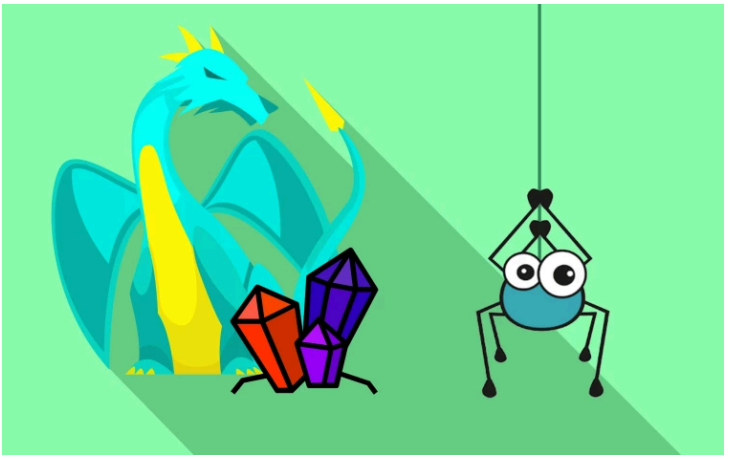
API Platform 3 Part 3: Custom Resources



Chapter 1: Setup & Ways to Extend API Platform

Fasten your scales, dragon enthusiasts! It's time to dive into the third episode of our riveting API Platform saga: the episode where things get... let's say: more advanced and interesting.

Episode 1 was our intro, and we covered a lot: pagination, filtering and a ton about serialization: how our API resource objects are turned into JSON and how the JSON sent by the user is turned *back* into those same objects.

Episode 2 was about security and included things like state processors - the key to running code before or after saving - custom fields, validation, voters, and more.

Custom Api Classes?

That's *all* good stuff. But, so far, all of our `#[ApiResource]` classes have been Doctrine entities. And that's fine! But as your API starts to look different from your entities, making that work adds complexity: serialization groups, extending normalizers, etc. At some point, it becomes easier and clearer to stop using your entity directly for your API and, instead, create a dedicated class. *That* is the biggest focus of this tutorial... and it'll take us deep into the concept of state providers and processors... which are basically the core to everything.

Project Setup

All right people, let's do this! I recommend POSTing up and coding along with me: it's more fun, and you'll get more out of this. Download the course code from this page and, when you unzip it, you'll find a `start/` directory with the same code that I have here - including the all-important `README.md` file, which contains all the deets to get this tutorial running.

The last step is to spin over, open a terminal into the project, and run

```
symfony serve -d
```

to start the built-in web server at https://127.0.0.1:8000. Say hello to: Treasure Connect! This is the same app we built in episodes one and two. I *have* made a few small changes - including fixing a few deprecations - but nothing major.

The most important page is `/api` where we can see our two API resources: Treasure and User. And we made these fairly complex! We have sub-resources, custom fields, complex security, etc. But again, for both `DragonTreasure` and `User`, the `#[ApiResource]` attribute is above an *entity* class. In a bit, we'll re-create this *same* API setup, but with dedicated classes.

src/Entity/User.php

```
↕ // ... lines 1 - 27
28 #[ApiResource(
↕ // ... line 29
30     operations: [
31         new Get(),
32         new GetCollection(),
↕ // ... lines 33 - 43
44     ],
45     normalizationContext: ['groups' => ['user:read']],
46     denormalizationContext: ['groups' => ['user:write']],
↕ // ... lines 47 - 50
51 )]
52 #[ApiResource(
53     uriTemplate: '/treasures/{treasure_id}/owner.{_format}',
54     operations: [new Get()],
↕ // ... lines 55 - 65
66 )]
↕ // ... lines 67 - 69
70 class User implements UserInterface, PasswordAuthenticatedUserInterface
↕ // ... lines 71 - 307
```

Custom Controllers? Event Listeners?

Before we hop in, I'm going to search for "API platform extending" to find one of my favorite pages on the API Platform documentation. It answers a simple but powerful question: what are all the different ways that I can extend API platform? For example, state processors are the best way to run code before or after you save something: a topic we talked about in the last tutorial.

So, this page is *great* and I want you to know about it. But I'm also here to mention a couple of things that we are *not* going to talk about. First, we are *not* going to talk about building

operations with custom controllers. Heck, that's not even in this list! The reason: there's always a better way - a different extension point - to do that. For example, you might create a custom operation or even a custom ApiResource class with a state processor that allows you to do whatever weird work your custom operation needs.

We're also *not* going to talk about event listeners: these kernel events. It's for the same reason: there are different extensions points we can use. These events also only work for REST: they won't work for GraphQL. And... it looks like the next version of API Platform - version 3.2 - may even *remove* these events in favor of a new internal system that leverages state providers and state processors even more.

Ok team: time to get to work. Next, let's use a state provider to add a totally custom field to one of our API resources. But unlike when we did this in the previous tutorial, this field will be properly documented in our API.

Chapter 2: State Providers, Processors & a Custom Field

API Platform 3 rolled out snazzy new concepts called State Providers and State Processors. We chatted about them in the last tutorial and we're going to dive even *deeper* in this tutorial.

Providers & Processors Basics

Nestled within the "Upgrade Guide" of API Platform's docs lives one of my favorite sections on this very topic. Each API resource class - whether it's an entity or a normal class - will have a State Provider. Its job is to *load* the data, like from the database... or wherever. Each API resource class will *also* have a State *Processor* whose jobs is to *save* the data, like on a POST or PATCH request. It's also responsible for *deleting*.

The big bonus is that if your API resource is an entity, you *automatically* get a set of State Providers and State Processors. For example, the `GetCollection` operation uses a core `CollectionProvider`, which queries the database for you. And there's a similar `ItemProvider` to fetch *one* item from the database.

Entities also gets a complimentary `PersistProcessor`, which, no surprise, persists your data to the database.

In Episode 2, we decorated the `PersistProcessor` for the `User` entity. This let us hash the plain password up here... before calling the core `PersistProcessor` to handle the saving.

src/State/UserHashPasswordStateProcessor.php

```
// ... lines 1 - 11
class UserHashPasswordStateProcessor implements ProcessorInterface
{
// ... lines 14 - 17
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        if ($data instanceof User && $data->getPlainPassword()) {
            $data->setPassword($this->userPasswordHasher->hashPassword($data, $data->getPlainPassword()));
        }

        $this->innerProcessor->process($data, $operation, $uriVariables, $context);
    }
}
```

Good & Better Ways to Add a Custom Field

We're talking about this because we can use a *similar* trick with the state *provider* to add a custom field: a field that you want in your API, but that doesn't live in the database.

In the last episode, we learned that one way to add a custom field is by extending the normalizer. We did this in `AddOwnerGroupsNormalizer`. Well, this does a *few* things, but importantly for us: if the object is a `DragonTreasure` - so if a `DragonTreasure` is being turned into JSON - *and* the currently authenticated user is the owner of that treasure, then add a totally custom `isMine` field.

src/Normalizer/AddOwnerGroupsNormalizer.php

```
// ... lines 1 - 12
class AddOwnerGroupsNormalizer implements NormalizerInterface, SerializerAwareInterface
{
// ... lines 15 - 18
    public function normalize(mixed $object, string $format = null, array $context = []): array|string|int|float|bool|\ArrayObject|null
    {
        if ($object instanceof DragonTreasure && $this->security->getUser() === $object->getOwner()) {
            $context['groups'][] = 'owner:read';
        }

        $normalized = $this->normalizer->normalize($object, $format, $context);

        if ($object instanceof DragonTreasure && $this->security->getUser() === $object->getOwner()) {
            $normalized['isMine'] = true;
        }

        return $normalized;
    }
// ... lines 33 - 54
}
```

We can see this in our tests: `tests/Functional/DragonTreasureResourceTest.php` Search for `isMine`. Yep: `testOwnerCanSeeIsPublishedAndIsMineFields`. The important part is the bottom: when the treasure is serialized, `isMine` should be in the response.

tests/Functional/DragonTreasureResourceTest.php

```
↕ // ... lines 1 - 13
14 class DragonTreasureResourceTest extends ApiTestCase
15 {
↕ // ... lines 16 - 196
197     public function testOwnerCanSeeIsPublishedAndIsMineFields(): void
198     {
↕ // ... lines 199 - 204
205         $this->browser()
206             ->actingAs($user)
207             ->patch('/api/treasures/'.$treasure->getId(), [
208                 'json' => [
209                     'value' => 12345,
210                 ],
211             ])
212             ->assertStatus(200)
213             ->assertJsonMatches('value', 12345)
214             ->assertJsonMatches('isPublished', true)
215             ->assertJsonMatches('isMine', true)
216         ;
217     }
218 }
```

This works great... except for one hiccup: in the documentation... there is *no* mention of the `isMine` field! It *will* be returned, but it's not documented.

If this matters to you, there are two better ways to handle this: add a non-persisted field to your entity - that's what we'll do in a moment - or create a totally custom API resource class. *That* will be our big topic later.

Adding the Non-Persisted Field

Step 1: remove the code in the normalizer... and just return. Copy the test method name... to make sure this fails:

src/Normalizer/AddOwnerGroupsNormalizer.php

```
// ... lines 1 - 12
class AddOwnerGroupsNormalizer implements NormalizerInterface, SerializerAwareInterface
{
// ... lines 15 - 18
    public function normalize(mixed $object, string $format = null, array $context = []): array|string|int|float|bool|\ArrayObject|null
    {
        if ($object instanceof DragonTreasure && $this->security->getUser() === $object->getOwner()) {
            $context['groups'][] = 'owner:read';
        }

        return $this->normalizer->normalize($object, $format, $context);
    }
// ... lines 27 - 48
}
```

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

And... yay failure! Expected `null` to be the same as `true` from line 215... because no more `isMine` field!

Step 2: add this field as a real property on our class: how about `private bool $isOwnedByAuthenticatedUser`. Notice this is a non-persisted property: it only exists to help our API. Doing this isn't super common, but *is* allowed. Skip down to the bottom to add a getter and setter.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 88
89 class DragonTreasure
90 {
↕ // ... lines 91 - 139
140     /**
141      * @var bool Non-persisted property to help determine if the treasure is owned by the authenticated user
142      */
143     private bool $isOwnedByAuthenticatedUser;
↕ // ... lines 144 - 256
257     public function isOwnedByAuthenticatedUser(): bool
258     {
259         return $this->isOwnedByAuthenticatedUser;
260     }
261
262     public function setIsOwnedByAuthenticatedUser(bool $isOwnedByAuthenticatedUser)
263     {
264         $this->isOwnedByAuthenticatedUser = $isOwnedByAuthenticatedUser;
265     }
266 }
```

Oh, and since the property doesn't have a default value, if the property hasn't been initialized, let's yell so we know.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 256
257     public function isOwnedByAuthenticatedUser(): bool
258     {
259         if (!isset($this->isOwnedByAuthenticatedUser)) {
260             throw new \LogicException('You must call setIsOwnedByAuthenticatedUser() before isOwnedByAuthenticatedUser()');
261         }
↕ // ... lines 262 - 263
264     }
↕ // ... lines 265 - 271
```

Last but not least, we need to expose this property to our API. Do that by putting it into the group called `treasure:read`... and then use `SerializedName` to call it `isMine` in the API.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 256
257     #[Groups(['treasure:read'])]
258     #[SerializedName('isMine')]
259     public function isOwnedByAuthenticatedUser(): bool
260     {
↕ // ... lines 261 - 265
266     }
↕ // ... lines 267 - 273
```

If we go run the test now:

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

We're greeted with a delicious 500 error! Thanks to the `zenstruck/browser` library, it saved that failed response to a file... which we can pop open in our browser. And... yup!

> *"You must call setIsOwnedByAuthenticatedUser()"*

So it's *trying* to expose the field to our API... but nothing is *setting* that property. How *will* we set it? With a positive attitude! And... mostly a custom state provider. That's next.

Chapter 3: Decorating the Core State Provider

To populate the non-persisted property on our entity, we'll leverage a custom state provider. Create one with:

```
php bin/console make:state-provider
```

Let's dub it `DragonTreasureStateProvider`.

Spin over and open this up in `src/State/`. Ok, it implements a `ProviderInterface` which requires one method: `provide()`. Our job is to return the `DragonTreasure` object for the current API request - which is a `Patch` request in our test.

src/State/DragonTreasureStateProvider.php

```
// ... lines 1 - 7
class DragonTreasureStateProvider implements ProviderInterface
{
    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
        // Retrieve the state from somewhere
    }
}
```

Before we think about doing that, `dd($operation)` so we can see if this is executed. When we try the test... the answer is that it is *not* called. We get the same error as before.

src/State/DragonTreasureStateProvider.php

```
// ... lines 1 - 9
    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
        dd($operation);
    }
// ... lines 14 - 15
```

So, creating a state provider and implementing `ProviderInterface` is *not* enough to make our class be used. And this is great! *We* get to control this on an resource-by-resource basis... or even on an operation-by-operation basis.

In `DragonTreasure`, way up on top, inside the `ApiResource` attribute, add `provider` then the service ID, which is the class in our case: `DragonTreasureStateProvider::class`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 19
20 use App\State\DragonTreasureStateProvider;
↕ // ... lines 21 - 30
31 #[ApiResource(
↕ // ... lines 32 - 64
65     provider: DragonTreasureStateProvider::class,
↕ // ... lines 66 - 68
69 )]
↕ // ... lines 70 - 90
91 class DragonTreasure
↕ // ... lines 92 - 275
```

So now, whenever API Platform needs to "load" a dragon treasure, it will call our provider. And our test is a perfect example. When we make a `PATCH` request, the first thing API Platform will do is ask the state provider to load this treasure. Then it will update it using the JSON.

Watch, when we run the test now:

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

We hit the dump!

Decorating the Provider

But... I *don't* want to do all the work of querying the database for the dragon treasures... because there's already a core entity provider that does all that! So let's use it!

Add a constructor... oh and I'll keep that `dd()` for now. Add a private `ProviderInterface $itemProvider` argument.

src/State/DragonTreasureStateProvider.php

```
// ... lines 1 - 5
use ApiPlatform\State\ProviderInterface;
// ... line 7
class DragonTreasureStateProvider implements ProviderInterface
{
    public function __construct(private ProviderInterface $itemProvider)
    {
    }
// ... lines 13 - 17
}
```

As a reminder: the `Get` one, `Patch`, `Put` and `Delete` operations all use the `ItemProvider`, which knows to query for a *single* item. Since our test uses `Patch`, we're going to focus on using *that* provider first.

If we run the test now, it fails. The error is:

> *"Cannot autowire service `DragonTreasureStateProvider`: argument `itemProvider` references `ProviderInterface`, but no such service exists."*

Often in Symfony, if we type-hint an interface, Symfony will pass us what we need. But in the case of `ProviderInterface`, there are *multiple* services that implement this - including the core `ItemProvider` and `CollectionProvider`.

This means that we need to *tell* Symfony which we want. Do that with the handy-dandy `#[Autowire]` attribute with `service` set to `ItemProvider::class` - make sure to get the one from `ORM`.

src/State/DragonTreasureStateProvider.php

```
// ... lines 1 - 7
use Symfony\Component\DependencyInjection\Attribute\Autowire;
// ... line 9
class DragonTreasureStateProvider implements ProviderInterface
{
    public function __construct(
        #[Autowire(service: ItemProvider::class)] private
ProviderInterface $itemProvider
    )
    {
    }
// ... lines 17 - 21
}
```

And yup! That *is* a valid service id. There is also a harder-to-remember service id, but API Platform provides a service alias so that we can just use this. Lovely!

Ok, go test go! Yes! We hit the dump which means that the item provider *was* injected. So now, we're dangerous. `$treasure` equals `$this->itemProvider->provide()` passing the 3 args.

src/State/DragonTreasureStateProvider.php

```
↕ // ... lines 1 - 18
19     public function provide(Operation $operation, array $uriVariables =
[], array $context = []): object|array|null
20     {
21         $treasure = $this->itemProvider->provide($operation,
$uriVariables, $context);
↕ // ... lines 22 - 29
30     }
↕ // ... lines 31 - 32
```

At this point, `$treasure` will be `null` or a valuable `DragonTreasure` object. If it is *not* a `DragonTreasure` instance, return null.

But if we *do* have a treasure, we're in business! Call `setIsOwnedByAuthenticatedUser()` and hardcode true for now. Then return `$treasure`.

src/State/DragonTreasureStateProvider.php

```
↕ // ... lines 1 - 18
19     public function provide(Operation $operation, array $uriVariables =
[], array $context = []): object|array|null
20     {
↕ // ... lines 21 - 22
23         if (!$treasure instanceof DragonTreasure) {
24             return $treasure;
25         }
26
27         $treasure->setIsOwnedByAuthenticatedUser(true);
28
29         return $treasure;
30     }
↕ // ... lines 31 - 32
```

Ok, go test go!

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

Shazam! We're green! So let's go set that value for real. This is easy enough: add a `private Security` argument... and make sure you first arg has a comma.

Then this is true if `$this->security->getUser()` equals `$treasure->getOwner()`.

src/State/DragonTreasureStateProvider.php

```
// ... lines 1 - 11
class DragonTreasureStateProvider implements ProviderInterface
{
    public function __construct(
// ... line 15
        private Security $security,
    )
    {
    }

    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
// ... lines 23 - 28
        $treasure->setIsOwnedByAuthenticatedUser($this->security->getUser() === $treasure->getOwner());

        return $treasure;
    }
}
```

And... then... the test still passes. Custom field accomplished! *And*, most importantly, it *is* documented inside our API.

However, we *did* just break our `GetCollection` endpoint. Let's fix that next.

Chapter 4: Decorating the CollectionProvider

Let's boldly do something that scares most us developers: run the *entire* test suite:

```
symfony php bin/phpunit
```

These were obediently passing when I started the tutorial... but they've decided to rebel! Let's pop open the failed response. Hmm:

> *"More than one result was found for query, although one row or none was expected."*

If you view the page source, this is coming from Doctrine... and eventually the core `ItemProvider` that we're calling. Back on the docs, the `GetCollection` operation - which is the operation used in this test - has a *different* provider: `CollectionProvider`.

