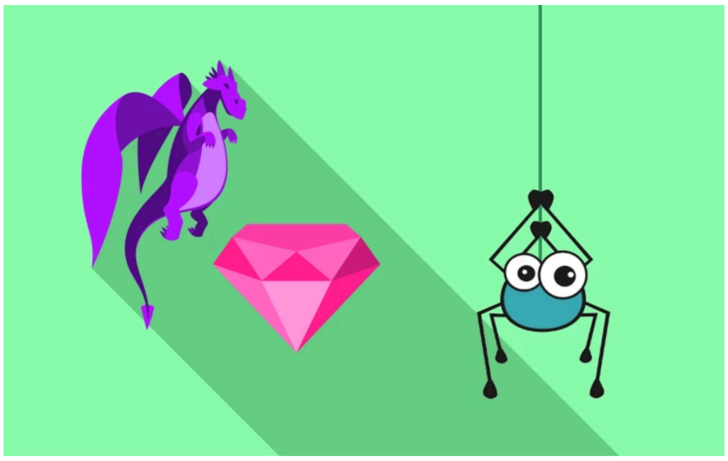
API Platform 3 Part 2: Security for your Treasures



Chapter 1: API Docs on Production?

Welcome back you wonderful JSON-returning people, to API Platform episode 2. In part 1, we got busy! We created a pretty killer API to store dragon treasures, though... we completely forgot to add security! Any small, hairy-footed creature could sneak in a back door... and we'd have absolutely no idea! So this time, we're talking *everything* related to security. Like authentication: should I use a session with a login form... or do I need API tokens? And authorization, like denying access to entire endpoints. Then we'll get into trickier things like showing or hiding results based on the user and even showing or hiding certain *fields* based on the user. We'll also talk about totally custom fields, the PATCH HTTP method and setting up an API test system your friends will be jealous of.

Project Setup

Now, you know the drill: to *really* dig into this stuff, you should code along with me. Download the code course code from this page. After you unzip it, you'll find a `start/` directory with the same code that you see here. Pop open this nifty `README.md` file and go through all the setup instructions.

I'm all the way down here at starting the `symfony` web server. So I'll spin over to a terminal that's already inside the project and run

```
symfony serve -d
```

to start a local web server in the background. Perfect! I'll hold `Cmd` and click that URL to pop that open in my browser. Hello Treasure Connect! This is the app we created in episode 1... though we worked exclusively on the API. We created endpoints for treasures, users *and* the ability to relate them.

This homepage is brand new for episode 2. It's a small Vue app that I built. It has a login form... but it doesn't work yet: it will be up to *us* to bring it to life.

Interactive Docs on Production?

Now before we dive into security, one question I sometimes get is:

> *“Hey Ryan, the interactive docs are super cool... but could I hide them on production?”*

If your API is private - it's just meant for your JavaScript - that might make sense because you don't want to advertise your endpoints to the world. However, I don't feel *too* compelled to hide the docs... because even if you do, the endpoints *still* exist. So you're going to need proper security anyways.

But yes, hiding them is possible, so let's see how. Even if you *will* show your docs, this is kind of an interesting process that shows how various parts of the system work together.

Find your terminal and run:

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

Remember: this command show all the *possible* configuration for API Platform. Let's see... search for "swagger". There we go. There's a section with things like `enable_swagger`, `enable_swagger_ui`, `enable_re_doc`, `enable_entrypoint`, and `enable_docs`. What does all that mean?

Hello ReDoc

First I want to show you what ReDoc is, because we didn't talk about that in the first tutorial. We're currently looking at the Swagger version of our documentation. But there's a competing format called ReDoc... and you can click on the "ReDoc" link at the bottom to see it! Yup! This is the *same* documentation info... but with a different layout! If you like this, it's there for you.

Disabling The Docs

Anyways, back at the terminal, there are a lot of "enable" configs. They're all related... but slightly different. For example, `enable_swagger` really refers to the OpenAPI documentation. Remember that's the JSON document that powers the Swagger and ReDoc API docs. Then, these are whether or not we want to *show* those two types of documentation frontends. And down here, `enable_entrypoint` and `enable_docs` control whether or not certain *routes* are added to our app.

I bet that didn't *completely* make sense, so let's play with this. Pretend that we want to disable the docs entirely. Ok! Open `config/packages/api_platform.yaml` and, to start, add `enable_docs: false`:

```
config/packages/api_platform.yaml
api_platform:
// ... lines 2 - 7
    enable_docs: false
```

As soon as you do that and refresh... alright! Our API documentation is gone... but with a 500 error. When you `enable_docs: false`, it literally removes the *route* to our documentation.

Let's back up. Going to `/api` was always kind of a shortcut to get to the docs. The *real* path was `/api/docs`, `/api/docs.json` or `.jsonld`. And these *are* now all 404's because we disabled that route. So yay our documentation is gone!

However, when you go to `/api`, this actually *isn't* a documentation page. This is known as the "entry point": it's our API homepage. This page *does* still exist... but it tries to *link* to our API docs... which don't exist, and it explodes.

To disable the entry point, move over and add `enable_entrypoint: false`:

```
config/packages/api_platform.yaml
api_platform:
// ... lines 2 - 8
    enable_entrypoint: false
```

Now going to `/api` give us... beautiful! A 404.

Ok, so we know we can go to `/api/treasures.json` or `.jsonld`. But what if we just go to `/api/treasures`? That... unfortunately is a 500 error! When our browser makes a request, it sends an `Accept` header that says that we want HTML. So we're asking our API for the `html` version of the treasures. And the `html` version is... the documentation. So it tries to link to the documentation and explodes.

To disable this, we can communicate to the system that we don't have Swagger or API documentation at all... so it should stop trying to link to it. Do that by setting `enable_swagger: false`:

```
config/packages/api_platform.yaml
api_platform:
// ... lines 2 - 9
    enable_swagger: false
```

Though... that just trades for another 500 error that says:

> *"Hey, you can't enable Swagger UI without enabling Swagger!"*

Fix that with `enable_swagger_ui: false`:

```
config/packages/api_platform.yaml
api_platform:
// ... lines 2 - 10
    enable_swagger_ui: false
```

And now... closer!

Disabling the HTML Format

> *"Serialization for the format `html` is not supported."*

The problem is that we're *still* requesting the `html` version of this resource. But now that we don't have any documentation, our API is like:

> *"Um... not really sure how to return an HTML version of this."*

And the truth is: if we totally disable our docs, we don't *need* an HTML format anymore! And so, we can disable it. Do that by, very simply, removing `html` from `formats`:

```
config/packages/api_platform.yaml
api_platform:
    formats:
        jsonld: [ 'application/ld+json' ]
        json: [ 'application/json' ]
        jsonhal: [ 'application/hal+json' ]

// ... lines 7 - 10
```

And... we actually have one other spot where we need to do that: in `src/Entity/DragonTreasure.php`. When we added our custom `csv` format... let's see here it is... we repeated all the formats including `html`. So take `html` off of there as well:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 26
#[ApiResource(
// ... lines 28 - 40
    formats: [
        'jsonld',
        'json',
        'jsonhal',
        'csv' => 'text/csv',
    ],
// ... lines 47 - 53
)]
// ... lines 55 - 72
class DragonTreasure
{
// ... lines 75 - 232
}
```

When we refresh now... got it! Since there's no HTML format, it defaults to `JSON-LD`. Our docs are now totally disabled.

Oh, and to disable the docs *just* for production, I would create an environment variable - like `ENABLE_API_DOCS` - then reference that in my config:

Tip

Actually, due to how the config is loaded, environment variables won't work here! Instead, you could disable docs in production only, via:

```
when@prod:
    api_platform:
        enable_swagger_ui: false
```

```
# config/packages/api_platform.yaml
api_platform:
    enable_swagger_ui: '%env(bool:ENABLE_API_DOCS)%'
```

But... I *do* like the documentation, so I'm going to undo this change... and this change as well to get our docs back.

config/packages/api_platform.yaml

```yaml
api_platform:
    formats:
        jsonld: [ 'application/ld+json' ]
        json: [ 'application/json' ]
        html: [ 'text/html' ]
        jsonhal: [ 'application/hal+json' ]

#    enable_docs: false
#    enable_entrypoint: false
#    enable_swagger: false
#    enable_swagger_ui: false
```

src/Entity/DragonTreasure.php

```php
// ... lines 1 - 26
#[ApiResource(
// ... lines 28 - 40
    formats: [
        'jsonld',
        'json',
        'html',
        'jsonhal',
        'csv' => 'text/csv',
    ],
// ... lines 48 - 54
)]
// ... lines 56 - 73
class DragonTreasure
{
// ... lines 76 - 233
}
```

Love it!

Next, let's have a fireside chat about authentication. You have a fancy API: do you need API tokens? Or something else?

Chapter 2: API Tokens? Session Cookies?

Join me, while we tell a tale as old as... the modern Internet: API authentication. A topic of hype, complexity and unlikely heroes. Characters include sessions, API tokens, OAuth, JSON web tokens! But what do we need for *our* situation?

The first thing I want you to ask is:

> *“Who will be using my API?”*

Is it your own JavaScript, or do you need to allow programmatic access? Like someone will write a script that will use your API?

We're going to go through both of these use-cases... and each has some extra complexities that we'll discuss along the way.

Everything is a Token!

By the way, when you think of API authentication, you typically think of an API token. And that's true! But it turns out that... pretty much *all* authentication is done by *some* sort of a token. Even session-based authentication is done by sending a cookie... which contains a unique, you guessed it, "token". It's a random string that PHP uses to find and load the related session data on the server.

So the trick is figuring out which *type* of token you need in each situation and how the end-user will *get* that token.

Use-Case 1: Building for your Own JavaScript

So let's talk about that first use-case: the user of your API is your own JavaScript.

Well, before we even dive into security, make sure your frontend and your API live on the same domain... like the *exact* same domain, not just a subdomain. Why? Because if they live on two different domains or subdomains, you have to deal with CORS: Cross-Origin Resource Sharing.

CORS not only adds complexity to your setup, it also hurts performance. Kévin Dunglas - the lead developer of API Platform - has a blog post about this. He even shows a strategy where your frontend and backend can live in totally different directories or repositories, but *still* live on the same domain thanks to some web server tricks.

If you *do*, for some reason, decide to put your API and frontend on different sub-domains, then you *will* need to worry about CORS headers and you can solve that with NelmioCorsBundle. But, I don't recommend it.

The case for Sessions

Anyways, back to security. If you're calling your API from your own JavaScript, the user is probably logging in via a login form with an email and password. It doesn't matter if that's a traditional login form or one that's built with a fancy JavaScript framework that submits via AJAX.

And, honestly, a *really* simple way to handle this use-case is *not* with API tokens, but with good ol' fashioned HTTP Basic authentication. Yea, where you literally pass the email & password to each endpoint. For example, the user enters their email and password, you make an API request to some endpoint just to make sure it's valid, then you store that email and password in JavaScript and send it on every single API request going forward. Your email & password works basically like an API token.

However, this has some practical challenges, like the question of *where* you securely store the email and password in JavaScript so you can continually use it. This is actually a problem in *general* with JavaScript and "credentials", including API tokens: you need to be *very* careful where you store those so that other JavaScript on your page can't read them. There *are* solutions: https://bit.ly/auth0-token-storage - but it adds complexity that you very likely don't need.

So instead, for your own JavaScript, you can use a session. When you start a session in Symfony, it returns an "HTTP only" cookie... and that cookie contains the session id. Though, the contents of the cookie aren't really important: it could be the session id or some sort of token you invented and are reading in Symfony. The really important thing is that because the cookie is "HTTP only", it can't be read by JavaScript: your JavaScript or anyone else's JavaScript. But whenever you make an API request to your domain, that cookie's *will* come with it... and your app will use it to log in the user.

So the API token in this situation is simply the "session id", which is stored securely in an HTTP-only cookie. Mmm. We will code through this use case.

Oh, and by the way, one edge-case with this situation is if you have a Single Sign On situation - an SSO. In that case, you'll authenticate with your SSO like a normal web app. When you finish, you'll have a token, which you can then use to either authenticate the user with a session like normal... or you can use that token directly from your JavaScript. That's a more advanced use case that we won't go through in this tutorial... though, we *will* talk about how to read & validate API tokens regardless of where those tokens came from.

Use-Case 2: Programmatic Access & API Tokens

The second big use-case for authentication is programmatic access. Some *code* will talk to your API... besides JavaScript from inside the browser.

In this case, the API clients absolutely *will* send some sort of an API token string. And so, you need to make your API able to read a token that's sent on each request, usually sent on an `Authorization` header:

```
$response = $httpClient->request(
    'GET',
    '/api/treasures',
    [
        'Authorization' => 'Bearer '.$apiToken,
    ],
);
```

How the user gets this token depends: there are kind of two main cases. The first one is the "GitHub personal access token" case. This is where a user can browse to a page on your site and click to create a new access token. Then they can copy that and go use it in some code.

The second big case is OAuth, which is just a fancy & secure way to *get* an access token. It's especially important when the "code" that's making the API requests is making those requests on "behalf" of some user on your system.

Like imagine a site - ReplyToAllCommentsWithHearts.com - that allows you to connect with GitHub. Once you do, that site can *then* make API requests to GitHub for your account, like making comments as your user. Or imagine an iPhone app where, to log in, you show the user the login form on your site. Then, via an OAuth flow, that mobile app will receive an access token it can use to talk to your API on behalf of that user.

We're going to talk about the personal access token method in this tutorial, including how to read and validate API tokens, no matter where they come from. We won't talk about the OAuth flow... and it's partially because it's a separate beast. Yes, if you have the use-case where you need to allow third parties to get API tokens for different users on your site, you *will* need some sort of OAuth server, whether you build it yourself or use some other solution. But once the OAuth server has done its work, the client that will talk to your API receives... a token! And then they'll use that token to talk to your API. So your API will need to read, validate, and understand that token, but it doesn't care *how* the API client got it.

Ok, let's put all this theory behind us and start going through the first use-case next: allowing our JavaScript to log in by sending an AJAX request.

Chapter 3: API Login Form with json_login

On the homepage, which is built in Vue, we have a login form. The goal is that, when we submit this, it will send an AJAX request with the email & password to and endpoint that will validate it.

The form itself is built over here in `assets/vue/LoginForm.vue`:

assets/vue/LoginForm.vue

```
<template>
    <form
        v-on:submit.prevent="handleSubmit"
        class="book shadow-md rounded px-8 pt-6 pb-8 mb-4 sm:w-1/2 md:w-1/3"
    >
// ... lines 6 - 45
    </form>
</template>

<script setup>

import { ref } from 'vue';
// ... lines 52 - 95
</script>
```

If you're not familiar with Vue, don't worry. We *will* do some light coding in it, but I'm mostly using it as an example to make some API requests.

Down near the bottom, on submit, we make a POST request to `/login` sending the `email` and `password` as JSON. So our *first* goal is to create *this* endpoint:

```
assets/vue/LoginForm.vue
// ... lines 1 - 48
<script setup>
// ... lines 50 - 65
const handleSubmit = async () => {
// ... lines 67 - 69
    const response = await fetch('/login', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            email: email.value,
            password: password.value
        })
    });
// ... lines 80 - 93
}

</script>
```

Creating the Login Controller

Fortunately, Symfony has a built-in mechanism *just* for this. To start, even though it won't do much, we need a new controller! In `src/Controller/`, create a new PHP class. Let's call it `SecurityController`. This will look very traditional: extend `AbstractController`, then add a `public function login()` that will return a `Response`, the one from `HttpFoundation`:

```
src/Controller/SecurityController.php
// ... lines 1 - 2
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
// ... lines 7 - 8
class SecurityController extends AbstractController
{
// ... line 11
    public function login(): Response
    {

    }
}
```

Above, give this a `Route` with a URL of `/login` to match what our JavaScript is sending to. Name the route `app_login`. Oh, and we don't really need to do this, but we can also add `methods: ['POST']`:

```
src/Controller/SecurityController.php
// ... lines 1 - 6
use Symfony\Component\Routing\Annotation\Route;

class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login', methods: ['POST'])]
    public function login(): Response
    {
// ... line 14
    }
}
```

There won't be a `/login` *page* on our site that we make a GET request to: we're only going to POST to this URL.

Returning the Current User Id

As you'll see in a minute, we're not going to *process* the `email` and `password` in this controller... but this *will* be executed after a successful login. So... what *should* we return after a successful login? I don't know! And honestly it mostly depends on what would be useful in our JavaScript. I haven't thought about it much yet, but maybe... the user id? Let's start there.

If authentication *was* successful, then, at this point, the user will be logged in like normal. To get the currently-authenticated user, I'm going to leverage a newer feature of Symfony. Add an argument with a PHP attribute called `#[CurrentUser]`. Then we can use the normal `User` type-hint, call it `$user` and default it to `null`, in case we're not logged in for some reason:

```
src/Controller/SecurityController.php
// ... lines 1 - 7
use Symfony\Component\Security\Http\Attribute\CurrentUser;

class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login', methods: ['POST'])]
    public function login(#[CurrentUser] $user = null): Response
    {
// ... lines 15 - 17
    }
}
```

We'll talk about how that's possible in a minute.

Then, return `$this->json()` with a `user` key set to `$user->getId()`:

```
src/Controller/SecurityController.php
// ... lines 1 - 9
class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login', methods: ['POST'])]
    public function login(#[CurrentUser] $user = null): Response
    {
        return $this->json([
            'user' => $user ? $user->getId() : null,
        ]);
    }
}
```

Cool! And that's *all* we need our controller to do.

Activating json_login

To activate the system that will do the *real* work of reading the email & password, head to `config/packages/security.yaml`. Under the firewall, add `json_login` and below that `check_path`... which should be set to the name of the *route* that we just created. So, `app_login`:

```
config/packages/security.yaml
security:
// ... lines 2 - 11
    firewalls:
// ... lines 13 - 15
        main:
// ... lines 17 - 18
            json_login:
                check_path: app_login
// ... lines 21 - 46
```

This activates a security listener: it's a bit of code that will now be watching *every* request to see if it is a POST request to this route. So, a POST to `/login`. If it *is*, it will decode the JSON on that request, read the `email` and `password` keys *off* of that JSON, validate the password and log us in.

Though, we *do* need to tell it what *keys* in the JSON we're using. Our JavaScript is sending `email` and `password`: super creative. So below this, set `username_path` to `email` and `password_path` to `password`:

```
config/packages/security.yaml
security:
// ... lines 2 - 11
    firewalls:
// ... lines 13 - 15
        main:
// ... lines 17 - 18
            json_login:
                check_path: app_login
                username_path: email
                password_path: password
// ... lines 23 - 48
```

The User Provider

Done! But wait! If we POST an `email` and `password` to this endpoint... how the heck does the system know how to *find* that user? How is it supposed to know that it should query the `user` table `WHERE email =` the email from the request?

Excellent question! In episode 1, we ran:

```
php ./bin/console make:user
```

This created a `User` entity with the basic security stuff that we need:

```php
// src/Entity/User.php
// ... lines 1 - 38
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 41 - 43
    private ?int $id = null;
// ... lines 45 - 49
    private ?string $email = null;
// ... lines 51 - 52
    private array $roles = [];
// ... lines 54 - 59
    private ?string $password = null;
// ... lines 61 - 64
    private ?string $username = null;
// ... lines 66 - 187
}
```

In `security.yaml`, it *also* created a user provider:

```yaml
# config/packages/security.yaml
security:
// ... lines 2 - 4
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
// ... lines 12 - 48
```

This is an entity provider: it tells the security system to find users in the database by querying by the `email` property. This means our system will decode the JSON, fetch the `email` key, query for a `User` with a matching email, then validate the password. In other words... we're ready!

Looking back at `LoginForm.vue`, the JavaScript is *also* ready: `handleSubmit()` will be called when we submit the form... and it makes the AJAX call:

assets/vue/LoginForm.vue

```
// ... lines 1 - 48
<script setup>
// ... lines 50 - 65
const handleSubmit = async () => {
    isLoading.value = true;
    error.value = '';

    const response = await fetch('/login', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            email: email.value,
            password: password.value
        })
    });

    isLoading.value = false;

    if (!response.ok) {
        const data = await response.json();
        console.log(data);
        // TODO: set error

        return;
    }

    email.value = '';
    password.value = '';
    //emit('user-authenticated', userIri);
}

</script>
```

So let's try this thing! Move over and refresh just to be sure. Try it with a fake email and password first. Submit and... nothing happened? Open up your browser's inspector and go to the console. Yes! You see a 401 status code and it dumped this error: invalid credentials. That's coming from right here in our JavaScript: after the request finishes, if the response is "not okay" - meaning there was a 4XX or 5XX status code - we decode the JSON and log it.

Apparently, when we fail authentication with `json_login`, it returns a small bit of JSON with "Invalid Credentials".

Next: let's turn this error into something we can *see* on the form, handle *another* error case, and then think about what to do when authentication is successful.

Chapter 4: Handling Authentication Errors

When we log in with an invalid email and password, it looks like the `json_login` system sends back some nice JSON with an `error` key set to "Invalid credentials". If we wanted to customize this, we could create a class that implements `AuthenticationFailureHandlerInterface`:

```
class AppAuthFailureHandler implements AuthenticationFailureHandlerInterface
{
    public function onAuthenticationFailure($request, $exception)
    {
        return new JsonResponse(
            ['something' => 'went wrong'],
            401
        );
    }
}
```

And then set its service ID onto the `failure_handler` option under `json_login`:

```
json_login:
    failure_handler: App\Security\AppAuthFailureHandler
```

Showing the Error on the Form

But, this is plenty good for us. So let's use it over in our `/assets/vue/LoginForm.vue`. We won't go too deeply into Vue, but I already have state called `error`, and if we *set* that, it will show up on the form:

```
assets/vue/LoginForm.vue
// ... lines 1 - 48
<script setup>
// ... lines 50 - 54
const error = ref('');
// ... lines 56 - 65
const handleSubmit = async () => {
// ... line 67
    error.value = '';
// ... lines 69 - 82
    if (!response.ok) {
        const data = await response.json();
        console.log(data);
        // TODO: set error

        return;
    }
// ... lines 90 - 93
}

</script>
```

After making the request, if the response is *not* okay, we're already decoding the JSON. Now let's say `error.value = data.error`:

```
assets/vue/LoginForm.vue
// ... lines 1 - 48
<script setup>
// ... lines 50 - 65
const handleSubmit = async () => {
// ... lines 67 - 82
    if (!response.ok) {
        const data = await response.json();
        error.value = data.error;

        return;
    }
// ... lines 89 - 92
}

</script>
```

To see if this works, make sure you have Webpack Encore running in the background so it recompiles our JavaScript. Refresh. And... you can click this little link to cheat and enter a valid email. But then type in a ridiculous password and... I love it! We see "Invalid credentials" on top with some red boxes!

json_login Requires Content-Type: application/json

So the AJAX call is working great. Though, there is *one* gotcha with the `json_login` security mechanism: it requires you to send a `Content-Type` header set to `application/json`. We *are* setting this on our Ajax call and you should to:

```
assets/vue/LoginForm.vue
// ... lines 1 - 48
<script setup>
// ... lines 50 - 65
const handleSubmit = async () => {
// ... lines 67 - 69
    const response = await fetch('/login', {
// ... line 71
        headers: {
            'Content-Type': 'application/json'
        },
// ... lines 75 - 78
    });
// ... lines 80 - 92
}

</script>
```

But... if someone forgets, we want to make sure that things don't go completely crazy.

Comment out that `Content-Type` header so we can see what happens:

```
assets/vue/LoginForm.vue
// ... lines 1 - 48
<script setup>
// ... lines 50 - 65
const handleSubmit = async () => {
// ... lines 67 - 69
    const response = await fetch('/login', {
// ... line 71
        headers: {
            //'Content-Type': 'application/json'
        },
// ... lines 75 - 78
    });
// ... lines 80 - 92
}

</script>
```

Then move over, refresh the page... type a ridiculous password and... it clears the form? Look down at the Network call. The endpoint returned a 200 status code with a `user` key set to `null`!

And... that makes sense! Because we're missing the header, the `json_login` mechanism did *nothing*. Instead, the request continued to our `SecurityController`... except that *this* time the user is *not* logged in. So, we return `user: null`... with a 200 status code.

That's a problem because it make it *look* like the Ajax call was successful. To fix this, if, for *any* reason the `json_login` mechanism was skipped... but the user *is* hitting our login endpoint, let's return a 401 status code that says:

> *"Hey! You need to log in!"*

So, if *not* `$user`, then `return $this->json()`... and this could look like anything. Let's include an `error` key explaining what probably went wrong: this matches the `error` key that `json_login` returns when the credentials fail, so our JavaScript will like this. Heck. I'll even fix my typo!

```
src/Controller/SecurityController.php
// ... lines 1 - 9
class SecurityController extends AbstractController
{
// ... line 12
    public function login(#[CurrentUser] $user = null): Response
    {
        if (!$user) {
            return $this->json([
                'error' => 'Invalid login request: check that the Content-Type header is "application/json".',
            ], 401);
        }
// ... lines 20 - 23
    }
}
```

Most importantly, for the second argument, pass a 401 for the status code.

Below, we can simplify... because now we know that there *will* be a user:

```
src/Controller/SecurityController.php
// ... lines 1 - 9
class SecurityController extends AbstractController
{
// ... line 12
    public function login(#[CurrentUser] $user = null): Response
    {
        if (!$user) {
            return $this->json([
                'error' => 'Invalid login request: check that the Content-Type header is "application/json".',
            ], 401);
        }

        return $this->json([
            'user' => $user->getId(),
        ]);
    }
}
```

Beautiful! Spin over and submit another bad password. Oh, gorgeous! The 401 status code triggers our error handling code, which displays the error on top. So awesome.

Go back to `LoginForm.vue` and put the `Content-Type` header back:

```
assets/vue/LoginForm.vue
// ... lines 1 - 48
<script setup>
// ... lines 50 - 65
const handleSubmit = async () => {
// ... lines 67 - 69
    const response = await fetch('/login', {
// ... line 71
        headers: {
            'Content-Type': 'application/json'
        },
// ... lines 75 - 78
    });
// ... lines 80 - 92
}

</script>
```

Next: let's login *successfully* and... figure out what we want to do when that happens! We're also going to talk about the session and how that authenticates our API requests.

Chapter 5: On Authentication Success

If you refresh the page and check the web debug toolbar, you can see that we're *not* logged in. Let's try using a real email and password. We can cheat by clicking the email and password links: this user exists in our `AppFixtures`, so it *should* work. And... okay... the boxes disappear! But nothing else happens. We'll improve that in a minute.

Thanks Session!

But for now, refresh the page and look at the web debug toolbar again. We're *authenticated*! Yea! *Just* by making a successful AJAX request to that login endpoint, that was enough to create the session and keep us logged in. Even better, if we started making requests to our API from JavaScript, those requests would be authenticated too. That's right! We don't need a fancy API token system where we attach a token to every request. We can just make a request and through the magic of cookies, that request will be authenticated.

Tip

In new API Platform projects, the default `config/packages/api_platform.yaml` file has configuration that makes your endpoints "stateless":

```
# config/packages/api_platform.yaml
api_platform:
    # ...
    defaults:
        stateless: true
```

If you want to be able to make API requests and rely in the session to stay authenticated, change this to: `stateless: false`.

REST and What Data to Return from our Authentication Endpoint?

So, logging in *worked*... but nothing happened on the page. What *should* we do after authentication? Once again, it doesn't really matter. If you're writing your auth system for your own JavaScript, you should do whatever is useful for your frontend. We're currently returning the `user` id. But we *could*, if we wanted, return the entire `user` object as JSON.

But there's one tiny problem with that. It's not super RESTful. This is one of those "REST purity" things. Every URL in your API, on a technical level, represents a different resource. This represents the *collection* resource, and this URL represents a *single* `User` resource. And if you have a different URL, that's understood to be a *different* resource. The *point* is that, in a perfect world, you would just return a `User` resource from a single URL instead of having five *different* endpoints to fetch a user.

If we return the `User` JSON from this endpoint, we're "technically" creating a new API resource. In fact, *anything* we return from this endpoint, from a REST point of view, becomes a new resource in our API. To be honest, this is all technical semantics and you should feel free to do whatever you want. But, I *do* have a fun suggestion.

Returning the IRI

To try be helpful to our frontend *and* somewhat RESTful, I have another idea. What if we return *nothing* from the endpoint.... but sneak the user's IRI onto the `Location` header of the response. Then, our frontend could use that to know *who* just logged in.

Let me show you. First, instead of returning the User ID, we're going to return the IRI, which will look something like `'/api/users/'.$user->getId()`. But I don't want to hard code that because we could potentially change the URL in the future. I'd rather have API Platform *generate* that *for* me.

And fortunately, API Platform gives us an autowireable service to do that! Before the optional argument, add a new argument type-hinted with `IriConverterInterface` and call it `$iriConverter`:

```
src/Controller/SecurityController.php
// ... lines 1 - 4
use ApiPlatform\Api\IriConverterInterface;
// ... lines 6 - 10
class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login', methods: ['POST'])]
    public function login(IriConverterInterface $iriConverter, #[CurrentUser] $user = null): Response
    {
// ... lines 16 - 24
    }
}
```

Then, down here, `return new Response()` (the one from `HttpFoundation`) with *no* content and a `204` status code:

```
src/Controller/SecurityController.php
// ... lines 1 - 10
class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login', methods: ['POST'])]
    public function login(IriConverterInterface $iriConverter, #[CurrentUser] $user = null): Response
    {
// ... lines 16 - 21
        return new Response(null, 204, [
// ... line 23
        ]);
    }
}
```

The `204` means it was "successful... but there's no content to return". We'll also pass a `Location` header set to `$iriConverter->getIriFromResource()`:

```
src/Controller/SecurityController.php
// ... lines 1 - 10
class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login', methods: ['POST'])]
    public function login(IriConverterInterface $iriConverter, #
[CurrentUser] $user = null): Response
    {
// ... lines 16 - 21
        return new Response(null, 204, [
            'Location' => $iriConverter->getIriFromResource($user),
        ]);
    }
}
```

So you can get the resource from an IRI or the IRI string from the resource, the resource being your object. Pass this `$user`.

Using the IRI in JavaScript

How nice is that? Now that we're returning this how can we use this in JavaScript? Ideally, after we log in, we would automatically show some user info over on the right. This area is built by another Vue file called `TreasureConnectApp.vue`:

```
assets/vue/controllers/TreasureConnectApp.vue
<template>
    <div class="purple flex flex-col min-h-screen">
// ... lines 3 - 5
        <div class="flex-auto flex flex-col sm:flex-row justify-center px-
8">
            <LoginForm
                v-on:user-authenticated="onUserAuthenticated"></LoginForm>
            <div
                class="book shadow-md rounded sm:ml-3 px-8 pt-8 pb-8 mb-4
sm:w-1/2 md:w-1/3 text-center">
                <div v-if="user">
                    Authenticated as: <strong>{{ user.username }}</strong>

                    | <a href="/logout" class="underline">Log out</a>
                </div>
                <div v-else>Not authenticated</div>
// ... lines 17 - 20
            </div>
        </div>
// ... line 23
    </div>
</template>

