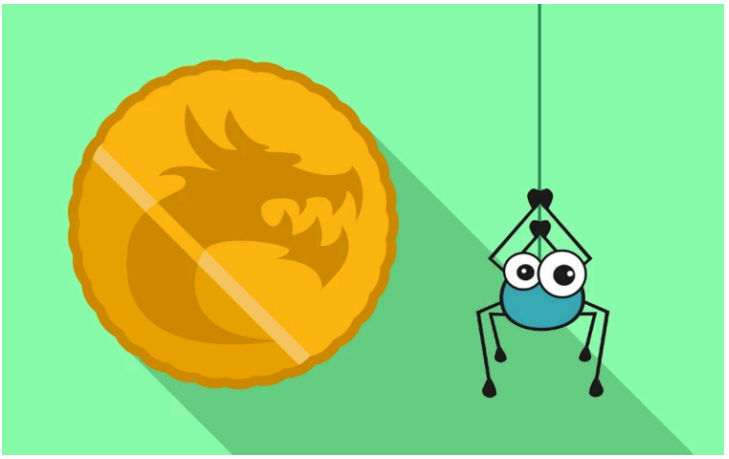
API Platform 3 Part 1: Mythically Good RESTful APIs



Chapter 1: Installing API Platform

Hello and *welcome* you beautiful people, to a tutorial that's near and dear to my heart: how to build magnificent castles with *Legos*. Oh, that would be awesome, wouldn't it? But really, we're here to talk about API Platform Version 3, which I promise is *as* fun as playing with Legos. Just don't tell my son I said that.

API Platform is, very simply, a tool on top of Symfony that allows us to build powerful APIs *and* love the process! It's been around for years and, honestly, it's *crushing it*. They have their own dedicated conference and, they've really outdone themselves with the latest version 3.

If you're new to API Platform, I wouldn't blame you if you said:

> *"Come on Ryan... creating an API isn't that hard. It's just returning JSON: a bunch of squigglies and brackets!"*

Ok, that *is* true (at least for the first few endpoints). But wow are there a lot of little details to keep track of. For example, if you have an API that returns product data, you'll want that product JSON to be returned in the *same* way with the *same* fields, across all endpoints. That process is called *serialization*. On top of that, a lot of APIs now return *extra* fields that describes what the data means. We're going to see and talk about something called "JSON-LD", which does *exactly* that.

What else? How about documentation? Ideally interactive documentation that's generated automatically... because we do *not* want to build and maintain that by hand. Even if you're building an API just for yourself, having documentation is *awesome*. Paginating collections is also super important, filtering and searching collections, validation and content-type negotiation, which is where that same product could be returned as JSON, CSV, or another format. So *yes*, creating an API endpoint *is* easy. But creating a rich API is another thing *entirely*. And *that's* the point of API Platform. Oh, and if you're familiar with API Platform Version 2, version 3 will feel *very* familiar. It's just cleaner, more modern, and more powerful. So get out your Legos, and let's do this!

The API Platform Distribution

There are *two* ways to install API Platform. If you find their site and click into the documentation, you'll see them talk about the API Platform "Distribution". This is pretty cool! It's a completely pre-made project with Docker that gives you a place to build your API with Symfony, a React admin area, scaffolding to create a Next.js frontend and more. Heck, it even gives you a production-ready web server with extra tools like Mercure for real-time updates. It's *the* most powerful way to use API Platform.

But... in this tutorial, we're *not* going to use it. I hate nice things! No, we'll start our Lego project from scratch: with a perfectly normal and boring Symfony app. Why? Because I want you to see *exactly* how everything works under the hood. Then, if you want to use this Distribution later on, you *totally* can.

Project Setup & Our Project

Okay, to be a *true* "API Platform JSON Returning Champion", you should code along with me! Download the source code from this page. And after unzipping it, you'll find a `start/` directory with the same code that you see here. This is a brand new Symfony 6.2 project with... absolutely *nothing* in it. Open up this `README.md` file for all the setup instructions. The last step will be to open the project in a terminal and use the Symfony binary to run:

```
symfony serve -d
```

This starts a local web server at `127.0.0.1:8000`. I'll cheat and click that link to open up... a *completely empty* Symfony 6.2 project. There's literally nothing here except for this demo homepage.

What *are* we going to build? As we all know, the internet is missing something terribly important: an application for dragons to boast about their stolen treasures! Because if there's one thing a dragon likes more than treasure, it's *bragging* about it. Yup, we'll create a rich API that lets tech savvy dragons post *new* treasures, *fetch* treasures, *search* treasures from other dragons, etc. And yes, I *did* just finish reading the Hobbit.

Installing API Platform

So, let's get API Platform installed! Spin back over to your terminal and run:

```
composer require api
```

This is a Symfony Flex alias. Up here, you can see it's actually installing something called `api-platform/api-pack`. If you're not familiar, a "pack" in Symfony is, kind of a *fake* package, that allows you to easily install a *set* of packages. If you scroll down, it installed `api-platform` *itself*, Doctrine, since I didn't already have that, and some other packages. At the bottom... let's see... the `doctrine-bundle` recipe is asking us if we want to include a `docker-compose.yml` file to help add a database to our project. How nice of it! This is *optional*, but I'm going to say "p" for "Yes permanently". And... done!

The first thing to see is in the `composer.json` file:

composer.json

```
{
// ... lines 2 - 5
    "require": {
        "php": ">=8.1",
        "ext-ctype": "*",
        "ext-iconv": "*",
        "api-platform/core": "^3.0",
        "doctrine/annotations": "^1.0",
        "doctrine/doctrine-bundle": "^2.8",
        "doctrine/doctrine-migrations-bundle": "^3.2",
        "doctrine/orm": "^2.14",
        "nelmio/cors-bundle": "^2.2",
        "phpdocumentor/reflection-docblock": "^5.3",
        "phpstan/phpdoc-parser": "^1.15",
        "symfony/asset": "6.2.*",
        "symfony/console": "6.2.*",
        "symfony/dotenv": "6.2.*",
        "symfony/expression-language": "6.2.*",
        "symfony/flex": "^2",
        "symfony/framework-bundle": "6.2.*",
        "symfony/property-access": "6.2.*",
        "symfony/property-info": "6.2.*",
        "symfony/runtime": "6.2.*",
        "symfony/security-bundle": "6.2.*",
        "symfony/serializer": "6.2.*",
        "symfony/twig-bundle": "6.2.*",
        "symfony/validator": "6.2.*",
        "symfony/yaml": "6.2.*"
    },
// ... lines 33 - 83
}
```

As promised, that API Platform pack added a bunch of packages into our project. Technically, these aren't *all* required, but this is going to give us a really rich experience building our API. And if you run

```
git status
```

... yep! It updated the usual files... and also added a bunch of config files for the new packages. It *looks* like there's a lot... but looks can be deceiving. All of these directories are empty... and

the config files are small and simple. We also have some `docker-compose` files that we'll use in a minute to spin up a database.

So... now that API Platform is installed... did that *give* us anything yet? It did! And it's cool! Go back to the browser and head to `/api`. Whoa! We have an API documentation page! It's *empty* because we don't, ya know, actually *have* an API just yet, but this is going to come to life very soon.

Next: Let's create our first Doctrine entity and "expose" that as an API Resource. It's time for some magic.

Chapter 2: Creating your First ApiResource

We're about to build an API for the *very* important job of allowing dragons to show off their treasure. Right now, our project doesn't have a *single* database entity... but we're going to need one to store all that treasure.

Generating our First Entity

Find your terminal and first run

```
composer require maker --dev
```

to install Maker Bundle. *Then* run:

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

Perfect! Let's call our entity `DragonTreasure`. Then it asks us a question that you maybe haven't seen before - `Mark this class as an API platform resource`? It asks because API Platform is installed. Say `no` because we're going to do this step *manually* in a moment.

Okay, let's start adding properties. Start with `name` as a string, with a Length of the default 255, and make it *not* nullable. Then, add `description` with a `text` type, and make *that* not nullable. We also need a `value`, like... how much the treasure is *worth*. That will be an `integer` not nullable. And we simply *must* have a `coolFactor`: dragons need to specify just *how* awesome this treasure is. That'll be a number from 1 to 10, so make it an `integer` and *not* nullable. Then, `createdAt` `datetime_immutable` that's not nullable... and *Finally*, add an `isPublished` property, which will be a `boolean` type, also not nullable. Hit " enter" to finish.

Phew! There's nothing very special so far. This created two classes: `DragonTreasureRepository` (which we're not going to worry about), and the `DragonTreasure` entity itself with `$id`, `$name`, `$description`, `$value`, etc

src/Entity/DragonTreasure.php

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

use App\Repository\DragonTreasureRepository;
use Doctrine\DBAL\Types\Types;
use Doctrine\ORM\Mapping as ORM;

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

    #[ORM\Column(length: 255)]
    private ?string $name = null;

    #[ORM\Column(type: Types::TEXT)]
    private ?string $description = null;

    #[ORM\Column]
    private ?int $value = null;

    #[ORM\Column]
    private ?int $coolFactor = null;

    #[ORM\Column]
    private ?\DateTimeImmutable $plunderedAt = null;

    #[ORM\Column]
    private ?bool $isPublished = null;

// ... lines 35 - 110
}
```

along with the getter and setter methods. Beautifully boring. There *is* one little bug in this version of MakerBundle, though. It generated an `isIsPublished()` method. Let's change that to `getIsPublished()`.

src/Entity/DragonTreasure.php

```
// ... lines 1 - 9
class DragonTreasure
{
// ... lines 12 - 99
    public function getIsPublished(): ?bool
    {
        return $this->isPublished;
    }

    public function setIsPublished(bool $isPublished): self
    {
// ... lines 107 - 109
    }
}
```

Setting up the Database

All right, so we have our entity. Now we need a migration for its table... but that might be a bit difficult since we don't have our database set up yet! I'm going to use Docker for this. The DoctrineBundle recipe gave us a nice `docker-compose.yml` file that boots up Postgres, so... let's use that! Spin over to your terminal and run:

> **Tip**
>
> In modern versions of Docker, run `docker compose up -d` instead of `docker-compose`.

```
docker-compose up -d
```

If you don't want to use Docker, feel free to start your own database engine and then, in `.env` or `.env.local`, configure DATABASE_URL. Because I'm using Docker as well as the `symfony` binary, I don't need to configure anything. The Symfony web server will automatically see the Docker database and set the `DATABASE_URL` environment variable *for* me.

Okay, to make the migration, run:

```
symfony console make:migration
```

This `symfony console` is just like `./bin/console` except it injects the `DATABASE_URL` environment variable so that the command can talk to the Docker database. Perfect! Spin over and check out the new migration file... just to make sure it doesn't contain any weird surprises.

migrations/Version20230104160057.php

```
// ... lines 1 - 4
namespace DoctrineMigrations;

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20230104160057 extends AbstractMigration
{
// ... lines 15 - 19
    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE SEQUENCE dragon_treasure_id_seq INCREMENT BY 1 MINVALUE 1 START 1');
        $this->addSql('CREATE TABLE dragon_treasure (id INT NOT NULL, name VARCHAR(255) NOT NULL, description TEXT NOT NULL, value INT NOT NULL, cool_factor INT NOT NULL, plundered_at TIMESTAMP(0) WITHOUT TIME ZONE NOT NULL, is_published BOOLEAN NOT NULL, PRIMARY KEY(id))');
        $this->addSql('COMMENT ON COLUMN dragon_treasure.plundered_at IS \'(DC2Type:datetime_immutable)\'');
    }

    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 dragon_treasure_id_seq CASCADE');
        $this->addSql('DROP TABLE dragon_treasure');
    }
}
```

Looks good! So spin back over and run this with:

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

Done!

Exposing our First API Resource

We now have an entity and a database table. But if you go and refresh the documentation... there's still nothing there. What we need to do is tell API Platform to expose our `DragonTreasure` entity as an API resource. To do this, go above the class and add a new attribute called `ApiResource`. Hit "tab" to add that `use` statement.

src/Entity/DragonTreasure.php

```
// ... lines 1 - 4
use ApiPlatform\Metadata\ApiResource;
// ... lines 6 - 9
#[ORM\Entity(repositoryClass: DragonTreasureRepository::class)]
#[ApiResource]
class DragonTreasure
{
// ... lines 14 - 112
}
```

Done! As soon as we do that... and refresh... whoa! The documentation is alive! It now shows that we have *six* different endpoints: One to retrieve *all* of the `DragonTreasure` resources, one to retrieve an *individual* `DragonTreasure`, one to *create* a `DragonTreasure`, two that edit a `DragonTreasure` plus one to delete it. And this is more than just documentation. These endpoints *work*.

Go over and click "Try it Out", then "Execute". It doesn't actually *return* anything because our database is empty, but it *does* gives us a 200 status code with some empty JSON. We'll talk about all of the other fancy keys in the response shortly.

Oh, but I do want to mention one thing. As we just saw, the easiest way to create a set of API endpoints is by adding this `ApiResource` attribute above your entity class. But you can actually add this attribute above *any* class: *not* just entities. That's something we're going to talk about in a future tutorial: it can be a nice way to separate what your API looks like from what

your entity looks like, especially in bigger APIs. But again, that's for *later*. Right now, using `ApiResource` on top of our entity is going to work great.

Let's discover this cool, interactive documentation a bit more. Where did this come from? How does our app magically have a bunch of new routes? And do dragons *really* love tacos? Let's find out next!

Chapter 3: Swagger UI: Interactive Docs

The amazing interactive documentation that we've stumbled across is *not* something from API platform! Nope, it's actually an open-source API documentation library called Swagger UI. And the *really* cool thing about Swagger UI is that, if someone create a file that *describes* any API, then that API can get all of this for free! I love free stuff! *We* get Swagger UI because API platform *provides* that description file out of the box. But more on that in a minute.

Playing with our New API

Let's play around with this. Use the POST endpoint to create a new `DragonTreasure`. We've recently plundered some "Gold coins"... which we got from "Scrooge McDuck". He's mad. For our purposes, none of the other fields really matter. Down here, hit " Execute" and... boom! When you scroll down, you can see that this made a POST request to `/api/dragon_treasures` and sent all of that data as JSON! Then, our API returned a "201" status code. A 201 status means that the request was successful and a resource was *created*. Then it returned this JSON, which includes an `id` of `1`. So, as I said, this isn't *just* documentation: we really *do* have a working API! There are a few extra fields here too: `@context`, `@id`, and `@type` We'll talk about those soon.

Now that we have a `DragonTreasure` to work with, open up this "GET" endpoint, click "Try it Out", then "Execute". Oh, I love it. Swagger just made a `GET` request to `/api/dragon_treasures` - this `?page=1` is optional. Our API returned information inside something called `hydra:member`, which isn't particularly important yet. What matters is that our API *did* return a list of all of the `DragonTreasures` we currently have, which is just this one.

So in *just* a few minutes of work, we have a fully featured API for our Doctrine entity. That is *cool*.

Content Negotiation

Copy the URL to the API endpoint, open a new tab, and paste that in. Whoa! This... returned HTML? But a second ago, Swagger said that it made a `GET` request to that URL... and it returned *JSON*. What's going on?

One feature of API Platform is called "Content Negotiation". It means that our API can return the same resource - like `DragonTreasure` - in *multiple* formats, like JSON, or HTML... or even things like CSV. Oh, an ASCII format would be *awesome*. Anyways, we *tell* API Platform which format we want by passing an `Accept` header in the request. When we use the interactive docs, it passes this `Accept` header *for* us set to `application/ld+json`. We'll talk about the `ld+json` part soon... but, thanks to this, our API returns JSON!

And even though we don't *see* it here, when you go to a page in your browser, your browser automatically sends an `Accept` header that says we want `text/html`. So, this is API Platform showing us the "HTML representation" of our dragon treasures..., which is just the documentation. Watch: when I open the endpoint this URL is for, it automatically executed it.

The point is: if we want to see the JSON representation of our dragon treasures, we need to pass this `Accept` header... which is super easy, for example, if you're writing JavaScript.

But passing a custom `Accept` header isn't so easy in a browser... and it *would* be nice to be able to see the JSON version of this. Fortunately, API Platform gives us a way to cheat. Remove the `?page=1` to simplify things. Then, at the end of any endpoint, you can add `.` followed by the *extension* of the format you want: like `.jsonld`.

Now we see the `DragonTreasure` resource in that format. API Platform also supports normal JSON out of the box, so we can see the same thing, but in pure, standard JSON.

Where do the new Routes Come From?

The fact that *all* of this works means that... we apparently have a new route for `/api` as well as a bunch of other new routes for each operation - like `GET /api/dragon_treasures`. But... where did these come from? How are they being dynamically added to our app?

To answer that, spin over to your terminal and run:

```
./bin/console debug:router
```

I'll make this a bit smaller so we can see everything. Yup! Each endpoint is represented by a normal, traditional route. *How* are these being added? When we installed API Platform, its recipe added a `config/routes/api_platform.yaml` file.

config/routes/api_platform.yaml

```yaml
api_platform:
    resource: .
    type: api_platform
    prefix: /api
```

This is actually a route *import*. It looks a little weird, but it activates API Platform when the routing system is loading. API Platform then finds all of the API resources in our app and generates a route for every endpoint.

The point is that all *we* need to focus on is creating these beautiful PHP classes and decorating them with `ApiResource`. API Platform takes care of all the heavy lifting of hooking up those endpoints. Of course, we'll need to tweak the configuration and talk about more advanced things, but hey! That's the point of this tutorial. And we're *already* off to an epic start.

Next: I want to talk about the secret behind how this Swagger UI documentation is generated. It's called *OpenAPI*.

Chapter 4: The Powerful OpenAPI Spec

Earlier, I said that these interactive docs come from an open source library called Swagger UI. And as long as you have some config that *describes* your API, like what endpoints it has and what fields are used on each endpoint, then you can generate these rich Swagger docs *automatically*.

Head to https://petstore3.swagger.io. This is really cool: it's is a demo project where Swagger UI is being used on a demo API. And, it has a link to the API configuration file that's powering this!

Hello OpenAPI!

Let's... see what that looks like! Woh! Yea, this JSON file fully describes the API, from basic information about the API itself, all the way down to the different URLs, like updating an existing pet, adding a new pet to the store, the responses... *everything*. *If* you have one of these files, then you can get Swagger *instantly*.

The format of this file is called *OpenAPI*, which is just a standard for *how* APIs should be described.

> **Tip**
>
> In newer projects, access the JSON docs at `/api/docs.jsonopenapi`.

Back over in our docs, we *must* have that same type of config file, right? We do! Head to `/api/docs.json` to see *our* version. Yup! It looks *very* similar. It has paths, it describes the different operations... everything. The *best* part is that API Platform reads our code and generates this giant file *for* us. Then, *because* we have this giant file, we get Swagger UI.

In fact, if you click on "View Page Source", you can see that this page works by embedding the actual JSON document right into the HTML. Then, there's some Swagger JavaScript that reads that and boots things up.

OpenAPI & Free Tools

This idea of having an OpenAPI specification that describes your API is powerful... because there are an increasing number of tools that can *use* it. For example, go back to the API Platform documentation and click on "Schema Generator". This is pretty wild: you can use a service called "Stoplight" to *design* your API. That will give you an OpenAPI specification document... and then you can use the Schema Generator to generate your PHP classes *from* that. We're not going to use that, but it's a cool idea.

There's also an admin generator built in React - we'll play with this later - and even ways to help generate JavaScript that talks to your API. For example, you can generate a Next.js frontend by having it read from your OpenAPI spec.

The *point* is, Swagger UI is awesome. But even *more* awesome is the OpenAPI spec document behind this... which can be used for other stuff.

Models / Schema in OpenAPI

In addition to the endpoints in Swagger, it also has something called "Schemas". These are your models... and there are two - one for JSON-LD and a normal one. We're going to talk about JSON-LD in a minute, but these are basically the same.

If you open one up, wow, this is smart. It knows that our `id` is an integer, `name` is a string, `coolFactor` is an integer, and `isPublished` is a boolean. All of this info is, once again, coming from this spec document. If we search for `isPublished` in here... yep! *There's* the model describing `isPublished` as `type` `boolean`. The best part is that API Platform is generating this by... just looking at our code!

For example, it sees that `coolFactor` has an integer type:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 class DragonTreasure
13 {
↕ // ... lines 14 - 28
29     private ?int $coolFactor = null;
↕ // ... lines 30 - 112
113 }
```

so it *advertises* it as an integer in OpenAPI. But it gets even *better*. Check out the `id`. It's set as `readOnly`. How does it know that? Well, `id` is a *private* property and there's *no* `setId()` method:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 class DragonTreasure
13 {
↕ // ... lines 14 - 16
17     private ?int $id = null;
↕ // ... lines 18 - 36
37     public function getId(): ?int
38     {
39         return $this->id;
40     }
↕ // ... lines 41 - 112
113 }
```

And so, it correctly inferred that `id` must be `readOnly`.

We can also help API Platform. Find the `$value` property... there it is... and add a little documentation above this so people know that `This is the estimated value of this treasure, in gold coins.`

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 class DragonTreasure
13 {
↕ // ... lines 14 - 24
25     /**
26      * The estimated value of this treasure, in gold coins.
27      */
28     #[ORM\Column]
29     private ?int $value = null;
↕ // ... lines 30 - 115
116 }
```

Head over, refresh... and check out the model down here. For `value`... it shows up! The point is: if you do a good job writing your PHP code and documenting it, you're going to get rich API documentation thanks to OpenAPI, with zero extra work.

