalizerInterface;
... lines 8 - 10
class UserNormalizer implements ContextAwareNormalizerInterface, CacheableSupportsMethodInterface, NormalizerAwareInterface
{
... lines 13 - 34
    public function supportsNormalization($data, $format = null, array $context = [])
    {
... lines 37 - 42
    }
... lines 44 - 53
}
```

For the flag, let's add a constant: private const ALREADY_CALLED set to, how about, USER_NORMALIZER_ALREADY_CALLED. Now, in normalize(), right *before* calling the original chain, set this: $context[self::ALREADY_CALLED] = true. Finally, in supportsNormalization(), if isset($context[self::ALREADY_CALLED]), return false. That will allow the "normal" normalizer to be used on the second call.

55 lines | src/Serializer/Normalizer/UserNormalizer.php

```
... lines 1 - 10
class UserNormalizer implements ContextAwareNormalizerInterface, CacheableSupportsMethodInterface, NormalizerAwareInterface
{
... lines 13 - 14
    private const ALREADY_CALLED = 'USER_NORMALIZER_ALREADY_CALLED';
... lines 16 - 19
    public function normalize($object, $format = null, array $context = array()): array
    {
... lines 22 - 25
        $context[self::ALREADY_CALLED] = true;
... lines 27 - 32
    }
... line 34
    public function supportsNormalization($data, $format = null, array $context = [])
    {
        // avoid recursion: only call once per object
        if (isset($context[self::ALREADY_CALLED])) {
            return false;
        }

        return $data instanceof User;
    }
... lines 44 - 53
}
```

We've done it! We've fixed the recursion! Let's celebrate by hitting Execute and... more recursion!

Removing hasCacheableSupportsMethod()

Because... we're missing *one* subtle detail. Find the hasCacheableSupportsMethod() method - this was generated for us - and return false:

```
55 lines | src/Serializer/Normalizer/UserNormalizer.php
... lines 1 - 49
    public function hasCacheableSupportsMethod(): bool
    {
        return false;
    }
... lines 54 - 55
```

Go back up, hit Execute and... it works! The phoneNumber field is still randomly included... because we have some random logic in our normalizer... but the @id and @type JSON-LD stuff is back!

The hasCacheableSupportsMethod() is an optional method that each normalizer can have... which relates to this optional interface - CacheableSupportsMethodInterface. The purpose of this interface - and the method - is performance.

Because *every* piece of the object graph is normalized, the serializer calls supportsNormalization()... a *lot* of times. If your supportsNormalization() method *only* relies on the $format and the *class* of $data - basically $data instanceof User, then you can return true from hasCacheableSupportsMethod(). When you do this, the serializer will only call supportsNormalization() once per class. That speeds things up.

But as *soon* as you rely on the $data itself or the $context, you need to return false... or remove this interface entirely, both have the same result. This forces the serializer to call our supportNormalization() method each time the chain is called. That *does* have a performance impact, but as long as your logic is fast, it should be minor. And *most* importantly, it fixes our issue!

Next, let's add the proper security logic to our class and then investigate *another* superpower of normalizers: the ability to add *completely* custom fields. We'll add a strange... but potentially useful boolean field to User called isMe.

Chapter 32: Normalizer & Completely Custom Fields

Our UserNormalizer is now totally set up. These classes are beautiful & flexible: we can add custom normalization groups on an object-by-object basis. They're also weird: you need to know about this NormalizerAwareInterface thing... and you need to understand the idea of setting a flag into the $context to avoid calling yourself recursively. But once you've got that set up, you're *gold*!

Options for Adding Custom Fields

And if you look more closely... we're even *more* dangerous than you might realize. The job of a normalizer is to turn an object - our User object - into an array of data and return it. You can tweak which data is included by adding more groups to the $context... but you could also add custom fields... right here!

Well, hold on a minute. Whenever possible, if you need to add a custom field, you should do it the "correct" way. In CheeseListing, when we wanted to add a custom field called shortDescription, we did that by adding a getShortDescription() method and putting it in the cheese:read group. Boom! Custom field!

Why is this the *correct* way of doing it? Because this causes the field to be seen & documented correctly.

But, there are two downsides - or maybe limitations - to this "correct" way of doing things. First, if you have *many* custom fields... it starts to get ugly: you might have a *bunch* of custom getter and setter methods *just* to support your API. And second, if you need a *service* to generate the data for the custom field, then you *can't* use this approach. Right now, I want to add a custom isMe field to User. We couldn't, for example, add a new isMe() method to User that returns true or false based on whether this User matches the currently-authenticated user... because we need a service to know who is logged in!

So... since we can't add an isMe field the "correct" way... how *can* we add it? There are two answers. First, the... sort of... "second" correct way is to use a DTO class. That's something we'll talk about in a future tutorial. It takes more work, but it *would* result in your custom fields being documented properly. Or second, you can hack the field into your response via a normalizer. That's what we'll do now.

Adding Proper Security

Oh, but before we get there, I almost forgot that we need to make this userIsOwner() method... actually work! Add a constructor to the top of this class and autowire the Security service. I'll hit Alt -> Enter and go to "Initialize Fields" to create that property and set it. Down in the method, say $authenticatedUser = $this->security->getUser() with some PHPDoc above this to tell my editor that this will be a User object or null if the user is *not* logged in. Then, if !$authenticatedUser, return false. Otherwise, return $authenticatedUser->getEmail() === $user->getEmail(). We could also compare the objects themselves.

```
70 lines | src/Serializer/Normalizer/UserNormalizer.php
... lines 1 - 11
class UserNormalizer implements ContextAwareNormalizerInterface, CacheableSupportsMethodInterface, NormalizerAwareInterface
{
... lines 14 - 17
    private $security;
... line 19
    public function __construct(Security $security)
    {
        $this->security = $security;
    }
... lines 24 - 52
    private function userIsOwner(User $user): bool
    {
        /** @var User|null $authenticatedUser */
        $authenticatedUser = $this->security->getUser();

        if (!$authenticatedUser) {
            return false;
        }

        return $authenticatedUser->getEmail() === $user->getEmail();
    }
... lines 64 - 68
}
```

Let's try this: if we fetch the collection of all users, the phoneNumber field should *only* be included in *our* user record. And... no phoneNumber, no phoneNumber and... yes! The phoneNumber shows up *only* on the third record: the user that we're logged in as.

Fixing the Tests

Oh, but this *does* break one of our tests. Run all of them:

```
$ php bin/phpunit
```

Most of these will pass, but... we *do* get one failure:

> Failed asserting that an array does not have the key phoneNumber on UserResourceTest.php line 66.

Let's open that test and see what's going on. Ah yes: this is the test where we check to make sure that if you set a phoneNumber on a User and make a GET request for that User, you do *not* get the phoneNumber field back *unless* you're logged in as an admin.

But we've now changed that: in addition to admin users, an authenticated user will *also* see their own phoneNumber. Because we're logging in as cheeseplease@example.com... and then fetching that same user's data, it *is* returning the phoneNumber field. That's the correct behavior.

To fix the test, change createUserAndLogin() to just createUser()... and remove the first argument. Now use $this->createUserAndLogin() to log in as a totally *different* user. Now we're making a GET request for the cheeseplease@example.com user data but we're *authenticated* as this *other* user. So, we should *not* see the phoneNumber field.