<script setup>
import { ref } from 'vue';
import LoginForm from '../LoginForm';
import coinLogoPath from '../../images/coinLogo.png';
import goldPilePath from '../../images/GoldPile.png';

defineProps(['entrypoint']);
const user = ref(null);

const onUserAuthenticated = async (userUri) => {
    const response = await fetch(userUri);
    user.value = await response.json();
}
</script>
```

I won't go into the details, but as long as that component has user data, it will print it out here. *And* `LoginForm.vue` is *already* set up to *pass* that user data to `TreasureConnectApp.vue`. Down at the bottom, after a successful authentication, *this* is where we clear the `email` and `password` state, which empties the boxes after we log in. If we emit an event called `user-authenticated` and pass it the `userIri`, `TreasureConnectApp.vue` is *already* set up to *listen* to this event. It will then make an AJAX request to `userIri`, get the JSON back, and populate its own data.

If you're not comfortable with Vue, that's ok. The *point* is that all we need to do is grab the IRI string from the `Location` header, emit this event, and everything should work.

To *read* the header, say `const userIri = response.headers.get('Location')`. I'll also uncomment this so we can `emit` it:

```
assets/vue/LoginForm.vue
// ... lines 1 - 48
<script setup>
// ... lines 50 - 65
const handleSubmit = async () => {
// ... lines 67 - 89
    email.value = '';
    password.value = '';
    const userIri = response.headers.get('Location');
    emit('user-authenticated', userIri);
}

</script>
```

This should be good! Move over and refresh. The first thing I want you to notice is that we're still logged in, but our Vue app doesn't *know* that we're logged in. We're going to fix that in a minute. Log in again using our valid email and password. And... *beautiful*! We made the `POST` request, it returned the IRI and then our JavaScript made a *second* request *to* that IRI to fetch the user data, which it displayed here.

Next: Let's talk about what it means to log *out* of an API. Then, I'll show you a simple way of telling your JavaScript *who* is logged in on page load. Because, right now, even though we are logged in, as soon as I refresh, our JavaScript thinks we're not. Lame.

Chapter 6: Logout & Passing API Data to JavaScript

What does it mean to "log out" of something? Like logging out of an API? Well, it's two things. First, it means invalidating whatever your token is, if possible. For example, if you have an API token, you would say to the API:

> *“Make this API token no longer valid.”*

In the case of session authentication, it's basically the same: it means removing the session from the session storage.

The second part of "logging out" is making whoever is using the token "forget" it. If you had an API token in JavaScript, you would *remove* it from JavaScript. For session authentication, it means deleting the cookie.

Adding the Ability to Log Out

Anyways, let's add the ability to log out of our session-based authentication. Back over in `SecurityController`, like before, we need a route and controller, even though this controller will never be called. I'll name the method `logout()` and we're going to return `void`. You'll see why in a second. Give this a `Route` of `/logout` and `name: app_logout`:

src/Controller/SecurityController.php

```php
// ... lines 1 - 10
class SecurityController extends AbstractController
{
// ... lines 13 - 26
    #[Route('/logout', name: 'app_logout')]
    public function logout(): void
    {
// ... line 30
    }
}
```

The reason I chose `void` is because we're going to throw an exception from inside the method. We've created this *entirely* because we need a route: Symfony's security system will intercept things before the controller is called:

```php
// src/Controller/SecurityController.php
// ... lines 1 - 10
class SecurityController extends AbstractController
{
// ... lines 13 - 26
    #[Route('/logout', name: 'app_logout')]
    public function logout(): void
    {
        throw new \Exception('This should never be reached!');
    }
}
```

To activate that magic, in `security.yaml`, add a key called `logout` with `path` below set to that new route name: `app_logout`:

```yaml
# config/packages/security.yaml
security:
// ... lines 2 - 11
    firewalls:
// ... lines 13 - 15
        main:
// ... lines 17 - 22
            logout:
                path: app_logout
// ... lines 25 - 50
```

This activates a listener that's now watching for requests to `/logout`. When there *is* a request to `/logout`, it will log the user out and redirect them.

All right, over here, our Vue app thinks we're not logged in, but we *are*: we can see it in the web debug toolbar. And if we manually go to `/logout`... boom! We *are* now logged out for real.

Getting the Current User Data in JavaScript

So we saw a moment ago that even when we *are* logged in and refresh, our Vue app has no *idea* that we're logged in. How could we fix that? One idea would be to create a `/me` API endpoint. Then, on load, our Vue app could make an AJAX request to that endpoint... which would either return `null` or the current user info. But, `/me` endpoints are super *not* RESTful. And there's a better way: dump the user information into JavaScript on page load.

Setting a Global user JavaScript Variable

There are two different ways to do this. The first is by setting a global variable. For example, in `templates/base.html.twig`, it doesn't really matter where, but inside the body, add a `script` tag. And here say `window.user =` and then `{{ app.user|serialize }}`. Serialize into `jsonld` and add a `|raw` so that it doesn't escape the output: we want raw JSON:

```
templates/base.html.twig
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body>
        <script>
            window.user = {{ app.user|serialize('jsonld')|raw }};
        </script>

        {% block body %}{% endblock %}
    </body>
</html>
```

How cool is that? In a minute, we'll read that from our JavaScript. If we refresh right now and look at the source, yea! We see `window.user = null`. And then when we log in and refresh the page, check it out: `window.user =` and a huge amount of data!

Serializing to JSON-LD in Twig

But there's something mysterious going on: it has the correct fields! Look closely, it has `email`, `username` and then `dragonTreasures`, which is what all this stuff is. It also, correctly, does *not* have `roles` or `password`.

So it seems that it's correctly reading our normalization groups! But how is that even possible? We're just saying "serialize this user to `jsonld`". This has nothing to do with API Platform and it's not being processed by API platform. But... our normalization groups are *configured* in API Platform. So how did the serializer know to use those?

The answer to that, as best I can tell, is that it's working... partially by accident. During serialization, API Platform sees that we're serializing an "API resource" and so it looks up the metadata for this class.

That's cool... but it's actually *not* perfect... and I like to be explicit anyway. Pass a 2nd argument to serialize, which is the context and set `groups` to `user:read`:

```
templates/base.html.twig
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body>
        <script>
            window.user = {{ app.user|serialize('jsonld', {
                'groups': ['user:read']
            })|raw }};
        </script>
// ... lines 22 - 23
    </body>
</html>
```

Now, watch what happens when we refresh. Like before, the correct properties on `User` will be exposed. But keep an eye on the embedded `dragonTreasures` property. Woh, it changed! That was actually *wrong* before: it was including *everything*, not just the stuff inside the `user:read` group.

Reading the Dynamic Data from Vue

Ok, let's go use this global variable over in JavaScript: in `TreasureConnectApp.vue`. Right now, the `user` data always starts as `null`. We can change that to `window.user`:

```
assets/vue/controllers/TreasureConnectApp.vue
// ... lines 1 - 26
<script setup>
// ... lines 28 - 32
defineProps(['entrypoint']);
const user = ref(window.user);
// ... lines 35 - 39
</script>
```

When we refresh... got it!

Next: if you're using Stimulus, an even better way to pass data to JavaScript is to use Stimulus values.

Chapter 7: Passing Values to Stimulus

Setting a global variable is fine. But if you're using Stimulus, there's a better way. We can pass server data as a *value* to a Stimulus controller.

Of course, this is a Vue app. But if you look in `templates/main/homepage.html.twig`, we're using the `symfony/ux-vue` package to render this:

```
templates/main/homepage.html.twig
// ... lines 1 - 2
{% block body %}
    <div {{ vue_component('TreasureConnectApp', {
        entrypoint: path('api_entrypoint')
    }) }}></div>
{% endblock %}
```

Behind the scenes, that activates a small Stimulus controller that starts & renders the Vue component. Any arguments that we pass here are sent to the Stimulus controller as a value... and then forwarded as props to the Vue app. So what we're going to do is "kind of" specific to Vue, but you could use this strategy to pass values to any Stimulus controller.

First in the Vue component, let's allow a new prop to be passed in called `user`:

```
assets/vue/controllers/TreasureConnectApp.vue
// ... lines 1 - 26
<script setup>
// ... lines 28 - 32
const props = defineProps(['entrypoint', 'user'])
// ... lines 34 - 40
</script>
```

If you're not using Vue, don't worry too much about the specifics. To make sure that's getting here `console.log(props.user)`. And initialize the *data* to `props.user`:

```
assets/vue/controllers/TreasureConnectApp.vue
// ... lines 1 - 26
<script setup>
// ... lines 28 - 32
const props = defineProps(['entrypoint', 'user'])
console.log(props.user);
const user = ref(props.user);
// ... lines 36 - 40
</script>
```

Next, over in `base.html.twig`, remove all that fancy `window.user` stuff:

```
templates/base.html.twig
<!DOCTYPE html>
<html>
// ... lines 3 - 15
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

And in `homepage.html.twig`, pass a new `user` prop set to `app.user`:

```
templates/main/homepage.html.twig
// ... lines 1 - 2
{% block body %}
    <div {{ vue_component('TreasureConnectApp', {
        entrypoint: path('api_entrypoint'),
        user: app.user
    }) }}></div>
{% endblock %}
```

Now if you move over and refresh, that's doesn't work? It looks like we're authenticated as... nothing?

Serializing Before Passing in the Value

If you dig a little, you'll see that we're sending the `user` to Stimulus as empty `{}`. Why? Because when you send data into Stimulus, it doesn't use the serializer to transform into JSON: it just uses `json_encode()`. And that's not good enough.

So, we need to serialize this ourselves. To do that, open `src/Controller/MainController.php`. Here's the controller that renders that template.

Autowire a service called `NormalizerInterface` and then pass a variable into our template called `userData` set to `$normalizer->normalize()`. Oh, but we need the user! Add another argument to the controller with the fancy new `#[CurrentUser]` attribute, type-hint `User`, say `$user`, and then = `null` in case we're not authenticated. Back down below, normalization will turn the object into an array. So pass `$user` and then the format for the array, which is `jsonld`: we want all the JSON-LD fields. Finally pass the serialization context with `'groups' => 'user:read'`:

```
src/Controller/MainController.php
// ... lines 1 - 4
use App\Entity\User;
// ... lines 6 - 8
use Symfony\Component\Security\Http\Attribute\CurrentUser;
use Symfony\Component\Serializer\Normalizer\NormalizerInterface;

class MainController extends AbstractController
{
    #[Route('/')]
    public function homepage(NormalizerInterface $normalizer, #[CurrentUser] User $user = null): Response
    {
        return $this->render('main/homepage.html.twig', [
            'userData' => $normalizer->normalize($user, 'jsonld', [
                'groups' => ['user:read'],
            ]),
        ]);
    }
}
```

Last step! In the template, set that `user` prop to `userData`:

```
templates/main/homepage.html.twig
// ... lines 1 - 2
{% block body %}
    <div {{ vue_component('TreasureConnectApp', {
// ... line 5
        user: userData,
    }) }}></div>
{% endblock %}
```

Since the Stimulus system *will* run that array through `json_encode()` that will transform that array into JSON. When we move over and refresh.... got it! You can see the entire JSON being passed into the Stimulus controller... and then that's passed to Vue as a prop.

Spin back over and make sure to get that `console.log()` out of there:

```
assets/vue/controllers/TreasureConnectApp.vue
// ... lines 1 - 26
<script setup>
// ... lines 28 - 33
console.log(props.user);
// ... lines 35 - 40
</script>
```

CSRF Protection

We haven't actually seen it yet, but when we start making requests to our API, those requests *will* be authenticated thanks to the session. When using sessions with your API, you might read about needing CSRF protection. Do we need CSRF tokens?

The quick answer is: probably not. As long as you use something called SameSite cookies - which are automatic in Symfony - then your API probably doesn't need to worry about CSRF protection. But be aware of two things. First, make sure that your GET requests don't have any side effects. Don't do something silly like allow the API client to make a GET request... but then you save something to the database. Second, some older browsers - like IE 11 - don't support SameSite cookies. So by forgoing CSRF tokens, you could be allowing a small percentage of your users to be susceptible to CSRF attacks.

If you want to learn more, our API Platform 2 tutorial has a whole chapter on SameSite cookies and CSRF tokens.

Next, let's turn to the other authentication use-case: API tokens.

Chapter 8: Token Types & The ApiToken Entity

Okay, so what if you need to allow programmatic access to your API?

The Types of Access Tokens

When you talk to an API via code, you send an API token, commonly known as an access token:

```
fetch('/api/kittens', {
    headers: {
        'Authorization': 'Bearer THE-ACCESS-TOKEN',
    }
});
```

Exactly how you *get* that token will vary. But there are two main cases.

First, as a user on the site, like a dragon, you want to generate an API token so that you can personally use it in a script you're writing. This is like a GitHub personal access token. These are literally created via a web interface. We're going to show this.

The second main use case is when a third party wants to make requests to your API on *behalf* of a user of your system. Like some new site called `DragonTreasureOrganizer.com` wants to be able to make an API request to *our* API on behalf of some of *our* users - like it will fetch the treasure's for a user and display them artfully on their site. In this situation, instead of our users generating tokens manually and then... like... entering them into that site, you'll offer *OAuth*. OAuth is basically a mechanism for normal users to securely give access tokens for their account to a third party. And so, your site, or somewhere in your infrastructure you'll have an OAuth server.

That's beyond the scope of this tutorial. But the important thing is that after OAuth is done, the API client wll end up with, you guessed it, an API token! So no matter which journey you're in, if you're doing programmatic access, your API users will end up with an access token. And then *your* job will be to read and understand that. We'll do *exactly* that.

JWT vs Database Storage?

So as I mentioned, we're going to show a system where we allow users to generate their own access tokens. So how do we do that? Again, there are two main ways. Death by choices!

The first is to generate something called a *JSON Web Token* or JWT. The cool thing about JWTs are that no database storage is needed. They're special strings that actually *contain* info inside of them. For example, you could create a JWT string that includes the user id and some scopes.

One *downside* of JWTs are that there's no easy way to "log out"... because there's no out-of-the-box way to invalidate JWTs. You give them an expiration when you create them... but then they're valid until then... no matter what, unless you add some extra complexity... which kinda defeats the purpose.

JWT's are trendy, popular and fun! But... you may not need them. They're awesome when you have a single sign-on system because, if that JWT is used to authenticate with multiple systems or APIs, each API can validate the JWT all on their own: without needing to make an API request to a central authentication system.

So you might end up using JWTs and there's a great bundle for them called LexikJWTAuthenticationBundle. JWT's are also the type of access token that OpenID gives you in the end.

Instead of JWTs, the second main option is dead simple: generate a random token string and store it in the database. This also allows you to invalidate access tokens by... just deleting them! This is what we'll do.

Generating the Entity

So let's get to work. To store API tokens, we need a new entity! Find your terminal and run:

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

And let's call it `ApiToken`. Say no to making this an API resource. In theory, you *could* allow users to authenticate via a login form or HTTP basic and then send a POST request to create

API tokens if you want to... but we won't.

Add an `ownedBy` property. This is going to be a `ManyToOne` to `User` and not `nullable`. And I'll say "yes" to the inverse. So the idea is that every `User` can have many API tokens. When an API token is used, we want to know which `User` it's related to. We'll use that during authentication. Calling the property `apiTokens` is fine and say no to orphan removal. Next property: `expiresAt`, make that a `datetime_immutable` and I'll say yes to `nullable`. Maybe we allow tokens to *never* expire by leaving this field blank. Next up is `token`, which will be a string. I'm going to set the length to `68` - we'll see why in a minute - not `nullable`. And finally, add a `scopes` property as a `json` type. This is going to be kind of cool: we'll store an array of "permissions" that this API token should have. Say, not `nullable` on that one as well. Hit enter to finish.

All right, spin over to your editor. No surprises: that created an `ApiToken` entity... and there's nothing very interesting inside of it:

```php
// src/Entity/ApiToken.php
// ... lines 1 - 2
namespace App\Entity;

use App\Repository\ApiTokenRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: ApiTokenRepository::class)]
class ApiToken
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\ManyToOne(inversedBy: 'apiTokens')]
    #[ORM\JoinColumn(nullable: false)]
    private ?User $ownedBy = null;

    #[ORM\Column(nullable: true)]
    private ?\DateTimeImmutable $expiresAt = null;

    #[ORM\Column(length: 68)]
    private string $token = null;

    #[ORM\Column]
    private array $scopes = [];
// ... lines 28 - 80
}
```

So let's go over and make the migration for it:

```
symfony console make:migration
```

Spin over and peek at that file to make sure it looks good. Yup! It creates the `api_token` table:

migrations/Version20230209183006.php

```php
// ... lines 1 - 12
final class Version20230209183006 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE SEQUENCE api_token_id_seq INCREMENT BY 1 MINVALUE 1 START 1');
        $this->addSql('CREATE TABLE api_token (id INT NOT NULL, owned_by_id INT NOT NULL, expires_at TIMESTAMP(0) WITHOUT TIME ZONE DEFAULT NULL, token VARCHAR(68) NOT NULL, scopes JSON NOT NULL, PRIMARY KEY(id))');
        $this->addSql('CREATE INDEX IDX_7BA2F5EB5E70BCD7 ON api_token (owned_by_id)');
        $this->addSql('COMMENT ON COLUMN api_token.expires_at IS \'(DC2Type:datetime_immutable)\'');
        $this->addSql('ALTER TABLE api_token ADD CONSTRAINT FK_7BA2F5EB5E70BCD7 FOREIGN KEY (owned_by_id) REFERENCES "user" (id) NOT DEFERRABLE INITIALLY IMMEDIATE');
    }

    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE SCHEMA public');
        $this->addSql('DROP SEQUENCE api_token_id_seq CASCADE');
        $this->addSql('ALTER TABLE api_token DROP CONSTRAINT FK_7BA2F5EB5E70BCD7');
        $this->addSql('DROP TABLE api_token');
    }
}
```

Run that with:

```
symfony console doctrine:migrations:migrate
```

And... awesome! Next: let's add a way to generate the random token string. Then, we'll talk about scopes and load up our fixtures with some API tokens.

Chapter 9: Generating the API Token & Fixtures

The most important property on `ApiToken` is the token string... which needs to be something random. Create a construct method with a `string $tokenType` argument:

```php
// src/Entity/ApiToken.php
// ... lines 1 - 8
class ApiToken
{
// ... lines 11 - 30
    public function __construct(string $tokenType = self::PERSONAL_ACCESS_TOKEN_PREFIX)
    {
// ... line 33
    }
// ... lines 35 - 87
}
```

This isn't mandatory, but GitHub has caught onto something neat - since they have different types of tokens, like personal access tokens and OAuth tokens - they give each token type its own prefix. It just helps figure out where each comes from.

We're only going to have one type, but we'll follow the idea. On top, to store the type prefix, add `private const PERSONAL_ACCESS_TOKEN_PREFIX = 'tcp_'`:

```php
// src/Entity/ApiToken.php
// ... lines 1 - 8
class ApiToken
{
    private const PERSONAL_ACCESS_TOKEN_PREFIX = 'tcp_';
// ... lines 12 - 87
}
```

I... just made up that prefix. Our site is called Treasure Connect... and this is a personal access token, so `tcp_`.

Below, for `string $tokenType =` default it to `self::PERSONAL_ACCESS_TOKEN_PREFIX`:

```
src/Entity/ApiToken.php
// ... lines 1 - 8
class ApiToken
{
// ... lines 11 - 30
    public function __construct(string $tokenType =
self::PERSONAL_ACCESS_TOKEN_PREFIX)
    {
// ... line 33
    }
// ... lines 35 - 87
}
```

Tip

For stronger security, avoid storing the plaintext token in the database. This is a bit more technical but you can find details at https://symfonycasts.com/api-token-hashed.

For the token itself, say `$this->token = $tokenType.` and then I'll use some code that will generate a random string that's 64 characters long:

```
src/Entity/ApiToken.php
// ... lines 1 - 8
class ApiToken
{
// ... lines 11 - 30
    public function __construct(string $tokenType =
self::PERSONAL_ACCESS_TOKEN_PREFIX)
    {
        $this->token = $tokenType.bin2hex(random_bytes(32));
    }
// ... lines 35 - 87
}
```

So that's 64 characters here plus the 4 character prefix equals 68. That's why I chose that length. And because we're setting the `$token` in the constructor, this doesn't need to `= null` or be nullable anymore. It will always be a `string`.

Setting up the Fixtures

Ok! This is set up! So let's add some API tokens to the database. At your terminal, run

```
php ./bin/console make:factory
```

so we can generate a Foundry factory for `ApiToken`. Go check out the new class: `src/Factory/ApiTokenFactory.php`. Down in `getDefaults()`:

```php
// src/Factory/ApiTokenFactory.php
// ... lines 1 - 29
final class ApiTokenFactory extends ModelFactory
{
// ... lines 32 - 46
    protected function getDefaults(): array
    {
        return [
            'ownedBy' => UserFactory::new(),
            'scopes' => [],
            'token' => self::faker()->text(64),
        ];
    }
// ... lines 55 - 69
}
```

This looks mostly fine, though we don't need to pass in the `token`. Oh, and I want to tweak the scopes:

```php
// src/Factory/ApiTokenFactory.php
// ... lines 1 - 29
final class ApiTokenFactory extends ModelFactory
{
// ... lines 32 - 46
    protected function getDefaults(): array
    {
        return [
            'ownedBy' => UserFactory::new(),
            'scopes' => [
// ... lines 52 - 53
            ],
        ];
    }
// ... lines 57 - 71
}
```

Typically, when you create an access token - whether it's a personal access token or one created through OAuth - you're able to *choose* which permissions that token will have: it does *not* automatically have *all* the permissions that a normal user would. I want to add *that* into our system as well.

Back over in `ApiToken`, at the top, after the first constant, I'll paste in a few more:

```
src/Entity/ApiToken.php
// ... lines 1 - 8
class ApiToken
{
// ... lines 11 - 12
    public const SCOPE_USER_EDIT = 'ROLE_USER_EDIT';
    public const SCOPE_TREASURE_CREATE = 'ROLE_TREASURE_CREATE';
    public const SCOPE_TREASURE_EDIT = 'ROLE_TREASURE_EDIT';
// ... lines 16 - 97
}
```

This defines three different scopes that a token can have. This isn't all the scopes we could imagine, but it's enough to make things realistic. So, when you create a token, you can choose whether that token should have permission to edit user data, or whether it can create treasures on behalf of the user or whether it can edit treasures on behalf of the user. I also added a `public const SCOPES` to describes them:

```
src/Entity/ApiToken.php
// ... lines 1 - 8
class ApiToken
{
// ... lines 11 - 16
    public const SCOPES = [
        self::SCOPE_USER_EDIT => 'Edit User',
        self::SCOPE_TREASURE_CREATE => 'Create Treasures',
        self::SCOPE_TREASURE_EDIT => 'Edit Treasures',
    ];
// ... lines 22 - 97
}
```

Back over in our `ApiTokenFactory`, let's, by default, give each `ApiToken` two of those three scopes:

```
src/Factory/ApiTokenFactory.php
// ... lines 1 - 29
final class ApiTokenFactory extends ModelFactory
{
// ... lines 32 - 46
    protected function getDefaults(): array
    {
        return [
            'ownedBy' => UserFactory::new(),
            'scopes' => [
                ApiToken::SCOPE_TREASURE_CREATE,
                ApiToken::SCOPE_USER_EDIT,
            ],
        ];
    }
// ... lines 57 - 71
}
```

Ok! `ApiTokenFactory` is ready. Last step: open `AppFixtures` so we can *create* some `ApiToken` fixtures. I want to make sure that, in our dummy data, each user has at least one or two API tokens. An easy way to do that, down here is to say `ApiTokenFactory::createMany()`. Since we have 10 users, let's create 30 tokens. Then pass that a callback function and, inside, return an override for the default data. We're going to override the `ownedBy` to be `UserFactory::random()`:

```
src/DataFixtures/AppFixtures.php
// ... lines 1 - 4
use App\Factory\ApiTokenFactory;
// ... lines 6 - 10
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
// ... lines 15 - 26
        ApiTokenFactory::createMany(30, function () {
            return [
                'ownedBy' => UserFactory::random(),
            ];
        });
    }
}
```

So this will create 30 tokens and assign them randomly to the 10, well really 11, users in the database. So on average, each user should have about three API tokens assigned to them. I'm doing this because, to keep life simple, we're *not* going to build a user interface where the user can *actually* click and create access tokens and select scopes. We're going to skip all that. Instead, since every user will already have some API tokens in the database, we can jump straight to learning how to read and *validate* those tokens.

Reload the fixtures with:

```
symfony console doctrine:fixtures:load
```

Showing the Tokens on the Frontend

And... beautiful! But since we're *not* going to build an interface for creating tokens, we at least need an easy way to *see* the tokens for a user... so we can test them in our API. When we're authenticated, we can show them right here.

This isn't a very important detail, so I'll do it real quick. Over in `User`, at the bottom, I'll paste in a function that returns an array of the valid API token strings for this user:

```php
// src/Entity/User.php
// ... lines 1 - 38
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 41 - 222
    /**
     * @return string[]
     */
    public function getValidTokenStrings(): array
    {
        return $this->getApiTokens()
            ->filter(fn (ApiToken $token) => $token->isValid())
            ->map(fn (ApiToken $token) => $token->getToken())
            ->toArray()
        ;
    }
}
```

In `ApiToken`, we also need an `isValid()` method... so I'll paste that as well:

```
src/Entity/ApiToken.php
// ... lines 1 - 8
class ApiToken
{
// ... lines 11 - 98
    public function isValid(): bool
    {
        return $this->expiresAt === null || $this->expiresAt > new
\DateTimeImmutable();
    }
}
```

You can get all of this from the code blocks on this page.

Next, open up `assets/vue/controllers/TreasureConnectApp.vue`... and add a new prop that can be passed in: `tokens`:

```
assets/vue/controllers/TreasureConnectApp.vue
// ... lines 1 - 34
<script setup>
// ... lines 36 - 40
const props = defineProps(['entrypoint', 'user', 'tokens'])
// ... lines 42 - 47
</script>
```

Thanks to that, we'll have a new `tokens` variable in the template. After the "Log Out" link, I'll paste in some code that renders those:

```
assets/vue/controllers/TreasureConnectApp.vue
<template>
    <div class="purple flex flex-col min-h-screen">
// ... lines 3 - 5
        <div class="flex-auto flex flex-col sm:flex-row justify-center px-8">
            <LoginForm
                v-on:user-authenticated="onUserAuthenticated"></LoginForm>
            <div
                class="book shadow-md rounded sm:ml-3 px-8 pt-8 pb-8 mb-4 sm:w-1/2 md:w-1/3 text-center">
                <div v-if="user">
// ... lines 12 - 13
                    | <a href="/logout" class="underline">Log out</a>
                    <br>
                    <h3 class="text-left font-semibold mt-2">Tokens</h3>
                    <div v-if="null === tokens">Refresh to see tokens...</div>
                    <dl v-else class="text-left max-w-md text-gray-900 divide-y divide-gray-200 dark:divide-gray-700">
                        <div class="flex flex-col py-3" v-for="token in tokens" :key="token">
                            <dd class="text-xs whitespace-normal break-words">{{ token }}</dd>
                        </div>
                    </dl>
                </div>
// ... lines 24 - 28
            </div>
        </div>
// ... line 31
    </div>
</template>
// ... lines 34 - 49
```

Last step: open `templates/main/homepage.html.twig`. This is where we're passing props to our Vue app. Pass a new one called `tokens` set to, if `app.user`, then `app.user.validTokenStrings`, else `null`:

```
templates/main/homepage.html.twig
// ... lines 1 - 2
{% block body %}
    <div {{ vue_component('TreasureConnectApp', {
// ... lines 5 - 6
        tokens: app.user ? app.user.validTokenStrings : null
    }) }}></div>
{% endblock %}
```

Let's try this! If we refresh, right now we are not logged in. Use our cheater links to log in. Notice that it doesn't show them immediately... we could improve our code to do that... but it's not a big deal. Refresh and... there they are! We have *two* tokens!

Next: let's write a system so that can *read* these tokens and authenticate the user instead of using session authentication.

Chapter 10: Access Token Authenticator

To authenticate with a token, an API client will send an `Authorization` header set to the word `Bearer` then the token string... which is just a standard practice:

```
$client->request('GET', '/api/treasures', [
    'headers' => [
        'Authorization' => 'Bearer TOKEN',
    ],
]);
```

Then something in our app will *read* that header, make sure the token is valid, authenticate the user and throw a big party to celebrate.

Activating access_token

Fortunately, Symfony has the perfect system *just* for this! Spin over and open up `config/packages/security.yaml`. Anywhere under your firewall add `access_token`:

config/packages/security.yaml

```yaml
security:
// ... lines 2 - 11
    firewalls:
// ... lines 13 - 15
        main:
// ... lines 17 - 24
            access_token:
// ... lines 26 - 52
```

This activates a listener that will watch every request to see if it has an `Authorization` header. If it does, it will read that and try to authenticate the user.

Though, it requires a helper class... because even though it knows where to *find* the token on the request... it has no idea what to do with! It doesn't know if it's a JWT that it should decode... or, in our case, that it can query the database for the matching record. So, to help it, add a `token_handler` option set to the id of a service we'll create: `App\Security\ApiTokenHandler`:

```
config/packages/security.yaml
security:
// ... lines 2 - 11
    firewalls:
// ... lines 13 - 15
        main:
// ... lines 17 - 24
            access_token:
                token_handler: App\Security\ApiTokenHandler
// ... lines 27 - 52
```

Stateless Firewall

By the way, if your security system only allows authentication via an API token, then you don't need session storage. In that case, you can set a `stateless: true` flag that tells the security system that when a user authenticates, not to bother storing the user info in the session. I'm going to remove that, because we *do* have a way to log in that relies on the session.

The Token Handler Class

Ok, let's go create that handler class. In the `src/` directory create a new sub-directory called `Security/` and inside of that a new PHP class called `ApiTokenHandler`. This is a beautifully simple class. Make it implement `AccessTokenHandlerInterface` and then go to "Code"->"Generate" or `Command`+`N` on a Mac and select "Implement Methods" to generate the one we need: `getUserBadgeFrom()`:

```
src/Security/ApiTokenHandler.php
// ... lines 1 - 2
namespace App\Security;

use Symfony\Component\Security\Http\AccessToken\AccessTokenHandlerInterface;
use Symfony\Component\Security\Http\Authenticator\Passport\Badge\UserBadge;

class ApiTokenHandler implements AccessTokenHandlerInterface
{
    public function getUserBadgeFrom(#[\SensitiveParameter] string $accessToken): UserBadge
    {
        // TODO: Implement getUserBadgeFrom() method.
    }
}
```

The `access_token` system knows how to *find* the token: it knows it will live on an `Authorization` header with the word `Bearer` in front of it. So it grabs that string then calls `getUserBadgeFrom()` and passes it to us. By the way this `#[\SensitiveParameter]` attribute is new feature in PHP. It's cool, but not important: it just makes sure that if an exception is thrown, this value won't be shown in the stacktrace.

