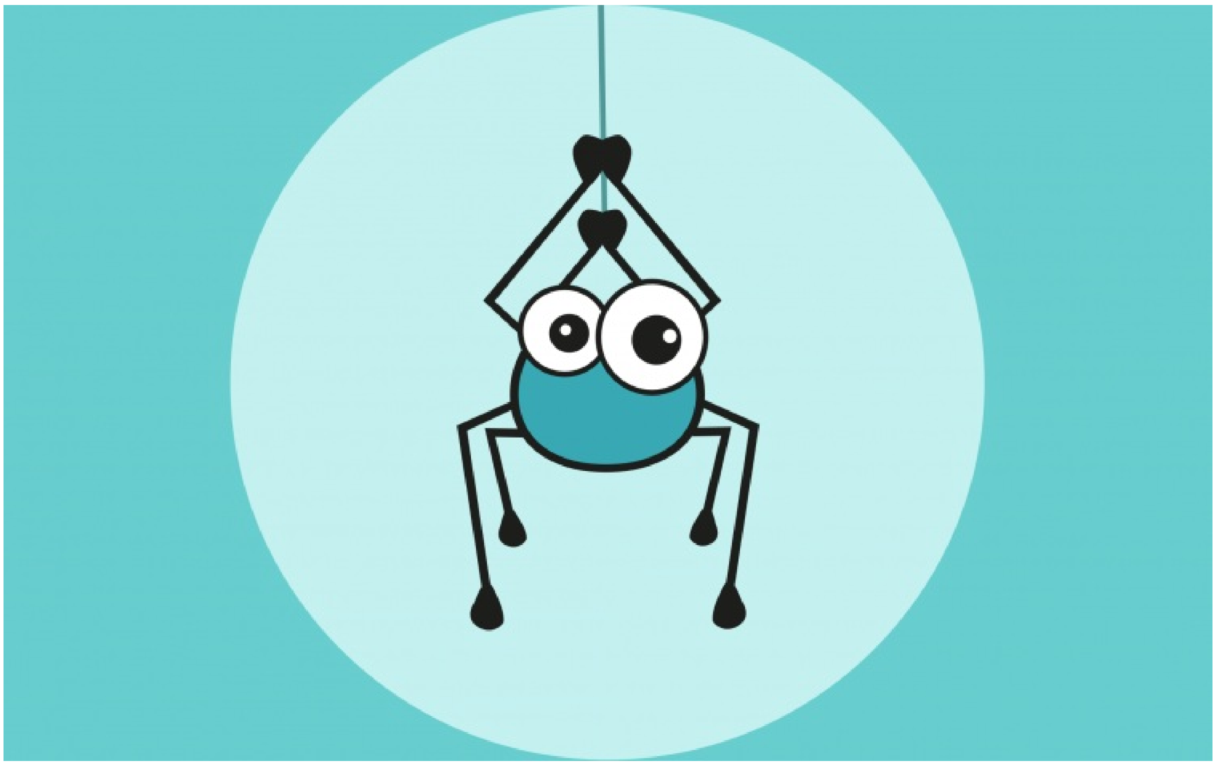
API Platform: Serious RESTful APIs



With <3 from SymfonyCasts

Chapter 1: API Platform Installation!

Yo friends! It's time to talk about... drum roll... how to bake a delicious cake that looks like an Oreo. Wait... ah! Wrong tutorial. It's time to talk about API Platform... so fun, it's *almost* as delicious as a cake shaped like an Oreo.

API Platform is *crushing* it these days. I feel like everywhere I turn, someone is *raving* about it! Its lead developer - Kévin Dunglas - is a core contributor of Symfony, *super* nice guy and is absolutely pushing the boundaries of what API's can do. We're going to see that first-hand. He was also nice enough to guide us on this tutorial!

Modern APIs are Hard. API Platform is not

If you only need to build a few API endpoints *just* to support some JavaScript, you might be thinking:

> What's the big deal? Returning some JSON here and there is already pretty easy!

I've had this same opinion for awhile. But little-by-little, I think this is becoming less and less true. *Just* like how frameworks were born when web apps became more and more complex, tools like API Platform have been created because the same things is currently happening with APIs.

These days, API's are more than just returning JSON: it's about being able to serialize and deserialize your models consistently, maybe into multiple formats, like JSON or XML, but also JSON-LD or HAL JSON. Then there's hypermedia, linked data, status codes, error formats, documentation - including API spec documentation that can power Swagger. Then there's security, CORS, access control and other important features like pagination, filtering, validation, content-type negotiation, GraphQL... and... honestly, I could keep going.

This is why API Platform exists: to allow us to build *killer* APIs and *love* the process! Oh, and that big list of stuff I just mentioned that an API needs? API Platform comes with *all* of it. And it's not just for building a *huge* API. It really is the perfect tool, even if you only need a few endpoints to power your own JavaScript.

API Platform Distribution

So let's do this! API Platform is an independent PHP library that's built on top of the Symfony components. You don't *need* to use it from inside a Symfony app, but, as you can see here, that's how they recommend using it, which is great for us.

If you follow their docs, they have their *own* API Platform *distribution*: a custom directory structure with a *bunch* of stuff: one directory for your Symfony-powered API, another for your JavaScript frontend, *another* for an admin frontend *all* wired together with Docker! Woh! It can feel a bit "big" to start with, but you get *all* of the features out-of-the-box... even more than I just described. If that sounds awesome, you can totally use that.

But we're going to do something different: we're going to install API Platform as a bundle into a normal, traditional Symfony app. It makes learning API Platform a bit easier. Once you're confident, for your project, you can do it this same way *or* jump in and use the official distribution. Like I said, it's *super* powerful.

Project Setup

Anyways, to become the API hero that we all need, you should *totally* code along with me by downloading the course code from this page. After you unzip it, you'll find a `start/` directory inside with the same code that you see here... which is *actually* just a new Symfony 4.2 skeleton project: there is *nothing* special installed or configured yet. Follow the `README.md` file for the setup instructions.

The *last* step will be to open a terminal, move into the project and start the Symfony server with:

```
$ symfony serve -d
```

This uses the `symfony` executable - an awesome little dev tool that you can get at https://symfony.com/download. This starts a web server on port 8000 that runs in the background. Which means that *we* can find our browser, head to `localhost:8000` and see... well, basically nothing! Just the nice welcome page you see in an empty Symfony app.

Installing API Platform

Now that we have our empty Symfony app, how can we install API Platform? Oh, it's so awesome. Find your terminal and run:

```
$ composer require api
```

That's it. You'll notice that this is installing something called `api-platform/api-pack`. If you remember from our Symfony series, a "pack" is sort of a "fake" library that helps install several thing at once.

Heck, you can see this at `https://github.com/api-platform/api-pack`: it's a single `composer.json` file that requires several libraries, like Doctrine, a CORS bundle that we'll talk about later, annotations, API Platform itself and a few parts of Symfony, like the validation system, security component and even twig, which is used to generate some really cool documentation that we'll see in a minute.

But, there's nothing *that* interesting yet: just API Platform and some standard Symfony packages.

Back in the terminal, it's done! And has some details on how to get started. A few recipes also ran that gave us some config files. Before we do *anything* else, go back to the browser and head to `https://localhost:8000/api` to see... woh! We have API documentation! Well, we don't even *have* any API yet... so there's nothing here. But this is going to be a *huge*, free feature you get with API Platform: as we build our API, this page will automatically update.

Let's see that next by creating and *exposing* our first API Resource.

Chapter 2: Our First ApiResource

Question: have you ever gone to the store and accidentally bought *too* much cheese? It's the story of my life. Or maybe you have the opposite problem: you're hosting a big party and you don't have *enough* cheese! This is our new billion dollar idea: a platform where you can sell that extra chunk of Brie you never finished or buy a truck-load of camembert from someone who go a little too excited at the cheese market. Yep, it's what the world is asking for: a peer-to-peer cheese marketplace. We're calling it: Cheese Whiz.

For the site, maybe we're going to make it an single-page application built in React or Vue... or maybe it'll be a little more traditional: a mixture of HTML pages and JavaScript that makes AJAX requests. And maybe we'll even have a mobile app. It doesn't really matter because *all* of these options mean that we need to be able to expose our core functionality via an API.

Generating the Entity

But to start: *forget* about the API and pretend like this is a normal, boring Symfony project. Step 1 is... hmm, probably do create some database entities.

Let's open up our `.env` file and tweak the `DATABASE_URL`. My computer uses `root` with no password... and how about `cheese_whiz` for the database name.

```
33 lines | .env
... lines 1 - 30
DATABASE_URL=mysql://root:@127.0.0.1:3306/cheese_whiz
... lines 32 - 33
```

You can also create a `.env.local` file and override `DATABASE_URL` there. Using `root` and no password is pretty standard, so I like to add this to `.env` and commit it as the default.

Cool! Next, at your terminal, run

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

to get Symfony's MakerBundle... so we can be lazy and generate our entity. When that finishes, run:

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

Call the first entity: `CheeseListing`, which will represent each "cheese" that's for sale on the site. Hit enter and... oh! It's asking:

> Mark this class as an API Platform resource?

MakerBundle asks this because it noticed that API Platform is installed. Say "yes". And before we add *any* fields, let's go see what that did! In my editor, yep! This created the usual `CheeseListing` and `CheeseListingRepository`. Nothing special there. Right now, the only property the entity has is `id`. So, what did answering "yes" to the API Platform resource question give us? This tiny annotation right here: `@ApiResource`.

```
111 lines | src/Entity/CheeseListing.php
... lines 1 - 7
/**
 * @ApiResource()
... line 10
 */
class CheeseListing
... lines 13 - 111
```

The *real* question is: what does that activate? We'll see that soon.

But first, let's add some fields. Let's see, each cheese listing probably needs a `title`, `string`, `255`, not nullable, a `description`, which will be a big text field, `price`, which I'll make an `integer` - this will be the price in *cents* - so $10 would be 1000, `createdAt` as a `datetime` and an `isPublished` boolean. Ok: good start! Hit enter to finish.

Congratulations! We have a *perfectly* boring `CheeseEntity` class: 7 properties with getters and setters.

```
111 lines | src/Entity/CheeseListing.php
... lines 1 - 11
class CheeseListing
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $title;

    /**
     * @ORM\Column(type="text")
     */
    private $description;

    /**
     * @ORM\Column(type="integer")
     */
    private $price;

    /**
     * @ORM\Column(type="datetime")
     */
    private $createdAt;

    /**
     * @ORM\Column(type="boolean")
     */
    private $isPublished;
... lines 45 - 109
}
```

Next, generate the migration with:

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

Oh! Migrations isn't installed yet! No problem, follow the recommendation:

```
$ composer require migrations
```

But before we try generating it again, I need to make sure my database exists:

```
$ php bin/console doctrine:database:create
```

And *now* run `make:migration`:

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

Let's go check that out to make sure there aren't any surprises:

> CREATE TABLE cheese_listing...

```
36 lines | src/Migrations/Version20190508193750.php
... lines 1 - 12
final class Version20190508193750 extends AbstractMigration
{
... lines 15 - 19
    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\

        $this->addSql('CREATE TABLE cheese_listing (id INT AUTO_INCREMENT NOT NULL, title VARCHAR(255) NOT NULL, description
    }
... lines 27 - 34
}
```

Yea! Looks good! Close that and run:

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

Say Hello to your API!

Brilliant! At this point, we have a *completely* traditional Doctrine entity *except* for this one, `@ApiResource()` annotation. But *that* changes everything. This tells API Platform that you want to *expose* this class as an API.

Check it out: refresh the `/api` page. Woh! Suddenly this is saying that we have *five* new endpoints, or "operations"! A `GET` operation to retrieve a collection of CheeseListings, a `POST` operation to create a new one, `GET` to retrieve a single `CheeseListing`, `DELETE` to... ya know... delete and `PUT` to update an existing `CheeseListing`. That's a full, API-based CRUD!

And this isn't just documentation: these new endpoints *already* work. Let's check them out next, say hello to something called JSON-LD and learn a bit about how this magic works behind the scenes.

Chapter 3: Swagger: Instant, Interactive API Docs

We're currently looking at something called Swagger: an open source API documentation interface. We're going talk more about it soon, but the idea is basically this: *if* you have an API - built in any language - and you create some configuration that *describes* that API in a format that Swagger understands, boom! Swagger can render this beautiful *interactive* documentation for you. Behind the scenes, API Platform is already preparing that configuration for Swagger.

Let's play with it! Open the POST endpoint. It says what this does and shows how the JSON should look to use it. Nice! Click "Try it out"! Let's see what's in my kitchen - some "Half-eaten blue cheese", which is still... *probably* ok to eat. We'll sell it for $1. What a bargain! And... Execute!

Um... what happened? Scroll down. Woh! It just made a `POST` request to `/api/cheese_listings` and sent our JSON! Our app *responded* with a 201 status code and... some weird-looking JSON keys: `@context`, `@id` and `@type`. *Then* it has the normal data for the new cheese listing: the auto-increment `id`, `title`, etc. Hey! We *already* have a working API... and this just proved it!

Close up the POST and open the `GET` that returns a collection of cheese listings. Try this one out too: Execute! Yep... there's our *one* listing... but it's *not* raw JSON. This extra stuff is called JSON-LD. It's just normal *JSON*, but with special *keys* - like `@context` - that have a specific meaning. Understanding JSON-LD is an important part of leveraging API Platform - and we'll talk more about it soon.

Anyways, to make things more interesting - go back to the POST endpoint and create a second cheese listing - a giant block of cheddar cheese... for $10. Execute! Same result: 201 status code and id 2.

Try the collection `GET` endpoint again. And... alright! Two results, with ids 1 and 2. And if we want to fetch just *one* cheese listing, we can do that with the other `GET` endpoint. As you can see, the `id` of the cheese listing that we want to fetch is part of the URL. This time, when we click to try it, cool! It gives us a box for the id. Use "2" and... Execute!

This makes a very simple GET request to `/api/cheese_listings/2`, which returns a `200` status code and the familiar JSON format.

Content-Type Negotiation

How cool is this! A full API "CRUD" with *no* work. Of course, the trick will be to *customize* this to our exact needs. But sheesh! This is an *awesome* start.

Let's try to hit our API directly - *outside* of Swagger - just to make sure this isn't all an elaborate trick. Copy the URL, open a new tab, paste and... hello JSON! Woh! Hello... API doc page again?

It scrolled us down to the documentation for this endpoint and executed it with id 2... which is cool... but what's going on? Do we *actually* have a working API or not?

Built into API Platform is something called Content-Type negotiation. Conveniently, when you execute an operation, Swagger shows you how you could make that *same* request using curl at the command line. And it includes one *critical* piece:

> `-H "accept: application/ld+json"`

That says: make a request with an `Accept` header set to `application/ld+json`. The request is *hinting* to API Platform that it should return the data in this JSON-LD format. Whether you realize it or not, your browser *also* sends this header: as `text/html`... cause... it's a browser. That basically tells API Platform:

> Hey! I want the CheeseListing with id 2 in HTML format.

API Platform responds by doing its best to do exactly that: it returns the HTML Swagger page with CheeseListing id 2 already showing.

Faking the Content-Type

This isn't a problem for an API client because setting an `Accept` header is easy. But... is there some way to kinda... "test" the endpoint in a browser? Totally! You can cheat: add `.jsonld` to the end of the URL.

Boom! *This* is our API endpoint in the JSON-LD format. I called this "cheating" because this little "trick" of adding the extension is *really* only meant for development. In the real world, you should set the `Accept` header instead, like if you were making an AJAX request from JavaScript.

And, check this out: change the extension to `.json`. That looks a bit more familiar!

This is a *great* example of the API Platform philosophy: instead of thinking about routes, controllers and responses, API Platform wants you to think about creating API resources - like `CheeseListing` - and then *exposing* that resource in a variety of different formats, like JSON-LD, normal JSON, XML or exposing it through a GraphQL interface.

Where do These Routes Come From?

Of course, as *awesome* as that is, if you're like me, you're probably thinking:

> This is cool... but how did all these endpoints get magically added to my app?

After all, we don't *normally* add an annotation to an entity... and suddenly get a bunch of functional pages!

Find your terminal and run:

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

Cool! API platform is bringing in several new routes: `api_entrypoint` is sort of the "homepage" of our api, which, by the way, can be returned as HTML - like we've been seeing - *or* as JSON-LD, for a machine-readable "index" of what's included in our API. More on that later. There's also a `/api/docs` URL - which, for HTML is the same as going to `/api`, another called `/api/context` - more on that in a minute - and below, 5 routes for the 5 new endpoints. When we add more resources later, we'll see more routes.

When we installed the API Platform pack, its recipe added a `config/routes/api_platform.yaml` file.