```
80 lines | tests/Functional/UserResourceTest.php
... lines 1 - 8
class UserResourceTest extends CustomApiTestCase
{
... lines 11 - 50
    public function testGetUser()
    {
... line 53
        $user = $this->createUser('cheeseplease@example.com', 'foo');
        $this->createUserAndLogIn($client, 'authenticated@example.com', 'foo');
... lines 56 - 78
    }
}
```

Run the tests again:

```
$ php bin/phpunit
```

And... all green.

Adding the Custom isMe Field

Ok, back to our original mission... which will be *delightfully* simple: adding a custom isMe field to User. Because $data is an array, we can add whatever fields we want. Up here, I'll create a variable called $isOwner set to what we have in the if statement: $this->userIsOwner($object). Now we can use $isOwner in the if *and* add the custom field: $data['isMe'] = $isOwner.

```
71 lines | src/Serializer/Normalizer/UserNormalizer.php
... lines 1 - 11
class UserNormalizer implements ContextAwareNormalizerInterface, CacheableSupportsMethodInterface, NormalizerAwareInterface
{
... lines 14 - 27
    public function normalize($object, $format = null, array $context = array()): array
    {
        $isOwner = $this->userIsOwner($object);
        if ($isOwner) {
            $context['groups'][] = 'owner:read';
        }
... lines 34 - 38
        $data['isMe'] = $isOwner;
... lines 40 - 41
    }
... lines 43 - 69
}
```

Et voilà! Test it! Execute the operation again and... there it is: isMe false, false and true! Just remember the downside to this approach: our documentation has *no* idea that this isMe field exists. If we refresh this page and open the docs for fetching a single User... yep! There's no mention of isMe. Of course, you *could* add a public function isMe() in User, put it in the user:read group, always return false, then *override* the isMe key in your normalizer with the *real* value. That would give you the custom field *and* the docs. But sheesh... that's... getting kinda hacky.

Next, let's look more at the owner field on CheeseListing. It's interesting: we're currently allowing the user to *set* this property when they POST to create a User. Does that make sense? Or should it be set automatically? And if we *do* want an API user to be able to send the owner field via the JSON, how do we prevent them from creating a CheeseListing and setting the owner to some *other* user? It's time to see where security & validation meet.

Chapter 33: Locking down the CheeseListing.owner Field

Inside CheeseListing, the owner property - that's the related User object that owns this CheeseListing - is required in the database. *And*, thanks to the cheese:write group, it's a *writable* field in our API.

In fact, even though I've forgotten to add an @Assert\NotBlank constraint to this property, an API client *must* send this field when creating a new CheeseListing. They can also *change* the owner by sending the field via a PUT request. We even added some fanciness where, when you create or edit a CheeseListing, you can modify the owner's username all at the same time. This works because that property is in the cheese:write group... oops. I forgot to change this group when we were refactoring - that's how it should look. The cheese:item:get group means this field will be embedded when we GET a *single* CheeseListing - that's the "item operation" - and cheese:write means it's writable when using *any* write operation for CheeseListing. That's some crazy stuff we set up on our previous tutorial.

Removing username Updatable Fanciness

But now, I want to simplify in two ways. First, I only want the owner property to be set when we *create* the CheeseListing, I don't want it to be *changeable*. And second, let's get rid of the fanciness of being able to edit the username property via a CheeseListing operation. For that second part, remove the cheese:write group from username. We can now also take off the @Assert\Valid() annotation. This caused the User to be validated during the CheeseListing operations... which was needed to make sure someone didn't set the username to an invalid value while updating a CheeseListing.

207 lines | src/Entity/CheeseListing.php

```
... lines 1 - 47
class CheeseListing
{
... lines 50 - 94
    /**
... lines 96 - 97
     * @Groups({"cheese:read", "cheese:write"})
... line 99
     */
    private $owner;
... lines 102 - 205
}
```

Making owner *only* Settable on POST

Now, how can we make the owner property settable for the POST operation but *not* the PUT operation?

Open up AutoGroupResourceMetadataFactory. This monster automatically adds three serialization groups in all situations. We can use this last one to include a field *only* for a specific operation. Change cheese:write to cheese:collection:post. That follows the pattern: "short name", colon, collection, colon, then the operation name: post.

206 lines | src/Entity/CheeseListing.php

```
... lines 1 - 94
    /**
... lines 96 - 97
     * @Groups({"cheese:read", "cheese:collection:post"})
     */
    private $owner;
... lines 101 - 206
```

Congratulations, the owner can no longer be changed.

Should Owner Be Set Automatically?

But... hold up. Isn't it kinda weird that we allow the API client to send the owner field at all? I mean... shouldn't we instead *not* make owner part of our API and then write some code to automatically set this to the currently-authenticated user?

Um, maybe. Automatically setting the owner property *is* kinda nice... and it would also make our API easier to use. We *will* talk about how to do this later. But I don't want to completely remove owner from my API. Why? Well, what if we created an admin interface where admin users could create cheese listings on *behalf* of other users. In that case, we *would* want the owner field to be part of our API.

But... hmm if we allow the owner field to be sent... we can't just allow API clients to create a CheeseListing and set the owner to whoever they want. Sure, maybe an admin user should be able to do this... but how can we prevent a normal user from setting the owner to someone else?

Two Ways to Protect a Writable Field

Backing up, if you need to control some behavior around the way a field is *set* based on the authenticated user, you have two options. First, you could prevent some users from writing to the field entirely. That's done by putting the property into a special serialization group then dynamically adding that group either in a context builder or in a custom normalizer. We've done that already with admin:read and admin:write.

Second, if the field *should* be writable by *all* users... but the data that's *allowed* to be set on the field depends on *who* is logged in, then the solution is validation.

Here's our goal: prevent an API client from POSTing an owner value that is *different* than their own IRI... with an exception added for admin users: they can set owner to anyone.

Let's codify this into a test first. Open CheeseListingResourceTest. Inside testCreateCheeseListing(), we're basically verifying that you *do* need to be logged in to use the operation. We get a 401 before we're authenticated... then after logging in, we get a 400 status code because access will be granted... but our empty data will fail validation.

Let's make this test more interesting! Create a new $authenticatedUser variable set to who we're logged in as. Then create an $otherUser variable set to... *another* user in the database.

75 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 13
    public function testCreateCheeseListing()
    {
... lines 16 - 21
        $authenticatedUser = $this->createUserAndLogIn($client, 'cheeseplease@example.com', 'foo');
        $otherUser = $this->createUser('otheruser@example.com', 'foo');
... lines 24 - 43
    }
... lines 45 - 73
}
```

Here's the plan: I want to make *another* POST request to /api/cheeses with *valid* data... except that we'll set the owner field to this $otherUser... a user that we are *not* logged in as. Start by creating a $cheesyData variable set to an array with title, description and price. These are the three required fields other than owner.

75 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 13
    public function testCreateCheeseListing()
    {
... lines 16 - 29
        $cheesyData = [
            'title' => 'Mystery cheese... kinda green',
            'description' => 'What mysteries does it hold?',
            'price' => 5000
        ];
... lines 35 - 43
    }
... lines 45 - 75
```

Now, copy the request and status code assertion from before, paste down here and set the json to $cheesyData *plus* the owner property set to /api/users/ and then $otherUser->getId().

75 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 13
    public function testCreateCheeseListing()
    {
... lines 16 - 34
        $client->request('POST', '/api/cheeses', [
            'json' => $cheesyData + ['owner' => '/api/users/'.$otherUser->getId()],
        ]);
... lines 38 - 43
    }
... lines 45 - 75
```

In this case, the status code should *still* be 400 once we've coded all of this: passing the wrong owner will be a *validation* error. I'll add a little message to the assertion to make it obvious why it's failing:

75 lines | tests/Functional/CheeseListingResourceTest.php

```
... lines 1 - 13
    public function testCreateCheeseListing()
    {
... lines 16 - 37
        $this->assertResponseStatusCodeSame(400, 'not passing the correct owner');
... lines 39 - 43
    }
... lines 45 - 75
```

    not passing the correct owner

I like it! We're logging in as cheeseplease@example.com... then we're trying to create a CheeseListing that's owned by a totally *different* user. This is the behavior we want to prevent.