Unfortunately, when I set `provider` inside the `#[ApiResource]` attribute... that set the provider for *every* operation. It *is* possible to set the `provider` for a specific operation... like this. But... I *like* having a single provider for my entire API resource - it's simpler.

To make that happen, we just need to realize that this provider will be called both when fetching a *single* item and when fetching a *collection* of items. For this test, our provider is being called to fetch a collection... then we're calling the item provider... and weird stuff happens.

`dd()` the `$operation` again...

src/State/DragonTreasureStateProvider.php

```
↕ // ... lines 1 - 11
12 class DragonTreasureStateProvider implements ProviderInterface
13 {
↕ // ... lines 14 - 20
21     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
22     {
23         dd($operation);
↕ // ... lines 24 - 32
33     }
34 }
```

then copy the failing test name... and run just that one:

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

Excellent! A `GetCollection` object. We can use *that* to figure out which provider we need!

Let's get the core `CollectionProvider` injected. Copy the first argument, duplicate it, and set it to use the `CollectionProvider` service from ORM. Name it `$collectionProvider`.

src/State/DragonTreasureStateProvider.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Doctrine\Orm\State\CollectionProvider;
↕ // ... lines 6 - 13
14 class DragonTreasureStateProvider implements ProviderInterface
15 {
16     public function __construct(
↕ // ... line 17
18         #[Autowire(service: CollectionProvider::class)] private
   ProviderInterface $collectionProvider,
↕ // ... line 19
20     )
21     {
22     }
↕ // ... lines 23 - 39
40 }
```

Below, check to see if `$operation` is an instance of `CollectionOperationInterface`. Ok, really, only *one* operation - `GetCollection` - uses the collection provider... but in case a

custom operation were added, anything that needs a collection will implement this interface. In this situation, return `$this->collectionProvider->provide()` and pass in the args. And... don't forget the method name!

src/State/DragonTreasureStateProvider.php

```
↕ // ... lines 1 - 23
24     public function provide(Operation $operation, array $uriVariables =
[], array $context = []): object|array|null
25     {
26         if ($operation instanceof CollectionOperationInterface) {
27             return $this->collectionProvider->provide($operation,
$uriVariables, $context);
28         }
↕ // ... lines 29 - 38
39     }
↕ // ... lines 40 - 41
```

Alrighty! Spin over or run the test again:

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

And... it still explodes. Something about expected null to be the same as 5. Check the response. Ah! It's our error again! For the item operation, we *are* setting that property. Now, we need to do the same thing here: loop over each treasure and set that.

The Paginator Object

But first, what *does* the collection provider return - an array of treasures? Copy the entire call, `dd()` it... and run the test again:

src/State/DragonTreasureStateProvider.php

```
↕ // ... lines 1 - 23
24     public function provide(Operation $operation, array $uriVariables =
[], array $context = []): object|array|null
25     {
26         if ($operation instanceof CollectionOperationInterface) {
27             dd($this->collectionProvider->provide($operation,
$uriVariables, $context));
↕ // ... line 28
29         }
↕ // ... lines 30 - 39
40     }
↕ // ... lines 41 - 42
```

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

Let's see... it's a `Paginator` *object*! That's important: *that* is what powers the pagination for our collection endpoints. Ok, it's not actually *that* important right now - we can loop over this object to get each `DragonTreasure` - but we'll come back to this later when we create a custom resource.

Delete the `dd()` and, instead of the return, say `$paginator` equals. I'll help my editor by saying that this is an `iterable` of `DragonTreasure`. Now, `foreach` `$paginator` as `$treasure`... and then I'll steal the code from below... and paste.

Now that we've modified each item, `return $paginator`.

src/State/DragonTreasureStateProvider.php

```
↕ // ... lines 1 - 23
24      public function provide(Operation $operation, array $uriVariables =
        [], array $context = []): object|array|null
25      {
26          if ($operation instanceof CollectionOperationInterface) {
27              /** @var $paginator iterable<DragonTreasure> */
28              $paginator = $this->collectionProvider->provide($operation,
        $uriVariables, $context);
29
30              foreach ($paginator as $treasure) {
31                  $treasure->setIsOwnedByAuthenticatedUser($this->security-
        >getUser() === $treasure->getOwner());
32              }
33
34              return $paginator;
35          }
↕ // ... lines 36 - 45
46      }
↕ // ... lines 47 - 48
```

Let's try it again!

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

It fails *again*... but at the very end: `DragonTreasureResourceTest` line 37. Let's go check that out. So all the way up here, we create some treasures, make a `->get()` request to the collection endpoint, verify some things, and then, below, we grab the first item and check to make sure it has the right fields. Apparently the `isMine` property *is* there... but wasn't expected?

That's my bad. On a previous adventure, when we added the `isMine` property, we *only* added it when it was `true`. If a `DragonTreasure` did *not* belong to me, the field wasn't there at all... and it probably should have been. So let's update the test. And now... it's green!

tests/Functional/DragonTreasureResourceTest.php

```
↕ // ... lines 1 - 13
14 class DragonTreasureResourceTest extends ApiTestCase
15 {
↕ // ... lines 16 - 18
19     public function testGetCollectionOfTreasures(): void
20     {
↕ // ... lines 21 - 35
36         $this->assertSame(array_keys($json->decoded()['hydra:member'][0]),
   [
↕ // ... lines 37 - 45
46             'isMine',
47         ]);
48     }
↕ // ... lines 49 - 218
219 }
```

Re-run *everything*

```
symfony php bin/phpunit
```

POST: No State Provider

Uhhh. down to one failure: `testPostToCreateTreasure` - with a 500 error. Pop that open in our browser. Bah! It's our:

> *"You must call `setIsOwnedByAuthenticatedUser()`."*

But how is that possible? No matter what, we *are* setting that value inside our state provider! However... the `POST` operation is unique: it's the only operation that does *not* use a provider. Ok, `Delete` doesn't *show* a provider, but it uses the `ItemProvider` to load the one item it's about to delete.

For `Post`, the JSON is deserialized *directly* into a `TreasureEntity`.. then saved. The state provider is never needed or used.... which means when it serializes to JSON, that property is *still* not set.

The fix is in the state processor for `DragonTreasure`: right before or after saving, we need to run this same logic. Copy this. We *do* have a state processor already for `DragonTreasure`.

It's meant to set the owner if it's not set... but let's hijack it for this. Right after the save, paste that. Oh, but the way we created this in the previous episode means that it's called for *every* ApiResource. So we need the same if statement from up here: if `$data` is an `instanceof` `DragonTreasure`, then set that property. I'll... update a couple of variables.

src/State/DragonTreasureSetOwnerProcessor.php

```
// ... lines 1 - 11
class DragonTreasureSetOwnerProcessor implements ProcessorInterface
{
// ... lines 14 - 17
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
// ... lines 20 - 25
        if ($data instanceof DragonTreasure) {
            $data->setIsOwnedByAuthenticatedUser($data->getOwner() === $this->security->getUser());
        }
    }
}
```

So, the object saves, we set the property... and *then* it's serialized to JSON. Try those tests again:

```
symfony php bin/phpunit
```

All green! Woo! So we already know that we can run code before or after an item saves by having a custom state processor. But what if we need to run code only when something *specific* changes? Like when a `DragonTreasure` changes from unpublished to published. We'll dive into that next, starting with making our state processor a bit simpler.

Chapter 5: Simpler State Processor

Publishing a `DragonTreasure` is easy: make a `Patch` request to the treasure endpoint with `isPublished` set to true and... celebration! But... what if, when a `DragonTreasure` is published, we need to run some custom code - maybe trigger some notifications on the site.

One option is to create a custom operation - like maybe `POST /api/treasures/5/publish`. You *can* do that - and it might be fun to look at in a future tutorial. But who wants extra work? We can keep that simple `Patch` request and *still* run the code that we want. How? By using a state processor and *detecting* the change.

Let's start by creating a test that publishes a treasure. At the bottom, copy this last test, paste, and rename it `testPublishTreasure`. We start with a user that owns a treasure with `isPublished` `false`. Then we log in as that user, make a `->patch()` request to `/api/treasures/` using the id... and send `isPublished: true`. This should be a 200 status code... and then `->assertJsonMatches()` that `isPublished` is `true`.

tests/Functional/DragonTreasureResourceTest.php

```
↕ // ... lines 1 - 13
14 class DragonTreasureResourceTest extends ApiTestCase
15 {
↕ // ... lines 16 - 219
220     public function testPublishTreasure(): void
221     {
222         $user = UserFactory::createOne();
223         $treasure = DragonTreasureFactory::createOne([
224             'owner' => $user,
225             'isPublished' => false,
226         ]);
227 
228         $this->browser()
229             ->actingAs($user)
230             ->patch('/api/treasures/'.$treasure->getId(), [
231                 'json' => [
232                     'isPublished' => true,
233                 ],
234             ])
235             ->assertStatus(200)
236             ->assertJsonMatches('isPublished', true)
237         ;
238     }
239 }
```

Simple enough! Copy that test name, spin over and run it:

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

Whoops! It fails: expected `false` to be the same as `true`. That's from the last line: the JSON still has `isPublished` false. Maybe... the field isn't writable? Check the groups above that property. Ah: in a previous tutorial, we made this field writable by *admin* users, but not normal users. Add `treasure:write`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 90
91 class DragonTreasure
92 {
↕ // ... lines 93 - 130
131     #[Groups(['admin:read', 'admin:write', 'owner:read',
'treasure:write'])]
132     private bool $isPublished = false;
↕ // ... lines 133 - 273
274 }
```

That means anyone with access to the `Patch` operation can write to this field... which in reality, thanks to the `security` on that operation... and a custom voter we created... is just admin users and the owner.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 30
31 #[ApiResource(
↕ // ... lines 32 - 33
34     operations: [
↕ // ... lines 35 - 43
44         new Patch(
45             security: 'is_granted("EDIT", object)',
46         ),
↕ // ... lines 47 - 49
50     ],
↕ // ... lines 51 - 68
69 )]
↕ // ... lines 70 - 90
91 class DragonTreasure
↕ // ... lines 92 - 275
```

Try the test now:

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

Got it! To run some code when the treasure is published, we need a state processor. And we already have one for `DragonTreasure! We originally created it to set the owner to the currently authenticated user. So... should we jam the new code into here or create a second processor?

It's up to you, but I like to have *one* processor per resource class. It just makes my life simpler. But let's rename this class to be more clear: `DragonTreasureStateProcessor`.

Changing How Our State Processor Decorates

In the last tutorial, we learned that there are *two* ways to add a custom state provider or processor into the system. We used the first method a few minutes ago with the state provider: create a normal boring service... use `#[Autowire]` to inject the core services... then set the `provider` option on `DragonTreasure` to point to it.

The *other* way - which we did in the last tutorial for this class - is to *decorate* the core processor. Here, we decorated the `PersistProcessor` from Doctrine... which means that whenever *any* API resource is saved, when it tries to use the core `PersistProcessor`, *our* service is called instead. This was easy to set up because all we needed was `#[AsDecorator]` and... bam! Our service started being called for *all* our resources. But that's *also* why we need this extra code that checks *which* object is being saved.

src/State/DragonTreasureStateProcessor.php

```
↕ // ... lines 1 - 10
11 #[AsDecorator('api_platform.doctrine.orm.state.persist_processor')]
12 class DragonTreasureStateProcessor implements ProcessorInterface
13 {
↕ // ... lines 14 - 17
18     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
19     {
20         if ($data instanceof DragonTreasure && $data->getOwner() === null && $this->security->getUser()) {
21             $data->setOwner($this->security->getUser());
22         }
↕ // ... lines 23 - 28
29     }
30 }
```

Both ways are fine. But for consistency with the provider, let's refactor this to use the *other* method. This is 3 steps. First, remove `#[AsDecorator]`. Suddenly, instead of overriding a core service, this is a normal, boring service that *nobody* is using at the moment. Second, because we're no longer decorating a core service, Symfony won't know what to pass for `$innerProcessor`. Break this onto multiple lines... then use the `#[Autowire]` trick to point to the core `PersistProcessor`. And I'll clean up the old `use` statement.

src/State/DragonTreasureStateProcessor.php

```
↕ // ... lines 1 - 11
12 class DragonTreasureStateProcessor implements ProcessorInterface
13 {
14     public function __construct(
15         #[Autowire(service: PersistProcessor::class)]
16         private ProcessorInterface $innerProcessor,
17         private Security $security
18     )
19     {
20     }
↕ // ... lines 21 - 33
34 }
```

Step 3 is to tell API Platform *when* to use this processor. In `DragonTreasure`, we want this to be used for both our `Post` and `Patch` operations. Set `processor` to `DragonTreasureStateProcessor::class`... and repeat that down for `Patch`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 19
20 use App\State\DragonTreasureStateProcessor;
↕ // ... lines 21 - 31
32 #[ApiResource(
↕ // ... lines 33 - 34
35     operations: [
↕ // ... lines 36 - 41
42         new Post(
↕ // ... line 43
44             processor: DragonTreasureStateProcessor::class,
45         ),
46         new Patch(
↕ // ... line 47
48             processor: DragonTreasureStateProcessor::class,
49         ),
↕ // ... lines 50 - 71
72 )]
↕ // ... lines 73 - 93
94 class DragonTreasure
↕ // ... lines 95 - 278
```

Done! API Platform will call our processor... and it contains the core `PersistProcessor` so we can make it do the *real* work. Re-run the test to give us infinite confidence:

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

That feels *great*.

And the nice thing about doing the processor with *this* method is that you don't need this conditional code: this will *always* be a `DragonTreasure`. To help my editor and prove it, `assert()` that `$data` is an `instanceof` `DragonTreasure`.

src/State/DragonTreasureStateProcessor.php

```
// ... lines 1 - 11
class DragonTreasureStateProcessor implements ProcessorInterface
{
// ... lines 14 - 21
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        assert($data instanceof DragonTreasure);
// ... lines 25 - 29
    }
}
```

And my editor is already yelling:

> *“Hey this code down here isn't needed anymore dude!”*

So, remove that too. Now that we've refactored our state processor, let's get back to the task at hand: running custom code when a treasure becomes published.

Chapter 6: Running Code "On Publish"

Oh, quick, minor thing about state processors. The `make:state-processor` command created the `process()` method with a `void` return. And... that makes sense. API Platform passes us the data and our job is just to save that... not return anything.

However, *technically* the `process()` method *can* return something. And, for consistency, I *will* return something. Remove the `void` type and, at the bottom, return `$data`.

src/State/DragonTreasureStateProcessor.php

```
↕ // ... lines 1 - 11
12 class DragonTreasureStateProcessor implements ProcessorInterface
13 {
↕ // ... lines 14 - 21
22     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
23     {
↕ // ... lines 24 - 30
31         return $data;
32     }
33 }
```

I'll repeat this in `UserHashPasswordStateProcessor` for consistency.

src/State/UserHashPasswordStateProcessor.php

```
↕ // ... lines 1 - 11
12 class UserHashPasswordStateProcessor implements ProcessorInterface
13 {
↕ // ... lines 14 - 17
18     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
19     {
↕ // ... lines 20 - 23
24         return $this->innerProcessor->process($data, $operation, $uriVariables, $context);
25     }
26 }
```

Here's the deal: *if* you return something, *that* will be the "thing" that is ultimately serialized and returned as JSON. If you do *not* return anything, it will serialize `$data`. So, by returning

`$data`... we're not changing any behavior. But it's interesting to know that you *could* return something different.

Detecting Changes: previous_data vs UnitOfWork

Ok, back to our goal. After we save, we need to detect if the `isPublished` field *changed* from false to true, so we can run some custom code. But by the time the state processor is called, the JSON from the user has already been used to update the object. So `$data` will already have `isPublished` true.

In the last tutorial, we had a similar situation with a validator where we needed to check if the *owner* of a `DragonTreasure` had changed. This logic lives in `TreasureAllowToChangeValidator`. We start with `$value` - which is a collection of `DragonTreasure` objects, loop over them, then use Doctrine's `UnitOfWork` to see what each `DragonTreasure` looked like when it was *originally* loaded from the database.

src/Validator/TreasuresAllowedOwnerChangeValidator.php

```
↕ // ... lines 1 - 10
11 class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
12 {
↕ // ... lines 13 - 16
17     public function validate($value, Constraint $constraint): void
18     {
↕ // ... lines 19 - 27
28         $unitOfWork = $this->entityManager->getUnitOfWork();
29         foreach ($value as $dragonTreasure) {
↕ // ... lines 30 - 31
32             $originalData = $unitOfWork-
>getOriginalEntityData($dragonTreasure);
33             $originalOwnerId = $originalData['owner_id'];
34             $newOwnerId = $dragonTreasure->getOwner()->getId();
35
36             if (!$originalOwnerId || $originalOwnerId === $newOwnerId) {
37                 return;
38             }
39
40             // the owner is being changed
41             $this->context->buildViolation($constraint->message)
42                 ->addViolation();
43         }
44     }
45 }
```

Should we use that same trick here to see what the `isPublished` property originally looked like? We *could*... but there's an easier way!

API Platform has a concept of "previous data". When the request starts, API Platform *clones* the top-level object. So, if we're editing a `DragonTreasure`, it grabs that from the database using our state provider, clones it and, then keeps that "original" clone around in case it comes in handy. We can use *that* to see if the value of `isPublished` changed.

But wait, why didn't we just this "previous data" thing in the last tutorial for the validator? The reason is subtle. For the validator, the top-level object was a `User` object. When PHP clones an object, it's a "shallow" clone: any string, int or boolean properties are copied to the clone. But any *object* properties - like the `DragonTreasure` objects - are *not* copied: the clone and the original `User` objects both point to the *same* `DragonTreasure` objects in memory. So when the `owner` of those treasures is updated... that affected both the main object *and* the "previous object" clone. *That* is why we had to go deeper and use `UnitOfWork`.

But in *this* case, the `isPublished` property is a boring scalar boolean property. So if we can get the previous data, that will have the correct, original, `isPublished` value.