5 lines | config/routes/api_platform.yaml

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

This is how API Platform magically adds the routes. It's not very interesting, but see that `type: api_platform`? That basically says:

> Hey! I want API Platform to be able to dynamically add whatever routes it wants.

It does that by finding all the classes marked with `@ApiResource` - just one right now - creating 5 new routes for the 5 operations, and prefixing all the URLs with `/api`. If you want your API URLs to live at the root of the domain, just change this to `prefix: /`.

I hope you're already excited, but there is *so* much more going on than meets the eye! Next, let's talk about the OpenAPI spec: an industry-standard API *description* format that gives your API Swagger superpowers... for free. Yes, we need to talk a *little* bit of theory - but you will *not* regret it.

Chapter 4: OpenAPI Specification

Confession time: this tutorial is about a *lot* more than just API Platform. The world of APIs has undergone *massive* changes over the past few years, introducing new hypermedia formats, standards, specifications, performance tools and more! API Platform lives right in the middle of these: bringing bleeding-edge best practices right into your app. If you *truly* want to master API Platform, you need to understand modern API development.

I told you earlier that what we're looking at is called Swagger. Swagger is basically an API documentation interface - a sort of, interactive README. Google for Swagger and open their site. Under tools, the one *we're* using is called Swagger UI.

Yep!

> Swagger UI allows anyone to visualize and interact with your API's resources without having any of the implementation in place.

Literally, you could first *describe* your API - what endpoints it will have, what it will return, what fields to expect - and then use Swagger UI to visualize your future API, *before* writing even *one* line of code for it.

Let me show you what I mean: they have a live demo that looks *very* similar to our API docs. See that `swagger.json` URL on top? Copy that, open a new tab, and paste. Woh! It's a *huge* JSON file that describes the API! *This* is how Swagger UI works: it reads this JSON file and builds a visual, interactive interface for it. Heck, this API might not even *exist*! As long as you have this JSON description file, you can use Swagger UI.

The JSON file contains all your paths, a description of what each does, the parameters of the input, what output to expect, details related to security... it basically tries to *completely* describe your API.

So *if* you have one of these JSON configuration files, you can plug it into Swagger UI and... boom! You get a rich, descriptive interface.

Hello OpenAPI

The format of this file is called OpenAPI. So, Swagger UI is the interface and it understands this sort of, official spec format for describing APIs called OpenAPI. To make things a bit *more* confusing, the OpenAPI spec *used* to be called Swagger. Starting with OpenAPI 3.0, it's called OpenAPI and Swagger is just the interface.

Phew!

Anyways, this is all really cool... but creating an API is already enough work, without needing to try to build and maintain this gigantic JSON document on the side. Which is why API Platform does it for you.

Remember: API Platform's philosophy is this: create some resources, tweak any configuration you need - we haven't done that, but will soon - and let API Platform *expose* those resources as an API. It does that, but to be an *extra* good friend, it *also* creates an OpenAPI specification. Check it out: go to `/api/docs.json`.

Hello giant OpenAPI spec document! Notice it says `swagger: "2.0"`. OpenAPI version 3 is still pretty new, so API Platform 2 still uses the old format. Add `?spec_version=3` to the URL to see... yep! This is that same document in the latest format version.

Now, go back to our API doc homepage and view the HTML source. Ha! The OpenAPI JSON data is *already* being included on this page via a little `swagger-data` script tag! *That* is how this page is working!

To generate Swagger UI from OpenAPI version 3, you can add the same `?spec_version=3` to the URL. Yep, you can see the `OAS3` tag. That doesn't change a *lot* on the frontend, but there *are* a few new pieces of information that Swagger can now use thanks to the new spec version.

What else Can OpenAPI Do? Code Generation!

But... other than the fact that it gives us this nice Swagger UI, why should we care that there's some giant OpenAPI JSON spec being created behind the scenes? Back on the Swagger site, one of the *other* tools is called Swagger CodeGen: a tool for creating an SDK for your API in almost any language! Think about it: if your API is fully-documented in a machine-understandable language, shouldn't we be able to generate a JavaScript or PHP library that's *customized* for talking with your API? You totally can!

The last thing I want to point out is that, in addition to the endpoints, or "paths", the OpenAPI specification also has information about "models". In the JSON spec, scroll *all* the way to the bottom: it describes our `CheeseListing` model and the fields to expect when sending and receiving this model. You can see this same info in Swagger.

And woh! It somehow *already* knows that the `id` is an `integer` and that it's `readonly`. It also knows price is an `integer` and `createdAt` is a string in a `datetime` format. That's awesome! API Platform *reads* that information directly from our code, which means that our API docs stay up-to-date without us needing to think about it. We'll learn more about how that works along the way.

But before we get there, we need to talk about one other *super* important thing that we're already seeing: the JSON-LD and Hydra format that's being returned by our API responses.

Chapter 5: JSON-LD: Context for your Data

A typical API returns JSON. Go to `/api/cheese_listings/2.json`. When I think of an API, this is what I *traditionally* picture in my head.

Your Data Lacks Meaning

But, what is the *significance* of these fields? What *exactly* do the `title` or `description` fields *mean*? Are they plain text? Can they contain HTML? Does the description describe this *type* of cheese in general, or is this specific to the condition of the *exact* cheese I'm selling? What about price? Is that a string, float, integer? Is it in US Dollars? Euro? Is it measured in cents?

If you're a human... you *are* a human, right? A human can *usually* "infer" some meaning from the field names *or* find some human-readable documentation to help learn *exactly* what each field represents. But, there's no way for a *machine* to understand *anything* about what these fields mean or their types. Even a *smart* algorithm could get confused! A field called `title` could be the "title" of something - like the title of a book - *or* it could be the title of a person - Mr, Mrs, etc.

RDF & HTML

This is what JSON-LD aims to solve. Ok, honestly, there is *a lot* going on these days with this problem of:

> How do we give data on the web *context* or *meaning* that computers can understand?

So let's hit some basic points. There's this thing called RDF - Resource Description Framework - which is a, sort of, set of *rules* about how we can "describe" the *meaning* of data. It's a bit abstract, but it's a guide on how you can say that one piece of data has this "type" or one resource is a "subclass" of some other "type". In HTML, you can add attributes to your elements to add RDF metadata - saying that some `div` describes a *Person* and that this Person's `name` and `telephone` are these other pieces of data:

```
<p typeof="http://schema.org/Person">
  My name is
  <span property="http://schema.org/Person#name">Manu Sporny</span>
  and you can give me a ring via
  <span property="http://schema.org/Person#telephone">1-800-555-0199</span>.
</p>

<!-- or equivalent using vocab -->
<p vocab="http://schema.org/" typeof="Person">
  My name is
  <span property="name">Manu Sporny</span>
  and you can give me a ring via
  <span property="telephone">1-800-555-0199</span>.
</p>
```

This makes your unstructured HTML understandable by machines. It's even *more* understandable if 2 different sites use the *exact* same definition of "Person", which is why the "types" are URLs and sites try to re-use existing types rather than invent new ones.

Kinda cool!

Hello JSON-LD

JSON-LD allows us to do this *same* thing for JSON. Change the URL from `.json` to `.jsonld`. This has the *same* data, but with a few extra fields: `@context`, `@id` and `@type`. JSON-LD is nothing more than a "standard" that describes a few extra fields that your JSON can have - all starting with `@` - that help machines learn more about your API.

JSON-LD: @id

So, first: `@id`. In a RESTful API, *every* URL represents a resource and should have its own unique identifier. JSON-LD makes this official by saying that every resource should have an `@id` field... which *might* seem redundant right now... because... we're *also* outputting our *own* `id` field. But there are two special things about `@id`. First, anyone, or any HTTP client, that understands JSON-LD will know to look for `@id`. It's the official "key" for the unique identifier. Our `id` column is something specific to our API. Second, in JSON-LD, *everything* is done with URLs. Saying the `id` is 2 is cool... but saying the `id` is `/api/cheese_listing/2` is *infinitely* more useful! That's a URL that someone could use to get details about this resource! It's also unique within our entire API... or really... if you include our domain name - it's a unique identifier for that resource across the entire web!

This URL is actually called an IRI: Internationalized Resource Identifier. We're going to use IRI's everywhere instead of integer ids.

JSON-LD @context and @type

The other two JSON-LD keys - `@context` and `@type` work together. The idea is *really* cool: if we add an `@type` key to *every* resource and then define the exact *fields* of that type somewhere, that gives us two superpowers. First, we instantly know if two different JSON structures are in fact *both* describing a cheese listing... or if they just *look* similar and are actually describing different things. And second, we can look at the definition of this type to learn more about it: what properties it has and even the type of each property.

Heck, this is nothing new! We do this *all* the time in PHP! When we create a *class* instead of just an array, we are giving our data a "type". It allows us to know *exactly* what type of data we're dealing with *and* we can look at the class to learn more about its properties. So... yea - the `@type` field sorta transforms this data from a structureless array into a concrete class that we can understand!

But... *where* is this `CheeseListing` type defined? That's where `@context` comes in: it basically says:

> Hey! To get more details, or "context" about the fields used in this data, go to this other URL.

For this to make sense, we need to think like a machine: a machine that *desperately* wants to learn as much as possible about our API, its fields and what they mean. When a machine sees that `@context`, it follows it. Yea, let's *literally* put that URL in the browser: `/api/contexts/CheeseListing`. And... interesting. It's another `@context`. Without going into *too* much crazy detail, `@context` allows us to use "shortcut" property names - called "terms". Our actual JSON response includes fields like `title` and `description`. But as far as JSON-LD is concerned, when you take the `@context` into account, it's as *if* the response looks something like this:

```
{
    "@context": {
        "@vocab": "https://localhost:8000/api/docs.jsonld#"
    },
    "@id": "/api/cheese_listing/2",
    "@type": "CheeseListing",
    "CheeseListing/title": "Giant block of cheddar cheese",
    "CheeseListing/description": "mmmmmm",
    "CheeseListing/price": 1000,
}
```

The idea is that we know that, in general, *this* resource is a `CheeseListing` type, and when we find its docs, we should find information also about the meaning and types of the `CheeseListing/title` or `CheeseListing/price` properties. Where does that documentation live? Follow the `@vocab` link to `/api/docs.jsonld`.

This is a *full* description of our API in JSON-LD. And, check it out. It has a section called `supportedClasses`, with a `CheeseListing` class and all the different properties below it! *This* is how a machine can understand what the `CheeseListing/title` property means: it has a label, details on whether or not it's required, whether or not it's readable and whether or not it's writeable. For `CheeseListing/price`, it already knows that this is an integer.

This is *powerful* information for a machine! And if you're thinking:

> Wait a second! Isn't this exactly the same info that the OpenAPI spec gave us?

Well, you're not wrong. But more on that in a little while.

Anyways, the *really* cool thing is that API Platform is getting all of the data about our class and its properties from *our* code! For example, look at the `CheeseListing/price` property: it has a title, type of `xmls:integer` and some data.

By the way, even that `xmls:integer` type comes from *another* document. I didn't show it, but at the top of this page, we're referencing *another* document that defines *more* types, including what the `xmls:integer` "type" means in a machine-readable format.

Anyways, back in our code, above price, add some phpdoc:

> The price of this delicious cheese in cents.

Refresh our JSON-LD document now. Boom! Suddenly we have a `hydra:description` field! We're going to talk about what "Hydra" is next.

How this Looks to a Machine

I know, I know, this is all a bit confusing, well, it is for *me* at least. But, try to picture what this looks like to a *machine*. Go back to the original JSON: it said `@type: "CheeseListing"`. By "following" the `@context` URL, then following `@vocab` - *almost* the same way that we follow links inside a browser - we can eventually find details about what that "type" actually means! And by referencing external documents under `@context`, we can, sort of, "import" more types. When a machine sees `xmls:integer`, it knows it can follow this `xmls` link to find out more about that type. And if *all* APIs used this same identifier for integer types, well, suddenly, APIs would become *super* understandable by machines.

Anyways, you don't need to be able to read these documents and make perfect sense of them. As long as you understand what all of this "linked data" and shared "types" are trying to accomplish, you're good.

Ok, we're *almost* done with all this theoretical stuff - I *promise*. But first, we need talk about what "Hydra" is, and see a few other cool entries that are already under `hydra:supportedClass`.

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

So, at least on a high level, we understand that each resource will have an `@type` key and that *this* page - via the `supportedClass` and `supportedProperty` keys - *defines* what that type means - what properties it has, and a lot of info about each property.

Right now, we only have one API resource, so we only have one entry under `supportedClass`, right? Surprise! There's another one called `Entrypoint`! And *another* one called `ConstraintViolation`, which defines the *resource* that's returned when our API has a validation error.

The Entrypoint Resource

Let's talk about this `Entrypoint` class: it's a pretty beautiful idea. We already know that when we go to `/api`, we get, sort of, the HTML version of an API "homepage". Whelp, there is *also* a JSON-LD version of this page! There's a link to see it at the bottom of this page - but let's get to it a different way.

Find your terminal: we can use curl to see what the "homepage" looks like for the JSON-LD format:

```
$ curl -X GET 'https://localhost:8000/api' -H "accept: application/ld+json"
```

In other words: please make a GET request to `/api`, but advertise that you would like the JSON-LD format back. I'll also pipe that to `jq` - a utility that makes JSON look pretty - just skip that if you don't have it installed. And... boom!

Say hello to your API's homepage! Because *every* URL represents a unique resource, even *this* page is a resource: it's... an "Entrypoint" resource. It has the same `@context`, `@id` and `@type`, with one real "property" called `cheeseListing`. That property is the IRI of the cheese listing collection resource.

Heck, this is *described* in our JSON-LD document! The `Entrypoint` class has one property: `cheeseListing` with the type `hydra:Link` - that's interesting. And, it's pretty ugly, but the `rdfs:range` part is apparently a way to describe that the resource this property refers to is a "collection" that will have a `hydra:member` property, which will be an array where each item is a `CheeseListing` type. Woh!

Hello Hydra

So JSON-LD is all about adding more context to your data by specifying that our resources will contain special keys like `@context`, `@id` and `@type`. It's still normal JSON, but if a client understands JSON-LD, it's going to be able to get a *lot* more information about your API, automatically.

But in API Platform, there is one other thing that you're going to see *all* the time, and we're already seeing it! Hydra, which is *more* than just a many-headed water monster from Greek mythology.

Go back to `/api/docs.jsonld`. In the same way that this points to the `xmls` external document so that we can reference things like `xmls:integer`, we're *also* pointing to an external document called `hydra` that defines more "types" or "vocab" we can use.

Here's the idea: JSON-LD gave us the *system* for saying that this piece of data is this type and this piece of data is this other type. Hydra is an *extension* of JSON-LD that adds new *vocab*. It says:

> Hold on a second. JSON-LD is great and fun and an excellent dinner party guest. But to *really* allow a client and a server to communicate, we need *more* shared language! We need a way to define "classes" within my API, the properties of those classes and whether or not each is readable and writeable. Oh, and we also need to be able to communicate the *operations* that a resource supports: can I make a DELETE request to this resource to remove it? Can I make a PUT request to update it? What data format should I expect back from each operation? And what is the true identity of Batman?

Hydra took the JSON-LD system and added new "terminology" - called "vocab" - that makes it possible to *fully* define *every* aspect of your API.

Hydra Versus OpenAPI

At this point, you're almost *definitely* thinking:

> But wait, this *seriously* sounds like the *exact* same thing that we got from our OpenAPI JSON doc.

And, um... yea! Change the URL to `/api/docs.json`. This is the OpenAPI specification. And if we change that to `.jsonld`, suddenly we have the JSON-LD specification with Hydra.

So why do we have both? First, yes, these two documents basically do the same thing: they describe your API in a machine-readable format. The JSON-LD and Hydra format is a bit more powerful than OpenAPI: it's able to describe a few things that OpenAPI can't. But OpenAPI is more common and has more tools built around it. So, in some cases, having an OpenAPI specification will be useful - like to use Swagger - and other times, the JSON-LD Hydra document will be useful. With API Platform, you get both.

Phew! Ok, enough theory! Let's get back to building and customizing our API.

Chapter 7: API Debugging with the Profiler

Debugging an API... can be tough... because you don't see the results - or errors - as big HTML pages. So, to help us along the way, let's level-up our debugging ability! In a traditional Symfony app, one of the *best* features is the web debug toolbar... which we don't see down here right now because it's not installed yet.

Using the Profiler in an API

But... should we even bother? I mean, it's not like we can see the web debug toolbar on a JSON API response, right? Of course we can! Well, sort of.

Find your terminal and get the profiler installed with:

```
$ composer require profiler --dev
```

You can also run `composer require debug --dev` to install a few extra tools. This installs the `WebProfilerBundle`, which adds a couple of configuration files to help it do its magic.