Next: Let's talk about these weird `@` fields, like `@id`, `@type`, and `@context`. These come from something called JSON-LD: a powerful addition to JSON that API Platform leverages.

Chapter 5: JSON-LD: Giving Meaning to your Data

I've just used the `GET` collection endpoint to fetch *all* of my resources... which shows that we have a treasure with `id=1`. I'll close up this operation... and use this other `GET` endpoint. Click "Try it out", put "1" in for the ID, and click "Execute".

What does our Data Mean?

Beautiful! But... I have some questions. Specifically: what is the *significance* of these fields? What do `name` or `description` or `value` actually *mean*? Is the description plain text? HTML? Is `name` a short name for the item or a *proper* name? Is this value in dollars? Euros? French fries? What the heck is `coolFactor`? And why am I asking *you* all of these unfair questions?

If you're a human (you are... right?), then you can probably figure out a lot of the "meaning" of these fields on your own. But *machines* - okay, maybe *minus* futuristic AIs - well, they *can't* figure this out. They don't know what these keys *mean*. So... how *can* we give context and meaning to our data?

RDF: Resource Description Framework

First, there's this thing called "RDF" or "Resource Description Framework", which is a set of rules about how we describe the meaning of data so that computers can understand. It's... *boring* and abstract, but basically a guide on how you can define that one piece of data has a certain *type*, or one resource is a subclass of some *other* type. In HTML, you can add attributes to your elements to add this RDF metadata. You could say that this `<div>` describes a "person", and that this person's name and telephone are these other pieces of data. This makes the random HTML in your site *understandable* by machines. It's even better if two different sites use the exact same definition of "person", which is why the types are URLs... and sites try to reuse existing types rather than invent new ones.

Hello JSON-LD

Why are we talking about this? Because JSON-LD attempts to do the same thing for our API. Our API endpoints are returning JSON. But the `content-type` header in the response says that this is `application/ld+json`.

When you see `application/ld+json`, it means that the data *is* JSON... but with extra fields that have special meaning according to a giant JSON-LD spec document. So, quite literally, JSON-LD is JSON... with extra goodies.

The @id Field

For example, every resource, like `DragonTreasure`, has three `@` fields. The most important is probably `@id`. This is the unique identifier to the resource. It's basically the same as `id`, but it's even *better* because it's a URL. So instead of just saying `"id": 1`, you have `@id` `/api/dragon_treasures/1`. That means that, first, the string will be unique across all of our API resource classes and second, this URL is handy! You can pop this into your browser, and, if you have the `accept` header or add `.jsonld` to the end... whoops... let me get rid of my extra `/`... yeah! You can *see* that resource. So `@id` is just like `id`... but better.

The @type and @context Fields

Another special field is `@type`. This describes the *type* of resource, like what fields it has. And if we see two different resources that *both* have `@type` `DragonTreasure`, we know that they represent the *same* thing.

You can think of `@type` almost like a class, which we can use to find out what fields it has and the *type* of each field. Though... where *can* we actually see that info?

This is where `@context` comes in handy. Copy the context URL, paste it into your browser, and... beautiful! We get this very simple document that says that `DragonTreasure` has `name`, `description`, `value`, `coolFactor`, `createdAt`, and `isPublished` fields. If we want even *more* information about what those mean, we can follow the `@vocab` link... to get to *another* page of info.

Here, we can see *all* the classes in our API - like `DragonTreasure` - and *all* of its properties, like `name`. We can also see things like `required: false`, `readable: true`, `writeable: true` and also that it's a `string`. And we have this info for *every* field. Look:

down at `value`. We can see that this is an `integer`. This `xmls:integer` refers to *another* document, up on top, which, if we followed it, would describe `xmls:integer` in more detail.

At this point, you might be saying:

> *"Hey! This seems a lot like the OpenAPI spec doc!"*

And you're *right*. We'll talk more about that in a few minutes.

You also might be thinking:

> *"Um... I kind of get what you're saying... but this is confusing."*

And you would *also* be right! It's hard, as a mere human, to follow all of these links to find the fields and their types. But imagine what this would look like to a *machine*. It's an information *gold mine*!

Oh, and I want to mention that, if you look under `value`... `hydra:description`... it picked up the PHP documentation that we added to that field earlier.

Adding Extra Info

We can also add extra information above the class to describe this *model*. We *could* do this via PHP documentation like normal, but `ApiResource` also has some options we can pass. One is `description`. Let's *describe* this as `A rare and valuable treasure.`

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 10
11 #[ApiResource(
12     description: 'A rare and valuable treasure.'
13 )]
14 class DragonTreasure
15 {
↕ // ... lines 16 - 117
118 }
```

Now, when we refresh the page... and search for "rare" (I'll close a few things here...), yup! It added the description to the `DragonTreasure` type. And, not surprisingly, this data also shows up over here inside Swagger, because it was *also* added to the OpenAPI spec doc.

The point is, thanks to JSON-LD, we have extra fields in every response that give each resource a unique id *and* a way to discover *exactly* what that "type" looks like.

Next: we need to discuss one last piece of *theory*: what these `hydra` things mean.

Chapter 6: Hydra: Describing API Classes, Operations & More

We're looking at the JSON-LD documentation that describes our API. Right now, we know that we only have one API resource: `DragonTreasure`. But if you look down at the `supportedClasses` section, there are actually a *bunch* of supported classes. There's one called `Entrypoint`, another called `ConstraintViolation`, and another called `ConstraintViolationList`. Those last two will come up later when we talk about validation errors.

Entrypoint: Your API Homepage

But this `Entrypoint` is really interesting. It's called "The API entrypoint", and it's actually describing what the *homepage* for our API *looks like*. We don't always think about our APIs having a homepage, but they *can* and they *should*.

And, welcome to *our* API homepage - HTML style! If you scroll down to the bottom, you can see other formats. Click "JSON-LD" and... say "hello" to the API homepage in JSON-LD format! This returns an API resource called `Entrypoint`, whose whole job is to tell us where we can find info about the *other* API resources. It's like links on a homepage! You can *discover* the API by going to this `Entrypoint` and following the `@context` link... which points to this.

Hello Hydra

Anyways, the *purpose* of JSON-LD is to add those three extra fields to your API resources: `@id`, `@type`, and `@context`. Then we can leverage `@context` to point to *other* documentation to get more metadata or *more* context. For example, at the top of the JSON-LD documentation, it points to several *other* documents that add more *meaning* to JSON-LD.

And, there's one really important one here called `hydra`. Hydra is, in short an *extension* to JSON-LD: it describes even *more* fields that you can add to JSON-LD and what they mean.

Think about it: if we want to *totally* describe our API, we need to be able to communicate things like what classes we have, their properties, whether each is *readable* or *writeable*, and what *operations* each class supports. That communication is done down here... and it's actually part of *Hydra*. Yup, if you use JSON-LD by itself... it doesn't have a predefined way to advertise what your models look like. But then Hydra says:

> *"What if we allow the API classes to be described with a key called* `hydra:supportedClasses`*?"*

Here's the big picture: API Platform allows us to fetch JSON-LD API documentation that contains extra `hydra` fields. The end result is a system that *fully describes* our API. They describe the models we have, the operations... *everything*.

Why Hydra *and* OpenAPI?

And *yes*, if this sounds *very* similar to the point of OpenAPI, you're *absolutely* correct. Both of them do the same thing: describe our API. In fact, if you go to `/api/docs.json`, *this* is the OpenAPI description of our API. If we replace the `.json` with `.jsonld`, this is the *JSON-LD Hydra* description of the *same* API. Why do we have both? Hydra is a bit more powerful: there are certain things it can describe that OpenAPI can't. But OpenAPI is a lot more common and has more tools built on top of it. API platform provides *both*... in case you need them!

Next: Let's add some serious debugging tools to our API Platform setup.

Chapter 7: API Debugging with the Profiler

We're going to be doing some seriously cool and complex stuff with API platform. So before we get there, I want to make sure we have a *really* awesome debugging setup. Because... sometimes debugging APIs can be a pain! Ever made an Ajax request from JavaScript... and the endpoint explodes in a 500 error full of HTML? Yea, not super helpful.

Installing the Profiler

One of the best features of Symfony is its Web Debug Toolbar. But if we're building an API... there's not going to be a Web Debug Toolbar on the bottom of these JSON responses. So should we even bother installing that package? The answer is: absolutely!

Spin over to the terminal and run:

```
composer require debug
```

This is another Symfony Flex alias that installs `symfony/debug-pack`. If you pop over to your `composer.json` file, this installed a bunch of good stuff: a logger... then down in `require-dev`, it also added DebugBundle and WebProfilerBundle, which is the most important thing for what we'll talk about.

composer.json

```
{
// ... lines 2 - 83
    "require-dev": {
        "symfony/debug-bundle": "6.2.*",
// ... line 86
        "symfony/monolog-bundle": "^3.0",
        "symfony/stopwatch": "6.2.*",
        "symfony/web-profiler-bundle": "6.2.*"
    }
}
```

AJAX Requests in the Web Debug Toolbar

Head back to our documentation homepage and refresh. Sweet! We get the Web Debug Toolbar down on the bottom! Though... that doesn't really help us because... all of this info is *literally* for the *documentation* page itself. Not particularly useful.

What we *really* want is all of this profiler info for any API request we make. And that's *super* possible. Use the GET collection endpoint. Hit "Try it out" and then watch closely down here on the Web Debug Toolbar. When I hit "Execute"... boom! Because that made an AJAX request the AJAX icon on the Web Debug Toolbar showed up! Want to see *all* the deep profiler info for that request? Just click the little link on that panel. Yup, as you can see here, we're now looking at the profiler for the `GET /api/dragon_treasures` API call.

API Platform & Serializer in the Profiler

And there's lots of cool stuff in here. Obviously, there's the Performance section and all the normal goodies. But one of *my* favorite parts is the "Exception" tab. If you have an API endpoint and that API endpoint explodes with an error - it happens - you can open this part of the profiler to see the *full* beautiful HTML exception: including the stack trace in all its glory. *So* handy.

I have two other favorite spots when working on an API. The first, no surprise, is the "API Platform" tab. This gives us info about the configuration for all of our API resources. We're going to talk more about this config, but this shows you the *current* and possible options that you could put inside of this `ApiResource` attribute. That's pretty cool. For example, this shows a `description` option...

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 10
11 #[ApiResource(
12     description: 'A rare and valuable treasure.'
13 )]
14 class DragonTreasure
15 {
↕ // ... lines 16 - 117
118 }
```

and we already have that!

The other really useful section in the profiler is relatively new: it's for the "Serializer". We're going to be talking a *lot* about Symfony's serializer and this tool will help us get a look at what's going on internally.

Finding the Profiler for an API Request

So the big takeaway is that every API request actually has a profiler! And there are a few ways to find it. We just say the first: if you're making an AJAX request - even if it's via your own JavaScript - then you can use the web debug toolbar.

And, if you look down here a bit, these are the response headers our API returned. One is called `X-Debug-Token-Link` which offers us a *second* way to find the profiler for any API request. This is exactly the URL we were just at.

The last way is... maybe the simplest. Suppose we go directly to `/api/dragon_treasure.json`. From here, there's no easy way to get to the profiler. But now, open up a new tab and manually go to `/_profiler`. Yup! This shows us a list of the latest request to our app... include the GET request we just made! If you click the little token link... boom! We're inside *that* profiler.

You can click this "Last 10" at any point to get back to that list... and find whichever request you need.

Sweet debugging tools, check! Next: let's talk about the concept of "operations" in API platform, which represent these six endpoints. How can we configure these? Or disable one? Or add more? Let's find out!

Chapter 8: Operations / Endpoints

API Platform works by taking a class like `DragonTreasure` and saying that you want to expose it as a *resource* in your API. We do that by adding the `ApiResource` attribute:

src/Entity/DragonTreasure.php

```
// ... lines 1 - 10
#[ApiResource(
    description: 'A rare and valuable treasure.'
)]
class DragonTreasure
{
// ... lines 16 - 117
}
```

Right now, we're putting this above a Doctrine entity, though, in a future tutorial, we'll learn that you can really put `ApiResource` above *any* class.

Hello Operations

Out-of-the-box, every `ApiResource` includes 6 endpoints, which API Platform calls operations. You can actually see these in the profiler. This is the profiler for `GET /api/dragon_treasures.json`. Click on the "API Platform" section. On top, we see metadata for this API resource. Below, we see the operations. This... is more info than we need right now, but there's `Get`, `GetCollection`, `Post`, `Put`, `Patch` and finally `Delete`. These are the same things we see on the Swagger documentation.

Let's take a quick look at these. First, which operations *return* data? Actually, *all* of them - except for `Delete`. This `Get`, the `Post`, `Put` and `Patch` endpoints all return a *single* resource - so a single treasure. And `GET /api/dragon_treasures` returns a *collection*.

Which endpoints do we *send* data to when we use them? That's `POST` to create, and `PUT` and `PATCH` to update. We don't send any data for `DELETE` or either `GET` operation.

PUT vs PATCH

Most of the endpoints are pretty self-explanatory: get a collection of treasures, a single treasure, create a treasure and delete a treasure. The only confusing ones are put versus patch. `PUT` says "replaces" and `PATCH` says "updates". That... sounds like two ways of saying the same thing!

> **Tip**
>
> In API Platform 4, `PUT` will become a "replace": meaning if you *only* sent a single field, all of the other fields in your resource will be set to null: your object is completely "replaced" by the JSON you send. Starting in API Platform 3.1, you can "opt into" this new behavior by adding an `extraProperties` option to every `ApiResource`:
>
> ```
> #[ApiResource(
>     // ...
>
>     extraProperties: [
>         'standard_put' => true,
>     ],
> )]#
> ```

The topic of PUT versus PATCH in APIs can get spicy. But in API Platform, at least today, PUT and PATCH work the same: they're both used to update a resource. And we'll see them in action along the way.

Customizing Operations

One of the things that you might want to do is customize or *remove* some of these operations... or even add *more* operations. How could we do that? As we saw on the profiler, each operation is backed by a *class*.

Back over above the `DragonTreasure` class, after `description`, add an `operations` key. Notice that I'm getting auto-completion for the options because these are *named* arguments to the constructor of the `ApiResource` class. I'll show you that in a minute.

Set this to an array and then repeat *every* operation we currently have. So, `new Get()`, hit tab to auto-complete that, `GetCollection`, `Post`, `Put`, `Patch` and `Delete`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 5
6  use ApiPlatform\Metadata\Delete;
7  use ApiPlatform\Metadata\Get;
8  use ApiPlatform\Metadata\GetCollection;
9  use ApiPlatform\Metadata\Patch;
10 use ApiPlatform\Metadata\Post;
11 use ApiPlatform\Metadata\Put;
↕ // ... lines 12 - 16
17 #[ApiResource(
18     description: 'A rare and valuable treasure.',
19     operations: [
20         new Get(),
21         new GetCollection(),
22         new Post(),
23         new Put(),
24         new Patch(),
25         new Delete(),
26     ]
27 )]
28 class DragonTreasure
29 {
↕ // ... lines 30 - 131
132 }
```

Now, if we move over to the Swagger documentation and refresh... absolutely nothing changes! That's what we wanted. We've just repeated *exactly* the *default* configuration. But *now* we're free to customize things. For example, suppose we don't want treasures to be deleted... because a dragon would never allow their treasure to be stolen. Remove `Delete`.. and I'll even remove the `use` statement.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 5
6  use ApiPlatform\Metadata\Get;
7  use ApiPlatform\Metadata\GetCollection;
8  use ApiPlatform\Metadata\Patch;
9  use ApiPlatform\Metadata\Post;
10 use ApiPlatform\Metadata\Put;
↕ // ... lines 11 - 15
16 #[ApiResource(
17     description: 'A rare and valuable treasure.',
18     operations: [
19         new Get(),
20         new GetCollection(),
21         new Post(),
22         new Put(),
23         new Patch(),
24     ]
25 )]
26 class DragonTreasure
27 {
↕ // ... lines 28 - 129
130 }
```

Now when we refresh, the `DELETE` operation is gone.

ApiResource Options

Ok, so every attribute we use is *actually* a class. And knowing that is *powerful*. Hold command or control and click on `ApiResource` to open it. This is *really* cool. Every argument to the constructor is an *option* that we can pass to the attribute. And almost all of these have a link to the documentation where you can read more. We'll talk about the most important items, but this is a great resource to know about.

Changing the shortName

One argument is called `shortName`. If you look over at Swagger, our "model" is currently known as `DragonTreasure`, which obviously matches the class. This is called the "short name". And by default, the URLs - `/api/dragon_treasures` - are generated from that.

Let's say that we instead want to shorten all of this to just "treasure". No problem: set `shortName` to `Treasure`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 15
16 #[ApiResource(
17     shortName: 'Treasure',
↕ // ... lines 18 - 25
26 )]
27 class DragonTreasure
28 {
↕ // ... lines 29 - 130
131 }
```

As soon as we do that, watch the name and URLs. Nice. This resource is now known as "Treasure" and the URLs updated to reflect that.

Operation Options

Though, that's not the only way to configure the URLs. Just like with `ApiResource`, each operation is *also* a class. Hold Command (or Ctrl) and Click to open up the `Get` class. Once again, these constructor arguments are options... and most have documentation.

One important argument is `uriTemplate`. Yup, we can control what the URL looks like on an operation by operation basis.

Check it out. Remember, `Get` is how you fetch a *single* resource. Add `uriTemplate` set to `/dragon-plunder/{id}` where that last part will be the placeholder for the dynamic id. For `GetCollection`, let's *also* pass `uriTemplate` set to `/dragon-plunder`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 15
16 #[ApiResource(
↕ // ... lines 17 - 18
19     operations: [
20         new Get(uriTemplate: '/dragon-plunder/{id}'),
21         new GetCollection(uriTemplate: '/dragon-plunder'),
↕ // ... lines 22 - 24
25     ]
26 )]
27 class DragonTreasure
28 {
↕ // ... lines 29 - 130
131 }
```

Ok! Let's go check the docs! Beautiful! The other operations keep the old URL, but those use the *new* style. Later, when we talk about subresources, we'll go deeper into `uriTemplate` and its sister option `uriVariables`.

Ok... since it's a bit silly to have two operations with weird URLs, let's remove that customization.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 15
16 #[ApiResource(
↕ // ... lines 17 - 18
19     operations: [
20         new Get(),
21         new GetCollection(),
↕ // ... lines 22 - 24
25     ]
26 )]
27 class DragonTreasure
28 {
↕ // ... lines 29 - 130
131 }
```

Now that we know a bunch about `ApiResource` and these operations, it's time to talk about the *heart* of API Platform: Symfony's serializer. That's next.

Chapter 9: The Serializer

The key behind how API platform turns our objects into JSON... and also how it transforms JSON back into objects is Symfony's Serializer. `symfony/serializer` is a standalone component that you can use outside of API platform and it's *awesome*. You give it any input - like an object or something else - and it transform that into any format, like `JSON`, `XML` or `CSV`.

The Internals of the Serializer

As you can see in this fancy diagram, it goes through two steps. First, it takes your data and *normalizes* it into an array. Second, it *encodes* that into the final format. It can also do the same thing in reverse. If we're starting with JSON, like we're sending JSON to our API, it first *decodes* it to an array and then *denormalizes* it back into an object.

For all of this to happen, internally, there are many different normalizer objects that know how to work with different data. For example, there's a `DateTimeNormalizer` that's really great at handling `DateTime` objects. Check it out: our entity has a `createdAt` field, which is a `DateTime` object:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 26
27 class DragonTreasure
28 {
↕ // ... lines 29 - 48
49     #[ORM\Column]
50     private ?\DateTimeImmutable $plunderedAt = null;
↕ // ... lines 51 - 130
131 }
```

If you look at our API, when we try the `GET` endpoint, this is returned as a special date time *string*. The `DateTimeNormalizer` is responsible for doing that.