Great! So... how *do* we get the previous data? Notice we're passed an argument called `$context`... which is full of useful info. Let's `dd()` that.

src/State/DragonTreasureStateProcessor.php

```
↕ // ... lines 1 - 11
12 class DragonTreasureStateProcessor implements ProcessorInterface
13 {
↕ // ... lines 14 - 21
22     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
23     {
24         dd($context);
↕ // ... lines 25 - 32
33     }
34 }
```

Then copy the test name we're working on and... run it:

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

Oooo: a bunch of good stuff here. We have the current operation object... and here it is: `previous_data`. Check out that beautiful `isPublished` property: it's false!

Get rid of the `dd()`. At the bottom, say `$previousData = $context['previous_data']`. And, if it's not there - which will happen for a `POST` request - set it to `null`. I'll paste in the rest of the code that detects if `isPublished` changed from false to true. Actually... this is not the *best* code I've ever written - it's kinda confusing and won't let you publish *immediately* via a `POST`... but it'll work for our purposes. Inside, add a dump.

src/State/DragonTreasureStateProcessor.php

```
// ... lines 1 - 21
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
    {
// ... lines 24 - 30
        $previousData = $context['previous_data'] ?? null;
        if ($previousData instanceof DragonTreasure
            && $data->getIsPublished()
            && $previousData->getIsPublished() !== $data->getIsPublished()
        ) {
            dd('published!');
        }
// ... lines 38 - 39
    }
// ... lines 41 - 42
```

Let's do it! Run the test:

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

And... we hit the dump!

Testing for and Creating Notifications

Our project has an unused `Notification` entity that I created before recording *just* for this feature: it relates to a treasure and has a message. Nothing fancy. Let's create one of these when we publish... by *first* testing for it. TDD!

At the end of the test, say `NotificationFactory` - that's a Foundry factory that I created, `::repository()` - to get a repository helper - then `->assert()->count(1)`.

tests/Functional/DragonTreasureResourceTest.php

```
↕ // ... lines 1 - 14
15 class DragonTreasureResourceTest extends ApiTestCase
16 {
↕ // ... lines 17 - 220
221     public function testPublishTreasure(): void
222     {
↕ // ... lines 223 - 239
240         NotificationFactory::repository()->assert()->count(1);
241     }
242 }
```

With Foundry, our database is always empty at the start of a test: so checking for 1 row is perfect.

Back in the processor, remove the `dd()`... then check that the test fails our new assertion:

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

Excellent! Back over, start by autowiring a private `EntityManagerInterface $entityManager`. Then, below, I'll paste in some boring code that creates a `Notification` and persists it.

src/State/DragonTreasureStateProcessor.php

```
↕ // ... lines 1 - 13
14 class DragonTreasureStateProcessor implements ProcessorInterface
15 {
↕ // ... lines 16 - 24
25     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
26     {
↕ // ... lines 27 - 33
34         $previousData = $context['previous_data'] ?? null;
35         if ($previousData instanceof DragonTreasure
36             && $data->getIsPublished()
37             && $previousData->getIsPublished() !== $data->getIsPublished()
38         ) {
39             $notification = new Notification();
40             $notification->setDragonTreasure($data);
41             $notification->setMessage('Treasure has been published!');
42             $this->entityManager->persist($notification);
43             $this->entityManager->flush();
44         }
↕ // ... lines 45 - 46
47     }
48 }
```

Coolio. And the test says...

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

... that we rock! Next up: time to get crazy by creating a totally *custom* ApiResource class that is *not* an entity.

Chapter 7: Totally Custom Resource

So far, we have *two* API resource classes: `DragonTreasure` and `User`. And both are *entity* classes. But having your `#[ApiResource]` attribute above an entity class *isn't* a requirement. You can create any normal boring PHP class you want, sprinkle this `#[ApiResource]` attribute on top, and wham, *bam*! It becomes part of your API. *Well*, there is *some* work left, but we'll see that in a moment.

Why *would* you want to create a custom class for your API instead of using an entity? Two main reasons. First: because the data you're serving *doesn't* come from the database... or it comes from a mixture of *different* database tables. Or *second*: the data you're fetching *is* coming from the database... but because your API looks different enough from your entity, you want to clean things up by having a class for your API *separate* from your entity class. We'll play with both cases, starting with the first: when your data comes from somewhere *other* than a database.

Creating the Class

Here's the situation: each day, we post a one-of-a-kind *quest* for our dragons to complete. We want to expose these quests as a new API resource. They'll be able to list all *past* quests, fetch a *single* quest by the date, or update the *status* of a quest if they complete it. That's pretty easy. *But* we're *not* going to store this data in the database. We're going to pretend that the data comes from somewhere else.

So, instead of making an entity, we're going to create a brand-new class and put it in this `ApiResource/` directory. This directory was added *for* us by the API Platform recipe when we originally installed it... and it's meant to be the home for your API resource classes. Add a new PHP class... and let's call it `DailyQuest`.

To make this part of your API, just add `#[ApiResource]` above the class.

src/ApiResource/DailyQuest.php

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

use ApiPlatform\Metadata\ApiResource;

#[ApiResource]
class DailyQuest
{

}
```

That's it! Swing by the docs and... tada! It's *already* in our API documentation! Though, it *does* look a bit odd: the single `GET` operation is missing. Normally, we would see something like `/api/daily_quests/{id}`. We'll uncover the mystery of *why* that's missing in a minute.

ApiResource Class Directories

Oh, and, by the way: to find all of our API resource classes, API Platform scans just *two* directories looking for this attribute: `src/Entity/` and `src/ApiResource`. Though, this *can* be tweaked in `/config/packages/api_platform.yaml` with a mapping paths config.

Okay, so... how could this *possibly*, *already* be part of our API? It's just a class. Heck, it doesn't even have any properties! Try the `GET` collection endpoint. Hit "Execute" and... we get a 404. So... it's not *actually* working. If we try the `POST` endpoint - we're just sending empty data - it returns a 201 status code as if it was *successful*... but behind the scenes, absolutely nothing just happened. No data was created *or* saved.

Look back at our favorite "upgrade" page on the documentation: the one that talks about providers and processors. If we add the `#[ApiResource]` attribute above an *entity* class, we get these processors and providers for free. It turns out that... this is really the *only* difference between adding `#[ApiResource]` above a random class and adding above an entity. When you use `#[ApiResource]` on an *entity*, API Platform automatically gives you processors and providers. When you create a *custom* class, you start with *no* providers and *no* processors. This means that API Platform has no idea how to *load* data when you make a `GET` request... nor how to *process* the data at the end of a `POST` or `PATCH` request.

Adding those missing pieces is *our* job! Let's start that *next*.

Chapter 8: Custom Resource State Provider

We have a shiny new API resource class and... for the most part, we'll use it like normal.

Customizing ApiResource Options

For example, instead of `DailyQuests`, maybe we change the `shortName` to just `Quest`. When we peek at the docs, as expected, the title changes... along with all the URLs.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 6
#[ApiResource(
    shortName: 'Quest'
)]
class DailyQuest
{

}
```

Making the State Provider

To be able to load data and have this collection endpoint *not* return a 404, we need a *state provider*. And it's not *just* the `GET` endpoints. The `PUT` endpoint uses a state provider, as well as `DELETE` and `PATCH`: these all first *load* the resource, before editing or deleting it.

So let's make a state provider! We've done this before. At your terminal, run:

```
./bin/console make:state-provider
```

Call it `DailyQuestStateProvider`. *Awesome* name!

Spin back over, open the `State/` directory and... there it is! Our job is simple: to return the `DailyQuest` object or objects for the current operation.

src/State/DailyQuestStateProvider.php

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

use ApiPlatform\Metadata\Operation;
use ApiPlatform\State\ProviderInterface;

class DailyQuestStateProvider implements ProviderInterface
{
    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
        // Retrieve the state from somewhere
    }
}
```

Let's start *super* basic: return an array with two hard-coded `new DailyQuest()` objects. They're both empty... because that class doesn't have any properties.

src/State/DailyQuestStateProvider.php

```
// ... lines 1 - 10
    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
        return [
            new DailyQuest(),
            new DailyQuest(),
        ];
    }
// ... lines 18 - 19
```

To tell API Platform to *use* the shiny new provider, in `DailyQuest`, add `provider` set to `DailyQuestStateProvider::class`.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 5
use App\State\DailyQuestStateProvider;

#[ApiResource(
// ... line 9
    provider: DailyQuestStateProvider::class,
)]
class DailyQuest
// ... lines 13 - 16
```

Let's give this a whirl! Dash back over to the docs to "Execute" the collection endpoint. And... *yes*! No more 404! We get a 200... and it returned 2 items! All they have are the JSON-LD fields - `@id` and `@type` - but that makes sense since the class doesn't have any other properties.

Adding the Identifier

So, yay! *But*, before we run wild and add more properties, we need to talk about why the `GET` *one* endpoint is missing. We *have* the `GET` *collection* endpoint, but no `GET`-a-single-item endpoint. Why?

Every API resource needs an "identifier". Right now, our class does *not* have an identifier... and that causes the two GET routes to collide. Let me show you!

Spin over and run:

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

I love this. API Platform creates an actual route for *every* operation of every API resource. I'll make this a little smaller... better. You can see all the routes for the quests. Here's the one for `_get_collection` and, above it, the one for `_get_single`... but with the *same* URL!

Usually, the URL would be `/api/quests/{id}`... where `id` is known as the identifier. But... our `DailyQuest` doesn't have *any* properties... so API Platform has *no* idea what to use for the identifier.

So what's the solution? The easiest is to add an `$id` property: `public int $id`... and, for simplicity, let's add a constructor where we can pass the `int $id`. Set the property inside.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 11
class DailyQuest
{
    public int $id;

    public function __construct(int $id)
    {
        $this->id = $id;
    }
}
```

Over in `DailyQuestStateProvider`, invent a few IDs: how about `4` and `5`.

src/State/DailyQuestStateProvider.php

```
// ... lines 1 - 8
class DailyQuestStateProvider implements ProviderInterface
{
    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
        return [
            new DailyQuest(4),
            new DailyQuest(5),
        ];
    }
}
```

Cool, *now* dump the routes again:

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

Behold! The single `GET` has a *different* URL with `{id}`. The `id` was also missing from `put`, `patch`, and `delete`... and it's there now too. Over on the docs, when we refresh... we see the *same* thing.

The identifier is important because it's used in the URLs... and so it's also used to generate the `@id` IRI string for each item. Here, you can see the `@id` is now pointing to `/api/quests/4`.

A non-traditional Identifier with identifier: true

But wait, *how* did API Platform know that the `id` is the all-important "identifier"... and not just some normal property? I'm... *honestly*... not entirely sure. But it seems that the *name* `id` is special... *somewhere* in API platform. If you name a property `id`, API Platform says:

> *“Oh, that must be your identifier!”*

And... it's usually not wrong! *But*, there *is* a more explicit way to say that a property is an identifier. Next, instead of an integer identifier, let's see if we can use a *date* identifier, so we have URLs like `/api/quests/2023-06-05`.

Chapter 9: Using a Custom (Date) Identifier

For our `DailyQuest` API endpoints, we set up an `id` as the identifier. But what we *really* want is a date... so we can have fancy URLs like `/api/quests/2023-06-05`.

Let's try it! In `DailyQuest`, instead of `public int $id`, say `public \DateTimeInterface $day`. And in the constructor, replace the argument with `\DateTimeInterface $day`... and `$this->day = $day`.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 11
class DailyQuest
{
    public \DateTimeInterface $day;

    public function __construct(\DateTimeInterface $day)
    {
        $this->day = $day;
    }
}
```

Next, in `DailyQuestStateProvider`, we'll say... how about `new \DateTime('now')` and `new \DateTime('yesterday')`.

src/State/DailyQuestStateProvider.php

```
// ... lines 1 - 8
class DailyQuestStateProvider implements ProviderInterface
{
    public function provide(Operation $operation, array $uriVariables =
[], array $context = []): object|array|null
    {
        return [
            new DailyQuest(new \DateTime('now')),
            new DailyQuest(new \DateTime('yesterday')),
        ];
    }
}
```

When we refresh the docs... we're back to where we were before: we're missing the ID on `PUT`, `DELETE`, and `PATCH`, and our single `GET` is *gone*. That's because API Platform doesn't know

that the `$day` property is meant to be our identifier. Though, if we try the `GET` collection endpoint... hey! The `day` field *does* show up inside the JSON like a normal property!

What we want to do is tell API Platform:

> *"Hey! This isn't a normal property: `day` is our identifier."*

We do that by adding an `#[ApiProperty]` attribute above this with `identifier: true`.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 4
use ApiPlatform\Metadata\ApiProperty;
// ... lines 6 - 12
class DailyQuest
{
    #[ApiProperty(identifier: true)]
    public \DateTimeInterface $day;
// ... lines 17 - 21
}
```

Debugging IRI Generation Errors

When we check, this *does*, in fact, fix all of our routes. But when we try the collection endpoint... we get a 400 error:

> *"Unable to generate an IRI for the item of type `DailyQuest`."*

So API Platform *loaded* our two `DailyQuest` objects... but when it tried to generate the `@id` property (the IRI), for some reason, it exploded!

To find out more, go down to the web debug toolbar and open up that request in the profiler. On the Exception tab, there were *two* exceptions on this page: a *nested* exception situation.

The top level - `Unable to generate an IRI` - doesn't really tell us *why* there was a problem. Down here, we can see:

> *"We were not able to resolve the identifier matching parameter "day"."*

This error isn't *super* clear either, but it's closer. It's *really* saying:

> *“Yo! I tried to generate the IRI by using the `day` field... but that's a `DateTimeInterface` object... and I don't know how to convert that to a string.”*

We actually chose a pretty tricky IRI to work with, and I think that's cool. API Platform *does* have a system called "URI variable transformer". The `{day}` is a *variable* in the route... and you can help "transform" the `DateTimeInterface` object into something that can be used in that string. The "Identifiers" documentation talks about this.

But there's also a simple solution. Create a new function called `getDayString()` which will return a `string`. Inside, `return $this->day->format()` with the format we want: `Y-m-d`.

Making a Method the Identifier

The trick is to make this *method* the identifier: move the `ApiProperty` from the actual property... down above this.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 12
class DailyQuest
{
// ... lines 15 - 21
    #[ApiProperty(identifier: true)]
    public function getDayString(): string
    {
        return $this->day->format('Y-m-d');
    }
}
```

Perfect! Back over here... the routes still look correct. You can see we have `{dayString}` now. And when we try our `GET` collection endpoint... check it out! We see `"@id": "/api/quests/` and then the date string. That's *exactly* what we wanted!

Though, now we have a `dayString` field in the JSON... as well as the `day` itself. Let's think. We really *don't* need the `day` field at all: it exists internally just to help the `dayString`. And because the `dayString` is in the URL, having that as a field *also* seems unnecessary. Can we hide these?

Hiding Specific Fields from your API

Sure! And we don't even need to use serialization groups! We're going to go deeper into this later, but above the `day` property, we can hide this *entirely* from our API by using an `#[Ignore]` attribute from Symfony's serializer.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\Serializer\Annotation\Ignore;
↕ // ... lines 9 - 13
14 class DailyQuest
15 {
16     #[Ignore]
17     public \DateTimeInterface $day;
↕ // ... lines 18 - 28
29 }
```

If we head over here and "Execute" that... boom! That field is gone: it can't be read or written.

We *could* do the same thing for `getDayString()`. But *another* option is to say `readable: false`. This means it won't be *readable*, but it will still *technically* be writable. However, because there's no `setDayString`, it's not actually writable.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 13
14 class DailyQuest
15 {
↕ // ... lines 16 - 23
24     #[ApiProperty(readable: false, identifier: true)]
25     public function getDayString(): string
↕ // ... lines 26 - 28
29 }
```

Now, when we "Execute" this... that field disappears too.

This is the setup we want! We have the ID we want, we don't have any extra fields that we *don't* want, and we can now *add* whatever fields that we *do* want. To do *that*, we're going to build an Enum.

Create a `src/Enum/` directory... and, inside, a new PHP class, or really enum, called `DailyQuestStatusEnum`. I'll paste some code here.

src/Enum/DailyQuestStatusEnum.php

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

enum DailyQuestStatusEnum: string
{
    case ACTIVE = 'active';
    case COMPLETED = 'completed';
}
```

This is just a way for us to keep track of the *status* of each `DailyQuest`. Back over in that class, let's add some properties: `public string $questName`, `public string $description`.... and whatever other properties we need in our API, like `public int $difficultyLevel`, and a `public DailyQuestStatusEnum` called `$status`.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 14
class DailyQuest
{
// ... lines 17 - 18
    public string $questName;
    public string $description;
    public int $difficultyLevel;
    public DailyQuestStatusEnum $status;
// ... lines 23 - 33
}
```

Null Fields are Hidden

Nice! Let's try this! Head over... and Execute! Hmm, we don't see any of the new fields yet. That's because they're not *populated* and, by default, API Platform *hides* fields that are null or uninitialized.

But if we refresh the page and go down to the documentation for the response... it shows that these *are* part of the API.

Head over to `DailyQuestStateProvider` so we can populate them. Say `return $this->createQuests()`: a new private function we'll create. I'll paste that in as well: you can grab the code from the code block on this page.

src/State/DailyQuestStateProvider.php

```
// ... lines 1 - 9
class DailyQuestStateProvider implements ProviderInterface
{
    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
        return $this->createQuests();
    }

    private function createQuests(): array
    {
        $quests = [];
        for ($i = 0; $i < 50; $i++) {
            $quest = new DailyQuest(new \DateTimeImmutable(sprintf('- %d days', $i)));
            $quest->questName = sprintf('Quest %d', $i);
            $quest->description = sprintf('Description %d', $i);
            $quest->difficultyLevel = $i % 10;
            $quest->status = $i % 2 === 0 ? DailyQuestStatusEnum::ACTIVE : DailyQuestStatusEnum::COMPLETED;

            $quests[$quest->getDayString()] = $quest;
        }

        return $quests;
    }
}
```

This creates *50* quests - each one a day further in the past - and populates simple data for the rest of the fields. Some of the quests will be `ACTIVE`, and others `COMPLETED`.