Thanks to these, when we refresh... there it is! The web debug toolbar floating on the bottom. This is *literally* the web debug toolbar for this *documentation* page... which probably isn't that interesting.

But if we start making requests... check it out. When we execute an operation via Swagger, it makes an AJAX request to complete the operation. And Symfony's web debug toolbar has a cool little feature where it *tracks* those AJAX requests and adds them to a list! Every time I hit execute, I get a new one!

The *real* magic is that you can click the little "sha" link to see the profiler for that API request! So... yea! You can't see the web debug toolbar for a response that returns JSON, but you *can* still see the *profiler*, which contains *way* more data anyways, like the POST parameters, the request headers, request content - which is really important when you're sending JSON - and all the goodies that you expect - cache, performance, security, Doctrine, etc.

Finding the Profile for an API Request

In addition to the little web debug toolbar AJAX tracker we just saw, there are a few *other* ways to find the profiler for a specific API request. First, every response has an `x-debug-token-link` header with a URL to its profiler page, which you can read to figure out where to go. Or, you can just go to `/_profiler` to see a list of the most recent request. Here's the one for `/api/cheese_listings`. Click the token to jump into its profiler.

The API Platform Panel

Oh, and API Platform adds its *own* profiler panel, which is a nice way to see which *resource* this request was operating on and the metadata for it, including this item operation and collection operation stuff - we'll talk about those really soon. It also shows info about "data providers" and "data persisters" - two important concepts we'll talk about later.

But before we get there, back on the documentation page, we need to talk about these five endpoints - called *operations* - and how we can customize them.

Chapter 8: Operations

Let's get to work customizing our API. A RESTful API is all about *resources*. We have *one* resource - our `CheeseListing` - and, by default, API Platform generated 5 endpoints for it. These are called "operations".

Collection and Item Operations

Operations are divided into two categories. First, "collection" operations. These are the URLs that don't include `{id}` and where the "resource" you're operating on is *technically* the "collection of cheese listings". For example, you're "getting" the collection or you're "adding" to the collection with POST.

And second - "item" operations. These are the URLs that *do* have the `{id}` part, when you're "operating" on a *single* cheese listing resource.

The *first* thing we can customize is which operations we actually want! Above `CheeseListing`, inside the annotation, add `collectionOperations={}` with `"get"` and `"post"` inside. Then `itemOperations` with `{"get", "put", "delete"}`.

```
116 lines | src/Entity/CheeseListing.php
... lines 1 - 7
/**
 * @ApiResource(
 *     collectionOperations={"get", "post"},
 *     itemOperations={"get", "put", "delete"}
 * )
... line 13
 */
class CheeseListing
... lines 16 - 116
```

A lot of mastering API Platform comes down to learning about what options you can pass inside this annotation. *This* is basically the default configuration: we want *all* five operations. So not surprisingly, when we refresh, we see absolutely no changes. But what if we don't want to allow users to delete a cheese listing? Maybe instead, in the future, we'll add a way to "archive" them. Remove `"delete"`.

```
116 lines | src/Entity/CheeseListing.php
... lines 1 - 7
/**
 * @ApiResource(
... line 10
 *     itemOperations={"get", "put"}
 * )
... line 13
 */
... lines 15 - 116
```

As *soon* as we do that... boom! It's gone from our documentation. Simple, right? Yep! But a bunch of cool things just happened. Remember that, behind the scenes, the Swagger UI is built off of an Open API spec document, which you can see at `/api/docs.json`. The reason the "delete" endpoint disappeared from Swagger is that it disappeared from here. API Platform is keeping our "spec" document up to date. If you looked at the JSON-LD spec doc, you'd see the same thing.

And of course, it also completely removed the endpoint - you can see that by running:

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

Yep, just `GET`, `POST`, `GET` and `PUT`.

Customizing the Resource URL (shortName)

Hmm, now that I'm looking at this, I don't love the `cheese_listings` part of the URLs... API Platform generates this from the class name. And really, in an API, you shouldn't obsess about how your URLs look - it's just not important, especially - as you'll see - when your API responses include links to other resources. But... we *can* control this.

Flip back over and add another option: `shortName` set to `cheeses`.

```
117 lines | src/Entity/CheeseListing.php
... lines 1 - 7
/**
 * @ApiResource(
... lines 10 - 11
 *     shortName="cheeses"
... lines 13 - 14
 */
... lines 16 - 117
```

Now run `debug:router` again:

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

Hey! `/api/cheeses`! Much better! And we see the same thing now on our API docs.

Customizing Operation Route Details

Ok: so we can control *which* operations we want on a resource. And later, we'll learn how to add *custom* operations. But we can *also* control quite a lot about the individual operations.

We know that each operation generates a route, and API Platform gives you full control over how that route looks. Check it out: break `itemOperations` onto multiple lines. Then, instead of just saying `"get"`, we can say `"get"={}` and pass this extra configuration.

Try `"path"=` set to, I don't know, `"/i❤ cheeses/{id}"`.

```
120 lines | src/Entity/CheeseListing.php
... lines 1 - 7
/**
 * @ApiResource(
... line 10
 *     itemOperations={
 *         "get"={"path"="/i❤ cheeses/{id}"},
 *         "put"
 *     },
... line 15
 * )
... line 17
 */
... lines 19 - 120
```

Go check out the docs! Ha! That works! What else can you put here? Quite a lot! To start, *anything* that can be defined on a route, can be added here - like `method`, `hosts`, etc.

What else? Well, along the way, we'll learn about other, API-Platform-specific stuff that you can put here, like `access_control` for security and ways to control the serialization process.

In fact, let's learn about that process right now! How does API Platform transform our `CheeseListing` object - with all these private properties - into the JSON that we've been seeing? And when we create a *new* `CheeseListing`, how is it converting our *input* JSON into a `CheeseListing` object?

Understanding the serialization process may be *the* most important piece to unlocking API Platform.

Chapter 9: The Serializer

Google for Symfony serializer and find a page called The Serializer Component.

API Platform is built on top of the Symfony components. And the *entire* process of how it turns our `CheeseListing` object into JSON... and JSON back into a `CheeseListing` object, is done by Symfony's Serializer! If we understand how *it* works, we're in business!

And, at least on the surface, it's beautifully simple. Check out the diagram that shows how it works. Going from an object to JSON is called serialization, and from JSON back into an object called deserialization. To do that, internally, it goes through a process called normalizing: it *first* takes your object and turns it into an array. And *then* it's encoded into JSON or whatever format.

How Objects are Turned into Raw Data

There are actually a *bunch* of different "normalizer" classes that help with this job - like one that's really good at converting `DateTime` objects to a string and back. But the *main* class - the one at the *heart* of this process - is called the `ObjectNormalizer`. Behind the scenes, *it* uses another Symfony component called `PropertyAccess`, which has one superpower: if you give it a property name, like `title`, it's really good at finding and using getter and setter methods to access that property.

In other words, when API platform tries to "normalize" an object into an array, it uses the getter and setter methods to do that!

For example, it sees that there's a `getId()` method, and so, it turns that into an `id` key on the array... and eventually in the JSON. It does the same thing for `getTitle()` - that becomes `title`. It's just that simple!

When we *send* data, it does the same thing! Because we have a `setTitle()` method, we can send JSON with a `title` key. The normalizer will take the value we're sending, call `setTitle()` and pass it!

It's a simple, but neat way to allow your API clients to interact with your object, your API resource, using its getter and setter methods. By the way, the PropertyAccess component also supports public properties, hassers, issers, adders, removers - basically a bunch of common method naming conventions in addition to getters and setters.

Adding a Custom "Field"

Anyways, now that we know how this works, we're *super* dangerous! Seriously! Right now, we're able to send a `description` field. Let's pretend that this property can contain HTML in the database. But most of our users don't really understand HTML and, instead, just type into a box with line breaks. Let's create a new, *custom* field called `textDescription`. If an API client sends a `textDescription` field, we'll convert the new lines into HTML breaks before saving it on the `description` property.

How can we create a totally new, custom input field for our resource? Find `setDescription()`, duplicate it, and name it `setTextDescription()`. Inside, say, `$this->description = nl2br($description);`. It's a silly example, but even forgetting about API Platform, this is good, boring, object-oriented coding: we've added a way to set the description *if* you want new lines to be converted to line breaks.

```
127 lines | src/Entity/CheeseListing.php
... lines 1 - 18
class CheeseListing
{
... lines 21 - 83
    public function setTextDescription(string $description): self
    {
        $this->description = nl2br($description);

        return $this;
    }
... lines 90 - 125
}
```

But *now*, refresh, and open up the POST operation again. Woh! It says that we can *still* send a `description` field, but we can *also* pass `textDescription`! But if your try the GET operation... we still *only* get back `description`.

That makes sense! We added a *setter* method - which makes it possible to *send* this field - but we did *not* add a *getter* method. You can also see the new field described down in the models section.

Removing "description" as Input

But, we *probably* don't want to allow the user to send both `description` *and* `textDescription`. I mean, you *could*, but it's a little weird - if the client sent both, they would bump into each other and the last key would win because its setter method would be called last. So, let's remove `setDescription()`.

Refresh now. I love it! To create or update a cheese listing, the client will send `textDescription`. But when they fetch the data, they'll always get back `description`. In fact, let's try it... with id 1. Open the PUT operation and set `textDescription` to something with a few line breaks. I *only* want to update this *one* field, so we can just remove the other fields. And... execute! 200 status code and... a `description` field with some line breaks!

By the way, the fact that our input fields don't match our output fields is *totally* ok. Consistency *is* super nice - and I'll show you soon how we can fix this inconsistency. But there's no rule that says your input data needs to match your output data.

Removing createdAt Input

Ok, what else can we do? Well, having a `createdAt` field on the output is great, but it probably doesn't make sense to allow the client to send this: the server should set it automatically.

No problem! Don't want the `createdAt` field to be allowed in the input? Find the `setCreatedAt()` method and remove it. To auto-set it, it's back to good, old-fashioned object-oriented programming. Add `public function __construct()` and, inside, `$this->createdAt = new \DateTimeImmutable()`.

118 lines | src/Entity/CheeseListing.php

```
... lines 1 - 54
    public function __construct()
    {
        $this->createdAt = new \DateTimeImmutable();
    }
... lines 59 - 118
```

Go refresh the docs. Yep, it's gone here... but when we try the GET operation, it *is* still in the output.

Adding a Custom Date Field

We're on a roll! So let's customize one more thing! Let's say that, in addition to the `createdAt` field - which is in this ugly, but standard format - we *also* want to return the date as a string - something like `5 minutes ago` or `1 month ago`.

To help us do that, find your terminal and run:

```
$ composer require nesbot/carbon
```

This is a handy DateTime utility that can easily give us that string. Oh, while this is installing, I'll go back to the top of my entity and remove the custom `path` on the `get` operation. That's a cool example... but let's *not* make our API weird for no reason.

```
118 lines | src/Entity/CheeseListing.php
... lines 1 - 7
/**
 * @ApiResource(
... line 10
 *     itemOperations={
 *          "get"={},
... line 13
 *     },
... line 15
 * )
... line 17
 */
... lines 19 - 118
```

Yep, that looks better.

Back at the terminal.... done! In `CheeseListing`, find `getCreatedAt()`, go below it, and add `public function getCreatedAtAgo()` with a `string` return type. Then, `return Carbon::instance($this->getCreatedAt())->diffForHumans()`.

```
124 lines | src/Entity/CheeseListing.php
... lines 1 - 106
    public function getCreatedAtAgo(): string
    {
        return Carbon::instance($this->getCreatedAt())->diffForHumans();
    }
... lines 111 - 124
```

You know the drill: *just* by adding a getter, when we refresh... and look at the model, we have a *new* `createdAtAgo` - *readonly* field! And, by the way, it *also* knows that `description` is readonly because it has no setter.

Scroll up and try the GET collection operation. And... cool: `createdAt` *and* `createdAtAgo`.

As *nice* as it is to control things by simply tweaking your getter and setter methods, it's not *ideal*. For example, to prevent an API client from setting the `createdAt` field, we *had* to remove the `setCreatedAt()` method. But, what if, *somewhere* in my app - like a command that imports legacy cheese listings - we *do* need to manually set the `createdAt` date? Let's learn how to control this with serialization groups.

Chapter 10: Serialization Groups

If the only way to control the input and output of our API was by controlling the getters and setters on our entity, it wouldn't be that flexible... and *could* be a bit dangerous. You might add a new getter or setter method for something internal and not realize that you were exposing new data in your API!

The solution for this - and the way that I recommend doing things in all cases - is to use serialization groups.

Adding a Group for Normalization

In the annotation, add `normalizationContext`. Remember, normalization is when you're going from your object to an array. So this option is related to when you are *reading* data from your API. Context is basically "options" that you pass to that process. The most common option by far is called `"groups"`, which you set to another array. Add one string here: `cheese_listing:read`.

```
125 lines | src/Entity/CheeseListing.php
... lines 1 - 8
/**
 * @ApiResource(
... lines 11 - 16
 *     normalizationContext={"groups"={"cheese_listing:read"}}
 * )
... line 19
 */
class CheeseListing
... lines 22 - 125
```

Thanks to this, when an object is being serialized, the serializer will *only* include fields that are in this `cheese_listing:read` group, because, in a second, we're going to start adding groups to each property.

But right now, we haven't added *any* groups to *anything*. And so, if you go over and try your get collection operation... oh! Ah! A huge error!

Debugging Errors

Let's... pretend like I did that on purpose and see how to debug it! The problem is that the giant HTML error is... a bit hard to read. One way to see the error is to use our trick from earlier: go to `https://localhost:8000/_profiler/`.

Woh! Ok, there are two types of errors: runtime errors, where something went wrong *specific* to that request, and build errors, where some invalid configuration is killing *every* page. Most of the time, if you see an exception, there is *still* a profiler you can find for that request by using the trick of going to this URL, finding that request in the list - usually right on top - and clicking the sha into its profiler. Once you're there, you can click an "Exception" tab on the left to see the big, beautiful normal exception.

If you get a build error that kills *every* page, it's even easier: you'll see it when trying to access anything.

Anyways, the problem here is with my annotation syntax. I do this a lot - which is no big deal as long as you know how to debug the error. And, yep! I forgot a comma at the end.

Adding Groups to Fields

Refresh again! The profiler works, so now we can go back over and hit execute again. Check it out - we have `@id` and `@type` from JSON-LD... but it doesn't contain *any* real fields because *none* of them are in the new `cheese_listing:read` group!

Copy the `cheese_listing:read` group name. To *add* fields to this, above title, use `@Groups()`, `{""}` and paste. Let's also put that above `description`... and `price`.

```
129 lines | src/Entity/CheeseListing.php
... lines 1 - 7
use Symfony\Component\Serializer\Annotation\Groups;
... lines 9 - 21
class CheeseListing
{
... lines 24 - 30
    /**
... line 32
     * @Groups({"cheese_listing:read"})
     */
    private $title;
... line 36
    /**
... line 38
     * @Groups({"cheese_listing:read"})
     */
    private $description;
... line 42
    /**
... lines 44 - 46
     * @Groups({"cheese_listing:read"})
     */
    private $price;
... lines 50 - 127
}
```

Flip back over and try it again. Beautiful! We get those *three* exact fields. I *love* this control.

By the way - the name `cheese_listing:read`... I just made that up - you could use anything. *But*, I *will* be following a group naming convention that I recommend. It'll give you flexibility, but keep things organized.

Adding Denormalization Groups

Now we can do the same thing with the input data. Copy `normalizationContext`, paste, and add `de` in front to make `denormalizationContext`. This time, use the group: `cheese_listing:write`

```
130 lines | src/Entity/CheeseListing.php
... lines 1 - 9
/**
 * @ApiResource(
... lines 12 - 18
 *     denormalizationContext={"groups"={"cheese_listing:write"}}
 * )
... line 21
 */
... lines 23 - 130
```

Copy that and... let's see... just add this to `title` and `price` for now. We actually *don't* want to add it to `description`. Instead, we'll talk about how to add this group to the fake `textDescription` in a minute.

```
130 lines | src/Entity/CheeseListing.php
... lines 1 - 22
class CheeseListing
{
... lines 25 - 31
    /**
... line 33
     * @Groups({"cheese_listing:read", "cheese_listing:write"})
     */
    private $title;
... lines 37 - 43
    /**
... lines 45 - 47
     * @Groups({"cheese_listing:read", "cheese_listing:write"})
     */
    private $price;
... lines 51 - 128
}
```

Move over and refresh again. Open up the POST endpoint.... yea - the *only* fields we can pass now are `title` and `price`!