Our job here is to query the database using the `$accessToken` and then return which *user* it relates to. To do that, we need the `ApiTokenRepository`! Add a construct method with a `private ApiTokenRepository $apiTokenRepository` argument:

```
src/Security/ApiTokenHandler.php
// ... lines 1 - 4
use App\Repository\ApiTokenRepository;
// ... lines 6 - 9
class ApiTokenHandler implements AccessTokenHandlerInterface
{
    public function __construct(private ApiTokenRepository $apiTokenRepository)
    {
    }
// ... lines 15 - 25
}
```

Below, say `$token = $this->apiTokenRepository` and then call `findOneBy()` passing it an array, so it will query where the `token` field equals `$accessToken`:

```
src/Security/ApiTokenHandler.php
// ... lines 1 - 9
class ApiTokenHandler implements AccessTokenHandlerInterface
{
// ... lines 12 - 15
    public function getUserBadgeFrom(#[\SensitiveParameter] string $accessToken): UserBadge
    {
        $token = $this->apiTokenRepository->findOneBy(['token' => $accessToken]);
// ... lines 19 - 24
    }
}
```

If authentication should fail for *any* reason, we need to throw a type of *security* exception. For example, if the token doesn't exist, throw a new `BadCredentialsException`: the one from Symfony components:

```
src/Security/ApiTokenHandler.php
// ... lines 1 - 5
use Symfony\Component\Security\Core\Exception\BadCredentialsException;
// ... lines 7 - 9
class ApiTokenHandler implements AccessTokenHandlerInterface
{
// ... lines 12 - 15
    public function getUserBadgeFrom(#[\SensitiveParameter] string $accessToken): UserBadge
    {
        $token = $this->apiTokenRepository->findOneBy(['token' => $accessToken]);

        if (!$token) {
            throw new BadCredentialsException();
        }
// ... lines 23 - 24
    }
}
```

That will cause authentication to fail... but we don't need to pass a message. This will return a "Bad Credentials." message to the user.

At this point, we *have* found the `ApiToken` entity. But, ultimately our security system wants to authenticate a *user*... not an "API Token". We do that by returning a `UserBadge` that, sort of, wraps the `User` object. Watch: return a `new UserBadge()`. The first argument is the "user identifier". Pass `$token->getOwnedBy()` to get the `User` and then `->getUserIdentifier()`:

```
src/Security/ApiTokenHandler.php
// ... lines 1 - 7
use Symfony\Component\Security\Http\Authenticator\Passport\Badge\UserBadge;

class ApiTokenHandler implements AccessTokenHandlerInterface
{
// ... lines 12 - 15
    public function getUserBadgeFrom(#[\SensitiveParameter] string $accessToken): UserBadge
    {
// ... lines 18 - 23
        return new UserBadge($token->getOwnedBy()->getUserIdentifier());
    }
}
```

How the User Object is Loaded

Notice that we're not *actually* returning the `User` object. That's mostly because... we don't need to! Let me explain. Hold `Command` or `Ctrl` and click `getUserIdentifier()`. What this *really* returns is the user's `email`. So we're returning a `UserBadge` with the user's `email` inside. What happens next is the same thing that happens when we send an `email` to the `json_login` authentication endpoint. Symfony's security system takes that email and, because we have this user provider, it knows to query the database for a `User` with that `email`.

So it's going to query the database *again* for the `User` via the email... which is a bit unnecessary, but fine. If you want to avoid that, you could pass a callable to the second argument and return `$token->getOwnedBy()`. But this will work fine as it is.

Oh, and it's probably a good idea to check and make sure the token is valid! If not `$token->isValid()`, then we could throw another `BadCredentialsException`. But if you want to customize the message, you can also throw a new `CustomUserMessageAuthenticationException` with "Token expired" to return *that* message to the user:

```php
// src/Security/ApiTokenHandler.php
// ... lines 1 - 6
use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationExc
// ... lines 8 - 10
class ApiTokenHandler implements AccessTokenHandlerInterface
{
// ... lines 13 - 16
    public function getUserBadgeFrom(#[\SensitiveParameter] string $accessToken): UserBadge
    {
// ... lines 19 - 24
        if (!$token->isValid()) {
            throw new CustomUserMessageAuthenticationException('Token expired');
        }

        return new UserBadge($token->getOwnedBy()->getUserIdentifier());
    }
}
```

Using the Token in Swagger?

And... done! So... how do we try this? Well, ideally we could try it in our Swagger docs. I'm going to open a new tab... then log out. But I'll keep my original tab open... so I can steal these valid tokens!

Head to the API docs. How can we tell this interface to send an API token when it makes the requests? Well you may have noticed an "Authorize" button. But when we click it... it's empty! That's because we haven't, yet, told Open API *how* users are able to authenticate. Fortunately we can do this via API Platform.

Open up `config/packages/api_platform.yaml`. And a new key called `swagger`, though we're *actually* configuring the OpenAPI docs. To add a new way of authenticating, set `api_keys` to activate that type, then `access_token`... which can be anything you want. Below this, give this authentication mechanism a name... and `type: header` because we want to pass the token as a header:

```yaml
config/packages/api_platform.yaml
api_platform:
// ... lines 2 - 7
    swagger:
        api_keys:
            access_token:
                name: Authorization
                type: header
// ... lines 13 - 18
```

This will tell Swagger - via our OpenAPI docs - that we can send API tokens via the `Authorization` header. *Now* when we click the "Authorize" button... yea! It says "Name: Authorization", "In Header".

To use this, we need to start with the word `Bearer` then a space... because it doesn't fill that in for us. More on that in a minute. Let's first try an invalid token. Hit "Authorize". That didn't actually make any requests yet: it just stored the token in JavaScript.

Let's try the get treasure collection endpoint. When we execute... awesome! A 401! We don't *need* to be authenticated to use this endpoint, but because we passed an `Authorization` header with `Bearer` and then a token, the new `access_token` system caught that, passed the string to our handler... but then we couldn't find a matching token in the database, so we threw the `BadCredentialsException`

You can see it down here: the API returned an empty response, but with a header containing `invalid_token` and `error_description`: "Invalid credentials.".

Checking the Token Authentication is Working

So the *bad* case is working. Let's try the happy case! In the other tab, copy one of the valid tokens. Then slide back up, hit "Authorize", then "Log out". Logging out just means that it "forgets" the API token we set a minute ago. Re-type `Bearer `, paste, hit "Authorize", close... and let's go down and try this endpoint again. And... woohoo! A 200!

So it *seems* like that worked... but how can we tell? Whelp, down on the web debug toolbar, click to open the profiler for that request. On the Security tab... yes! We're logged in as Bernie. Success!

The only thing I *don't* like is needing to type that `Bearer` string in the authorization box. That's not super user-friendly. So next, let's fix that by learning how we can customize the OpenAPI spec document that Swagger uses.

Chapter 11: Customizing the OpenAPI Docs

To use API tokens in Swagger, we need to type the word "Bearer" and *then* the token. Lame! Especially if we intend for this to be used by real users. So how can we fix that?

The OpenAPI Spec is the Key

Remember that Swagger is *entirely* generated from the OpenAPI spec document that API Platform builds. You can see this document either by viewing the page source - you can see it all right there - or by going to `/api/docs.json`. A few minutes ago, we added some config to API Platform called `Authorization`:

config/packages/api_platform.yaml

```yaml
api_platform:
// ... lines 2 - 7
    swagger:
        api_keys:
            access_token:
                name: Authorization
                type: header
// ... lines 13 - 18
```

The end result is that it added these security sections down here. Yup, it's that simple: this config triggered these new sections in this JSON document: nothing else. Swagger then reads that and knows to make this "Authorization" available.

So I did some digging directly on the OpenAPI site and I found out that it *does* have a way to define an authentication scheme where you do *not* need to pass the "Bearer" part manually. Unfortunately, unless I'm missing it, API Platform's config does *not* support adding that. So are we done for? No way! And for an *awesome* reason.

Creating our OpenApiFactory

To create this JSON document, internally, API Platform creates an `OpenApi` object, populates all this data onto it and then sends it through Symfony's serializer. This is important because we can *tweak* the `OpenApi` object *before* it goes through the serializer. How? The `OpenApi` object is created via a core `OpenApiFactory`... and we can *decorate* that.

Check it out: over in the `src/` directory, create a new directory called `ApiPlatform/`... and inside, a new PHP class called `OpenApiFactoryDecorator`. Make this implement `OpenApiFactoryInterface`. Then go to "Code"->"Generate" or `Command`+`N` on a Mac to implement the one method we need: `__invoke()`:

```
src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 2
namespace App\ApiPlatform;

use ApiPlatform\OpenApi\Factory\OpenApiFactoryInterface;
use ApiPlatform\OpenApi\OpenApi;

class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
    public function __invoke(array $context = []): OpenApi
    {
        // TODO: Implement __invoke() method.
    }
}
```

Hello Service Decoration!

Right now, a core `OpenApiFactory` service exists in API Platform that creates the `OpenApi` object with all this data on it. Here's our sneaky plan: we're going to tell Symfony to use *our* new class as the `OpenApiFactory` *instead of* the *core* one. But... we definitely do *not* want to re-implement *all* of the core logic. To avoid that, we'll *also* tell Symfony to pass us the original, *core* `OpenApiFactory`.

You might be familiar with what we're doing. It's *class decoration*: an object-oriented strategy for extending classes. It's *really* easy to do in Symfony and API Platform leverages it a lot.

Whenever you do decoration, you will always create a constructor that accepts the *interface* that you're decorating. So `OpenApiFactoryInterface`. I'll call this `$decorated`. Oh, and let me put `private` in front of that:

```
src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 4
use ApiPlatform\OpenApi\Factory\OpenApiFactoryInterface;
// ... lines 6 - 9
class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
    public function __construct(private OpenApiFactoryInterface
$decorated)
    {
    }
// ... lines 15 - 23
}
```

Perfect.

Down here, to start, say `$openApi = $this->decorated` and then call the `__invoke()` method passing the same argument: `$context`:

```
src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 9
class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
// ... lines 12 - 15
    public function __invoke(array $context = []): OpenApi
    {
        $openApi = $this->decorated->__invoke($context);
// ... lines 19 - 22
    }
}
```

That will call the core factory which will do *all* the hard work of creating the full `OpenApi` object. Down here, return that:

```
src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 9
class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
// ... lines 12 - 15
    public function __invoke(array $context = []): OpenApi
    {
        $openApi = $this->decorated->__invoke($context);
// ... lines 19 - 21
        return $openApi;
    }
}
```

And in between? Yup, that's where *we* can *mess* with things! To make sure this is working, for now, just dump the `$openApi` object:

```
src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 9
class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
// ... lines 12 - 15
    public function __invoke(array $context = []): OpenApi
    {
        $openApi = $this->decorated->__invoke($context);

        dump($openApi);

        return $openApi;
    }
}
```

The #[AsDecorator] Attribute

At this moment, from an object-oriented point of view, this class *is* set up correctly for decoration. But Symfony's container is still set up to use the *normal* `OpenApiFactory`: it's not going to use *our* new service at all. We somehow need to tell the container that, first, the core `OpenApiFactory` service should be *replaced* by *our* service, and second, that the *original* core service should be passed *to* us.

How can we do that? Above the class, add an attribute called `#[AsDecorator]` and hit tab to add that `use` statement. Pass this the service id of the original, core `OpenApiFactory`. You can do some digging to find this or usually the documentation will tell you. API platform actually *documents* decorating this service, so right in their docs, you'll find that the service id is `api_platform.openapi.factory`:

```
src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 6
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;

#[AsDecorator('api_platform.openapi.factory')]
class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
// ... lines 12 - 23
}
```

That's it! Thanks to this, anyone that was previously using the core `api_platform.openapi.factory` service will receive *our* service instead. But the original one will be passed to us.

So... it should be working! To test it, head to the API homepage and refresh. Yes! When this page loads, it renders the OpenAPI JSON document in the background. The dump in the web debug toolbar proves that it hit our code! And check out that beautiful `OpenApi` object: it has everything including `security`, which matches what we saw in the JSON. So now, we can tweak that!

Customizing the OpenAPI Config

The code I'll put here is a bit specific to the `OpenApi` object and the exact config that I know we need in the final Open API JSON:

```php
// src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 9
#[AsDecorator('api_platform.openapi.factory')]
class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
// ... lines 13 - 16
    public function __invoke(array $context = []): OpenApi
    {
        $openApi = $this->decorated->__invoke($context);

        $securitySchemes = $openApi->getComponents()->getSecuritySchemes() ?: new \ArrayObject();
// ... lines 22 - 26
        return $openApi;
    }
}
```

We fetch the `$securitySchemes`, and then override `access_token`. This matches the name we used in the config. Set that to a new `SecurityScheme()` object with two named arguments: `type: 'http'` and `scheme: 'bearer'`:

```
src/ApiPlatform/OpenApiFactoryDecorator.php
// ... lines 1 - 5
use ApiPlatform\OpenApi\Model\SecurityScheme;
// ... lines 7 - 9
#[AsDecorator('api_platform.openapi.factory')]
class OpenApiFactoryDecorator implements OpenApiFactoryInterface
{
// ... lines 13 - 16
    public function __invoke(array $context = []): OpenApi
    {
        $openApi = $this->decorated->__invoke($context);

        $securitySchemes = $openApi->getComponents()->getSecuritySchemes()
?: new \ArrayObject();
        $securitySchemes['access_token'] = new SecurityScheme(
            type: 'http',
            scheme: 'bearer',
        );

        return $openApi;
    }
}
```

That's it! First refresh the raw JSON document so we can see what this looks like. Let me search for "Bearer". There we go! We modified what the JSON looks like!

What does Swagger think about this new config? Refresh and hit "Authorize". Ok cool: `access_token`, `http, Bearer`. Go steal an API token... paste *without* saying `Bearer` first and hit "Authorize". Let's test the same endpoint. Whoops, I need to hit "Try it out". And... gorgeous! Look at that `Authorization` header! It passed `Bearer` *for* us. Mission accomplished.

By the way, you might think, because we're completely overriding the `access_token` config, that we could just delete it from `api_platform.yaml`. Unfortunately, for subtle reasons that have to do with how the security documentation is generated, we *do* still need this. But I'll say `# overridden in OpenApiFactoryDecorator`:

```
config/packages/api_platform.yaml
api_platform:
// ... lines 2 - 7
    swagger:
        api_keys:
            # overridden in OpenApiFactoryDecorator
            access_token:
// ... lines 12 - 19
```

This was just *one* example of how you could extend your Open API spec doc. But if you ever need to tweak something else, now you know how.

Next, let's talk about scopes.

Chapter 12: API Token Scopes

Each `ApiToken` has an array of scopes, though we're not using that yet. The idea is cool: when a token is created, you can select which permissions it has. Like maybe a token gives the permission to create new treasures but not edit existing treasures. To allow that, we're going to map the scopes of a token to *roles* in Symfony.

How are Roles Loaded Now?

Right now in `ApiTokenHandler`, we're basically returning the user... and then the system authenticates *fully* as that user. This means we get whatever roles are on that `User` object. How could we *change* that so that we authenticate as this user... but with a *different* set of roles? A set based on the scopes from the token?

We're using the `access_token` security system. Hit `Shift`+`Shift` and open a core class called `AccessTokenAuthenticator`. This is cool: it's the *actual* code behind that authentication system! For example, this is where it grabs the token off of the request and calls *our* token handler's `getUserBadgeFrom()` method.

The *roles* the user will have are *also* determined here: down inside `createToken()`. The "token" is, sort of, a "wrapper" around the `User` object in the security system. And *this* is where we pass it the roles it should have. As you can see, no matter what, the roles will be `$passport->getUser()->getRoles()`. In other words, we *always* get the roles by calling `getRoles()` on the `User` class... which just returns the `roles` property.

Setting up the Custom Roles System

So there's no great hook point. We *could* create a *custom* authenticator class and implement our *own* `createToken()` method. But that's a bummer because we would need to completely reimplement the logic form this authenticator class. So, instead we can... kind of cheat.

Start in `User`. Scroll up to the top where we have our properties. Add a new one: `private ?array` called `$accessTokenScopes` and initialize it to `null`:

```
src/Entity/User.php
// ... lines 1 - 38
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 41 - 54
    /* Scopes given during API authentication */
    private ?array $accessTokenScopes = null;
// ... lines 57 - 248
}
```

Notice that this is *not* a persisted column. It's just a place to temporarily store the scopes that the user should have. Next, down at the bottom add a new public method called `markAsTokenAuthenticated()` with an `array $scopes` argument. We're going to call this during authentication. Inside, say `$this->accessTokenScopes = $scopes`:

```
src/Entity/User.php
// ... lines 1 - 38
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 41 - 244
    public function markAsTokenAuthenticated(array $scopes)
    {
        $this->accessTokenScopes = $scopes;
    }
}
```

Here's where things get interesting. Search for the `getRoles()` method. We know that, no matter what, Symfony will call this during authentication and whatever this returns, that's the roles the user will have. *We're* going to "sneak in" our scope roles.

First if the `$accessTokenScopes` property is `null`, that means we're logging in as a *normal* user. In this case, set `$roles` to `$this->roles` so that we get *all* the `$roles` on the `User`. Then add an extra role called `ROLE_FULL_USER`:

```
src/Entity/User.php
// ... lines 1 - 38
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 41 - 113
    public function getRoles(): array
    {
        if (null === $this->accessTokenScopes) {
            // logged in via the full user mechanism
            $roles = $this->roles;
            $roles[] = 'ROLE_FULL_USER';
        } else {
// ... line 121
        }
// ... lines 123 - 127
    }
// ... lines 129 - 248
}
```

We'll talk about that in a minute.

Else, if we *did* log in via an access token, say `$roles = $this->accessTokenScopes`:

```
src/Entity/User.php
// ... lines 1 - 38
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 41 - 113
    public function getRoles(): array
    {
        if (null === $this->accessTokenScopes) {
            // logged in via the full user mechanism
            $roles = $this->roles;
            $roles[] = 'ROLE_FULL_USER';
        } else {
            $roles = $this->accessTokenScopes;
        }
// ... lines 123 - 127
    }
// ... lines 129 - 248
}
```

And, in both cases, make sure that we *always* have `ROLE_USER`:

```
src/Entity/User.php
// ... lines 1 - 38
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 41 - 113
    public function getRoles(): array
    {
        if (null === $this->accessTokenScopes) {
            // logged in via the full user mechanism
            $roles = $this->roles;
            $roles[] = 'ROLE_FULL_USER';
        } else {
            $roles = $this->accessTokenScopes;
        }

        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }
// ... lines 129 - 248
}
```

With this in place, head over to `ApiTokenHandler`. Right before we return `UserBadge`, add `$token->getOwnedBy()->markAsTokenAuthenticated()` and pass `$token->getScopes()`:

```
src/Security/ApiTokenHandler.php
// ... lines 1 - 10
class ApiTokenHandler implements AccessTokenHandlerInterface
{
// ... lines 13 - 16
    public function getUserBadgeFrom(#[\SensitiveParameter] string $accessToken): UserBadge
    {
// ... lines 19 - 28
        $token->getOwnedBy()->markAsTokenAuthenticated($token->getScopes());

        return new UserBadge($token->getOwnedBy()->getUserIdentifier());
    }
}
```

Done! Let's take it for a test drive! Back over on Swagger, it already has our API token... so we can just re-execute the request. Cool: we see the `Authorization` header. Did it authenticate with the correct scopes?

Click to open the profiler for that request... and head down to "Security". It did! Look: we're logged in as that user, but with `ROLE_USER`, `ROLE_USER_EDIT` and `ROLE_TREASURE_CREATE`: the two scopes from the token. But if we were to log in via the login form, instead of these scopes, we would have whatever roles the user *normally* has, plus `ROLE_FULL_USER`.

Giving Normal Users sudo Access with role_hierarchy

In the next chapter, we'll use these roles to protect different API operations. For example, to use the POST treasures endpoint, we'll require `ROLE_TREASURE_CREATE`. But we *also* need to make sure that if a user logs in via the login form, they can still use this operation, even though they won't have that exact role. That is where `ROLE_FULL_USER` comes in handy.

Open `config/packages/security.yaml` and, anywhere, add `role_hierarchy`... I recommend spelling it correctly. Say `ROLE_FULL_USER`. So, if you're logged in as a full user, we're going to give you all possible scopes that a token could have. Copy the three scope roles: `ROLE_USER_EDIT`, `ROLE_TREASURE_CREATE` and `ROLE_TREASURE_EDIT`:

```
config/packages/security.yaml
security:
// ... lines 2 - 12
    role_hierarchy:
        ROLE_FULL_USER: [ROLE_USER_EDIT, ROLE_TREASURE_CREATE, ROLE_TREASURE_EDIT]
// ... lines 15 - 56
```

We *do* need to be careful to make sure that if we add more scopes, we add them here too.

Thanks to this, if we protect something by requiring `ROLE_USER_EDIT`, users that are logged in via the login form *will* have access.

Ok team, we are done with authentication! Woo! Next, let's start into "authorization" by learning how to lock down operations so that only certain users can access them.

Chapter 13: Deny Access with The "security" Option

We've just talked a lot about authentication: that's the way we tell the API *who* we are. Now we turn to authorization, which is all about denying access to certain operations and other things *based* on who you are.

Using access_control

There are multiple ways to control access to something. The simplest is in `config/packages/security.yaml`. Just like normal Symfony security, down here, we have an `access_control` section:

```yaml
config/packages/security.yaml
security:
// ... lines 2 - 37
    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
// ... lines 43 - 56
```

If you want to lock down a specific URL pattern by a specific role, use `access_control`. You could use this, for example, to require that the user has a role to use *anything* in your API by targeting URLs starting with `/api`.

Hello "security" Option

In a traditional web app, I *do* use `access_control` for several things. But most of the time I put my authorization rules inside *controllers*. But... of course, with API Platform, we don't *have* controllers. All we have are API resource classes, like `DragonTreasure`. So instead of putting security rules in controllers, we'll attach them to our *operations*.

For example, let's make the POST request to create a new `DragonTreasure` require the user to be authenticated. Do that by adding a *very* handy `security` option. Set that to a string and inside, say `is_granted()`, double quotes then `ROLE_TREASURE_CREATE`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 26
#[ApiResource(
// ... lines 28 - 29
    operations: [
// ... lines 31 - 36
        new Post(
            security: 'is_granted("ROLE_TREASURE_CREATE")',
        ),
// ... lines 40 - 41
    ],
// ... lines 43 - 56
)]
// ... lines 58 - 75
class DragonTreasure
{
// ... lines 78 - 235
}
```

We *could* simply use `ROLE_USER` here if we just wanted to make sure that the user is logged in. But we have a cool system where, if you use an API token for authentication, that token will have specific scopes. One possible scope is called `SCOPE_TREASURE_CREATE`... which maps to `ROLE_TREASURE_CREATE`. So we look for *that*. Also, in `security.yaml`, via `role_hierarchy`, if you log in via the login form, you get `ROLE_FULL_USER`... and then you automatically also get `ROLE_TREASURE_CREATE`.

In other words, by using `ROLE_TREASURE_CREATE`, access will be granted either because you logged in via the login form *or* you authenticated using an API token that has that scope.

Let's try it. Make sure you're logged out. I'll refresh. Yup, you can see on the web debug toolbar that I'm *not* logged in... and Swagger does *not* currently have an API token.

Let's test the POST endpoint. Try it out.. and... just Execute with the example data. And... yes! A 401 status code with type `hydra:error`!

More about the "security" Attribute

The `security` option actually holds an *expression* using Symfony's expression language. And you can get pretty fancy with it. Though, we're going to try to keep things simple. And later, we'll learn how to offload complex rules to voters.

Let's add a few more rules. `Put` and `Patch` are both edits. These are especially interesting because, to use these, not only do we need to be logged in, we probably need to be the *owner* of this `DragonTreasure`. We don't want *other* people to edit *our* goodies.

We're going to worry about the ownership part later. But for now, let's at least add `security` with `is_granted()` then `ROLE_TREASURE_EDIT`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Put(
            security: 'is_granted("ROLE_TREASURE_EDIT")',
        ),
// ... lines 44 - 49
    ],
// ... lines 51 - 64
)]
// ... lines 66 - 83
class DragonTreasure
{
// ... lines 86 - 243
}
```

Once again, I'm using the scope role. Copy that, and duplicate it down here for `Patch`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 43
        new Patch(
            security: 'is_granted("ROLE_TREASURE_EDIT")',
        ),
// ... lines 47 - 49
    ],
// ... lines 51 - 64
)]
// ... lines 66 - 83
class DragonTreasure
{
// ... lines 86 - 243
}
```

Oh, and earlier, we removed the `Delete` operation. Let's add that back with `security` set to look for `ROLE_ADMIN`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 46
        new Delete(
            security: 'is_granted("ROLE_ADMIN")',
        ),
    ],
// ... lines 51 - 64
)]
// ... lines 66 - 83
class DragonTreasure
{
// ... lines 86 - 243
}
```

If we decided later to add a scope that allowed API tokens to delete treasures, we could add that and change this to `ROLE_TRESURE_DELETE`.

Let's make sure this works! Use the GET collection endpoint. Try that out. This operation does *not* require authentication... so it works just fine. And we have a treasure with ID 1. Close this up, open the PUT operation, hit "Try it out", 1, "Execute" and... alright! We get a 401 here too!

Adding "security" to an Entire Clas

So adding the `security` option to the individual operations is probably the most common thing to do. But you can also add it to the `ApiResource` itself to apply to the entire class. For example, on `User`, we probably want *every* operation to require authentication... except for the `Post` to create, because that's how you would register a new user.

So up here, add `security` and look for `ROLE_USER`... just to check that we're logged in:

```
src/Entity/User.php
// ... lines 1 - 20
#[ApiResource(
// ... lines 22 - 23
    security: 'is_granted("ROLE_USER")',
)]
// ... lines 26 - 40
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 43 - 250
}
```

And because this class has a sub resource... and this *also* allows us to fetch a user, be sure to add `security` here too:

```
src/Entity/User.php
// ... lines 1 - 25
#[ApiResource(
// ... lines 27 - 35
    security: 'is_granted("ROLE_USER")',
)]
// ... lines 38 - 40
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 43 - 250
}
```

Keep close track of security if you're using subresources.

Ok, so now *every* operation on `User` requires you to be logged in. But... we *don't* want that for the `Post` operation. To add flexibility, go up to the first `ApiResource`, add the `operations` option, and, real quick, list all the normal operations, `new Get()`, `new GetCollection()`, `new Post()`, `new Put()`, `new Patch()`, and `new Delete()`:

src/Entity/User.php

```php
// ... lines 1 - 25
#[ApiResource(
    // Now add `operations` set to the 6 normal operations
    operations: [
        new Get(),
        new GetCollection(),
        new Post(
// ... line 32
        ),
        new Put(
// ... line 35
        ),
        new Patch(
// ... line 38
        ),
        new Delete(),
    ],
// ... lines 42 - 44
)]
// ... lines 46 - 60
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 63 - 270
}
```

Now that we have those, we can customize them. For `Post`, since we want this to *not* require authentication, say `security:` `'is_granted()` passing a special fake role called `PUBLIC_ACCESS`:

```
src/Entity/User.php
// ... lines 1 - 25
#[ApiResource(
    // Now add `operations` set to the 6 normal operations
    operations: [
// ... lines 29 - 30
        new Post(
            security: 'is_granted("PUBLIC_ACCESS")',
        ),
// ... lines 34 - 40
    ],
// ... lines 42 - 44
)]
// ... lines 46 - 60
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 63 - 270
}
```

This will *override* the security rule that we're passing on the resource level. Oh, and while we're here, for `Put`, set `security` to look for `ROLE_USER_EDIT` since we have a scope role for editing users. Repeat that down here for `Patch`:

```
src/Entity/User.php
// ... lines 1 - 25
#[ApiResource(
    // Now add `operations` set to the 6 normal operations
    operations: [
// ... lines 29 - 33
        new Put(
            security: 'is_granted("ROLE_USER_EDIT")'
        ),
        new Patch(
            security: 'is_granted("ROLE_USER_EDIT")'
        ),
// ... line 40
    ],
// ... lines 42 - 44
)]
// ... lines 46 - 60
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 63 - 270
}
```

I love it! Refresh the whole page. We're most interested in the `POST` users endpoint. We are *not* authenticated, so hit "Try it out" and I'll leave the default data. "Execute" and... we nailed it! A 201 status. That *did* allow anonymous access.

Checking the Security Decisions

Oh, and super fun: if you ever want to see the security *decisions* that were made during a request, open the profiler for that request, go down to the "Security" section then "Access Decision". For this request, only one decision made by the security system: it was for `PUBLIC_ACCESS`, and that *was* allowed.

Next: our API is getting complex... and it's only going to get *more* complex. It's time to stop testing our endpoints manually via Swagger and start testing them with automated tests.

Chapter 14: Bootstrapping a Killer Test System

Our API is getting more and more complex. And doing *manually* testing is *not* a great long-term plan. So let's install some tools to get a killer test setup.

Installing the test-pack

Step one: at your terminal run:

```
composer require test
```

This is a flex alias for a package called `symfony/test-pack`. Remember: packs are shortcut packages that actually install a bunch of *other* packages. For example, when this finishes... and we check out `composer.json`, you can see down in `require-dev` that this added PHPUnit itself as well as a few other tools from Symfony to help testing:

```
composer.json
{
// ... lines 2 - 87
    "require-dev": {
// ... line 89
        "phpunit/phpunit": "^9.5",
        "symfony/browser-kit": "6.2.*",
        "symfony/css-selector": "6.2.*",
// ... lines 93 - 95
        "symfony/phpunit-bridge": "^6.2",
// ... lines 97 - 99
    }
}
```

It also executed a recipe which added a number of files. We have `phpunit.xml.dist`, a `tests/` directory, `.env.test` for test-specific environment variables and even a little `bin/phpunit` executable shortcut that we'll use to run our tests.

Hello browser Library

No surprise, Symfony has tools for testing and these can be used to test an API. Heck, API Platform even has their *own* tools built on *top* of those to make testing an API even easier. And yet, I'm going to be stubborn and use a totally *different* tool that I've fallen in love with.

It's called Browser, and it's *also* built on top of Symfony's testing tools: almost like a nicer interface above that strong base. It's just... super fun to use. Browser gives us a fluid interface that can be used for testing web apps, like you see here, or testing APIs. It can also can be used to test pages that use JavaScript.

Let's get this guy installed. Copy the `composer require` line, spin back over and run that:

```
composer require zenstruck/browser --dev
```

While that's doing its thing, it's optional, but there's an "extension" that you can add to `phpunit.xml.dist`. Add it down here on the bottom:

```
phpunit.xml.dist
// ... lines 1 - 3
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="vendor/phpunit/phpunit/phpunit.xsd"
         backupGlobals="false"
         colors="true"
         bootstrap="tests/bootstrap.php"
         convertDeprecationsToExceptions="false"
>
// ... lines 11 - 35
    <extensions>
        <extension class="Zenstruck\Browser\Test\BrowserExtension" />
    </extensions>
// ... lines 39 - 45
</phpunit>
```

In the future, if you're using PHPUnit 10, this will likely be replaced by some `listener` config.

This adds a few extra features to browser. Like, when a test fails, it will automatically save the last response to a file. We'll see this soon. And if you're using JavaScript testing, it'll take screenshots of failures!

Creating our First Test

Ok, we're ready for our first test. In the `tests/` directory, it doesn't matter how you organize things, but I'm going to create a `Functional/` directory because we're going to be making functional tests to our API. Yup, we'll literally create an API client, make GET or POST requests and then assert that we get back the correct output.