Oh, and notice that I'm using `getDayString()` as the key for this array. We don't *need* to do that: they keys in the array returned by your collection provider are *not* important. I only did this because it's going to be handy in a few minutes when we create the get one operation.

Testing time! Move over, hit "Execute" again and... look at that! We have 50 items with data on *all* of them. That's *gorgeous*!

Next: Let's get our provider working for the item operations: meaning when we fetch a *single* item. The item provider is used for the GET one operation, `PUT`, `PATCH` and `DELETE`.

Chapter 10: Custom Resource Item Provider

Let's try to get a *single* item. I'll change the date, hit "Execute", and... *200 status code*. Hold your horses... this is returning a *collection*: the *exact* same data as our *collection endpoint*!

Collection vs Item Operations

Ok, each *operation* can have its own provider. But when we put `provider` directly under `#[ApiResource]`, *this* becomes the provider for *every* operation. That's peachy... given you don't forget that *some* operations fetch a *collection* of resources while other fetch a *single* item.

Inside our provider, the `$operation` helps us know the difference. `dd()` that...

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 9
10 class DailyQuestStateProvider implements ProviderInterface
11 {
12     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
13     {
14         dd($operation);
↕ // ... line 15
16     }
↕ // ... lines 17 - 32
33 }
```

Then, over here, copy the URL, paste it in a new tab and add `.jsonld` to the end. There we go! This is a `Get` operation. If we try to fetch the collection, it's `GetCollection`.

Back in the provider,
`if ($operation instanceof CollectionOperationInterface)`,
`return $this->createQuests()`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 4
5  use ApiPlatform\Metadata\CollectionOperationInterface;
↕ // ... lines 6 - 10
11 class DailyQuestStateProvider implements ProviderInterface
12 {
13     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
14     {
15         if ($operation instanceof CollectionOperationInterface) {
16             return $this->createQuests();
17         }
↕ // ... lines 18 - 19
20     }
↕ // ... lines 21 - 36
37 }
```

Below, we know this is an "item" operation.

URI Variables

So this *does* keep the *collection* operation working. Now, we need a way to extract the date string from the URL so we can find the *one* quest that matches. How can we get that? `dd($uriVariables)`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 12
13     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
14     {
↕ // ... lines 15 - 18
19         dd($uriVariables);
20     }
↕ // ... lines 21 - 38
```

When we refresh... behold: there's a `dayString` inside! Notice that, in `DailyQuest`, we never configure what the URL should look like. You *can* do that, but by default, API Platform *automatically* figures out what the route and URL should look like. Run:

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

For the item endpoints, it's `/api/quests/{dayString}`: the `dayString` is a wildcard in the route. In the provider, `$uriVariables` will contain *every* variable part of the URI - so `dayString` in our case. That makes us *dangerous*.

Returning a Single Items

Down here, we need to return a *single* `DailyQuest` or *null*. Say `$quests = $this->createQuests()`, then `return $quests[$uriVariables['dayString']]` or `null` if it's not set.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 12
13     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
14     {
↕ // ... lines 15 - 18
19         $quests = $this->createQuests();
20
21         return $quests[$uriVariables['dayString']] ?? null;
22     }
↕ // ... lines 23 - 40
```

Remember: this works because the array uses `dayString` for each key. In a *real* app, we would want to do this more efficiently: it doesn't make sense to load *every* quest... just to return *one*. But for our test app, this will work fine.

Ok, try that endpoint. Got it! One result. And if we try a random date that *doesn't* exist... like "2013"... we get a 404. API Platform sees that we returned `null` and it handled the 404 for us.

We are now the proud parents of a fully functional state provider! Though we'll talk about this more soon - including topics like pagination. But next: let's shift our focus to creating a *state processor* for our custom resource.

Chapter 11: Custom Resource State Processor

We haven't configured the `operations` key on our `#[ApiResource]`. And so, we get *every* default operation. But we really only need a few. Add `operations` with a `new GetCollection()`, `new Get()` to fetch a single quest and `new Patch()` so users can update the status of an *existing* quest when they complete it.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 6
7  use ApiPlatform\Metadata\Get;
8  use ApiPlatform\Metadata\GetCollection;
9  use ApiPlatform\Metadata\Patch;
↕ // ... lines 10 - 13
14 #[ApiResource(
↕ // ... line 15
16     operations: [
17         new GetCollection(),
18         new Get(),
19         new Patch(),
20     ],
↕ // ... line 21
22 )]
23 class DailyQuest
↕ // ... lines 24 - 43
```

Upon refreshing... I love it!

Speaking of that `Patch` operation, when it's used, API Platform will call the state processor, so we can save... or do whatever we want. We don't have one yet, so that'll be our next job.

Adding a Patch Test

But let's start with a test. Down in `tests/Functional/`, create a new class called `DailyQuestResourceTest`. Make this extend the `ApiTestCase` that we created in the last tutorial and `use ResetDatabase` from Foundry to make sure our database is *empty* at the start of every test. Also `use Factories`.

tests/Functional/DailyQuestResourceTest.php

```
// ... lines 1 - 4
use Zenstruck\Foundry\Test\Factories;
use Zenstruck\Foundry\Test\ResetDatabase;

class DailyQuestResourceTest extends ApiTestCase
{
    use ResetDatabase;
    use Factories;
}
```

Ok, we don't *need* these... since we're not going to talk to the database... but if we decide to later on, we're *ready*.

Down here, add `public function testPatchCanUpdateStatus()`. The first thing we need is a `new \DateTime()` that represents `$yesterday`: `-1 day`.

tests/Functional/DailyQuestResourceTest.php

```
// ... lines 1 - 12
    public function testPatchCanUpdateStatus()
    {
        $yesterday = new \DateTime('-1 day');
// ... lines 16 - 26
    }
// ... lines 28 - 29
```

Remember: in our provider, we're creating daily quests for today through the last 50 days. When we make a `PATCH` request, our item provider is called to "load" the object. So we need to use a date that we know will be found.

Now say `$this->browser()`, `->patch()`... and the URL: `/api/quests/` with `$yesterday->format('Y-m-d')`. Pass a *second* options argument with `json` and an array with `'status' => 'completed'`.

The `status` field is an enum... but because it's backed by a string, the serializer will deserialize it from the string `active` or `completed`. Finish with `->assertStatus(200)`, `->dump()` (that will be handy in a second), and then `->assertJsonMatches()` to check that `status` changed to `completed`.

tests/Functional/DailyQuestResourceTest.php

```
↕ // ... lines 1 - 12
13     public function testPatchCanUpdateStatus()
14     {
↕ // ... line 15
16         $this->browser()
17             ->patch('/api/quests/'.$yesterday->format('Y-m-d'), [
18                 'json' => [
19                     'status' => 'completed',
20                 ],
↕ // ... line 21
22             ])
23             ->assertStatus(200)
24             ->dump()
25             ->assertJsonMatches('status', 'completed')
26         ;
27     }
↕ // ... lines 28 - 29
```

Wonderful! We're not really going to *save* the updated status... but we should at least see that the final JSON has `status` `completed`. Copy this test name... and over here, run: `symfony php bin/phpunit --filter=` and paste that name:

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

And... whoops! We get a 415. The error says:

> *"The content-type `application/json` is not supported."*

Ah... I forgot to add a header to my `PATCH` request. Add `headers` set to an array with `Content-Type`, `application/merge-patch+json`.

tests/Functional/DailyQuestResourceTest.php

```
↕ // ... lines 1 - 12
13     public function testPatchCanUpdateStatus()
14     {
↕ // ... line 15
16         $this->browser()
17             ->patch('/api/quests/'.$yesterday->format('Y-m-d'), [
↕ // ... lines 18 - 20
21                 'headers' => ['Content-Type' => 'application/merge-
   patch+json']
22             ])
↕ // ... lines 23 - 25
26         ;
27     }
↕ // ... lines 28 - 29
```

We talked about this in the last tutorial: this tells the system what *type* of patch we have. This is the only one that's supported right now, but it's still required.

If we try this... it *passes*! But wait, I think I tricked myself! Comment-out the `status` and then the test... still passes? Yup, change that to `-2 days`... and `$yesterday` to just `$day`.

In our provider, we make every other quest active or complete: and yesterday *starts* as complete. Whoops! When we try the test now... it fails. Add the `status` back to the JSON and now... got it! The test passes!

Behind the scenes, here's the process. One: API Platform calls our provider to fetch the *one* `DailyQuest` for this date. Two: the serializer updates that `DailyQuest` using the JSON sent on the request. Three: the state processor is called. And four: the `DailyQuest` is serialized back into JSON.

Creating the State Processor

Except... in our case, there is no step three... because we haven't created a state processor yet! Let's add one!

```
php bin/console make:state-processor
```

and call it `DailyQuestStateProcessor`.

Yet another name *sparkling* with genius. Go check it out: it's empty and *full* of potential.

src/State/DailyQuestStateProcessor.php

```
↕ // ... lines 1 - 7
8  class DailyQuestStateProcessor implements ProcessorInterface
9  {
10     public function process(mixed $data, Operation $operation, array
   $uriVariables = [], array $context = []): void
11     {
12         // Handle the state
13     }
14 }
```

In `DailyQuest`, the processor should be used for the `Patch` operation, so add `processor: DailyQuestStateProcessor::class`.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 14
15 #[ApiResource(
↕ // ... line 16
17     operations: [
↕ // ... lines 18 - 19
20         new Patch(
21             processor: DailyQuestStateProcessor::class,
22         ),
23     ],
↕ // ... line 24
25 )]
26 class DailyQuest
↕ // ... lines 27 - 46
```

To prove that this is working, `dd($data)`.

src/State/DailyQuestStateProcessor.php

```
↕ // ... lines 1 - 7
8  class DailyQuestStateProcessor implements ProcessorInterface
9  {
10     public function process(mixed $data, Operation $operation, array
   $uriVariables = [], array $context = []): void
11     {
12         dd($data);
13     }
14 }
```

Okay! Try the test again:

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

And... boom! The `status` is set to `completed`.

By the way, we added the `processor` option directly to the `Patch()` operation, but we can *also* put it down here on the `#[ApiResource()]` attribute directly.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 14
#[ApiResource(
// ... line 16
    operations: [
// ... lines 18 - 19
        new Patch(),
// ... line 21
    ],
// ... line 23
    processor: DailyQuestStateProcessor::class,
)]
class DailyQuest
// ... lines 27 - 46
```

That makes no difference... because this is the only operation we have that even *uses* a processor: GET method operations *never* call a processor.

State Processor Logic

Anyway, this is *normally* where we would save the data or... do *something*, like send an email if this were a "reset password" API resource.

To make things a *bit* realistic, let's add a `$lastUpdated` property to `DailyQuest` and update it here. Add `public \DateTimeInterface $lastUpdated`.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 25
26 class DailyQuest
27 {
↕ // ... lines 28 - 33
34     public \DateTimeInterface $lastUpdated;
↕ // ... lines 35 - 45
46 }
```

Then populate that inside the state provider: `$quest->lastUpdated` equals `new \DateTimeImmutable()`... with some randomness: between 10 and 100 days ago.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 10
11 class DailyQuestStateProvider implements ProviderInterface
12 {
↕ // ... lines 13 - 23
24     private function createQuests(): array
25     {
↕ // ... line 26
27         for ($i = 0; $i < 50; $i++) {
↕ // ... lines 28 - 32
33             $quest->lastUpdated = new \DateTimeImmutable(sprintf('- %d days', rand(10, 100)));
↕ // ... lines 34 - 35
36         }
↕ // ... lines 37 - 38
39     }
40 }
```

Finally, head over to the state processor. We know that this is only used for `DailyQuest` objects... so `$data` *will* be one of those. Help your editor with `assert($data instanceof DailyQuest)` and, below, `$data->lastUpdated = new \DateTimeImmutable('now')`.

src/State/DailyQuestStateProcessor.php

```
// ... lines 1 - 8
class DailyQuestStateProcessor implements ProcessorInterface
{
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        assert($data instanceof DailyQuest);

        $data->lastUpdated = new \DateTimeImmutable('now');
    }
}
```

Cool! We don't have a test assertion for that field, but we *are* still dumping the response... and we can see it here. I'm looking at my watch and... that *is* the correct time in my little corner of the world. Our state processor is alive!

Celebrate by going back to the test and removing that dump.

Next: Let's make our resource more interesting by adding a relation to *another* API resource: a relation to *dragon treasure*.

Chapter 12: Relating Custom ApiResources

Inside `DailyQuest`, add a new property: `public array $treasures`.

src/ApiResource/DailyQuest.php

```
// ... lines 1 - 25
class DailyQuest
{
// ... lines 28 - 34
    public array $treasures = [];
// ... lines 36 - 46
}
```

This will hold an array of *dragon treasures* that you can *win* if you complete this quest: treasures like a fancy magician's hat... a talking frog... the world's *second* largest slinky... or *all four* corner pieces of a brownie! Mmmmmm...

Adding an array Relations Property

In PHP land, this is just like any other property. Over in our provider, populate it: `$quest->treasures =` ... and then we'll set that to something. Instead of a boring empty array, we need some `DragonTreasure` objects. Up at the top, add `public function __construct()` to autowire a `private DragonTreasureRepository $treasureRepository`.

src/State/DailyQuestStateProvider.php

```
// ... lines 1 - 9
use App\Repository\DragonTreasureRepository;

class DailyQuestStateProvider implements ProviderInterface
{
    public function __construct(
        private DragonTreasureRepository $treasureRepository,
    )
    {
    }
// ... lines 19 - 52
}
```

Below, grab some treasures: `$treasures = $this->treasureRepository->findBy()` passing an empty array for the criteria - so it'll return everything - *no* `orderBy`, and a limit of `10`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 9
10 use App\Repository\DragonTreasureRepository;
11
12 class DailyQuestStateProvider implements ProviderInterface
13 {
14     public function __construct(
15         private DragonTreasureRepository $treasureRepository,
16     )
17     {
18     }
↕ // ... lines 19 - 30
31     private function createQuests(): array
32     {
33         $treasures = $this->treasureRepository->findBy([], [], 10);
↕ // ... lines 34 - 51
52     }
53 }
```

Yea, we're just finding the first 10 treasures in the database. I'll paste in some boring code that will grab a random *set* of these `DragonTreasure` objects. Put *that* onto the `treasures` property.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 9
10 use App\Repository\DragonTreasureRepository;
11
12 class DailyQuestStateProvider implements ProviderInterface
13 {
14     public function __construct(
15         private DragonTreasureRepository $treasureRepository,
16     )
17     {
18     }
↕ // ... lines 19 - 30
31     private function createQuests(): array
32     {
33         $treasures = $this->treasureRepository->findBy([], [], 10);
↕ // ... lines 34 - 35
36         for ($i = 0; $i < 50; $i++) {
↕ // ... lines 37 - 43
44             $randomTreasuresKeys = array_rand($treasures, rand(1, 3));
45             $randomTreasures = array_map(fn($key) => $treasures[$key],
   (array) $randomTreasuresKeys);
46             $quest->treasures = $randomTreasures;
↕ // ... lines 47 - 48
49         }
↕ // ... lines 50 - 51
52     }
53 }
```

Cool! And, even though we don't care right now, to make sure our test keeps passing, at the top here, add `DragonTreasureFactory::createMany(5)`... because if there are *zero* treasures, weird things will happen in our provider... and the dragons will stage their fiery uprising.

tests/Functional/DailyQuestResourceTest.php

```
↕ // ... lines 1 - 4
5 use App\Factory\DragonTreasureFactory;
↕ // ... lines 6 - 8
9 class DailyQuestResourceTest extends ApiTestCase
10 {
↕ // ... lines 11 - 13
14     public function testPatchCanUpdateStatus()
15     {
16         // quests need at least some treasures to be available
17         DragonTreasureFactory::createMany(5);
↕ // ... lines 18 - 29
30     }
31 }
```

Ok, does this new property show up in our API? Head to `/api/quests.jsonld` to see.. a familiar error:

> *"You must call `setIsOwnedByAuthenticatedUser()` before `isOwnedByAuthenticatedUser()`."*

We know this: it comes from `DragonTreasure`... all the way at the bottom.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 93
94 class DragonTreasure
95 {
↕ // ... lines 96 - 263
264     public function isOwnedByAuthenticatedUser(): bool
265     {
266         if (!isset($this->isOwnedByAuthenticatedUser)) {
267             throw new \LogicException('You must call
    setIsOwnedByAuthenticatedUser() before isOwnedByAuthenticatedUser()');
268         }
269
270         return $this->isOwnedByAuthenticatedUser;
271     }
↕ // ... lines 272 - 276
277 }
```

Apparently, the serializer is trying to access this field, but we never set it... which makes sense... because the provider and processor for `DragonTreasure` aren't called when we're using a `DailyQuest` endpoint.

Why The Relation is Embedded

But... hold on a second. This shouldn't even be a problem. Let me show you what I mean. To temporarily silence this error, and understand what's going on, find that property... there it is... and give it a default value of `false`.

src/Entity/DragonTreasure.php

```
↕ // ... lines 1 - 93
94 class DragonTreasure
95 {
↕ // ... lines 96 - 147
148     private bool $isOwnedByAuthenticatedUser = false;
↕ // ... lines 149 - 276
277 }
```

Spin over, refresh, and... *whoa*! It *works*! Here's our daily quest... and *here* are the treasures. *But*... this is not, *quite* what we expected. Each treasure is an *embedded object*.

Remember: when you have a relationship to an object that is an `ApiResource`, like `DragonTreasure`, that object should only be *embedded* if the parent class and child class share serialization groups. Like, if we had `normalizationContext` with `groups` set to `quest:read` like this... where the `quest:read` group is above `$treasures`, *and*, in `DragonTreasure`, we had at least one property that *also* had `quest:read` on it.

But, if you do *not* have this situation - heck, we're not using groups at all - then the serializer should render each `DragonTreasure` as an IRI string. This should be an array of strings not embedded objects!

The *problem* is that the serializer looks at this `$treasures` property and doesn't realize that it holds an array of `DragonTreasure` objects. It knows it's an array, but before it starts serializing, it doesn't know *what* is inside. And so, instead of sending them through the system that serializes `ApiResource` objects, it sends them through the code that serializes *normal* objects... which results in it just serializing all the properties.