While we're here, copy these two lines again and change $otherUser to $authenticatedUser. This *should* be allowed, so change the assertion to look for the happy 201 status code.

```
75 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 13
    public function testCreateCheeseListing()
    {
... lines 16 - 39
        $client->request('POST', '/api/cheeses', [
            'json' => $cheesyData + ['owner' => '/api/users/'.$authenticatedUser->getId()],
        ]);
        $this->assertResponseStatusCodeSame(201);
    }
... lines 45 - 75
```

You know the drill: once you've written a test, you get to celebrate by watching it fail! Copy the method name, flip over to your terminal and run:

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

And... it fails!

> Failed asserting response status code is 400 - got 201 Created.

So we *are* currently able to create cheese listings and set the owner as a different user. Cool! Next, let's prevent this with a custom validator.

Chapter 34: Custom Validator

Here's the situation: all authenticated users should have access to create a CheeseListing... and one of the fields that can be passed is owner. But the data passed to the owner field may be valid or invalid depending on who you're authenticated as. For a normal user, I'm supposed to set this to my own IRI: if I try to set it to a *different* IRI, that should be denied. But for an *admin* user, they *should* be allowed to set the IRI to anyone.

When the *value* of a field may be allowed or not allowed based on who is authenticated, that should be protected via *validation*... which is why we're expecting a 400 status code - not a 403.

Generating the Custom Validator

Ok, so how can we make sure the owner field is set to the currently-authenticated user? Via a *custom* validator. Find your terminal and kick things off with:

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

Call it IsValidOwner. If you're not familiar with validators, each validation constraint consists of *two* classes - you can see them both inside the src/Validator/ directory. The first class represents the *annotation* that we will use to activate this... and it's usually empty, except for a few properties that are typically public. Each public property will become an *option* that you can pass to the annotation. More on that in a minute.

18 lines | src/Validator/IsValidOwner.php

```
... lines 1 - 9
class IsValidOwner extends Constraint
{
    /*
     * Any public properties become valid options for the annotation.
     * Then, use these in your validator class.
     */
    public $message = 'The value "{{ value }}" is not valid.';
}
```

The other class, which typically has the same name plus the word "Validator", is what will be called to do the actual *work* of validation. The validation system will pass us the $value that we're validating and then we can do whatever business logic we need to determine if it's valid or not. If the value is invalid, you can use this cool buildViolation() thing to set an error.

```
24 lines | src/Validator/IsValidOwnerValidator.php
... lines 1 - 7
class IsValidOwnerValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        /* @var $constraint \App\Validator\IsValidOwner */

        if (null === $value || '' === $value) {
            return;
        }

        // TODO: implement the validation here
        $this->context->buildViolation($constraint->message)
            ->setParameter('{{ value }}', $value)
            ->addViolation();
    }
}
```

Using the Validation Constraint

To see this in practice, open up CheeseListing. The property that we need to validate is $owner: we want to make sure that it is set to a "valid" owner... based on whatever crazy logic we want. To activate the new validator, add @IsValidOwner(). This is where we could customize the message option... or *any* public properties we decide to put on the annotation class.

```
208 lines | src/Entity/CheeseListing.php
... lines 1 - 10
use App\Validator\IsValidOwner;
... lines 12 - 48
class CheeseListing
{
... lines 51 - 95
    /**
... lines 97 - 99
     * @IsValidOwner()
     */
    private $owner;
... lines 103 - 206
}
```

Actually, let's change the *default* value for $message:

> Cannot set owner to a different user

```
18 lines | src/Validator/IsValidOwner.php
... lines 1 - 9
class IsValidOwner extends Constraint
{
... lines 12 - 15
    public $message = 'Cannot set owner to a different user';
}
```

Ok, now that we've added this annotation, whenever the CheeseListing object is being validated, the validation system will *now* call validate() on IsValidOwnerValidator. The $value will be the value of the $owner property. So, a User object. It also passes us $constraint, which will be an instance of the IsValidOwner class where the public properties are populated with any options that we may have passed to the annotation.

Avoid Validating Empty Values

The *first* thing the validator does is interesting... it checks to see if the $value is, sort of, empty - if it's null. If it *is* null, instead of adding a validation error, it does the opposite! It returns.. which means that, as far as this validator is concerned, the value is valid. Why? The philosophy is that, if you want this field to be *required*, you should add an *additional* annotation to the property - the @Assert\NotBlank constraint. We'll do that a bit later. That means that *our* validator only has to do its job if there *is* a value set.

The setParameter() Wildcard

To see if this is working... ah, let's just try it! Sure, we haven't added any logic yet... and so this constraint will *always* have an error... but let's make sure we at least see that error!

Oh, and this setParameter() thing is a way for you to add "wildcards" to the message. Like, if you set {{ value }} to the email of the User object, you could reference that dynamically in your message with that same {{ value }}. We don't need that... so let's remove it. Oh, and just to be *totally* clear, the $constraint->message part is referencing the $message property on the annotation class. So, we *should* see our customized error.

Let's try it! Go tests go!

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

If we scroll up... awesome! It's failing: it's getting back a 400 bad request with:

> Cannot set owner to different user

Hey! That's our validation message! The failure comes from CheeseListingResourceTest line 44. Once we use a *valid* owner IRI, validation *still* fails because... of course... right now our new validator *always* adds a violation.

Let's fix that next: let's add *real* logic to make sure the $owner is set to the currently-authenticated user. *Then* we'll go further and allow admin users to set the $owner to anyone.

Chapter 35: Security Logic in the Validator

Our custom validator *is* being used... but it doesn't have any *real* logic yet: it *always* add a "violation". Meaning, this validator *always* fails. Let's fix that: let's fail if the owner - that's the $value we're validating - is being set to a User that's *different* than the currently-authenticated user.

To figure out who's logged in, add public function __construct() and autowire our favorite Security service. I'll hit Alt + Enter and go to "Initialize fields" to create that property and set it.

46 lines | src/Validator/IsValidOwnerValidator.php

```
... lines 1 - 9
class IsValidOwnerValidator extends ConstraintValidator
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }
... lines 18 - 44
}
```

Making sure the User Is Authenticated

For the logic itself, start with $user = $this->security->getUser(). For this particular operation, we've added security to *guarantee* that the user will be authenticated. But just to be safe - or in case we decide to use this validation constraint on some other operation - let's double-check that the user *is* logged in: if !$user instanceof User, add a violation.