Figuring out Which Fields to Serialize

There's also another really important normalizer called the `ObjectNormalizer`. Its job is to read properties off of an object so that *those* properties can be normalized. To do that, it uses another component called `property-access`. *That* component is smart.

For example, looking at our API, when we make a GET request to the collection endpoint, one of the fields it returns is `name`. But if we look at the class, `name` is a *private* property:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 26
27 class DragonTreasure
28 {
↕ // ... lines 29 - 33
34     #[ORM\Column(length: 255)]
35     private ?string $name = null;
↕ // ... lines 36 - 130
131 }
```

So how the heck is it reading that?

That's where the `PropertyAccess` component comes in. It first looks to see if the `name` property is public. And if it's not, it then looks for a `getName()` method:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 26
27 class DragonTreasure
28 {
↕ // ... lines 29 - 33
34     #[ORM\Column(length: 255)]
35     private ?string $name = null;
↕ // ... lines 36 - 59
60     public function getName(): ?string
61     {
62         return $this->name;
63     }
↕ // ... lines 64 - 130
131 }
```

So *that* is what's actually called when building the JSON.

The same thing happens when we *send* JSON, like to create or update a `DragonTreasure`. PropertyAccess looks at each field in the JSON and, if that field is settable, like via a `setName()` method, it sets it:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 26
27 class DragonTreasure
28 {
↕ // ... lines 29 - 33
34     #[ORM\Column(length: 255)]
35     private ?string $name = null;
↕ // ... lines 36 - 59
60     public function getName(): ?string
61     {
62         return $this->name;
63     }
64
65     public function setName(string $name): self
66     {
67         $this->name = $name;
68
69         return $this;
70     }
↕ // ... lines 71 - 130
131 }
```

And, it's even a bit cooler than that: it will even look for getter or setter methods that don't correspond to *any* real property! You can use this to create "extra" fields in your API that don't exist as properties in your class.

Adding a Virtual "textDescription" Field

Let's try that! Pretend that, when we're creating or editing a treasure, instead of sending a `description` field, we want to be able to send a `textDescription` field that contains plaintext... but with line breaks. Then, in our code, we'll transform those lines breaks into HTML `<br>` tags.

Let me show you what I mean. Copy the `setDescription()` method. Then, below, paste and call this new method `setTextDescription()`. It's basically going to set the `description` property... but call `nl2br()` on it first. That function literally transforms new lines into `<br>` tags. If you've been around as long as I have, you remember when `nl2br` was super cool:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 26
27 class DragonTreasure
28 {
↕ // ... lines 29 - 83
84     public function setTextDescription(string $description): self
85     {
86         $this->description = nl2br($description);
87 
88         return $this;
89     }
↕ // ... lines 90 - 137
138 }
```

Anyways, with *just* that change, refresh the documentation and open the POST or PUT endpoints. Woh! We have a new field called `textDescription`! Yup! The serializer *saw* the `setTextDescription()` method and determined that `textDescription` is a "settable" virtual property!

However, we *don't* see this on the GET endpoint. And that's perfect! There is no `getTextDescription()` method, so there will *not* be a new field here. The new field is *writable*, but not readable.

Let's take this endpoint for a spin! First... I need to execute the GET collection endpoint so I can see what ids we have in the database. Perfect: I have a Treasure with ID 1. Close this up. Let's try the PUT endpoint to do our first update. When you use the PUT endpoint, you *don't* need send every field: only the fields you want to change.

Tip

If you're starting a new API Platform project, this `PUT` request will fail! You can try a `PATCH` request instead. This relates to a new `standard_put` config in `config/packages/api_platform.yaml`, which we talk about a bit later in the tutorial.

Pass `textDescription`... and I'll include `\n` to represent some new lines in JSON.

When we try it, yes! 200 status code. And check it out: the `description` field has those `<br>` tags!

Removing Fields

Ok, so now that we have `setTextDescription()`... maybe that's the *only* way that we want to allow that field to be set. To enforce that, eradicate the `setDescription()` method.

Now when we refresh... and look at the PUT endpoint, we still have `textDescription`, but the `description` field is gone! The serializer realizes that it's no longer settable and removed it from our API. It would still be *returned* because it's something that we can read, but it's no longer writeable.

This is all *really* awesome. *We* simply worry about writing our class the way we want then API Platform builds our API accordingly.

Making the plunderedAt Field Readonly

Ok, what else? Well, it *is* a little weird that we can set the `createdAt` field: that's usually set internally and automatically. Let's fix that.

Oh, but, ya know what? I meant to call this field `plunderedAt`. I'll refactor and rename that property... then let PhpStorm also rename my getter and setter methods.

Cool! This will *also* cause the column in my database to change... so spin over to your console and run:

```
symfony console make:migration
```

I'll live dangerously and run that immediately:

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

Done! Thanks to that rename... over in the API, excellent: the field is now `plunderedAt`.

Ok, so forget about the API for a moment: let's just do a little cleanup. The purpose of this `plunderedAt` field is for it to be set automatically whenever we create a new `DragonTreasure`.

To do that, create a `public function __construct()` and, inside, say `this->plunderedAt = new DateTimeImmutable()`. And now we don't need the `= null` on the property.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 26
27 class DragonTreasure
28 {
↕ // ... lines 29 - 48
49     #[ORM\Column]
50     private \DateTimeImmutable $plunderedAt;
↕ // ... lines 51 - 54
55     public function __construct()
56     {
57         $this->plunderedAt = new \DateTimeImmutable();
58     }
↕ // ... lines 59 - 128
129 }
```

And if we search for `setPlunderedAt`, we don't really need that method anymore! Remove it!

This now means that the `plunderedAt` property is readable but not writeable. So, no shocker, when we refresh and open up the `PUT` or `POST` endpoint, `plunderedAt` is absent. But if we look at what the model would look like if we *fetched* a treasure, `plunderedAt` is still there.

Adding a Fake "Date Ago" Field

All right, one more goal! Let's add a virtual field called `plunderedAtAgo` that returns a human-readable version of the date, like "two months ago". To do this, we need to install a new package:

```
composer require nesbot/carbon
```

Once this finishes... find the `getPlunderedAt()` method, copy it, paste below, it will return a `string` and call it `getPlunderedAtAgo()`. Inside, return `Carbon::instance($this->getPlunderedAt()))` then `->diffForHumans()`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 use Carbon\Carbon;
↕ // ... lines 13 - 27
28 class DragonTreasure
29 {
↕ // ... lines 30 - 118
119     /**
120      * A human-readable representation of when this treasure was plundered.
121      */
122     public function getPlunderedAtAgo(): string
123     {
124         return Carbon::instance($this->plunderedAt)->diffForHumans();
125     }
↕ // ... lines 126 - 137
138 }
```

So, as we now understand, there is *no* `plunderedAtAgo` property... but the `serializer` *should* see this as readable via its getter and expose it. Oh, and while I'm here, I'll add a little documentation above to describe the field's meaning.

Ok, let's try this. As soon as we refresh and open a `GET` endpoint, we see the new field under the example! We can also see the fields we'll receive down in the Schemas section. Back up, let's try the `GET` endpoint with ID `one`. And... how cool is that?

Next: what if we *do* want to have certain getter or setter methods in our class, like `setDescription()`, but we do *not* want that to be part of our API? The answer: serialization groups.

Chapter 10: Serialization Groups: Choosing Fields

Right now, whether or not a field in our class is readable or writable in the API is *entirely* determined by whether or not that property is readable or writable in our class (basically, whether or not it has a getter or setter method). But what if you *need* a getter or setter... but *don't* want that field exposed in the API? For that, we have two options.

A DTO Class?

Option número uno: create a DTO class for the API resource. This is something we'll save for another day... in a future tutorial. But in a nutshell, it's where you create a dedicated class for your `DragonTreasure` API... and then move the `ApiResource` attribute onto *that*. The key thing is that you'll design the new class to look *exactly* like your API... because modeling your API will be its *only* job. It takes a little more work to set things up, but the advantage is that you then have a dedicated class for your API. Done!

Hello Serialization Groups

The *second* solution, and the one we're going to use, is *serialization groups*. Check it out. Over on the `ApiResource` attribute, add a new option called `normalizationContext`. If you recall, "normalization" is the process of going from an object to an array, like when you're making a `GET` request to read a treasure. The `normalizationContext` is basically *options* that are passed to the serializer during that process. And the *one* option that's most important is `groups`. Set that to one group called `treasure:read`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 16
17 #[ApiResource(
↕ // ... lines 18 - 26
27     normalizationContext: [
28         'groups' => ['treasure:read'],
29     ]
30 )]
31 class DragonTreasure
32 {
↕ // ... lines 33 - 140
141 }
```

We'll talk about what this does in a minute. But you can see the pattern I'm using for the group: the name of the class (it could be `dragon_treasure` if we wanted) then `:read`... because normalization means that we're *reading* this class. You can name these groups however you want: this is my standard.

So... what does that *do*? Let's find out! Refresh the documentation... and, to make life easier, go to the URL: `/api/dragon_treasures.jsonld`. Whoops! It's just `treasures.jsonld` now. There we go. And... absolutely nothing is returned! Ok, we have the hydra fields, but this `hydra:member` contains the array of treasures. It *is* returning one treasure... but other than `@id` and `@type`... there are no actual fields!

How Serialization Groups Work

Here's the deal. As soon as we add a `normalizationContext` with a group, when our object is normalized, the serializer will *only* include properties that have this group on it. And since we haven't added *any* groups to our properties, it returns *nothing*.

How do we add groups? With *another* attribute! Above the `$name` property, say `#[Groups]`, hit "tab" to add its `use` statement and then `treasure:read`. Repeat this above the `$description` field... because we want *that* to be readable... and then the `$value` field... and finally `$coolFactor`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 14
15 use Symfony\Component\Serializer\Annotation\Groups;
↕ // ... lines 16 - 31
32 class DragonTreasure
33 {
↕ // ... lines 34 - 39
40     #[Groups(['treasure:read'])]
41     private ?string $name = null;
↕ // ... lines 42 - 43
44     #[Groups(['treasure:read'])]
45     private ?string $description = null;
↕ // ... lines 46 - 50
51     #[Groups(['treasure:read'])]
52     private ?int $value = null;
↕ // ... lines 53 - 54
55     #[Groups(['treasure:read'])]
56     private ?int $coolFactor = null;
↕ // ... lines 57 - 145
146 }
```

Good start. Move over and refresh the endpoint. Now... got it! We see `name`, `description`, `value`, and `coolFactor`.

DenormalizationContext: Controlling Writable Groups

We now have control over which fields are *readable*... and we can do the same thing to choose which fields should be *writeable* in the API. That's called "de-normalization", and I bet you can guess what we're going to do. Copy `normalizationContext`, paste, change it to `denormalizationContext`... and use `treasure:write`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 17
18 #[ApiResource(
↕ // ... lines 19 - 30
31     denormalizationContext: [
32         'groups' => ['treasure:write'],
33     ]
34 )]
35 class DragonTreasure
36 {
↕ // ... lines 37 - 148
149 }
```

Now head down to the `$name` property and add `treasure:write`. I'm going to skip `$description` (remember that we actually *deleted* our `setDescription()` method earlier on purpose)... but add this to `$value`... and `$coolFactor`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 34
35 class DragonTreasure
36 {
↕ // ... lines 37 - 42
43     #[Groups(['treasure:read', 'treasure:write'])]
44     private ?string $name = null;
↕ // ... lines 45 - 53
54     #[Groups(['treasure:read', 'treasure:write'])]
55     private ?int $value = null;
↕ // ... lines 56 - 57
58     #[Groups(['treasure:read', 'treasure:write'])]
59     private ?int $coolFactor = null;
↕ // ... lines 60 - 148
149 }
```

Oh, it's *mad* at me! As soon as we pass *multiple* groups, we need to make this an *array*. Add some `[]` around those three properties. Much happier.

To check if this is A-OK, refresh the documentation... open up the `PUT` endpoint, and... sweet! We see `name`, `value`, and `coolFactor`, which are currently the *only* fields that are *writable* in our API.

Adding Groups To Methods

We *are* missing a few things, though. Earlier, we made a `getPlunderedAtAgo()` method...

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 34
35 class DragonTreasure
36 {
↕ // ... lines 37 - 132
133     public function getPlunderedAtAgo(): string
134     {
135         return Carbon::instance($this->plunderedAt)->diffForHumans();
136     }
↕ // ... lines 137 - 148
149 }
```

and we want this to be included when we *read* our resource. Right now, if we we check the endpoint, it's *not* there.

To fix this, we can *also* add groups above methods. Say `#[Groups(['treasure:read'])]`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 34
35 class DragonTreasure
36 {
↕ // ... lines 37 - 132
133     #[Groups(['treasure:read'])]
134     public function getPlunderedAtAgo(): string
135     {
136         return Carbon::instance($this->plunderedAt)->diffForHumans();
137     }
↕ // ... lines 138 - 149
150 }
```

And when we go check... *voilà*, it pops up.

Let's also find the `setTextDescription()` method... and do the same thing: `#[Groups([treasure:write])]`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 34
35 class DragonTreasure
36 {
↕ // ... lines 37 - 93
94     #[Groups(['treasure:write'])]
95     public function setTextDescription(string $description): self
96     {
↕ // ... lines 97 - 99
100     }
↕ // ... lines 101 - 150
151 }
```

Awesome! If we head back to the documentation, the field is not currently there... but when we refresh... and check out the `PUT` endpoint again... `textDescription` is *back*!

Re-Adding Methods

Hey, now we can re-add any of the getter or setter methods we removed earlier! Like, maybe I *do* need a `setDescription()` method in my code for something. Copy `setName()` to be

lazy, paste and change "name" to "description" in a few places.

Got it! And even though we have that setter back, when we look at the `PUT` endpoint, `description` *doesn't* show up. We have complete control over our fields thanks to the denormalization groups. Do the same thing for `setPlunderedAt()`... because sometimes it's handy - in data fixtures especially - to be able to *set* this manually.

And... done!

Adding Field Defaults

So we know that *fetching* a resource works. Now let's see if we can *create* a new resource. Click on the `POST` endpoint, hit "Try it out", and... let's fill in some info about our new treasure, which is, of course, a `Giant jar of pickles`. This is *very* valuable and has a `coolFactor` of `10`. I'll also add a description... though this jar of pickles speaks for itself.

When we try this... oh, dear... we get a 500 error:

> *"An exception occurred while executing a query: Not null violation, `null` value in column `isPublished`."*

We slimmed our API down to *only* the fields that we want *writeable*... but there's still one property that *must* be set in the database. Scroll up and find `isPublished`. Yup, it currently defaults to `null`. Change that to `= false`... and now the property will *never* be `null`.

If we try it... the `Giant jar of pickles` is pickled into the database! It works!

Next: let's explore a few more cool serialization tricks to give us even more control.

Chapter 11: Serialization Tricks

We've sort of tricked the system to allow a `textDescription` field when we send data. This is made possible thanks to our `setTextDescription()` method, which runs `nl2br()` on the description that's sent to our API. This means that the user *sends* a `textDescription` field when editing or creating a treasure... but they *receive* a `description` field when reading.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 34
35 class DragonTreasure
36 {
↕ // ... lines 37 - 93
94     #[Groups(['treasure:write'])]
95     public function setTextDescription(string $description): self
96     {
97         $this->description = nl2br($description);
98 
99         return $this;
100     }
↕ // ... lines 101 - 150
151 }
```

And that's totally *fine*: you're allowed to have different input fields versus output fields. But it *would* be a bit cooler if, in this case, *both* were just called `description`.

SerializedName: Controlling the Field Name

So... can we control the *name* of a field? *Absolutely*! We do this, as you may have predicted, via another wonderful attribute. This one is called `SerializedName`. Pass it `description`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 15
16 use Symfony\Component\Serializer\Annotation\SerializedName;
↕ // ... lines 17 - 35
36 class DragonTreasure
37 {
↕ // ... lines 38 - 101
102     #[SerializedName('description')]
103     #[Groups(['treasure:write'])]
104     public function setTextDescription(string $description): self
105     {
106         $this->description = nl2br($description);
107 
108         return $this;
109     }
↕ // ... lines 110 - 166
167 }
```

This won't change how the field is *read*, but if we refresh the docs... and look at the `PUT` endpoint... yep! We can now *send* a field called `description`.

Constructor Arguments

What about constructor arguments in our entity? When we make a `POST` request, for example, we know it uses the setter methods to write the data onto the properties.

Now try this: find `setName()` and remove it. Then go to the constructor and add a `string $name` argument there instead. Below, say `$this->name = $name`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 35
36 class DragonTreasure
37 {
↕ // ... lines 38 - 67
68     public function __construct(string $name)
69     {
70         $this->name = $name;
71         $this->plunderedAt = new \DateTimeImmutable();
72     }
↕ // ... lines 73 - 160
161 }
```

From an object-oriented perspective, the field can be passed when the object is *created*, but after that, it's read-only. Heck, if you wanted to get fancy, you could add `readonly` to the property.

Let's see what this looks like in our documentation. Open up the `POST` endpoint. It looks like we can *still* send a `name` field! Test by hitting "Try it out"... and let's add a `Giant slinky` we won from a real-life giant in... a rather tense poker match. It's pretty valuable, has a `coolFactor` of `8`, and give it a `description`. Let's see what happens. Hit "Execute" and... it worked! And we can see in the response that the `name` *was* set. How is that possible?

Well, if you go down and look at the `PUT` endpoint, you'll see that it *also* advertises `name` here. But... go up find the id of the treasure we just created - its 4 for me, put 4 in here to edit... then send *just* the name field to change it. And... it *didn't* change! Yup, just like with our code, once a `DragonTreasure` is created, the name *can't* be changed.

But... how did the `POST` request set the name... if there's no setter? The *answer* is that the serializer is smart enough to set constructor arguments... *if* the argument name matches the property name. Yup, the fact that the arg is called `name` and the property is *also* called `name` is what makes this work.

Watch: change the argument to `treasureName` in both places:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 35
36 class DragonTreasure
37 {
↕ // ... lines 38 - 67
68     public function __construct(string $treasureName)
69     {
70         $this->name = $treasureName;
71         $this->plunderedAt = new \DateTimeImmutable();
72     }
↕ // ... lines 73 - 160
161 }
```

Now, spin over, refresh, and check out the POST endpoint. The field is *gone*. API Platform sees that we have a `treasureName` argument that *could* be sent, but since `treasureName` doesn't correspond to any property, that field doesn't have any serialization groups. So it's not used. I'll change that back to `name`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 35
36 class DragonTreasure
37 {
↕ // ... lines 38 - 67
68     public function __construct(string $name)
69     {
70         $this->name = $name;
71         $this->plunderedAt = new \DateTimeImmutable();
72     }
↕ // ... lines 73 - 160
161 }
```

By using `name`, it looks at the `name` property, and reads *its* serialization groups.

Optional Vs Required Constructor Args

However, there *is* still one problem with constructor arguments that you should be aware of. Refresh the docs.