So `normalizationContext` and `denormalizationContext` are two totally separate configs for the two directions: *reading* our data - normalization - and *writing* our data - denormalization.

The Open API Read & Write Models

At the bottom of the docs, you'll *also* notice that we now have two models: the *read* model - that's the normalization context with `title`, `description` and `price`, and the write model with `title` and `price`.

And, it's not really important, but you can control these names if you want. Add another option: `swagger_definition_name` set to "Read". And then the same thing below... set to Write.

```
130 lines | src/Entity/CheeseListing.php
... lines 1 - 9
/**
 * @ApiResource(
... lines 12 - 17
 *     normalizationContext={"groups"={"cheese_listing:read"}, "swagger_definition_name"="Read"},
 *     denormalizationContext={"groups"={"cheese_listing:write"}, "swagger_definition_name"="Write"}
 * )
... line 21
 */
... lines 23 - 130
```

I don't normally care about this, but if you want to control it, you can.

Adding Groups to Fake Fields

But, we're missing some fields! When we *read* the data, we get back `title`, `description` and `price`. But what about our `createdAt` field or our custom `createdAtAgo` field?

Let's pretend that we *only* want to expose `createdAtAgo`. No problem! Just add the `@Groups` annotation to that property... oh wait... there *is* no `createdAtAgo` property. Ah, it's just as easy: find the *getter* and put the annotation there: `@Groups({"cheese_listing:read"})`. And while we're here, I'll add some documentation to that method:

> How long ago in text that this cheese listing was added.

```
135 lines | src/Entity/CheeseListing.php
... lines 1 - 22
class CheeseListing
{
... lines 25 - 112
    /**
     * How long ago in text that this cheese listing was added.
     *
     * @Groups("cheese_listing:read")
     */
    public function getCreatedAtAgo(): string
... lines 119 - 133
}
```

Let's try it! Refresh the docs. Down in the models section... nice! There's our new `createdAtAgo` readonly field. *And* that documentation we added shows up here. Nice! No surprise that when we try it... the field shows up.

For denormalization - for *sending* data - we need to re-add our fake `textDescription` field. Search for the `setTextDescription()` method. To prevent API clients from sending us the `description` field directly, we removed the `setDescription()` method. Above `setTextDescription()`, add `@Groups({"cheese_listing:write"})`. And again, let's give this some extra docs.

```
140 lines | src/Entity/CheeseListing.php
... lines 1 - 88
    /**
     * The description of the cheese as raw text.
     *
     * @Groups("cheese_listing:write")
     */
    public function setTextDescription(string $description): self
... lines 95 - 140
```

This time, when we refresh the docs, you can see it on the write model and, of course, on the data that we can send to the POST operation.

Have Whatever Getters and Setters You Want

And... this leads us to some great news! *If* we decide that something internally in our app *does* need to set the description property directly, it's now perfectly ok to re-add the original `setDescription()` method. That won't become part of our API.

```
147 lines | src/Entity/CheeseListing.php
... lines 1 - 88
    public function setDescription(string $description): self
    {
        $this->description = $description;

        return $this;
    }
... lines 95 - 147
```

Default isPublished Value

Let's try *all* of this out. Refresh the docs page. Let's create a new listing: Delicious chèvre - excuse my French - for $25 and a description with some line breaks. Execute!

Woh! A 500 error! I could go look at this exception in the profiler, but this one is pretty easy to read: an exception in our query: `is_published` cannot be null. Oh, that makes sense: the user isn't sending `is_published`... so *nobody* is setting it. And it's set to not null in the database. No worries: default the property to `false`.

```
147 lines | src/Entity/CheeseListing.php
... lines 1 - 22
class CheeseListing
{
... lines 25 - 59
    private $isPublished = false;
... lines 61 - 145
}
```

Tip

Actually, the auto-validation was not enabled by default in Symfony 4.3, but may be in Symfony 4.4.

By the way, if you're using Symfony 4.3, instead of a Doctrine error, you may have gotten a validation error. That's due to a new feature where Doctrine database constraints can automatically be used to add validation. So, if you see a validation error, awesome!

Anyways, try to execute it again. It works! We have *exactly* the input fields and output fields that we want. The `isPublished` field isn't exposed at *all* in our API, but it *is* being set in the background.

Next, let's learn a few more serialization tricks - like how to control the field name and how to handle constructor arguments.

Chapter 11: @SerializedName & Constructor Args

When we *read* a `CheeseListing` resource, we get a `description` field. But when we *send* data, it's called `textDescription`. And... that's technically fine: our input fields don't need to match our output fields. *But*... if we *could* make these the same, that might make life easier for anyone using our API.

It's pretty easy to guess how these properties are created: the keys inside the JSON literally match the names of the properties inside our class. And in the case of a fake property like `textDescription`, API Platform strips off the "set" part and makes it lower camel case. By the way, like everything in API Platform, the way fields are transformed into keys *is* something you can control at a global level: it's called a "name converter".

Controlling Field Names: @SerializedName

Anyways, it would be kinda nice if the input field were just called `description`. We'd have input `description`, output `description`. Sure, internally, *we* would know `setTextDescription()` was called on input and `getDescription()` on output, but the user wouldn't need to care or worry about that.

And... yes! You can *totally* control this with a *super* useful annotation. Above `setTextDescription()`, add `@SerializedName()` with `description`.

```
149 lines | src/Entity/CheeseListing.php
... lines 1 - 23
class CheeseListing
... lines 25 - 96
    /**
... lines 98 - 100
     * @SerializedName("description")
     */
    public function setTextDescription(string $description): self
... lines 104 - 147
}
```

Refresh the docs! If we try the GET operation... that hasn't changed: still `description`. But for the POST operation... yes! The field is *now* called `description`, but the serializer will call `setTextDescription()` internally.

What about Constructor Arguments

Ok, so we know that the serializer likes to work by calling getter and setter methods... or by using public properties or a few other things like hasser or isser methods. But what if I want to give my class a constructor? Well, right now we *do* have a constructor, but it doesn't have any required arguments. That means that the serializer has *no* problems instantiating this class when we POST a new `CheeseListing`.

But... yea know what? Because *every* `CheeseListing` needs a title, I'd like to give this a new required argument called `$title`. You definitely don't need to do this, but for a lot of people, it makes sense: if a class has required properties: force them to be passed in via the constructor!

And now that we have *this*, you might *also* decide that you don't want to have a `setTitle()` method anymore! From an object-oriented perspective, this makes the `title` property immutable: you can only set it *once* when *creating* the `CheeseListing`. It's kind of a silly example. In the real world, we probably *would* want the title to be changeable. But, from an object-oriented perspective, there *are* situations when you want to do exactly this.

Oh, and don't forget to say `$this->title = $title` in the constructor.

```
143 lines | src/Entity/CheeseListing.php
... lines 1 - 23
class CheeseListing
{
... lines 26 - 62
    public function __construct(string $title)
    {
        $this->title = $title;
... line 66
    }
... lines 68 - 141
}
```

The question *now* is... will the serializer be able to work with this? Is it going to be super angry that we removed `setTitle()`? And when we POST to add a new one, will it be able to instantiate the `CheeseListing` even though it has a required arg?

Whelp! Let's try it! How about crumbs of some blue cheese... for $5. Execute and... it worked! The title is correct!

Um... how the heck did that work? Because the *only* way to set the title is via the constructor, it *apparently* knew to pass the title key there? How?

The answer is... magic! I'm kidding! The answer is... by complete luck! No, I'm still totally lying. The answer is because of the argument's *name*.

Check this out: change the argument to `$name`, and update the code below. From an object-oriented perspective, that shouldn't change anything. But hit execute again.

```
143 lines | src/Entity/CheeseListing.php
... lines 1 - 62
    public function __construct(string $name)
    {
        $this->title = $name;
... line 66
    }
... lines 68 - 143
```

Huge error! A 400 status code:

> Cannot create an instance of `CheeseListing` from serialized data because its constructor requires parameter "name" to be present.

My compliments to the creator of that error message - it's awesome! When the serializer sees a constructor argument named... `$name`, it looks for a `name` key in the JSON that we're sending. If that doesn't exist, boom! Error!

So as *long* as we call the argument `$title`, it all works nicely.

Constructor Argument can Change Validation Errors

But there *is* one edge case. Pretend that we're creating a new `CheeseListing` and we forget to send the `title` field entirely - like, we have a bug in our JavaScript code. Hit Execute.

We *do* get back a 400 error... which is perfect: it means that the person making the request has something wrong with their request. But, the `hydra:title` isn't very clear:

> An error occurred

Fascinating! The `hydra:description` is *way* more descriptive... actually a bit *too* descriptive - it shows off some internal things about our API... that I maybe don't want to make public. At least the `trace` won't show up on production.

Showing these details inside `hydra:description` *might* be ok with you... But if you want to avoid this, you need to rely on validation, which is a topic that we'll talk about in a few minutes. *But*, what you need to know *now* is that validation can't happen unless the serializer is able to successfully create the `CheeseListing` object. In other words, you need to help the serializer out by making this argument optional.

```
143 lines | src/Entity/CheeseListing.php
... lines 1 - 62
    public function __construct(string $title = null)
    {
... lines 65 - 66
    }
... lines 68 - 143
```

If you try this again... ha! A 500 error! It *does* create the `CheeseListing` object successfully... then explodes when it tries to add a null title in the database. But, that's *exactly* what we want - because it will allow validation to do its work... once we add that in a few minutes.

Tip

Actually, the auto-validation was not enabled by default in Symfony 4.3, but may be in Symfony 4.4.

Oh, and if you're using Symfony 4.3, you may *already* see a validation error! That's because of a new feature that can automatically convert your database rules - the fact that we've told Doctrine that `title` is required in the database - into validation rules. Fun fact, this feature was contributed to Symfony by Kèvin Dunglas - the lead developer of API Platform. Sheesh Kèvin! Take a break once in awhile!

Next: let's explore *filters*: a powerful system for allowing your API clients to search and filter through our CheeseListing resources.

Chapter 12: Filtering & Searching

We're doing awesome! We understand how to expose a class as an API resource, we can choose which operations we want, and have full control over the input and output fields, including some "fake" fields like `textDescription`. There's a lot more to know, but we're doing great!

So what else does *every* API need? I can think of a few things, like pagination and validation. We'll talk about both of those soon. But what about filtering? Your API client - which might just be *your* JavaScript code, isn't going to *always* want to fetch *every* single `CheeseListing` in the system. What if you need the ability to only see published listings? Or what if you have a search on the front-end and need to find by title? These are called "filters": ways to see a "subset" of a collection based on some criteria. And API Platform comes with a *bunch* of them built-in!

Filtering by Published/Unpublished

Let's start by making it possible to *only* return *published* cheese listings. Well, in a future tutorial, we're going to make it possible to *automatically* hide unpublished listings from the collection. But, for now, our cheese listing collection returns *everything*. So let's *at least* make it possible for an API client to *ask* for only the published ones.

At the top of `CheeseListing`, activate our *first* filter with `@ApiFilter()`. Then, choose the *specific* filter by its class name: `BooleanFilter::class` ... because we're filtering on a *boolean* property. Finish by passing the `properties={}` option set to `"isPublished"`.

```
145 lines | src/Entity/CheeseListing.php
... lines 1 - 11
/**
... lines 13 - 22
 * @ApiFilter(BooleanFilter::class, properties={"isPublished"})
... line 24
 */
class CheeseListing
... lines 27 - 145
```

Cool! Let's see what this did! Refresh! Oh... what it *did* was break our app!

> The filter class `BooleanFilter` does not implement `FilterInterface`.

It's not *super* clear, but that error means that we forgot a `use` statement. This `BooleanFilter::class` is referencing a specific *class* and we need a `use` statement for it. It's... kind of a strange way to use class names, which is why PhpStorm didn't autocomplete it for us.

No problem, at the top of your class, add `use BooleanFilter`. But... careful... most filters support Doctrine ORM *and* Doctrine with MongoDB. Make sure to choose the class for the ORM.

```
146 lines | src/Entity/CheeseListing.php
... lines 1 - 6
use ApiPlatform\Core\Bridge\Doctrine\Orm\Filter\BooleanFilter;
... lines 8 - 146
```

Ok, *now* move over and refresh again.

We're back to life! Click "Try it out". Hey! We have a little `isPublished` filter input box! If we leave that blank and execute... looks like 4 results.

Choose `true` for `isPublished` and try it again. We're down to two results! And check out how this works with the URL: it's still `/api/cheeses`, but with a *gorgeous* `?isPublished=true` or `?isPublished=false`. So *just* like that, our API users can filter a collection on a boolean field.

Oh! Also, down in the response, there's a new `hydra:search` property. OoooOOO. It's a bit techy, but this is *explaining* that you can now search using an `isPublished` query parameter. It also gives information about which property this relates to on the resource.

Text Searching: SearchFilter

How else can we filter? What about searching by text? On top of the class, add another filter: `@ApiFilter()`. This one is called `SearchFilter::class` and has the same `properties` option... but with a bit more config. Say `title` set to `partial`. There are also settings to match on an `exact` string, the `start` of a string, `end` or a string or on `word_start`.

Anyways, *this* time, I remember that we need to add the `use` statement manually. Say `use SearchFilter` and auto-complete the one for the ORM.

```
148 lines | src/Entity/CheeseListing.php
... lines 1 - 7
use ApiPlatform\Core\Bridge\Doctrine\Orm\Filter\SearchFilter;
... lines 9 - 13
/**
... lines 15 - 25
 * @ApiFilter(SearchFilter::class, properties={"title": "partial"})
... line 27
 */
class CheeseListing
... lines 30 - 148
```

Oh, and before we check this out, I'll click to open `SearchFilter`. This lives in a directory called `Filter` and... if I double click it... hey! We can see a *bunch* of other ones: `ExistsFilter`, `DateFilter`, `RangeFilter`, `OrderFilter` and more. These are all documented - but you can also jump right in and see how they work.

Anyways, go refresh the docs, open the `GET` collection operation and click to try it. *Now* we have a `title` filter box. Try... um... `cheese` and... Execute.

Oh, magnificent! It adds `?title=cheese` to the URL... and matched three of our four listings. The `hydra:search` property now contains a second entry advertising this new way to filter.

If we want to be able to search by another property, we can add that too: `description` set to `partial`.

This is easy to set up, but this type of database search is still pretty basic. *Fortunately*, while we won't cover it in this tutorial, if you need a truly robust search, API Platform can integrate with Elasticsearch: exposing your Elasticsearch data as readable API resources. That's pretty freaking cool!

Let's check out *two* more filters: a "range" filter, which will be *super* useful for our price property and another one that's... a bit special. Instead of filtering the number of results, it allows an API client to choose a subset of *properties* to return in the result. That's next.

Chapter 13: PropertyFilter: Sparse Fieldsets

In just a few minutes, we've given our API clients the ability to filter by published cheese listings and search by title and description. They may *also* need the ability to filter by *price*. That sounds like a job for... `RangeFilter`! Add another `@ApiFilter()` with `RangeFilter::class`. Let's immediately go up and add the `use` statement for that - the one for the ORM. Then, `properties={"price"}`.

150 lines | src/Entity/CheeseListing.php

```
... lines 1 - 7
use ApiPlatform\Core\Bridge\Doctrine\Orm\Filter\RangeFilter;
... lines 9 - 14
/**
 * @ApiResource(
... lines 17 - 27
 * @ApiFilter(RangeFilter::class, properties={"price"})
... line 29
 */
class CheeseListing
... lines 32 - 150
```

This filter is a bit nuts. Flip over, refresh the docs, and look at the GET collection operation. Woh! We now have a *bunch* of filter boxes, for price between, greater than, less than, greater than or equal, etc. Let's look for everything greater than 20 and... Execute. This adds `?price[gt]=20` to the URL. Oh, except, that's a search for everything greater than 20 cents! Try 1000 instead.

This returns just one item and, once again, it advertises the new filters down inside `hydra:search`.

Filters are super fun. Tons of filters come built-in, but you can *totally* add your own. From a high-level, a filter is basically a way for you to modify the Doctrine query that's made when fetching a collection.

Adding a Short Description

There's *one* more filter I want to talk about... and it's a bit special: instead of returning less results, it's all about returning less *fields*. Let's pretend that most descriptions are *super* long and contain HTML. On the front-end, we want to be able to fetch a collection of cheese listings, but we're *only* going to display a very *short* version of the description. To make that super easy, let's add a new field that returns this. Search for `getDescription()` and add a new method below called `public function getShortDescription()`. This will return a nullable string, in case description isn't set yet. Let's immediately add this to a group - `cheese_listing:read` so that it shows up in the API.