Create a new class called `DragonTreasureResourceTest`. A normal test extends `TestCase` from PHPUnit. But make this extend `KernelTestCase`: a class from Symfony that extends `TestCase`... but gives us access to Symfony's engine:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 2
namespace App\Tests\Functional;

use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

class DragonTreasureResourceTest extends KernelTestCase
{

}
```

Let's start by testing the GET collection endpoint to make sure we get back the data we expect. To activate the browser library, at the top, add a trait with `use HasBrowser`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 5
use Zenstruck\Browser\Test\HasBrowser;

class DragonTreasureResourceTest extends KernelTestCase
{
    use HasBrowser;
// ... lines 11 - 18
}
```

Next, add a new test method: `public function`, how about `testGetCollectionOfTreasures()`... which will return `void`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 7
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 10 - 11
    public function testGetCollectionOfTreasures(): void
    {
// ... lines 14 - 17
    }
}
```

Using browser is dead simple thanks to that trait: `$this->browser()`. Now we can make GET, POST, PATCH or whatever request we want. Make a GET request to `/api/treasures` and then, just to see what that looks like, use this nifty `->dump()` function:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 7
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 10 - 11
    public function testGetCollectionOfTreasures(): void
    {
        $this->browser()
            ->get('/api/treasures')
            ->dump()
        ;
    }
}
```

Running our Tests through the symfony Binary

How cool is that? Let's see what it looks like. To execute our test, we could run:

```
php ./vendor/bin/phpunit
```

That works just fine. But one of the recipes also added a shortcut file:

```
php bin/phpunit
```

When we run that, ooh, let's see. The `dump()` *did* happen: it dumped out the response... which was some sort of error. It says:

> *"SQLSTATE: connection to server port 5432 failed."*

Hmm, it can't connect to our database. Our database is running via a Docker container... and then, because we're using the `symfony` web server, when we use the site via a browser, the `symfony` web server detects the Docker container and sets the `DATABASE_URL` environment variable *for* us. That's how our API has been able to talk to the Docker database.

When we've run *commands* that need to talk to the database, we've been running them like `symfony console make:migration`... because when we execute things through `symfony`, it adds the `DATABASE_URL` environment variable... and *then* runs the command.

So, when we simply run `php bin/phpunit`... the real `DATABASE_URL` is missing. To fix that, run:

```
symfony php bin/phpunit
```

It's the same thing... except it lets `symfony` add the `DATABASE_URL` environment variable. And now... we see the dump again! Scroll to the top. Better! Now the error says:

> *"Database `app_test` does not exist."*

Test-Specific Database

Interesting. To understand what's happening, open `config/packages/doctrine.yaml`. Scroll down to a `when@test` section. This is cool: when we're in the `test` environment, there's a bit of config called `dbname_suffix`. Thanks to this, Doctrine will take our *normal* database name and add `_test` to it:

```yaml
config/packages/doctrine.yaml
// ... lines 1 - 18
when@test:
    doctrine:
        dbal:
            # "TEST_TOKEN" is typically set by ParaTest
            dbname_suffix: '_test%env(default::TEST_TOKEN)%'
// ... lines 24 - 44
```

This next part is specific to a library called ParaTest where you can run tests in parallel. Since we're not using that, it's just an empty string and not something we need to worry about.

Anyway, that's how we end up with an `_test` at the end of our database name. And we want that! We don't want our `dev` and `test` environments to use the same database because it gets annoying when they run over each other's data.

By the way, if you're *not* using the `symfony` Binary and Docker setup... and you're configuring your database manually, be aware that in the `test` environment, the `.env.local` file is *not* read:

```
.env.test
# define your env variables for the test env here
KERNEL_CLASS='App\Kernel'
APP_SECRET='$ecretf0rt3st'
SYMFONY_DEPRECATIONS_HELPER=999999
PANTHER_APP_ENV=panther
PANTHER_ERROR_SCREENSHOT_DIR=./var/error-screenshots
```

The `test` environment is special: it skips reading `.env.local` and only reads `.env.test`. You can also create a `.env.test.local` for env vars that are read in the `test` environment but that won't be committed to your repository.

The ResetDatabaseTrait

Ok, in the `test` environment, we're missing the database. We could easily fix this by running:

```
symfony console doctrine:database:create --env=test
```

But that's *way* too much work. Instead, add one more trait to our test class: `use ResetDatabase`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 6
use Zenstruck\Foundry\Test\ResetDatabase;

class DragonTreasureResourceTest extends KernelTestCase
{
// ... line 11
    use ResetDatabase;
// ... lines 13 - 20
}
```

This comes from Foundry: the library we've been using to create dummy fixtures via the factory classes. `ResetDatabase` is *amazing*. It automatically makes sure that the database is *cleared* before each test. So if you have two tests, your second test isn't going to mess up because of some data that the first test added.

It's also going to create the database automatically for us. Check it out. Run

```
symfony php bin/phpunit
```

again and check out the dump. That's our response! It's our beautiful JSON-LD! We don't have any *items* in the collection yet, but it *is* working.

And notice that, when we make this request, we are *not* sending an `Accept` header on the request. Remember, when we use the Swagger UI... it actually *does* send an `Accept` header that advertises that we want `application/ld+json`.

We *can* add that to our test if we want. But if we pass nothing, we get JSON-LD back because that's the *default* format of our API.

Next: let's properly finish this test, including seeding the database with data and learning about Browser's API assertions.

Chapter 15: JSON Test Assertions & Seeding the Database

Let's make this test real with data and assertions.

There are two main ways to do assertions with Browser. First, it comes with a bunch of built-in methods to help, like `->assertJson()`. Or... you can always just grab the JSON that comes back from an endpoint and check things using the built-in PHPUnit assertions you know and love. We'll see both.

Let's start by checking `->assertJson()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 8
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 11 - 13
    public function testGetCollectionOfTreasures(): void
    {
        $this->browser()
            ->get('/api/treasures')
// ... line 18
            ->assertJson()
        ;
    }
}
```

When we run that:

```
symfony php bin/phpunit
```

It passes! Cool! We know that this response should have a `hydra:totalItems` property set to the number of results. Right now, our database is empty... but we can at least assert that it matches zero.

To do that, use `->assertJsonMatches()`.

This is a special method from Browser that uses a special syntax that allows us to read different parts off the JSON. We'll dig into it in a minute.

But this one is simple: assert that `hydra:totalItems` equals `0`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 8
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 11 - 13
    public function testGetCollectionOfTreasures(): void
    {
        $this->browser()
// ... lines 17 - 18
            ->assertJson()
            ->assertJsonMatches('hydra:totalItems', 0)
        ;
    }
}
```

When we try this:

```
symfony php bin/phpunit
```

It fails! But with a great error:

> "*`mtdowling/jmespath.php` is required to search JSON*"

Hello JMESPath

Ah, we need to install that! Copy the `composer require` line, find your terminal, and run it:

```
composer require mtdowling/jmespath.php --dev
```

This "JMESPath" thing is actually super cool: it's a "query language" for reading different parts of any JSON. For example, if this is your JSON and you want to read the `a` key, just say `a`. Simple.

But you can also do deeper, like: `a.b.c.d`. Or, get crazier: grab the `1` index, or grab `a.b.c`, then the `0` index, `.d`, the `1` index then the `0` index. You can even slice the array in different ways. Basically... you can go nuts.

But we're *not* going to lose our minds with this. It's a handy syntax... but if things get too complex, we can always test the JSON manually, which we'll do in a bit.

Anyway, now that we have the library installed, let's run the test again.

```
symfony php bin/phpunit
```

It still fails! With a weird error:

> *"Syntax error at character 5 `hydra:totalItems`."*

Unfortunately, the `:` is a special character inside of JMESPath. So whenever we have a `:`, we need to put quotes around that key:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 8
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 11 - 13
    public function testGetCollectionOfTreasures(): void
    {
        $this->browser()
// ... lines 17 - 19
            ->assertJsonMatches('"hydra:totalItems"', 0)
        ;
    }
}
```

Not ideal, but not a huge inconvenience.

Now when we try it:

```
symfony php bin/phpunit
```

It passes!

Seeding the Database

But... this isn't a very interesting test: we're just asserting that we get nothing back... because the database is empty. To make our test *real*, we need data: we need to *seed* the database with data at the start of the test.

Tip

To use Foundry factories in a test, also add a `use Factories;` trait to the top of your test class. Things worked without that in this case, but in the future, you'll likely get an error.

Fortunately, Foundry makes that dead-simple. At the top, call `DragonTreasureFactory::createMany()` and let's create 5 treasures. Now, below, assert that we get 5 results:

tests/Functional/DragonTreasureResourceTest.php

```
// ... lines 1 - 4
use App\Factory\DragonTreasureFactory;
// ... lines 6 - 9
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 12 - 14
    public function testGetCollectionOfTreasures(): void
    {
        DragonTreasureFactory::createMany(5);

        $this->browser()
// ... lines 20 - 22
            ->assertJsonMatches('"hydra:totalItems"', 5)
// ... line 24
        ;
    }
}
```

It's just that simple. And actually, let me put our dump back so we can see the result:

```php
// tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 9
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 12 - 14
    public function testGetCollectionOfTreasures(): void
    {
// ... lines 17 - 18
        $this->browser()
// ... line 20
            ->dump()
// ... line 22
            ->assertJsonMatches('"hydra:totalItems"', 5)
// ... line 24
        ;
    }
}
```

Try it now:

```
symfony php bin/phpunit
```

It passes! And if you look up, yea! The response has 5 treasures! Dang, that was easy.

Next: let's use JMESPath to assert something more challenging. Then we'll back up and see how we can dig into Browser to give us infinite flexibility - and simplicity - when it comes to testing JSON.

Chapter 16: Advanced & Flexible JSON Test Assertions

We might also want to test that we get the correct fields in the response for each item. Can we do that with JMESPath? Sure! The `assertJsonMatches()` method is really handy. And actually, if you hold command or control and click into it, when we call `assertJsonMatches()`, behind the scenes, it calls `$this->json()`. This creates a `Json` object... which has even *more* useful methods. The `Browser` instance itself gives us access to `assertJsonMatches()`. But if we want to use any of its other methods, we need to do a bit more work.

The first way to use the `Json` object is via Browser's `use()` method. Pass this a callback with a `Json $json` argument:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 6
use Zenstruck\Browser\Json;
// ... lines 8 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 15
    public function testGetCollectionOfTreasures(): void
    {
// ... lines 18 - 19
        $this->browser()
// ... lines 21 - 24
            ->use(function(Json $json) {
// ... line 26
            })
        ;
    }
}
```

This is a magic feature of browser: it reads the type-hint of the argument, and knows to pass us the `Json` object. You could also type-hint a `CookieJar` object, `Crawler` or a few other things.

The point is: because we type-hinted the argument with `Json`, it will grab the `Json` object for the last response and pass it to us. Let's use it to do some experimenting. We want to check what the *keys* are for the first item inside of `hydra:member`. To help figure the expression we need, let's use a method called `search()`. This allows us to use a `JMESPath` expression and get back the result. Do double quotes then `hydra:member` to see what it returns. And... remove the other dump:

```
tests/Functional/DragonTreasureResourceTest.php
```

```php
// ... lines 1 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 15
    public function testGetCollectionOfTreasures(): void
    {
// ... lines 18 - 19
        $this->browser()
// ... lines 21 - 24
            ->use(function(Json $json) {
                dump($json->search('"hydra:member"'));
            })
        ;
    }
}
```

Ok! Run that test again:

```
symfony php bin/phpunit
```

It passes... but more importantly, look at the dump! It's the array of 5 items. Ok... let's grab the `0` index. After the `hydra:member` double quotes, add `[0]`. Then surround the *entire* thing with a `keys()` function from JMESPath:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 15
    public function testGetCollectionOfTreasures(): void
    {
// ... lines 18 - 19
        $this->browser()
// ... lines 21 - 24
            ->use(function(Json $json) {
                dump($json->search('keys("hydra:member"[0])'));
            })
        ;
    }
}
```

Try that now:

```
symfony php bin/phpunit
```

Oh that's lovely. And it's probably one of the more complex things that you'll do. Now that we've got the path right, turn that into an assertion. You can do that by setting this to a variable - like `$keys` - and using a normal assertion. Or you can change `search` to `assertMatches()` and pass a second argument: the array of the expected fields:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 15
    public function testGetCollectionOfTreasures(): void
    {
// ... lines 18 - 19
        $this->browser()
// ... lines 21 - 24
            ->use(function(Json $json) {
                $json->assertMatches('keys("hydra:member"[0])', [
                    '@id',
                    '@type',
                    'name',
                    'description',
                    'value',
                    'coolFactor',
                    'owner',
                    'shortDescription',
                    'plunderedAtAgo',
                ]);
            })
        ;
    }
}
```

We should be good! Try it:

```
symfony php bin/phpunit
```

It passes! And yes, we *could* now remove the `use()` method and move this to a normal `->assertJsonMatches()` call.

Doing Normal JSON Assertions

As cool as this JMESPath stuff is, it *is* another thing to learn and it *can* get complex. So what's the alternative?

Assign the entire `$browser` chain to a new `$json` variable and then add `->json()` to the end. *Most* methods on `Browser` return... a `Browser`, which let's us do all the fun chaining. But a few, like `->json()` let us "break out" of browser so we can do something custom.

This allows us to remove the `use()` function here and replace the assertions with more traditional PHPUnit code:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 15
    public function testGetCollectionOfTreasures(): void
    {
        DragonTreasureFactory::createMany(5);

        $json = $this->browser()
            ->get('/api/treasures')
            ->assertJson()
            ->assertJsonMatches('"hydra:totalItems"', 5)
            ->assertJsonMatches('length("hydra:member")', 5)
            ->json()
        ;

        $json->assertMatches('keys("hydra:member"[0])', [
            '@id',
            '@type',
            'name',
            'description',
            'value',
            'coolFactor',
            'owner',
            'shortDescription',
            'plunderedAtAgo',
        ]);
    }
}
```

We could *still* use the `Json` object directly... that passes... or to remove all fanciness, change to `$this->assertSame()` that `$json->decoded()['hydra:member'][0]` - `array_keys()` around everything - matches our array:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 15
    public function testGetCollectionOfTreasures(): void
    {
        DragonTreasureFactory::createMany(5);

        $json = $this->browser()
            ->get('/api/treasures')
            ->assertJson()
            ->assertJsonMatches('"hydra:totalItems"', 5)
            ->assertJsonMatches('length("hydra:member")', 5)
            ->json()
        ;

        $this->assertSame(array_keys($json->decoded()['hydra:member'][0]),
[
            '@id',
            '@type',
            'name',
            'description',
            'value',
            'coolFactor',
            'owner',
            'shortDescription',
            'plunderedAtAgo',
        ]);
    }
}
```

And of course... that passes to!

So, a lot of power... but also a lot of flexibility to write tests how you want.

Next, let's add tests for authentication: both logging in via our login form and via an API token.

Chapter 17: Testing Authentication

Let's create a test to post and create a new treasure. Say `public function testPostToCreateTreasure()` that returns `void`. And start the same way as before: `$this->browser()->post('/api/treasures')`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 40
    public function testPostToCreateTreasure(): void
    {
        $this->browser()
            ->post('/api/treasures', [
// ... line 45
            ])
// ... lines 47 - 48
        ;
    }
}
```

In this case we need to *send* data. The second argument to any of these `post()` or `get()` methods is an array of options, which can include `headers`, `query` parameters or other stuff. One key is `json`, which you can set to an array, which will be JSON-encoded for you. Start by sending empty JSON... then `->assertStatus(422)`. To see what the response looks like, add `->dump()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 13 - 40
    public function testPostToCreateTreasure(): void
    {
        $this->browser()
            ->post('/api/treasures', [
                'json' => [],
            ])
            ->assertStatus(422)
            ->dump()
        ;
    }
}
```

Awesome! Copy the test method name. I want to focus *just* on this one test. To do that, run:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

And... oh! Current response status code is 401, but 422 expected.

Dumped Failed Responses in Browser

When a test fails with browser, it automatically saves the last response to a file... which is *awesome*. It's actually in the `var/` directory. In my terminal, I can hold `Command` and click to open that in my browser. *That* is nice. You'll see me do this a bunch of times.

Ok, so this returned a 401 status code. Of course: the endpoint requires authentication! Our app has *two* ways to authenticate: via the login form and session or via an API token. We're going to test both, starting with the login form.

Logging in during the Test

To log in as a user... that user first needs to *exist* in the database. Remember: at the start of each test, our database is empty. It's then *our* job to populate it with whatever we need.

Create a user with `UserFactory::createOne(['password' => 'pass'])` so that we know what the password will be. Then, before we make the POST request to create a treasure, `->post()` to `/login` and send `json` with `email` set to `$user->getEmail()` - to use whatever random email address Faker chose - then `password` set to `pass`. To make sure that worked, `->assertStatus(204)`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 5
use App\Factory\UserFactory;
// ... lines 7 - 11
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 14 - 41
    public function testPostToCreateTreasure(): void
    {
        $user = UserFactory::createOne(['password' => 'pass']);

        $this->browser()
            ->post('/login', [
                'json' => [
                    'email' => $user->getEmail(),
                    'password' => 'pass',
                ],
            ])
            ->assertStatus(204)
// ... lines 54 - 58
        ;
    }
}
```

That's the status code we're returning after successful authentication.

Let's give this a try! Move over and run the test:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

It passes! We're getting the 422 status code and see the validation messages!

Shortcut to Logging in: actingAs()

So... logging in is... just that easy! And I *would* recommend having a test that specifically POSTs to your login endpoint like we just did, to make sure its working correctly.

However, in all of my *other* tests... when I simply need to be authenticated to do the *real* work, there's a faster way to log in. Instead of making the POST request, say `->actingAs($user)`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 11
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 14 - 41
    public function testPostToCreateTreasure(): void
    {
// ... lines 44 - 45
        $this->browser()
            ->actingAs($user)
// ... lines 48 - 52
        ;
    }
}
```

This is a sneaky way of taking the `User` object and pushing it directly into Symfony's security system without making any requests. It's easier, and faster. And now, we don't care what the password is at all, so we can simplify that.

Let's check it:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Still good!

Testing Successful Treasure Creation

Let's do another `POST` down here. Keep chaining and add `->post()`. Actually... I'm lazy. Copy the existing `->post()`... and use that. But this time, send real data: I'll quickly type in some... these can be anything. The last key we need is `owner`. Right now, we *are* required to send the `owner` when we create a treasure. Soon, we'll make that optional: if we don't send it, it will default to whoever is authenticated. But for now, set it to `/api/users/` then `$user->getId()`. Finish with `assertStatus(201)`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 11
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 14 - 41
    public function testPostToCreateTreasure(): void
    {
        $user = UserFactory::createOne();

        $this->browser()
            ->actingAs($user)
            ->post('/api/treasures', [
                'json' => [],
            ])
            ->assertStatus(422)
            ->post('/api/treasures', [
                'json' => [
                    'name' => 'A shiny thing',
                    'description' => 'It sparkles when I wave it in the
air.',
                    'value' => 1000,
                    'coolFactor' => 5,
                    'owner' => '/api/users/'.$user->getId(),
                ],
            ])
            ->assertStatus(201)
        ;
    }
}
```

Because 201 is what the API returns when an object is created.

Alright, go test, go:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Still passing! We're on a roll! Add a `->dump()` to help us debug then a sanity check: `->assertJsonMatches()` that `name` is `A shiny thing`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 11
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 14 - 41
    public function testPostToCreateTreasure(): void
    {
// ... lines 44 - 45
        $this->browser()
// ... lines 47 - 60
            ->assertStatus(201)
            ->dump()
            ->assertJsonMatches('name', 'A shiny thing')
        ;
    }
}
```

When we try that:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Sending the Accept: application/ld+json Header

No surprise: all green. But look at the dumped response: it's *not* JSON-LD! We're getting back standard JSON. You can see it in the `Content-Type` header: `'application/json'`, *not* `application/ld+json`, which is what I was expecting.

Let's find out what's going on next and fix it globally by customizing how Browser works across our entire test suite.

Chapter 18: Customizing Browser Globally

Our test works... but the API is sending us back JSON, not JSON-LD. Why?

When we made the `GET` request earlier, we did *not* include an `Accept` header to indicate which format we wanted back. But... JSON-LD is our API's *default* format, so it sent that back.

However, when we make a `->post()` request with the `json` key, that adds a `Content-Type` header set to `application/json` - which is fine - but it *also* adds an `Accept` header set to `application/json`. Yup, we're telling the server that we want plain JSON back, not JSON-LD.

I want to use JSON-LD everywhere. How can we do that? The second argument to `->post()` can be an array *or* an object called `HttpOptions`. Say `HttpOptions::json()`... and then pass the array directly. Let me... get my syntax right:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 7
use Zenstruck\Browser\HttpOptions;
// ... lines 9 - 12
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 15 - 42
    public function testPostToCreateTreasure(): void
    {
// ... lines 45 - 52
                ->post('/api/treasures', HttpOptions::json([
                    'name' => 'A shiny thing',
                    'description' => 'It sparkles when I wave it in the air.',
                    'value' => 1000,
                    'coolFactor' => 5,
                    'owner' => '/api/users/'.$user->getId(),
                ]))
// ... lines 60 - 62
            ;
    }
}
```

So far, this is equivalent to what we had before. But now we can *change* some options by saying `->withHeader()` passing `Accept` and `application/ld+json`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends KernelTestCase
{
// ... lines 15 - 42
    public function testPostToCreateTreasure(): void
    {
// ... lines 45 - 52
            ->post('/api/treasures', HttpOptions::json([
                'name' => 'A shiny thing',
                'description' => 'It sparkles when I wave it in the air.',
                'value' => 1000,
                'coolFactor' => 5,
                'owner' => '/api/users/'.$user->getId(),
            ])->withHeader('Accept', 'application/ld+json'))
// ... lines 60 - 62
        ;
    }
}
```

We *could* have also done this with the *array* of options: it has a key called `headers`. But the object is kind of nice.

Let's make sure this fixes things. Run the test:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Globally Sending the Header

And... we're back to JSON-LD! It's got the right fields and the `application/ld+json` response `Content-Type` header.

So.... that's cool... but doing this *every* time we make a request to our API in the tests is... mega lame. We need this to happen automatically.

A nice way to do that is to leverage a base test class. Inside of `tests/`, actually inside of `tests/Functional/`, create a new PHP class called `ApiTestCase`. I'm going to make this `abstract` and extend `KernelTestCase`:

tests/Functional/ApiTestCase.php

```php
// ... lines 1 - 2
namespace App\Tests\Functional;

use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
// ... lines 6 - 9
abstract class ApiTestCase extends KernelTestCase
{
// ... lines 12 - 25
}
```

Inside, add the `HasBrowser` trait. But we're going to do something sneaky: we're going to import the `browser()` method but *call* it `baseKernelBrowser`:

tests/Functional/ApiTestCase.php

```php
// ... lines 1 - 7
use Zenstruck\Browser\Test\HasBrowser;

abstract class ApiTestCase extends KernelTestCase
{
    use HasBrowser {
        browser as baseKernelBrowser;
    }
// ... lines 15 - 25
}
```

Why the heck are we doing that? Re-implement the `browser()` method... then call `$this->baseKernelBrowser()` passing it `$options` and `$server`. But *now* call *another* method: `->setDefaultHttpOptions()`. Pass this `HttpOptions::create()` then `->withHeader()`, `Accept`, `application/ld+json`:

```
tests/Functional/ApiTestCase.php
// ... lines 1 - 5
use Zenstruck\Browser\HttpOptions;
// ... lines 7 - 9
abstract class ApiTestCase extends KernelTestCase
{
// ... lines 12 - 15
    protected function browser(array $options = [], array $server = [])
    {
        return $this->baseKernelBrowser($options, $server)
            ->setDefaultHttpOptions(
                HttpOptions::create()
                    ->withHeader('Accept', 'application/ld+json')

            )
        ;
    }
}
```

Done! Back in our real test class, extend `ApiTestCase`: get the one that's from *our* app:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 11
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 14 - 63
}
```

That's it! When we say `$this->browser()`, it now calls *our* `browser()` method, which changes that default option. Celebrate by removing `withHeader()`... and you could revert back to the array of options with a `json` key if you want.

Let's try it.

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

And... uh oh. That's a strange error:

> *“Cannot override final method `_resetBrowserClients()`”*

This... is because we're importing the trait from the parent class *and* our class... which makes the trait go bananas. Remove the one inside our test class:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 8
use Zenstruck\Browser\Test\HasBrowser;
// ... lines 10 - 11
class DragonTreasureResourceTest extends ApiTestCase
{
    use HasBrowser;
// ... lines 15 - 63
}
```

we don't need it anymore. I'll also do a little cleanup on my `use` statements.

And now:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Got it! We get back JSON-LD with zero extra work. Remove that `dump()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 11
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 14 - 41
    public function testPostToCreateTreasure(): void
    {
// ... lines 44 - 45
        $this->browser()
// ... lines 47 - 59
            ->dump()
// ... line 61
        ;
    }
}
```

Next: let's write another test that uses our API token authentication.

Chapter 19: Testing Token Authentication

What about a test like this... but where we log in with an API key? Let's do that! Create a new method: public function `testPostToCreateTreasureWithApiKey()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 13 - 61
    public function testPostToCreateTreasureWithApiKey(): void
    {
// ... lines 64 - 70
    }
}
```

This will start pretty much the same as before. I'll copy the top of the previous test, remove the `actingAs()`... and add a `dump()` near the bottom:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 10
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 13 - 61
    public function testPostToCreateTreasureWithApiKey(): void
    {
        $this->browser()
            ->post('/api/treasures', [
                'json' => [],
            ])
            ->dump()
            ->assertStatus(422)
        ;
    }
}
```

So, like before, we're sending invalid data and expect a 422 status code.

Copy that method name, then spin over and run *just* this test:

```
symfony php bin/phpunit --filter=testPostToCreateTreasureWithApiKey
```

And... no surprise: we get a 401 status code because we're *not* authenticated.

Let's send an `Authorization` header, but an invalid one to start. Pass a `headers` key set to an array with `Authorization` and then word `Bearer` and then... `foo`.

This should still fail:

```
symfony php bin/phpunit --filter=testPostToCreateTreasureWithApiKey
```

And... it does! But with a different error message: `invalid_token`. Nice!

Using a Real Token

To pass a *real* token, we need to put a real token into the database. Do that with `$token = ApiTokenFactory::createOne()`:

```php
// tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 63
    public function testPostToCreateTreasureWithApiKey(): void
    {
        $token = ApiTokenFactory::createOne([
// ... line 67
        ]);
// ... lines 69 - 79
    }
}
```

Do we need to control any fields on this? We actually *do*. Open up `DragonTreasure`. If we scroll up, the `Post` operation requires `ROLE_TREASURE_CREATE`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 37
        new Post(
            security: 'is_granted("ROLE_TREASURE_CREATE")',
        ),
// ... lines 41 - 49
    ],
// ... lines 51 - 64
)]
// ... lines 66 - 83
class DragonTreasure
{
// ... lines 86 - 243
}
```

When we authenticate via the login form, thanks to `role_hierarchy`, we always have that. But when using an API key, to get that role, the token needs the corresponding scope.

To make sure we have it, back in the test, set the `scopes` property to `ApiToken::SCOPE_TREASURE_CREATE`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 4
use App\Entity\ApiToken;
// ... lines 6 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 63
    public function testPostToCreateTreasureWithApiKey(): void
    {
        $token = ApiTokenFactory::createOne([
            'scopes' => [ApiToken::SCOPE_TREASURE_CREATE]
        ]);
// ... lines 69 - 79
    }
}
```

Now pass this to the header: `$token->getToken()`. Oh... and let me fix `scopes`: that should be an array:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 63
    public function testPostToCreateTreasureWithApiKey(): void
    {
        $token = ApiTokenFactory::createOne([
            'scopes' => [ApiToken::SCOPE_TREASURE_CREATE]
        ]);
// ... line 69
        $this->browser()
            ->post('/api/treasures', [
// ... line 72
                'headers' => [
                    'Authorization' => 'Bearer '.$token->getToken()
                ]
            ])
// ... lines 77 - 78
        ;
    }
}
```

I think we're ready! Run that test:

```
symfony php bin/phpunit --filter=testPostToCreateTreasureWithApiKey
```

And... got it! We see the beautiful 422 validation errors!

Testing a Token with a Bad Scope

Let's test to make sure we *don't* have access if our token is *missing* this scope. Copy the entire test method... then paste below. Call it `testPostToCreateTreasureDeniedWithoutScope()`.

This time, set `scopes` to something else, like `SCOPE_TREASURE_EDIT`. Below, we now expect a 403 status code:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 80
    public function testPostToCreateTreasureDeniedWithoutScope(): void
    {
        $token = ApiTokenFactory::createOne([
            'scopes' => [ApiToken::SCOPE_TREASURE_EDIT]
        ]);

        $this->browser()
            ->post('/api/treasures', [
                'json' => [],
                'headers' => [
                    'Authorization' => 'Bearer '.$token->getToken()
                ]
            ])
            ->assertStatus(403)
        ;
    }
}
```

This time, let's run *all* the tests:

```
symfony php bin/phpunit
```

And... all green! A 422 then a 403. Go remove the dumps from both those spots.

By the way, if you use API tokens a lot in your tests, passing the `Authorization` header can get annoying. Browser has a way where we can create a *custom* Browser object with custom methods. For example, you could add an `authWithToken()` method, pass an array of scopes, and then it would create that token and set it into the header

```
$this->browser()
    ->authWithToken([ApiToken::SCOPE_TREASURE_CREATE])
    // ...
;
```

This totally does *not* work right now, but check out Browser's docs to learn how.

Next: in API Platform 3.1, the behavior of the `PUT` operation is changing. Let's talk about how, and what we need to do in our code to prepare for it.

Chapter 20: New PUT Behavior

Find your terminal and manually clear the cache directory:

```
rm -rf var/cache/*
```

I'm doing this so that, when we run all or our tests

```
symfony php bin/phpunit
```

we see a deprecation warning, which is fascinating. It says:

> *“Since API Platform 3.1: in API Platform 4, `PUT` will always replace the data. set `extraProperties["standard_put"]` to `true` on every operation to avoid breaking PUT's behavior. Use `PATCH` for the old behavior.”*

Okay... what does that mean? Right now, it means nothing has changed: our `PUT` operation behaves like it always has. But, in API Platform 4, the behavior of `PUT` will change dramatically. And, at some point between now and then, we need to opt *into* that new behavior so that it doesn't suddenly break when we upgrade to version 4 in the future.

What's Changing in PUT

So what's changing exactly? Head over to the API docs and refresh. Use the `GET` collection endpoint... and hit "Execute", so we can get a valid ID.

Great: we have a treasure with ID 1.

Right now, if we send a `PUT` request with this ID, we can send just *one* field to update just that *one* thing. For example, we can send `description` to change *only* that.

Oh, but before we Execute this, we *do* need to be logged in. In my other tab, I'll fill in the login form. Perfect. *Now* execute the `PUT` operation.

Yup: we pass only the `description` field, and it *updates* only the `description` field: all the other fields remain the same.

Whelp, it turns out that this is *not* how `PUT` is supposed to work according to the HTTP Spec. `PUT` is *supposed* to be a "replace". What I mean is, if we send only one field, the `PUT` operation is supposed to take that new resource - which is just the one field - and *replace* the existing resource. That's a complicated way of saying that, when using PUT, you need to send *every* field, even the fields that aren't changing. Otherwise, they'll be set to `null`.

If that sounds kind of crazy, I kind of agree, but there are valid technical reasons for why this is the case. The point is that: this is how `PUT` is *supposed* to work and in API Platform 4, this is how `PUT` *will* work.