What would happen if our user *doesn't* pass a `name` at all? Hit "Execute" to find out. Ok! We get an error with a 400 status code... *but* it's not a very *good* error. It says:

> *"Cannot create an instance of `App\Entity\DragonTreasure` from serialized data because its constructor requires parameter `name` to be present."*

That's... actually *too* technical. What we *really* want is to allow *validation* to take care of this... and we'll talk about validation soon. But in order for validation to work, the serializer needs to be able to do its job: it needs to be able to *instantiate* the object:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 35
36 class DragonTreasure
37 {
↕ // ... lines 38 - 67
68     public function __construct(string $name = null)
69     {
70         $this->name = $name;
71         $this->plunderedAt = new \DateTimeImmutable();
72     }
↕ // ... lines 73 - 160
161 }
```

Ok, try this now... better! Ok, it's *worse* - a 500 error - but we'll fix that with validation in a few minutes. The point is: the serializer *was* able to create our object.

Next: To help us while we're developing, let's add a rich set of data fixtures. Then we'll play with a great feature that API Platform gives us for *free*: *pagination*

Chapter 12: Pagination & Foundry Fixtures

We're going to start doing more with our API... so it's time to bring this thing to life with some data fixtures!

For this, I like to use Foundry along with DoctrineFixturesBundle. So, run

```
composer require foundry orm-fixtures --dev
```

to install both as `dev` dependencies. Once that finishes, run

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

Adding the Foundry Factory

If you haven't used Foundry before, for each entity, you create a *factory* class that's really good at *creating* that entity. I'll hit zero to generate the one for `DragonTreasure`.

The end result is a new `src/Factory/DragonTreasureFactory.php` file:

src/Factory/DragonTreasureFactory.php

```
// ... lines 1 - 2
namespace App\Factory;

use App\Entity\DragonTreasure;
use App\Repository\DragonTreasureRepository;
use Zenstruck\Foundry\ModelFactory;
use Zenstruck\Foundry\Proxy;
use Zenstruck\Foundry\RepositoryProxy;

/**
 * @extends ModelFactory<DragonTreasure>
 *
 * @method        DragonTreasure|Proxy create(array|callable $attributes = [])
 * @method static DragonTreasure|Proxy createOne(array $attributes = [])
 * @method static DragonTreasure|Proxy find(object|array|mixed $criteria)
 * @method static DragonTreasure|Proxy findOrCreate(array $attributes)
 * @method static DragonTreasure|Proxy first(string $sortedField = 'id')
 * @method static DragonTreasure|Proxy last(string $sortedField = 'id')
 * @method static DragonTreasure|Proxy random(array $attributes = [])
 * @method static DragonTreasure|Proxy randomOrCreate(array $attributes = [])
 * @method static DragonTreasureRepository|RepositoryProxy repository()
 * @method static DragonTreasure[]|Proxy[] all()
 * @method static DragonTreasure[]|Proxy[] createMany(int $number, array|callable $attributes = [])
 * @method static DragonTreasure[]|Proxy[] createSequence(array|callable $sequence)
 * @method static DragonTreasure[]|Proxy[] findBy(array $attributes)
 * @method static DragonTreasure[]|Proxy[] randomRange(int $min, int $max, array $attributes = [])
 * @method static DragonTreasure[]|Proxy[] randomSet(int $number, array $attributes = [])
 */
final class DragonTreasureFactory extends ModelFactory
{
// ... lines 32 - 36
    public function __construct()
    {
        parent::__construct();
    }
// ... lines 41 - 46
    protected function getDefaults(): array
    {
        return [
            'coolFactor' => self::faker()->randomNumber(),
            'description' => self::faker()->text(),
```

```
52              'isPublished' => self::faker()->boolean(),
53              'name' => self::faker()->text(255),
54              'plunderedAt' =>
    \DateTimeImmutable::createFromMutable(self::faker()->dateTime()),
55              'value' => self::faker()->randomNumber(),
56          ];
57      }
↕   // ... lines 58 - 61
62      protected function initialize(): self
63      {
64          return $this
65              // ->afterInstantiate(function(DragonTreasure
    $dragonTreasure): void {})
66          ;
67      }
68
69      protected static function getClass(): string
70      {
71          return DragonTreasure::class;
72      }
73  }
```

This class is just really good at creating `DragonTreasure` objects. It even has a bunch of nice random data ready to be used!

To make this *even* fancier, I'm going to paste over with some code that I've dragon-ized. Oh, and we also need a `TREASURE_NAMES` constant... which I'll also paste on top. You can grab all of this from the code block on this page.

src/Factory/DragonTreasureFactory.php

```
// ... lines 1 - 29
final class DragonTreasureFactory extends ModelFactory
{
    private const TREASURE_NAMES = ['pile of gold coins', 'diamond-encrusted throne', 'rare magic staff', 'enchanted sword', 'set of intricately crafted goblets', 'collection of ancient tomes', 'hoard of shiny gemstones', 'chest filled with priceless works of art', 'giant pearl', 'crown made of pure platinum', 'giant egg (possibly a dragon egg?)', 'set of ornate armor', 'set of golden utensils', 'statue carved from a single block of marble', 'collection of rare, antique weapons', 'box of rare, exotic chocolates', 'set of ornate jewelry', 'set of rare, antique books', 'giant ball of yarn', 'life-sized statue of the dragon itself', 'collection of old, used toothbrushes', 'box of mismatched socks', 'set of outdated electronics (such as CRT TVs or floppy disks)', 'giant jar of pickles', 'collection of novelty mugs with silly sayings', 'pile of old board games', 'giant slinky', 'collection of rare, exotic hats'];
// ... lines 33 - 46
    protected function getDefaults(): array
    {
        return [
            'coolFactor' => self::faker()->numberBetween(1, 10),
            'description' => self::faker()->paragraph(),
            'isPublished' => self::faker()->boolean(),
            'name' => self::faker()->randomElement(self::TREASURE_NAMES),
            'plunderedAt' => \DateTimeImmutable::createFromMutable(self::faker()->dateTimeBetween('-1 year')),
            'value' => self::faker()->numberBetween(1000, 1000000),
        ];
    }
// ... lines 58 - 72
}
```

Ok, so this class is done. Step two: to actually *create* some fixtures, open `src/DataFixtures/AppFixtures.php`. I'll clear out the `load()` method. All we need is: `DragonTreasureFactory::createMany(40)` to create a healthy trove of 40 treasures:

src/DataFixtures/AppFixtures.php

```
// ... lines 1 - 2
namespace App\DataFixtures;

use App\Factory\DragonTreasureFactory;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        DragonTreasureFactory::createMany(40);
    }
}
```

Let's try this thing! Back at your terminal, run:

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

Say "yes" and... it looks like it worked! Back on our API docs, refresh... then let's try the `GET` collection endpoint. Hit execute.

We have Pagination!

Oh, so cool! Look at all those beautiful treasures! Remember, we added *40*. But if you look closely... even though the `IDs` don't start at 1, we can see that there are definitely *less* than 40 here. The response *says* `hydra:totalItems: 40`, but it only shows 25.

Down here, this `hydra:view` kind of explains why: there's built-in pagination! Right now we're looking at page 1.. and we can also see the URLs for the last page and the *next* page.

So yes, API endpoints that return a collection need pagination... just like a website. And with API Platform, it just works.

To play with this, let's go to `/api/treasures.jsonld`. This is page 1... and then we can add `?page=2` to see page 2. That's the easiest thing I'll do all day.

Digging Into API Platform Configuration

Now if you need to, you can *change* a bunch of pagination options. Let's see if we can tweak the number of items per page from 25 to 10.

To start digging into the config, open up your terminal and run:

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

There are a lot of things that you can configure on API Platform. And this command shows us the *current* configuration. So for example, you can add a `title` and `description` to your API. This becomes part of the OpenAPI Spec... and so it shows up on your documentation.

If you search for `pagination` - we don't want the one under `graphql`... we want the one under `collection` - we can see several pagination-related options. But, again, this is showing us the *current* configuration... it doesn't necessarily show us *all* possible keys.

To see *that*, instead of `debug:config`, run:

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

`debug:config` shows you the current configuration. `config:dump` shows you a full tree of *possible* configuration. Now... we see `pagination_items_per_page`. That sounds like what we want!

This is actually *really* cool. All of these options live under something called `defaults`. And these are snake-case versions of the *exact* same options that you'll find inside the `ApiResource` attribute. Setting any of these `defaults` in the config causes that to be the default value passed to that option for *every* `ApiResource` in your system. Pretty cool.

So, if we wanted to change the items per page *globally*, we could do it with this config. *Or*, if we want to change it *only* for one resource, we can do it above the class.

Customizing Max Items Per Page

Find the `ApiResource` attribute and add `paginationItemsPerPage` set to 10:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 18
19 #[ApiResource(
↕ // ... lines 20 - 34
35     paginationItemsPerPage: 10
36 )]
37 class DragonTreasure
38 {
↕ // ... lines 39 - 161
162 }
```

Again, you can see that the options we already have... are included in the `defaults` config.

Move over and head back to page 1. And... voilà! A much shorter list. Also, there are now 4 pages of treasure instead of 2.

Oh, and FYI: you can also make it so that the *user* of your API can determine how many items to show per page via a query parameter. Check the documentation for how to do that.

Ok, now that we have a lot of data, let's add the ability for our Dragon API users to search and filter through the treasures. Like maybe a dragon is searching for a a treasure of individually wrapped candies among all this loot. That's next.

Chapter 13: Filters: Searching Results

Some of our dragon treasures are currently published and some are unpublished. That's thanks to `DragonTreasureFactory`, where we randomly publish some but not others.

Right now, the API is returning *every* last dragon treasure. In the future, we're going to make it so that our API automatically returns only *published* treasures. But to start, let's at least make it possible for our API clients to filter out unpublished results if they want to.

Hello ApiFilter

How? By Leveraging *filters*. API Platform comes with a *bunch* of built-in filters that allow you to filter the collections of results by text, booleans, dates and much more.

Here's how it works: above your class, add an attribute called `ApiFilter`.

There are typically two ingredients that you need to pass to this. The first is which filter *class* you want to use. And if you look at the documentation, there's a bunch of them, like one called `BooleanFilter` that we'll use now and another called `SearchFilter` that we'll use in a few minutes.

Pass this `BooleanFilter` - the one from `ORM`, since we're using the Doctrine ORM - because we want to allow the user to filter on a boolean field.

The second thing you need top pass is `properties` set to an array of which fields or properties you want to use this filter on. Set this to `isPublished`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Doctrine\Orm\Filter\BooleanFilter;
6 use ApiPlatform\Metadata\ApiFilter;
↕ // ... lines 7 - 38
39 #[ApiFilter(BooleanFilter::class, properties: ['isPublished'])]
40 class DragonTreasure
41 {
↕ // ... lines 42 - 164
165 }
```

Using the Filter in the Request

All right! Go back to the documentation and check out the GET collection endpoint. When we try this... there's a new `isPublished` field! First, just hit "Execute" without setting that. When we scroll all the way down, there we go! `hydra:totalItems: 40`. Now set `isPublished` to `true` and try it again.

Yes! We have `hydra:totalItems: 16`. It's alive! And check out *how* the filtering happens. It's dead simple via a query parameter: `isPublished=true`. Oh, and it gets cooler. Look at the response: we have `hydra:view`, which shows the pagination and now we *also* have a new `hydra:search`. Yea, API Platform actually *documents* this new way of searching right in the response. It's saying:

> *"Hey, if you want, you can add a `?isPublished=true` query parameter to filter these results."*

Pretty stinking cool.

Adding Filters Directly Above Properties

Now, when you read about filters inside of the API Platform docs, they pretty much always show it *above* the class, like we have. But you can *also* put the filter above the *property* it relates to.

Watch: copy the `ApiFilter` line, remove it, and go down to `$isPublished`. Paste this above. And now, we don't need the `properties` option anymore... API Platform figures that out on its own:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 38
39 class DragonTreasure
40 {
↕ // ... lines 41 - 68
69     #[ApiFilter(BooleanFilter::class)]
70     private bool $isPublished = false;
↕ // ... lines 71 - 164
165 }
```

The result? The same as before. I won't try it, but if you peek at the collection endpoint, it still has the `isPublished` filter field.

SearchFilter: Filter by Text

What else can we do? Another really handy filter is `SearchFilter`. Let's make it possible to search by text on the `name` property. This looks *almost* the same: above `$name`, add `ApiFilter`. In this case we want `SearchFilter`: again, get the one for the ORM. This filter *also* accepts an option. You can see here that, in addition to `properties`, `ApiFilter` has an argument called `strategy`. That doesn't apply to all filters, but it *does* apply to this one. Set `strategy` to `partial`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 5
6 use ApiPlatform\Doctrine\Orm\Filter\SearchFilter;
↕ // ... lines 7 - 39
40 class DragonTreasure
41 {
↕ // ... lines 42 - 48
49     #[ApiFilter(SearchFilter::class, strategy: 'partial')]
50     private ?string $name = null;
↕ // ... lines 51 - 166
167 }
```

This will allow us to search on the `name` property for a *partial* match. It's a "fuzzy" search. Other strategies include `exact`, `start` and more.

Let's give it a shot! Refresh the docs page. And... now the collection endpoint has *another* filter box. Search for `rare` and hit Execute. Let's see, down here... yes! Apparently 15 of the results have `rare` somewhere in the `name`.

And again, this works by adding a simple `?name=rare` to the URL.

Oh, let's also make the `description` field searchable:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 5
6 use ApiPlatform\Doctrine\Orm\Filter\SearchFilter;
↕ // ... lines 7 - 39
40 class DragonTreasure
41 {
↕ // ... lines 42 - 48
49     #[ApiFilter(SearchFilter::class, strategy: 'partial')]
50     private ?string $name = null;
↕ // ... lines 51 - 53
54     #[ApiFilter(SearchFilter::class, strategy: 'partial')]
55     private ?string $description = null;
↕ // ... lines 56 - 167
168 }
```

And now... that shows up in the API too!

The `SearchFilter` is easy to set up... but it's a fairly simple fuzzy search. If you want something more complex - like ElasticSearch - API Platform *does* support that. You can even create your *own* custom filters, which we'll do in a future tutorial.

Alrighty: next, let's see two more filters: one simple and one weird... A filter that, instead of hiding *results*, allows the API user to hide certain *fields* in the response.

Chapter 14: PropertyFilter: Sparse Fieldsets

Since dragons *love* expensive treasure, let's add a way for them to filter based on the *value*, like within a range. There's a built-in filter for that called `RangeFilter`. Find the `$value` property and, like we did before, use `#[ApiFilter()]` and inside `RangeFilter` (the one from ORM) `::class`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 6
7 use ApiPlatform\Doctrine\Orm\Filter\RangeFilter;
↕ // ... lines 8 - 40
41 class DragonTreasure
42 {
↕ // ... lines 43 - 62
63     #[ApiFilter(RangeFilter::class)]
64     private ?int $value = null;
↕ // ... lines 65 - 169
170 }
```

This one doesn't need any other options, so... we're done! Dang... that was easy. When we refresh... open it up, and hit "Try it out".... look at that! We have *a ton* of new filters - `value[between]`, `value[gt]` (or "greater than"), `value[gte]`, etc. Let's try `value[gt]`... with a random number... maybe `500000`. When we click "Execute"... yup! It updated the URL here. It's... not the prettiest URL ever - due to the encoding - but it *works* like a charm. And down in the results... apparently there are 18 treasures worth more than that!

PropertyFilter

The *last* filter I want to show you... isn't really a filter at all. It's a way our API clients to choose which *fields* they want returned... instead of which *results*.

To show this off, find the `getDescription()` method. Pretend that we want to return a shorter, truncated version of the description. To do this, copy the `getDescription()` method, paste it below, and rename it to `getShortDescription()`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 41
42 class DragonTreasure
43 {
↕ // ... lines 44 - 99
100     public function getShortDescription(): string
101     {
↕ // ... line 102
103     }
↕ // ... lines 104 - 176
177 }
```

To *truncate* this, we can use the `u()` function from Symfony. Type `u` and make sure to hit "tab" to autocomplete that. This is a rare *function* that Symfony gives us... and hitting "tab" *did* add a `use` statement for it:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 20
21 use function Symfony\Component\String\u;
↕ // ... lines 22 - 41
42 class DragonTreasure
43 {
↕ // ... lines 44 - 99
100     public function getShortDescription(): string
101     {
102         return u($this->getDescription())->truncate(40, '...');
103     }
↕ // ... lines 104 - 176
177 }
```

This creates an object with all sorts of string-related goodies on it, including `truncate()`. Pass 40 to truncate at `40` characters followed by `...`.

Method done! To expose this to our API, above, add the `Groups` attribute with `treasure:read`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 20
21 use function Symfony\Component\String\u;
↕ // ... lines 22 - 41
42 class DragonTreasure
43 {
↕ // ... lines 44 - 98
99     #[Groups(['treasure:read'])]
100    public function getShortDescription(): string
101    {
102        return u($this->getDescription())->truncate(40, '...');
103    }
↕ // ... lines 104 - 176
177 }
```

Beautiful! Okay, head back to the documentation and refresh. Open the `GET` endpoint, hit "Try it out", "Execute" and... *nice*. Here's our truncated description!

Though... it *is* weird that we now return *two* descriptions: a short one and the regular one. If our API client wants the short description, it may not want us to *also* return the full-length description... for the sake of bandwidth.

What can we do? Introducing: the `PropertyFilter`! Head back to `DragonTreasure`. Unlike the others, this filter *must* go above the class. So right here, say `ApiFilter`, and then `PropertyFilter` (in this case, there's only *one* of them) `::class`. There *are* some options you can pass to this - which you can find in the docs - but we don't need any of them:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 14
15 use ApiPlatform\Serializer\Filter\PropertyFilter;
↕ // ... lines 16 - 42
43 #[ApiFilter(PropertyFilter::class)]
44 class DragonTreasure
45 {
↕ // ... lines 46 - 178
179 }
```

So... what does that *do*? Head back, refresh the documentation, open up the GET collection endpoint, and hit "Try it out". Woh! We now see a `properties[]` box and we can add items to it. Let's try it! Add a new string called `name` and another called `description`.

Moment of truth. Hit "Execute", and... there it is! It popped these onto the URL like normal. But look at the response: it *only* contains the `name` and `description` fields. Well... it *also*

contains the JSON-LD fields, but the *real* data is *just* those two fields.

If we removed the `properties` strings, we would get the normal, full response. So, by default, you get *all* fields. But users can now *choose* fewer fields if they want to.

What about Vulcain?

This all *works* quite nicely. But if you look at the API Platform documentation for the `PropertyFilter`, they actually recommend a different solution: something called "Vulcain". Nope, not Spock's home planet. We're talking about a protocol that adds features to your web server. It was created by the API Platform team, and if we scroll down a bit, they have a really good example.

Pretend that we have the following API. If we make a request to `/books`, we get these two books back. Simple enough. Then maybe we want to get more info about the *first* book, so we make a request to *that* URL - `/books/1`. Great! But... now we want details about the author, so we make a request to `/authors/1`.

So, to get *all* the book information and all the author information, we ultimately needed to make *four* requests: the original, plus 3 more. That's not great for performance.

What Vulcain allows you to do is just make this *first* request... but tell the server that it should *push* the data from the other requests *to* you.

We can see this best in JavaScript, and there's a little example down here. In this case, imagine that we're making a request to `/books/1` but we know that we also need the author information. So, when we make the request, we include a special `Preload` header. This tells the server:

> *"Hey! After returning the book data, use a server push to send me the information found by following the `author` IRI."*

The *really* cool thing is that your JavaScript doesn't really change. You *still* use `fetch()` to make a second request to the `bookJSON.author` URL... except that this will return *instantly* because the browser already has the data.

I'm not going to get into all the specifics, but the `Preload` on the first example is even more impressive: `/member/*/author`. That tells the server to push all the data as if we had *also*

requested each of the `member` keys - so all the books - *and* their author URLs.

The point is: if you use Vulcain, your API users can make *tiny* changes to enjoy huge performance benefits... without us needing to add a lot of fanciness to our API.

Next: Let's talk about *formats*. We know that our API can return JSON-LD, JSON, and even HTML representations of our resources. Let's add two *new* formats, including a CSV format, which will be the *fastest* CSV export feature you've ever built.

Chapter 15: More Formats: HAL & CSV

API Platform supports multiple input and output formats. We know that we can go to `/api/treasures.json` to see JSON, `.jsonld` to see JSON-LD or even `.html` to see the HTML output format.

Accept Header & Content Negotiation

But adding this extension to the end of the URL is just a hack that API Platform allows. To choose the format we want the API to return, we're supposed to send an `Accept` header. And we can see this when we use the interactive docs. This makes a request with an `Accept` header set to `application/ld+json`. Setting this header is easy to do in JavaScript, and if you *don't* set it, JSON-LD is the default format.

API Platform uses *three* formats out of the box. You can see them down here on the bottom of the docs page. But, what in our app says that we want to use these three formats specifically? To answer that, head over to your terminal and run:

```
./bin/console debug:config api_platform
```

Inside the config, check out this `formats` key... which, by default, is set to those three formats. This basically says that if the `Accept` header is `application/ld+json`, use the JSON-LD format. *Internally*, it means that when Symfony is serializing our data, it will *serialize* to JSON-LD or JSON.

Adding a New Format

As a challenge, let's add a *fourth* format. To do that, we just need to add a new item to this config... but without completely *replacing* the existing formats. Copy these, then open the `/config/packages/` directory. We don't have an `api_platform.yaml` file yet, so let's

create one. Inside that, say `api_platform` and paste those below. And while we don't *have* to, I'm going to change this to use a shorter, more attractive version of this config:

config/packages/api_platform.yaml

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

Done!

If we go and refresh right now, everything works the same. We have the *same* formats below... because we simply repeated the default config.

The *new* format we're going to add is another type of JSON called HAL. Here's what's going on. We all understand the JSON format. But then, to add more *meaning* to JSON - like certain keys that your JSON should have and their meaning - some people come out with standards that *extend* JSON. JSON-LD is one example and HAL is a *competing* standard. I don't often use HAL... so we're mostly doing this to see an example of what adding a format looks like.

Oh, and the `Content-Type` for HAL is supposed to be `application/hal+json`:

config/packages/api_platform.yaml

```yaml
api_platform:
    formats:
# ... lines 3 - 5
        jsonhal: [ 'application/hal+json' ]
```

As soon as we do that, when we refresh... it shows *nothing*? I'm pretty sure Symfony didn't see my new config file. Hop over here and clear the cache with:

```
./bin/console cache:clear
```

Refresh again and... there we go! We now see `jsonhal`! And if we click, it takes us to the `jsonhal` version of our API homepage!

Let's try an endpoint with this format. Click on the `GET` request, "Try it out", and, down here, we can select which "media type" to request. Select `application/hal+json`, hit "Execute", and... there it is!

You can see that it's JSON... and it has the same results, but it looks a bit different. It has things like `_embedded` and `_links`... which are part of the HAL standard... and not worth talking about right now.

By the way, the *reason* this new format worked *simply* by adding a tiny bit of config is that the serializer already *understands* the `jsonhal` format. So when we request with this `Accept` header, API Platform asks the serializer to serialize *into* the `jsonhal` format... and *it* knows how to do that.

Adding a CSV Format

Okay, let's do something that's a bit more practical. What if our dragon users need to return the treasures in CSV format... like so they can import them into Quickbooks for tax purposes.

Well, CSV *is* a format that Symfony's Serializer understands out of the box. We know that we *could* add CSV right into this config file. But as an added challenge, instead of enabling the CSV for *every* API resource in our system, let's just add it to `DragonTreasure`.

Find the `ApiResource` attribute and, at the bottom, add `formats`. Just like with the configuration, if we simple put `csv` here, that will remove the other formats. To do this right, we need to list all of them: `jsonld`, `json`, `html`, and `jsonhal`. Each of these will read the configuration to know which content type to use. At the end, add `csv`. But because `csv` doesn't exist in the config, we need to tell it which content type will activate this. So set it to `text/csv`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 24
25 #[ApiResource(
↕ // ... lines 26 - 41
42     formats: [
43         'jsonld',
44         'json',
45         'html',
46         'jsonhal',
47         'csv' => 'text/csv',
48     ],
49 )]
↕ // ... line 50
51 class DragonTreasure
52 {
↕ // ... lines 53 - 185
186 }
```

Oh, but my editor is mad! It says:

> *"Named arguments order does not match parameters order"*

We know that each PHP attribute is a class... and when we pass arguments to the attribute, we're actually passing *named* arguments to that class's constructor. And, with named args, the *order* of the args doesn't matter. I actually don't think PhpStorm should be highlighting this as a problem... but if you're annoyed like I am, you can hit "Sort arguments" and... *there*. It moved `formats` up a little higher, it's happy, and *we* won't have to stare at that yellow underline.

All right, head over, refresh, open up our collection endpoint and hit "Try it out". This time, down here, select `text/csv` then... "Engage"! *Hello* CSV. *Too easy*!

Once again, this works because Symfony's Serializer *understands* the CSV format. So *it* does all the work.

In fact, open the profiler for that request... and go down to the serializer section. Yep! We can see that it's using the `csv` format... which activates a `CsvEncoder`. *That's* why we get our nice results. If you needed to return your results in a *custom* format that's *not* supported by the serializer, you could add your *own* encoder to the system to handle that. It's *super* flexible

Next: Let's talk about *validation*!

Chapter 16: Validation

There are a *bunch* of different ways for the users of our API to mess things up, like bad JSON or doing silly things like passing a negative number for the `value` field. This is dragon gold, not dragon debt!

Invalid JSON

This chapter is all about handling these bad things in a graceful way. Try the POST endpoint. Let's send some invalid JSON. Hit Execute. Awesome! A `400` error! That's what we want. 400 - or any status code that starts with 4 - means that the *client* - the user of the API - made a mistake. 400 specifically means "bad request".

In the response, the type is `hydra:error` and it says: `An error occurred` and `Syntax Error`. Oh, and this `trace` only shows in the debug environment: it won't be shown on production.

So this is pretty sweet! Invalid JSON is handled out-of-the-box.

Business Rules Validation Constraints

Let's try something different, like sending *empty* JSON. *This* gives us the dreaded 500 error. Boo. Internally, API platform creates a `DragonTreasure` object... but doesn't set any data on it. And then it explodes when it hits the database because some of the columns are `null`.

And, we expected this! We're missing validation. Adding validation to our API is exactly like adding validation *anywhere* in Symfony. For example, find the `name` property. We need `name` to be required. So, add the `NotBlank` constraint, and hit tab. Oh, but I'm going to go find the `NotBlank` `use` statement... and change this to `Assert`. That's optional... but it's the way the cool kids tend do it in Symfony. Now say `Assert\NotBlank`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 19
20 use Symfony\Component\Validator\Constraints as Assert;
↕ // ... lines 21 - 51
52 class DragonTreasure
53 {
↕ // ... lines 54 - 61
62     #[Assert\NotBlank]
↕ // ... line 63
64     private ?string $name = null;
↕ // ... lines 65 - 188
189 }
```

Below, add one more: `Length`. Let's say that the name should be at least two characters, `max` 50 characters... and add a `maxMessage`: `Describe your loot in 50 chars or less`:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 19
20 use Symfony\Component\Validator\Constraints as Assert;
↕ // ... lines 21 - 51
52 class DragonTreasure
53 {
↕ // ... lines 54 - 61
62     #[Assert\NotBlank]
63     #[Assert\Length(min: 2, max: 50, maxMessage: 'Describe your loot in 50
   chars or less')]
64     private ?string $name = null;
↕ // ... lines 65 - 188
189 }
```

How Errors Look in the Response

Good start! Let's try it again. Take that same empty JSON, hit Execute, and yes! A 422 response! This is a really common response code that usually means there was a validation error. And behold! The `@type` is `ConstraintViolationList`. This is a special JSON-LD type added by API Platform. Earlier, we saw this documented in the `JSON-LD` documentation.

Watch: go to `/api/docs.jsonld` and search for a `ConstraintViolation`. There it is! API Platform adds two classes - `ConstraintViolation` and `ConstraintViolationList` to describe how validation errors will look. A `ConstraintViolationList` is basically just a collection of `ConstraintViolations`... and then it describes what the `ConstraintViolation` properties are.

We can see these over here: we have a `violations` property with `propertyPath` and then the `message` below.

Adding More Constraints

Ok! Let's sneak in a few more constraints. Add `NotBlank` above `description`... and `GreaterThanOrEqual` to `0` above `value` to avoid negatives. Finally, for `coolFactor` use `GreaterThanOrEqual` to 0 and also `LessThanOrEqual` to 10. So something between 0 and 10:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 51
52 class DragonTreasure
53 {
↕ // ... lines 54 - 68
69     #[Assert\NotBlank]
70     private ?string $description = null;
↕ // ... lines 71 - 77
78     #[Assert\GreaterThanOrEqual(0)]
79     private ?int $value = null;
↕ // ... lines 80 - 82
83     #[Assert\GreaterThanOrEqual(0)]
84     #[Assert\LessThanOrEqual(10)]
85     private ?int $coolFactor = null;
↕ // ... lines 86 - 192
193 }
```

And while we're here, we don't need to do this, but I'm going to initialize `$value` to 0 and `$coolFactor` to 0. This makes both of those *not* required in the API: if the user doesn't send them, they'll default to 0:

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 51
52 class DragonTreasure
53 {
↕ // ... lines 54 - 68
69     #[Assert\NotBlank]
70     private ?string $description = null;
↕ // ... lines 71 - 77
78     #[Assert\GreaterThanOrEqual(0)]
79     private ?int $value = 0;
↕ // ... lines 80 - 82
83     #[Assert\GreaterThanOrEqual(0)]
84     #[Assert\LessThanOrEqual(10)]
85     private ?int $coolFactor = 0;
↕ // ... lines 86 - 192
193 }
```

Ok, go back and try that same endpoint. Look at that beautiful validation! Also try setting `coolFactor` to `11`. Yup! No treasure is *that* cool... well, unless it's a giant plate of nachos.

Passing Bad Types

Ok, there's one last way that a user can send bad stuff: by passing the wrong *type*. So `coolFactor: 11` will fail our validation rules. But what if we pass a `string` instead? Yikes! Hit Execute. Okay: a `400` status code, that's good. Though, it's not a validation error, it has a different type. But it *does* tell the user what happened:

> *"the type of the `coolFactor` attribute must be `int`, `string` given."*

Good enough! This is thanks to the `setCoolFactor()` method. The system sees the `int` type and so it rejects the string with this error.

So the only thing that we need to worry about in our app is writing good code that properly uses `type` and adding validation constraints: the safety net that catches business rule violations... like `value` should be greater than 0 or `description` is required. API Platform handles the rest.

Next: our API only has one resource: `DragonTreasure`. Let's add a second resource - a `User` resource - so that we can *link* which user owns which treasure in the API.

Chapter 17: Creating a User Entity

We won't talk about security in this tutorial. But even still, we *do* need the concept of a user... because each treasure in the database will be *owned* by a user... or really, by a dragon. Later, we'll use this to allow API users to see which treasures belong to which user and a bunch more.

make:user

So, let's create that `User` class. Find your terminal and run:

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

We could use `make:entity`, but `make:user` will set up a bit of the security stuff that we'll need in a *future* tutorial. Let's call the class `User`, yes we *are* going to store these in the database, and set `email` as the main identifier field.

Next it asks if we need to hash and check user passwords. If the hashed version of user passwords will be stored in your system, say yes to this. If your users won't have passwords - or some external system checks the passwords - answer no. I'll say yes to this.

This didn't do much... in a good way! It gave us a `User` entity, the repository class... and a small update to `config/packages/security.yaml`. Yup, it just sets up the user provider: nothing special. And again, we'll talk about that in a future tutorial.

Adding a username Property

Ok, inside the `src/Entity/` directory, we have our new `User` entity class with `id`, `email` and `password` properties... and getters and setters below. Nothing fancy. This implements two interfaces that we need for security... but those aren't important right now.

src/Entity/User.php

```
↕ // ... lines 1 - 2
3  namespace App\Entity;
4
5  use App\Repository\UserRepository;
6  use Doctrine\ORM\Mapping as ORM;
7  use
   Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
8  use Symfony\Component\Security\Core\User\UserInterface;
9
10 #[ORM\Entity(repositoryClass: UserRepository::class)]
11 #[ORM\Table(name: '`user`')]
12 class User implements UserInterface, PasswordAuthenticatedUserInterface
13 {
14     #[ORM\Id]
15     #[ORM\GeneratedValue]
16     #[ORM\Column]
17     private ?int $id = null;
18
19     #[ORM\Column(length: 180, unique: true)]
20     private ?string $email = null;
21
22     #[ORM\Column]
23     private array $roles = [];
24
25     /**
26      * @var string The hashed password
27      */
28     #[ORM\Column]
29     private ?string $password = null;
↕ // ... lines 30 - 99
100 }
```

Oh, but I *do* want to add one more field to this class: a `username` that we can show in the API.

So, spin back over to your terminal and this time run:

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

Update the `User` class, add a `username` property, `255` length is good, not null... and done. Hit enter one more time to exit.

Back over on the class... perfect! There's the new field. While we're here, add `unique: true` to make this unique in the database.

src/Entity/User.php

```
↕ // ... lines 1 - 11
12 class User implements UserInterface, PasswordAuthenticatedUserInterface
13 {
↕ // ... lines 14 - 30
31     #[ORM\Column(length: 255, unique: true)]
32     private ?string $username = null;
↕ // ... lines 33 - 114
115 }
```

Entity done! Let's make a migration for it. Back at the terminal run:

```
symfony console make:migration
```

Then... spin over and open that new migration file. No surprises: it creates the `user` table:

migrations/Version20230104193724.php

```
// ... lines 1 - 2
declare(strict_types=1);

namespace DoctrineMigrations;

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20230104193724 extends AbstractMigration
{
// ... lines 15 - 19
    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE SEQUENCE "user_id_seq" INCREMENT BY 1 MINVALUE 1 START 1');
        $this->addSql('CREATE TABLE "user" (id INT NOT NULL, email VARCHAR(180) NOT NULL, roles JSON NOT NULL, password VARCHAR(255) NOT NULL, username VARCHAR(255) NOT NULL, PRIMARY KEY(id))');
        $this->addSql('CREATE UNIQUE INDEX UNIQ_8D93D649E7927C74 ON "user" (email)');
        $this->addSql('CREATE UNIQUE INDEX UNIQ_8D93D649F85E0677 ON "user" (username)');
    }
// ... lines 28 - 35
}
```

Close that up and run it with:

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

Adding the Factory & Fixtures

Sweet! Though, I think our new entity deserves some juicy data fixtures. Let's use Foundry like we did for `DragonTreasure`. Start by running

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

to generate the factory for `User`.

Like before, in the `src/Factory/` directory, we have a new class - `UserFactory` - which is really good at creating `User` objects. The main thing we need to tweak is `getDefaults()` to make the data even better. I'm going to paste in new contents for the entire class, which you can copy from the code block on this page.

src/Factory/UserFactory.php

```
// ... lines 1 - 2
namespace App\Factory;

use App\Entity\User;
use App\Repository\UserRepository;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
use Zenstruck\Foundry\ModelFactory;
use Zenstruck\Foundry\Proxy;
use Zenstruck\Foundry\RepositoryProxy;

/**
 * @extends ModelFactory<User>
// ... lines 14 - 29
 */
final class UserFactory extends ModelFactory
{
    const USERNAMES = [
        'FlamingInferno',
        'ScaleSorcerer',
        'TheDragonWithBadBreath',
        'BurnedOut',
        'ForgotMyOwnName',
        'ClumsyClaws',
        'HoarderOfUselessTrinkets',
    ];
// ... lines 42 - 47
    public function __construct(
        private UserPasswordHasherInterface $passwordHasher
    )
    {
        parent::__construct();
    }
// ... lines 54 - 59
    protected function getDefaults(): array
    {
        return [
            'email' => self::faker()->email(),
            'password' => 'password',
            'username' => self::faker()->randomElement(self::USERNAMES) . self::faker()->randomNumber(3),
        ];
    }
// ... lines 68 - 71
    protected function initialize(): self
    {
        return $this
```

```
75              ->afterInstantiate(function(User $user): void {
76                  $user->setPassword($this->passwordHasher->hashPassword(
77                      $user,
78                      $user->getPassword()
79                  ));
80              })
81          ;
82      }
83
84      protected static function getClass(): string
85      {
86          return User::class;
87      }
88  }
```

This updates `getDefaults()` to have a little more pizazz and sets the `password` to `password`. I know, creative. I'm also leveraging an `afterInstantiation` hook to hash that password.

Finally, to actually create some fixtures, open up `AppFixtures`. Pretty simple here: `UserFactory::createMany()` and let's create 10.

src/DataFixtures/AppFixtures.php

```
↕ // ... lines 1 - 5
6  use App\Factory\UserFactory;
↕ // ... lines 7 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager): void
13     {
14         DragonTreasureFactory::createMany(40);
15         UserFactory::createMany(10);
16     }
17 }
```

Let's see if that worked! Spin over and run:

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

No errors!

Status check: we have a `User` entity and we created a migration for it. Heck, we even loaded some schweet data fixtures! But it is not, yet, part of our API. If you refresh the documentation, there's still only `Treasure`.

Let's make this part of our API next.

Chapter 18: User API Resource

We have a `User` entity... but it is not *yet* part of our API. How *do* we make it part of the API? Ah, we already know! Go above the class and add the `ApiResource` attribute.

src/Entity/User.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Metadata\ApiResource;
↕ // ... lines 6 - 12
13 #[ApiResource]
14 class User implements UserInterface, PasswordAuthenticatedUserInterface
15 {
↕ // ... lines 16 - 116
117 }
```

Refresh the docs. Look at that! Six fresh new endpoints for the `User` class! And thanks to our fixtures, we *should* be able to see data immediately. Let's try the collection endpoint. Execute and... it's alive.

Though... it *is* a little weird that fields like `roles` and `password` show up. Ah, we'll worry about that in a minute.

API Platform & UUIDs

Before we keep rolling forward, I want to mention one quick thing about UUIDs. As you can see, we're using auto-increment IDs for of our API - it's always `/api/users/` then the entity id. But you can *totally* use a `UUID` instead. And that's something we'll do in a future tutorial.

But... why *would* you use UUIDs? Well, sometimes it can make life easier in JavaScript when working with frontend frameworks. You can actually *generate* the `UUID` in JavaScript and then send that to your API when creating a new resource. This can help because your JavaScript knows the ID of the resource immediately and can update the state... instead of waiting for the Ajax request to finish to get the new auto-increment id.

Anyways, my point is: API Platform *does* support `UUIDs`. You could add a new UUID column, then tell API Platform that it should be your *identifier*. Oh, but keep in mind that some database

engines - like MySQL - can have poor performance if you make the UUID the primary key. In that case, just keep `id` as the primary key, and add an *extra* UUID column.

Adding the Serialization Groups

Anyways, back to our `User` resource! Right now, it's returning *way* too many fields. Fortunately, we know how to fix that. Up on `ApiResource`, add a `normalizationContext` key with `groups` set to `user:read` to follow the same pattern that we used in `DragonTreasure`. Also add `denormalizationContext` set to `user:write`.

src/Entity/User.php

```
↕ // ... lines 1 - 13
14 #[ApiResource(
15     normalizationContext: ['groups' => ['user:read']],
16     denormalizationContext: ['groups' => ['user:write']],
17 )]
18 class User implements UserInterface, PasswordAuthenticatedUserInterface
19 {
↕ // ... lines 20 - 123
124 }
```

Now we can just decorate the fields that we want in the API. We don't need `id`... since we always have `@id`, which is more useful. But we *do* want `email`. So add the `#Groups()` attribute, hit tab to add that `use` statement and pass both `user:read` and `user:write`.

src/Entity/User.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\Serializer\Annotation\Groups;
↕ // ... lines 11 - 13
14 #[ApiResource(
15     normalizationContext: ['groups' => ['user:read']],
16     denormalizationContext: ['groups' => ['user:write']],
17 )]
18 class User implements UserInterface, PasswordAuthenticatedUserInterface
19 {
↕ // ... lines 20 - 25
26     #[Groups(['user:read', 'user:write'])]
27     private ?string $email = null;
↕ // ... lines 28 - 123
124 }
```

Copy that... and go down to `password`. We *do* need the password to be writeable but not readable. So add `user:write`.

src/Entity/User.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\Serializer\Annotation\Groups;
↕ // ... lines 11 - 13
14 #[ApiResource(
15     normalizationContext: ['groups' => ['user:read']],
16     denormalizationContext: ['groups' => ['user:write']],
17 )]
18 class User implements UserInterface, PasswordAuthenticatedUserInterface
19 {
↕ // ... lines 20 - 25
26     #[Groups(['user:read', 'user:write'])]
27     private ?string $email = null;
↕ // ... lines 28 - 35
36     #[Groups(['user:write'])]
37     private ?string $password = null;
↕ // ... lines 38 - 123
124 }
```

Now this still isn't quite correct. The `password` field should hold the *hashed* password. But our users will, of course, send the plaintext passwords via the API when creating a user or updating their password. Then *we* will hash it. That's something we're going to solve in a future tutorial when we talk more about security. But this will be good enough for now.

Oh, and above `username`, also add `user:read` and `user:write`.

src/Entity/User.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\Serializer\Annotation\Groups;
↕ // ... lines 11 - 13
14 #[ApiResource(
15     normalizationContext: ['groups' => ['user:read']],
16     denormalizationContext: ['groups' => ['user:write']],
17 )]
18 class User implements UserInterface, PasswordAuthenticatedUserInterface
19 {
↕ // ... lines 20 - 25
26     #[Groups(['user:read', 'user:write'])]
27     private ?string $email = null;
↕ // ... lines 28 - 35
36     #[Groups(['user:write'])]
37     private ?string $password = null;
↕ // ... lines 38 - 39
40     #[Groups(['user:read', 'user:write'])]
41     private ?string $username = null;
↕ // ... lines 42 - 123
124 }
```

Cool! Refresh the docs... and open up the collections endpoint to give it a go. The result... exactly what we wanted! Only `email` and `username` come back.