162 lines | src/Entity/CheeseListing.php

```
... lines 1 - 30
class CheeseListing
{
... lines 33 - 90
    /**
     * @Groups("cheese_listing:read")
     */
    public function getShortDescription(): ?string
    {
... lines 96 - 100
    }
... lines 102 - 160
}
```

Inside, if the `description` is already less than 40 characters, just return it. Otherwise, return a `substr` of the description - get the first 40 characters, then a little `...` at the end. Oh, and, in a real project, to make this better - you should probably use `strip_tags()` on description before doing any of this so that we don't split any HTML tags.

```
162 lines | src/Entity/CheeseListing.php
... lines 1 - 93
    public function getShortDescription(): string
    {
        if (strlen($this->description) < 40) {
            return $this->description;
        }

        return substr($this->description, 0, 40).'...';
    }
... lines 102 - 162
```

Refresh the docs... then open the GET item operation. Let's look for cheese listing id 1. And... there it is! The description was just *barely* longer than 40 characters. I'll copy the URL, put it into a new tab, and add `.jsonld` on the end to see this better.

At this point, adding the new field was *nothing* special. But... if some parts of my frontend *only* need the `shortDescription`... it's a bit wasteful for the API to *also* send the `description` field... especially if that field is *really*, *really* big! Is it possible for an API client to tell our API to *not* return certain fields?

Hello PropertyFilter

At the top of our class, add another filter with `PropertyFilter::class`. Move up, type `use PropertyFilter` and hit tab to auto-complete. This time, there's only one of these classes.

```
164 lines | src/Entity/CheeseListing.php
... lines 1 - 9
use ApiPlatform\Core\Serializer\Filter\PropertyFilter;
... lines 11 - 15
/**
 * @ApiResource(
... lines 18 - 29
 * @ApiFilter(PropertyFilter::class)
... line 31
 */
class CheeseListing
... lines 34 - 164
```

This filter *does* have some options, but it works perfectly well without doing anything else.

Go refresh our docs. Hmm, this doesn't make any difference here... this isn't a feature of our API that can be expressed in the OpenAPI spec doc.

But, this resource in our API *does* have a new super-power. In the other tab, choose the exact properties you want with `?properties[]=title&properties[]=shortDescription`. Hit it! Beautiful! We still get the standard JSON-LD fields, but then we *only* get back those two fields. This idea is sometimes called a "sparse fieldset", and it's a great way to allow your API client to ask for *exactly* what they want, while still organizing everything around concrete API resources.

Oh, and the user can't try to select *new* fields that aren't a part of our original data - you can't try to get `isPublished` - it just doesn't work, though you *can* enable this.

Next: let's talk about pagination. Yea, APIs *totally* need pagination! If we have 10,000 cheese listings in the database, we *can't* return *all* of them at once.

Chapter 14: Pagination

If we have a million, or a thousand... or even a *hundred* cheese listings, we can't return *all* of them when someone makes a GET request to `/api/cheeses`! The way that's solved in an API is *really* the same as on the web: pagination! And in API Platform... ah, it's so *boring*... you get a powerful, flexible pagination system... without doing... *anything*.

Let's go to the POST operation and create a few more cheese listings. I'll put in some simple data... and execute a bunch of times.. in fast forward. On a real project, I'd use data fixtures to help me get useful dummy data.

We should have about 10 or so thanks to the 4 we started with. Now, head up to the GET collection operation... and hit Execute. We *still* see *all* the results. That's because API Platform shows *30* results per page, by default. Because I don't feel like adding 20 more manually, this is a *great* time to learn how to change that!

Controlling Items Per Page

First, this can be changed globally in your `config/packages/api_platform.yaml` file. I won't show it now, but always remember that you can run:

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

to see a list of all of the valid configuration for that file and their current values. That would reveal a `collection.pagination` section that is *full* of config.

But we can also control the number of items per page on a resource-by-resource basis. Inside the `@ApiResource` annotation, add `attributes={}` ... which is a key that holds a variety of random configuration for API Platform. And then, `"pagination_items_per_page": 10` .

167 lines | src/Entity/CheeseListing.php

```
... lines 1 - 15
/**
 * @ApiResource(
... lines 18 - 25
 *     attributes={
 *         "pagination_items_per_page"=10
 *     }
 * )
... lines 30 - 34
 */
class CheeseListing
... lines 37 - 167
```

I mentioned earlier that a lot of API Platform is learning exactly *what* you can configure inside of this annotation and how. *This* is a perfect example.

Go back to the docs - no need to refresh. Just hit Execute. Let's see... the total items are 11... but if you counted, this is only showing *10* results! Hello pagination! We also have a new `hydra:view` property. This advertises that pagination is happening and how we can "browse" through the other pages: we can follow `hydra:first` , `hydra:last` and `hydra:next` to go to the first, last or next page. The URLs look exactly like I want: `?page=1` , `?page=2` and so on.

Open a new tab and go back to `/api/cheeses.jsonld` . Yep, the first 10 results. Now add `?page=2` ... to see the one, last result.

Filtering *also* still works. Try `.jsonld?title=cheese` . That returns... only 10 results... so no pagination! That's no fun. Let's go back to the docs, open the POST endpoint and add a few more. Oh, but let's make sure that we add one with "cheese" in the title. Hit Execute a few times.

Now go refresh the GET collection operation with `?title=cheese` . Nice! We have 13 total results and this shows the first 10.

What's *really* nice is that the pagination links *include* the filter! That is *super* useful in JavaScript: you don't need to try to hack the URL together manually by combining the `page` and filter information: just read the links from hydra and use them.

Next, we know that our API can return JSON-LD, JSON and HTML. And... that's probably all we need, right? Let's see how easy it is to add *more* formats... including making our cheese listings downloadable as a CSV.

Chapter 15: More Formats: HAL & CSV

API Platform supports multiple input and output formats. You can see this by going to `/api/cheeses.json` to get "raw" JSON or `.jsonld` or even `.html`, which loads the HTML documentation. But adding the extension like this is kind of a "hack" that API Platform added just to make things easier to play with.

Instead, you're supposed to choose what "format", or "representation" you want for a resource via content negotiation. The documentation already does this and shows it in the examples: it sends an `Accept` header, which API Platform uses to figure out which format the serializer should use.

Adding a new Format: HAL

Out-of-the-box, API Platform uses 3 formats... but it *actually* supports a bunch more: JSON-API, HAL JSON, XML, YAML and CSV. Find your terminal and run:

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

This is our current API Platform configuration, *including* default values. Check out `formats`. Hey! It shows the 3 formats that we've seen so far and the mime types for each - that's the value that should be sent in the `Accept` header to activate them.

Let's add *another* format. To do that, copy this entire formats section. Then open `config/packages/api_platform.yaml` and paste here.

```
14 lines | config/packages/api_platform.yaml
api_platform:
... lines 2 - 3
    formats:
        jsonld:
            mime_types:
                - application/ld+json
        json:
            mime_types:
                - application/json
        html:
            mime_types:
                - text/html
```

This will make sure that we *keep* these three formats. Now, let's add a new one: `jsonhal`. This is one of the *other* formats that API Platform supports out-of-the box. Below, add `mime_types:` then the standard content type for this format: `application/hal+json`.

```
17 lines | config/packages/api_platform.yaml
api_platform:
... lines 2 - 13
        jsonhal:
            mime_types:
                - application/hal+json
```

Cool! And *just* like that... our *entire* API supports a new format! Refresh the docs and open the GET operation to see cheese listing 1. Before you hit execute, open the format drop down and... hey hey! Select `application/hal+json`. Execute!

Say hello to the JSON HAL format: a, sort of "competing" format with JSON-LD or JSON-API, all of which aim to standardize how you should *structure* your JSON: where you data should live, where links should live, etc.

In HAL, you have an `_links` property. It only has a link to `self` now, but this often contains links to other resources.

This is more fun if we try the GET collection operation: select `application/hal+json` and hit Execute. It's kinda cool to see how the different formats "advertise" pagination. HAL uses `_links` with `first`, `last` and `next` keys. If we were on page 2, there would also be a `prev` field.

Having this format available may or may not be handy for you - the awesome part is you can *choose* whatever you want. *And*, understanding formats unlocks other interesting possibilities.

CSV Format

For example, what if, for some reason, you or someone who uses your API wants to be able to fetch the cheese listing resources as CSV? Yea, that's totally possible! But instead of making that format available globally for *every* resource, let's activate it for *only* our `CheeseListing`.

Back inside that class, once again under this special `attributes` key, add `"formats"`. If you want to keep all the existing formats, you'll need to list them here: `jsonld`, `json`, then... let's see, ah yep, `html` and `jsonhal`. To add a *new* format, say `csv`, but *set* this to a new array with `text/csv` inside.

168 lines | src/Entity/CheeseListing.php

```
... lines 1 - 15
/**
 * @ApiResource(
... lines 18 - 25
 *     attributes={
... line 27
 *         "formats"={"jsonld", "json", "html", "jsonhal", "csv"={"text/csv"}}
 *     }
 * )
... lines 31 - 35
 */
class CheeseListing
... lines 38 - 168
```

This is the mime type for the format. We didn't need to add mime types for the *other* formats because they're already set up in our config file.

Let's try it! Go refresh the docs. Suddenly, *only* for *this* resource... which, ok, we only have one resource right now... but CheeseListing *now* has a CSV format. Select it and Execute.

There it is! And we can try this directly in the browser by adding `.csv` on the end. My browser downloaded it... so let's flip over and `cat` that file to see what it looks like. The line breaks look a bit weird, but that *is* valid CSV.

A better example is getting the full list: `/api/cheeses.csv`. Let's go see what that looks like in the terminal as well. This is awesome! Fastest CSV download feature I've *ever* built.

And... yea! You can *also* create your own format and activate it in this same way. It's a powerful idea: our *one* API Resource can be *represented* in any number of different ways, including formats - like CSV - which you don't need... until that one random situation when you suddenly *really* need them.

Next, it's time to stop letting users create cheese listings with *any* crazy data they want. It's time to add validation!

Chapter 16: Validation

There are a *bunch* of different ways that an API client can send bad data: they might send malformed JSON... or send a blank `title` field... maybe because a user forgot to fill in a field on the frontend. The job of our API is to respond to *all* of these situations in an informative and consistent way so that errors can be easily understood, parsed and communicated back to humans.

Invalid JSON Handling

This is one of the areas where API Platform *really* excels. Let's do some experimenting: what happens if we accidentally send some invalid JSON? Remove the last curly brace.

Try it! Woh! Nice! This comes back with a *new* "type" of resource: a `hydra:error`. If an API client understands Hydra, they'll instantly know that this response contains error details. And even if someone has *never* heard of Hydra before, this is a *super* clear response. And, *most* importantly, *every* error has the same structure.

The status code is also 400 - which means the client made an error in the request - and `hydra:description` says "Syntax error". Without doing anything, API Platform is already handling this case. Oh, and the `trace`, while maybe useful right now during development, will *not* show up in the production environment.

Field Validation

What happens if we just delete everything and send an *empty* request? Oh... that's *still* technically invalid JSON. Try just `{}`.

Ah... *this* time we get a 500 error: the database is exploding because some of the columns cannot be null. Oh, and like I mentioned earlier, if you're using Symfony 4.3, you might already see a validation error instead of a database error because of a new feature where validation rules are automatically added by reading the Doctrine database rules.

But, whether you're seeing a 500 error, or Symfony is *at least* adding some basic validation for you, the input data that's allowed is something *we* want to control: I want to decide the *exact* rules for each field.

Tip

Actually, the auto-validation was not enabled by default in Symfony 4.3, but may be in Symfony 4.4.

Adding validation rules is... oh, so nice. And, unless you're new to Symfony, this will look *delightfully* boring. Above `title`, to make it required, add `@Assert\NotBlank()`. Let's also add `@Assert\Length()` here with, how about, `min=2` and `max=50`. Heck, let's even set the `maxMessage` to

> Describe your cheese in 50 chars or less

```
177 lines | src/Entity/CheeseListing.php
... lines 1 - 14
use Symfony\Component\Validator\Constraints as Assert;
... lines 16 - 37
class CheeseListing
{
... lines 40 - 46
    /**
... lines 48 - 49
     * @Assert\NotBlank()
     * @Assert\Length(
     *     min=2,
     *     max=50,
     *     maxMessage="Describe your cheese in 50 chars or less"
     * )
     */
    private $title;
... lines 58 - 175
}
```

What else? Above `description`, add `@Assert\NotBlank`. And for price, `@Assert\NotBlank()`. You could also add a `GreaterThan` constraint to make sure this is above zero.

```
177 lines | src/Entity/CheeseListing.php
... lines 1 - 37
class CheeseListing
{
... lines 40 - 58
    /**
... lines 60 - 61
     * @Assert\NotBlank()
     */
    private $description;
... line 65
    /**
... lines 67 - 70
     * @Assert\NotBlank()
     */
    private $price;
... lines 74 - 175
}
```

Ok, switch back over and try sending *no* data again. Woh! It's awesome! The `@type` is `ConstraintViolationList`! That's one of the types that was described by our JSON-LD documentation!

Go to `/api/docs.jsonld`. Down under `supportedClasses`, there's `EntryPoint` and here is `ConstraintViolation` and `ConstraintViolationList`, which describes what each of these types look like.

And the data on the response is really useful: a `violations` array where each error has a `propertyPath` - so we know what field that error is coming from - and `message`. So... it all just... works!

And if you try passing a `title` that's longer than 50 characters... and execute, there's our custom message.

Validation for Passing Invalid Types

Perfect! We're done! But wait... aren't we missing a bit of validation on the `price` field? We have `@NotBlank`... but what's preventing us from sending *text* for this field? Anything?

Let's try it! Set the price to `apple`, and execute.

Ha! It *fails* with a 400 status code! That's awesome! It says:

> The type of the price attribute must be int, string given

If you look closely, it's failing during the deserialization process. It's not *technically* a validation error - it's a serialization error. But to the API client, it looks just about the same, except that this returns an Error type instead of a `ConstraintViolationList`... which probably makes sense: if some JavaScript is making this request, that JavaScript should probably have some built-in validation rules to prevent the user from *ever* adding text to the price field.

The point is: API Platform, well, really, the serializer, knows the types of your fields and will make sure that nothing insane gets passed. It *knows* that price is an integer from *two* sources actually: the Doctrine `@ORM\Column` metadata on the field *and* the argument type-hint on `setPrice()`.

The only thing *we* really need to worry about is adding "business rules" validation: adding the `@Assert` validation constraints to say that this field is required, that field has a min length, etc. Basically, validation in API Platform works *exactly* like validation in *every* Symfony app. And API Platform takes care of the boring work of mapping serialization and validation failures to a 400 status code and descriptive, consistent error responses.

Next, let's create a *second* API Resource! A User! Because things will get *really* interesting when we start creating *relations* between resources.

Chapter 17: Creating the User Entity

We're not going to talk *specifically* about security in this tutorial - we'll do that in our next course and give it proper attention. But, even forgetting about security and logging in and all that, there's a *pretty* good chance that your API will have some concept of "users". In our case, a "user" will post a cheese listing and becomes its "owner". And maybe later, in order to buy a CheeseListing, one User might send a message to another User. It's time to take our app to the next level by creating that entity.

make:user

And even though I'm telling you not to think about security, instead of creating the user entity with `make:entity` like I normally would, I'm actually going to use `make:user`,

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

Yea, this *will* set up a few security-related things... but nothing that we'll use yet. Watch part 2 of this series for all that stuff.

Anyways, call the class `User`, and I *do* want to store users in the database. For the unique display name, I'm going to have users log in via email, so use that. And then:

> Does this app need to hash or check user passwords?

We'll talk more about this in the security tutorial. But *if* users will need to log in to your site via a password *and* your app will be responsible for checking to see if that password is valid - you're not just sending the password to some *other* service to be verified - then answer yes. It doesn't matter if the user will enter the password via an iPhone app that talks to your API or via a login form - answer yes if your app is responsible for managing user passwords.

I'll use the Argon2i password hasher. But! If you don't see this question, that's ok! Starting in Symfony 4.3, you don't need to choose a password hashing algorithm because Symfony can choose the best available automatically. Really cool stuff.