46 lines | src/Validator/IsValidOwnerValidator.php

```
... lines 1 - 18
    public function validate($value, Constraint $constraint)
    {
... lines 21 - 26
        $user = $this->security->getUser();
        if (!$user instanceof User) {
... lines 29 - 32
        }
... lines 34 - 43
    }
... lines 45 - 46
```

I could hardcode the message... but to be a bit fancier - and make it configurable each time we use this constraint - on the annotation class, add a public property called $anonymousMessage set to:

> Cannot set owner unless you are authenticated

20 lines | src/Validator/IsValidOwner.php

```
... lines 1 - 9
class IsValidOwner extends Constraint
{
... lines 12 - 15
    public $message = 'Cannot set owner to a different user';
... lines 17 - 18
}
```

Back in the validator, do the same thing as below: $this->context->buildViolation(), then the message - $constraint->anonymousMessage - and ->addViolation(). Finally, return so the function doesn't keep running: set the validation error and exit.

```
46 lines | src/Validator/IsValidOwnerValidator.php
... lines 1 - 9
class IsValidOwnerValidator extends ConstraintValidator
{
... lines 12 - 18
    public function validate($value, Constraint $constraint)
    {
... lines 21 - 27
        if (!$user instanceof User) {
            $this->context->buildViolation($constraint->anonymousMessage)
                ->addViolation();

            return;
        }
... lines 34 - 43
    }
}
```

Checking the Owner

At this point, we know the user is authenticated. We *also* know that the $value variable *should* be a User object... because we're expecting that the IsValidOwner annotation will be set on a property that contains a User. But... because I *love* making mistakes... I might someday accidentally put this onto some *other* property that does *not* contain a User object. If that happens, no problem! Let's send future me a clear message that I'm doing something... well, kinda silly: if !$value instanceof User, throw new \InvalidArgumentException() with

> @IsValidOwner constraint must be put on a property containing a User object

```
46 lines | src/Validator/IsValidOwnerValidator.php
... lines 1 - 18
    public function validate($value, Constraint $constraint)
    {
... lines 21 - 34
        if (!$value instanceof User) {
            throw new \InvalidArgumentException('@IsValidOwner constraint must be put on a property containing a User object');
        }
... lines 38 - 43
    }
... lines 45 - 46
```

I'm not using a violation here because this isn't a user-input problem - it's a programmer bug.

Finally, we know that both $user and $value are User objects. All we need to do now is compare them. So if $value->getId() !== $user->getId(), we have a problem. And yes, you could also compare the objects themselves instead of the id's.

Move the violation code into the if statement and... we're done!

46 lines | src/Validator/IsValidOwnerValidator.php

```
... lines 1 - 18
    public function validate($value, Constraint $constraint)
    {
... lines 21 - 39
        if ($value->getId() !== $user->getId()) {
            $this->context->buildViolation($constraint->message)
                ->addViolation();
        }
    }
... lines 45 - 46
```

If someone sends the owner IRI of a different User, boom! Validation error! Let's see if our test passes:

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

And... it does!

Allowing Admins to do Anything

The *nice* thing about this setup is that the owner field is *still* part of our API. We did that for a very specific reason: we want to make it possible for an *admin* user to send an owner field set to *any* user. Right now, our validator would *block* that. So... let's make it a *little* bit smarter.

Because we already have the Security object autowired here, jump straight to check for the admin role: if $this->security->isGranted('ROLE_ADMIN'), then return. That will prevent the *real* owner check from happening below.

51 lines | src/Validator/IsValidOwnerValidator.php

```
... lines 1 - 18
    public function validate($value, Constraint $constraint)
    {
... lines 21 - 34
        // allow admin users to change owners
        if ($this->security->isGranted('ROLE_ADMIN')) {
            return;
        }
... lines 39 - 48
    }
... lines 50 - 51
```

Requiring Owner

A few minutes ago, we talked about how the validator starts by checking to see if the $value is null. That would happen if an API client simply *forgets* to send the owner field. In that situation, our validator does *not* add a violation. Instead, it returns... which basically means:

> In the eyes of this validator, the value *is* valid.

We did that because *if* you want the field to be required, that should be done by adding a *separate* validation constraint - the NotBlank annotation. The fact that we're *missing* this is a bug... and let's prove it before we fix it.

In CheeseListingResourceTest, move this $cheesyData up above one more request.

```
76 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 13
    public function testCreateCheeseListing()
    {
... lines 16 - 24
        $cheesyData = [
            'title' => 'Mystery cheese... kinda green',
            'description' => 'What mysteries does it hold?',
            'price' => 5000
        ];
... lines 30 - 44
    }
... lines 46 - 74
}
```

This request is the one that makes sure that, after we log in, we *are* authorized to use this operation: we get a 400 status code due to a validation error, but not the 403 that we would expect if we were denied access.

Now, pass $cheesyData to that operation. We should *still* get a 400 response: we're passing *some* data, but we're still missing the owner field... which *should* be required.

```
76 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 13
    public function testCreateCheeseListing()
    {
... lines 16 - 30
        $client->request('POST', '/api/cheeses', [
            'json' => $cheesyData,
        ]);
... lines 34 - 44
    }
... lines 46 - 76
```

However, when we run the test:

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

It explodes into a big, giant 500 error! It's trying to insert a record into the database with a null owner_id column.

To fix this, above owner, add the missing @Assert\NotBlank().

```
209 lines | src/Entity/CheeseListing.php
... lines 1 - 48
class CheeseListing
{
... lines 51 - 95
    /**
... lines 97 - 100
     * @Assert\NotBlank()
     */
    private $owner;
... lines 104 - 207
}
```

The value must now *not* be blank *and* it must be a valid owner. Try the test again:

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

It's green! Next, allowing owner to be part of our API is great for admin users. But it's kind of inconvenient for *normal* users. Could we *allow* the owner field to be sent when creating a CheeseListing... but automatically set it to the currently-authenticated user if the field is *not* sent? And if so, how? That's next.

Chapter 36: Auto-set the Owner: Entity Listener

We decided to make the owner property a field that an API client *must* send when creating a CheeseListing. That gives us some flexibility: an admin user can send this field set to *any* User, which might be handy for a future admin section. To make sure the owner is valid, we've added a custom validator that even has an edge-case that allows admin users to do this.

But the most *common* use-case - when a normal user wants to create a CheeseListing under their *own* account - is a bit annoying: they're *forced* to pass the owner field... but it must be set to their *own* user's IRI. That's perfectly explicit and straightforward. But... couldn't we make life easier by automatically setting owner to the currently-authenticated user if that field isn't sent?

Let's try it... but start by doing this in our test... which will be a *tiny* change. When we send a POST request to /api/cheeses with title, description and price, we expect it to return a 400 error because we forgot to send the owner field. Let's change this to expect a 201 status code. Once we finish this feature, only sending title, description and price *will* work.

76 lines | tests/Functional/CheeseListingResourceTest.php

```
// ... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
// ... lines 12 - 13
    public function testCreateCheeseListing()
    {
// ... lines 16 - 33
        $this->assertResponseStatusCodeSame(201);
// ... lines 35 - 44
    }
// ... lines 46 - 74
}
```

To start, take off the NotBlank constraint from $owner - we definitely don't want it to be required anymore.

208 lines | src/Entity/CheeseListing.php

```
// ... lines 1 - 48
class CheeseListing
{
// ... lines 51 - 95
    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="cheeseListings")
     * @ORM\JoinColumn(nullable=false)
     * @Groups({"cheese:read", "cheese:collection:post"})
     * @IsValidOwner()
     */
    private $owner;
// ... lines 103 - 206
}
```

If we run the tests now...

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

Yep! It fails... we're getting a 500 error because it's trying to insert into cheese_listing with an owner_id that is null.

How to Automatically set the Field?

So, how *can* we automatically set the owner on a CheeseListing if it's not already set? We have a few options! Which in programming... is almost never a good thing. Hmm. Don't worry, I'll tell you which one *I* would use and why.