And if we were to *create* a new user... yup! The writable fields are `email`, `username`, and `password`.

Adding Validation

Ok, what else are we missing? How about validation? If we try the POST endpoint with empty data... we get that nasty 500 error. Fixing time!

Back over in the file, start *above* the class to make sure that both `email` and `username` are `unique`. Add `UniqueEntity` passing `fields` set to `email`... and we can even include a message. Repeat that *same* thing... but change `email` to `username`.

src/Entity/User.php

```
↕ // ... lines 1 - 10
11 use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
↕ // ... lines 12 - 18
19 #[UniqueEntity(fields: ['email'], message: 'There is already an account with this email')]
20 #[UniqueEntity(fields: ['username'], message: 'It looks like another dragon took your username. ROAR!')]
21 class User implements UserInterface, PasswordAuthenticatedUserInterface
22 {
↕ // ... lines 23 - 129
130 }
```

Next, down in `email`, add `NotBlank`... then I'll add the `Assert` in front... and tweak the `use` statement so it works just like last time.

src/Entity/User.php

```
↕ // ... lines 1 - 10
11 use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
12 use Symfony\Component\Validator\Constraints as Assert;
↕ // ... lines 13 - 18
19 #[UniqueEntity(fields: ['email'], message: 'There is already an account with this email')]
20 #[UniqueEntity(fields: ['username'], message: 'It looks like another dragon took your username. ROAR!')]
21 class User implements UserInterface, PasswordAuthenticatedUserInterface
22 {
↕ // ... lines 23 - 29
30     #[Assert\NotBlank]
↕ // ... line 31
32     private ?string $email = null;
↕ // ... lines 33 - 129
130 }
```

Nice. email needs one more - `Assert\Email` - and above `username`, add `NotBlank`.

src/Entity/User.php

```
↕ // ... lines 1 - 10
11 use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
12 use Symfony\Component\Validator\Constraints as Assert;
↕ // ... lines 13 - 18
19 #[UniqueEntity(fields: ['email'], message: 'There is already an account
   with this email')]
20 #[UniqueEntity(fields: ['username'], message: 'It looks like another
   dragon took your username. ROAR!')]
21 class User implements UserInterface, PasswordAuthenticatedUserInterface
22 {
↕ // ... lines 23 - 29
30     #[Assert\NotBlank]
31     #[Assert\Email]
32     private ?string $email = null;
↕ // ... lines 33 - 45
46     #[Assert\NotBlank]
47     private ?string $username = null;
↕ // ... lines 48 - 129
130 }
```

I'm not too worried about `password` right now... because it's already a bit weird.

Let's try this! Scroll up and *just* send a `password` field. And... yes! The nice 422 status code with validation errors. Try valid data now: pass an `email` and `username`... though I'm not sure this guy's actually a dragon... we might need a captcha.

Hit Execute. That's it! 201 status code with `email` and `username` returned!

Our resource has validation, pagination and contains great *information*! And we could even easily add filtering. In other words, we're crushing it!

And *now* we get to the *really* interesting part. We need to "relate" our two resources so that each treasure is *owned* by a user. What does that look like in API Platform? It's super interesting, and it's next.

Chapter 19: Relating Resources

In our app, each `DragonTreasure` should be owned by a single dragon... or `User` in our system. To set this up, forget about the API for a moment and let's just model this in the database.

Adding the ManyToOne Relation

Spin over to your terminal and run:

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

Let's modify the `DragonTreasure` entity to add an `owner` property... and then this will be a `ManyToOne` relation. If you're not sure which relation you need, you can always type `relation` and get a nice little wizard.

This will be a relation to `User`... and then it asks if the new `owner` property is allowed to be null in the database. Every `DragonTreasure` *must* have an owner... so say "no". Next: do we we want to map the other side of the relationship? So basically, do we want the ability to say, `$user->getDragonTreasures()` in our code? I'm going to say yes to this. And you might answer "yes" for two reasons. Either because being able to say `$user->getDragonTreasures()` would be useful in your code *or*, as we'll see a bit later, because you want to be able to fetch a `User` in your API and instantly see what treasures it has.

Anyways, the property - `dragonTreasures` inside of `User` is fine.... and finally, for `orphanRemoval`, say no. We'll also talk about that later.

And... done! Hit enter to exit.

So this had nothing to do with API Platform. Our `DragonTreasure` entity now has a new `owner` property with `getOwner()` and `setOwner()` methods.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 51
52 class DragonTreasure
53 {
↕ // ... lines 54 - 93
94     #[ORM\ManyToOne(inversedBy: 'dragonTreasures')]
95     #[ORM\JoinColumn(nullable: false)]
96     private ?User $owner = null;
↕ // ... lines 97 - 197
198     public function getOwner(): ?User
199     {
200         return $this->owner;
201     }
202
203     public function setOwner(?User $owner): self
204     {
205         $this->owner = $owner;
206
207         return $this;
208     }
209 }
```

And over in `User` we have a new `dragonTreasures` property, which is a `OneToMany` back to `DragonTreasure`. At the bottom, it generated `getDragonTreasures()`, `addDragonTreasure()`, and `removeDragonTreasure()`. Very standard stuff.

src/Entity/User.php

```php
// ... lines 1 - 6
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
// ... lines 9 - 22
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 25 - 50
    #[ORM\OneToMany(mappedBy: 'owner', targetEntity: DragonTreasure::class)]
    private Collection $dragonTreasures;

    public function __construct()
    {
        $this->dragonTreasures = new ArrayCollection();
    }
// ... lines 58 - 140
    /**
     * @return Collection<int, DragonTreasure>
     */
    public function getDragonTreasures(): Collection
    {
        return $this->dragonTreasures;
    }

    public function addDragonTreasure(DragonTreasure $treasure): self
    {
        if (!$this->dragonTreasures->contains($treasure)) {
            $this->dragonTreasures->add($treasure);
            $treasure->setOwner($this);
        }

        return $this;
    }

    public function removeDragonTreasure(DragonTreasure $treasure): self
    {
        if ($this->dragonTreasures->removeElement($treasure)) {
            // set the owning side to null (unless already changed)
            if ($treasure->getOwner() === $this) {
                $treasure->setOwner(null);
            }
        }

        return $this;
    }
}
```

Let's create a migration for this:

```
symfony console make:migration
```

We'll do our standard double-check to make sure the migration isn't trying to mine bitcoin. Yep, all boring SQL queries here.

migrations/Version20230104200643.php

```
// ... lines 1 - 12
final class Version20230104200643 extends AbstractMigration
{
// ... lines 15 - 19
    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('ALTER TABLE dragon_treasure ADD owner_id INT NOT NULL');
        $this->addSql('ALTER TABLE dragon_treasure ADD CONSTRAINT FK_9E31BF5F7E3C61F9 FOREIGN KEY (owner_id) REFERENCES "user" (id) NOT DEFERRABLE INITIALLY IMMEDIATE');
        $this->addSql('CREATE INDEX IDX_9E31BF5F7E3C61F9 ON dragon_treasure (owner_id)');
    }
// ... lines 27 - 35
}
```

Run it with:

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

Resetting the Database

And it explodes in our face. Rude! But... it shouldn't be too surprising. We already have about 40 `DragonTreasure` records in our database. So when the migration tries to add the `owner_id` column to the table - which does *not* allow null - our database is stumped: it has no idea what value to put for those existing treasures.

If our app were already on production, we'd have to do a bit more work to fix this. We talk about that in our Doctrine tutorial. But since this *isn't* on production, we can cheat and just to turn the database off and on again. To do that run:

```
symfony console doctrine:database:drop --force
```

Then:

```
symfony console doctrine:database:create
```

And the migration, which *should* work now that our database is empty.

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

Setting up the Fixtures

Finally, re-add some data with:

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

And oh, this fails for the same reason! It's trying to create Dragon Treasures without an owner. To fix that, there are two options. In `DragonTreasureFactory`, add a new `owner` field to `getDefaults()` set to `UserFactory::new()`.

src/Factory/DragonTreasureFactory.php

```
↕ // ... lines 1 - 29
30 final class DragonTreasureFactory extends ModelFactory
31 {
↕ // ... lines 32 - 46
47     protected function getDefaults(): array
48     {
49         return [
↕ // ... lines 50 - 55
56             'owner' => UserFactory::new(),
57         ];
58     }
↕ // ... lines 59 - 73
74 }
```

I'm not going to go into the specifics of Foundry - and Foundry has great docs on how to work with relationships - but this will create a *new* `User` each time it creates a new `DragonTreasure`... and then will relate them. So that's nice to have as a default.

But in `AppFixtures`, let's *override* that to do something cooler. Move the `DragonTreasureFactory` call after `UserFactory`... then pass a second argument, which is a way to override the defaults. By passing a callback, each time a `DragonTreasure` is created - so 40 times - it will call this method and we can return unique data to use for overriding the defaults for that treasure. Return `owner` set to `UserFactory::random()`:

src/DataFixtures/AppFixtures.php

```
↕ // ... lines 1 - 9
10 class AppFixtures extends Fixture
11 {
12     public function load(ObjectManager $manager): void
13     {
14         UserFactory::createMany(10);
15         DragonTreasureFactory::createMany(40, function () {
16             return [
17                 'owner' => UserFactory::random(),
18             ];
19         });
20     }
21 }
```

That'll find a random `User` object and set it as the `owner`. So we'll have 40 `DragonTreasure`s each randomly hoarded by one of these 10 `User`s.

Let's try it! Run:

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

This time... success!

Exposing the "owner" in the API

Ok, so now `DragonTreasure` has a new `owner` relation property... and `User` has a new `dragonTreasures` relation property.

Will... that new `owner` property show up in the API? Try the GET collection endpoint for treasure. And... the new field does *not* show up! That makes sense! The `owner` property is *not* inside the normalization group.

So *if* we want to expose the `owner` property in the API, just like any other field, we need to add groups to it. Copy the groups from `coolFactor`... and paste them here.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 51
52 class DragonTreasure
53 {
↕ // ... lines 54 - 95
96     #[Groups(['treasure:read', 'treasure:write'])]
97     private ?User $owner = null;
↕ // ... lines 98 - 209
210 }
```

This makes the property readable *and* writable. And yes, later, we'll learn how to set the `owner` property automatically so that the API user doesn't need to send that manually. But for now, having the API client send the `owner` field will work great.

Anyways, what does this new `owner` property look like? Hit "Execute" and... woh! The `owner` property is set to a URL! Well, really, the *IRI* of the `User`.

I *love* this. When I first started working with API Platform, I thought relationship properties might just use the object's id. Like `owner: 1`. But this is *way* more useful... because it tells our API client exactly *how* they could get more information about this user: just follow the URL!

Writing a Relation Property

So, by default, a relation is returned as a URL. But what does it look like to *set* a relation field? Refresh the page, open the POST endpoint, try it, and I'll paste in all of the fields *except* for `owner`. What *do* we use for `owner`? I don't know! Let's try setting it to an id, like `1`.

Moment of truth. Hit execute. Let's see... a 400 status code! And check out the error:

> *"Expected IRI or nested document for attribute `owner`, integer given."*

So I passed the `ID` of the owner and... it doesn't like that. What *should* we put here? Well, the IRI of course! Let's find out more about that next.

Chapter 20: Relations & Iris

When we tried to create a `DragonTreasure` with this `owner`, we set the field to the owner's database id. And we found out that API Platform did *not* like that. It said: "expected IRI". But what *is* an `IRI`?

We mentioned this term *once* earlier in the tutorial. Go back down to the GET `/api/users` collection endpoint. We know that every resource has an `@id` field set to the URL where you can fetch that resource. This is the IRI or "International Resource Identifier". It's meant to be a unique identifier across your *entire* API - like across *all* resources.

Think about it: the number "1" is *not* a unique identifier - we might have a `DragonTreasure` with that id *and* a `User`. But the IRI *is* unique. And, a URL is also just a heck of a lot more handy than an integer anyways.

So when we want to *set* a relation property, we need to *also* use the IRI, like `/api/users/1`.

When we hit Execute, it works! A `201` status code. In the returned JSON, no surprise, the `owner` field also comes *back* as an IRI.

The takeaway from all of this is delightfully simple. Relations are just normal fields... but we get and *set* them via their IRI string. This is such a beautiful and clean way to handle this.

Adding a Collection dragonTreasures Relation Field

Ok, let's talk about the *other* side of this relationship. Refresh the whole page and go to the `GET` one user endpoint. Try this with a real user id - like 1 for me. And... there's the data.

So the question I have *now* is: could we add a `dragonTreasures` field that shows *all* the treasures that this user owns?

Well, let's think about it. We know that the serializer works by grabbing all accessible properties on an object that are in the normalization group. And... we *do* have a `dragonTreasures` property on `User`.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 50
51     #[ORM\OneToMany(mappedBy: 'owner', targetEntity:
   DragonTreasure::class)]
52     private Collection $dragonTreasures;
↕ // ... lines 53 - 169
170 }
```

So... it *should* just work! To expose the field to the API, add it to the serialization group `user:read`. Later, we'll talk about how we can *write* to a collection field... but for now, just make it readable.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 50
51     #[ORM\OneToMany(mappedBy: 'owner', targetEntity:
   DragonTreasure::class)]
52     #[Groups(['user:read'])]
53     private Collection $dragonTreasures;
↕ // ... lines 54 - 170
171 }
```

Ok! Refresh... and look at the same `GET` endpoint. Down here, cool! It shows a new `dragonTreasures` field in the example response. Let's try it: use the same id, hit "Execute" and... oh, gorgeous: it returns an array of IRI strings! I love that! And, of course, if we need more information about these, we can make a request to any of these URLs to get all the shiny details.

And to get *really* fancy, you could use Vulcain so that users can "preload" those relations... meaning the server will push the data directly to the client.

But as cool as this is, it *does* lead me to a question: what if needing the `DragonTreasure` data for a user is *so* common that, to avoid extra requests, we want to embed the data right here - like JSON objects instead of IRI strings?

Can we do that? Absolutely. Let's find out how next.

Chapter 21: Embedded Relations

So when two resources are related in our API, they show up as an IRI string, or collection of strings. But you might wonder:

> *“Hey, could we include the `DragonTreasure` data right here instead of the IRI so that I don't need to make a second, third or fourth request to get that data?”*

Absolutely! And, again, you can also do something really cool with Vulcain... but let's learn how to embed data.

Embedding Vs IRI via Normalization Groups

When the `User` object is being serialized, it uses the normalization groups to determine which fields to include. In this case, we have one group called `user:read`. That's why `email`, `username` and `dragonTreasures` are all returned.

src/Entity/User.php

```
↕ // ... lines 1 - 16
17 #[ApiResource(
18     normalizationContext: ['groups' => ['user:read']],
↕ // ... line 19
20 )]
↕ // ... lines 21 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 30
31     #[Groups(['user:read', 'user:write'])]
↕ // ... lines 32 - 33
34     private ?string $email = null;
↕ // ... lines 35 - 46
47     #[Groups(['user:read', 'user:write'])]
↕ // ... line 48
49     private ?string $username = null;
↕ // ... lines 50 - 51
52     #[Groups(['user:read'])]
53     private Collection $dragonTreasures;
↕ // ... lines 54 - 170
171 }
```

To transform the `dragonTreasures` property into *embedded* data, we need to go into `DragonTreasure` and add this *same* `user:read` group to at least *one* field. Watch: above `name`, add `user:read`. Then... go down and also add this for `value`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 51
52 class DragonTreasure
53 {
↕ // ... lines 54 - 59
60     #[Groups(['treasure:read', 'treasure:write', 'user:read'])]
↕ // ... lines 61 - 63
64     private ?string $name = null;
↕ // ... lines 65 - 75
76     #[Groups(['treasure:read', 'treasure:write', 'user:read'])]
↕ // ... lines 77 - 78
79     private ?int $value = 0;
↕ // ... lines 80 - 209
210 }
```

Yup, as soon as we have even *one* property inside of `DragonTreasure` that's in the `user:read` normalization group, the *way* the `dragonTreasures` field looks will totally change.

Watch: when we execute that... awesome! Instead of an array of IRI strings, it's an array of *objects*, with `name` and `value`... and of course the normal `@id` and `@type` fields.

So: when you have a relation field, it will either be represented as an IRI string *or* an object... and this depends entirely on your normalization groups.

Embedding the Other Direction

Let's try this same thing in the other direction. We have a `treasure` whose id is 2. Head up to the GET a single treasure endpoint... try it... and enter 2 for the id.

No surprise, we see `owner` as an IRI string. Could we turn that into an embedded object instead? Of course! We know that `DragonTreasure` uses the `treasure:read` normalization group. So, go into `User` and add that to the `username` property: `treasure:read`.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 46
47     #[Groups(['user:read', 'user:write', 'treasure:read'])]
↕ // ... line 48
49     private ?string $username = null;
↕ // ... lines 50 - 170
171 }
```

With *just* that change... when we try it... yes! The `owner` field just got transformed into an embedded object!

Embedded for One Endpoint, IRI for Another

Ok, let's also fetch a collection of `treasures`: just request all of them. Thanks to the change we just made, *every* single treasure's `owner` property is now an object.

That gives me a wild, hare-brained idea. What if having all the `owner` information when I fetch a *single* `DragonTreasure` is cool... but maybe it feels like overkill to have that data returned from the collection endpoint. Could we embed the `owner` when fetching a *single* `treasure`... but then use the IRI string when fetching a collection?

The answer is... no! I'm kidding - of course! We can do whatever crazy things we want! Though, the more weird things you add to your API, the trickier life gets. So choose your adventures wisely!

Doing this is a two-step process. First in `DragonTreasure`, find the `Get` operation, which is the operation for fetching a *single* treasure. One of the options that you can pass *into* an operation is the `normalizationContext`... which will override the default. Add `normalizationContext`, then `groups` set to the standard `treasure:read`. Then add a *second* group that's specific to this operation: `treasure:item:get`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 25
26 #[ApiResource(
↕ // ... lines 27 - 28
29     operations: [
30         new Get(
31             normalizationContext: [
32                 'groups' => ['treasure:read', 'treasure:item:get'],
33             ],
34         ),
↕ // ... lines 35 - 38
39     ],
↕ // ... lines 40 - 53
54 )]
↕ // ... line 55
56 class DragonTreasure
57 {
↕ // ... lines 58 - 213
214 }
```

You can call this whatever you want... but I like this convention: resource name followed by `item` or `collection` then the HTTP method, like `get` or `post`.

And yes, I *did* forget the `groups` key: I'll fix that in a minute.

Anyways, if I *had* coded this correctly, it would mean that when this operation is used, the serializer will include all fields that are in at least *one* of these two groups.

Now we can leverage that. Copy the new group name. Then, over in `User`, above `username`, instead of `treasure:read`, paste that new group.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 46
47     #[Groups(['user:read', 'user:write', 'treasure:item:get'])]
↕ // ... line 48
49     private ?string $username = null;
↕ // ... lines 50 - 170
171 }
```

Let's check it out! Try the GET collection endpoint again. Yes! We're back to `owner` being an IRI string. And if we try the GET *one* endpoint.. oh, the owner is... also an IRI here too? That's my bad. Back on `normalization_context` I forgot to say `groups`. I was basically setting two meaningless options into `normalization_context`.

Let's try that again. This time... got it!