Let's go see what this did! I'm happy to say... not much! First, we now have a `User` entity. And... there's nothing special about it: it *does* have a few extra security-related methods, like `getRoles()`, `getPassword()`, `getSalt()` and `eraseCredentials()`, but they won't affect what we're doing. Mostly we have a normal, boring entity with `$id`, `$email`, a `$roles` array property, and `$password`, which will eventually store the hashed password.

```
114 lines | src/Entity/User.php
... lines 1 - 2
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 */
class User implements UserInterface
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */
    private $email;

    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    /**
     * @var string The hashed password
     * @ORM\Column(type="string")
     */
    private $password;
... lines 35 - 112
}
```

This also created the normal `UserRepository` and made a *couple* of changes to `security.yaml`: it set up `encoders` - this might say `auto` for you, thanks to the new Symfony 4.3 feature - and the user provider. All things to talk more about later. So... just forget they're here and instead say... yay! We have a `User` entity!

```
33 lines | config/packages/security.yaml
security:
    encoders:
        App\Entity\User:
            algorithm: argon2i
... lines 5 - 6
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: email
... lines 13 - 33
```

```
51 lines | src/Repository/UserRepository.php
... lines 1 - 2
namespace App\Repository;

use App\Entity\User;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Symfony\Bridge\Doctrine\RegistryInterface;

/**
 * @method User|null find($id, $lockMode = null, $lockVersion = null)
 * @method User|null findOneBy(array $criteria, array $orderBy = null)
 * @method User[]    findAll()
 * @method User[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class UserRepository extends ServiceEntityRepository
{
    public function __construct(RegistryInterface $registry)
    {
        parent::__construct($registry, User::class);
    }
... lines 21 - 49
}
```

Adding username Field

Thanks to the command, the entity has an `email` property, and I'm planning to make users log in by using that. But I *also* want each user to have a "username" that we can display publicly. Let's add that: find your terminal and run:

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

Update `User` and add `username` as a `string`, 255, not nullable in the database, and hit enter to finish.

Now open up `User`... and scroll down to `getUsername()`. The `make:user` command generated this and returned `$this->email`... because that's what I chose as my "display" name for security. Now that we really *do* have a username field, return `$this->username`.

```
126 lines | src/Entity/User.php
... lines 1 - 10
class User implements UserInterface
{
... lines 13 - 35
    /**
     * @ORM\Column(type="string", length=255)
     */
    private $username;
... lines 40 - 62
    public function getUsername(): string
    {
        return (string) $this->username;
    }
... lines 67 - 124
}
```

Oh, and while we're making this class, just, *amazing*, the `make:user` command knew that `email` should be unique, so it added `unique=true`. Let's *also* add that to `username`: `unique=true`.

```
126 lines | src/Entity/User.php
... lines 1 - 35
    /**
     * @ORM\Column(type="string", length=255, unique=true)
     */
    private $username;
... lines 40 - 126
```

That is a *nice* entity! Let's sync up our database by running:

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

Move over... and double-check the SQL: `CREATE TABLE user` - looks good!

```
36 lines | src/Migrations/Version20190509185722.php
... lines 1 - 12
final class Version20190509185722 extends AbstractMigration
{
... lines 15 - 19
    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\

        $this->addSql('CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, email VARCHAR(180) NOT NULL, roles JSON NOT N
    }
... lines 27 - 34
}
```

Run it with:

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

Perfect! We have a *gorgeous* new Doctrine entity... but as far as API Platform is concerned, we still only have one API resource: `CheeseListing`.

Next: let's expose `User` as an API Resource and use all of our new knowledge to *perfect* that new resource in... about 5 minutes.

Chapter 18: User API Resource

I want to expose our new `User` entity as an API resource. And we know how to do that! Add... `@ApiResource`!

```
128 lines | src/Entity/User.php
... lines 1 - 4
use ApiPlatform\Core\Annotation\ApiResource;
... lines 6 - 8
/**
 * @ApiResource()
... line 11
 */
class User implements UserInterface
... lines 14 - 128
```

Just like that! Yes! Our API docs show one new resource with five new endpoints, or operations. And at the bottom, here's the new `User` model.

Hmm, but it's a bit strange: both the hashed `password` field and `roles` array are part of the API. Yea, we could create a new user right now and pass whatever roles *we* think that user should have! That might be ok for an admin user to be able to do, but not anyone. Let's take control of things.

UUID's?

Oh, one thing I want you to notice is that, so far, the primary key is always being used as the "id" in our API. This *is* something that's flexible in API Platform. In fact, instead of using an auto-increment id, *one* option is to use a UUID. We're not going to use them in this tutorial, but using a UUID as your identifier *is* something that's supported by Doctrine and API Platform. UUIDs work with any database, but they *are* stored more efficiently in PostgreSQL than MySQL, though we use some UUID's in MySQL in some parts of SymfonyCasts.

But... why am I telling you about UUID's? What's wrong with auto-increment ids? Nothing... but.... UUID's *may* help simplify your JavaScript code. Suppose we write some JavaScript to create a new `CheeseListing`. With auto-increment ids, the process looks like this: make a POST request to `/api/cheeses`, wait for the response, then read the `@id` off of the response and store it somewhere... because you'll usually need to know the id of each cheese listing. With UUID's, the process looks like this: generate a UUID in JavaScript - that's *totally* legal - send the POST request and... that's it! With UUID's, you don't need to wait for the AJAX call to finish so you can read the id: *you* created the UUID in JavaScript, so you already know it. *That* is why UUID's can often be really nice.

To make this all work, you'll need to configure your entity to use a UUID *and* add a `setId()` method so that it's possible for API Platform to set it. Or you can create the auto-increment id and add a *separate* UUID property. API Platform has an annotation to mark a field as the "identifier".

Normalization & Denormalization Groups

Anyways, let's take control of the serialization process so we can remove any weird fields - like having the encoded password be returned. We'll do the *exact* same thing we did in `CheeseListing`: add normalization and denormalization groups. Copy the two context lines, open up `User` and paste. I'm going to remove the `swagger_definition_name` part - we don't really need that. For normalization, use `user:read` and for denormalization, `user:write`.

```
135 lines | src/Entity/User.php
... lines 1 - 9
/**
 * @ApiResource(
 *     normalizationContext={"groups"={"user:read"}},
 *     denormalizationContext={"groups"={"user:write"}},
 * )
... line 15
 */
class User implements UserInterface
... lines 18 - 135
```

We're following the same pattern we've been using. Now... let's think: what fields do we need to expose? For `$email`, add `@Groups({})` with `"user:read", "user:write"`: this is a readable and writable field. Copy that, paste above `password` and make it only `user:write`.

```
135 lines | src/Entity/User.php
... lines 1 - 7
use Symfony\Component\Serializer\Annotation\Groups;
... lines 9 - 16
class User implements UserInterface
{
... lines 19 - 25
    /**
... line 27
     * @Groups({"user:read", "user:write"})
     */
    private $email;
... lines 31 - 36
    /**
... lines 38 - 39
     * @Groups({"user:write"})
     */
    private $password;
... lines 43 - 133
}
```

This... doesn't really make sense yet. I mean, it's not *readable* anymore, which makes *perfect* sense. But this will eventually store the *encoded* password, which is *not* something that an API client will set directly. But... we're going to worry about all of that in our security tutorial. For now, because password is a required field in the database, let's temporarily make it writable so it doesn't get in our way.

Finally, make `username` readable and writable as well.

```
135 lines | src/Entity/User.php
... lines 1 - 16
class User implements UserInterface
{
... lines 19 - 43
    /**
... line 45
     * @Groups({"user:read", "user:write"})
     */
    private $username;
... lines 49 - 133
}
```

Let's try it! Refresh the docs. *Just* like with `CheeseListing` we now have *two* models: we can read `email` and `username` and we can *write* `email`, `password` and `username`.

The only other thing we need to make this a *fully* functional API resource is validation. To start, both `$email` and `$username` need to be unique. At the top of the class, add `@UniqueEntity()` with `fields={"username"}`, and *another* `@UniqueEntity()` with `fields={"email"}`.

```
142 lines | src/Entity/User.php
... lines 1 - 6
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
... lines 8 - 11
/**
... lines 13 - 16
 * @UniqueEntity(fields={"username"})
 * @UniqueEntity(fields={"email"})
... line 19
 */
class User implements UserInterface
... lines 22 - 142
```

Then, let's see, `$email` should be `@Assert\NotBlank()` and `@Assert\Email()`, and `$username` needs to be `@Assert\NotBlank()`. I won't worry about password yet, that needs to be properly fixed anyways in the security tutorial.

```
142 lines | src/Entity/User.php
... lines 1 - 9
use Symfony\Component\Validator\Constraints as Assert;
... lines 11 - 20
class User implements UserInterface
{
... lines 23 - 29
    /**
... lines 31 - 32
     * @Assert\NotBlank()
     * @Assert\Email()
     */
    private $email;
... lines 37 - 49
    /**
... lines 51 - 52
     * @Assert\NotBlank()
     */
    private $username;
... lines 56 - 140
}
```

So, I think we're good! Refresh the documentation and let's start creating users! Click "Try it out". I'll use my real-life personal email address: `cheeselover1@example.com`. The password doesn't matter... and let's make the username match the email without the domain... so I don't confuse myself. Execute!

Woohoo! 201 success! Let's create *one* more user... *just* to have some better data to play with.

Failing Validation

Oh, and what if we send up empty JSON? Try that. Yea! 400 status code.

Ok... we're done! We have 1 new resource, *five* new operations, control over the input and output fields, validation, pagination and we could easily add filtering. Um... that's amazing! *This* is the power of API Platform. And as you get better and better at using it, you'll develop even faster.

But ultimately, we created the new `User` API resource *not* just because creating users is fun: we did it so we could *relate* each `CheeseListing` to the `User` that "owns" it. In an API, relations are a *key* concept. And you're going to *love* how they work in API Platform.

Chapter 19: Relating Resources

We have a cheese resource and a user resource. Let's link them together! Ok, the *real* problem we need to solve is this: each `CheeseListing` will be "owned" by a single user, which is something we need to set up in the database but *also* something we need to expose in our API: when I look at a `CheeseListing` resource, I need to know which user posted it!

Creating the Database Relationship

Let's set up the database first. Find your terminal and run:

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

Let's update the `CheeseListing` entity and add a new `owner` property. This will be a `relation` to the `User` entity... which will be a `ManyToOne` relationship: every `CheeseListing` has one `User`. Should this new property be nullable in the database? Say no: *every* `CheeseListing` *must* have an `owner` in our system.

Next, it asks a *super* important question: do we want to add a new property to `User` so that we can access and update cheese listings on it - like `$user->getCheeseListings()`. Doing this is optional, and there are *two* reasons why you might want it. First, if you think writing `$user->getCheeseListings()` in your code might be convenient, you'll want it! *Second*, when you fetch a `User` in our API, if you want to be able to see what cheese listings this user owns as a property in the JSON, you'll *also* want this. More on that soon.

Anyways, say yes, call the property `cheeseListings` and say no to `orphanRemoval`. If you're not familiar with that option... then you don't need it. And... bonus! A bit later in this tutorial, I'll show you why and when this option *is* useful.

Hit enter to finish! As usual, this did a few things: it added an `$owner` property to `CheeseListing` along with `getOwner()` and `setOwner()` methods. Over on `User`, it added a `$cheeseListings` property with a `getCheeseListings()` method... but *not* a `setCheeseListings()` method. Instead, `make:entity` generated `addCheeseListing()` and `removeCheeseListing()` methods. Those will come in handy later.

195 lines | src/Entity/CheeseListing.php

```
... lines 1 - 37
class CheeseListing
{
... lines 40 - 84
    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\User", inversedBy="cheeseListings")
     * @ORM\JoinColumn(nullable=false)
     */
    private $owner;
... lines 90 - 182
    public function getOwner(): ?User
    {
        return $this->owner;
    }

    public function setOwner(?User $owner): self
    {
        $this->owner = $owner;

        return $this;
    }
}
```

185 lines | src/Entity/User.php

```
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 58
    /**
     * @ORM\OneToMany(targetEntity="App\Entity\CheeseListing", mappedBy="owner")
     */
    private $cheeseListings;
... line 63
    public function __construct()
    {
        $this->cheeseListings = new ArrayCollection();
    }
... lines 68 - 156
    public function getCheeseListings(): Collection
    {
        return $this->cheeseListings;
    }

    public function addCheeseListing(CheeseListing $cheeseListing): self
    {
        if (!$this->cheeseListings->contains($cheeseListing)) {
            $this->cheeseListings[] = $cheeseListing;
            $cheeseListing->setOwner($this);
        }

        return $this;
    }

    public function removeCheeseListing(CheeseListing $cheeseListing): self
    {
        if ($this->cheeseListings->contains($cheeseListing)) {
            $this->cheeseListings->removeElement($cheeseListing);
            // set the owning side to null (unless already changed)
            if ($cheeseListing->getOwner() === $this) {
                $cheeseListing->setOwner(null);
            }
        }

        return $this;
    }
}
```

Let's create the migration:

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

And open that up... *just* to make sure it doesn't contain anything extra.

```
40 lines | src/Migrations/Version20190509190403.php
... lines 1 - 12
final class Version20190509190403 extends AbstractMigration
{
... lines 15 - 19
    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration can only be executed safely on \'mysql\

        $this->addSql('ALTER TABLE cheese_listing ADD owner_id INT NOT NULL');
        $this->addSql('ALTER TABLE cheese_listing ADD CONSTRAINT FK_356577D47E3C61F9 FOREIGN KEY (owner_id) REFERENCE
        $this->addSql('CREATE INDEX IDX_356577D47E3C61F9 ON cheese_listing (owner_id)');
    }
... lines 29 - 38
}
```

Looks good - altering the table and setting up the foreign key. Execute that:

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

Oh no! It exploded!

> Cannot add or update a child row, a foreign key constraint fails

... on the `owner_id` column of `cheese_listing`. Above the `owner` property, we set `nullable=false`, which means that the `owner_id` column in the table *cannot* be null. But... because our `cheese_listing` table already has some rows in it, when we try to add that new column... it doesn't know what value to use for the existing rows and it explodes.

It's a *classic* migration failure. If our site were already on production, we would need to make this migration fancier by adding the new column first as nullable, set the values, then change it to not nullable. But because we're not there yet... we can just drop all our data and try again. Run:

```
$ php bin/console doctrine:schema:drop --help
```

... because this has an option I can't remember. Ah, here it is: `--full-database` will make sure we drop *every* table, *including* `migration_versions`. Run:

```
$ php bin/console doctrine:schema:drop --full-database --force
```

Now we can run *every* migration to create our schema from scratch:

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

Nice!

Exposing the Relation Property

Back to work! In `CheeseListing`, we have a new property and a new getter and setter. But because we're using normalization and denormalization groups, this new stuff is *not* exposed in our API.

To begin with, here's the goal: when we *create* a `CheeseListing`, an API client should be able to specify *who* the owner is. And when we read a `CheeseListing`, we should be able to *see* who owns it. That might feel a bit weird at first: are we *really* going to allow an API client to create a `CheeseListing` and freely choose *who* its owner is? For now, yes: setting the owner on a cheese listing is *no* different than setting *any* other field. Later, once we have a real security system, we'll start locking things down so that I can't create a `CheeseListing` and say that someone else owns it.

Anyways, to make `owner` part of our API, copy the `@Groups()` off of `$price`... and add those above `$owner`.

```
196 lines | src/Entity/CheeseListing.php
... lines 1 - 37
class CheeseListing
{
... lines 40 - 84
    /**
... lines 86 - 87
     * @Groups({"cheese_listing:read", "cheese_listing:write"})
     */
    private $owner;
... lines 91 - 194
}
```

Let's try it! Move over and refresh the docs. But before we look at `CheeseListing`, let's create a `User` so we have some data to play with. I'll give this an email, any password, a username and... Execute. Great - 201 success. Tweak the data and create one more user.

Now, the moment of truth: click to create a new `CheeseListing`. Interesting... it says that `owner` is a "string"... which *might* be surprising... aren't we going to set this to the integer id? Let's find out. Try to sell a block of unknown cheese for $20, and add a description.

For owner, what do we put here? Let's see... the two users we just created had ids 2 and 1. Okay! Set owner to `1` and Execute!

Woh! It *fails* with a 400 status code!

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

It turns out that setting owner to the id is *not* correct! Next, let's fix this, talk more about IRIs and add a new `cheeseListings` property to our `User` API resource.

Chapter 20: Relations and IRIs

I just tried to create a `CheeseListing` by setting the `owner` property to 1: the id of a real user in the database. But... it didn't like it! Why? Because in API Platform and, commonly, in modern API development in general, we do *not* use ids to refer to resources: we use IRIs. For me, this was strange at first... but I quickly fell in *love* with this. Why pass around integer ids when URLs are *so* much more useful?

Check out the response of the user we just created: like *every* JSON-LD response, it contains an `@id` property... that isn't an id, it's an IRI! And *this* is what you'll use whenever you need to refer to this resource.

Head back up to the `CheeseListing` POST operation and set `owner` to `/api/users/1`. Execute that. This time... it works!