Our options include an API Platform event listener - a topic we haven't talked about yet - an API Platform data persister or a Doctrine event listener. The first two - an API Platform event listener or data persister - have the same possible downside: the owner would only be automatically set when a CheeseListing is created through the API. Depending on what you're trying to accomplish, that might be *exactly* what you want - you may want this magic to *only* affect your API operations.

But... in general... if I save a CheeseListing - no matter if it's being saved as part of an API call or in some *other* part of my system - and the owner is null, I think automatically setting the owner makes sense. So, instead of making this feature *only* work for our API endpoints, let's use a Doctrine event listener and make it work *everywhere*.

Event Listener vs Entity Listener

To set this via Doctrine, we can create an event listener *or* an "entity" listener... which are basically two, effectively *identical* ways to run some code before or after an entity is saved, updated or deleted. We'll use an "entity" listener.

In the src/ directory, create a Doctrine/ directory... though, like usual, the name of the directory and class doesn't matter. Put a new class inside called, how about, CheeseListingSetOwnerListener. This will be an "entity listener": a class with one or more functions that Doctrine will call before or after certain things happen to a specific entity. In our case, we want to run some code *before* a CheeseListing is created. That's called "pre persist" in Doctrine. Add public function prePersist() with a CheeseListing argument.

13 lines | src/Doctrine/CheeseListingSetOwnerListener.php

```
... lines 1 - 4
use App\Entity\CheeseListing;

class CheeseListingSetOwnerListener
{
    public function prePersist(CheeseListing $cheeseListing)
    {
    }
}
```

Two things about this. First, the name of this method *is* important: Doctrine will look at all the public functions in this class and use the *names* to determine which methods should be called when. Calling this prePersist() will mean that Doctrine will call us *before* persisting - i.e. inserting - a CheeseListing. You can also add other methods like postPersist(), preUpdate() or preRemove().

@ORM\EntityListeners Annotation

Second, this method will *only* be called when a CheeseListing is being saved. How does Doctrine know to only call this entity listener for cheese listings? Well, it doesn't happen magically thanks to the type-hint. Nope, to hook all of this up, we need to add some config to the CheeseListing entity. At the top, add a new annotation. Actually... let's reorganize the annotations first... and move @ORM\Entity to the bottom... so it's not mixed up in the middle of all the API Platform stuff. Now add @ORM\EntityListeners() and pass this an array with one item inside: the *full* class name of the entity listener class: App\Doctrine\... and then I'll get lazy and copy the class name: CheeseListingSetOwnerListener.

209 lines | src/Entity/CheeseListing.php

```
... lines 1 - 17
/**
... lines 19 - 47
 * @ORM\EntityListeners({"App\Doctrine\CheeseListingSetOwnerListener"})
 */
class CheeseListing
... lines 51 - 209
```

That's it for the basic setup! Thanks to this annotation and the method being called prePersist(), Doctrine will automatically call this *before* it persists - meaning *inserts* - a new CheeseListing.

Entity Listener Logic

The logic for *setting* the owner is pretty simple! To find the currently-authenticated user, add an __construct() method, type-hint the Security service and then press Alt + Enter and select "Initialize fields" to create that property and set it.

```
28 lines | src/Doctrine/CheeseListingSetOwnerListener.php
... lines 1 - 5
use Symfony\Component\Security\Core\Security;
... line 7
class CheeseListingSetOwnerListener
{
    private $security;
... line 11
    public function __construct(Security $security)
    {
        $this->security = $security;
    }
... lines 16 - 26
}
```

Next, inside the method, start by seeing if the owner was already set: if $cheeseListing->getOwner(), just return: we don't want to override that.

```
28 lines | src/Doctrine/CheeseListingSetOwnerListener.php
... lines 1 - 16
    public function prePersist(CheeseListing $cheeseListing)
    {
        if ($cheeseListing->getOwner()) {
            return;
        }
... lines 22 - 25
    }
... lines 27 - 28
```

Then if $this->security->getUser() - so *if* there is a currently-authenticated User, call $cheeseListing->setOwner($this->security->getUser()).

```
28 lines | src/Doctrine/CheeseListingSetOwnerListener.php
... lines 1 - 16
    public function prePersist(CheeseListing $cheeseListing)
    {
        if ($cheeseListing->getOwner()) {
... lines 20 - 22
        if ($this->security->getUser()) {
            $cheeseListing->setOwner($this->security->getUser());
        }
    }
... lines 27 - 28
```

Cool! Go tests go!

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

And... it passes! I'm kidding... that *exploded*. Hmm, it says:

> Too few arguments to CheeseListingSetOwnerListener::__construct() 0 passed.

Huh. Who's instantiating that class? Usually in Symfony, we expect any "service class" - any class that's *not* a simple data-holding object like our entities - to be instantiated by Symfony's container. That's important because Symfony's container is responsible for all the autowiring magic.

But... if you look at the stack trace... it looks like Doctrine *itself* is trying to instantiate the class. Why is Doctrine trying to create this object *instead* of asking the *container* for it?

Tagging the Service

The answer is... that's... sort of... just how it works? Um, ok, better explanation. When used as an independent library, Doctrine typically handles instantiating these "entity listener" classes itself. However, when integrated with Symfony, you *can* tell Doctrine to *instead* fetch that service from the container. But... you need a *little* bit of extra config.

Open config/services.yaml and override the automatically-registered service definition: App\Doctrine\ and go grab the CheeseListingSetOwnerListener class name again. We're doing this so that we can add a *little* bit of *extra* service configuration. Specifically, we need to add a *tag* called doctrine.orm.entity_listener.

```
43 lines | config/services.yaml
... lines 1 - 8
services:
... lines 10 - 41
    App\Doctrine\CheeseListingSetOwnerListener:
        tags: [doctrine.orm.entity_listener]
```

This says:

> Hey Doctrine! This service is an *entity listener*. So when you need the CheeseListingSetOwnerListener object to do the entity listener stuff, use *this* service instead of trying to instantiate it yourself.

And *that* will let Symfony do its normal, autowiring logic. Try the test one last time:

```
$ php bin/phpunit --filter=testCreateCheeseListing
```

And... we're good! We've got the best of all worlds! The flexibility for an API client to *send* the owner property, validation when they do, and an automatic fallback if they don't.

Next, let's talk about the last big piece of access control: filtering a collection result to only the items that an API client should see. For example, when we make a GET request to /api/cheeses, we should probably *not* return *unpublished* cheese listings... unless you're an admin.

Chapter 37: Query Extension: Auto-Filter a Collection

The CheeseListing entity has a property on it called $isPublished, which defaults to false. We haven't talked about this property much, but the idea is pretty simple: when a CheeseListing is first created, it will *not* be published. Then, once a user has perfected all their cheesy details, they will "publish" it and only *then* will it be publicly available on the site.

This means that we have some work to do! We need to automatically filter the CheeseListing collection to *only* show *published* items.

Adding a Test

Let's put this into a simple test first. Inside CheeseListingResourceTest add public function testGetCheeseListingCollection() and then go to work: $client = self::createClient() and $user = $this->createUser() with any email and password.