Honestly, it makes `PUT` less useful. So you'll notice that I'll pretty much exclusively use `PATCH` going forward.

Moving to the new PUT Behavior

So whether we like it or not, at some point between now and API platform 4, we need to tell API Platform that it is okay for it to change the behavior of `PUT` to the "new" way. Let's do that now by adding some extra config to every `ApiResource` attribute in our app.

Tip

To solve this globally for all your resources at once, you can add this as a default in the API Platform configuration:

```
# config/packages/api_platform.yaml
    api_platform:
        defaults:
        extra_properties:
            standard_put: true
```

Open `src/Entity/DragonTreasure.php`... and add a new option called `extraProperties` set to an array with `standard_put` set to `true`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 64
    extraProperties: [
        'standard_put' => true,
    ],
)]
// ... lines 69 - 89
class DragonTreasure
{
// ... lines 92 - 249
}
```

That's it! Copy that... because we're going to need that down here on this `ApiResource`... even though it doesn't have a `PUT` operation:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 64
    extraProperties: [
        'standard_put' => true,
    ],
)]
#[ApiResource(
// ... lines 70 - 81
    extraProperties: [
        'standard_put' => true,
    ],
)]
// ... lines 86 - 89
class DragonTreasure
{
// ... lines 92 - 249
}
```

Then, over in `User`, add that to both of the `ApiResource` spots as well:

```
src/Entity/User.php
// ... lines 1 - 25
#[ApiResource(
// ... lines 27 - 44
    extraProperties: [
        'standard_put' => true,
    ],
)]
#[ApiResource(
// ... lines 50 - 59
    extraProperties: [
        'standard_put' => true,
    ],
)]
// ... lines 64 - 66
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 69 - 276
}
```

Now when we run our tests, the deprecation is gone! We're not *using* the `PUT` operation in any tests, so everything still passes.

Seeing the New Behavior

To see the new behavior, try out the `PUT` endpoint again: still sending just *one* field. This time... check it out! A 422 validation error! All the fields that we did *not* include were set to null... and that caused the validation failure.

So... this makes `PUT` a bit less useful... and we'll lean a lot more on `PATCH`. If you don't want to have a `PUT` operation at all anymore, that makes a lot of sense. One *unique* thing about the new `PUT` behavior is that you could use it to create *new* objects... which could be useful in some edge-cases... or an absolute nightmare from a security standpoint as we now need to worry about objects being edited or *created* via the same `PUT` operation. For that reason, as we go along, you'll see me remove the `PUT` operation in some cases.

Next: let's get more complex with security by making sure that a `DragonTreasure` can only be edited by its owner.

Chapter 21: Only Allow Owners to Edit

New security quest: I want to allow *only* the *owner* of a treasure to edit it. Right now, you're allowed to edit a treasure as long as you have this role. But that means you can edit *anyone's* treasure. Someone keep changing my Velvis painting's `coolFactor` to 0. That's super uncool.

TDD: Testing the only Owners can Edit

Let's write a test for this. At the bottom say `public function testPatchToUpdateTreasure()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 97
    public function testPatchToUpdateTreasure()
    {
// ... lines 100 - 112
    }
}
```

And we'll start like normal: `$user = UserFactory::createOne()` then `$this->browser->actingAs($user)`.

Since we're editing a treasure, let's `->patch()` to `/api/treasures/`... and then we need a treasure to edit! Create one on top: `$treasure = DragonTreasureFactory::createOne()`. And for this test, we want to make sure that the `owner` is *definitely* this `$user`. Finish the URL with `$treasure->getId()`.

For the data, send some `json` to update *just* the `value` field to `12345`, then `assertStatus(200)` and `assertJsonMatches('value', 12345)`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 97
    public function testPatchToUpdateTreasure()
    {
        $user = UserFactory::createOne();
        $treasure = DragonTreasureFactory::createOne(['owner' => $user]);

        $this->browser()
            ->actingAs($user)
            ->patch('/api/treasures/'.$treasure->getId(), [
                'json' => [
                    'value' => 12345,
                ],
            ])
            ->assertStatus(200)
            ->assertJsonMatches('value', 12345)
        ;
    }
}
```

Excellent! This *should* be allowed because we're the `owner`. Copy the method name, then find your terminal and run it:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

No surprise, it passes.

Now let's try the *other* case: let's log in as someone *else* and try to update this treasure.

Copy the entire `$browser` section. We *could* create another test method, but this will work fine all in one. Before this, add `$user2 = UserFactory::createOne()` - then log in as *that* user. This time, change the `value` to `6789` and, since this should *not* be allowed, assert that the status code is 403:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 97
    public function testPatchToUpdateTreasure()
    {
// ... lines 100 - 113
        $user2 = UserFactory::createOne();
        $this->browser()
            ->actingAs($user2)
            ->patch('/api/treasures/'.$treasure->getId(), [
                'json' => [
                    'value' => 6789,
                ],
            ])
            ->assertStatus(403)
        ;
    }
}
```

When we try the test now:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

It fails! This *is* being allowed: the API returned a 200!

More Complex security Expressions

So how can we make it so that only the *owner* of a treasure can edit it? Well, over in `DragonTreasure`, the answer is all about the `security` option:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Put(
            security: 'is_granted("ROLE_TREASURE_EDIT")',
        ),
        new Patch(
            security: 'is_granted("ROLE_TREASURE_EDIT")',
        ),
// ... lines 47 - 49
    ],
// ... lines 51 - 67
)]
// ... lines 69 - 89
class DragonTreasure
{
// ... lines 92 - 249
}
```

One thing that gets tricky with `Put()` and `Patch()` is that *both* are used to edit users. So if you're going to have both, you need keep their `security` options in sync. I'm actually going to remove `Put()` so we can focus on `Patch()`.

The string inside of `security` is an *expression*... and we can get kinda fancy. We can grant access if you have `ROLE_TREASURE_EDIT` *and* if `object.owner == user`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Patch(
            security: 'is_granted("ROLE_TREASURE_EDIT") and object.owner == user',
        ),
// ... lines 44 - 46
    ],
// ... lines 48 - 64
)]
// ... lines 66 - 86
class DragonTreasure
{
// ... lines 89 - 246
}
```

Inside the security expression, Symfony gives us a few variable. One is `user`, which is the current `User` object. Another is `object`, which will be the current object for this operation. So the `DragonTreasure` object. So we're saying that access should be allowed if the `DragonTreasure`s `owner` is equal to the currently authenticated `user`. That's... exactly what we want!

So, try the test again!

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

And... uh oh! We downgraded to a 500 error! This is where that saved log file comes in handy. I'll click to open that up. If this is hard to read, view the page source. Much better. It says:

> *“Cannot access private property `DragonTreasure::$owner`.”*

And it's coming from Symfony's `ExpressionLanguage`. Ah, I know what's wrong. The expression language is *like* Twig... but not *exactly* the same. We can't do fancy things like `.owner` when `owner` is a private property. We need to call the public method:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Patch(
            security: 'is_granted("ROLE_TREASURE_EDIT") and object.getOwner() == user',
        ),
// ... lines 44 - 46
    ],
// ... lines 48 - 64
)]
// ... lines 66 - 86
class DragonTreasure
{
// ... lines 89 - 246
}
```

Drumroll please:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

It passes with flying colors!

Preventing Changing Owners: securityPostDenormalize

But you know me, I've *gotta* make it trickier. Copy part of the test. This time, log in as the owner and edit our *own* treasure. So far, this is all good. But now try to *change* the `owner` to someone else: `$user2->getId()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 97
    public function testPatchToUpdateTreasure()
    {
// ... lines 100 - 126
        $this->browser()
// ... line 128
            ->patch('/api/treasures/'.$treasure->getId(), [
                'json' => [
                    // change the owner to someone else
                    'owner' => '/api/users/'.$user2->getId(),
                ],
            ])
// ... line 135
        ;
    }
}
```

Now maybe this *is* something you want to allow. Maybe you say:

> *"If you can edit a `DragonTreasure`, then you're free to assign it a different owner."*

But let's pretend that we want to prevent this. So `assertStatus(403)`. Do you think the test will pass? Try it:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

It fails! It *did* allow us to change the `owner`! Spin back over to `DragonTreasure`. The `security` expression is run *before* the new data is deserialized onto the object. In other words, `object` will be the `DragonTreasure` from the *database*, *before* any of the new JSON is applied to it. This means that it's checking that the *current* `owner` is equal to the currently logged-in user, which is the main case that we want to protect.

But sometimes you want to run security *after* the new data has been put onto the object. In that case, use an option called `securityPostDenormalize`. Remember denormalize is the process of taking the data and putting it onto the object. So `security` will still run first... and make sure we're the original owner. Now we can also say `object.getOwner() == user`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Patch(
            security: 'is_granted("ROLE_TREASURE_EDIT") and object.getOwner() == user',
            securityPostDenormalize: 'object.getOwner() == user',
        ),
// ... lines 45 - 47
    ],
// ... lines 49 - 65
)]
// ... lines 67 - 87
class DragonTreasure
{
// ... lines 90 - 247
}
```

That looks identical... but this time `object` will be the `DragonTreasure` with the *new* data. So we're checking that the *new* owner is *also* equal to the currently logged-in user.

By the way, in `securityPostDenormalize`, you also have a `previous_object` variable, which is equal to the object before denormalization. So, it's identical to `object` up in the `security` option. But, we don't need that.

Try the test now:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

We got it!

Security vs Validation

This last example highlights two different *types* of security checks. The first check determines whether or not the user can perform this operation at *all*. Like: is the current user allowed to make a `PATCH` request to this treasure? That depends on the current user and the current DragonTreasure in the database.

But the second check is saying:

> *“Okay, now that I know I'm allowed to make a `PATCH` request, am I allowed to change the data in this exact way?”*

This depends on the currently logged-in user and the *data* that's being sent.

I'm bringing up this difference because, for me, the first case - where you're trying to figure out whether an operation is allowed at all - regardless of what data is being sent - that *is* a job for security. And this is exactly how I would implement it.

However, the second case - where you're trying to figure out whether the user is allowed to send this exact data - like are they allowed to change the `owner` or not - for me, I think that's better handled by the validation layer.

I'm going to keep this in the security layer right now. But later when we talk about custom validation, we'll move this into that.

Up next: can we flex the `security` option enough to *also* let admin users edit *anyone's* treasure? Stay tuned!

Chapter 22: Allow Admin Users to Edit any Treasure

We've got things set up so that only the owner of a treasure can edit it. *Now*, a new requirement has come down from on-high: admin users should be able to edit *any* treasure. That means a user that has `ROLE_ADMIN`.

To the test-mobile! Add a `public function testAdminCanPatchToEditTreasure()`. Then create an admin user with `UserFactory::createOne()` passing roles set to `ROLE_ADMIN`:

tests/Functional/DragonTreasureResourceTest.php

```php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 138
    public function testAdminCanPatchToEditTreasure(): void
    {
        $admin = UserFactory::createOne(['roles' => ['ROLE_ADMIN']]);
    }
}
```

Foundry State Methods

That'll work fine. But if we need to create a lot of admin users in our tests, we can add a shortcut to Foundry. Open `UserFactory`. We're going to create something called a "state" method. Anywhere inside, add a public function called, how about `withRoles()` that has an `array $roles` argument and returns `self`, which will make this more convenient when we use it. Then `return $this->addState(['roles' => $roles])`:

```
src/Factory/UserFactory.php
// ... lines 1 - 30
final class UserFactory extends ModelFactory
{
// ... lines 33 - 54
    public function withRoles(array $roles): self
    {
        return $this->addState(['roles' => $roles]);
    }
// ... lines 59 - 92
}
```

Whatever we pass to `addState()` becomes part of the data that will be used to make this user.

To use the state method, the code changes to `UserFactory::new()`. Instead of creating a `User` object, this instantiates a new `UserFactory`... and then we can call `withRoles()` and pass `ROLE_ADMIN`:

So, we're "crafting" what we want the user to look like. When we're done, call `create()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 138
    public function testAdminCanPatchToEditTreasure(): void
    {
        $admin = UserFactory::new()->withRoles(['ROLE_ADMIN'])->create();
    }
}
```

`createOne()` is a static shortcut method. But since we have an instance of the factory, use `create()`.

But we can go even further. Back in `UserFactory`, add another state method called `asAdmin()` that returns `self`. Inside return `$this->withRoles(['ROLE_ADMIN'])`:

```
src/Factory/UserFactory.php
// ... lines 1 - 30
final class UserFactory extends ModelFactory
{
// ... lines 33 - 59
    public function asAdmin(): self
    {
        return $this->withRoles(['ROLE_ADMIN']);
    }
// ... lines 64 - 97
}
```

Thanks to that, we can simplify to `UserFactory::new()->asAdmin()->create()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 138
    public function testAdminCanPatchToEditTreasure(): void
    {
        $admin = UserFactory::new()->asAdmin()->create();
    }
}
```

Nice!

Writing the Test

Now let's get this test going. Create a new `$treasure` set to `DragonTreasureFactory::createOne()`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 138
    public function testAdminCanPatchToEditTreasure(): void
    {
        $admin = UserFactory::new()->asAdmin()->create();
        $treasure = DragonTreasureFactory::createOne();
// ... lines 143 - 153
    }
}
```

Because we're not passing an `owner`, this will create a new `User` in the background and use *that* as the `owner`. This means that our admin user will *not* be the owner.

Now, `$this->browser()->actingAs($adminUser)` then `->patch()` to `/api/treasures/`, `$treasure->getId()`, sending `json` to update `value` to the same `12345`. `->assertStatus(200)` and `assertJsonMatches()`, `value`, `12345`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 138
    public function testAdminCanPatchToEditTreasure(): void
    {
        $admin = UserFactory::new()->asAdmin()->create();
        $treasure = DragonTreasureFactory::createOne();

        $this->browser()
            ->actingAs($admin)
            ->patch('/api/treasures/'.$treasure->getId(), [
                'json' => [
                    'value' => 12345,
                ],
            ])
            ->assertStatus(200)
            ->assertJsonMatches('value', 12345)
        ;
    }
}
```

Cool! Copy the method name. Let's try it:

```
symfony php bin/phpunit --filter=testAdminCanPatchToEditTreasure
```

And... okay! We haven't implemented this yet, so it fails.

Allowing Admins to Edit Anything

So, how *do* we allow admins to edit any treasure? Well, at first, it's relatively easy because we have total control via the `security` expression. So we can add something like `if is_granted("ROLE_ADMIN") OR` and then put parentheses around the other use-case:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Patch(
            security: 'is_granted("ROLE_ADMIN") or
(is_granted("ROLE_TREASURE_EDIT") and object.getOwner() == user)',
// ... line 43
        ),
// ... lines 45 - 47
    ],
// ... lines 49 - 65
)]
// ... lines 67 - 87
class DragonTreasure
{
// ... lines 90 - 247
}
```

Let's make sure it works!

```
symfony php bin/phpunit --filter=testAdminCanPatchToEditTreasure
```

A 500 error! Let's see what's going on. Click to open this.

> *“Unexpected token "name" around position 26.”*

So... that was an accident. Change `OR` to `or`. And... also move this new logic into `securityPostDenormalize`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Patch(
            security: 'is_granted("ROLE_ADMIN") or
(is_granted("ROLE_TREASURE_EDIT") and object.getOwner() == user)',
            securityPostDenormalize: 'is_granted("ROLE_ADMIN") or
object.getOwner() == user',
        ),
// ... lines 45 - 47
    ],
// ... lines 49 - 65
)]
// ... lines 67 - 87
class DragonTreasure
{
// ... lines 90 - 247
}
```

Then try the test again:

```
symfony php bin/phpunit --filter=testAdminCanPatchToEditTreasure
```

Got it! But my screw-up brings up a great point: the `security` expression is getting *too* complex. It's about as readable as a single-line PERL script... and we do *not* want to make mistakes when it comes to security.

So next, let's centralize this logic with a voter.

Chapter 23: Security Voter

Our security is turning into a madhouse, which I don't like. I want my security logic to be simple and centralized. The way to do that in Symfony is with a *voter*. Let's go create one.

At the command line, run:

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

Call it `DragonTreasureVoter`. It's pretty common to have one voter per *entity* that you need security logic for. So this voter will make all decisions related to `DragonTreasure`: can the current user edit one, delete one, view one: whatever we eventually need.

Go open it up: `src/Security/Voter/DragonTreasureVoter.php`:

src/Security/Voter/DragonTreasureVoter.php

```php
// ... lines 1 - 2
namespace App\Security\Voter;

use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;
use Symfony\Component\Security\Core\User\UserInterface;

class DragonTreasureVoter extends Voter
{
    public const EDIT = 'POST_EDIT';
    public const VIEW = 'POST_VIEW';

    protected function supports(string $attribute, mixed $subject): bool
    {
        // replace with your own logic
        // https://symfony.com/doc/current/security/voters.html
        return in_array($attribute, [self::EDIT, self::VIEW])
            && $subject instanceof \App\Entity\DragonTreasure;
    }

    protected function voteOnAttribute(string $attribute, mixed $subject,
TokenInterface $token): bool
    {
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case self::EDIT:
                // logic to determine if the user can EDIT
                // return true or false
                break;
            case self::VIEW:
                // logic to determine if the user can VIEW
                // return true or false
                break;
        }

        return false;
    }
}
```

Before we talk about this class, let me show you how we'll *use* it. In `DragonTreasure`, we're *still* going to use the `is_granted()` function. But for the first argument, pass `EDIT`... which is just a string I'm making up: you'll see how that's used in the voter. Then pass `object`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Patch(
            security: 'is_granted("EDIT", object)',
// ... line 43
        ),
// ... lines 45 - 47
    ],
// ... lines 49 - 65
)]
// ... lines 67 - 87
class DragonTreasure
{
// ... lines 90 - 247
}
```

We normally pass `is_granted()` a single argument: a role! But you can *also* pass it any random string like `EDIT`... as long as you have a voter set up to handle that. If your voter needs some extra info to make its decision, you can pass that as the second argument.

On a high level, we're asking the security system whether or not the current user is allowed to `EDIT` this `DragonTreasure` object. `DragonTreasureVoter` will make that decision.

Copy this and paste it down for `securityPostDenormalize`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 27
#[ApiResource(
// ... lines 29 - 30
    operations: [
// ... lines 32 - 40
        new Patch(
            security: 'is_granted("EDIT", object)',
            securityPostDenormalize: 'is_granted("EDIT", object)',
        ),
// ... lines 45 - 47
    ],
// ... lines 49 - 65
)]
// ... lines 67 - 87
class DragonTreasure
{
// ... lines 90 - 247
}
```

How Voters Works

So here's the deal: anytime that `is_granted()` is called - from *anywhere*, not just from API Platform - Symfony loops through a list of "voter" classes and tries to figure out which one knows how to make that decision. When we check for a role, there's an existing voter that knows how to handle that. In the case of `EDIT`, there is *no* core voter that knows how to handle that. So we'll make `DragonTreasureVoter` able to handle it.

To determine who can handle an `isGranted` call, Symfony calls `supports()` on each voter passing the same two arguments. For our case, `$attribute` will be `EDIT` and `$subject` will be the `DragonTreasure` object:

```
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 8
class DragonTreasureVoter extends Voter
{
// ... lines 11 - 13
    protected function supports(string $attribute, mixed $subject): bool
    {
// ... lines 16 - 19
    }
// ... lines 21 - 43
}
```

MakeBundle generated a voter that handles checking if we can "edit" or "view" a `DragonTreasure`. We don't need that "view" right now, so I'll delete it. Below, change this to an instance of `DragonTreasure` and I'll retype the end and hit tab to add the `use` statement... just to clean things up:

```
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 9
class DragonTreasureVoter extends Voter
{
    public const EDIT = 'EDIT';

    protected function supports(string $attribute, mixed $subject): bool
    {
        return in_array($attribute, [self::EDIT])
            && $subject instanceof DragonTreasure;
    }
// ... lines 19 - 38
}
```

So if someone calls `isGranted()` and passes the string `EDIT` and a `DragonTreasure` object, *we* know how to make that decision.

Oh, and I need to change the constant value to `EDIT` to match the `EDIT` string we're passing to `is_granted()`.

If we return `true` from `supports()`, Symfony will then call `voteOnAttribute()`. Very simply: we return `true` if the user should have access, `false` otherwise.

To start, just `return false`:

```
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 9
class DragonTreasureVoter extends Voter
{
// ... lines 12 - 19
    protected function voteOnAttribute(string $attribute, mixed $subject,
TokenInterface $token): bool
    {
        return false;
// ... lines 23 - 37
    }
}
```

If we've played our cards right, our voter will swoop in like an overactive superhero every time we make a PATCH request and slam the access door shut. Before we try test that theory, remove the "view" case down here:

```
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 9
class DragonTreasureVoter extends Voter
{
// ... lines 12 - 19
    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
    {
        return false;
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case self::EDIT:
                // logic to determine if the user can EDIT
                // return true or false
                break;
        }

        return false;
    }
}
```

Ok, let's make sure our tests fail! Run:

```
symfony php bin/phpunit
```

And... yes! Two tests fail: both because access is denied. Our voter *is* being called.

Adding the Voter Logic

Back in the class, `voteOnAttribute()` is passed the attribute - `EDIT` - the `$subject` - a `DragonTreasure` object and a `$token`, which is a wrapper around the current `User` object.

So we're first checking to make sure that the user is *actually* authenticated.

After that, `assert()` that `$subject` is an instance of `DragonTreasure` because this method should *only* ever be called when `supports()` return `true`:

src/Security/Voter/DragonTreasureVoter.php

```php
// ... lines 1 - 9
class DragonTreasureVoter extends Voter
{
// ... lines 12 - 19
    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
    {
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        assert($subject instanceof DragonTreasure);

        // ... (check conditions and return true to grant permission) ...
// ... lines 31 - 40
    }
}
```

I'm mostly writing this to help my editor know that `$subject` is a `DragonTreasure`: `assert()` is a handy way to do that.

The `switch` statement only has one `case` right now. And *this* is where our logic will live. Very simply: if `$subject` - that's the `DragonTreasure` - `->getOwner()` equals `$user`, then return `true`. Otherwise, it will hit the `break` and return `false`:

```php
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 9
class DragonTreasureVoter extends Voter
{
// ... lines 12 - 19
    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
    {
// ... lines 22 - 29
        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case self::EDIT:
                if ($subject->getOwner() === $user) {
                    return true;
                }

                break;
        }

        return false;
    }
}
```

This isn't *all* the logic we need, but it's a good start!

Try the tests now:

```
symfony php bin/phpunit
```

Down to one failure!

Checking for Roles in the Voter

What's next? Well, we don't have a test for it, but if we authenticate with an API token, in order to edit a treasure, you need to `ROLE_TREASURE_EDIT`, which you can get via the token scope.

So, in the voter, we need to check if the user has that role. Add a `__construct()` method and autowire `Security` - the one from SecurityBundle - `$security`:

```
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 5
use Symfony\Bundle\SecurityBundle\Security;
// ... lines 7 - 10
class DragonTreasureVoter extends Voter
{
// ... lines 13 - 14
    public function __construct(private Security $security)
    {
    }
// ... lines 18 - 50
}
```

Then, below, before we check the owner, if not `$this->security->isGranted('ROLE_TREASURE_EDIT')`, then *definitely* return `false`:

```
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 10
class DragonTreasureVoter extends Voter
{
// ... lines 13 - 24
    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
    {
// ... lines 27 - 35
        switch ($attribute) {
            case self::EDIT:
                if (!$this->security->isGranted('ROLE_TREASURE_EDIT')) {
                    return false;
                }

                if ($subject->getOwner() === $user) {
                    return true;
                }

                break;
        }
// ... lines 48 - 49
    }
}
```

The last test that's failing is testing that an admin can patch to edit *any* treasure. Because we've already injected the `Security` service, this is easy.

Let's pretend admin users will be able to do *anything*. So above the `switch`, if `$this->security->isGranted('ROLE_ADMIN')`, then return `true`:

```
src/Security/Voter/DragonTreasureVoter.php
// ... lines 1 - 10
class DragonTreasureVoter extends Voter
{
// ... lines 13 - 24
    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
    {
// ... lines 27 - 32
        if ($this->security->isGranted('ROLE_ADMIN')) {
            return true;
        }

        assert($subject instanceof DragonTreasure);
// ... lines 38 - 53
    }
}
```

Moment of truth:

```
symfony php bin/phpunit
```

Voilà! Our logic has found a cozy home inside the voter, the `security` expression is now so simple it's almost scary, and we got to write our logic in PHP.

Next: let's explore hiding certain fields in the response based on the user.

Chapter 24: Conditional Fields by User: ApiProperty

We control which fields are readable and writable via serialization groups. But what if you have a field that should be included in the API... but *only* for certain users? Sadly, groups can't pull off that kind of magic on their own.

For example, find the `$isPublished` field and let's make this part of our API by adding the `treasure:read` and `treasure:write` groups:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 87
class DragonTreasure
{
// ... lines 90 - 127
    #[Groups(['treasure:read', 'treasure:write'])]
    private bool $isPublished = false;
// ... lines 130 - 248
}
```

Now if we spin over and try the tests:

```
symfony php bin/phpunit
```

This makes one test fail: `testGetCollectionOfTreasures` sees that `isPublished` is being returned... and it's not expecting it.

Here's the plan: we'll sneak the field into our API but *only* for admin users *or* owners of this `DragonTreasure`. How can we pull that off?

Hello ApiProperty

Well, surprise! We don't often need it, but we can add an `ApiProperty` attribute above any property to help *further* configure it. It has a bunch of stuff, like a description that helps with your documentation and many edge-case things. There's even one called `readable`. If we said `readable: false`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 88
class DragonTreasure
{
// ... lines 91 - 129
    #[ApiProperty(readable: false)]
    private bool $isPublished = false;
// ... lines 132 - 250
}
```

Then the serialization groups would say that this *should* be included in the response... but then this would override that. Watch: if we try the tests:

```
symfony php bin/phpunit
```

They pass because the field is gone.

The security Option

For *our* mission, we can leverage a super cool option called `security`. Set it to `is_granted("ROLE_ADMIN")`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 8
use ApiPlatform\Metadata\ApiProperty;
// ... lines 10 - 88
class DragonTreasure
{
// ... lines 91 - 129
    #[ApiProperty(security: 'is_granted("ROLE_ADMIN")')]
    private bool $isPublished = false;
// ... lines 132 - 250
}
```

That's it! If this expression return false, `isPublished` will *not* be included in the API: it won't be readable *or* writable.

And when we run the tests now:

```
symfony php bin/phpunit
```

They still pass, which means `isPublished` is *not* being returned.

Now let's go test the "happy" path where this field *is* returned. Pop open `DragonTreasureResourceTest`. Here's the original test: `testGetCollectionOfTreasures()`. We're anonymous, so `isPublished` isn't returned.

Now scroll down to `testAdminCanPatchToEditTreasure()`. When we create the `DragonTreasure`, let's make sure it always starts with `isPublished => false`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 138
    public function testAdminCanPatchToEditTreasure(): void
    {
        $admin = UserFactory::new()->asAdmin()->create();
        $treasure = DragonTreasureFactory::createOne([
            'isPublished' => false,
        ]);
// ... lines 145 - 156
    }
}
```

Then, down here, `assertJsonMatches('isPublished', false)` to test that the field *is* returned:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 138
    public function testAdminCanPatchToEditTreasure(): void
    {
        $admin = UserFactory::new()->asAdmin()->create();
        $treasure = DragonTreasureFactory::createOne([
            'isPublished' => false,
        ]);

        $this->browser()
// ... lines 147 - 154
            ->assertJsonMatches('isPublished', false)
        ;
    }
}
```

Copy the test name, spin over and add `--filter` to run *just* that test:

```
symfony php bin/phpunit --filter=testAdminCanPatchToEditTreasure
```

And... it passes! The field *is* being returned when we're an admin.

Also Returning isPublished for the Owner

What about if we're the *owner* of the treasure? Copy the test... rename it to `testOwnerCanSeeIsPublishedField()`... and let's tweak a few things. Rename `$admin` to `$user`, simplify this to `DragonTreasureFactory::createOne()` and make sure the `owner` is set to our new `$user`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 158
    public function testOwnerCanSeeIsPublishedField(): void
    {
        $user = UserFactory::new()->create();
        $treasure = DragonTreasureFactory::createOne([
            'isPublished' => false,
            'owner' => $user,
        ]);

        $this->browser()
            ->actingAs($user)
            ->patch('/api/treasures/'.$treasure->getId(), [
                'json' => [
                    'value' => 12345,
                ],
            ])
            ->assertStatus(200)
            ->assertJsonMatches('value', 12345)
            ->assertJsonMatches('isPublished', false)
        ;
    }
}
```

We *could* change this to a GET request... but PATCH is fine. In either situation, we want to make sure the `isPublished` field is returned.