This isn't a problem with entities because the serializer is smart: it reads the Doctrine relationship metadata to figure out that a property is a *collection* of some *other* `#[ApiResource]` object. Long story short, this is simple to fix... it's just hard to understand at first. Above the property, add some PHPDoc to help the serializer: `@var DragonTreasure[]`.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 9
10 use App\Entity\DragonTreasure;
↕ // ... lines 11 - 26
27 class DailyQuest
28 {
↕ // ... lines 29 - 35
36     /**
37      * @var DragonTreasure[]
38      */
39     public array $treasures = [];
↕ // ... lines 40 - 50
51 }
```

Try it now... bam! We get IRI strings! I won't bother, but we could undo the default value we added because this object won't be serialized...which is what gave us this error in the first place.

So, other than the embedded object surprise, adding relations to our custom resource is no biggie! Next: instead of embedding `DragonTreasure` objects directly, let's see how we can invent a *new* class and new data structure to represent these treasures.

Chapter 13: Embedding Custom DTO's

One goal of the daily quests resource is to showcase the bountiful treasures a dragon can win by completing a quest. Embedding an array of `DragonTreasure` objects and showing their IRIs is a nice way to do that! But it's not the only way.

Creating the Custom (non-ApiResource) Class

Idea time: forget about pointing to the exact treasures. What if we simply render the name, cool factor, and value of each as a custom array of embedded data? Check it out. In the `src/ApiResource/` directory, though this class could live anywhere, create a *new* class called `DailyQuestTreasure`. This will represent the treasure that you could win by completing a `DailyQuest`.

Inside, create a `public function __construct` with a `public string $name`, `public int $value` and `public int $coolFactor`. I'm using public properties for simplicity and even including all three as arguments to the constructor to make life *even* easier.

src/ApiResource/QuestTreasure.php

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

class QuestTreasure
{
    public function __construct(
        public string $name,
        public int $value,
        public int $coolFactor,
    )
    {
    }
}
```

But, I am *not* going to make this an `ApiResource`. Well, we *could* do that... if we need our API users to be able to fetch `DailyQuestTreasure` data directly... or update them. But that's not the point of this class. This will simply be a data structure that we *attach* to `DailyQuest`.

Over in `DailyQuest`, this will no longer hold an array of `DragonTreasure` objects: it will hold an array of `QuestTreasure` objects. Oh, actually, to keep things shorter... there we go... call it `QuestTreasure`... then over here, `QuestTreasure`.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 26
27 class DailyQuest
28 {
↕ // ... lines 29 - 35
36     /**
37      * @var QuestTreasure[]
38      */
39     public array $treasures = [];
↕ // ... lines 40 - 50
51 }
```

Now that we have the property set up, head to the provider to populate it. Instead of setting the random dragon treasures onto this *directly*, we need to create an array of `QuestTreasure` objects. For each over the random treasures as `$treasure`... then `$questTreasures[]` equals new `QuestTreasure` and pass in the data: `$treasure->getName()`, `$treasure->getValue()` and `$treasure->getCoolFactor()`. Finish with `$quest->treasures = $questTreasures`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 8
9  use App\ApiResource\QuestTreasure;
↕ // ... lines 10 - 12
13 class DailyQuestStateProvider implements ProviderInterface
14 {
↕ // ... lines 15 - 31
32     private function createQuests(): array
33     {
↕ // ... lines 34 - 36
37         for ($i = 0; $i < 50; $i++) {
↕ // ... lines 38 - 46
47             $questTreasures = [];
48             foreach ($randomTreasures as $treasure) {
49                 $questTreasures[] = new QuestTreasure(
50                     $treasure->getName(),
51                     $treasure->getValue(),
52                     $treasure->getCoolFactor(),
53                 );
54             }
55             $quest->treasures = $questTreasures;
↕ // ... lines 56 - 57
58         }
↕ // ... lines 59 - 60
61     }
62 }
```

"Relations" that are Normal Objects

Before and after this change, our `DailyQuest` class had a property that held an array of objects. The *key* difference is that, before, it held an array of objects that were API resources. But now, it holds an array of normal, boring objects that are *not* API resources.

What difference does that make? Check it out. Boom! Embedded objects! When API Platform serializes the `treasures` property, it sees that our `QuestTreasure` is *not* an `ApiResource`. So it serializes it in the normal way: by embedding each property.

This is beautifully simple. And it's something I want you to remember: you can always create new data classes if you want to embed some extra data.

The .well-known genId

But I bet you noticed this weird `@id` with `.well-known/genId`. This... is a randomly-generated string which exists, I believe, because JSON-LD resources are *supposed* to have an `@id`. But since we don't *really* have a place where you can fetch individual Quest Treasures... API Platform gives us this fake one.

Now, in theory, you could turn that off by saying `#[ApiProperty()]` with `genId: false`.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 4
5  use ApiPlatform\Metadata\ApiProperty;
↕ // ... lines 6 - 26
27 class DailyQuest
28 {
↕ // ... lines 29 - 38
39     #[ApiProperty(genId: false)]
40     public array $treasures = [];
↕ // ... lines 41 - 51
52 }
```

Unfortunately, this doesn't seem to work for array properties... maybe I'm doing something wrong. I get that id. But it *does* work for single objects. To prove it, change this to a single `QuestTreasure`. We don't need our `@var` anymore because this now has a proper type.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 26
27 class DailyQuest
28 {
↕ // ... lines 29 - 35
36     #[ApiProperty(genId: false)]
37     public QuestTreasure $treasure;
↕ // ... lines 38 - 48
49 }
```

Over in our provider, I'll change a few things *super* quickly... to get just *one* random `QuestTreasure`. Finish with `$quest->treasure` equals this one `QuestTreasure`. Use `$randomTreasure` for all the variable names.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 12
13 class DailyQuestStateProvider implements ProviderInterface
14 {
↕ // ... lines 15 - 31
32     private function createQuests(): array
33     {
↕ // ... lines 34 - 36
37         for ($i = 0; $i < 50; $i++) {
↕ // ... lines 38 - 44
45             $randomTreasure = $treasures[array_rand($treasures)];
46             $quest->treasure = new QuestTreasure(
47                 $randomTreasure->getName(),
48                 $randomTreasure->getValue(),
49                 $randomTreasure->getCoolFactor(),
50             );
↕ // ... lines 51 - 52
53         }
↕ // ... lines 54 - 55
56     }
57 }
```

I love it! Now when we refresh... we see *one* embedded object and *no* generated `@id` field.

Next up: with a custom resource like this, we don't get pagination on our collection resource automatically. Yup, it's returning *all* 50 items. So let's add that.

Chapter 14: Pagination on a Custom Resource

When we fetch the collection of quests, we see all *50* of them! There's no pagination... a fact I can prove because, at the bottom we don't see any extra data about pagination.

Usually... if we peek at the treasures collection... at the bottom of the response, API Platform adds a `hydra:view` field that describes how you can paginate through these resources. But over here for quests... zilch!

Pagination Comes from the Provider

But where *does* pagination come from in API Platform? It turns out that pagination is completely the responsibility of your state provider. It's... pretty simple actually. *Whatever* your collection provider returns - whether it's an array of quests... or some sort of iterable of quests - is what is serialized to JSON. *But*, if it returns an iterable object that happens to implement a special `PaginatorInterface`, API Platform will see that and render the `hydra:view` pagination details.

Using The TraversablePaginator

So, if we want our collection to support pagination, step one is, instead of returning this array, to return an object that implements that interface. And, fortunately, API Platform already has a class that can help us!

Set the array to a `$quests` variable. Then return new `TraversablePaginator` from API Platform. This takes a few arguments. First, a traversable - basically the results that should be shown for the *current* page. Right now, we'll still use *all* 50 quests. Oh, except this needs to be an *iterable*... so wrap it in a new `ArrayIterator`.

Next is the current page - hardcode that to 1 for now - then items per page - hardcode that to 10 - and finally the total number of items, which for now, I'm just going to count `$quests`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 6
7 use ApiPlatform\State\Pagination\TraversablePaginator;
↕ // ... lines 8 - 13
14 class DailyQuestStateProvider implements ProviderInterface
15 {
↕ // ... lines 16 - 21
22     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
23     {
24         if ($operation instanceof CollectionOperationInterface) {
25             $quests = $this->createQuests();
26 
27             return new TraversablePaginator(
28                 new \ArrayIterator($quests),
29                 1,
30                 10,
31                 count($quests),
32             );
33         }
↕ // ... lines 34 - 37
38     }
↕ // ... lines 39 - 64
65 }
```

This is *not* a very smart paginator yet: it will always be on page 1 and will show every result. But when we go over, refresh... and scroll to the bottom, we *do* see the pagination info! According to this, there are 5 pages of results... which makes sense: 10 items per page and 50 total items. But you'll also see that we're *still* returning 50 items. There's no *real* pagination happening!

Why? Because it's up to *us* to figure out which page we're on and to pass only the *correct* results to the paginator. If we pass it 50 items, it'll render 50 items, regardless of what we tell it are the max per page.

Organizing our Variables

To help us do that, let's set a few variables: `$currentPage` hardcoded to 1, `$itemsPerPage` hardcoded to 10 and `$totalItems`. For this, call a new private method `countTotalQuests()`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 13
14 class DailyQuestStateProvider implements ProviderInterface
15 {
↕ // ... lines 16 - 21
22     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
23     {
24         if ($operation instanceof CollectionOperationInterface) {
25             $currentPage = 1;
26             $itemsPerPage = 10;
27             $totalItems = $this->countTotalQuests();
↕ // ... lines 28 - 36
37         }
↕ // ... lines 38 - 41
42     }
↕ // ... lines 43 - 74
75 }
```

I'll hit Alt+Enter and add that method at the bottom. This will return an `int`... and I'm just going to return 50...

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 13
14 class DailyQuestStateProvider implements ProviderInterface
15 {
↕ // ... lines 16 - 21
22     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
23     {
24         if ($operation instanceof CollectionOperationInterface) {
25             $currentPage = 1;
26             $itemsPerPage = 10;
27             $totalItems = $this->countTotalQuests();
↕ // ... lines 28 - 36
37         }
↕ // ... lines 38 - 41
42     }
↕ // ... lines 43 - 70
71     private function countTotalQuests(): int
72     {
73         return 50;
74     }
75 }
```

because that's the *total* possible quests we have in our "fake" database. If you were using a database, you'd count every available row. Change the code in `createQuests()` to use this.

This probably looks a bit silly: why am I creating a private method to return something so simple? Well, what I *really* want to highlight are the two distinct "jobs" of pagination. First, to return the correct subset of the 50 results - which we'll do in a moment. Second, to return the count of the *total* number of items. When you use Doctrine, it executes 2 separate queries for this: one to fetch the current page's results with a LIMIT and OFFSET, and a second COUNT query to count *every* row.

Current Page, Limit, Offset: The Pagination Service

Ok, back on top, let's use these variables: `$currentPage`, `$itemsPerPage` and `$totalItems`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 13
14 class DailyQuestStateProvider implements ProviderInterface
15 {
↕ // ... lines 16 - 21
22     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
23     {
24         if ($operation instanceof CollectionOperationInterface) {
25             $currentPage = 1;
26             $itemsPerPage = 10;
27             $totalItems = $this->countTotalQuests();
↕ // ... lines 28 - 30
31             return new TraversablePaginator(
32                 new \ArrayIterator($quests),
33                 $currentPage,
34                 $itemsPerPage,
35                 $totalItems,
36             );
37         }
↕ // ... lines 38 - 41
42     }
↕ // ... lines 43 - 70
71     private function countTotalQuests(): int
72     {
73         return 50;
74     }
75 }
```

Ok cool... but what we *really* need to do is determine the *actual* current page and then use that to return only a *subset* of the results. Like, if we're showing 10 per page... and we're on page 2, we should return quests 11 through 20.

Pagination works via a `?page` query parameter: `?page=2` should mean we're on page 2. But our code isn't reading this yet. Look: it still thinks we're on page 1... because we've hardcoded that. To get the correct page, we *could* try to read the query parameter directly... but we don't need to! API Platform gives us a service that already holds *all* the pagination info.

On top, add a second constructor argument called `private Pagination` - from API platform `$pagination`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 6
7  use ApiPlatform\State\Pagination\Pagination;
↕ // ... lines 8 - 14
15 class DailyQuestStateProvider implements ProviderInterface
16 {
17     public function __construct(
18         private DragonTreasureRepository $treasureRepository,
19         private Pagination $pagination,
20     )
21     {
22     }
↕ // ... lines 23 - 78
79 }
```

Below, set `$currentPage` to `$this->pagination->getPage()`, which needs the `$context` that we have as an argument on this method. Then `$itemsPerPage` set to `$this->pagination->getLimit()` passing `$operation` and `$context`. We can also get an `$offset` in a similar way, which is *super* handy. If we're on page 2 and the limit is 10, the `Pagination` service will calculate that the offset should be 11. Dump all four variables below.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 6
7 use ApiPlatform\State\Pagination\Pagination;
↕ // ... lines 8 - 14
15 class DailyQuestStateProvider implements ProviderInterface
16 {
17     public function __construct(
18         private DragonTreasureRepository $treasureRepository,
19         private Pagination $pagination,
20     )
21     {
22     }
23
24     public function provide(Operation $operation, array $uriVariables =
[], array $context = []): object|array|null
25     {
26         if ($operation instanceof CollectionOperationInterface) {
27             $currentPage = $this->pagination->getPage($context);
28             $itemsPerPage = $this->pagination->getLimit($operation,
$context);
29             $offset = $this->pagination->getOffset($operation, $context);
30             $totalItems = $this->countTotalQuests();
31             dd($currentPage, $itemsPerPage, $offset, $totalItems);
↕ // ... lines 32 - 40
41         }
↕ // ... lines 42 - 45
46     }
↕ // ... lines 47 - 78
79 }
```

Let's check this out! Go back to page 1, refresh and look at that! Page 1, 30 items per page, the limit and offset 0. If we go to `page=2`, then it's page 2, the number per page is still 30 and the offset is 30.

Where is it getting 30 as the items per page? That's the default in API Platform for any resource. But this is something you can configure on your `#[ApiResource]` attribute: change `paginationItemsPerPage` to, how about, 10.

src/ApiResource/DailyQuest.php

```
↕ // ... lines 1 - 15
16 #[ApiResource(
↕ // ... lines 17 - 23
24     paginationItemsPerPage: 10,
↕ // ... lines 25 - 26
27 )]
28 class DailyQuest
29 {
↕ // ... lines 30 - 49
50 }
```

Now try it. That changes to 10 and the offset is 10. If we go to page 3, our per page is still 10. And now it's saying:

> *"Hey, since we're on page 3, you should start at result 20."*

Fetching the Correct Results for the Current Page

We're in *great* shape now. Our *final* job is to use this info to return the correct *subset* of results, instead of *all* the quests. To do that, pass `$offset` and `$itemsPerPage` to `createQuests()`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 14
15 class DailyQuestStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 23
24     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
25     {
26         if ($operation instanceof CollectionOperationInterface) {
↕ // ... lines 27 - 29
30             $totalItems = $this->countTotalQuests();
31
32             $quests = $this->createQuests($offset, $itemsPerPage);
↕ // ... lines 33 - 39
40         }
↕ // ... lines 41 - 44
45     }
↕ // ... lines 46 - 77
78 }
```

Down here, add `int $offset` and `int $limit` with a default of 50. And use those: `$i = $offset` and then `$i <=` `$offset` plus `$limit`.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 14
15 class DailyQuestStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 23
24     public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
25     {
26         if ($operation instanceof CollectionOperationInterface) {
↕ // ... lines 27 - 29
30             $totalItems = $this->countTotalQuests();
31
32             $quests = $this->createQuests($offset, $itemsPerPage);
↕ // ... lines 33 - 39
40         }
↕ // ... lines 41 - 44
45     }
↕ // ... line 46
47     private function createQuests(int $offset, int $limit = 50): array
48     {
↕ // ... lines 49 - 52
53         for ($i = $offset; $i < ($offset + $limit) && $i < $totalQuests; $i++) {
↕ // ... lines 54 - 68
69         }
↕ // ... lines 70 - 71
72     }
↕ // ... lines 73 - 77
78 }
```

Ok team check it out! We're on page 3 and... these are the items from page 3! It's more obvious if we go to page 1. See the descriptions: description 1, 2, 3 and so on. So, pagination is working on our collection!

Though, in this simple example, I need to make sure I don't break the item provider. Because we're looking up the day string as an array key, we need to return *all* the quests. To make sure that happens, pass 0 and 50.

src/State/DailyQuestStateProvider.php

```
↕ // ... lines 1 - 14
15 class DailyQuestStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 23
24     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
25     {
↕ // ... lines 26 - 41
42         $quests = $this->createQuests(0, $this->countTotalQuests());
↕ // ... lines 43 - 44
45     }
↕ // ... lines 46 - 77
78 }
```

In a real app, you would make this smarter by, for example, querying for the *one* item you need... instead of loading *all* of them.

So that's pagination for a custom resource. What about filtering? We're going to talk about creating custom filters in a future tutorial. But spoiler alert: the filtering logic is *also* something that happens right here inside the collection provider.

Next: let's remove all the API resource stuff from our `User` entity and add it to a new class that's going to be dedicated to our API. Woh.

Chapter 15: User Class Dto

The *fastest* way to get started with API Platform is by adding these `#[ApiResource]` attributes above your entity classes. That's because API Platform gives you free state providers that query from the database (which includes pagination and filters) and free state *processors* that save things *to* the database.

To use DTOs or Not?

But, as we've seen with `DailyQuest`, that's not *required*. And if your API starts to look pretty different from your entities - like you have fields in your API that don't exist in your entity or are named differently - it might make sense to *separate* your entity and API resource classes.

Right now, our entities *are* API resources... and that *has* added some complexity. For example, we have a custom `isMine` field which is powered by this `isOwnedByAuthenticatedUser` property: a non-persisted property that we populate via a state provider. And one of the most *noticeable* things is our huge use of serialization groups. We *have to* use serialization groups, like `treasure:read`, so that we can include the properties we *want* and avoid the properties that we *don't* want.

This *has* saved us some time... but increased complexity. So let's get *crazy* and use a dedicated *class* for our API *from the start*. That's often referred to as a "DTO", or "Data Transfer Object". I'll use that term a lot - but for us, it just means "the dedicated class for our API" - like the `DailyQuest` class.

Removing the API Stuff from User

Alright, folks, commence cleanup! It's time to wipe out all the API-related grime from our pristine `User` entity. Remove the `#[ApiResource()]` attribute... both of them, filters and validation. You *may* still want validation constraints if you're using your entity with the form system... but since we're not, let's clear it. I'm also clearing anything related to serialization... and hunting down hopefully everything that's hiding.

Woh. This class is *a lot* smaller now. I think that's everything... the `use` statements on top look good... so... awesome!

Let's also remove the state processor for `User`, which hashes the plain password. We *are* going to re-implement many of the things we just deleted, but I want to start with a clean look at things.

Alright, go check out the API docs. We're reduced to "Quest" and "Treasure". I love it!

Creating the DTO / Dedicated ApiResource Class

We're going to start like we did with the `DailyQuest`. In the `src/ApiResource/` directory, create a new class called `UserApi`... to indicate this is the *user* class for our API. Inside, add `#[ApiResource]` above it.

src/ApiResource/UserApi.php

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

use ApiPlatform\Metadata\ApiResource;

#[ApiResource]
class UserApi
{

}
```

So far, this is just like any other custom API resource. It shows up in the docs... and if we try the `GET` collection operation, it fails with a 404. Heck, we're even missing the "ID" part in the URL of the item operations.