And check it out, when it transforms the new `CheeseListing` into JSON, the `owner` property is that same IRI. *That* is why Swagger documents this as a "string"... which isn't *totally* accurate. Sure, on the surface, `owner` *is* a string... and that's what Swagger is showing in the `cheeses-Write` model.

But *we* know... with our *human* brains, that this string is special: it *actually* represents a "link" to a related resource. And... even though Swagger doesn't quite understand this, check out the JSON-LD documentation: at `/api/docs.jsonld`. Let's see, search for owner. Ha! *This* is a bit smarter: JSON-LD knows that this is a Link... with some fancy metadata to basically say that the link is to a `User` resource.

The big takeaway is this: a relation is just a normal property, except that it's represented in your API with its IRI. Pretty cool.

Adding cheesesListings to User

What about the other side of the relationship? Use the docs to go fetch the `CheeseListing` with id = 1. Yep, here's all the info, including the `owner` as an IRI. But what if we want to go the other direction?

Let's refresh to close everything up. Go fetch the `User` resource with id 1. Pretty boring: `email` and `username`. What if you *also* want to see what cheeses this user has posted?

That's *just* as easy. Inside `User` find the `$username` property, copy the `@Groups` annotation, then paste above the `$cheeseListings` property. But... for now, let's *only* make this readable: just `user:read`. We're going to talk about how you can *modify* collection relationships later.

186 lines | src/Entity/User.php

```
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 58
    /**
... line 60
     * @Groups("user:read")
     */
    private $cheeseListings;
... lines 64 - 184
}
```

Ok, refresh and open the GET item operation for User. Before even trying this, it's *already* advertising that it will *now* return a `cheeseListings` property, which, interesting, will be an array of *strings*. Let's see what `User` id 1 looks like. Execute!

Ah.. it *is* an array! An array of IRI strings - of *course*. By default, when you relate two resources, API Platform will output the related resource as an IRI or an *array* of IRIs, which is beautifully simple. If the API client needs more info, they can make another request to that URL.

Or... if you want to avoid that extra request, you *could* choose instead to *embed* the cheese listing data *right* into the user resource's JSON. Let's chat about that next.

Chapter 21: Embedded Relation

When two resources are related to each other, this can be expressed in two different ways in an API. The first is with IRIs - basically a "link" to the other resource. We can't see the data for the related `CheeseListing`, but if we need it, we could make a second request to this URL and... boom! We've got it.

But, for performance purposes, you might say:

> You know what? I don't want to have to make one request to fetch user data and then one *more* request to get the data for each cheese listing they own: I want to get it all at once!

And *that* describes the *second* way of expressing a relationship: instead of just returning a link to a cheese listing, you can *embed* its data right inside!

Embedded CheeseListing into User

As a reminder, when we normalize a `User`, we include everything in the `user:read` group. So that means `$email`, `$username` and `$cheeseListings`, which is why that property shows up at all.

To make this property return *data*, instead of just an IRI, here's what you need to do: go into the related entity - so `CheeseListing` - and add this `user:read` group to at least one property. For example, add `user:read` above `$title`... and how about also above `$price`.

196 lines | src/Entity/CheeseListing.php

```
... lines 1 - 37
class CheeseListing
{
... lines 40 - 46
    /**
... line 48
     * @Groups({"cheese_listing:read", "cheese_listing:write", "user:read"})
... lines 50 - 55
     */
    private $title;
... lines 58 - 65
    /**
... lines 67 - 69
     * @Groups({"cheese_listing:read", "cheese_listing:write", "user:read"})
... line 71
     */
    private $price;
... lines 74 - 194
}
```

Let's see what happens! We don't even need to refresh, just Execute. Woh! Instead of an array of strings, it's now an array of *objects*! Well, this user only owns *one* CheeseListing, but you get the idea. Each item has the standard `@type` and `@id` *plus* whatever properties we added to the group: `title` and `price`.

It's beautifully simple: the serializer knows to serialize all fields in the `user:read` group. It first looks at `User` and finds `email`, `username` and `cheeseListings`. It *then* keeps *going* and, inside of `CheeseListing`, finds that group on `title` and `price`.

Relation Strings vs Objects

This means that each relation property may be a *string* - the IRI - *or* an *object*. And an API client can tell the difference. If you get back an object, you know it will have `@id`, `@type` and some other data properties. If you get back a string, you know it's an IRI that you can use to go get the real data.

Embedding User into CheeseListing

We can do the *same* thing on the other side of the relationship. Use the docs to get the `CheeseListing` with `id = 1`. Yep! The `owner` property is a string. But it *might* be convenient for the CheeseListing JSON to *at least* contain the username of the owner... so we don't need to go fetch the *entire* User just to display who owns it.

Inside `CheeseListing`, the normalization process will serialize everything in the `cheese_listing:read` group. Copy that. The `owner` property, of course, already has this group above it, which is why we see it in our API. Inside `User`, find `$username`... and add `cheese_listing:read` to that.

```
186 lines | src/Entity/User.php
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 51
    /**
... line 53
     * @Groups({"user:read", "user:write", "cheese_listing:read"})
... line 55
     */
    private $username;
... lines 58 - 184
}
```

Let's try this thing! Move back over and... Execute! And... ha! Perfect! It expands to an object and includes the `username`.

Embedding Data Only on when GETing a Single Item

Does it work if we GET the *collection* of cheese listings? Try it out! Well... ok, there's only *one* `CheeseListing` in the database right now, but of course! It embeds the owner in the same way.

So... about that... new challenge! What if we want to embed the `owner` data when I fetch a *single* `CheeseListing`... but, to keep the response from being gigantic... we *don't* want to embed the data when we fetch the *collection*. Is that possible?

Totally! Again, for `CheeseListing`, when we normalize, we include everything in the `cheese_listing:read` group. That's true regardless of whether we're GETting the collection of cheese listings or just GETting a single item. *But*, *tons* of things - including groups - can be changed on an operation-by-operation basis.

For example, under `itemOperations`, break the `get` operation configuration onto multiple lines and add `normalization_context`. One of the tricky things with the config here is that the top-level keys are lower camel case, like `normalizationContext`. But deeper keys are usually snake case, like `normalization_context`. That... can be a little inconsistent - and it's easy to mess these up. Be careful.

Anyways, the goal is to *override* the normalization context, but *only* for this *one* operation. Set this to the normal `groups` and another array. Inside, we're going to say:

> Hey! When you are getting a *single* item, I want to include all of the properties that have the `cheese_listing:read` group like normal. But I *also* want to include any properties in a new `cheese_listing:item:get` group.

```
198 lines | src/Entity/CheeseListing.php
... lines 1 - 16
/**
 * @ApiResource(
... line 19
 *     itemOperations={
 *         "get"={
 *             "normalization_context"={"groups"={"cheese_listing:read", "cheese_listing:item:get"}},
 *         },
... line 24
 *     },
... lines 26 - 32
 * )
... lines 34 - 38
 */
class CheeseListing
... lines 41 - 198
```

We'll talk more about it later - but I'm using a *specific* naming convention for this operation-specific group - the "entity name", colon, item or collection, colon, then the HTTP method - `get`, `post`, `put`, etc.

If we re-fetch a single `CheeseListing`.... it makes no difference: we're including a new group for serialization - yaaaay - but nothing is *in* the new group.

Here's the magic. Copy the new group name, open `User`, and above the `$username` property, replace `cheese_listing:read` with `cheese_listing:item:get`.

```
186 lines | src/Entity/User.php
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 51
    /**
... line 53
     * @Groups({"user:read", "user:write", "cheese_listing:item:get"})
... line 55
     */
    private $username;
... lines 58 - 184
}
```

That's it! Move back to the documentation and fetch a *single* `CheeseListing`. And... *perfect* - it *still* embeds the owner - there's the username. But now, close that up and go to the GET *collection* endpoint. Execute! Yes! Owner is back to an IRI!

These serialization groups *can* get a little complex to think about, but *wow* are they powerful.

Next... when we fetch a `CheeseListing`, some of the owner's data is *embedded* into the response. So... I have kind of a crazy question: when we're updating a `CheeseListing`... could we *also* update some data on the owner by *sending* embedded data? Um... yea! That's next.

Chapter 22: Embedded Write

Here's an interesting question: if we fetch a single `CheeseListing`, we can see that the `username` comes through on the `owner` property. And obviously, if we, edit a specific `CheeseListing`, we can *totally* change the owner to a different owner. Let's actually try this: let's *just* set `owner` to `/api/users/2`. Execute and... yep! It updated!

That's great, and it works pretty much like a normal, scalar property. But... looking back at the results from the GET operation... here it is, if we can *read* the `username` property off of the related owner, instead of changing the owner entirely, could we *update* the current owner's username *while* updating a `CheeseListing`?

It's kind of a weird example, but editing data *through* an embedded relation *is* possible... and, at the very least, it's an *awesome* way to *really* understand how the serializer works.

Trying to Update the Embedded owner

Anyways... let's just try it! Instead of setting owner to an IRI, set it to an object and try to update the `username` to `cultured_cheese_head`. Go, go, go!

And... it doesn't work:

> Nested documents for attribute "owner" are not allowed. Use IRIs instead.

So... is this possible, or not?

Well, the *whole* reason that `username` is embedded when serializing a `CheeseListing` is that, above `username`, we've added the `cheese_listing:item:get` group, which is one of the groups that's used in the "get" item operation.

The same logic is used when *writing* a field, or, denormalizing it. If we want `username` to be writable while denormalizing a `CheeseListing`, we need to put it in a group that's *used* during denormalization. In this case, that's `cheese_listing:write`.

Copy that and paste it above `username`.

186 lines | src/Entity/User.php

```
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 51
    /**
... line 53
     * @Groups({"user:read", "user:write", "cheese_listing:item:get", "cheese_listing:write"})
... line 55
     */
    private $username;
... lines 58 - 184
}
```

As *soon* as we do that - because the `owner` property already has this group - the embedded `username` property can be written! Let's go back and try it: we're still trying to pass an *object* with `username`. Execute!

Sending New Objects vs References in JSON

And... oh... it *still* doesn't work! But the error is fascinating!

> A new entity was found through the relationship `CheeseListing.owner` that was not configured to cascade persist operations for entity User.

If you've been around Doctrine for awhile, you might recognize this strange error. Ignoring API Platform for a moment, it means that something created a *totally* new `User` object, *set* it onto the `CheeseListing.owner` property and then tried to save. But because nobody ever called `$entityManager->persist()` on the new `User` object, Doctrine panics!

So... yep! Instead of querying for the existing owner and *updating* it, API Platform took our data and used it to create a totally *new* `User` object! That's not what we wanted at all! How can we tell it to *update* the existing `User` object instead?

Here's the answer, or really, here's the simple rule: *if* we send an array of data, or really, an "object" in JSON, API Platform assumes that this is a new object and so... creates a new object. If you want to *signal* that you instead want to update an *existing* object, just add the `@id` property. Set it to `/api/users/2`. Thanks to this, API Platform will query for that user and *modify* it.

Let's try it again. It *works*! Well... it *probably* worked - it looks successful, but we can't see the username here. Scroll down and look for the user with id 2.

There it is!

Creating new Users?

So, we *now* know that, when updating... or really creating... a `CheeseListing`, we can send embedded `owner` data *and* signal to API Platform that it should update an existing `owner` via the `@id` property.

And when we *don't* add `@id`, it tries to create a *new* `User` object... which didn't work because of that persist error. But, we can *totally* fix that problem with a cascade persist... which I'll show in a few minutes to solve a different problem.

So wait... does this mean that, in theory, we *could* create a brand new `User` while editing a `CheeseListing`? The answer is.... yes! Well... *almost*. There are 2 things preventing it right now: first, the missing cascade persist, which gave us that big Doctrine error. And second, on `User`, we would also need to expose the `$password` and `$email` fields because these are both required in the database. When you start making embedded things writeable, it honestly adds complexity. Make sure you keep track of what and what is *not* possible in your API. I *don't* want users to be created accidentally while updating a `CheeseListing`, so this is perfect.

Embedded Validation

But, there is *one* weird thing remaining. Set `username` to an empty string. That shouldn't work because we have a `@NotBlank()` above `$username`.

Try to update anyways. Oh, of course! I get the cascade 500 error - let me put the `@id` property back on. Try it again.

Woh! A 200 status code! It looks like it worked! Go down and fetch this user... with id=2. They have no username! Gasp!

This... is a bit of a gotcha. When we modify the `CheeseListing`, the validation rules are executed: `@Assert\NotBlank()`, `@Assert\Length()`, etc. But when the validator sees the embedded `owner` object, it does *not* continue down into that object to validate *it*. That's *usually* what we want: if we were *only* updating a `CheeseListing`, why should it *also* try to validate a related `User` object that we didn't even modify? It shouldn't!

But when you're doing *embedded* object updates like we are, that changes: we *do* want validation to continue down into this object. To force that, above the `owner` property, add `@Assert\Valid()`.

199 lines | src/Entity/CheeseListing.php

```
... lines 1 - 39
class CheeseListing
{
... lines 42 - 86
    /**
... lines 88 - 90
     * @Assert\Valid()
     */
    private $owner;
... lines 94 - 197
}
```

Ok, go back, and... try our edit endpoint again. Execute. Got it!

> owner.username: This value should not be blank

Nice! Let's go back and give this a valid username... just so we don't have a bad user sitting in our database. Perfect!

Being able to make modifications on embedded properties is pretty cool... but it *does* add complexity. Do it if you need it, but also remember that we can update a `CheeseListing` and a `User` more simply by making two requests to two endpoints.

Next, let's get even *crazier* and talking about updating *collections*: what happens if we start to try to modify the `cheeseListings` property directly on a `User`?

Chapter 23: Adding Items to a Collection Property

Use the docs to check out the `User` with id=2. When we *read* a resource, we can decide to expose any property - and a property that holds a *collection*, like `cheeseListings`, is no different. We exposed that property by adding `@Groups("user:read")` above it. And because this holds a collection of related *objects*, we can *also* decide whether the `cheeseListings` property should be exposed as an array of IRI strings *or* as an array of embedded objects, by adding this same group to at least one property inside `CheeseListing` itself.

Great. New challenge! We can read the `cheeseListings` property on `User`... but could we also *modify* this property?

For example, well, it's a bit of a strange example, but let's pretend that an admin wants to be able to edit a `User` and make them the owner of some existing `CheeseListing` objects in the system. You can *already* do this by editing a `CheeseListing` and changing its `owner`. But could we also do it by editing a `User` and passing a `cheeseListings` property?

Actually, let's get even a bit *crazier*! I want to be able to create a new `User` *and* specify one or more cheese listings that this `User` should own... all in one request.

Making cheeseListings Modifiable

Right now, the `cheeseListings` property is not modifiable. The reason is simple: that property *only* has the read group. Cool! I'll make that group an array and add `user:write`.

```
186 lines | src/Entity/User.php
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 58
    /**
... line 60
     * @Groups({"user:read", "user:write"})
     */
    private $cheeseListings;
... lines 64 - 184
}
```

Now, go back, refresh the docs and look at the POST operation: we *do* have a `cheeseListings` property. Let's do this! Start with the boring user info: email, password doesn't matter and username. For `cheeseListings`, this needs to be an array... because this property *holds* an array. Inside, add just one item - an *IRI* - `/api/cheeses/1`.

In a perfect world, this will create a new `User` and *then* go fetch the `CheeseListing` with id `1` and change it to be owned by this user. Deep breath. Execute!

It worked? I mean, it worked! A 201 status code: it created the new `User` and that `User` now owns this `CheeseListing`! Wait a second... how *did* that work?

Adder and Remover Methods for Collections

Check it out: we understand how `email`, `password` and `username` are handled: when we POST, the serializer will call `setEmail()`. In this case, we're sending a `cheeseListings` field... but if we go look for `setCheeseListings()`, it doesn't exist!

Instead, search for `addCheeseListing()`. Ahhh. The `make:entity` command is smart: when it generates a collection relationship like this, instead of generating a `setCheeseListings()` method, it generates `addCheeseListing()` and `removeCheeseListing()`. And the serializer is smart enough to use those! It sees the one `CheeseListing` IRI we're sending, queries the database for that object, calls `addCheeseListing()` and passes it as an argument.

The *whole* reason `make:entity` generates the adder - instead of just `setCheeseListings()` - is that it lets us *do* things when a cheese listing is added or removed. And that is *key*! Check it out: inside the generated code, it calls `$cheeseListing->setOwner($this)`. *That* is the reason why the owner *changed* to the new user, for this `CheeseListing` with id=1.

Then... everything just saves!

Next: when we're creating or editing a user, instead of *reassigning* an existing `CheeseListing` to a new owner, let's make it possible to create totally *new* cheese listings. Yep, we're getting crazy! But this will let us learn even *more* about how the serializer thinks and works.