Since we haven't *implemented* this yet... let's make sure it fails. Copy the method name and try it:

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedField
```

Failure achieved! And we know how to solve this! On the `security` option, we *could* inline the logic with `or object.getOwner() === user`. But remember: we created the voter so that we don't need to do crazy stuff like that! Instead, say `is_granted()`, `EDIT` then `object`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 88
class DragonTreasure
{
// ... lines 91 - 129
    #[ApiProperty(security: 'is_granted("EDIT", object)')]
    private bool $isPublished = false;
// ... lines 132 - 250
}
```

Try the test now:

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedField
```

The Special securityPostDenormalize

Got it! Oh, and I haven't used it much, but there's also a `securityPostDenormalize` option. Just like with the `securityPostDenormalize` option on each operation, this runs *after* the new data is deserialized onto the object. What's interesting is that if the expression returns `false`, the data on the object is actually *reverted*.

For example, suppose the `isPublished` property started as `false` and then the user sent some JSON to change it to `true`. But then, `securityPostDenormalize` returned `false`. In that case, API Platform will *revert* the `isPublished` property *back* to its original value: it will change it from `false` *back* to `true`. Oh, and by the way, `securityPostDenormalize` is *not* executed on `GET` requests: it only happens when data is being deserialized. So be sure to put your main security logic in `security` and only use `securityPostDenormalize` if you need it.

Up next on our to-do list: let's level-up our user operations to *hash* the password before saving to the database. We'll need a fresh, non-persisted plain password property to make it happen.

Chapter 25: User Test + Plain Password

We have a pretty nice `DragonTreasureResourceTest`, so let's bootstrap one for User.

Bootstrapping the User Test

Create a new PHP class called, how about, `UserResourceTest`. Make it extend our custom `ApiTestCase`, then we just need to `use ResetDatabase`:

Tip

To use Foundry factories in a test, also add a `use Factories;` trait to the top of your test class. Things worked without that in this case, but in the future, you'll likely get an error.

tests/Functional/UserResourceTest.php

```php
// ... lines 1 - 2
namespace App\Tests\Functional;

use Zenstruck\Foundry\Test\ResetDatabase;

class UserResourceTest extends ApiTestCase
{
    use ResetDatabase;
// ... lines 10 - 14
}
```

We don't need `HasBrowser` because that's already done in the base class.

Start with `public function testPostToCreateUser()`:

```
tests/Functional/UserResourceTest.php
// ... lines 1 - 6
class UserResourceTest extends ApiTestCase
{
// ... lines 9 - 10
    public function testPostToCreateUser(): void
    {

    }
}
```

Make a `->post()` request to `/api/users`, toss in some `json` with `email` and `password`, and `assertStatus(201)`.

And now that we've created the new user, let's jump right in and test if we can log in with their credentials! Make another `->post()` request to `/login`, *also* pass some `json` - copy the `email` and `password` from above - then `assertSuccessful()`:

```
tests/Functional/UserResourceTest.php
// ... lines 1 - 6
class UserResourceTest extends ApiTestCase
{
// ... lines 9 - 10
    public function testPostToCreateUser(): void
    {
        $this->browser()
            ->post('/api/users', [
                'json' => [
                    'email' => 'draggin_in_the_morning@coffee.com',
                    'username' => 'draggin_in_the_morning',
                    'password' => 'password',
                ]
            ])
            ->assertStatus(201)
            ->post('/login', [
                'json' => [
                    'email' => 'draggin_in_the_morning@coffee.com',
                    'password' => 'password',
                ]
            ])
            ->assertSuccessful()
        ;
    }
}
```

Let's give this a go: `symfony php bin/phpunit` and run the entire `tests/Functional/UserResourceTest.php` file:

```
symfony php bin/phpunit tests/Functional/UserResourceTest.php
```

And... ok! A 422 status code, but 201 expected. Let's see: this means something went wrong creating the user. Let's pop open the last response. Ah! My bad: I forgot to pass the required `username` field: we're failing validation!

Pass `username`... set to anything:

```
tests/Functional/UserResourceTest.php
// ... lines 1 - 6
class UserResourceTest extends ApiTestCase
{
// ... lines 9 - 10
    public function testPostToCreateUser(): void
    {
        $this->browser()
            ->post('/api/users', [
                'json' => [
// ... line 16
                    'username' => 'draggin_in_the_morning',
// ... line 18
                ]
            ])
// ... lines 21 - 28
        ;
    }
}
```

Try that again:

```
symfony php bin/phpunit tests/Functional/UserResourceTest.php
```

That's what I wanted:

> *“Expected successful status code, but got 401.”*

So the failure is down here. We *were* able to create the user... but when we tried to log in, it failed. If you were with us for episode one, you might remember why! We never set up our API to *hash* the password.

Check it out: inside `User`, we *did* make `password` part of our API. The user sends the plain-text password they want... then we're saving that directly into the database. That's a *huge* security problem... and it makes it impossible to log in as this user, because Symfony expects the `password` property to hold a *hashed* password.

Setting up the plainPassword Field

So our goal is clear: allow the user to send a *plain* password, but then hash it before it's stored in the database. To do this, instead of temporarily storing the plain-text password on the `password` property, let's create a totally *new* property: `private ?string $plainPassword = null`:

```
src/Entity/User.php
// ... lines 1 - 66
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 69 - 92
    private ?string $plainPassword = null;
// ... lines 94 - 290
}
```

This will *not* be stored in the database: it's just a temporary spot to hold the plain password before we hash it and set that on the *real* `password` property.

Down at the bottom, I'll go to "Code"->"Generate", or `Command`+`N` on a Mac, and generate a "Getter and setter" for this. Let's clean this up a bit: accept only a string, and the PHPDoc is redundant:

```
src/Entity/User.php
// ... lines 1 - 66
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 69 - 279
    public function setPlainPassword(string $plainPassword): User
    {
        $this->plainPassword = $plainPassword;

        return $this;
    }

    public function getPlainPassword(): ?string
    {
        return $this->plainPassword;
    }
}
```

Next, scroll all the way to the top and find `password`. *Remove* this from our API entirely:

```
src/Entity/User.php
// ... lines 1 - 67
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 70 - 86
    /**
     * @var string The hashed password
     */
    #[ORM\Column]
    private ?string $password = null;
// ... lines 92 - 292
}
```

Instead, expose `plainPassword`... but use `SerializedName` so it's called `password`:

```
src/Entity/User.php
// ... lines 1 - 67
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 70 - 92
    #[Groups(['user:write'])]
    #[SerializedName('password')]
    private ?string $plainPassword = null;
// ... lines 96 - 292
}
```

So we're obviously not done yet... and if you run the tests:

```
symfony php bin/phpunit tests/Functional/UserResourceTest.php
```

Things are worse! A 500 error because of a not null violation. We're sending `password`, that's stored on `plainPassword`... then we're doing absolutely nothing with it. So the *real* `password` property stays null and explodes when it hits the database.

So here's the million-dollar question: how can we hash the `plainPassword` property? Or, in simpler terms, how can we run code in API Platform *after* the data is deserialized but *before* it's saved to the database? The answer is: *state processors*. Let's dive into this powerful concept next.

Chapter 26: State Processors: Hashing the User Password

When an API client creates a user, they send a `password` field, which gets set onto the `plainPassword` property. Now, we need to hash that password *before* the `User` is saved to the database. Like we showed when working with Foundry, hashing a password is simple: grab the `UserPasswordHasherInterface` service then call a method on it:

```
src/Factory/UserFactory.php
// ... lines 1 - 6
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
// ... lines 8 - 30
final class UserFactory extends ModelFactory
{
// ... lines 33 - 47
    public function __construct(
        private UserPasswordHasherInterface $passwordHasher
    )
    {
// ... line 52
    }
// ... lines 54 - 81
    protected function initialize(): self
    {
        return $this
            ->afterInstantiate(function(User $user): void {
                $user->setPassword($this->passwordHasher->hashPassword(
                    $user,
                    $user->getPassword()
                ));
            })
        ;
    }
// ... lines 93 - 97
}
```

But to pull this off, we need a "hook" in API platform: we need some way to run code after our data is deserialized onto the `User` object, but before it's saved.

In our tutorial about API platform 2, we used a Doctrine listener for this, which would still work. Though, it does some negatives, like being super magical - it's hard to debug if it doesn't work - and you need to do some weird stuff to make sure it runs when *editing* a user's password.

Hello State Processors

Fortunately, In API platform 3, we have a shiny new tool that we can leverage. It's called a state processor. And actually, our `User` class is *already* using a state processor!

Find the API Platform 2 to 3 upgrade guide... and search for processor. Let's see... here we go. It has a section called *providers and processors*. We'll talk about providers later.

According to this, if you have an `ApiResource` class that is an *entity* - like in our app - then, for example, your `Put` operation already uses a state processor called `PersistProcessor`! The `Post` operation also uses that, and `Delete` has one called `RemoveProcessor`.

State processors are cool. After the sent data is deserialized onto the object, we... need to do something! Most of the time, that "something" is: save the object to the database. And that's *precisely* what `PersistProcessor` does! Yea, our entity changes are saved to the database *entirely* thanks to that built-in state processor!

Creating the Custom State Processor

So here's the plan: we're going to hook into the state processor system and add our own. Step one, run a new command from API Platform:

```
php ./bin/console make:state-processor
```

Let's call it `UserHashPasswordProcessor`. Perfect.

Spin over, go into `src/`, open the new `State/` directory and check out `UserHashPasswordStateProcessor`:

```php
// src/State/UserHashPasswordStateProcessor.php
// ... lines 1 - 2
namespace App\State;

use ApiPlatform\Metadata\Operation;
use ApiPlatform\State\ProcessorInterface;

class UserHashPasswordStateProcessor implements ProcessorInterface
{
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        // Handle the state
    }
}
```

It's delightfully simple: API platform will call this method, pass us data, tell us which operation is happening... and a few other things. Then... we just do whatever we want. Send emails, save things to the database, or RickRoll someone watching a screencast!

Activating this processor is simple in theory. We could go to the `Post` operation, add a `processor` option and set it to our service id: `UserHashPasswordStateProcessor::class`.

Unfortunately... if we did that, it would *replace* the `PersistProcessor` that it's using now. And... we don't want that: we want our new processor to run... and *then* also the existing `PersistProcessor`. But... each operation can only have *one* processor.

Setting up Decoration

No worries! We can do this by *decorating* `PersistProcessor`. Decoration always follows the same pattern. First, add a constructor that accept an argument with the same interface as our class: `private ProcessorInterface` and I'll call it `$innerProcessor`:

```
src/State/UserHashPasswordStateProcessor.php
// ... lines 1 - 5
use ApiPlatform\State\ProcessorInterface;
// ... lines 7 - 9
class UserHashPasswordStateProcessor implements ProcessorInterface
{
    public function __construct(private ProcessorInterface
$innerProcessor)
    {
    }
// ... lines 15 - 21
}
```

After I add a `dump()` to see if this is working, we'll do step 2: call the decorated service method: `$this->innerProcessor->process()` passing `$data`, `$operation`, `$uriVariables` and... yes, `$context`:

Tip

In API Platform 3.2 and higher, you should `return $this->innerProcessor->process()`. This is also a safe thing to do in 3.0 & 3.1.

```
src/State/UserHashPasswordStateProcessor.php
// ... lines 1 - 9
class UserHashPasswordStateProcessor implements ProcessorInterface
{
// ... lines 12 - 15
    public function process(mixed $data, Operation $operation, array
$uriVariables = [], array $context = []): void
    {
        dump('ALIVE!');

        $this->innerProcessor->process($data, $operation, $uriVariables,
$context);
    }
}
```

Love it: our *class* is set up for decoration. *Now* we need to tell Symfony to *use* it. Internally, `PersistProcessor` from API Platform is a service. We're going to tell Symfony that whenever *anything* needs that `PersistProcessor` service, it should be passed *our* service instead... but also that Symfony should pass *us* the *original* `PersistProcessor`.

To do that, add `#[AsDecorator()]` and pass the id of the service. You can usually find this in the documentation, or you can use the `debug:container` command to search for it. The docs say it's `api_platform.doctrine.orm.state.persist_processor`:

```
src/State/UserHashPasswordStateProcessor.php
// ... lines 1 - 6
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;

#[AsDecorator('api_platform.doctrine.orm.state.persist_processor')]
class UserHashPasswordStateProcessor implements ProcessorInterface
{
// ... lines 12 - 21
}
```

Decoration done! We're not *doing* anything yet, but let's see if it hits our dump! Run the test:

```
symfony php bin/phpunit tests/Functional/UserResourceTest.php
```

And... there it is! It's still a 500, but it *is* using our processor!

Adding the Hashing Logic

Now we can get to work. Because of how we did the service decoration, our new processor will be called whenever *any* entity is processed... whether it's a `User`, `DragonTreasure` or something else. So, start by checking if `$data` is an `instanceof User`... *and* if `$data->getPlainPassword()`... because if we're editing a user, and no `password` is sent, no need for us to do anything:

```
src/State/UserHashPasswordStateProcessor.php
// ... lines 1 - 11
class UserHashPasswordStateProcessor implements ProcessorInterface
{
// ... lines 14 - 17
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        if ($data instanceof User && $data->getPlainPassword()) {
// ... line 21
        }

        $this->innerProcessor->process($data, $operation, $uriVariables, $context);
    }
}
```

By the way, the official documentation for decorating state processors is slightly different. It looks more complex to me, but the end result is a processor that's only called for one entity, not all of them.

To hash the password, add a second argument to the constructor: `private UserPasswordHasherInterface` called `$userPasswordHasher`:

```
src/State/UserHashPasswordStateProcessor.php
// ... lines 1 - 8
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
// ... lines 10 - 11
class UserHashPasswordStateProcessor implements ProcessorInterface
{
    public function __construct(private ProcessorInterface $innerProcessor, private UserPasswordHasherInterface $userPasswordHasher)
    {
    }
// ... lines 17 - 25
}
```

Below, say `$data->setPassword()` set to `$this->userPasswordHasher->hashPassword()` passing it the `User`, which is `$data` and the plain password: `$data->getPlainPassword()`:

```
src/State/UserHashPasswordStateProcessor.php
// ... lines 1 - 11
class UserHashPasswordStateProcessor implements ProcessorInterface
{
// ... lines 14 - 17
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        if ($data instanceof User && $data->getPlainPassword()) {
            $data->setPassword($this->userPasswordHasher->hashPassword($data, $data->getPlainPassword()));
        }

        $this->innerProcessor->process($data, $operation, $uriVariables, $context);
    }
}
```

And this all happens before we call the *inner* processor that actually saves the object.

Let's try this thing! Run that test:

```
symfony php bin/phpunit tests/Functional/UserResourceTest.php
```

Victory! After creating a user in our API, we *can* then log in as that user.

User.eraseCredentials()

Oh, and it's minor, but once you have a `plainPassword` property, inside of `User`, there's a method called `eraseCredentials()`. Uncomment `$this->plainPassword = null`:

```
src/Entity/User.php
// ... lines 1 - 67
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 70 - 186
    public function eraseCredentials()
    {
        // If you store any temporary, sensitive data on the user, clear it here
        $this->plainPassword = null;
    }
// ... lines 192 - 292
}
```

This makes sure that if the object is serialized into the session, the sensitive `plainPassword` is cleared first.

Next: let's fix some validation issues via `validationGroups` and discover something special about the `Patch` operation.

Chapter 27: Validation Groups & Patch Formats

Now that the `plainPassword` property is a legitimate part of our API, let's add some validation... because you can't create a new user without a password! Add `Assert\NotBlank`:

src/Entity/User.php

```php
// ... lines 1 - 67
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 70 - 94
    #[Assert\NotBlank]
    private ?string $plainPassword = null;
// ... lines 97 - 293
}
```

Piece of cake! Well, that just created a new problem... but let's blindly move forward and pretend that everything is fine.

Copy the first test and paste to create a second method that will make sure we can *update* users. Call it `testPatchToUpdateUser()`. This one is simple: make a new user - `$user = UserFactory::createOne()`, add `actingAs($user)` then `->patch()` to `/api/users/` then `$user->getId()` to edit ourselves.

For the `json`, just send `username`, add `assertStatus(200)`.... then we don't need any of this other stuff:

```
tests/Functional/UserResourceTest.php
// ... lines 1 - 7
class UserResourceTest extends ApiTestCase
{
// ... lines 10 - 32
    public function testPatchToUpdateUser(): void
    {
        $user = UserFactory::createOne();

        $this->browser()
            ->actingAs($user)
            ->patch('/api/users/' . $user->getId(), [
                'json' => [
                    'username' => 'changed',
                ],
            ])
            ->assertStatus(200);
    }
}
```

As a reminder, up on the `Patch` operation for `User`... here it is, we're requiring that the user has `ROLE_USER_EDIT`. Because we're logging in as a "full" user, we should have that... and everything should work fine... famous last words.

Run:

```
symfony php bin/phpunit --filter=testPatchToUpdateUser
```

PATCH: The Most Interesting HTTP Method in the World

And... oh! 200 expected, got 415. That's a new one! Click to open the last response... then I'll View Source to make it more clear. Interesting:

> *“The content-Type: `application/json` is not supported. Supported MIME types are `application/merge-patch+json`.”*

Let's unpack this. We're making a `PATCH` request... and `PATCH` requests are quite simple: we send a subset of fields, and only *those* fields are updated.

Whelp, it turns out that the `PATCH` HTTP method can get a whole heck of a lot more interesting than this. In the greater interwebs, there are competing *formats* for how the data should look when using a PATCH request and each format *means* something different.

Currently, API Platform supports only one of these formats: `application/merge-patch+json`. This format is... kind of what you expect. It says: if you send a single field, only that single field will be changed. But it also has other rules, like how you could set `email` to `null`... and that would actually *remove* the `email` field. That doesn't really make sense in our API, but the point is: the format defines rules about how your JSON should look for a `PATCH` request and what that means. If you want to know more, there's a document that describes everything: it's quite short and readable.

So, API platform only supports *one* format for PATCH requests at the moment. But, in the future, they might support more. And so, when you make a `PATCH` request, API Platform requires you to send a `Content-Type` header set to `application/merge-patch+json`... so that you're *explicitly* telling API platform *which* format your JSON is using.

In other words, to fix our error, pass a `headers` key with `Content-Type` set to `application/merge-patch+json`:

```
tests/Functional/UserResourceTest.php
// ... lines 1 - 7
class UserResourceTest extends ApiTestCase
{
// ... lines 10 - 32
    public function testPatchToUpdateUser(): void
    {
// ... lines 35 - 36
        $this->browser()
// ... line 38
            ->patch('/api/users/' . $user->getId(), [
// ... lines 40 - 42
                'headers' => ['Content-Type' => 'application/merge-patch+json']
            ])
// ... line 45
    }
}
```

Try this now:

```
symfony php bin/phpunit --filter=testPatchToUpdateUser
```

It *still* fails, but now it's a validation error! The takeaway is simple: PATCH requests require this `Content-Type` header.

But wait! We did a bunch of `PATCH` requests over in `DragonTreasureResourceTest` and those worked fine *without* the header! What the what?

That... was kind of on accident. Inside `DragonTreasure`, in the first tutorial... here it is, we added a `formats` key so that we could add CSV support:

src/Entity/DragonTreasure.php

```php
// ... lines 1 - 28
#[ApiResource(
// ... lines 30 - 49
    formats: [
        'jsonld',
        'json',
        'html',
        'jsonhal',
        'csv' => 'text/csv',
    ],
// ... lines 57 - 66
)]
// ... lines 68 - 252
```

It turns out that, for some complex internal reasons, by adding `formats`, we *removed* the requirement for needing that header. So we were "getting away" with *not* setting the header in `DragonTreasureResourceTest`... even though we *should* be setting it. It may have been better to set `formats` on the `GetCollection` operation only... since that's the only spot we need CSV.

Anyway, that's why we didn't need it before, but we *do* need it now. By the way, if adding this header every time you call `->patch` is annoying, this is another situation where you could add a custom method to browser - like `->apiPatch()` - which would work the same, but add that header automatically.

Fixing the Validation Groups

Ok, back to the test! It's failing with a 422. Open the error response. Ah, it's from `plainPassword`: this field should not be blank!

The `plainPassword` property is *not* persisted to the database. So, it's always empty at the start of an API request. When we create a `User`, we absolutely *do* want this field to be required. But when we're editing a `User`, we *don't* need this field to be set. They *can* set it in order to change their password, but that's optional.

This is the first spot where we need *conditional* validation: validation should happen on one operation, but not on others. The way to fix this is with validation groups, which is very similar to serialization groups.

Find the `Post` operation and pass a new option called `validationContext` with, you guessed it, `groups`! Set this to an array with a group called `Default` with a capital D. Then invent a second group: `postValidation`:

src/Entity/User.php

```
// ... lines 1 - 26
#[ApiResource(
// ... line 28
    operations: [
// ... lines 30 - 31
        new Post(
// ... line 33
            validationContext: ['groups' => ['Default', 'postValidation']],
        ),
// ... lines 36 - 42
    ],
// ... lines 44 - 49
)]
// ... lines 51 - 296
```

When the validator validates an object, by default, it validates everything that's in a group called `Default`. And any time you have a constraint, by default that constraint is *in* that `Default` group. So what we're saying here is:

> *"We want to validate all the normal constraints plus any constraints that are in the `postValidation` group."*

Now we can take that `postValidation`, go down to `plainPassword` and set `groups` to `postValidation`:

```
src/Entity/User.php
// ... lines 1 - 68
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 71 - 95
    #[Assert\NotBlank(groups: ['postValidation'])]
    private ?string $plainPassword = null;
// ... lines 98 - 294
}
```

That *removes* this constraint from the `Default` group and *only* includes it in the `postValidation` group. Thanks to this, other operations like `Patch` will *not* run this, but the `Post` operation *will*.

Run the test now:

```
symfony php bin/phpunit --filter=testPatchToUpdateUser
```

We're unstoppable! In fact, *all* of our tests are passing!

Careful: PUT Can Create Objects

But head's up! In `User`, we still have both `Put` and `Patch`. I haven't played with it much yet, but the new `Put` behavior, in theory, *does* support *creating* objects. This can make things tricky: do we need to require the password or not? It depends! This might be another reason for removing the `Put` operation to keep life simple. That gives us one operation for creating and one operation for editing.

Next: let's explore making our serialization groups *dynamic* based on the user. This will give us another way to include or not include fields based on who is logged in. And it'll lead us towards adding super custom fields.

Chapter 28: Dynamic Groups: Context Builder

In `DragonTreasure`, find the `$isPublished` field. Earlier we added this `ApiProperty` `security` thing so that the field is only returned for admin users or owners of this treasure. This is a simple and 100% valid way to handle this situation.

However, there *is* another way to handle fields that should be dynamic based on the current user... and it may or may not have two advantages depending on your situation.

The security Options vs Dynamic Groups

First, check out the documentation. Open the GET endpoint for a single `DragonTreasure`. And, even without trying it, you can see that `isPublished` *is* a field that is correctly advertised in our docs.

So, that's good, right? Yea! Well, probably. If `isPublished` were truly an internal, admin-only field, we might *not* want it advertised to the world.

The second possible problem with `security` is that, if you have this option on *many* properties, it's going to run that security check a *lot* of times when returning a collection of objects. Honestly, that probably *won't* cause performance issues, but it's something to be aware of.

Inventing New Serialization Groups

To solve these two possible problems - and, honestly, just to learn more about how API Platform works under the hood - I want to show you an alternative solution. Remove the `ApiProperty` attribute:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 88
class DragonTreasure
{
// ... lines 91 - 129
    #[ApiProperty(security: 'is_granted("EDIT", object)')]
    private bool $isPublished = false;
// ... lines 132 - 250
}
```

And replace it with two new groups. We're not going to use the normal `treasure:read` and `treasure:write`... because then the fields would *always* be part of our API. Instead, use `admin:read` and `admin:write`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 88
class DragonTreasure
{
// ... lines 91 - 128
    #[Groups(['admin:read', 'admin:write'])]
    private bool $isPublished = false;
// ... lines 131 - 249
}
```

This won't work yet... because these groups are *never* used. But here's the idea: if the current user is an admin, then when we serialize, we'll *add* these two groups.

The tricky part is, right now, groups are static! We set them way up here on the `ApiResource` attribute - or on a specific operation - and that's it! But we *can* make them dynamic.

Hello ContextBuilder

Internally, API Platform has a system called a context builder, which is responsible for building the normalization or denormalization contexts that are then passed into the serializer. *And*, we can hook *into* that to *change* the context: like to add extra groups.

Let's do it! Over in `src/ApiPlatform/`, create a new class called `AdminGroupsContextBuilder`... and make this implement `SerializerContextBuilderInterface`:

```
src/ApiPlatform/AdminGroupsContextBuilder.php
// ... lines 1 - 2
namespace App\ApiPlatform;

use ApiPlatform\Serializer\SerializerContextBuilderInterface;
// ... lines 6 - 7
class AdminGroupsContextBuilder implements
SerializerContextBuilderInterface
{
// ... lines 10 - 13
}
```

Then, go to "Code"->"Generate" - or `Command`+`N` on a Mac - and select "Implement methods" to create the one we need: `createFromRequest()`:

```
src/ApiPlatform/AdminGroupsContextBuilder.php
// ... lines 1 - 5
use Symfony\Component\HttpFoundation\Request;

class AdminGroupsContextBuilder implements
SerializerContextBuilderInterface
{
    public function createFromRequest(Request $request, bool
$normalization, array $extractedAttributes = null): array
    {
        // TODO: Implement createFromRequest() method.
    }
}
```

It's pretty simple: API Platform will call this, pass us the `Request`, whether or not we're normalizing or denormalizing... and then *we* return the `context` array that should be passed to the serializer.

Let's do some Decoration!

Like we've seen a few times already, our intention is *not* to *replace* the core context builder. Nope, we want the core context builder to do its thing... and *then* we'll add our own stuff.

To do this, once again, we'll use service decoration. We know how this works: add a `__construct()` method that accepts a private `SerializerContextBuilderInterface` and I'll call this `$decorated`:

```
src/ApiPlatform/AdminGroupsContextBuilder.php
// ... lines 1 - 7
class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
    public function __construct(private SerializerContextBuilderInterface $decorated)
    {
    }
// ... lines 13 - 20
}
```

Then, down here, say `$context = this->decorated->createFromRequest()` passing `$request`, `$normalization` and `$extractedAttributes`. Add a `dump()` to make sure this is working and return `$context`:

```
src/ApiPlatform/AdminGroupsContextBuilder.php
// ... lines 1 - 7
class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
// ... lines 10 - 13
    public function createFromRequest(Request $request, bool $normalization, array $extractedAttributes = null): array
    {
        $context = $this->decorated->createFromRequest($request, $normalization, $extractedAttributes);
        dump('I AM WORKING!');

        return $context;
    }
}
```

To tell Symfony to use *our* context builder in place of the real one, add our `#[AsDecorator()]`.

Here, we need the service ID of whatever the *core* context builder is. That's something you can find in the docs: it's `api_platform.serializer.context_builder`:

```
src/ApiPlatform/AdminGroupsContextBuilder.php
// ... lines 1 - 5
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;
// ... lines 7 - 8
#[AsDecorator('api_platform.serializer.context_builder')]
class AdminGroupsContextBuilder implements
SerializerContextBuilderInterface
{
// ... lines 12 - 22
}
```

Oh, but be careful when using `SerializerContextBuilderInterface`: there are *two* of them. One of is from GraphQL: make sure you select the one from `ApiPlatform\Serializer`, unless you *are* using GraphQL.

Ok! Let's see if it hits our dump! Run *all* of our tests: I also want to see which fail:

```
symfony php bin/phpunit
```

And... okay! We see the dump a *bunch* of times, followed by two failures. The first is `testAdminCanPatchToEditTreasure`. That's the case we're working on right now. We'll worry about `testOwnerCanSeeIsPublishedFieldI` in a minute.

Copy the test method name and rerun that with `--filter=`:

```
symfony php bin/phpunit --filter=testAdminCanPatchToEditTreasure
```

When the Context Builder is Called

Perfect! We see the dump: actually *three* times, which is interesting. Open up that test so we can see what's going on. Yup! We're making a *single* `PATCH` request to `/api/treasure/1`. So, the context builder is called 3 times during just one request?

It is! It's called one time when API Platform is querying and loading the `DragonTreasure` from the database. That's... kind of an odd situation because the context is meant to be used for the serializer... but we're simply querying for the object. But anyway, that's the first time.

The next two make sense: it's called when the JSON we're sending is denormalized into the object... and a third time when the final `DragonTreasure` is normalized back into JSON.

Anyway, let's hop in and add the dynamic groups. To determine if the user is an admin, add a second constructor argument - `private Security` from `SecurityBundle` called `$security`:

src/ApiPlatform/AdminGroupsContextBuilder.php

```php
// ... lines 1 - 5
use Symfony\Bundle\SecurityBundle\Security;
// ... lines 7 - 9
#[AsDecorator('api_platform.serializer.context_builder')]
class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
    public function __construct(private SerializerContextBuilderInterface $decorated, private Security $security)
    {
    }
// ... lines 16 - 26
}
```

Then down here, if `isset($context['groups'])` and `$this->security->isGranted('ROLE_ADMIN')`, then we'll add the groups: `$context['groups'][] =`. If we're currently normalizing, add `admin:read` else add `admin:write`:

```
src/ApiPlatform/AdminGroupsContextBuilder.php
// ... lines 1 - 10
class AdminGroupsContextBuilder implements SerializerContextBuilderInterface
{
// ... lines 13 - 16
    public function createFromRequest(Request $request, bool $normalization, array $extractedAttributes = null): array
    {
        $context = $this->decorated->createFromRequest($request, $normalization, $extractedAttributes);

        if (isset($context['groups']) && $this->security->isGranted('ROLE_ADMIN')) {
            $context['groups'][] = $normalization ? 'admin:read' : 'admin:write';
        }
// ... lines 24 - 25
    }
}
```

Now, you might be wondering why we're checking if `isset($context['groups'])`. Well, it doesn't apply to our app, but imagine if we were serializing an object that didn't have *any* `groups` on it - like we never set the `normalizationContext` on that `ApiResource`. In that case, adding these `groups` would cause it to return *less* fields! Remember, if there are *no* serialization groups, the serializer returns *every* accessible field. But as soon as you add even *one* group, it only serializes the things *in* that one group. So if there aren't any `groups`, do nothing and let *everything* be serialized or deserialized like normal.

Ok! Let's try the test now!

```
symfony php bin/phpunit --filter=testAdminCanPatchToEditTreasure
```

It passes! The `isPublished` field *is* being returned if we're an admin user. But... go refresh the docs... and open the GET one treasure endpoint. Now we do *not* see `isPublished` advertised as a field in our docs... even though it *will* be returned if we're an admin. That might be good or bad. It *is* possible to make the docs load dynamically based on *who* is logged in, but that's not something we're going to tackle in this tutorial. We *did* talk about that in our API platform 2 tutorial... but the config system has changed.

Let's dig into the next method, which tests that an *owner* can see the `isPublished` field. This is currently failing... and it's even trickier than the admin situation because we need to include or *not* include the `isPublished` field on an object-by-object basis.

Chapter 29: Custom Normalizer

Copy the test method - `testOwnerCanSeeIsPublishedField`. We just added some magic so that *admin* users can see the `isPublished` property. This method tests for our next mission: that *owners* of a `DragonTreasure` can *also* see this.

Run it with:

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedField
```

And... it fails: expected `null` to be the same as `false`, because the field isn't returned at all.

To fix this, over in `DragonTreasure`, add a third special group: `owner:read`:

src/Entity/DragonTreasure.php

```
// ... lines 1 - 88
class DragonTreasure
{
// ... lines 91 - 128
    #[Groups(['admin:read', 'admin:write', 'owner:read'])]
    private bool $isPublished = false;
// ... lines 131 - 249
}
```

Can you see where we're going with this? If we are the *owner* of a `DragonTreasure`, we'll add this group and then the field will be included. However, pulling this off is tricky.

As we talked about in the last video, normalization groups start *static*: they live up here in our config. The context builder allows us to make these groups dynamic *per request*. So, if we're an admin user, we can add an extra `admin:read` group, which will be used when serializing *every* object for this entire request.

But in this situation, we need to make the group dynamic per *object*. Imagine if we're returning 10 `DragonTreasure`'s: the user may only own *one* of them, so only that *one* `DragonTreasure` should be normalized using this extra group.

The Job of Normalizers

To handle *this* level of control, we need a custom normalizer. Normalizers are core to Symfony's serializer. They're responsible for turning a piece of data - like an `ApiResource` object or a `DateTime` object that lives on a property - into a scalar or array value. By creating a custom normalizer, you can do pretty much *any* weird thing you want!

Find your terminal and run:

```
php  bin/console debug:container --tag=serializer.normalizer
```

I love this: it shows us *every* single normalizer in our app! We can see stuff that's responsible for normalizing UUIDs.... this is what normalizes any of our `ApiResource` objects to `JSON-LD` and here's one for a `DateTime`. There's a *ton* of interesting stuff.

Our goal is to create our *own* normalizer, decorate an existing *core* normalizer, then add the dynamic group before that core normalizer is called.