To fix that, in `UserApi`, add a `public ?int $id = null` property... because our users *will* still be identified by their database id. Oh, and I'm using a public property *just* to make life easier... and because this class will stay simple, so it's not a big deal.

src/ApiResource/UserApi.php

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

use ApiPlatform\Metadata\ApiResource;

#[ApiResource]
class UserApi
{
    public ?int $id = null;
}
```

The moment we do this... API Platform *recognizes* that `id` as the identifier, and our operations are *looking good*.

While we're here, let's also tweak the `shortName`. This is called `UserApi`, which is a *terrible* name - so change it: `shortName: 'User'`.

src/ApiResource/UserApi.php

```
// ... lines 1 - 6
#[ApiResource(
    shortName: 'User',
)]
class UserApi
{
    public ?int $id = null;
}
```

Suddenly... this is *starting* to look like what we had before!

The *big* missing pieces, like with `DailyQuest`, are the state provider and state processor. Let's add the state provider next.... but with a *twist* that leverages a brand-new feature that's going to save us a *ton* of work.

Chapter 16: stateOptions + entityClass Magic

When we create a non-entity API resource, *we're* responsible for loading and saving the data. What's *frustrating* is that, if we make a custom state provider for `UserApi`, it will do the *exact* same thing as the core Doctrine state provider: query the database. It's a *bummer* to reinvent all of that logic ourselves. This, historically, has been the Achille's heel of DTO's.

Checking out the Core CollectionProvider

Crack open the core `CollectionProvider` from Doctrine ORM. If you ever wanted to see what the `CollectionProvider` looks like, *here* it is! It's more complex than I imagined. It creates the `QueryBuilder`, calls `handleLinks()` (which intelligently joins to other tables based on the data you need), and houses the query extension system. In the last tutorial, we created a query extension for `DragonTreasure` so it would only return *published* items. And *part* of that extension system, though we can't see it here, is where pagination and filtering is added.

So, this class gives us a *lot*... and I want to reuse it. So, darn it, let's yolo this thing and try to!

Trying to use the CollectionProvider

Head over to `UserApi`, say `provider`, and point to `CollectionProvider` (the one from Doctrine ORM).

src/ApiResource/UserApi.php

```
// ... lines 1 - 4
use ApiPlatform\Doctrine\Orm\State\CollectionProvider;
// ... lines 6 - 7
#[ApiResource(
    shortName: 'User',
    provider: CollectionProvider::class,
)]
class UserApi
{
    public ?int $id = null;
}
```

Let's see what happens! At the browser, go to the endpoint *directly* - `/api/users.jsonld`. And... we get an *error*:

> *"Call to a member function `getRepository()` on null."*

Coming from the core `CollectionProvider`. Boo. But not surprising. Our `UserApi` isn't an entity... and so when it tries to figure out how to query for it, explosions!

Hello stateOptions + entityClas

But *psst*... want to hear a secret? There *is* a way we can hint to the provider that data for this class should come from the `User` entity. It looks like this: `stateOptions` set it to a `new Options` object (making sure to grab the one from ORM), and inside, `entityClass: User::class`.

src/ApiResource/UserApi.php

```
// ... lines 1 - 4
use ApiPlatform\Doctrine\Orm\State\CollectionProvider;
use ApiPlatform\Doctrine\Orm\State\Options;
// ... line 7
use App\Entity\User;

#[ApiResource(
    shortName: 'User',
    provider: CollectionProvider::class,
    stateOptions: new Options(entityClass: User::class),
)]
class UserApi
{
    public ?int $id = null;
}
```

Let's see what happens now! When we head over and refresh... whoa! It looks like that worked! We see "totalItems: 11"... with items 1-11 all right here. We only have an `$id` property, but I guess that makes sense... since we only have an `$id` property inside our `UserApi`.

Let's add a few more properties! How about `public ?string $email = null` and `public ?string $username = null`. Both of these properties also live in our `User` entity.

src/ApiResource/UserApi.php

```
// ... lines 1 - 14
class UserApi
{
    public ?int $id = null;

    public ?string $email = null;

    public ?string $username = null;
}
```

When we refresh... those pop up too! This is *working*.... but how? What the heck is going on?

How this all Works

If we could peek under API Platform's hood, we would see that the underlying API resource objects *are* `UserApi`. So what we're seeing here *is* the JSON for a collection of `UserApi` objects.

But there are *several* places in the system that look for `stateOptions` and, if it's present, will use the `entityClass` from that. The `CollectionProvider` we opened a moment ago - the one from Doctrine ORM - is one of those cases. It grabs the `entityClass` from `stateOptions` if there is one... then uses that when it does the query.

In fact, as soon as we have this `stateOptions` + `entityClass` thing, API Platform sets the provider and the processor *automatically* to the core Doctrine ones. So we don't even *need* to have the `provider` key: it's set for us.

src/ApiResource/UserApi.php

```
// ... lines 1 - 8
#[ApiResource(
    shortName: 'User',
    stateOptions: new Options(entityClass: User::class),
)]
class UserApi
{
// ... lines 15 - 19
}
```

Okay, but if the provider is querying for `User` *entity* objects, *how* and *when* is that converted to `UserApi` objects... so that they can be serialized to JSON? The *answer* is *during* serialization... and it's a bid odd. Thanks to `stateOptions`, API Platform is actually serializing the `User` *entity* object. But to get the list of the properties that it should serialize, it reads the metadata from `UserApi`. Then, it grabs the property values *from* `User`... and puts them onto a `UserApi` instance. Essentially, it serializes the `User` entity *into* a `UserApi` object... and *then* to JSON.

This seems to work well... but with one, major limitation.

Limitation: No Custom Properties

Add a property that is *not* on our entity, like `public int $flameThrowingDistance = 0`. There is *no* `$flameThrowingDistance` property over on `User`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 12
13 class UserApi
14 {
↕ // ... lines 15 - 20
21     public int $flameThrowingDistance = 0;
22 }
```

When we try this... *explosion*! If we scroll down a bit, we see that this comes from the *normalizer* system... which is part of the serializer. It looks at `UserApi`, thinks "Oh, I need a `$flameThrowingDistance` field", tries to fetch that from `User`, and, since it's not there, boom!

So the colossal, monstrous, titanic limitation of the `entityClass` strategy is... we *can't* have extra fields on our `UserApi` class. But no worries: we'll find a path around this in the next chapter. For now, remove the extra property.

Oh, and one *other* limitation that you may have noticed is that we don't have the JSON-LD fields `@id` or `@type`. We'll handle that while we're fixing the issue with custom fields... like the multitasking wizards we are.

Adding a Relation Property

Let's add another property: `public array $dragonTreasures = []`? We *do* have a `$dragonTreasures` property over on `User` that holds a collection of `DragonTreasure` objects.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 12
13 class UserApi
14 {
↕ // ... lines 15 - 19
20
21     public array $dragonTreasures = [];
22 }
```

So if we go over and test this out... it works fine! Though, *surprisingly*, it's *embedding* the `dragonTreasures` instead of returning them as IRIs. This is the same problem we saw earlier, and the fix is the same.

I *do* want to point out one interesting thing about this, though. When it embeds the `dragonTreasures`, one of the properties is `owner`. Right now, that *owner* is actually the `User` entity. Since the `User` *entity* is no longer an API resource, it adds this random `genid` thing.

I'll talk about this more in a bit, but once we start creating DTOs and using *those* instead of entities, we'll probably want to use DTOs for *all* of our API resources... instead of mixing entities and DTOs... because it creates issues like this.

Anyway, fix this by advertising that this is an `array` of `DragonTreasure`. I'm using a slightly different array syntax there, but it doesn't really matter.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 6
7 use App\Entity\DragonTreasure;
↕ // ... lines 8 - 13
14 class UserApi
15 {
↕ // ... lines 16 - 21
22     /**
23      * @var array<int, DragonTreasure>
24      */
25     public array $dragonTreasures = [];
26 }
```

If we try this again... back to IRIs! Woo!

Built-in Pagination

So far, we know that `stateOptions` does *three* things. One: It automatically sets the provider and processor to use the core Doctrine provider and processor. Two: the provider is smart enough to query from this entity. This also works for *single* items, like `/users/1.jsonld`. And three: The serializer *serializes* the `User` entity *into* a `UserApi` object.

The fact that `stateOptions` causes the core Doctrine state provider to be used has some *very* important other side effects. First, we get pagination *for free*. Add `paginationItemsPerPage: 5`, go over, and refresh. We see that the total number of items is "11"... but it only shows *five*... and the pages are down here.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 9
10 #[ApiResource(
11     shortName: 'User',
12     paginationItemsPerPage: 5,
13     stateOptions: new Options(entityClass: User::class),
14 )]
15 class UserApi
16 {
↕ // ... lines 17 - 26
27 }
```

Second, the collection provider also makes the query extension system work. We don't have any query extensions for `User`, but we *do* have one for `DragonTreasure`. Later on, when we convert `DragonTreasure` to its own DTO class, this extension is *still* going to work.

The third and final goodie is that the *filter* system still works! Watch: above `UserApi`, add `#[ApiFilter()]` with `SearchFilter::class` and `properties:` with `username` set to `partial`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Doctrine\Orm\Filter\SearchFilter;
↕ // ... line 6
7 use ApiPlatform\Metadata\ApiFilter;
↕ // ... lines 8 - 16
17 #[ApiFilter(SearchFilter::class, properties: [
18     'username' => 'partial',
19 ])]
20 class UserApi
21 {
↕ // ... lines 22 - 31
32 }
```

Go back and look at the documentation... *whoops*. I autocompleted the `SearchFilter` from ODM. Delete that, then I'll hit Alt+Enter to grab the one from `ORM`.

Refresh the docs again... and look at the `/api/users` endpoint. It *is* advertising that there's a `username` filter, and it *is* going to work! In the other tab, add `?username=Clumsy`.

And... yes! It only returns those 5 results! So the filter system works! Though, one thing to note is that, when we say `username`, we're referring to the `$username` property on the `User` *entity*. As far as the filter is concerned, we don't even *need* a `username` in `UserApi`.

So: we're reusing all of this core Doctrine provider logic, we have pagination, filters and.... it's the best thing since ice cream sandwiches. Except... for that big, scary limitation: that our DTO can't have custom fields. And... that's really the whole point of a DTO: to gain the flexibility of having different fields than your entity. So let's see how to fix that limitation *next*.

Chapter 17: Entities, DTO's & The "Central" Object

This entity class thing seems almost too good to be true. It gives us all the flexibility, in theory, of a custom class, while reusing all the core Doctrine provider and processor logic. But hold your horses because there are *two*, albeit *fixable*, snags.

Most importantly, we're not allowed to have custom property names. This will cause an error when it tries to serialize. Second, I haven't mentioned it yet, but *write* operations - like `POST` or `PATCH` - don't work at all. Well... if we, *posted* to our endpoint, the data *would* be deserialized... but it wouldn't be saved to the database.

The Problem with Write Operations

We can try this because we already have a test for it. Open `UserResourceTest` and, down here, copy `testPostToCreateUser()`. Spin over and run that with:

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

And... 400 error! Open that up. Uh oh:

> *"Unable to generate an IRI for the item of type `App\ApiResource\UserApi`."*

Here's what happens. The serializer *deserializes* this JSON into a `UserApi` object. Yay! That `UserApi` object is *then* passed to the core Doctrine *persist* processor: the thing that normally saves entities to the database. But because `UserApi` is *not* an entity, that processor does... *nothing*. *Then*, when `UserApi` is serialized back to JSON, the `$id` is *still* null - because nothing was ever saved to the database - and... so the IRI can't be generated for it.

We *could* fix this by creating a custom state processor for `UserApi` that saves this to the database. But even if we *did*, the write operations, like `POST` and `PATCH`, just aren't designed to work out of the box with this `entityClass` solution. The reason... is a bit technical, but important.

Understanding the "Central Object" for an Operation

Internally, for every API request, API Platform has a *central* object that it's working on. If we fetch a *single* item, that central object *is* that single item. And that's really important. It's used in various places, like the `security` attribute: when we use `is_granted`, the `object` variable will be that "central" object. For example, if we make a `Patch()` request, that means we're editing a dragon treasure... so the central object will be a `DragonTreasure` entity. Easy peasy!

What's the catch? Well, when you use the `entityClass` solution with a *read* operation (so, one of these `GET` requests), the central object will be the *entity*. So the `User` entity will be the central object. But with a *write* operation (most importantly, the `POST` operation to create a new user), that central object will suddenly be a `UserApi` object. That causes some *serious* inconsistency: the central object will sometimes be an entity... and other times the DTO. Good luck making a `security` system that works with both of those... and isn't completely confusing.

Also, when the `User` *entity* is the central object, *that's* when we run into the problem that prevents us from having custom fields on our DTO.

So, if we could make the `UserApi` be the central object in *all* cases, *then* we'd have consistent security... and we could *also* fix our big custom properties problem.

How can we pull that off? By writing a custom state provider that *returns* `UserApi` objects. Think about it: because the core Doctrine collection provider returns `User` *entity* objects, *those* become the central objects. If we, instead, return `UserDto` objects, problem solved. If this doesn't all make sense yet, I'm not surprised. Let's walk through this step-by-step.

Decorating the Core State Provider

Start by running:

```
php bin/console make:state-provider
```

Call it `EntityToDtoStateProvider`. My goal is to create a *generic* state provider that will work for *all* cases where we have an API resource class that pulls data from an entity. So, we'll

mostly keep user-specific code out of here.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 2
3  namespace App\State;
4
5  use ApiPlatform\Metadata\Operation;
6  use ApiPlatform\State\ProviderInterface;
7
8  class EntityToDtoStateProvider implements ProviderInterface
9  {
10     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
11     {
12         // Retrieve the state from somewhere
13     }
14 }
```

Over in `UserApi`, set `provider` to `EntityToDtoStateProvider`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 9
10 use App\State\EntityToDtoStateProvider;
↕ // ... lines 11 - 12
13 #[ApiResource(
↕ // ... lines 14 - 15
16     provider: EntityToDtoStateProvider::class,
↕ // ... line 17
18 )]
↕ // ... lines 19 - 21
22 class UserApi
23 {
↕ // ... lines 24 - 33
34 }
```

Ok! In `EntityToDtoStateProvider`, we could *manually* query for our `User` entity objects, turn those into `UserApi` objects... then return them. *But* that's the whole thing we're trying to *avoid*! We want to continue to reuse all of that nice Doctrine query logic: that's the beauty of `stateOptions`.