Chapter 24: Creating Embedded Objects

Instead of assigning an existing `CheeseListing` to the user, could we create a totally new one by embedding its data? Let's find out!

This time, we won't send an IRI string, we'll send an *object* of data. Let's see... we need a `title` and... I'll cheat and look at the `POST` endpoint for cheeses. Right: we need `title`, `price` `owner` and `description`. Set `price` to 20 bucks and pass a `description`. But I'm not going to send an `owner` property. Why? Well... forget about API Platform and just imagine you're *using* this API. If we're sending a POST request to `/api/users` to create a new user... isn't it pretty obvious that we want the new cheese listing to be owned by *this* new user? Of course, it's our job to actually make this *work*, but this *is* how I would *want* it to work.

Oh, and before we try this, change the `email` and `username` to make sure they're unique in the database.

Ready? Execute! It works! No no, I'm totally lying - it's not *that* easy. We've got a familiar error:

> Nested documents for attribute "cheeseListings" are not allowed. Use IRIs instead.

Allowing Embedded cheeseListings to be Denormalized

Ok, let's back up. The `cheeseListings` field is *writable* in our API because the `cheeseListings` property has the `user:write` group above it. But if we did *nothing* else, this would mean that we can pass an array of IRIs to this property, but *not* a JSON object of embedded data.

To allow *that*, we need to go into `CheeseListing` and add that `user:write` group to all the properties that we want to allow to be passed. For example, we know that, in order to create a `CheeseListing`, we need to be able to set `title`, `description` and `price`. So, let's add that group! `user:write` above `title`, `price` and... down here, look for `setTextDescription()` ... and add it there.

199 lines | src/Entity/CheeseListing.php

```
... lines 1 - 39
class CheeseListing
{
... lines 42 - 48
    /**
... line 50
     * @Groups({"cheese_listing:read", "cheese_listing:write", "user:read", "user:write"})
... lines 52 - 57
     */
    private $title;
... lines 60 - 67
    /**
... lines 69 - 71
     * @Groups({"cheese_listing:read", "cheese_listing:write", "user:read", "user:write"})
... line 73
     */
    private $price;
... lines 76 - 134
    /**
... lines 136 - 137
     * @Groups({"cheese_listing:write", "user:write"})
... line 139
     */
    public function setTextDescription(string $description): self
... lines 142 - 197
}
```

I *love* how clean it is to choose which fields you want to allow to be embedded... but life *is* getting more complicated. Just keep that "complexity" cost in mind if you decide to support this kind of stuff in your API

Cascade Persist

Anyways, let's try it! Ooh - a 500 error. We're closer! And we know this error too!

> A new entity was found through the `User.cheeseListings` relation that was not configured to cascade persist.

Excellent! This tells me that API Platform *is* creating a new `CheeseListing` and it *is* setting it onto the `cheeseListings` property of the new `User`. But nothing ever calls `$entityManager->persist()` on that new `CheeseListing`, which is why Doctrine isn't sure what to do when trying to save the User.

If this were a traditional Symfony app where I'm personally writing the code to create and save these objects, I'd probably just find where that `CheeseListing` is being created and call `$entityManager->persist()` on it. But because API Platform is handling all of that for us, we can use a different solution.

Open `User`, find the `$cheeseListings` property, and add `cascade={"persist"}`. Thanks to this, whenever a `User` is persisted, Doctrine will automatically persist any `CheeseListing` objects in this collection.

```
186 lines | src/Entity/User.php
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 58
    /**
     * @ORM\OneToMany(targetEntity="App\Entity\CheeseListing", mappedBy="owner", cascade={"persist"})
... line 61
     */
    private $cheeseListings;
... lines 64 - 184
}
```

Ok, let's see what happens. Execute! Woh, it worked! This created a new `User`, a new `CheeseListing` *and* linked them together in the database.

But who set CheeseListing.owner?

But... how did Doctrine... or API Platform know to set the `owner` property on the new `CheeseListing` to the new `User`... if we didn't pass an `owner` key in the JSON? If you create a `CheeseListing` the *normal* way, that's totally required!

This works... *not* because of any API Platform or Doctrine magic, but thanks to some good, old-fashioned, well-written code in our entity. Internally, the serializer instantiated a new `CheeseListing`, set data on it and *then* called `$user->addCheeseListing()`, passing that new object as the argument. And *that* code takes care of calling `$cheeseListing->setOwner()` and setting it to `$this` User. I *love* that: our generated code from `make:entity` and the serializer are working together. What's gonna work? Team work!

Embedded Validation

But, like when we embedded the `owner` data while editing a `CheeseListing`, when you allow embedded resources to be changed or created like this, you need to pay special attention to validation. For example, change the `email` and `username` so they're unique again. This is now a valid user. But set the `title` of the `CheeseListing` to an empty string. Will validation stop this?

Nope! It *allowed* the `CheeseListing` to save with no title, *even* though we have validation to prevent that! That's because, as we talked about earlier, when the validator processes the `User` object, it doesn't automatically cascade down into the `cheeseListings` array and *also* validate those objects. You can force that by adding `@Assert\Valid()`.

```
187 lines | src/Entity/User.php
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 58
    /**
... lines 60 - 61
     * @Assert\Valid()
     */
    private $cheeseListings;
... lines 65 - 185
}
```

Let's make sure that did the trick: go back up, bump the `email` and `username` to be unique again and... Execute! Perfect! A 400 status code because:

> the `cheeseListings[0].title` field should not be blank.

Ok, we've talked about how to *add* new cheese listings to an user - either by passing the IRI of an existing `CheeseListing` or embedding data to create a *new* `CheeseListing`. But what would happen if a user had 2 cheese listings... and we made a request to edit that `User`... and only included the IRI of *one* of those listings? That should... *remove* the missing `CheeseListing` from the user, right? Does that work? And if so, does it set that CheeseListing's `owner` to null? Or does it delete it entirely? Let's find some answers next!

Chapter 25: Removing Items from a Collection

Close up the POST operation. I want to make a GET request to the collection of users. Let's see here - the user with id 4 has one `CheeseListing` attached to it - id 2. Ok, close up that operation and open up the operation for `PUT`: I want to *edit* that User. Enter 4 for the id.

First, I'm going to do something that we've already seen: let's *just* update the `cheeseListings` field: set it to an array with one IRI inside: `/api/cheeses/2`. If we did *nothing* else, this would set this property to... *exactly* what it already equals: user id 4 already has this *one* `CheeseListing`.

But now, add *another* IRI: `/api/cheeses/3`. That already exists, but is owned by another user. When I hit Execute.... pfff - I get a syntax error, because I left an extra comma on my JSON. Boo Ryan. Let's... try that again. This time... bah! A 400 status code:

> This value should not be blank

My experiments with validation just came back to bite me! We set the `title` for `CheeseListing` 3 to an empty string in the database... it's basically a "bad" record that snuck in when we were playing with embedded validation. We could fix that title.. or... just change this to `/api/cheeses/1`. Execute!

The Serializer only Calls Adders for New Items

This time, it works! But, no surprise - we've basically done this! Internally, the serializer sees the existing `CheeseListing` IRI - `/api/cheeses/2`, realizes that this is already set on our `User`, and... does nothing. I mean, maybe it goes and gets a coffee or takes a walk. But, it most definitely does *not* call `$user->addCheeseListing()`... or really do anything. But when it sees the *new* IRI - `/api/cheeses/1`, it figures out that this `CheeseListing` does *not* exist on the `User` yet, and so, it *does* call `$user->addCheeseListing()`. That's why adder and remover methods are so handy: the serializer is smart enough to *only* call them when an object is *truly* being added or removed.

Removing Items from a Collection

Now, let's do the *opposite*: pretend that we want to *remove* a `CheeseListing` from this `User` - remove `/api/cheeses/2`. What do you think will happen? Execute and... woh! An integrity constraint error!

> An exception occurred when executing UPDATE cheese_listing SET owner_id=NULL - column `owner_id` cannot be null.

This is cool! The serializer *noticed* that we *removed* the `CheeseListing` with id = 2. And so, it *correctly* called `$user->removeCheeseListing()` and passed `CheeseListing` id 2. Then, our generated code set the owner on that `CheeseListing` to null.

Depending on the situation and the nature of the relationship and entities, this might be *exactly* what you want! Or, if this were a ManyToMany relationship, the result of *that* generated code would basically be to "unlink" the two objects.

orphanRemoval

But in our case, we don't *ever* want a `CheeseListing` to be an "orphan" in the database. In fact... that's exactly why we made `owner` `nullable=false` and why we're seeing this error! Nope, if a `CheeseListing` is removed from a `User`... I guess we really need to just delete that `CheeseListing` *entirely*!

And... yea, doing that is easy! All the way back up above the `$cheeseListings` property, add `orphanRemoval=true`.

```
187 lines | src/Entity/User.php
... lines 1 - 22
class User implements UserInterface
{
... lines 25 - 58
    /**
     * @ORM\OneToMany(targetEntity="App\Entity\CheeseListing", mappedBy="owner", cascade={"persist"}, orphanRemoval=true)
... lines 61 - 62
     */
    private $cheeseListings;
... lines 65 - 185
}
```

This means, *if* any of the `CheeseListings` in this array suddenly... are *not* in this array, Doctrine will delete them. Just, realize that if you try to *reassign* a `CheeseListing` to another `User`, it will *still* delete that `CheeseListing`. So, just make sure you only use this when that's *not* a use-case. We've been changing the owner of cheese listings a bunch... but only as an example: it doesn't *really* make sense, so this is perfect.

Execute one more time. It works... and *only* `/api/cheeses/1` is there. And if we go *all* the way back up to fetch the collection of cheese listings... yea, `CheeseListing` id 2 is gone.

Next, when you combine relations and filtering... well... you get some *pretty* serious power.

Chapter 26: Filtering on Relations

Go directly to `/api/users/5.jsonld`. This user owns one `CheeseListing`... and we've decided to embed the `title` and `price` fields instead of just showing the IRI. Great!

Earlier, we talked about a really cool filter called `PropertyFilter`, which allows us to, for example, add `?properties[]=username` to the URL if we *only* want to get back that *one* field. We added that to `CheeseListing`, but not `User`. Let's fix that!

Above `User`, add `@ApiFilter(PropertyFilter::class)`. And remember, we need to manually add the `use` statement for filter classes: `use PropertyFilter`.

190 lines | src/Entity/User.php

```
... lines 1 - 6
use ApiPlatform\Core\Serializer\Filter\PropertyFilter;
... lines 8 - 15
/**
... lines 17 - 20
 * @ApiFilter(PropertyFilter::class)
... lines 22 - 24
 */
class User implements UserInterface
... lines 27 - 190
```

And... we're done! When we refresh, it works! Other than the standard JSON-LD properties, we *only* see `username`.

Selecting Embedded Relation Properties

But wait there's more! Remove the `?properties[]=` part for a second so we can see the full response. What if we wanted to fetch only the `username` property and the `title` property of the embedded `cheeseListings`? Is that possible? Totally! You just need to know the syntax. Put back the `?properties[]=username`. *Now* add `&properties[`, but inside of the square brackets, put `cheeseListings`. Then `[]=` and the property name: `title`. Hit it! Nice! Well, the `title` is empty on this `CheeseListing`, but you get the idea. The point is this: `PropertyFilter` kicks butt and can be used to filter embedded data without any extra work.

Searching on Related Properties

Speaking of filters, we gave `CheeseListing` a *bunch* of them, including the ability to search by `title` or `description` and filter by `price`. Let's add another one.

Scroll to the top of `CheeseListing` to find `SearchFilter`. Let's break this onto multiple lines.

202 lines | src/Entity/CheeseListing.php

```
... lines 1 - 16
/**
... lines 18 - 34
 * @ApiFilter(SearchFilter::class, properties={
 *     "title": "partial",
 *     "description": "partial"
 * })
... lines 39 - 41
 */
class CheeseListing
... lines 44 - 202
```

Searching by `title` and `description` is great. But what if I want to search by *owner*: find all the `CheeseListings` owned by a specific `User`? Well, we can already do this a different way: fetch that user's data and look at its `cheeseListings` property. But having it as a filter might be super useful. Heck, then we could search for all cheese listings owned by a specific user *and* that match some title! And... if users start to have *many* `cheeseListings`, we might decide *not* to expose that property on `User` at all: the list might be too long. The advantage of a filter is that we can get all the cheese listings for a user in a paginated collection.

To do this... add `owner` set to `exact`.

```
203 lines | src/Entity/CheeseListing.php
... lines 1 - 16
/**
... lines 18 - 34
 * @ApiFilter(SearchFilter::class, properties={
... lines 36 - 37
 *     "owner": "exact"
 * })
... lines 40 - 42
 */
class CheeseListing
... lines 45 - 203
```

Go refresh the docs and try the GET endpoint. Hey! We've got a new filter box! We can even find by *multiple* owners. Inside the box, add the *IRI* - `/api/users/4`. You *can* also filter by `id`, but the IRI is recommended.

Execute and... yes! We get the *one* `CheeseListing` for that `User`. And the syntax on the URL is *beautifully* simple: `?owner=` and the IRI... which only looks ugly because it's URL-encoded.

Searching Cheese Listings by Owner Username

But we can get even crazier! Add one more filter: `owner.username` set to `partial`.

```
204 lines | src/Entity/CheeseListing.php
... lines 1 - 16
/**
... lines 18 - 34
 * @ApiFilter(SearchFilter::class, properties={
... lines 36 - 38
 *     "owner.username": "partial"
 * })
... lines 41 - 43
 */
class CheeseListing
... lines 46 - 204
```

This is pretty sweet. Refresh the docs again and open up the collection operation. Here's our new filter box, for `owner.username`. Check this out: Search for "head" because we have a bunch of cheesehead usernames. Execute! This finds two cheese listings owned by users 4 and 5.

Let's fetch all the users... just to be sure and... yep! Users 4 and 5 match that username search. Let's try searching for this `cheesehead3` exactly. Put that in the box and... Execute! Got it! The exact search works too. And, even though we're filtering *across* a relationship, the URL is pretty clean: `owner.username=cheesehead3`.

Ok just *one* more short topic for this part of our tutorial: subresources.

Chapter 27: Subresources

At some point, an API client... which might just be our JavaScript, will probably want to get a list of all of the `cheeseListings` for a specific `User`. And... we can already do this in two different ways: search for a specific owner here via our filter... or fetch the specific `User` and look at its `cheeseListings` property.

If you think about it, a `CheeseListing` *almost* feels like a "child" resource of a `User`: cheese listings *belong* to users. And for that reason, some people might *like* to be able to fetch the cheese listings for a user by going to a URL like this: `/api/users/4/cheeses`... or something similar.

But... that doesn't work. This idea is called a "subresource". Right now, each resource has its own, sort of, base URL: `/api/cheeses` and `/api/users`. But it *is* possible to, kind of, "move" cheeses *under* users.

Here's how: in `User`, find the `$cheeseListings` property and add `@ApiSubresource`.

```
192 lines | src/Entity/User.php
... lines 1 - 6
use ApiPlatform\Core\Annotation\ApiSubresource;
... lines 8 - 26
class User implements UserInterface
{
... lines 29 - 62
    /**
... lines 64 - 66
     * @ApiSubresource()
     */
    private $cheeseListings;
... lines 70 - 190
}
```

Let's go refresh the docs! Woh! We have a new endpoint! `/api/users/{id}/cheese_listings`. It shows up in two places... because it's kind of related to users... and kind of related to cheese listings. The URL is *cheese_listings* by default, but that can be customized.

So... let's try it! Change the URL to `/cheese_listings`. Oh, and add the `.jsonld` on the end. There it is! The collection resource for all cheeses that are owned by this `User`.

Subresources are kinda cool! But... they're also a bit unnecessary: we *already* added a way to get the collection of cheese listings for a user via the `SearchFilter` on `CheeseListing`. And using subresources means that you have more endpoints to keep track of, and, when we get to security, more endpoints means more access control to think about.

So, use subresources if you want, but I don't recommend adding them *everywhere*, there *is* a cost from added complexity. Oh, and by the way, there is a *ton* of stuff you can customize on subresources, like normalization groups, the URL, etc. It's all in the docs and it's pretty similar to the types of customizations we've seen so far.

For our app, I'm going to remove the subresource to keep things simple.

And... we're done! Well, there is a *lot* more cool stuff to cover - including security! That's the topic of the next tutorial in this series. But give yourself a jumping high-five! We've already unlocked a *huge* amount of power! We can expose entities as API resources, customize the operations, take *full* control of the serializer in a *bunch* of different ways and a *ton* more. So start building your gorgeous new API, tell us about it and, as always, if you have questions, you can find us in the comments section.

Alright friends, seeya next time!