Creating the Normalizer Class

So let's get to work! Over in `src/` - it doesn't really matter how we organize things - I'm going to create a new directory called `Normalizer`. Let me collapse a few things... so it's easier to look at. Inside that, add a new class called, how about, `AddOwnerGroupsNormalizer`. All normalizers must implement `NormalizerInterface`... then go to "Code"->"Generate" or `Command`+`N` on a Mac and select "Implement methods" to add the two we need:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 2
namespace App\Normalizer;

use Symfony\Component\Serializer\Normalizer\NormalizerInterface;

class AddOwnerGroupsNormalizer implements NormalizerInterface
{
    public function normalize(mixed $object, string $format = null, array
$context = [])
    {
        // TODO: Implement normalize() method.
    }

    public function supportsNormalization(mixed $data, string $format =
null)
    {
        // TODO: Implement supportsNormalization() method.
    }
}
```

Here's how this works: as soon as we implement `NormalizerInterface`, anytime *any* piece of data is being normalized, it will call our `supportsNormalization()` method. There, we can decide whether or not we know how to normalize that thing. If we return `true`, the serializer will then call `normalize()`, pass us that data, and then we return the normalized version.

And actually, to avoid some deprecation errors, pop open the parent class. The return type is this crazy array thingy. Copy that... and add it as the return type. You don't *have* to do this - everything would work without it - but you'd get a deprecation warning in your tests.

Down for `supportsNormalization()`, in Symfony 7, there will be an `array $context` argument... and the method will return a `bool`:

```php
// src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 6
class AddOwnerGroupsNormalizer implements NormalizerInterface
{
    public function normalize(mixed $object, string $format = null, array $context = []): array|string|int|float|bool|\ArrayObject|null
// ... lines 10 - 12

    public function supportsNormalization(mixed $data, string $format = null, array $context = []): bool
// ... lines 15 - 17
}
```

Which Service do We Decorate?

Before we fill this in or set up decoration, we need to think about *which* core service we're going to decorate. Here's my idea: if we replace the *main* core `normalizer` service with *this* class, we could add the group then call the decorated normalizer... so that everything then works like usual, except that it has the extra group.

Back at the terminal, run:

```
bin/console debug:container normalizer
```

We get back a *bunch* of results. That makes sense: there's a *main* `normalizer`, but then the `normalizer` itself has lots of *other* normalizers inside of it to handle different types of data. So... where is the top level normalizer? It's actually not even in this list: it called `serializer`. Though, as we'll see next, even *that* isn't quite right.

Chapter 30: Normalizer Decoration & "Normalizer Aware"

Our mission is clear: set up our normalizer to decorate Symfony's *core* normalizer service so that we can add the `owner:read` group when necessary and *then* call the decorated normalizer.

Setting up for Decoration

And we know decoration! Add `public function __construct()` with `private NormalizerInterface $normalizer`:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 4
use Symfony\Component\Serializer\Normalizer\NormalizerInterface;

class AddOwnerGroupsNormalizer implements NormalizerInterface
{
    public function __construct(private NormalizerInterface $normalizer)
    {
    }
// ... lines 12 - 23
}
```

Below in `normalize()`, add a `dump()` then `return $this->normalizer->normalize()` passing `$object` `$format`, and `$context`. For `supportsNormalization()`, do the same thing: call `supportsNormalization()` on the decorated class and pass the args:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 6
class AddOwnerGroupsNormalizer implements NormalizerInterface
{
// ... lines 9 - 12
    public function normalize(mixed $object, string $format = null, array
$context = []): array|string|int|float|bool|\ArrayObject|null
    {
        dump('IT WORKS!');

        return $this->normalizer->normalize($object, $format, $context);
    }

    public function supportsNormalization(mixed $data, string $format =
null, array $context = []): bool
    {
        return $this->normalizer->supportsNormalization($data, $format);
    }
}
```

To complete decoration, head to the top of the class. I'll remove a few old `use` statements... then say `#[AsDecorator]` passing `serializer`, which I mentioned is the service id for the top-level main normalizer:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 4
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;
// ... lines 6 - 7
#[AsDecorator('serializer')]
class AddOwnerGroupsNormalizer implements NormalizerInterface
{
// ... lines 11 - 25
}
```

Ok! We haven't made any changes yet... so we should still see the one failing test. Try it:

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedField
```

Woh! An explosion! Wow.

> *"`ValidationExceptionListener::__construct()`: Argument #1 (`$serializer`) must be of type `SerializerInterface`, `AddOwnerGroupsNormalizer` given."*

Okay? When we add `#[AsDecorator('serializer')]`, it means that our service *replaces* the service known as `serializer`. So, everyone that's depending on the `serializer` service will now be passed *us*... and then the original `serializer` is passed to *our* constructor.

So, what's the problem? Decoration has worked several times before. The problem is that the `serializer` service in Symfony is... kind of big. It implements `NormalizerInterface`, but also `DenormalizerInterface`, `EncoderInterface`, `DecoderInterface` and `SerializerInterface`! But our object only implements *one* of these . And so, when our class is passed to something that expects an object with one of those *other* 4 interfaces, it explodes.

If we truly wanted to decorate the `serializer` service, we would need to implement all *five* of those interfaces... which is just a ugly and too much. And that's fine!

Decorating a Lower-Level Normalizer

Instead of decorating the *top* level `normalizer`, let's decorate one *specific* normalizer: the one that's responsible for normalizing `ApiResource` objects into `JSON-LD`. This is another spot where you can rely on the documentation to give you the exact service ID you need. It's `api_platform.jsonld.normalizer.item`:

src/Normalizer/AddOwnerGroupsNormalizer.php

```
// ... lines 1 - 4
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;
// ... lines 6 - 7
#[AsDecorator('api_platform.jsonld.normalizer.item')]
class AddOwnerGroupsNormalizer implements NormalizerInterface
{
// ... lines 11 - 25
}
```

Try the test again: `testOwnerCanSeeIsPublishedField`

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedField
```

Yes! We see our dump! And... a 400 error? Let me pop open the response so we can see it. Strange:

> *"The injected serializer must be an instance of `NormalizerInterface`."*

And it's coming from deep inside of API Platform's serializer code. So... decorating normalizers is *not* a very friendly process. It's well-documented, but weird. When you decorate this specific normalizer, you also need to implement `SerializerAwareInterface`. And that's going to require you to have a `setSerializer()` method. Oh, let me import that `use` statement: I don't know why that didn't come automatically:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 6
use Symfony\Component\Serializer\SerializerAwareInterface;
// ... lines 8 - 10
class AddOwnerGroupsNormalizer implements NormalizerInterface, SerializerAwareInterface
{
// ... lines 13 - 28
    public function setSerializer(SerializerInterface $serializer)
    {
// ... lines 31 - 33
    }
}
```

There we go.

Inside, say, if `$this->normalizer` is an `instanceof SerializerAwareInterface`, then call `$this->normalizer->setSerializer($serializer)`:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 10
class AddOwnerGroupsNormalizer implements NormalizerInterface, SerializerAwareInterface
{
// ... lines 13 - 28
    public function setSerializer(SerializerInterface $serializer)
    {
        if ($this->normalizer instanceof SerializerAwareInterface) {
            $this->normalizer->setSerializer($serializer);
        }
    }
}
```

I don't even want to get into the details of this: it just happens that the normalizer we're decorating implements another interface... so we need to *also* implement it.

Let's try this again.

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedField
```

Finally, we have the dump *and* it's failing the assertion we expect... since we haven't added the group yet. Let's do that!

Adding the Dynamic Group

Remember the goal: if we own this `DragonTreasure`, we want to add the `owner:read` group. On the constructor, autowire the `Security` service as a property:

```php
// src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 5
use Symfony\Bundle\SecurityBundle\Security;
// ... lines 7 - 12
class AddOwnerGroupsNormalizer implements NormalizerInterface, SerializerAwareInterface
{
    public function __construct(private NormalizerInterface $normalizer, private Security $security)
    {
    }
// ... lines 18 - 38
}
```

Then, down here, if `$object` is an `instanceof DragonTreasure` - because this method will be called for *all* of our API resource classes - *and* `$this->security->getUser()` equals `$object->getOwner()`, then call `$context['groups'][]` to add `owner:read`:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 4
use App\Entity\DragonTreasure;
// ... lines 6 - 12
class AddOwnerGroupsNormalizer implements NormalizerInterface, SerializerAwareInterface
{
// ... lines 15 - 18
    public function normalize(mixed $object, string $format = null, array $context = []): array|string|int|float|bool|\ArrayObject|null
    {
        if ($object instanceof DragonTreasure && $this->security->getUser() === $object->getOwner()) {
            $context['groups'][] = 'owner:read';
        }
// ... lines 24 - 25
    }
// ... lines 27 - 38
}
```

Phew! Try that test one more time:

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedField
```

We got it! We can now return different fields on an object-by-object basis.

Also Decorating the Denormalizer

If you want to *also* add `owner:write` during *denormalization*, you would need to implement a second interface. I'm not going to do the whole thing... but you would implement `DenormalizerInterface`, add the two methods needed, call the decorated service... and change the argument to be a union type of `NormalizerInterface` *and* `DenormalizerInterface`.

Finally, the service that you're decorating for denormalization is different: it's `api_platform.serializer.normalizer.item`. However, if you want to decorate *both* the normalizer and denormalizer in the same class, you'd need to remove `#[AsDecorator]` and move the decoration config to `services.yaml`... because a single service can't decorate two things at once. API Platform covers that in their docs.

Ok, I'm going to undo all of that... and just stick with adding `owner:read`. Next: now that we have a custom normalizer, we can easily do wacky things like adding a *totally* custom field to our API that doesn't exist in our class.

Chapter 31: Totally Custom Fields

Let's get wild. I want to add a totally custom, crazy new field to our `DragonTreasure` API that does *not* correspond to any property in our class. Well, actually, we learned in part 1 of this series that adding custom fields is possible by creating a getter method and adding a serialization group above it. *But*, that solution only works if we can calculate the field's value solely from the data on the object. If, for example, we need to call a *service* to get the data, then we're out of luck.

Adding a new field whose data is calculated from a service is another trick up the custom normalizer's sleeve. And since we already have one set up, I thought we'd use it to see how this works.

Testing for the IsMe Field

Go to `DragonTreasureResourceTest` and find `testOwnerCanSeeIsPublishedField()`. Rename this to `testOwnerCanSeeIsPublishedAndIsMineFields()`:

tests/Functional/DragonTreasureResourceTest.php

```
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 158
    public function testOwnerCanSeeIsPublishedAndIsMineFields(): void
    {
// ... lines 161 - 178
    }
}
```

This is a bit silly, but if we own a `DragonTreasure`, we're going to add a new boolean property called `$isMine` set to `true`. So, down at the bottom, we'll say `isMine` and expect it to be `true`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 158
    public function testOwnerCanSeeIsPublishedAndIsMineFields(): void
    {
// ... lines 161 - 166
        $this->browser()
// ... lines 168 - 175
            ->assertJsonMatches('isPublished', false)
            ->assertJsonMatches('isMine', true)
        ;
    }
}
```

Copy that method name, then spin over and run this test:

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedAndIsMineFields
```

Tada! It's `null` because the field doesn't exist yet.

Returning the Custom Field

So how can we add this? Now that we've gone through the pain of getting the normalizer set up, it's easy! The normalizer system will do its thing, return the normalized data, then, between that and the `return` statement, we can... just mess with it!

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 12
class AddOwnerGroupsNormalizer implements NormalizerInterface,
SerializerAwareInterface
{
// ... lines 15 - 18
    public function normalize(mixed $object, string $format = null, array
$context = []): array|string|int|float|bool|\ArrayObject|null
    {
        if ($object instanceof DragonTreasure && $this->security-
>getUser() === $object->getOwner()) {
            $context['groups'][] = 'owner:read';
        }

        $normalized = $this->normalizer->normalize($object, $format,
$context);
// ... lines 26 - 30
        return $normalized;
    }
// ... lines 33 - 44
}
```

Copy the if statement from up here. I could be more clever and reuse code, but it's fine. If the object is a `DragonTreasure` and we own this `DragonTreasure`, we will say `$normalized['isMine'] = true`:

```
src/Normalizer/AddOwnerGroupsNormalizer.php
// ... lines 1 - 12
class AddOwnerGroupsNormalizer implements NormalizerInterface, SerializerAwareInterface
{
// ... lines 15 - 18
    public function normalize(mixed $object, string $format = null, array $context = []): array|string|int|float|bool|\ArrayObject|null
    {
// ... lines 21 - 24
        $normalized = $this->normalizer->normalize($object, $format, $context);

        if ($object instanceof DragonTreasure && $this->security->getUser() === $object->getOwner()) {
            $normalized['isMine'] = true;
        }

        return $normalized;
    }
// ... lines 33 - 44
}
```

That's it! When we run the test:

```
symfony php bin/phpunit --filter=testOwnerCanSeeIsPublishedAndIsMineFields
```

All green!

Custom Fields Missing in the Docs

But there's a practical downside to these custom fields: they will *not* be documented in our API. Our API docs have *no* idea that this exists!

If you *do* need a super-duper custom field that requires service logic... and you *do* need it to be documented, you have two options. First, you could add a non-persisted `isMe` property to your class then populate it with a state provider. We haven't talked about state providers yet, but they're how data is loaded. For example, our classes are *already* using a *Doctrine* state provider behind the scenes to query the database. We'll cover state providers in part 3 of this series.

The second solution would be to use the custom normalizer like we did, then try to add the field to the OpenAPI docs manually via the OpenAPI factory trick that we showed earlier.

Next: suppose a user *is* allowed to edit something... but there are certain changes to the data that they are *not* allowed to make - like they could set a field to `foo` but they aren't allowed to change it to `bar` because they don't have enough permissions. How should we handle that? It's security meets validation.

Chapter 32: Custom Validator

If you need to control *how* a field like `isPublished` is *set* based on *who* is logged in, you have two different situations.

Protecting a Field vs Protecting its Data

First, if you need to prevent certain users from writing to this field *entirely*, that's what security is for. The easiest option is to use the `#[ApiProperty(security: ...)]` option that we used earlier above the property. Or you could get fancier and add a dynamic `admin:write` group via a context builder. Either way, we're preventing this field from being written *entirely*.

The second situation is when a user *should* be allowed to write to a field... but the valid data they're allowed to *set* depends on who they are. Like maybe a user is allowed to set `isPublished` to `false`... but they're not allowed to set it to `true` unless they're an admin.

Let me give you a different example. Right now, when you create a `DragonTreasure`, we force the client to pass an `owner`. We can see this in `testPostToCreateTreasure()`. We're going to fix this in a few minutes so that we can leave this field *off*... and then it'll be set automatically to whoever is authenticated.

But right now, the `owner` field is allowed and required. But *who* they are allowed to *assign* as the `owner` depends on who is logged in. For normal users, they should only be allowed to assign *themselves* as a user. But for admins, they should be able to assign *anyone* as the `owner`. Heck, maybe in the future we get crazier and there are clans of dragons... and you can create treasures and assign them to anyone in your clan The point is: the question isn't *if* we can set this field, but *what* data we're *allowed* to set it to. And that depends on *who* we are.

Solving with Security or Validation?

Ok, actually, we solved this problem earlier for the `Patch()` operation. Let me show you. Find `testPatchToUpdateTreasure()`. Then... let's run just that test:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

And... it passes. This test checks 3 things. First, we log in as the user that owns the `DragonTreasure` and make an update. That's the happy case!

Next, we log in as a *different* user and try to edit the first user's `DragonTreasure`. That is *not* allowed. And *that* is a proper use of `security`: we don't own this `DragonTreasure`, so we are not *at all* allowed to edit it. That's what the `security` line is protecting.

For the last part, we log in again as the owner of this `DragonTreasure`. But then we try to change the owner to someone else. That's also *not* allowed and *this* is the situation we're talking about. It's currently handled by `securityPostDenormalize()`. But I want to handle it instead with *validation*. Why? Because the question we're answering is this:

> *“Is the `owner` data that's sent valid?”*

And... validating data is... the job of validation!

Remove the `securityPostDenormalize()`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 28
#[ApiResource(
// ... lines 30 - 31
    operations: [
// ... lines 33 - 41
        new Patch(
// ... line 43
            securityPostDenormalize: 'is_granted("EDIT", object)',
        ),
// ... lines 46 - 48
    ],
// ... lines 50 - 66
)]
// ... lines 68 - 88
class DragonTreasure
{
// ... lines 91 - 249
}
```

And to prove this was important, run the test again:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

Yup! It failed on line 132... which is this one down here. Let's rewrite this with a custom validator, which is actually a lot nicer.

Creating the Custom Validation

Oh but because this will fail via validation when we're done, change to `assertStatus(422)`:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 97
    public function testPatchToUpdateTreasure()
    {
// ... lines 100 - 126
        $this->browser()
// ... lines 128 - 134
            ->assertStatus(422)
        ;
    }
// ... lines 138 - 179
}
```

The idea is that we *are* allowed to PATCH this user, but we sent invalid data: we can't set this owner to someone *other* than ourselves.

Ok, head to the command line and run:

```
php ./bin/console make:validator
```

Give it a cool name like `IsValidOwnerValidator`. In Symfony, validators are *two* different classes. Open `src/Validator/IsValidOwner.php` first:

```
src/Validator/IsValidOwner.php
// ... lines 1 - 2
namespace App\Validator;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 *
 * @Target({"PROPERTY", "METHOD", "ANNOTATION"})
 */
#[\Attribute(\Attribute::TARGET_PROPERTY | \Attribute::TARGET_METHOD |
\Attribute::IS_REPEATABLE)]
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.';
}
```

This lightweight class will be used as the *attribute*... and it just holds options that we can configure, like `$message`, which is enough. Let's change the default message to something a bit more helpful:

```
src/Validator/IsValidOwner.php
// ... lines 1 - 12
class IsValidOwner extends Constraint
{
// ... lines 15 - 18
    public string $message = 'You are not allowed to set the owner to this
value.';
}
```

The second class is the one that will be executed to handle the logic:

```
src/Validator/IsValidOwnerValidator.php
// ... lines 1 - 2
namespace App\Validator;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class IsValidOwnerValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        /* @var App\Validator\IsValidOwner $constraint */

        if (null === $value || '' === $value) {
            return;
        }

        // TODO: implement the validation here
        $this->context->buildViolation($constraint->message)
            ->setParameter('{{ value }}', $value)
            ->addViolation();
    }
}
```

We'll look at that in a moment... but let's *use* the new constraint first. Over in `DragonTreasure`, down on the `owner` property... there we go... add the new attribute: `IsValidOwner`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 19
use App\Validator\IsValidOwner;
// ... lines 21 - 88
class DragonTreasure
{
// ... lines 91 - 135
    #[IsValidOwner]
// ... line 137
    private ?User $owner = null;
// ... lines 139 - 250
}
```

Filling in the Validator Logic

Now that we have this, when our object is validated, Symfony will call `IsValidOwnerValidator` and pass us the `$value` - which will be the `User` object - and the constraint, which will be `IsValidOwner`.

Let's do some clean up. Remove the `var` and replace it with `assert($constraint instanceof IsValidOwner)`:

src/Validator/IsValidOwnerValidator.php

```php
// ... lines 1 - 8
class IsValidOwnerValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        assert($constraint instanceof IsValidOwner);

        if (null === $value || '' === $value) {
            return;
        }
// ... lines 18 - 23
    }
}
```

That's just to help my editor: we know that Symfony will always pass us that. Next, notice that it's checking to see if the `$value` is null or blank. And if is, it does nothing. If the `$owner` property is empty, that should really be handled by a *different* constraint.

Back in `DragonTreasure`, add `#[Assert\NotNull]`:

src/Entity/DragonTreasure.php

```php
// ... lines 1 - 88
class DragonTreasure
{
// ... lines 91 - 136
    #[Assert\NotNull]
// ... line 138
    private ?User $owner = null;
// ... lines 140 - 251
}
```

So if they forget to send `owner`, *this* will handle that validation error. Back inside *our* validator, if we have that situation, we can just return:

```
src/Validator/IsValidOwnerValidator.php
// ... lines 1 - 8
class IsValidOwnerValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
// ... lines 13 - 14
        if (null === $value || '' === $value) {
            return;
        }
// ... lines 18 - 23
    }
}
```

Below this, add one more `assert()` that `$value` is an `instanceof User`.

Really, Symfony will pass us whatever value is attached to this property... but *we* know that this will *always* be a `User`:

```
src/Validator/IsValidOwnerValidator.php
// ... lines 1 - 8
class IsValidOwnerValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
// ... lines 13 - 14
        if (null === $value || '' === $value) {
            return;
        }

        // constraint is only meant to be used above a User property
        assert($value instanceof User);
// ... lines 21 - 23
    }
}
```

Finally, delete `setParameter()` - that's not needed in our case - and `$constraint->message` is reading the `$message` property:

```
src/Validator/IsValidOwnerValidator.php
// ... lines 1 - 8
class IsValidOwnerValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        assert($constraint instanceof IsValidOwner);

        if (null === $value || '' === $value) {
            return;
        }

        // constraint is only meant to be used above a User property
        assert($value instanceof User);

        $this->context->buildViolation($constraint->message)
            ->addViolation();
    }
}
```

At this point, we have a functional validator! Except... it's going to fail in all situations. Ah, let's at least make sure it's being called. Run our test:

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

Beautiful failure! A 422 coming from `DragonTreasureResourceTest` line 110... because our constraint is *never* satisfied.

Checking for Ownership in the Validator

Finally we can add our business logic. To do the owner check, we need to know who's logged in. Add a `__construct()` method, autowire our favorite `Security` class... and I'll put `private` in front of that, so it becomes a property:

```
src/Validator/IsValidOwnerValidator.php
// ... lines 1 - 5
use Symfony\Bundle\SecurityBundle\Security;
// ... lines 7 - 9
class IsValidOwnerValidator extends ConstraintValidator
{
    public function __construct(private Security $security)
    {
    }
// ... lines 15 - 34
}
```

Below, set `$user = $this->security->getUser()`. And if there is *no* user for some reason, throw a `LogicException` to make things explode:

```
src/Validator/IsValidOwnerValidator.php
// ... lines 1 - 9
class IsValidOwnerValidator extends ConstraintValidator
{
// ... lines 12 - 15
    public function validate($value, Constraint $constraint)
    {
// ... lines 18 - 23
        // constraint is only meant to be used above a User property
        assert($value instanceof User);

        $user = $this->security->getUser();
        if (!$user) {
            throw new \LogicException('IsOwnerValidator should only be used when a user is logged in.');
        }
// ... lines 31 - 33
    }
}
```

Why not trigger a validation error? We could... but in our app, if an anonymous user is somehow successfully *changing* a `DragonTreasure`... we have some sort of misconfiguration.

Finally, if `$value` does not equal `$user` - so if the `owner` is *not* the `User` - add that validation failure:

src/Validator/IsValidOwnerValidator.php

```php
// ... lines 1 - 9
class IsValidOwnerValidator extends ConstraintValidator
{
// ... lines 12 - 15
    public function validate($value, Constraint $constraint)
    {
// ... lines 18 - 31
        if ($value !== $user) {
            $this->context->buildViolation($constraint->message)
                ->addViolation();
        }
    }
}
```

That's it! Let's try this thing!

```
symfony php bin/phpunit --filter=testPatchToUpdateTreasure
```

And... bingo! Whether we're creating or editing a `DragonTreasure`, we are not allowed to set the owner to someone that is *not* us.

And we can add whatever other fanciness we want. Like if the user is an admin, return so that admin users are allowed to assign the `owner` to *anyone*:

```
src/Validator/IsValidOwnerValidator.php
// ... lines 1 - 9
class IsValidOwnerValidator extends ConstraintValidator
{
// ... lines 12 - 15
    public function validate($value, Constraint $constraint)
    {
// ... lines 18 - 26
        $user = $this->security->getUser();
        if (!$user) {
            throw new \LogicException('IsOwnerValidator should only be used when a user is logged in.');
        }

        if ($this->security->isGranted('ROLE_ADMIN')) {
            return;
        }

        if ($value !== $user) {
            $this->context->buildViolation($constraint->message)
                ->addViolation();
        }
    }
}
```

I love this. But... there's still one big security hole: a hole that will allow a user to *steal* the treasures of someone else! Not cool! Let's find out what that is next and crush it.

Chapter 33: Validating how Values Change

We still have a massive problem making sure treasures don't end up stolen! We just covered the main case: if you make a POST or a PUT request to a treasure endpoint, thanks to our new validation, we make sure you assign the owner to yourself, unless you're an admin. Yay!

But in our API, when POSTing or PATCHing to a *user* endpoint, you are allowed to send a `dragonTreasures` field. This, unfortunately allows treasures to be stolen. Simply send a `PATCH` request to modify your *own* `User` record... then set the `dragonTreasures` field to an array containing the IRI strings of some treasures that you do *not* own. Whoops!

The easiest solution would be to... make the field *not* writable. So, inside of `User`, for `dragonTreasures`, we would keep this *readable*, but remove the write group. That would force everyone to use the `/api/treasures` endpoints to manage their treasures.

The Trickiness of this Problem

If you *do* want to keep the writable `dragonTreasures` field... you can, but this problem *is* tricky to solve.

Let's think: if you send a `dragonTreasures` field that contains the IRI of a treasure you do *not* own, that should trigger a validation error. Ok... so maybe we add a validation constraint above this property? The problem is that, by the time that validation runs, the treasures sent over in the JSON have *already* been set onto this `dragonTreasures` property. And importantly, the `owner` on those treasures has already been updated to *this* `User`!

Remember: when the serializer sees a `DragonTreasure` that is not already owned by this user, it will call `addDragonTreasure()`... which then calls `setOwner($this)`. So, by the time validation runs, it's going to look like we *are* the owner of the treasure... even though we originally weren't!

Using Previous Data?

What can we do? Well, API Platform *does* have a concept of "previous data". API Platform *clones* the data before deserializing the new JSON onto it, which means it *is* possible to get what the `User` object *originally* looked like.

Unfortunately, that clone is *shallow*, meaning that it clones scalar fields - like `username` - but any objects - like the `DragonTreasure` objects are *not* cloned. There's no way via API Platform to see what they originally looked like.

Testing for the Bug

So, we *are* going to solve this with validation... but with the help of a special class from Doctrine called the `UnitOfWork`.

Alrighty, let's whip up a test to shine a light on this pesky bug. Inside `tests/Functional/`, open `UserResourceTest`. Copy the previous test, paste, and call it `testTreasuresCannotBeStolen()`. Create a second user with `UserFactory::createOne()`... and we need a `DragonTreasure` that we're going to try to steal. Assign its `owner` to `$otherUser`:

tests/Functional/UserResourceTest.php

```
// ... lines 1 - 4
use App\Factory\DragonTreasureFactory;
// ... lines 6 - 8
class UserResourceTest extends ApiTestCase
{
// ... lines 11 - 48
    public function testTreasuresCannotBeStolen(): void
    {
        $user = UserFactory::createOne();
        $otherUser = UserFactory::createOne();
        $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
$otherUser]);
// ... lines 54 - 66
    }
}
```

Let's do this! We log in as `$user`, update ourselves - which is allowed - then, for the JSON, sure, maybe we still send `username`... but we also send `dragonTreasures` set to an array with `/api/treasures/` and `$dragonTreasure->getId()`.

At the bottom, assert that this returns a 422:

```
tests/Functional/UserResourceTest.php
// ... lines 1 - 4
use App\Factory\DragonTreasureFactory;
// ... lines 6 - 8
class UserResourceTest extends ApiTestCase
{
// ... lines 11 - 48
    public function testTreasuresCannotBeStolen(): void
    {
        $user = UserFactory::createOne();
        $otherUser = UserFactory::createOne();
        $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
$otherUser]);

        $this->browser()
            ->actingAs($user)
            ->patch('/api/users/' . $user->getId(), [
                'json' => [
                    'username' => 'changed',
                    'dragonTreasures' => [
                        '/api/treasures/' . $dragonTreasure->getId(),
                    ],
                ],
                'headers' => ['Content-Type' => 'application/merge-
patch+json']
            ])
            ->assertStatus(422);
    }
}
```

Ok! Copy the method name. We're expecting this to fail:

```
symfony php bin/phpunit --filter=testTreasuresCannotBeStolen
```

And... it does! Status code 200, which means we *are* allowing treasure to be stolen! Gasp!

Creating the Validator

Ok, let's cook up a new validator class:

```
php ./bin/console make:validator
```

Call it `TreasuresAllowedOwnerChange`.

Go use this immediately. Above the `dragonTreasures` property, add `#[TreasuresAllowedOwnerChange]`:

```
src/Entity/User.php
// ... lines 1 - 15
use App\Validator\TreasuresAllowedOwnerChange;
// ... lines 17 - 69
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 72 - 107
    #[TreasuresAllowedOwnerChange]
    private Collection $dragonTreasures;
// ... lines 110 - 296
}
```

Next, over in `src/Validator/`, open up the validator class. We'll do some basic cleanup: use the `assert()` function to assert that `$constraint` is an instance of `TreasuresAllowedOwnerChange`. And also assert that `value` is an instance of `Collection` from Doctrine:

```
src/Validator/TreasuresAllowedOwnerChangeValidator.php
// ... lines 1 - 8
class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        assert($constraint instanceof TreasuresAllowedOwnerChange);

        if (null === $value || '' === $value) {
            return;
        }

        // meant to be used above a Collection field
        assert($value instanceof Collection);
// ... lines 21 - 25
    }
}
```

We know that this will be used above this property... so it will be some sort of collection of `DragonTreasures`.

Enter UnitOfWork

But... this will be the collection of `DragonTreasure` objects *after* they've been modified. We need to ask Doctrine what each `DragonTreasure` looked like when it was *originally* queried from the database. To do that, we need to grab an internal object from Doctrine called the `UnitOfWork`.

On top, add a constructor, autowire `EntityManagerInterface $entityManager`... and make that's a private property:

```
src/Validator/TreasuresAllowedOwnerChangeValidator.php
// ... lines 1 - 6
use Doctrine\ORM\EntityManagerInterface;
// ... lines 8 - 10
class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
{
    public function __construct(private EntityManagerInterface $entityManager)
    {
    }
// ... lines 16 - 40
}
```

Below, grab the unit of work with `$unitOfWork = $this->entityManager->getUnitOfWork()`:

```php
// src/Validator/TreasuresAllowedOwnerChangeValidator.php
// ... lines 1 - 10
class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
{
// ... lines 13 - 16
    public function validate($value, Constraint $constraint)
    {
// ... lines 19 - 24
        // meant to be used above a Collection field
        assert($value instanceof Collection);

        $unitOfWork = $this->entityManager->getUnitOfWork();
// ... lines 29 - 39
    }
}
```

This is a powerful object that keeps track of *how* entity objects are changing and is responsible for knowing which objects need to be inserted, updated or deleted from the database when the entity manager flushes.

Next, `foreach` over `$value` - which will be a collection - `as $dragonTreasure`. To help my editor, I'll assert that `$dragonTreasure` is an instance of `DragonTreasure`. And *now*, get the original data: `$originalData = $unitOfWork->getOriginalEntityData($dragonTreasure)`.

Pretty sweet right? Let's `dd($dragonTreasure)` and `$originalData` so we can see what they look like:

```php
src/Validator/TreasuresAllowedOwnerChangeValidator.php
// ... lines 1 - 10
class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
{
// ... lines 13 - 16
    public function validate($value, Constraint $constraint)
    {
// ... lines 19 - 27
        $unitOfWork = $this->entityManager->getUnitOfWork();
        foreach ($value as $dragonTreasure) {
            assert($dragonTreasure instanceof DragonTreasure);

            $originalData = $unitOfWork->getOriginalEntityData($dragonTreasure);
            dd($dragonTreasure, $originalData);
        }
// ... lines 35 - 39
    }
}
```

Go test go:

```
symfony php bin/phpunit --filter=testTreasuresCannotBeStolen
```

Yes! It hit the dump! And this is cool! The first part is the *updated* `DragonTreasure` object and its owner has ID 1. It's not super obvious, but `$user` will be id 1 and `$otherUser` will be id 2. So the owner was originally ID 2, but yeah: user id 1 has stolen it! Below this, we see the *original* data as an array. And its owner was ID 2!

This info makes us dangerous. Back inside our validator, say `$originalOwnerId` = `originalData['owner_id']`. And to be super clear, set `$newOwnerId` to `$dragonTreasure->getOwner()->getId()`.

If these don't match, we have a problem. Well actually, if we don't have an `$originalOwnerId`, we're creating a *new* `DragonTreasure` and that's ok. So if there is no `$originalOwnerId` or the `$originalOwnerId` is equal to the `$newOwnerId`, we're good!

Else... there's some plundering happening! Move the `$violationBuilder` up, but remove `setParameter()`:

```
src/Validator/TreasuresAllowedOwnerChangeValidator.php
// ... lines 1 - 10
class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
{
// ... lines 13 - 16
    public function validate($value, Constraint $constraint)
    {
// ... lines 19 - 27
        $unitOfWork = $this->entityManager->getUnitOfWork();
        foreach ($value as $dragonTreasure) {
            assert($dragonTreasure instanceof DragonTreasure);

            $originalData = $unitOfWork->getOriginalEntityData($dragonTreasure);
            $originalOwnerId = $originalData['owner_id'];
            $newOwnerId = $dragonTreasure->getOwner()->getId();

            if (!$originalOwnerId || $originalOwnerId === $newOwnerId) {
                return;
            }

            // the owner is being changed
            $this->context->buildViolation($constraint->message)
                ->addViolation();
        }
    }
}
```

That's it!

Oh, but I never customized the error message. In the `Constraint` class, give the `$message` property a better default message:

```
src/Validator/TreasuresAllowedOwnerChange.php
// ... lines 1 - 12
class TreasuresAllowedOwnerChange extends Constraint
{
// ... lines 15 - 18
    public string $message = 'One of the treasures illegally changed owners.';
}
```

All right team, moment of truth! Run that test:

```
symfony php bin/phpunit --filter=testTreasuresCannotBeStolen
```

Nailed it! Treasure stealing is officially off the table. Oh, and though I didn't do it, we could also inject the `Security` service to allow admin users to do whatever they want.

Up next: when we create a `DragonTreasure`, we *must* send the `owner` field. Let's finally make that optional. If we don't pass the `owner`, we'll set it to the currently authenticated user. To do that, we need to hook into API platform's "saving" process one more time.

Chapter 34: Auto Setting the "owner"

Every `DragonTreasure` must have an `owner`... and to set that, when you `POST` to create a treasure, we *require* that field. I think we should make that optional. So, in the test, *stop* sending the `owner` field:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 41
    public function testPostToCreateTreasure(): void
    {
// ... lines 44 - 45
        $this->browser()
// ... lines 47 - 51
            ->post('/api/treasures', HttpOptions::json([
// ... lines 53 - 56
                'owner' => '/api/users/'.$user->getId(),
            ]))
// ... lines 59 - 60
        ;
    }
// ... lines 63 - 179
}
```

When this happens, let's automatically set it to the currently-authenticated user.