To do that, like we've done before, we're going to *decorate* the core Doctrine provider. Say `public function __construct()` with `private ProviderInterface $collectionProvider`. And to help Symfony know which

to pass in, use the `#[Autowire()]` attribute and say `service: CollectionProvider` (make sure you get the one from Doctrine ORM), followed by `::class`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 9
10 use App\State\EntityToDtoStateProvider;
↕ // ... lines 11 - 12
13 #[ApiResource(
↕ // ... lines 14 - 15
16     provider: EntityToDtoStateProvider::class,
↕ // ... line 17
18 )]
↕ // ... lines 19 - 21
22 class UserApi
23 {
↕ // ... lines 24 - 33
34 }
```

Down here, add `$entities = $this->collectionProvider->provide()`, passing `$operation`, `$uriVariables`, and `$context`. Below, `dd($entities)`

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Doctrine\Orm\State\CollectionProvider;
↕ // ... lines 6 - 9
10 class EntityToDtoStateProvider implements ProviderInterface
11 {
12     public function __construct(
13         #[Autowire(service: CollectionProvider::class)] private
    ProviderInterface $collectionProvider
14     )
15     {
16
17     }
18
19     public function provide(Operation $operation, array $uriVariables =
    [], array $context = []): object|array|null
20     {
21         $entities = $this->collectionProvider->provide($operation,
    $uriVariables, $context);
22         dd($entities);
23     }
24 }
```

Let's see what happens! Head back over, refresh the endpoint, and... *got it*! We *are* calling the core provider, and it's returning a *paginator* object. To see what's hiding *inside* that `Paginator`, say `dd(iterator_to_array($entities))`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 9
10 class EntityToDtoStateProvider implements ProviderInterface
11 {
↕ // ... lines 12 - 18
19     public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
20     {
21         $entities = $this->collectionProvider->provide($operation, $uriVariables, $context);
22         dd(iterator_to_array($entities));
23     }
24 }
```

Back over here... this show *five* `User` entity objects.

At this point, our new provider isn't doing... *anything* special. If we returned `$entities`, we'd be *exactly* where we started: with `User` entities as the central object. Our goal is to return `UserApi` objects... and we're going to do that *next*.

Chapter 18: Provider: Transforming Entities to DTOs

Let's keep track of the goal. When we first used `stateOptions`, it triggered the core Doctrine collection provider to be used. That's *great*... except that it returns `User` *entities*, meaning that *those* became the *central* objects for the `UserApi` endpoints. That causes a serious limitation when serializing: our `UserApi` properties need to match our `User` properties... otherwise the serializer explodes.

To fix that and give us *full* control, we've created our own state provider that calls the core collection provider. But instead of returning these `User` entity objects, we're going to return `UserApi` objects so that *they* become the *central* objects and serialize *normally*.

Mapping to the DTO

Create a `$dtos` array and `foreach` over `$entities as $entity`. *Then* add to the `$dtos` array by calling a new method: `mapEntityToDto($entity)`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 9
10 class EntityToDtoStateProvider implements ProviderInterface
11 {
↕ // ... lines 12 - 18
19     public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
20     {
21         $entities = $this->collectionProvider->provide($operation, $uriVariables, $context);
22         dd(iterator_to_array($entities));
23     }
24 }
```

Hit "alt" + "enter" to add that method at the bottom. This will return an `object`. Well... it will be a `UserApi` object... but we're trying to keep this class generic. I'll paste in some logic - you can copy this from the code block on this page - then hit "alt" + "enter" to add the missing `use`

statement. This code *is* user-specific... but we'll make it more generic later, so we can reuse this class for dragon treasures.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 7
8 use App\ApiResource\UserApi;
↕ // ... lines 9 - 10
11 class EntityToDtoStateProvider implements ProviderInterface
12 {
↕ // ... lines 13 - 19
20     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
21     {
↕ // ... lines 22 - 23
24         $dtos = [];
25         foreach ($entities as $entity) {
26             $dtos[] = $this->mapEntityToDto($entity);
27         }
↕ // ... lines 28 - 29
30     }
31
32     private function mapEntityToDto(object $entity): object
33     {
34         $dto = new UserApi();
35         $dto->id = $entity->getId();
36         $dto->email = $entity->getEmail();
37         $dto->username = $entity->getUsername();
38         $dto->dragonTreasures = $entity->getDragonTreasures()->toArray();
39
40         return $dto;
41     }
42 }
```

But isn't this refreshingly boring and understandable code? Just transferring properties from the `User` `$entity`... onto the DTO. The only thing that's *kind of* fancy is where we change this collection to an array... because this property is an `array` on `UserApi`.

Finally, at the bottom of `provide()`, `return $dtos`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 7
8 use App\ApiResource\UserApi;
↕ // ... lines 9 - 10
11 class EntityToDtoStateProvider implements ProviderInterface
12 {
↕ // ... lines 13 - 19
20     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
21     {
↕ // ... lines 22 - 23
24         $dtos = [];
25         foreach ($entities as $entity) {
26             $dtos[] = $this->mapEntityToDto($entity);
27         }
28
29         return $dtos;
30     }
31
32     private function mapEntityToDto(object $entity): object
33     {
34         $dto = new UserApi();
35         $dto->id = $entity->getId();
36         $dto->email = $entity->getEmail();
37         $dto->username = $entity->getUsername();
38         $dto->dragonTreasures = $entity->getDragonTreasures()->toArray();
39
40         return $dto;
41     }
42 }
```

Thanks to this, the central objects will be `UserApi` objects... and *these* will be serialized normally: no fanciness where the serializer tries to go from a `User` entity into a `UserApi`.

Drumoll please! Tada! It works... with the same result as before! But *now* we have the power to add custom properties.

Adding Custom Properties

Add back the `public int $flameThrowingDistance`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 21
22 class UserApi
23 {
↕ // ... lines 24 - 34
35     public int $flameThrowingDistance = 0;
36 }
```

Then, in the provider, *this* is where we have an opportunity to set those custom properties, like `$dto->flameThrowingDistance = rand(1, 10)`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 10
11 class EntityToDtoStateProvider implements ProviderInterface
12 {
↕ // ... lines 13 - 31
32     private function mapEntityToDto(object $entity): object
33     {
↕ // ... lines 34 - 38
39         $dto->flameThrowingDistance = rand(1, 10);
40
41         return $dto;
42     }
43 }
```

And... *voilà*! We are so freakin' dangerous right now! We're reusing the core Doctrine `CollectionProvider`, but with the ability to add custom fields. Oh! And I forgot to mention: the JSON-LD fields `@id` and `@type` are *back*. We did it!

Fixing Pagination

Though, it looks like we're now missing *pagination*. The filter is documented... but the `hydra:view` field that documents the pagination is gone! Ok, really, pagination *does* still work. Watch: if I go to `?page=2`, the first "user 1" user... becomes *"user 6"*. Yup, internally, the core `CollectionProvider` from Doctrine is *still* reading the current page and querying for the correct set of objects *for* that page. We're missing the `hdra:view` field at the bottom that *describes* the pagination simply because we're no longer returning an object that implements `PaginationInterface`.

Remember, this `$entities` variable is actually a `Pagination` object. Now that we're just returning an array, it makes API Platform *think* that we don't support pagination.

The solution is dead-simple. Instead of returning `$dtos`, `return new TraversablePaginator()` with a new `\ArrayIterator()` of `$dtos`. For the other arguments, we can grab those from the original paginator. To help, `assert($entities instanceof Paginator)` (the one from Doctrine ORM). Then, down here, use `$entities->getCurrentPage()`, `$entities->getItemsPerPage()`, and `$entities->getTotalItems()`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Doctrine\Orm\Paginator;
↕ // ... lines 6 - 7
8 use ApiPlatform\State\Pagination\TraversablePaginator;
↕ // ... lines 9 - 12
13 class EntityToDtoStateProvider implements ProviderInterface
14 {
↕ // ... lines 15 - 21
22     public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
23     {
24         $entities = $this->collectionProvider->provide($operation, $uriVariables, $context);
25         assert($entities instanceof Paginator);
↕ // ... lines 26 - 31
32         return new TraversablePaginator(
33             new \ArrayIterator($dtos),
34             $entities->getCurrentPage(),
35             $entities->getItemsPerPage(),
36             $entities->getTotalItems()
37         );
38     }
↕ // ... lines 39 - 50
51 }
```

The core collection provider already did all that hard work for us. What a pal. Refresh now. The results don't change... but down here, `hydra:view` is back!

Next: Let's get this working for our item operations, like `GET` one or `PATCH`. We'll also leverage our new system to add something to `UserApi` that we *previously* had.... but this time, we're going to do it in a much cooler way.

Chapter 19: Entity -> DTO Item State Provider

What about the item endpoint? If we go to `/api/users/6.jsonld`... it *looks* like it works... but it's a trap! It's just the collection *format*... with a single item!

We know that there are *two* core providers: `CollectionProvider` and an *item* provider, whose job is to return one item or null. Because we set `provider` to `EntityToDtoStateProvider`, it's using this *one* `provider` for *every* operation. And that's ok... as long as we make it smart enough to handle both cases.

We saw how to do this earlier: `$operation` is the key. Add `if ($operation instanceof CollectionOperationInterface)`. Now we can warp all of this code up here. Lovely!

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Doctrine\Orm\Paginator;
↕ // ... lines 6 - 7
8 use ApiPlatform\State\Pagination\TraversablePaginator;
↕ // ... lines 9 - 12
13 class EntityToDtoStateProvider implements ProviderInterface
14 {
↕ // ... lines 15 - 21
22     public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
23     {
24         $entities = $this->collectionProvider->provide($operation, $uriVariables, $context);
25         assert($entities instanceof Paginator);
↕ // ... lines 26 - 31
32         return new TraversablePaginator(
33             new \ArrayIterator($dtos),
34             $entities->getCurrentPage(),
35             $entities->getItemsPerPage(),
36             $entities->getTotalItems()
37         );
38     }
↕ // ... lines 39 - 50
51 }
```

Below, this will be our item provider. `dd($uriVariables)`.

src/State/EntityToDtoStateProvider.php

```
// ... lines 1 - 4
use ApiPlatform\Metadata\CollectionOperationInterface;
// ... lines 6 - 13
class EntityToDtoStateProvider implements ProviderInterface
{
// ... lines 16 - 22
    public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
    {
        if ($operation instanceof CollectionOperationInterface) {
            $entities = $this->collectionProvider->provide($operation, $uriVariables, $context);
            assert($entities instanceof Paginator);

            $dtos = [];
            foreach ($entities as $entity) {
                $dtos[] = $this->mapEntityToDto($entity);
            }

            return new TraversablePaginator(
                new \ArrayIterator($dtos),
                $entities->getCurrentPage(),
                $entities->getItemsPerPage(),
                $entities->getTotalItems()
            );
        }

        dd($uriVariables);
    }
// ... lines 44 - 55
}
```

Calling the Core Item Provider

When we try the item operation... nice! That's what we expect to see: the `id` value, which is the dynamic part of the route.

Just like with the collection provider, we do *not* want to do the querying work manually. Instead, we'll... "delegate" it the core Doctrine item provider. Add a second argument... we can just copy the first... type-hinted with `ItemProvider` (the one from Doctrine ORM), and called `$itemProvider`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 4
5 use ApiPlatform\Doctrine\Orm\State\ItemProvider;
↕ // ... lines 6 - 14
15 class EntityToDtoStateProvider implements ProviderInterface
16 {
17     public function __construct(
18         #[Autowire(service: CollectionProvider::class)] private
   ProviderInterface $collectionProvider,
19         #[Autowire(service: ItemProvider::class)] private
   ProviderInterface $itemProvider,
20     )
21     {
22
23     }
↕ // ... lines 24 - 63
64 }
```

I like it! Back below, let *it* do the work with `$entity = $this->itemProvider->provide()` passing `$operation`, `$uriVariables` and `$context`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 14
15 class EntityToDtoStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 24
25     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
26     {
↕ // ... lines 27 - 43
44         $entity = $this->itemProvider->provide($operation, $uriVariables,
   $context);
↕ // ... lines 45 - 50
51     }
↕ // ... lines 52 - 63
64 }
```

This will give us an `$entity` object or null. If we *don't* have an `$entity` object, `return null`. That will trigger a 404. But if we *do* have an `$entity` object, we don't want to return that directly. Remember, the whole point of this class is to take the `$entity` object and *transform* it into a `UserApi` DTO.

So instead, `return $this->mapEntityToDto($entity)`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 14
15 class EntityToDtoStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 24
25     public function provide(Operation $operation, array $uriVariables = [], array $context = []): object|array|null
26     {
↕ // ... lines 27 - 43
44         $entity = $this->itemProvider->provide($operation, $uriVariables, $context);
45
46         if (!$entity) {
47             return null;
48         }
49
50         return $this->mapEntityToDto($entity);
51     }
↕ // ... lines 52 - 63
64 }
```

That feels good. And... the endpoint works *beautifully*. If we try an *invalid* id, our provider returns null and API Platform takes care of the 404.

Only Showing Published Dragon Treasures

Side note: if you follow some of these related treasures, they *may* 404 as well. Let's see... we have 21 and 27. 21 works for me... and for 27... that *also* works... of course. Anyway, the reason some *might* 404 is that, right now, if I go back, the `dragonTreasures` property includes *all* the treasures related to this user: even the *unpublished* ones. But in a previous tutorial, we created a query extension that *prevented* unpublished treasures from being loaded.

Back when the `User` entity was our API resource, we *avoided* returning unpublished treasures from this property. We created `getPublishedDragonTreasures()` and made *that* the `dragonTreasures` property.

But in our state provider, we're setting *all* of them. This is an easy fix: change to `getPublishedDragonTreasures()`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 14
15 class EntityToDtoStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 52
53     private function mapEntityToDto(object $entity): object
54     {
↕ // ... lines 55 - 58
59         $dto->dragonTreasures = $entity->getPublishedDragonTreasures()->getValues();
↕ // ... lines 60 - 62
63     }
64 }
```

Actually, undo that... then refresh the collection endpoint. Ok, we see treasures 16 and 40 down here... then after using the new method... only 16! "40" is *unpublished*.

That was *easy*! And it highlights something cool. In order to have a `dragonTreasures` field that returned something special when our `User` *entity* was an ApiResource, we needed a dedicated method and a `SerializedName` attribute. But with a custom class, we don't need any weirdness. We can do *whatever* we want in the state provider. Our classes stay shiny and clean!

Next: Let's get our users *saving* with a state processor: a delicate dance that involves handling new *and* existing users.

Chapter 20: DTO -> Entity State Processor

We've checked off the "provider" side of things for our new `UserApi` class. So let's shift our focus to the *processor* so we can *save* things. And we *do* have some rather delightful tests for our `User` endpoints. Open `UserResourceTest`.

The Anatomy of the Request & State Processor

Ok, `testPostToCreateUser()`, posts some data, creates the user, then tests to make sure that the password we posted *works* by logging in. Add `->dump()` to help us see what's going on. Then, copy that method name and run it:

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

No surprise... it fails:

> *"Current response status code is 400, but 201 expected."*

The dump is really helpful. It's our favorite error!

> *"Unable to generate an IRI for the item of type `UserApi`."*

We already talked about what's happening: the JSON is deserialized into a `UserApi` object. Good! *Then* the core Doctrine `PersistProcessor` is called because that's the default `processor` when using `stateOptions`. But... because our `UserApi` *isn't* an entity, `PersistProcessor` does *nothing*. Finally, API Platform serializes the `UserApi` *back* into JSON... but without the `id` populated, it fails to generate the IRI.

Watch! Over in `UserApi`, temporarily default `$id` to `5`. When we try the test *now*...

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

It *appears* to work. Ok, it fails... but only later... down here in `UserResourceTest` line 33. It *is* getting through the POST successfully.

Creating the State Processor

Look at the response on top, it *is* returning this user JSON. But, still, nothing is *saving*. Change the id back to null. We need to fix this lack of saving by creating a new state processor. So spin over and run:

```
php bin/console make:state-processor
```

Call it `EntityClassDtoStateProcessor` because, again, we're going to make this class generic so that it works for *any* API resource class that's tied to a Doctrine entity. We'll use it later for `DragonTreasure`.

src/State/EntityClassDtoStateProcessor.php

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

use ApiPlatform\Metadata\Operation;
use ApiPlatform\State\ProcessorInterface;

class EntityClassDtoStateProcessor implements ProcessorInterface
{
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        // Handle the state
    }
}
```

With the empty processor generated, go hook it up in `UserApi` with `processor: EntityClassDtoStateProcessor::class`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 9
10 use App\State\EntityClassDtoStateProcessor;
↕ // ... lines 11 - 13
14 #[ApiResource(
↕ // ... lines 15 - 17
18     processor: EntityClassDtoStateProcessor::class,
↕ // ... line 19
20 )]
↕ // ... lines 21 - 23
24 class UserApi
25 {
↕ // ... lines 26 - 37
38 }
```

Henceforth, every time we POST, PATCH, or DELETE this resource, *this* processor will be called.

Mapping the DTO Back to an Entity

But what is this `$data` variable exactly? You may have a guess, but just in case, let's `dd($data)`... and rerun the test.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 9
10 use App\State\EntityClassDtoStateProcessor;
↕ // ... lines 11 - 13
14 #[ApiResource(
↕ // ... lines 15 - 17
18     processor: EntityClassDtoStateProcessor::class,
↕ // ... line 19
20 )]
↕ // ... lines 21 - 23
24 class UserApi
25 {
↕ // ... lines 26 - 37
38 }
```

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

Yup, it's a `UserApi` object! The JSON we sent is deserialized into this `UserApi` object, and then *that* is passed to our state processor. The `UserApi` object is the "central object" inside of API Platform for this request.

Our job in the state processor is simple but important: to convert this `UserApi` back to a `User` entity so that we can save it. Say `assert($data instanceof UserApi)` and, inside, `$entity =` set to a new helper function: `$this->mapDtoToEntity($data)`. Below, `dd($entity)`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 6
7  use App\ApiResource\UserApi;
↕ // ... lines 8 - 10
11 class EntityClassDtoStateProcessor implements ProcessorInterface
12 {
↕ // ... lines 13 - 19
20     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
21     {
22         assert($data instanceof UserApi);
23
24         $entity = $this->mapDtoToEntity($data);
25         dd($entity);
26     }
↕ // ... lines 27 - 47
48 }
```

Then go add that new `private function mapDtoToEntity()`, which will accept an `object $dto` argument and return another `object`.

Again, we know this will *really* accept a `UserApi` object and return a `User` entity... but we're trying to keep this class generic so we can reuse it later. Though we *are* going to have some user-specific code down here temporarily. In fact, to help our editor, add another `assert($dto instanceof UserApi)`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 6
7  use App\ApiResource\UserApi;
↕ // ... lines 8 - 10
11 class EntityClassDtoStateProcessor implements ProcessorInterface
12 {
↕ // ... lines 13 - 19
20     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
21     {
22         assert($data instanceof UserApi);
23
24         $entity = $this->mapDtoToEntity($data);
25         dd($entity);
26     }
27
28     private function mapDtoToEntity(object $dto): object
29     {
30         assert($dto instanceof UserApi);
↕ // ... lines 31 - 46
47     }
48 }
```

Querying for the Existing Entity

We need to think about two different cases. The first is when we POST to create a brand-new user. In that case, `$dto` will have a `null` id. And that means we should create a fresh `User` object. The *other* case is if we were making, for example, a `PATCH` request to edit a user. In *that* case, the item *provider* will first *load* that `User` entity from the database... our provider will turn that into a `UserApi` object with `id` equal to `6`... and that will eventually be passed to us here. If the `id` is 6... we *don't* want to create a new `User` object: we want to *query* the database for tha *existing* `User`. Our job is to handle *both* situations.

Undo the changes to the test so we don't break anything... and now, `if $dto->id`, we need to query for an existing `User`. To do that, on top, add a constructor with `private UserRepository $userRepository`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 8
9  use App\Repository\UserRepository;
10
11 class EntityClassDtoStateProcessor implements ProcessorInterface
12 {
13     public function __construct(
14         private UserRepository $userRepository
15     )
16     {
17
18     }
↕ // ... lines 19 - 47
48 }
```

Back down here, say `$entity = $this->userRepository->find($dto->id)`.