106 lines | tests/Functional/CheeseListingResourceTest.php

```
// ... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
// ... lines 12 - 75
    public function testGetCheeseListingCollection()
    {
        $client = self::createClient();
        $user = $this->createUser('cheeseplese@example.com', 'foo');
// ... lines 80 - 103
    }
}
```

We're not logging *in* as this user simply because the CheeseListing collection operation doesn't require authentication. Now, let's make some cheese! I'll create $cheeseListing1... with a bunch of data... then use that to create $cheeseListing2... and $cheeseListing3. These are nice, boring, delicious CheeseListing objects. To save them, grab the entity manager - $em = $this->getEntityManager() - persist all three... and call flush().

```
106 lines | tests/Functional/CheeseListingResourceTest.php
    ... lines 1 - 75
    public function testGetCheeseListingCollection()
    {
    ... lines 78 - 80
        $cheeseListing1 = new CheeseListing('cheese1');
        $cheeseListing1->setOwner($user);
        $cheeseListing1->setPrice(1000);
        $cheeseListing1->setDescription('cheese');

        $cheeseListing2 = new CheeseListing('cheese2');
        $cheeseListing2->setOwner($user);
        $cheeseListing2->setPrice(1000);
        $cheeseListing2->setDescription('cheese');

        $cheeseListing3 = new CheeseListing('cheese3');
        $cheeseListing3->setOwner($user);
        $cheeseListing3->setPrice(1000);
        $cheeseListing3->setDescription('cheese');

        $em = $this->getEntityManager();
        $em->persist($cheeseListing1);
        $em->persist($cheeseListing2);
        $em->persist($cheeseListing3);
        $em->flush();
    ... lines 101 - 103
    }
    ... lines 105 - 106
```

The stage for our test is set. Because the default value for the isPublished property is false... all of these new listings will be *unpublished*. Fetch these by using $client->request() to make a GET request to /api/cheeses.

```
106 lines | tests/Functional/CheeseListingResourceTest.php
    ... lines 1 - 75
    public function testGetCheeseListingCollection()
    {
    ... lines 78 - 101
        $client->request('GET', '/api/cheeses');
    ... line 103
    }
    ... lines 105 - 106
```

Because we haven't added any logic to hide *unpublished* listings yet, at this moment, we would expect this to return *3* results. Move over to the docs... and try the operation. Ah, cool! Hydra adds a very useful field: hydra:totalItems. Let's use that! In the test, $this->assertJsonContains() that there will be a hydra:totalItems field set to 3.

```
106 lines | tests/Functional/CheeseListingResourceTest.php
    ... lines 1 - 75
    public function testGetCheeseListingCollection()
    {
    ... lines 78 - 102
        $this->assertJsonContains(['hydra:totalItems' => 3]);
    }
    ... lines 105 - 106
```

Copy the method name and flip over to your terminal: this test *should* pass because we have *not* added any filtering yet. Try it:

```
$ php bin/phpunit --filter=testGetCheeseListingCollection
```

And... green! I love when there are no surprises.

Now let's update the test for the behavior that we *want*. Allow $cheeseListing1 to stay *not* published, but publish the other two: $cheeseListing2->setIsPublished(true)... and then paste that below and rename it for $cheeseListing3.

Once we're done with the feature, we will only want the second *two* to be returned. Change the assertion to 2. Now we have a failing test.

108 lines | tests/Functional/CheeseListingResourceTest.php

```
    ... lines 1 - 75
    public function testGetCheeseListingCollection()
    {
    ... lines 78 - 89
        $cheeseListing2->setIsPublished(true);
    ... lines 91 - 95
        $cheeseListing3->setIsPublished(true);
    ... lines 97 - 104
        $this->assertJsonContains(['hydra:totalItems' => 2]);
    }
    ... lines 107 - 108
```

Automatic Filtering

So... how can we make this happen? How can we hide unpublished cheese listings? Well... in the first tutorial we talked about filters. For example, we added a boolean filter that allows us to add a ?isPublished=1 or ?isPublished=0 query parameter to the /api/cheeses operation. So... actually, an API client can *already* filter out unpublished cheese listings!

That's great! The problem is that we *no* longer want an API client to be able to *control* this: we need this filter to be automatic. An API client should *never* see unpublished listings. Filters are a good choice for *optional* stuff, but not this.

Another option is called a *Doctrine* filter. It's got a similar name, but is a totally different feature that comes from Doctrine - we talk about it in our Doctrine Queries tutorial. A Doctrine filter allows you to modify a query on a *global* level... like each time we query for a CheeseListing, we could *automatically* add WHERE is_published=1 to that query.

The downside to a Doctrine filter.... is also its strength: it's automatic - it modifies *every* query you make... which can be surprising if you ever need to *purposely* query for *all* cheese listings. Sure, you can work around that - but I tend to use Doctrine filters rarely.

The solution I prefer is *specific* to API Platform... which I *love* because it means that the filtering will only affect API operations... the rest of my app will continue to behave normally. It's called a "Doctrine extension".... which is basically a "hook" for you to change how a query is built for a specific resource or even for a specific *operation* of a resource. They're... basically... awesome.

Creating the Query Collection Extension

Let's get our extension going: in the src/ApiPlatform/ directory, create a new class called CheeseListingIsPublishedExtension. Make this implement QueryCollectionExtensionInterface and then go to the Code -> Generate menu - or Command + N on a Mac - and select "Implement Methods" to generate the *one* method that's required by this interface: applyToCollection().

```
15 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php
... lines 1 - 4
use ApiPlatform\Core\Bridge\Doctrine\Orm\Extension\QueryCollectionExtensionInterface;
use ApiPlatform\Core\Bridge\Doctrine\Orm\Util\QueryNameGeneratorInterface;
use Doctrine\ORM\QueryBuilder;
... line 8
class CheeseListingIsPublishedExtension implements QueryCollectionExtensionInterface
{
    public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $reso
    {

    }
}
```

Very simply: as soon as we create a class that implements this interface, *every* single time that API Platform makes a Doctrine query for a *collection* of results, like a collection of users or a collection of cheese listings, it will call this method and pass us a pre-built QueryBuilder. Then... we can mess with that query however we want!

Tip

If you're loading data from something *other* than Doctrine (or have a custom "data provider"... a topic we haven't talked about yet), then this class won't have any effect.

In fact, this is how pagination and filters work internally: both are implemented as Doctrine *extensions*.

Query Collection Extension Logic

Let's see... even though this method will be called whenever API Platform is querying for a collection of *any* type of object, we only want to do our logic for cheese listings. No problem: if $resourceClass !== CheeseListing::class return and do nothing.

```
22 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php
... lines 1 - 11
    public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $reso
    {
        if ($resourceClass !== CheeseListing::class) {
            return;
        }
... lines 17 - 20
    }
```

Now... we just need to add the WHERE isPublished=1 part to the query. To do that will require a *little* bit of fanciness. Start with $rootAlias = $queryBuilder->getRootAlias()[0]. I'll explain that in a minute. Then $queryBuilder->andWhere() with sprintf('%s.isPublished = :isPublished') passing $rootAlias to fill in that %s part. For the isPublished parameter, set it with ->setParameter('isPublished', true).

```
22 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php
... lines 1 - 11
    public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $reso
    {
... lines 14 - 17
        $rootAlias = $queryBuilder->getRootAliases()[0];
        $queryBuilder->andWhere(sprintf('%s.isPublished = :isPublished', $rootAlias))
            ->setParameter('isPublished', true);
    }
```

This *mostly* looks like normal query logic... the weird part being that "root alias" thing. Because someone *else* originally created this query, we don't know what "table alias" they're using to represent CheeseListing. Maybe c... so c.isPublished? Or cheese... so cheese.isPublished? We don't really know. The getRootAlias() method tells us that... and then we hack it into our query.

Oh, and by the way - in addition to $resourceClass, this method receives the $operationName. Normally only the get operation - like GET /api/cheese - would cause a query for a *collection* to be made... but if you created some custom operations and needed to tweak the query on an operation-by-operation basis, the $operationName can let you do that.