Make sure the test fails. Copy the method name... and run it:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Nailed it. Got a 422, 201 expected. That 422 is a validation error from the `owner` property: this value should not be null.

Removing the Owner Validation

If we're going to make it optional, we need to remove that `Assert\NotNull`:

```
src/Entity/DragonTreasure.php
// ... lines 1 - 88
class DragonTreasure
{
// ... lines 91 - 136
    #[Assert\NotNull]
// ... line 138
    private ?User $owner = null;
// ... lines 140 - 251
}
```

And now when we try the test:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Well hello there gorgeous 500 error! Probably it's because the null `owner_id` is going kaboom when it hits the database. Yup!

Using the State Processors

So: how can we automatically set this field when it's not sent? In the previous API Platform 2 tutorial, I did this with an entity listener, which is a fine solution. But in API Platform 3, just like when we hashed the user password, there's now a really nice system for this: the state processor system.

As a reminder, our POST and PATCH endpoints for `DragonTreasure` already have a state processor that comes from Doctrine: it's responsible for saving the object to the database. Our goal will feel familiar at this point: to *decorate* that state process so we can run extra code before saving.

Like before, start by running:

```
php bin/console make:state-processor
```

Call it `DragonTreasureSetOwnerProcessor`:

src/State/DragonTreasureSetOwnerProcessor.php

```
// ... lines 1 - 2
namespace App\State;

use ApiPlatform\Metadata\Operation;
use ApiPlatform\State\ProcessorInterface;

class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
    public function process(mixed $data, Operation $operation, array
$uriVariables = [], array $context = []): void
    {
        // Handle the state
    }
}
```

Over in `src/State/`, open that up. Ok, let's decorate! Add the construct method with `private ProcessorInterface $innerProcessor`:

src/State/DragonTreasureSetOwnerProcessor.php

```
// ... lines 1 - 5
use ApiPlatform\State\ProcessorInterface;
// ... lines 7 - 9
class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
    public function __construct(private ProcessorInterface
$innerProcessor)
    {
    }
// ... lines 15 - 19
}
```

Tip

In API Platform 3.2 and higher, you should `return $this->innerProcessor->process()`. This is also a safe thing to do in 3.0 & 3.1.

Then down in `process()`, call that! This method doesn't return anything - it has a `void` return - so we just need `$this->innerProcessor` - don't forget that part like I am - `->process()` passing `$data`, `$operation`, `$uriVariables` and `$context`:

```
src/State/DragonTreasureSetOwnerProcessor.php
// ... lines 1 - 9
class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
// ... lines 12 - 15
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        $this->innerProcessor->process($data, $operation, $uriVariables, $context);
    }
}
```

Now, to make Symfony *use* our state processor instead of the *normal* one from Doctrine, add `#[AsDecorator]`... and the id of the service is `api_platform.doctrine.orm.state.persist_processor`:

```
src/State/DragonTreasureSetOwnerProcessor.php
// ... lines 1 - 6
use Symfony\Component\DependencyInjection\Attribute\AsDecorator;

#[AsDecorator('api_platform.doctrine.orm.state.persist_processor')]
class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
// ... lines 12 - 19
}
```

Cool! Now, everything that uses that service in the system will be passed *our* service instead... and then the original will be passed into us.

Decorating Multiple Times is Ok!

Oh, and there's something cool going on. Look at `UserHashPasswordStateProcessor`. We're decorating the *same* thing there! Yea, we're decorating that service *twice*, which is totally allowed! Internally, this will create a, sort of, *chain* of decorated services.

Ok, let's get to work setting the owner. Autowire our favorite `Security` service so we can figure out who is logged in:

```
src/State/DragonTreasureSetOwnerProcessor.php
// ... lines 1 - 7
use Symfony\Bundle\SecurityBundle\Security;
// ... lines 9 - 11
class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
    public function __construct(private ProcessorInterface
$innerProcessor, private Security $security)
    {
    }
// ... lines 17 - 25
}
```

Then, before we do the saving, if `$data` is an `instanceof DragonTreasure` and `$data->getOwner()` is null *and* `$this->security->getUser()` - making sure the user is logged in - then `$data->setOwner($this->security->getUser())`:

```
src/State/DragonTreasureSetOwnerProcessor.php
// ... lines 1 - 11
class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
// ... lines 14 - 17
    public function process(mixed $data, Operation $operation, array
$uriVariables = [], array $context = []): void
    {
        if ($data instanceof DragonTreasure && $data->getOwner() === null
&& $this->security->getUser()) {
            $data->setOwner($this->security->getUser());
        }

        $this->innerProcessor->process($data, $operation, $uriVariables,
$context);
    }
}
```

That should do it! Run that test:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

Yikes! Allowed memory size exhausted. I smell recursion! Because... I'm calling `process()` on myself: I need `$this->innerProcessor->process()`:

```php
// src/State/DragonTreasureSetOwnerProcessor.php
// ... lines 1 - 11
class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
// ... lines 14 - 17
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
// ... lines 20 - 23
        $this->innerProcessor->process($data, $operation, $uriVariables, $context);
    }
}
```

Now:

```
symfony php bin/phpunit --filter=testPostToCreateTreasure
```

A passing test is *so* much cooler than recursion. And the owner field *is* now optional!

Next: we currently return *all* treasures from our GET collection endpoint, including *unpublished* treasures. Let's fix that by modifying the query behind that endpoint to hide them.

Chapter 35: Query Extension: Auto-Filter a Collection

When we get a collection of treasures, we currently return *every* treasure, even unpublished treasures. Probably some of these are unpublished. We *did* add a filter to control this... but let's be honest, that's not the best solution. Really, we need to *not* return unpublished treasures at all.

Find the API Platform Upgrade Guide... and search for the word "state" to find a section that talks about "providers" and "processors". We talked about state processors earlier, like the `PersistProcessor` on the `Put` and `Post` operations, which is responsible for saving the item to the database.

State Providers

But each operation also has something called a state *provider*. *This* is what's responsible for *loading* the object or collection of objects. For example, when we make a GET request for a single item, the `ItemProvider` is what's responsible for taking the ID and querying the database. There's also a `CollectionProvider` to load a collection of items.

So if we want to automatically hide unpublished treasures, one option would be to *decorate* this `CollectionProvider`, very much like we did with the `PersistProcessor`. Except... that won't *quite* work. Why? The `CollectionProvider` from Doctrine executes the query and returns the results. So all *we* would be able to do is *take* those results... then hide the ones we don't want. That's... not ideal for performance - imagine loading 50 treasures then only showing 10 - and it would confuse pagination. What we *really* want to do is *modify* the query itself: to add a `WHERE isPublished = true`.

Testing for the Behavior

Luckily for us, this `CollectionProvider` "provides" its *own* extension point that lets us do *exactly* that.

Before we dive in, let's update a test to show the behavior we want. Find `testGetCollectionOfTreasures()`. Take control of these 5 treasures and make them all `isPublished => true`:

```
tests/Functional/DragonTreasureResourceTest.php
```

```php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 16
    public function testGetCollectionOfTreasures(): void
    {
        DragonTreasureFactory::createMany(5, [
            'isPublished' => true,
        ]);
// ... lines 22 - 44
    }
// ... lines 46 - 183
}
```

because right now, in `DragonTreasureFactory`, `isPublished` is set to a random value:

```
src/Factory/DragonTreasureFactory.php
```

```php
// ... lines 1 - 29
final class DragonTreasureFactory extends ModelFactory
{
// ... lines 32 - 46
    protected function getDefaults(): array
    {
        return [
// ... lines 50 - 51
            'isPublished' => self::faker()->boolean(),
// ... lines 53 - 56
        ];
    }
// ... lines 59 - 73
}
```

Then add one more with `createOne()` and `isPublished` false:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 16
    public function testGetCollectionOfTreasures(): void
    {
        DragonTreasureFactory::createMany(5, [
            'isPublished' => true,
        ]);
        DragonTreasureFactory::createOne([
            'isPublished' => false,
        ]);
// ... lines 25 - 44
    }
// ... lines 46 - 183
}
```

Awesome! And we *still* want to assert that this returns just 5 items. So... let's make sure it fails:

```
symfony php bin/phpunit --filter=testGetCollectionOfTreasures
```

And... yea! It returns 6 items.

Collection Query Extensions

Ok, to modify the query for a collection endpoint, we're going to create something called a query extension. Anywhere in `src/` - I'll do it in the `ApiPlatform/` directory - create a new class called `DragonTreasureIsPublishedExtension`. Make this implement `QueryCollectionExtensionInterface`, then go to "Code"->"Generate" or `Command`+`N` on a Mac - and generate the one method we need: `applyToCollection()`:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 2
namespace App\ApiPlatform;

use ApiPlatform\Doctrine\Orm\Extension\QueryCollectionExtensionInterface;
use ApiPlatform\Doctrine\Orm\Util\QueryNameGeneratorInterface;
use ApiPlatform\Metadata\Operation;
use Doctrine\ORM\QueryBuilder;

class DragonTreasureIsPublishedExtension implements QueryCollectionExtensionInterface
{
    public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass, Operation $operation = null, array $context = []): void
    {
        // TODO: Implement applyToCollection() method.
    }
}
```

This is pretty cool: it passes us the `$queryBuilder` and a few other pieces of info. Then, we can *modify* that `QueryBuilder`. The best part? The `QueryBuilder` *already* takes into account things like pagination and any filters that have been applied. So those are *not* things we need to worry about.

Also, thanks to Symfony's autoconfiguration system, *just* by creating this class and making it implement this interface, it will *already* be called whenever a collection endpoint is used!

Query Extension Logic

In fact, it will be called for *any* resource. So the first thing we need is `if (DragonTreasure::class !== $resourceClass)` - fortunately it passes us the class name - then return:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 7
use App\Entity\DragonTreasure;
// ... lines 9 - 10
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface
{
    public function applyToCollection(QueryBuilder $queryBuilder,
QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass,
Operation $operation = null, array $context = []): void
    {
        if (DragonTreasure::class !== $resourceClass) {
            return;
        }
// ... lines 18 - 21
    }
}
```

Below, *this* is where we get to work. Now, every `QueryBuilder` object has a *root alias* that refers to the class or table that you're querying. Usually, *we* create the `QueryBuilder`... like from inside a repository we say something like `$this->createQueryBuilder('d')` and `d` becomes that "root alias". Then we use that in other parts of the query.

However, in *this* situation, *we* didn't create the `QueryBuilder`, so *we* never chose that root alias. It was chosen for us. What is it? It's: "banana". Actually, I have no idea what it is! But we can get it with `$queryBuilder->getRootAliases()[0]`:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 10
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface
{
    public function applyToCollection(QueryBuilder $queryBuilder,
QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass,
Operation $operation = null, array $context = []): void
    {
        if (DragonTreasure::class !== $resourceClass) {
            return;
        }

        $rootAlias = $queryBuilder->getRootAliases()[0];
// ... lines 20 - 21
    }
}
```

Now it's just normal query logic: `$queryBuilder->andWhere()` passing `sprintf()`. This looks a little weird: `%s.isPublished = :isPublished`, then pass `$rootAlias` followed by `->setParameter('isPublished', true)`:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
```

```php
// ... lines 1 - 10
class DragonTreasureIsPublishedExtension implements QueryCollectionExtensionInterface
{
    public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass, Operation $operation = null, array $context = []): void
    {
        if (DragonTreasure::class !== $resourceClass) {
            return;
        }

        $rootAlias = $queryBuilder->getRootAliases()[0];
        $queryBuilder->andWhere(sprintf('%s.isPublished = :isPublished', $rootAlias))
            ->setParameter('isPublished', true);
    }
}
```

Cool! Spin over to try this thing!

```
symfony php bin/console phpunit --filter=testGetCollectionOfTreasures
```

Mission accomplished! It's just that easy.

Query Extensions on SubResources?

By the way, will this also work for sub-resources? For example, over in our docs, we can *also* fetch a collection of treasures by going to `/api/users/{user_id}/treasures`. Will this *also* hide the unpublished treasures? The answer is... *yes*! So, it's not something you need to worry about. I won't show it, but this *also* uses the query extension.

Oh, and if you wanted admin users to be able to see unpublished treasures, you could add logic to *only* modify this query if the current user is *not* an admin.

Next up: this query extension fixes the collection endpoint! But... someone could *still* fetch a *single* unpublished treasure directly by its id. Let's fix that!

Chapter 36: 404 On Unpublished Items

We've stopped returning unpublished treasures from the treasure *collection* endpoint, but you *can* still fetch them from the GET one endpoint. That's because these `QueryCollectionExtensionInterface` classes are only called when we are fetching a *collection* of items: not when we're selecting a *single* item.

To prove this, go into our test. Duplicate the collection test, paste, and call it `testGetOneUnpublishedTreasure404s()`. Inside, create just one `DragonTreasure` that's unpublished... and make a `->get()` request to `/api/treasures/`... oh! I need a `$dragonTreasure` variable. That's better. Now add `$dragonTreasure->getId()`.

At the bottom, assert that the status is 404... and we don't need any of these assertions, or this `$json` variable:

```
tests/Functional/DragonTreasureResourceTest.php
```

```php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 46
    public function testGetOneUnpublishedTreasure404s(): void
    {
        $dragonTreasure = DragonTreasureFactory::createOne([
            'isPublished' => false,
        ]);

        $this->browser()
            ->get('/api/treasures/'.$dragonTreasure->getId())
            ->assertStatus(404);
    }
// ... lines 57 - 194
}
```

Very simple! Grab that method name and, you know the drill. Run *just* that test:

```
symfony php bin/phpunit --filter=testGetOneUnpublishedTreasure404s
```

And... yep! It currently returns a 200 status code.

Hello Query Item Extensions

How do we fix this? Well... just like how there's a `QueryCollectionExtensionInterface` for the collection endpoint, there's also a `QueryItemExtensionInterface` that's used whenever API Platform queries for a *single* item.

You can create a totally separate class for this... but you can also combine them. Add a second interface for `QueryItemExtensionInterface`. Then, scroll down and go to "Code"->"Generate" - or `Command`+`N` on a Mac - to add the one method we're missing: `applyToItem()`:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 5
use ApiPlatform\Doctrine\Orm\Extension\QueryItemExtensionInterface;
// ... lines 7 - 11
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
// ... lines 14 - 24
    public function applyToItem(QueryBuilder $queryBuilder,
QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass,
array $identifiers, Operation $operation = null, array $context = []):
void
    {
        // TODO: Implement applyToItem() method.
    }
}
```

Yea, it's almost identical to the collection method.... it works the same way... and we even need the same logic! So, copy the code we need, then go to the Refactor menu and say "Refactor this", which is also `Control`+`T` on a Mac. Select to extract this to a method... and call it `addIsPublishedWhere()`:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 11
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
// ... lines 14 - 23
    /**
     * @param string $resourceClass
     * @param QueryBuilder $queryBuilder
     * @return void
     */
    private function addIsPublishedWhere(string $resourceClass,
QueryBuilder $queryBuilder): void
    {
// ... lines 31 - 34
        $rootAlias = $queryBuilder->getRootAliases()[0];
        $queryBuilder->andWhere(sprintf('%s.isPublished = :isPublished',
$rootAlias))
            ->setParameter('isPublished', true);
    }
}
```

Awesome! I'll clean things up... and, you know what? I should have added this `if` statement inside there too. So let's move that:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 11
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
// ... lines 14 - 28
    private function addIsPublishedWhere(string $resourceClass,
QueryBuilder $queryBuilder): void
    {
        if (DragonTreasure::class !== $resourceClass) {
            return;
        }

        $rootAlias = $queryBuilder->getRootAliases()[0];
        $queryBuilder->andWhere(sprintf('%s.isPublished = :isPublished',
$rootAlias))
            ->setParameter('isPublished', true);
    }
}
```

Which means we need a `string $resourceClass` argument. Above, pass `$resourceClass` to the method:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 11
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
    public function applyToCollection(QueryBuilder $queryBuilder,
QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass,
Operation $operation = null, array $context = []): void
    {
        $this->addIsPublishedWhere($resourceClass, $queryBuilder);
    }
// ... lines 18 - 38
}
```

Perfect! Now, in `applyToItem()`, call that same method:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 11
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
// ... lines 14 - 18
    public function applyToItem(QueryBuilder $queryBuilder,
QueryNameGeneratorInterface $queryNameGenerator, string $resourceClass,
array $identifiers, Operation $operation = null, array $context = []):
void
    {
        $this->addIsPublishedWhere($resourceClass, $queryBuilder);
    }
// ... lines 23 - 38
}
```

Ok, we're ready! Try the test now:

```
symfony php bin/phpunit --filter=testGetOneUnpublishedTreasure404s
```

And... it passes!

Fixing our Test Suite

We've been tinkering with our code quite a bit, so it's time for a test-a-palooza! Run all the tests:

```
symfony php bin/phpunit
```

And... whoops! 3 failures - all coming from `DragonTreasureResourceTest`. The problem is that, when we created treasures in our tests, we weren't explicit about whether we wanted a published or unpublished treasure... and that value is set randomly in our factory.

To fix this, we could be explicit by controlling the `isPublished` field whenever we create a treasure. Or... we can be lazier and, in `DragonTreasureFactory`, set `isPublished` to true by default:

src/Factory/DragonTreasureFactory.php

```php
// ... lines 1 - 29
final class DragonTreasureFactory extends ModelFactory
{
// ... lines 32 - 46
    protected function getDefaults(): array
    {
        return [
// ... lines 50 - 51
            'isPublished' => true,
// ... lines 53 - 56
        ];
    }
// ... lines 59 - 73
}
```

Now, to keep our fixture data interesting, when we create the 40 dragon treasures, let's override `isPublished` and manually add some randomness: if a random number from 0 to 10 is greater than 3, then make it published:

```
src/DataFixtures/AppFixtures.php
// ... lines 1 - 10
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
// ... lines 15 - 20
        DragonTreasureFactory::createMany(40, function () {
            return [
// ... line 23
                'isPublished' => rand(0, 10) > 3,
            ];
        });
// ... lines 27 - 32
    }
}
```

That *should* fix most of our tests. Though search for `isPublished`. Ah yea, we're testing that an admin can `PATCH` to edit a treasure. We created an *unpublished* `DragonTreasure`... just so we could assert that this was in the response. Let's change this to `true` in both places:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 153
    public function testAdminCanPatchToEditTreasure(): void
    {
// ... line 156
        $treasure = DragonTreasureFactory::createOne([
            'isPublished' => true,
        ]);

        $this->browser()
// ... lines 162 - 169
            ->assertJsonMatches('isPublished', true)
        ;
    }
// ... lines 173 - 194
}
```

There's one other similar test: change `isPublished` to `true` here as well:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 173
    public function testOwnerCanSeeIsPublishedAndIsMineFields(): void
    {
// ... line 176
        $treasure = DragonTreasureFactory::createOne([
            'isPublished' => true,
// ... line 179
        ]);

        $this->browser()
// ... lines 183 - 190
            ->assertJsonMatches('isPublished', true)
// ... line 192
        ;
    }
}
```

Now try the tests:

```
symfony php bin/phpunit
```

Allowing Updates to an Unpublished Item

They're happy! I'm happy! Well, *mostly*. We still have one teensie problem. Find the first `PATCH` test. We're creating a *published* `DragonTreasure`, updating it... and it works just fine. Copy this entire test... paste it.. but delete the bottom part: we only need the top. Call this method `testPatchUnpublishedWorks()`... then make sure the `DragonTreasure` is *unpublished*:

```
tests/Functional/DragonTreasureResourceTest.php
// ... lines 1 - 12
class DragonTreasureResourceTest extends ApiTestCase
{
// ... lines 15 - 153
    public function testPatchUnpublishedWorks()
    {
// ... line 156
        $treasure = DragonTreasureFactory::createOne([
// ... line 158
            'isPublished' => false,
        ]);
// ... lines 161 - 171
    }
// ... lines 173 - 215
}
```

Think about it: if I have a `DragonTreasure` with `isPublished` `false`, I *should* be able to update it, right? This is *my* treasure... I created it and I'm still working on it. We want this to be allowed.

Will it? You can probably guess:

```
symfony php bin/phpunit --filter=testPatchUnpublishedWorks
```

Nope! We get a 404! This is both a feature... and a "gotcha"! When we create a `QueryCollectionExtensionInterface`, that's only used for this *one* collection endpoint. But when we create an `ItemExtensionInterface`, that's used *whenever* we fetch a single treasure: *including* for the `Delete`, `Patch` and `Put` operations. So, when an owner tries to `Patch` their own `DragonTreasure`, thanks to our query extension, it can't be found.

There are two solutions for this. First, in `applyToItem()`, API Platform passes us the `$operation`. So we could use this to determine if this a `Get`, `Patch` or `Delete` operation and only apply the logic for *some* of those.

And... this might make sense. After all, if you're *allowed* to edit or delete a treasure... that means you've already passed a security check... so we don't necessarily need to lock things down via this query extension.

The other solution is to change the query to allow owners to see their *own* treasures. One cool thing about this solution is that it will also allow unpublished treasures to be returned from the collection endpoint if the current user is the owner of that treasure.

Let's give this a shot. Add the `public function __construct()`... and autowire the amazing `Security` service:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 10
use Symfony\Bundle\SecurityBundle\Security;

class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
    public function __construct(private Security $security)
    {
    }
// ... lines 18 - 50
}
```

Below... life gets a bit trickier. Start with `$user = $this->security->getUser()`. *If* we have a user, we're going to modify the `QueryBuilder` in a similar... but slightly different way. Oh, actually, let me bring the `$rootAlias` up above my if statement. Now, if the user is logged in, add `OR %s.owner = :owner`... then pass in one more `rootAlias`... followed by `->setParameter('owner', $user)`.

Else, if there is no user, use the original query. And we need the `isPublished` parameter in both cases... so keep that at the bottom:

```
src/ApiPlatform/DragonTreasureIsPublishedExtension.php
// ... lines 1 - 12
class DragonTreasureIsPublishedExtension implements
QueryCollectionExtensionInterface, QueryItemExtensionInterface
{
// ... lines 15 - 33
    private function addIsPublishedWhere(string $resourceClass,
QueryBuilder $queryBuilder): void
    {
        if (DragonTreasure::class !== $resourceClass) {
            return;
        }

        $rootAlias = $queryBuilder->getRootAliases()[0];
        $user = $this->security->getUser();
        if ($user) {
            $queryBuilder->andWhere(sprintf('%s.isPublished = :isPublished
OR %s.owner = :owner', $rootAlias, $rootAlias))
                ->setParameter('owner', $user);
        } else {
            $queryBuilder->andWhere(sprintf('%s.isPublished =
:isPublished', $rootAlias));
        }

        $queryBuilder->setParameter('isPublished', true);
    }
}
```

I think I like that! Let's see what the test thinks:

```
symfony php bin/phpunit --filter=testPatchUnpublishedWorks
```

It likes it too! In fact, *all* of our tests seem happy.

Ok team: final topic. When we fetch a `User` resource, we return its dragon treasures. Does that collection *also* include *unpublished* treasures? Ah... yep it does! Let's talk about why and how to fix it next.

Chapter 37: Filtering Relation Collection

Hey, we've made a pretty fancy API! We've got a few sub-resources and embedded relation data, which is readable and writable. This is all super awesome... but it sure does crank up the complexity of our API, especially when it comes to security.

For example, we can no longer see unpublished treasures from the GET collection or GET single endpoints. But we *can* still see unpublished treasures if you fetch a user and read its `dragonTreasures` field.

Writing the Test

Let's whip up a test real quick to expose this problem. Open our `UserResourceTest`. At the bottom, add a public function `testUnpublishedTreasuresNotReturned()`. Inside that, create a user with `UserFactory::createOne()`. Then use `DragonTreasureFactory` to create a treasure that's `isPublished` false and has its `owner` set to the `$user`... just so we know *who* the owner is.

For the action, say `$this->browser()`... and we *do* need to log in to use the endpoint... but we don't care *who* we're logged in as... so say `actingAs()` `UserFactory::createOne()` to log in as someone else.

Then `->get()` `/api/users/` `$user->getId()`. Finish with `assertJsonMatches()` that the `length()` of `dragonTreasures` is zero - using a cool `length()` function from that JMESPath syntax:

```
tests/Functional/UserResourceTest.php
// ... lines 1 - 8
class UserResourceTest extends ApiTestCase
{
// ... lines 11 - 68
    public function testUnpublishedTreasuresNotReturned(): void
    {
        $user = UserFactory::createOne();
        DragonTreasureFactory::createOne([
            'isPublished' => false,
            'owner' => $user,
        ]);

        $this->browser()
            ->actingAs(UserFactory::createOne())
            ->get('/api/users/' . $user->getId())
            ->assertJsonMatches('length("dragonTreasures")', 0);
    }
}
```

Let's try it! Copy the method... and run it with `--filter=` that name:

```
symfony php bin/phpunit --filter=testUnpublishedTreasuresNotReturned
```

Ok! It expected 1 to be the same as 0 because we *are* returning the unpublished treasure... but we don't want to!

How Relations are Loaded

First... why *is* this unpublished `DragonTreasure` being returned? Didn't we build query extension classes to prevent *exactly* this?

Well.... an important thing to understand is that these query extension classes are used for the *main* query on an endpoint only. For example, if we use the GET collection endpoint for treasures, the "main" query is for those treasures and the query collection extension *is* called.

But when we make a call to a *user* endpoint - like to GET a single `User` - API Platform is *not* making a query for any treasures: it's making a query for that *one* `User`. Once it has that `User`, to get this `dragonTreasures` field, it does *not* make another query for those, at least not directly. Instead, if you open the `User` entity, API Platform - via the serializer - simply calls `getDragonTreasures()`.

So it queries for the `User`, calls `->getDragonTreasures()`... and whatever *that* returns is set onto the `dragonTreasures` field. And since this returns *all* related treasures, that's what we get: including the unpublished ones.

Adding a Filtered Getter Method

How can we fix this? By adding a *new* method that only returns the *published* treasures. Say `public function getPublishedDragonTreasures()`, which returns a `Collection`. Inside, we can get fancy: return `$this->dragonTreasures->filter()` passing that a callback with a `DragonTreasure $treasure` argument. *Then*, return `$treasure->getIsPublished()`:

```
src/Entity/User.php
// ... lines 1 - 69
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 72 - 216
    public function getPublishedDragonTreasures(): Collection
    {
        return $this->dragonTreasures->filter(static function
(DragonTreasure $treasure) {
            return $treasure->getIsPublished();
        });
    }
// ... lines 223 - 303
}
```

That's a nifty trick for looping through all the treasures and getting a shiny *new* collection with just the *published* ones.

Side note: one downside to this approach is that if a user has 100 treasures... but only 10 of them are published, internally, Doctrine will first query for all 100... even though we'll only return 10. If you have *large* collections, this can be a performance problem. In our Doctrine tutorial, we talk about fixing this with something called the Criteria system. But with both approaches, the result is the same: a method that returns a subset of the collection.

Swapping the Getter into our API

At this point, the new method will work, but it's not *yet* part of our API. Scroll up to the `dragonTreasures` property. It's currently readable and writable in our API. Make the property only writable:

```
src/Entity/User.php
// ... lines 1 - 69
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 72 - 105
    #[Groups(['user:write'])]
// ... lines 107 - 108
    private Collection $dragonTreasures;
// ... lines 110 - 305
}
```

Then, down on the new method, add `#[Groups('user:read')]` to make this part of our API and `#[SerializedName('dragonTreasures')]` to give it the original name:

```
src/Entity/User.php
// ... lines 1 - 69
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 72 - 216
    #[Groups(['user:read'])]
    #[SerializedName('dragonTreasures')]
    public function getPublishedDragonTreasures(): Collection
    {
// ... lines 221 - 223
    }
// ... lines 225 - 305
}
```

Drumroll! Try the test:

```
symfony php bin/phpunit --filter=testUnpublishedTreasuresNotReturned
```

It explodes! Because... I have a syntax error. Try it again. All green!

And... we're done! You did it! Thank you *so* much for joining me on this gigantic, cool, challenging journey into API Platform and security. Parts of this tutorial were pretty complex... because I want you to be able to solve *real*, tough security problems.

In the next tutorial, we're going to look at even *more* custom and powerful things that you can do with API Platform, including how to use classes for API resources that are *not* entities.

In the meantime, let us know what you're building and, as always, we're here for you in the comments section. Alright friends, see ya next time!

With <3 from SymfonyCasts