When you get fancy like this, it *does* get a bit harder to keep track of what serialization groups are being used and when. Though you can use the Profiler to help with that. For example, this is our most recent request for the single treasure.

If we open the profiler for that request... and go down to the Serializer section, we see the data that's being serialized... but more importantly the normalization context... including `groups` set to the two we expect.

This is also cool because you can see *other* context options that are set by API Platform. These control certain internal behavior.

Next: let's get crazy with our relationships by using a `DragonTreasure` endpoint to change the `username` field of that treasure's owner. Woh.

Chapter 22: Embedded Write

I'm going to try out the GET one treasure endpoint... using a real id. Perfect. Because of the changes we just made, the `owner` field is *embedded*.

What about *changing* the owner? Piece of crumb cake: as long as the field is writable... which ours is. Right now the `owner` is id 1. Use the PUT endpoint to update id 2. For the payload, set `owner` to `/api/users/3`.

And... execute! Bah! Syntax error. JSON is crabby. Remove the comma, try again and... yes! The `owner` comes back as the IRI `/api/users/3`.

Sending Embedded Data to Update

But *now* I want to do something wild! This treasure is owned by user 3. Let's go get their details. Open the GET one user endpoint, try it out, enter 3 and... there it is! The username is `burnout400`.

Here's the goal: while updating a `DragonTreasure` - so while using the PUT endpoint to `/api/treasures/{id}` - instead of changing from one owner to another, I want to change the existing owner's `username`. Something like this: instead of setting `owner` to the IRI string, set it to an object with `username` assigned to something new.

Would that work? Let's experiment! Hit Execute and it does *not*. It says:

> *"Nested documents for attribute `owner` are not allowed, use IRI instead."*

Allowing Writable Properties to be Embedded

So, at first glance, it looks like this isn't allowed: it looks like you can only use an IRI string here. But actually, this *is* allowed. The problem is that the `username` field is not *writable* via this *operation*.

Let's think about this. We're updating a `DragonTreasure`. This means that API Platform is using the `treasure:write` serialization group. That group *is* above the `owner` property, which is why we can change the `owner`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 25
26 #[ApiResource(
↕ // ... lines 27 - 49
50     denormalizationContext: [
51         'groups' => ['treasure:write'],
52     ],
↕ // ... line 53
54 )]
↕ // ... line 55
56 class DragonTreasure
57 {
↕ // ... lines 58 - 99
100     #[Groups(['treasure:read', 'treasure:write'])]
101     private ?User $owner = null;
↕ // ... lines 102 - 213
214 }
```

But if we want to be able to change the owner's `username`, then we *also* need to go into `User` and add that group here.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 46
47     #[Groups(['user:read', 'user:write', 'treasure:item:get',
   'treasure:write'])]
↕ // ... line 48
49     private ?string $username = null;
↕ // ... lines 50 - 170
171 }
```

This works exactly like embedded fields when we *read* them. Basically, since at least *one* field in `User` has the `treasure:write` group, we are *now* allowed to send an *object* to the `owner` field.

New vs Existing Objects in Embedded Data

Watch: fire it up again. It works... almost. We get a 500 error:

> *"A new entity was found through the relationship `DragonTreasure.owner`, but was not configured to `cascade` persist."*

Woh. This means that the serializer saw our data, created a *new* `User` object and then set the `username` onto it. Doctrine failed because we never told it to persist the new `User` object.

Though... that's not the point: the point is that we don't *want* a new `User`! We want to grab the existing owner and update *its* `username`.

By the way, to make this example more realistic, let's also add a `name` to the payload so we can pretend that we're *actually* updating the treasure... and decide to *also* update the `username` of the owner while we're in the neighborhood.

Anyways: how do we tell the serializer to use the *existing* owner instead of creating a new one? By adding an `@id` field set to the IRI of the user: `/api/users/3`.

That's it! When the serializer sees an object, if it does *not* have an `@id`, it creates a new object. If it *does* have an `@id`, it finds *that* object and then sets any data onto it.

So, moment of truth. When we try it... of course, another syntax error. Get it together Ryan! After fixing that... perfect! A 200 status code! Though... we can't really see if it updated the `username` here... since it just shows the owner.

Use the GET one `User` endpoint... find user 3... and check that sweet data! It *did* change the `username`.

Ok, so I realize that this example may not have been the most realistic, but being able to update related objects *does* have plenty of real use-cases.

Cascading the Persist to Create a new Object

Looking back at that `PUT` request, what if we *did* want to allow a new `User` object to be created and saved? Is that possible? It *is*!

First, we would need to add a `cascade: ['persist']` to the `treasure.owner` `ORM\Column` attribute. This is something we'll see later. And second, we would need to make

sure to expose all of the required fields as writable. Right now only `username` is writable... so we couldn't send `password` or `email`.

The Valid Constraint

Before we keep going, we *are* missing one small, but important, detail. Let's try this update one more time with the `@id`. But set `username` to an empty string.

Remember, the `username` field has a `NotBlank` above it, so this should fail validation. And yet, when we try it, we get a 200 status code! And if we go to the GET one user endpoint... yeah, the `username` is now empty! That's... a problem.

How did that happen? Because of how Symfony's validation system works.

The top-level entity - the object that we're modifying directly - is `DragonTreasure`. So the validation system looks at `DragonTreasure` and it executes all of the validation constraints. However, when it gets to an object like the `owner` property, it stops. It does *not* continue to validate *that* object as well.

If you want that to happen, you need to add a constraint to this called `Assert\Valid`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 55
56 class DragonTreasure
57 {
↕ // ... lines 58 - 100
101     #[Assert\Valid]
102     private ?User $owner = null;
↕ // ... lines 103 - 214
215 }
```

Now... on our PUT endpoint... if we try this again, yep! 422: `owner.username`, this value should not be blank.

Being able to update an embedded object is really neat & powerful. But the cost of this is making your API more and more complex. So while you *can* choose to do this - and you should if it's what you want - you might also choose to force the API client to update the treasure first... and then make a second request to update the user's username... instead of allowing them to do it all fancy at the same time.

Next: let's look at this relationship from the *other* side. When we're updating a `User`, could we also update the *treasures* that belong to that user? Let's find out!

Chapter 23: Adding Items to a Collection Property

Let's fetch a single user in our API: I know one exists with ID 2. And cool!

As we learned earlier, exposing a *collection* relation property is just like any other field: simply make sure that it's in the correct serialization group. And then you can go *further* with serialization groups to choose between making it return as an array of IRI strings or as an array of embedded objects, like we have now.

New question: could we also *modify* the `dragonTreasures` that a user owns from one of the user operations? The answer is, of course, yea! And we're going to do this in increasingly crazy ways.

Making the Collection Field Writable

Look at the POST endpoint. We don't see a `dragonTreasures` field right now because... the field simply isn't writable: it's not in the correct group. To remedy that, we know what to do: add `user:write`.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 51
52     #[Groups(['user:read', 'user:write'])]
53     private Collection $dragonTreasures;
↕ // ... lines 54 - 170
171 }
```

Easy peasy! When we refresh the docs, and check that endpoint... there we go: `dragonTreasures`. And it says that this field should be an array of strings: an array of IRI strings.

Let's try crafting a new user. Fill in the `email` and `username`. Then, let's assign the new user to a few *existing* treasures. Let's sneak up to the GET collection endpoint for treasures... and awesome. We have ids 2, 3 and 4.

Back down here, assign `owner` to an array with `/api/treasures/2`, `/api/treasures/3` and `/api/treasures/4`.

Makes sense, right? If the API can return `dragonTreasures` as an array of IRI strings, why can't we *send* an array of IRI strings? When we hit Execute... indeed! It worked perfectly!

And since each treasure can have only one owner... it means that we kinda stole those treasures from someone else! Sorry!

The adder & remover Methods for Collections

But... wait a second, how did that work? We know that when we send fields like `email`, `password`, and `username`, because those are private properties, the serializer calls the setter methods. When we pass `username`, it calls `setUsername()`.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 134
135     public function setUsername(string $username): self
136     {
137         $this->username = $username;
138 
139         return $this;
140     }
↕ // ... lines 141 - 170
171 }
```

So when we pass `dragonTreasures`, it must call `setDragonTreasures`, right?

Well guess what? We don't *have* a `setDragonTreasures()` method! But we *do* have an `addDragonTreasure()` method and a `removeDragonTreasure()` method.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 149
150     public function addDragonTreasure(DragonTreasure $treasure): self
151     {
152         if (!$this->dragonTreasures->contains($treasure)) {
153             $this->dragonTreasures->add($treasure);
154             $treasure->setOwner($this);
155         }
156 
157         return $this;
158     }
159 
160     public function removeDragonTreasure(DragonTreasure $treasure): self
161     {
162         if ($this->dragonTreasures->removeElement($treasure)) {
163             // set the owning side to null (unless already changed)
164             if ($treasure->getOwner() === $this) {
165                 $treasure->setOwner(null);
166             }
167         }
168 
169         return $this;
170     }
171 }
```

The serializer is really smart. It sees that the new `User` object has no `dragonTreasures`. So it recognizes that each of these three objects are *new* to this user and so it calls `addDragonTreasure()` once for each.

And the way that MakerBundle generated these methods is *critical*. It takes the new `DragonTreasure` and sets the `owner` to be *this* object. That's important because of how Doctrine handles relationships: setting the owner sets what's called the "owning" side of the relationship. Basically, without this, Doctrine wouldn't save this change to the database.

The takeaway is that, thanks to `addDragonTreasure()` and its magical powers, the `owner` of the `DragonTreasure` is changed from its old owner to the new `User`, and everything saves exactly like we want.

Next, let's get more complex by allowing treasures to be *created* when we're creating a new `User`. We're also going to allow treasures to be *removed* from a `User`... for the unlikely event that the dwarves take back the mountain. As if.

Chapter 24: Creating Embedded Objects

Is it possible to create a totally *new* `DragonTreasure` when we create a user? Like... instead of sending the IRI of an *existing* treasure, we send an object?

Let's try it! First, I'll change this to a unique email and username. Then, for `dragonTreasures`, clear those IRIs and, instead, pass a JSON object with the fields that we know are required. Our new dragon user just scored a copy of GoldenEye for N64! *Legendary*. Add a `description`... and a `value`.

In theory, this JSON body makes *sense*! But does it work? Hit "Execute" and... nope! Well, not yet. But we know this error!

> *"Nested documents for attribute `dragonTreasures` are not allowed. Use IRIs instead."*

Making dragonTreasures Accept JSON Objects

Inside `User`, if we scroll *way* up, the `$dragonTreasures` property *is* writable because it has `user:write`.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 51
52     #[Groups(['user:read', 'user:write'])]
53     private Collection $dragonTreasures;
↕ // ... lines 54 - 170
171 }
```

But we can't send an *object* for this property because we haven't added `user:write` to any of the fields *inside* of `DragonTreasure`. Let's fix that.

We want to be able to send `$name`, so add `user:write`... I'll skip `$description` but do the same for `$value`. Now search for `setTextDescription()` which is the *actual* description. Add `user:write` here too.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 55
56 class DragonTreasure
57 {
↕ // ... lines 58 - 63
64     #[Groups(['treasure:read', 'treasure:write', 'user:read',
'user:write'])]
↕ // ... lines 65 - 67
68     private ?string $name = null;
↕ // ... lines 69 - 79
80     #[Groups(['treasure:read', 'treasure:write', 'user:read',
'user:write'])]
↕ // ... lines 81 - 82
83     private ?int $value = 0;
↕ // ... lines 84 - 138
139     #[Groups(['treasure:write', 'user:write'])]
140     public function setTextDescription(string $description): self
141     {
↕ // ... lines 142 - 144
145     }
↕ // ... lines 146 - 214
215 }
```

Okay, *in theory*, we should *now* be able to send an embedded object. If we head over and try it again... we upgraded to a 500 error!

> *"A new entity was found through the relationship `User#dragonTreasures`"*

Cascading an Entity Relation Persist

This is great! We already know that when you send an embedded object, if you include `@id`, the serializer will fetch that object first and *then* update it. But if you *don't* have an `@id`, it will create a brand *new* object. Right now, it *is* creating a new object,... but nothing told the entity manager to *persist* it. *That's* why we're getting this error.

To solve this, we need to *cascade* persist this property. In `User`, on the `OneToMany` for `$dragonTreasures`, add a `cascade` option set to `['persist']`.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 50
51     #[ORM\OneToMany(mappedBy: 'owner', targetEntity:
   DragonTreasure::class, cascade: ['persist'])]
↕ // ... line 52
53     private Collection $dragonTreasures;
↕ // ... lines 54 - 170
171 }
```

This means that if we're saving a `User` object, it should magically persist any `$dragonTreasures` inside. And if we try it now... it works! That's awesome! And apparently, our new treasure `id` is `43`.

Let's open up a new browser tab and navigate to that URL... plus `.json`... actually, let's do `.jsonld`. Beautiful! We see that the `owner` is set to the new user that we just created.

How was owner Set? Again: The Smart Methods

But... hold your horses! We didn't *send* the `owner` field in the treasure data... so how did that field get set? Well, first, it *does* make sense that we didn't send an `owner` field for the new `DragonTreasure`... since the user that will own it didn't even exist yet! Ok, then, but who *did* set the `owner`?

Behind the scenes, the serializer creates a new `User` object *first*. *Then*, it creates a new `DragonTreasure` object. Finally, it sees that the new `DragonTreasure` is *not* assigned to the `User` yet, and it calls `addDragonTreasure()`. When it does that, the code down here sets the `owner`: just like we saw before. So our well-written code is taking care of all of those details *for* us.

src/Entity/User.php

```
// ... lines 1 - 22
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 25 - 149
    public function addDragonTreasure(DragonTreasure $treasure): self
    {
        if (!$this->dragonTreasures->contains($treasure)) {
            $this->dragonTreasures->add($treasure);
            $treasure->setOwner($this);
        }

        return $this;
    }
// ... lines 159 - 170
}
```

Adding the Valid Constraint

Anyways, you might remember from before that as soon as we allow a relation field to send embedded data... we need to add *one* tiny thing. I won't do it, but if we sent an empty `name` field, it *would* create a `DragonTreasure`... with an empty `name`, even though, over here, if we scroll up to the `name` property, it's required! Remember: when the system validates the `User` object, it will stop at `$dragonTreasures`. It won't *also* validate those objects. If you *do* want to validate them, add `#[Assert\Valid]`.

src/Entity/User.php

```
// ... lines 1 - 22
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
// ... lines 25 - 52
    #[Assert\Valid]
    private Collection $dragonTreasures;
// ... lines 55 - 171
}
```

Now that I have this, to prove that it's working, hit "Execute" and... awesome! We get a 422 status code telling us that `name` shouldn't be empty. I'll go put that back.

Sending Embedded Objects and IRI Strings at the Same Time

We now know that we can send IRI strings *or* embedded objects for a relation property - assuming we've setup the serialization groups to allow that. *And*, we can even *mix* them.

Let's say that we want to create a new `DragonTreasure` object, but we're also going to steal, *borrow*, a treasure from another dragon. This is *totally* allowed. Watch! When we hit "Execute"... we get a 201 status code. This returns treasure ids `44` (that's the new one) *and* `7`, which is the one we just stole.

Okay, we only have one more chapter about handling relationships. Let's see how we can *remove* a treasure from a user to *delete* that treasure. That's *next*.

Chapter 25: Removing Items from a Collection

Our brand-new user is the proud owner of *two* treasures with IDs `7` and `44`. Let's update this user to see if we can make some changes to `$dragonTreasures`. Use the `PUT` endpoint, click "Try it out", and... let's see... the `id` we need is `14`... so I'll enter that. I'll also remove every field *except* for `dragonTreasures` so we can focus.

We know that this currently has *two* dazzling treasures - `/api/treasures/7` and `/api/treasures/44`. So if we send this request, *in theory*, that should do... *nothing*! And if we look down here... yeah: it made no changes at all.

Suppose we want to add a *new* `DragonTreasure` to this resource. To do that, we list the two that it already has, along with `/api/treasures/8`. I'm totally guessing that's a valid `id`. When we hit "Execute"... that works *beautifully*. The serializer system noticed that it already had these first two, so it didn't do anything with those. It just added the new one with id `8`.

Removing an Item from a Collection

That's *cool*, but what I really want to talk about is *removing* a treasure. Let's say that our dragon left one of these treasures in their pants pocket and accidentally washed it in the laundry. I can't blame them. I lose my lip balm in there all the time. Since the treasure is soggy and useless now, we need to remove it from the list of treasures. No problem! We'll just mention the two our dragon *still* has and remove the other one. When we hit "Execute"... it *explodes*!

> *“An exception occurred while executing a query: [...] Not null violation: 7. null value in column "owner_id"”*

What happened? Well, our app set the `$owner` property for the `DragonTreasure` we just removed to `null`... and is now trying to save it. *But* since we have it set to `nullable: false`, it's failing.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 55
56 class DragonTreasure
57 {
↕ // ... lines 58 - 97
98     #[ORM\ManyToOne(inversedBy: 'dragonTreasures')]
99     #[ORM\JoinColumn(nullable: false)]
↕ // ... lines 100 - 101
102     private ?User $owner = null;
↕ // ... lines 103 - 214
215 }
```

But... let's take a step back and look at the *whole* picture. First, the serializer noticed that treasures `7` and `8` are *already* owned by the `User`... so it did nothing with those. But *then* it noticed that the treasure with id 44 - which *was* owned by this `User` - is missing!

Because of that, over on our `User` class, the serializer called `removeDragonTreasure()`. What's really important is that it takes that `DragonTreasure` and set the `owner` to `null` to break the relationship. Depending on your app, that might be *exactly* what you want. Maybe you allow `dragonTreasures` to have *no* `owner`... like... they're still undiscovered and waiting for a dragon to find them. If *that's* the case, you'll just want to make sure that your relationship allows `null`... and everything will save just fine.

But in *our* case, if a `DragonTreasure` no longer has an `owner`, we want to *delete* it *completely*. We can do that in `User`... *way* up on the `dragonTreasures` property. After `cascade`, add one more option here: `orphanRemoval: true`.

src/Entity/User.php

```
↕ // ... lines 1 - 22
23 class User implements UserInterface, PasswordAuthenticatedUserInterface
24 {
↕ // ... lines 25 - 50
51     #[ORM\OneToMany(mappedBy: 'owner', targetEntity:
   DragonTreasure::class, cascade: ['persist'], orphanRemoval: true)]
↕ // ... lines 52 - 53
54     private Collection $dragonTreasures;
↕ // ... lines 55 - 171
172 }
```

This tells Doctrine that if any of these `dragonTreasures` become "orphaned" - meaning they no longer have *any* owner - they should be *deleted*.

Let's try it. When we hit "Execute" again... got it! It saves just fine.

Next: Let's circle back to filters and see how we can use them to search across related resources.

Chapter 26: Filtering on Relations

Earlier, we added a bunch of nice filters to `DragonTreasure`. Let's add a few more - starting with `User` - so we can show off some filtering *superpowers* for relations.

Using PropertyFilter Across Relations

Start like normal: `ApiFilter` and let's first use `PropertyFilter::class`. Remember: this is kind of a fake filter that allows our API client to select which *fields* they want. And this is all pretty familiar so far.

src/Entity/User.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Metadata\ApiFilter;
↕ // ... line 6
7 use ApiPlatform\Serializer\Filter\PropertyFilter;
↕ // ... lines 8 - 22
23 #[ApiFilter(PropertyFilter::class)]
↕ // ... lines 24 - 25
26 class User implements UserInterface, PasswordAuthenticatedUserInterface
27 {
↕ // ... lines 28 - 174
175 }
```

When we head over, refresh, and go to the `GET` collection endpoint... we see a new `properties[]` field. We could choose to return just `username`... or `username` *and* `dragonTreasures`.

When we hit "Execute"... perfect! We see the two fields... where `dragonTreasures` is an array of objects, each containing the fields we chose to embedded.

Again, this is super duper normal. So let's try something more interesting. In fact, what we're going to try isn't supported directly in the interactive docs.

So, copy this URL... paste and add `.jsonld` to the end.

Here's the goal: I want to return the `username` field and then *only* the `name` field of each dragon treasure. The syntax is a bit ugly: it's `[dragonTreasures]`, followed by `[]=name`.

And just like that... it only shows `name`! So right out of the box, `PropertyFilter` allows us to reach across relationships.

Searching Relation Fields

Let's do something *else*. Head back to `DragonTreasure`. It might be handy to filter by the `$owner`: we could quickly get a list of all treasures for a specific user.

No sweat! Just add `ApiFilter` above the `$owner` property, passing in the trusty `SearchFilter::class` followed by `strategy: 'exact'`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 55
56 class DragonTreasure
57 {
↕ // ... lines 58 - 101
102     #[ApiFilter(SearchFilter::class, strategy: 'exact')]
103     private ?User $owner = null;
↕ // ... lines 104 - 215
216 }
```

Back over on the docs, if we open up the `GET` collection endpoint for treasures and give it a whirl... let's see... here we go - "owner". Enter something like `/api/users/4`... assuming that's actually a real user in our database, and... yes! Here are the *five* treasures owned by that user!