Anyways... we should be done! Run that test again:

```
$ php bin/phpunit --filter=testGetCheeseListingCollection
```

And... green! We are successfully hiding the unpublished cheese listings. Isn't that nice?

Next, let's add logic so that *admin* users will *still* see unpublished cheese listings. And what about the *item* operation? Sure, an unpublished cheese listing won't be included in the *collection* endpoint... but how can we prevent a user from making a GET request directly to fetch a *single*, unpublished CheeseListing?

Chapter 38: Automatic 404 on Unpublished Items

Unpublished cheese listings will *no* longer be returned from our collection endpoint: this extension class has taken care of that. Of course... if we want to have some sort of an admin section where admin users can see *all* cheese listings... that's a problem... because we've just filtered them out entirely!

No worries, let's add the same admin "exception" that we've added to a few other places. Start with public function __construct() so we can autowire the Security service. I'll hit Alt + Enter and click "Initialized fields" to create that property and set it.

34 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php

```
... lines 1 - 8
use Symfony\Component\Security\Core\Security;
... line 10
class CheeseListingIsPublishedExtension implements QueryCollectionExtensionInterface
{
    private $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }
... lines 19 - 33
}
```

Down in the method, very nicely, if $this->security->isGranted('ROLE_ADMIN'), return and do nothing.

34 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php

```
... lines 1 - 19
    public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $reso
    {
... lines 22 - 25
        if ($this->security->isGranted('ROLE_ADMIN')) {
            return;
        }
... lines 29 - 32
    }
```

Whoops, I added an extra exclamation point to make this *not*. Don't do that! I'll fix it in a few minutes.

Anyways, apart from my mistake, admin users can now fetch *every* CheeseListing once again.

Testing for 404 on Unpublished Items

That takes care of the collection stuff. But we're not done yet! We also don't want a user to be able to fetch an *individual* CheeseListing if it's unpublished. The collection query extension does *not* take care of this: this method is only called when API Platform needs to query for a *collection* of items - a different query is used for a *single* item.

Let's write a quick test for this. Copy the collection test method, paste the entire thing, rename it to testGetCheeseListingItem()... and I'll remove cheese listings two and three. This time, make the GET request to /api/cheeses/ and then $cheeseListing1->getId().

This is an *unpublished* CheeseListing... so we eventually want this to *not* be accessible. But... because we haven't added the logic yet, let's start by testing the current functionality. Assert that the response code is 200.

```
126 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 107
    public function testGetCheeseListingItem()
    {
        $client = self::createClient();
        $user = $this->createUser('cheeseplese@example.com', 'foo');

        $cheeseListing1 = new CheeseListing('cheese1');
        $cheeseListing1->setOwner($user);
        $cheeseListing1->setPrice(1000);
        $cheeseListing1->setDescription('cheese');

        $em = $this->getEntityManager();
        $em->persist($cheeseListing1);
        $em->flush();

        $client->request('GET', '/api/cheeses/'.$cheeseListing1->getId());
        $this->assertResponseStatusCodeSame(200);
    }
}
```

Copy that method name, and let's make sure it passes:

```
$ php bin/phpunit --filter=testGetCheeseListingItem
```

It does! But... that's not the behavior we want. To make this *really* obvious, let's say $cheeseListing->setIsPublished(false). That CheeseListing was already unpublished - that's the default - but this is more clear to me. For the status code, when a CheeseListing is unpublished, we *want* it to return a 404. Try the test now:

```
127 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 107
    public function testGetCheeseListingItem()
    {
... lines 110 - 116
        $cheeseListing1->setIsPublished(false);
... lines 118 - 123
        $this->assertResponseStatusCodeSame(404);
    }
... lines 126 - 127
```

```
$ php bin/phpunit --filter=testGetCheeseListingItem
```

Failing! We're ready.

The QueryItemExtensionInterface

So if the applyToCollection() method is only called when API Platform is making a query for a *collection* of items... how can we modify the query when API Platform needs a *single* item? Basically... the same way! Add a second interface: QueryItemExtensionInterface. This requires us to have one new method. Go to the Code -> Generate menu - or Command + N on a Mac - and select "Implement Methods" one more time. And... ha! We could have guessed that method name: applyToItem(). This is called whenever API Platform is making a query for a *single* item... and we basically want to make the *exact* same change to the query.

```
45 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php
... lines 1 - 5
use ApiPlatform\Core\Bridge\Doctrine\Orm\Extension\QueryItemExtensionInterface;
... lines 7 - 11
class CheeseListingIsPublishedExtension implements QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
... lines 14 - 25
    public function applyToItem(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $resourceC
    {
... line 28
    }
... lines 30 - 44
}
```

I'll hit Control+t, which, on a Mac, is the same as going to the Refactor menu on top and selecting "Refactor this". Let's extract this logic to a "Method" - call it addWhere.

```
45 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php
... lines 1 - 11
class CheeseListingIsPublishedExtension implements QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
... lines 14 - 20
    public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $reso
    {
        $this->addWhere($queryBuilder, $resourceClass);
    }
... lines 25 - 30
    private function addWhere(QueryBuilder $queryBuilder, string $resourceClass): void
    {
        if ($resourceClass !== CheeseListing::class) {
            return;
        }

        if ($this->security->isGranted('ROLE_ADMIN')) {
            return;
        }

        $rootAlias = $queryBuilder->getRootAliases()[0];
        $queryBuilder->andWhere(sprintf('%s.isPublished = :isPublished', $rootAlias))
            ->setParameter('isPublished', true);
    }
}
```

Cool! That gives us a new private function addWhere()... and applyToCollection() is already calling it. Do the same thing in applyToItem().

```
45 lines | src/ApiPlatform/CheeseListingIsPublishedExtension.php
    ... lines 1 - 25
    public function applyToItem(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $resourceC
    {
        $this->addWhere($queryBuilder, $resourceClass);
    }
    ... lines 30 - 45
```

Let's try this! Run the test again and...

```
$ php bin/phpunit --filter=testGetCheeseListingItem
```

It fails? Hmm. Oh... I reversed the check for ROLE_ADMIN. Get rid of that exclamation point... and try that test again.

```
$ php bin/phpunit --filter=testGetCheeseListingItem
```

We are green! How cool was that? We're able to modify the collection *and* item queries for a specific resource with one class and two methods.

There's just one more problem: the collection of cheese listings is returned in *two* places - the GET operation to /api/cheeses and... somewhere else. And that "somewhere else" is *not* filtering out the unpublished cheese listings. Oh nooooooo.

Let's find out more and fix that next!

Chapter 39: Filtering Related Collections

There are two places where our API returns a collection of cheese listings. The first is the GET operation for /api/cheeses... and our extension class takes care of filtering out unpublished listings. The second... is down here, when you fetch a single user. Remember - we decided to *embed* the collection of cheese listings that are owned by this user. But... surprise! Our query extension class does *not* filter this! Why? The extension class is only used when API Platform needs to make a *direct* query for a CheeseListing. In practice, this means it's used for the CheeseListing operations. But for a User resource, API platform queries for the User and then, to get the cheeseListings field, it simply calls $user->getCheeseListings(). And guess what? That method returns the *full*, unfiltered collection of related cheese listings.

Careful with Collections

When you decide to expose a collection relation like this in your API, I want you to keep something in mind: exposing a collection relationship is only practical if you know that the number of related items will always be... reasonably small. If a user could have *hundreds* of cheese listings... well... then API Platform will need to query, hydrate and return *all* of them whenever someone fetches that user's data. That's overkill and will *really* slow things down... if not eventually kill that API call entirely. In that case, it would be better to *not* expose a cheeseListings property on User... and instead instruct an API client to make a GET request to /api/cheeses & use the owner filter. The response will be paginated, which will keep things at a reasonable size.

IRIs Instead of Embedded Data?

But if you *do* know that a collection will never become too huge and you *do* want to expose it like this... how can we hide the unpublished listings? There are two options. Well... the first is only a partial solution: instead of embedding these two properties... and potentially exposing the data of an unpublished CheeseListing, you could configure API Platform to only return the IRI string.

As a reminder, each item under cheeseListings contains two fields: title and price. Why only those two fields? Because, in the CheeseListing entity, the title property is in a group called user:read... and the price property is *also* in that group. When API Platform serializes a User, we've configured it to use the user:read normalization group. By putting these two properties into that group, we're telling API Platform to *embed* these fields.

If we removed the user:read group from *all* the properties in CheeseListing, the cheeseListings field on User would suddenly become an array or IRI strings... *instead* of embedded data.

Why does that help us? Well... it sort of doesn't. That field would *still* contain the IRI's for *all* cheese listings owned by this user... but if an API client made a request to the IRI of an unpublished listing, it would 404. They wouldn't be able to see the *data* of the unpublished listing... which is great... but the IRI *would* still show up here... which is kinda weird.

Truly Filtering the Collection

If you *really* want to filter this properly, if you *really* want the cheeseListings property to only contain *published* listings, we *can* do that.

Let's modify our test a little to look for this. After we make a GET request for our unpublished CheesesListing and assert the 404, let's *also* make a GET request to /api/users/ and then $user->getId() - the id of the $user we created above that owns this CheeseListing. Change that line to createUserAndLogIn() and pass $client... because you need to be authenticated to fetch a single user's data.

```
131 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 107
    public function testGetCheeseListingItem()
    {
... line 110
        $user = $this->createUserAndLogIn($client, 'cheeseplese@example.com', 'foo');
... lines 112 - 125
        $client->request('GET', '/api/users/'.$user->getId());
... lines 127 - 128
    }
... lines 130 - 131
```

After the request, fetch the returned data with $data = $client->getResponse()->toArray(). We want to assert that the cheeseListings property is empty: this User *does* have one CheeseListing... but it's not published. Assert that with $this->assertEmpty($data['cheeseListings']).

```
131 lines | tests/Functional/CheeseListingResourceTest.php
... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
... lines 12 - 107
    public function testGetCheeseListingItem()
    {
... lines 110 - 126
        $data = $client->getResponse()->toArray();
        $this->assertEmpty($data['cheeseListings']);
    }
... lines 130 - 131
```

Let's make sure this fails...

```
$ php bin/phpunit --filter=testGetCheeseListingItem
```

And... it does:

> Failed asserting that an array is empty.

Adding getPublishedCheeseListings()

Great! So... how *can* we filter this collection? Let's think about it: we know that API Platform calls getCheeseListings() to get the data for the cheeseListings field. So... what if we made this method return only *published* cheese listings?

Yea... that's the key! Well, except... I don't want to modify *that* method: it's a getter method for the cheeseListings property... so it really should return that property exactly. Instead, create a new method: public function getPublishedCheeseListings() that will also return a Collection. Inside, return $this->cheeseListings->filter(), which is a method on Doctrine's collection object. Pass this a callback function(){} with a single CheeseListing argument. All that function needs is return $cheeseListing->getIsPublished().

250 lines | src/Entity/User.php

```
... lines 1 - 37
class User implements UserInterface
{
... lines 40 - 195
    public function getPublishedCheeseListings(): Collection
    {
        return $this->cheeseListings->filter(function(CheeseListing $cheeseListing) {
            return $cheeseListing->getIsPublished();
        });
    }
... lines 202 - 248
}
```

If you're not familiar with the filter() method, that's ok - it's a bit more common in the JavaScript world... or "functional programming" in general. The filter() method will loop over *all* of the CheeseListing objects in the collection and execute the callback for each one. If our callback returns true, that CheeseListing is added to a *new* collection... which is ultimately returned. If our callback returns false, it's not.

The end result is that this method returns a collection of only the *published* CheeseListing objects... which is perfect! Side note: this method is inefficient because Doctrine will query for *all* of the related cheese listings... just so we can then filter that list and return only *some* of them. If the number of items in the collection will always be pretty small, no big deal. But if you're worried about this, there *is* a more efficient way to filter the collection at the database level, which we talk about in our Doctrine Relations tutorial.

But no matter *how* you filter the collection, you'll now have a *new* method that returns only the *published* listings. Let's make it part of our API! Find the $cheeseListings property. Right now this is in the user:read and user:write groups. Copy that and take it *out* of the user:read group. We still want to *write* directly to this field... by letting the serializer call our addCheeseListing() and removeCheeseListing() methods, but we *won't* use it for *reading* data.

Instead, above the new method, paste the @Groups and put this in *just* user:read. If we stopped now, this would give us a new publishedCheeseListings property. We can improve that by adding @SerializedName("cheeseListings").

250 lines | src/Entity/User.php

```
... lines 1 - 37
class User implements UserInterface
{
... lines 40 - 183
    /**
     * @return Collection|CheeseListing[]
     */
    public function getCheeseListings(): Collection
    {
        return $this->cheeseListings;
    }

    /**
     * @Groups({"user:read"})
     * @SerializedName("cheeseListings")
     */
    public function getPublishedCheeseListings(): Collection
... lines 197 - 248
}
```

I love it! Our API *still* exposes a cheeseListings field... but it will now *only* contain *published* listings. But don't take my word for it, run that test!

```
$ php bin/phpunit --filter=testGetCheeseListingItem
```

Yes! It passes! To be safe, let's run *all* the tests:

```
$ php bin/phpunit
```

And... ooh - we do get one failure from testUpdateCheeseListing():

> Failed asserting that Response status code is 403

It looks like we got a 404. Find testUpdateCheeseListing(). The failure is coming from down here on line 67. We're testing that you can't update a CheeseListing that's owned by a different user... but instead of getting a 403, we're getting a 404.

The problem is that this CheeseListing is not published. This is *awesome*! Our query extension class is preventing us from fetching a single CheeseListing for editing... because it's not published. I wasn't even thinking about this case, but API Platform acted intelligently. Sure, you'll probably want to tweak the query extension class to allow for an *owner* to fetch their *own* unpublished cheese listings... but I'll leave that step for you.

Let's set this to be published... and run the test again:

132 lines | tests/Functional/CheeseListingResourceTest.php

```
// ... lines 1 - 9
class CheeseListingResourceTest extends CustomApiTestCase
{
// ... lines 12 - 46
    public function testUpdateCheeseListing()
    {
// ... lines 49 - 56
        $cheeseListing->setIsPublished(true);
// ... lines 58 - 74
    }
// ... lines 76 - 130
}
```

```
$ php bin/phpunit
```

All green! That's it friends! We made it! We added *one* type of API authentication - with a plan to discuss other types more in a future tutorial - and then customized access in *every* possible way I could think of: preventing access on an operation-by-operation basis, voters for more complex control, hiding fields based on the user, adding custom fields based on the user, validating data... again... based on who is logged in and even controlling database queries based on security. That... was awesome!

In an upcoming tutorial, we'll talk about custom operations, DTO objects and... any other customizations we can dream up. Are we still missing something you want covered? Let us know! In the mean time, go create some mean API endpoints and let us know what cool thing it's powering.

Alright friends, see ya next time!