But I want to get crazier: I want to find all treasures that are owned by a user matching a specific *username*. So instead of filtering on `owner`, we need to filter on `owner.username`.

How? Well, when we want to filter simply by `owner`, we can put the `ApiFilter` right above that property. But since we want to filter on `owner.username`, we can't put that above a property... because `owner.username` *isn't* a property. This is one of the cases where we need to put the filter above the *class*. And... that also means we need to add a `properties` option set to an array. Inside, say `'owner.username'` and set that to the `partial` strategy.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 55
56 #[ApiFilter(SearchFilter::class, properties: [
57     'owner.username' => 'partial',
58 ])]
59 class DragonTreasure
60 {
↕ // ... lines 61 - 218
219 }
```

Ok! Head back over and refresh. We know we have an owner whose username is "Smaug"... so let's go back to the `GET` collection endpoint and... here in `owner.username`, search for "maug"... and hit "Execute".

Let's see... That worked! This shows all treasures owned by any user whose username contains `maug`. Pretty cool!

Ok squad: get ready for the grand finale - *Subresources*. These have *seriously* changed in API Platform 3. Let's dive into them next.

Chapter 27: Subresources

We have *two* different ways to get the dragon treasures for a user. First, we could fetch the `User` and read its `dragonTreasures` property. The second is via the filter that we added a moment ago. In the API, that looks like `owner=/api/users/4` on the `GET` collection operation for treasures.

This is *my* go-to way of getting the data... because if I want to fetch treasures, it make sense to use a `treasures` endpoint. Besides, if a user owns a *lot* of treasures, that'll give us pagination!

But you may sometimes choose to add a *special* way to fetch a resource or collection of resources... almost like a vanity URL. For example, imagine that, to get this same collection, we want the user to be able to go to `/api/users/4/treasures.jsonld`. That, of course, doesn't work. But it *can* be done. This is called a *subresource*, and subresources are *much* nicer in API platform 3.

Adding a Subresource via Another ApiResource

Okay, let's *think*. This endpoint will return treasures. So to add this *subresource*, we need to update the `DragonTreasure` class.

How? By adding a *second* `ApiResource` attribute. We already have this main one, so now add a *new* one. But this time, control the URL with a `uriTemplate` option set to exactly what we want: `/users/{user_id}` for the wildcard part (we'll see how that's used in a moment) then `/treasures`.

That's it! Well... also add `.{_format}`. This is *optional*, but that's the magic that lets us "cheat" and add this `.jsonld` to the end of the URL.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 54
55 #[ApiResource(
56     uriTemplate: '/users/{user_id}/treasures.{_format}',
↕ // ... line 57
58 )]
↕ // ... lines 59 - 62
63 class DragonTreasure
64 {
↕ // ... lines 65 - 222
223 }
```

Next, add `operations`... because we don't need *all six*... we really need just *one*. So, say `[new GetCollection()]` because we will return a *collection* of treasures.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 54
55 #[ApiResource(
56     uriTemplate: '/users/{user_id}/treasures.{_format}',
57     operations: [new GetCollection()],
58 )]
↕ // ... lines 59 - 62
63 class DragonTreasure
64 {
↕ // ... lines 65 - 222
223 }
```

Ok, let's see what this did! Head back to the documentation and refresh. Suddenly we have... *three* resources and this one has the correct URL!

Oh, and we have *three* resources because, if you recall, we *customized* the `shortName`. Copy that and paste it onto the new `ApiResource` so they match. And to make PhpStorm happy, I'll put them in *order*.

src/Entity/DragonTreasure.php

```
// ... lines 1 - 54
#[ApiResource(
    uriTemplate: '/users/{user_id}/treasures.{_format}',
    shortName: 'Treasure',
    operations: [new GetCollection()],
)]
// ... lines 60 - 63
class DragonTreasure
{
// ... lines 66 - 223
}
```

Now when we refresh... perfect! *That's* what we want!

Understanding uriVariables

We now have a new operation for fetching treasures. But does it *work*? It *says* that it will retrieve a collection of treasure resources, so that's good. But... we have a *problem*. It thinks that we need to pass the `id` of a `DragonTreasure`... but it should be the id of a `User`! And even if we pass something, like `4`... and hit "Execute"... look at the URL! It didn't even use the `4`: it still has `{user_id}` in the URL! So *of course* it comes back with a 404 error.

The problem is that we need to help API Platform understand what `{user_id}` *means*. We need to tell it that this is the id of the *user* and that it should use that to query `WHERE owner_id` equals the value.

To do that, add a new option called `uriVariables`. This is where we describe any "wildcards" in your URL. Pass `user_id` set to a `new Link()` object. There are multiple... we want the one from `ApiPlatform\Metadata`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 use ApiPlatform\Metadata\Link;
↕ // ... lines 13 - 55
56 #[ApiResource(
57     uriTemplate: '/users/{user_id}/treasures.{_format}',
58     shortName: 'Treasure',
59     operations: [new GetCollection()],
60     uriVariables: [
61         'user_id' => new Link(
↕ // ... lines 62 - 63
64         ),
65     ],
66 )]
↕ // ... lines 67 - 70
71 class DragonTreasure
72 {
↕ // ... lines 73 - 230
231 }
```

This object needs *two* things. First, point to the *class* that the `{user_id}` is referring to. Do that by passing a `fromClass` option set to `User::class`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 use ApiPlatform\Metadata\Link;
↕ // ... lines 13 - 55
56 #[ApiResource(
57     uriTemplate: '/users/{user_id}/treasures.{_format}',
58     shortName: 'Treasure',
59     operations: [new GetCollection()],
60     uriVariables: [
61         'user_id' => new Link(
↕ // ... line 62
63             fromClass: User::class,
64         ),
65     ],
66 )]
↕ // ... lines 67 - 70
71 class DragonTreasure
72 {
↕ // ... lines 73 - 230
231 }
```

Second, we need to define which *property* on `User` *points* to `DragonTreasure` so that it can figure out how to structure the query. To do *this*, set `fromProperty` to `treasures`. So, inside

`User`, we're saying that this property describes the relationship. Oh, but I totally messed that up: the property is `dragonTreasures`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 use ApiPlatform\Metadata\Link;
↕ // ... lines 13 - 55
56 #[ApiResource(
57     uriTemplate: '/users/{user_id}/treasures.{_format}',
58     shortName: 'Treasure',
59     operations: [new GetCollection()],
60     uriVariables: [
61         'user_id' => new Link(
62             fromProperty: 'dragonTreasures',
63             fromClass: User::class,
64         ),
65     ],
66 )]
↕ // ... lines 67 - 70
71 class DragonTreasure
72 {
↕ // ... lines 73 - 230
231 }
```

Ok, cruise back over and refresh. Under the endpoint... yea! It says "User identifier". Let's put `4` in there again, hit "Execute" and... *got it*. There are the *five* treasures for this user!

And in the other browser tab... if we refresh... it *works*!

How the Query is Made

Behind the scenes, thanks to the `Link`, API Platform basically makes the following query:

> *"SELECT * FROM dragon_treasure WHERE owner_id ="*

whatever we pass for `{user_id}`. It knows how to make that query by looking at the Doctrine relationship and figuring out which column to use. It's *super* smart.

We can actually see this in the profiler. Go to `/_profiler`, click on our request... and, down here, we see 2 queries... which are basically the same: the 2nd is used for the "total items" for pagination.

If you click "View formatted query" on the main query... it's even more complex than I expected! It has an `INNER JOIN`... but it's basically selecting all the dragon treasures data where `owner_id` = the ID of that user.

What about toProperty?

By the way, if you look at the documentation, there's also a way to set all of this up via the *other* side of the relationship: by saying `toProperty: 'owner'`.

This still works... and works exactly the same. But I recommend sticking with `fromProperty`, which is consistent and, I think, more clear. The `toProperty` is needed only if you didn't map the *inverse* side of a relationship... like if there was *no* `dragonTreasures` property on `User`. Unless you have that situation, stick with `fromProperty`.

Don't Forget normalizationContext!

This is all working nicely except for one small problem. If you look back at the data, it shows the wrong *fields*! It's returning *everything*, like `id` and `isPublished`.

Those aren't supposed to be included thanks of our normalization groups. But since we haven't *specified* any normalization groups on the new `ApiResource`, the serializer returns everything.

To fix this, copy the `normalizationContext` and paste it down here. We don't need to worry about `denormalizationContext` because we don't have any operations that do any denormalizing.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 11
12 use ApiPlatform\Metadata\Link;
↕ // ... lines 13 - 55
56 #[ApiResource(
57     uriTemplate: '/users/{user_id}/treasures.{_format}',
58     shortName: 'Treasure',
59     operations: [new GetCollection()],
60     uriVariables: [
61         'user_id' => new Link(
62             fromProperty: 'dragonTreasures',
63             fromClass: User::class,
64         ),
65     ],
66     normalizationContext: [
67         'groups' => ['treasure:read'],
68     ],
69 )]
↕ // ... lines 70 - 73
74 class DragonTreasure
75 {
↕ // ... lines 76 - 233
234 }
```

If we refresh now... got it!

A Single Subresource Endpoint

Let's add *one* more subresource to see a slightly different case. I'll show you the URL I want first. We have a treasure with ID `11`. This means we can go to `/api/treasures/11.jsonld` to see that. *Now* I want to be able to add `/owner` to the end to get the *user* that owns this treasure. Right now, that doesn't work.... so let's *get* to work!

Because the resource that will be returned is a `User`, *that's* the class that needs the new API Resource.

Above it, add `#[ApiResource()]` with `uriTemplate` set to `/treasures/{treasure_id}` for the wildcard (though this can be called anything), followed by `/owner.{_format}`.

src/Entity/User.php

```
↕ // ... lines 1 - 24
25 #[ApiResource(
26     uriTemplate: '/treasures/{treasure_id}/owner.{_format}',
↕ // ... lines 27 - 34
35 )]
↕ // ... lines 36 - 38
39 class User implements UserInterface, PasswordAuthenticatedUserInterface
40 {
↕ // ... lines 41 - 187
188 }
```

Next pass `uriVariables` with `treasure_id` set to a `new Link()` - the one from `ApiPlatform\Metadata`. Inside, set `fromClass` to `DragonTreasure::class`. And since the *property* inside `DragonTreasure` that refers to this relationship is `owner`, add `fromProperty: 'owner'`.

src/Entity/User.php

```
↕ // ... lines 1 - 7
8 use ApiPlatform\Metadata\Link;
↕ // ... lines 9 - 24
25 #[ApiResource(
26     uriTemplate: '/treasures/{treasure_id}/owner.{_format}',
↕ // ... line 27
28     uriVariables: [
29         'treasure_id' => new Link(
30             fromProperty: 'owner',
31             fromClass: DragonTreasure::class,
32         ),
33     ],
↕ // ... line 34
35 )]
↕ // ... lines 36 - 38
39 class User implements UserInterface, PasswordAuthenticatedUserInterface
40 {
↕ // ... lines 41 - 187
188 }
```

We also know that we're going to need the `normalizationContext`... so copy that... and paste it here. Finally, we only want *one* operation: a `GET` operation to return a single `User`. So, add `operations` set to `[new Get()]`.

src/Entity/User.php

```
↕ // ... lines 1 - 6
7  use ApiPlatform\Metadata\Get;
8  use ApiPlatform\Metadata\Link;
↕ // ... lines 9 - 24
25 #[ApiResource(
26     uriTemplate: '/treasures/{treasure_id}/owner.{_format}',
27     operations: [new Get()],
28     uriVariables: [
29         'treasure_id' => new Link(
30             fromProperty: 'owner',
31             fromClass: DragonTreasure::class,
32         ),
33     ],
34     normalizationContext: ['groups' => ['user:read']],
35 )]
↕ // ... lines 36 - 38
39 class User implements UserInterface, PasswordAuthenticatedUserInterface
40 {
↕ // ... lines 41 - 187
188 }
```

That should do it! Move back over to the documentation, refresh, and take a look under "User". Yep! We have a new operation! And it even sees that the wildcard is a "DragonTreasure identifier".

If we go refresh the other tab... it works!

Ok team, I lied about this being the last topic because... it's bonus topic time! Next: let's create a React-based admin area automatically from our API docs. Woh.

Chapter 28: React Admin

Whoa! Look out! *Bonus* chapter! We *know* that our API is fully described using the Open API spec. Heck, we can even see it by going to `/api/docs.json`. This shows all our different endpoints and their fields. It gets this delicious info by reading our *code*, PHPdoc, and other things. And we know this is used to power the Swagger UI docs page. Our API is *also* described by JSON-LD and Hydra.

And, both of these types of API docs can be used to power *other* things.

For example, search for "react admin", to find an open source React-based admin system. This is *super* powerful and cool... and it's been around for a long time. And the way it works is... amazing: we point it at our API documentation and then... it just builds itself! I think we should take it for a test drive.

Search for "api platform react admin" to find the API Platform docs page all about this. This has some info... but what we're really after is over here. Click "Get Started". This walks us through all the details, even including CORS config if you have that problem.

So... let's do this!

Webpack Encore Setup

If you use the API Platform Docker distribution, this admin area comes pre-installed. But it's also easy enough to add manually. Right now, our app doesn't have *any* JavaScript, so we need to bootstrap everything. Find your terminal and run:

```
composer require encore
```

This installs WebpackEncoreBundle... and its recipe gives us a basic frontend setup. When that's done, install the Node assets with:

```
npm install
```

Okay, flip back over to the docs. API Platform has their own Node package that helps integrate with the admin. So let's get that installed. Copy the `npm install` line - you can also use `yarn` if you want - paste it in the terminal, and add a `-D` at the end.

```
npm install @api-platform/admin -D
```

That `-D` isn't super important, but I tend to install my assets as `devDependencies`.

UX React Setup

To get all of this working, ultimately, we're going to render a single React component into a page. To help with that, I'm going to install a UX package that's... just really good at rendering React components. It's optional, but nice.

Run:

```
composer require symfony/ux-react
```

Perfect. Now, spin over and search for "symfony ux react" to find its documentation. Copy this setup code: we need to add it to our `app.js` file... over here in `assets/`. Paste... and we don't need all of these comments. I'll also move this code down below the imports.

assets/app.js

```
// ... lines 1 - 12
import './bootstrap';
registerReactControllerComponents(require.context('./react/controllers', true, /\.(j|t)sx?$/));
```

Awesome! This basically says that it will look in an `assets/react/controllers/` directory and make every React component inside super easy to render in Twig. So, let's create that: in

`assets/`, add two new directories: `react/controllers/`. And *then* create a new file called `ReactAdmin.jsx`.

For the *contents*, go back to the API Platform docs... and it gives us *almost* exactly what we need. Copy this... and paste it inside our new file. But first, it doesn't *look* like it, but thanks to the JSX syntax, we're using React, so we need an `import React from 'react'`.

And... let's make sure we have that installed:

```
npm install react -D
```

Passing a Prop to the React Component

Second, take a look at the `entrypoint` prop. This is *so* cool. We pass the URL to our API homepage... and then React admin takes care of the rest. For us, this URL would be something like `https://localhost:8000/api`. But... I'd rather not hardcode a "localhost" into my JavaScript.

Instead, we're going to pass this in as a prop. To allow that, add a `props` argument... then say `props.entrypoint`.

assets/react/controllers/ReactAdmin.jsx

```
import { HydraAdmin } from "@api-platform/admin";
import React from 'react';

export default (props) => (
  <HydraAdmin entrypoint={props.entrypoint} />
);
```

How do we pass this in? We'll see that in *just* a minute.

Enabling React in Encore

All right, let's see if the system will even build. Fire it up:

```
npm run watch
```

And... *syntax error*! It sees this `.jsx` syntax and... has no idea what to do with it! That's because we haven't enabled React inside of WebpackEncore yet. Hit Ctrl+C to stop that... then spin over and open `webpack.config.js`. Find a comment that says `.enableReactPreset()`. There it is. Uncomment that.

webpack.config.js

```
↕ // ... lines 1 - 8
9 Encore
↕ // ... lines 10 - 64
65     // uncomment if you use React
66     .enableReactPreset()
↕ // ... lines 67 - 77
```

Now when we run

```
npm run watch
```

again... it *still* won't work! *But* it gives us the command we need to install the one missing package for React support! Copy that, run it:

```
npm install @babel/preset-react@^7.0.0 --save-dev
```

And *now* when we try

```
npm run watch
```

... it works! Time to *render* that React component.

Rendering the ReactAdmin Component

How do we do that? This is the easy part. In `src/Controller/`, create a new PHP class called `AdminController`. This is probably going to be the most *boring* controller you'll ever create. Make it extend `AbstractController`, and then add a `public function` called `dashboard()`, which will return a `Response`, though that's optional. Above this, add a `Route()` for `/admin`.

All we need inside is `return $this->render()` and then a template: `admin/dashboard.html.twig`.

src/Controller/AdminController.php

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

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class AdminController extends AbstractController
{
    #[Route('/admin')]
    public function dashboard(): Response
    {
        return $this->render('admin/dashboard.html.twig');
    }
}
```

Cool! Down in the `templates/` directory, create that `admin/` directory... and inside, a new file called `dashboard.html.twig`. Again, this is probably one of the most boring *templates* you'll ever make, at least at the start. Extend `base.html.twig` and add `block body` and `endblock`.

Now, how do we render the React component? Thanks to that UX React package, it's *super* easy. Create the element that it should render into then add `react_component()` followed by the name of the component. Since the file is called `ReactAdmin.jsx` in the `react/controllers/` directory, its name will be `ReactAdmin`.

templates/admin/dashboard.html.twig

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

{% block body %}
    <div {{ react_component('ReactAdmin', {
// ... line 5
    }) }}></div>
{% endblock %}
```

And here's where we pass in those props. Remember: we have one called `entrypoint`. Oh, but let me fix my indentation... and remember to add the `</div>`. We don't need anything *inside* the div... because that's where the React admin area will magically appear, like a rabbit out of a hat.

Pass the `entrypoint` prop set to the normal `path()` function. Now, we *just* need to figure out the route name that API Platform uses for the API homepage. This tab is running npm... so I'll open a new terminal tab and run:

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

Woh! Too big. That's better. Scroll up a bit, and... here it is. We want: `api_entrypoint`. Head back over, and pass that in.

templates/admin/dashboard.html.twig

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

{% block body %}
    <div {{ react_component('ReactAdmin', {
        entrypoint: path('api_entrypoint')
    }) }}></div>
{% endblock %}
```

Moment of truth! Find your browser, change the address to `/admin`, and... *hello* ReactAdmin! Woh! Behind the scenes, that made a request to our API entrypoint, saw all of the different API resources we have, and it created this admin! I know, isn't that insane?

We won't go *too* deep into this, though you *can* customize it and you almost definitely *will* need to customize it. But we get a lot of stuff out of the box. It's not *perfect*: it looks a little confused by our embedded `dragonTreasures`, but it's already *very* powerful. Even the validation works!

Watch: when I submit, it reads the server-side validation returned by our API and assigned each error to the correct field. And treasures is aware of our filters. It's *all* here!

If this is interesting to you, *definitely* check it out further.

All right, team! You did it! You got through the first API Platform tutorial, which is fundamental to *everything*. You now understand how resources are serialized, how resources relate to other resources, IRIs, etc. All of these things are going to empower you no matter what API you're building. In the next tutorial, we'll talk about users, security, custom validation, user-specific fields and other wild stuff. Let us know what you're building and, if you have any questions, we're here for you down in the comments section.

Alright, friends! Seeya next time!

With <3 from SymfonyCasts