If we *don't* find that `User`, throw a big giant exception that will trigger a 500 error with `Entity %d not found`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 8
9  use App\Repository\UserRepository;
10
11 class EntityClassDtoStateProcessor implements ProcessorInterface
12 {
13     public function __construct(
14         private UserRepository $userRepository
15     )
16     {
17
18     }
↕ // ... lines 19 - 27
28     private function mapDtoToEntity(object $dto): object
29     {
30         assert($dto instanceof UserApi);
31         if ($dto->id) {
32             $entity = $this->userRepository->find($dto->id);
33
34             if (!$entity) {
35                 throw new \Exception(sprintf('Entity %d not found', $dto->id));
36             }
↕ // ... lines 37 - 38
39         }
↕ // ... lines 40 - 46
47     }
48 }
```

You might be wondering:

> *"Shouldn't this trigger a 404 error instead?"*

The answer, in this case, is *no*. If we're in this situation, it means the item state provider has already successfully queried for a `User` with this id. So there should be *no way* for us to suddenly *not* find it. There *are* some exceptions to this, like if you allowed your user to *change* their `id`... *or* if you allowed users to create *brand-new* objects and *set* the id manually... but for *most* situations, including ours, if this happens, something went weird.

Next up, if we *don't* have an `id`, say `$entity = new User()`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 7
8  use App\Entity\User;
9  use App\Repository\UserRepository;
10
11 class EntityClassDtoStateProcessor implements ProcessorInterface
12 {
13     public function __construct(
14         private UserRepository $userRepository
15     )
16     {
17
18     }
↕ // ... lines 19 - 27
28     private function mapDtoToEntity(object $dto): object
29     {
30         assert($dto instanceof UserApi);
31         if ($dto->id) {
32             $entity = $this->userRepository->find($dto->id);
33
34             if (!$entity) {
35                 throw new \Exception(sprintf('Entity %d not found', $dto-
   >id));
36             }
37         } else {
38             $entity = new User();
39         }
↕ // ... lines 40 - 46
47     }
48 }
```

Done! In both cases, down here, we're going to map the `$dto` object to the `$entity` object. This code is boring... so I'll speed through this. For the password, put a `TODO` temporarily because we still need to hash that. *Also* add a `TODO` for `handle dragon treasures`. Just focus on the easy stuff... and at the bottom, `return $entity`.

src/State/EntityClassDtoStateProcessor.php

```
// ... lines 1 - 7
use App\Entity\User;
use App\Repository\UserRepository;

class EntityClassDtoStateProcessor implements ProcessorInterface
{
    public function __construct(
        private UserRepository $userRepository
    )
    {

    }
// ... lines 19 - 27
    private function mapDtoToEntity(object $dto): object
    {
        assert($dto instanceof UserApi);
        if ($dto->id) {
            $entity = $this->userRepository->find($dto->id);

            if (!$entity) {
                throw new \Exception(sprintf('Entity %d not found', $dto->id));
            }
        } else {
            $entity = new User();
        }

        $entity->setEmail($dto->email);
        $entity->setUsername($dto->username);
        $entity->setPassword('TODO properly');
        // TODO: handle dragon treasures

        return $entity;
    }
}
```

If we've done things correctly, we'll take the `UserApi`, transform that into an `$entity` and dump it. Rerun the test:

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

And... *404*! Let's see what happened here. Oh... *of course*. I never put my test back together. This should be `->post('/api/users')`. Try that again and... *got it*! There's our `User` entity object with the email and username transferred correctly!

Next: Let's *save* this by leveraging the core Doctrine `PersistProcessor` and `RemoveProcessor`. We'll also handle hashing the password. By the end, our user tests will be passing with *flying* colors.

Chapter 21: Leveraging the Core Processor

Look at us go! In our state processor, we have *successfully* transformed the `UserApi` into a `User` entity. So let's save it! We *could* inject the entity manager, persist and flush... and call it a day. But I'd rather offload that work to the core `PersistProcessor`. Search for that file and open it.

It does the simple persisting and flushing... but it *also* has some pretty complex logic for `PUT` operations. We're not really using those, but the point is: better to reuse this class than try to roll our own logic.

Calling the Core PersistProcessor

How we do that should be familiar by this point. Add a `private ProcessorInterface $persistProcessor`... and so Symfony knows *precisely* which service we want, include the `#[Autowire()]` attribute, with `service` set to `PersistProcessor` (in this case, there's only one to choose from) `::class`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 4
5  use ApiPlatform\Doctrine\Common\State\PersistProcessor;
↕ // ... line 6
7  use ApiPlatform\State\ProcessorInterface;
↕ // ... lines 8 - 10
11 use Symfony\Component\DependencyInjection\Attribute\Autowire;
12
13 class EntityClassDtoStateProcessor implements ProcessorInterface
14 {
15     public function __construct(
16         private UserRepository $userRepository,
17         #[Autowire(service: PersistProcessor::class)] private
   ProcessorInterface $persistProcessor,
18     )
19     {
20
21     }
↕ // ... lines 22 - 53
54 }
```

Very nice! Below, save with `$this->persistProcessor->process()` passing `$entity`, `$operation`, `$uriVariables`, and `$context`... which are all the same arguments we have up here.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 4
5  use ApiPlatform\Doctrine\Common\State\PersistProcessor;
↕ // ... line 6
7  use ApiPlatform\State\ProcessorInterface;
↕ // ... lines 8 - 10
11 use Symfony\Component\DependencyInjection\Attribute\Autowire;
12
13 class EntityClassDtoStateProcessor implements ProcessorInterface
14 {
15     public function __construct(
16         private UserRepository $userRepository,
17         #[Autowire(service: PersistProcessor::class)] private
   ProcessorInterface $persistProcessor,
18     )
19     {
20
21     }
↕ // ... line 22
23     public function process(mixed $data, Operation $operation, array
   $uriVariables = [], array $context = [])
24     {
↕ // ... lines 25 - 28
29         $this->persistProcessor->process($entity, $operation,
   $uriVariables, $context);
↕ // ... lines 30 - 31
32     }
↕ // ... lines 33 - 53
54 }
```

Oh, and like before, when we generated this class, it generated `process()` with a `void` return type. That's not exactly correct. You don't *have to* return anything from state processors, but you *can*. And whatever you *do* return - in this case, we'll return `$data` - will ultimately become the "thing" that is serialized and returned back to the user. If you don't return anything, it will use `$data`.

src/State/EntityClassDtoStateProcessor.php

```
// ... lines 1 - 4
use ApiPlatform\Doctrine\Common\State\PersistProcessor;
// ... line 6
use ApiPlatform\State\ProcessorInterface;
// ... lines 8 - 10
use Symfony\Component\DependencyInjection\Attribute\Autowire;

class EntityClassDtoStateProcessor implements ProcessorInterface
{
    public function __construct(
        private UserRepository $userRepository,
        #[Autowire(service: PersistProcessor::class)] private ProcessorInterface $persistProcessor,
    )
    {

    }
// ... line 22
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
    {
// ... lines 25 - 28
        $this->persistProcessor->process($entity, $operation, $uriVariables, $context);

        return $data;
    }
// ... lines 33 - 53
}
```

Setting the id onto the DTO

Ok, I think this should work (Famous last words...).

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

And... it bombs. We're still getting a 400 error, and it's *still* `Unable to generate an IRI for the item`.

So... what's going on? We map the `UserApi` to a new `User` object and *save* the new `User`... which causes Doctrine to assign the new `id` to that entity object. *But* we never take that new id and put it *back* onto our `UserApi`.

To fix this, after saving, add `$data->id = $entity->getId()`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 4
5  use ApiPlatform\Doctrine\Common\State\PersistProcessor;
↕ // ... line 6
7  use ApiPlatform\State\ProcessorInterface;
↕ // ... lines 8 - 10
11 use Symfony\Component\DependencyInjection\Attribute\Autowire;
12
13 class EntityClassDtoStateProcessor implements ProcessorInterface
14 {
15     public function __construct(
16         private UserRepository $userRepository,
17         #[Autowire(service: PersistProcessor::class)] private
   ProcessorInterface $persistProcessor,
18     )
19     {
20
21     }
↕ // ... line 22
23     public function process(mixed $data, Operation $operation, array
   $uriVariables = [], array $context = [])
24     {
↕ // ... lines 25 - 28
29         $this->persistProcessor->process($entity, $operation,
   $uriVariables, $context);
30         $data->id = $entity->getId();
31
32         return $data;
33     }
↕ // ... lines 34 - 54
55 }
```

And if we try it now...

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

it *still* fails... but we got further this time! The response looks good. It returned a 201 status code with the new user info. It's *failing* on the part of the test where it tries to *use* the password to log in. That's because our password is currently set to... `TODO`. We'll fix that in a minute.

Handling the Delete Operation

But *first*, when we set the `processor` on the top level `#[ApiResource]`, this became the processor for *all* operations: `POST`, `PUT`, `PATCH`, *and* `DELETE`. `POST`, `PUT`, and `PATCH` are all pretty much the same: save the object to the database. But `DELETE` is different: we're not saving, we're *removing*.

To handle that, check `if ($operation instanceof DeleteOperationInterface)`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 6
7  use ApiPlatform\Metadata\DeleteOperationInterface;
↕ // ... lines 8 - 14
15 class EntityClassDtoStateProcessor implements ProcessorInterface
16 {
↕ // ... lines 17 - 25
26     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
27     {
↕ // ... lines 28 - 31
32         if ($operation instanceof DeleteOperationInterface) {
↕ // ... lines 33 - 35
36         }
↕ // ... lines 37 - 41
42     }
↕ // ... lines 43 - 63
64 }
```

Like with saving, deleting isn't hard... but it's still better to offload this work to the core Doctrine remove processor. So, up here, copy the argument... and inject *another* processor: `RemoveProcessor`... and rename this to `$removeProcessor`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 5
6 use ApiPlatform\Doctrine\Common\State\RemoveProcessor;
7 use ApiPlatform\Metadata\DeleteOperationInterface;
↕ // ... lines 8 - 14
15 class EntityClassDtoStateProcessor implements ProcessorInterface
16 {
17     public function __construct(
↕ // ... lines 18 - 19
20         #[Autowire(service: RemoveProcessor::class)] private
ProcessorInterface $removeProcessor,
21     )
22     {
23
24     }
25
26     public function process(mixed $data, Operation $operation, array
$uriVariables = [], array $context = [])
27     {
↕ // ... lines 28 - 31
32         if ($operation instanceof DeleteOperationInterface) {
↕ // ... lines 33 - 35
36         }
↕ // ... lines 37 - 41
42     }
↕ // ... lines 43 - 63
64 }
```

Back down here, say `$this->removeProcessor->process()` and pass `$entity`, `$operation`, `$uriVariables`, and `$context` just like the other processor.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 5
6 use ApiPlatform\Doctrine\Common\State\RemoveProcessor;
7 use ApiPlatform\Metadata\DeleteOperationInterface;
↕ // ... lines 8 - 14
15 class EntityClassDtoStateProcessor implements ProcessorInterface
16 {
17     public function __construct(
↕ // ... lines 18 - 19
20         #[Autowire(service: RemoveProcessor::class)] private
ProcessorInterface $removeProcessor,
21     )
22     {
23
24     }
25
26     public function process(mixed $data, Operation $operation, array
$uriVariables = [], array $context = [])
27     {
↕ // ... lines 28 - 31
32         if ($operation instanceof DeleteOperationInterface) {
33             $this->removeProcessor->process($entity, $operation,
$uriVariables, $context);
↕ // ... lines 34 - 35
36         }
↕ // ... lines 37 - 41
42     }
↕ // ... lines 43 - 63
64 }
```

A key thing to note is that we're going to `return null`. In the case of a `DELETE` operation, we don't return *anything* in the response... which we accomplish by returning `null` from here. I don't have a test set up for this, but we'll take a leap of faith and assume it works. Ship it!

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 5
6  use ApiPlatform\Doctrine\Common\State\RemoveProcessor;
7  use ApiPlatform\Metadata\DeleteOperationInterface;
↕ // ... lines 8 - 14
15 class EntityClassDtoStateProcessor implements ProcessorInterface
16 {
17     public function __construct(
↕ // ... lines 18 - 19
20         #[Autowire(service: RemoveProcessor::class)] private
   ProcessorInterface $removeProcessor,
21     )
22     {
23
24     }
25
26     public function process(mixed $data, Operation $operation, array
   $uriVariables = [], array $context = [])
27     {
↕ // ... lines 28 - 31
32         if ($operation instanceof DeleteOperationInterface) {
33             $this->removeProcessor->process($entity, $operation,
   $uriVariables, $context);
34
35             return null;
36         }
↕ // ... lines 37 - 41
42     }
↕ // ... lines 43 - 63
64 }
```

Hashing the Password

Just one more problem to tackle: hashing the plain password. We've done this before, so no biggie. Before we do too much here, open `UserApi`... and add a `public ?string $password = null`... with a comment. This will *always* hold null or the "plaintext" password if the user sends one. We're *never* going to need to handle the *hashed* password in our API, so we don't need any space for that... which is nice!

Back in the processor, `if ($dto->password)`, *then* we know we need to hash that and set it on the user. If a *new* user is being created, this will always be set... but when updating a user, we'll make this field optional. If it's not set, do nothing so the user's current password stays.

To do the hashing, on top, add one more argument: `private UserPasswordHasherInterface $userPasswordHasher`. Then back below, `$entity->setPassword()` set to `$this->userPasswordHasher->hashPassword()`, passing `$entity` (the `User` object) and the plain password: `$dto->password`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 13
14 use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
↕ // ... line 15
16 class EntityClassDtoStateProcessor implements ProcessorInterface
17 {
18     public function __construct(
↕ // ... lines 19 - 21
22         private UserPasswordHasherInterface $userPasswordHasher,
23     )
24     {
25
26     }
↕ // ... lines 27 - 45
46     private function mapDtoToEntity(object $dto): object
47     {
↕ // ... lines 48 - 60
61         if ($dto->password) {
62             $entity->setPassword($this->userPasswordHasher-
>hashPassword($entity, $dto->password));
63         }
↕ // ... lines 64 - 66
67     }
68 }
```

Phew. Let's try the test again. And... it *fails*... with

> *"The annotation "@The" in property `UserApi::$password` was never imported."*

So... that's me tripping on my keyboard and adding an extra `@`. Remove that... then try again:

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

It *passes*! Which means it fully-logged in using that password! *Though*, uh oh, look at the dumped JSON response: this is after we `POST` to create the user. In the JSON response, it includes the plaintext `password` property that the user just set. *Whoops*!

The Flow of a Write Request

Let's break this down. Our state provider is used for all `GET` operations as well as the `PATCH` operation. And notice, we are *not* setting the `password` *ever*. We don't want to return that field in the JSON, so we're, correctly, *not* mapping it from our entity to our DTO. That's *good*!

But the `POST` operation is the *one* situation where the provider is never called. This data is deserialized directly into a new `UserApi` object and that's passed to our processor. *This* means that our DTO *does* have the plain password set on it... And, ultimately, *that* DTO object is what is serialized and sent back to the user.

This is a long way of saying that, in `UserApi`, this password is meant to be a *write-only* field. The user should *never* be able to *read* this. Next: let's talk about how we can do customizations like this inside of `UserApi`, while avoiding the *complexity* of serialization groups.

Chapter 22: Controlling Fields without Groups

When your API resource is on an entity, serialization groups are a *must* because you'll definitely have some properties that you want to show or *not* show. But serialization groups add *complexity*. One of the big benefits of having a separate class for your API is not *needing* serialization groups. Because... the whole point of your API class is to represent your API... so, in theory, you'll want *every* property to be part of your API.

But, in the real world, that's not always true. And we just ran into one case: `password` should be a write-only field. Let's try to replicate some of the complexity that our `User` entity *originally* had, but by avoiding serialization groups.

In `UserResourceTest`, down here, remove the `->dump()`... and after we `->assertStatus(201)`, assert that the `password` property is *not* returned. To do that, we can say `->use(function(Json $json))`. The `use()` function comes from browser and there are a few different objects - like `Json` - that you can *ask* it to pass you via the type-hint. In this case, browser takes the JSON from the last response, puts it into a `Json` object and passes it to us. Use it by saying `$json->assertMissing('password')`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 6
7  use Zenstruck\Browser\Json;
↕ // ... lines 8 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 15
16     public function testPostToCreateUser(): void
17     {
18         $this->browser()
↕ // ... lines 19 - 26
27             ->use(function (Json $json) {
28                 $json->assertMissing('password');
29             })
↕ // ... lines 30 - 36
37         ;
38     }
↕ // ... lines 39 - 87
88 }
```

If we try that now:

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

It *fails* because `password` *does* exist.

readable: false

Okay, let's take a tour of *how* we can customize our API fields without groups. One of the easiest, (and, coincidentally, my *favorite*) is to use `#[ApiProperty()]` with `readable: false`.

src/ApiResource/UserApi.php

```
// ... lines 1 - 7
use ApiPlatform\Metadata\ApiProperty;
// ... lines 9 - 24
class UserApi
{
// ... lines 27 - 35
    #[ApiProperty(readable: false)]
    public ?string $password = null;
// ... lines 38 - 44
}
```

We want this to be *writable*, but not *readable*.

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

And... that fixes things! *Beautiful*.

Let's repeat this for `id`... because `id` is pretty useless since we have `@id`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 15
16     public function testPostToCreateUser(): void
17     {
↕ // ... lines 18 - 26
27             ->use(function (Json $json) {
28                 $json->assertMissing('password');
29                 $json->assertMissing('id');
30             })
↕ // ... lines 31 - 38
39     }
↕ // ... lines 40 - 88
89 }
```

When we run that... it fails because `id` *is* being returned. So now, copy... just the `readable: false` part... add `#[ApiProperty]` above `id`, paste, and I'll also add `identifier: true`... just to be explicit.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 24
25 class UserApi
26 {
27     #[ApiProperty(readable: false, identifier: true)]
28     public ?int $id = null;
↕ // ... lines 29 - 45
46 }
```

And now...

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

That *passes*.

writable: false

Let's keep going. Copy the next test name - `testPatchToUpdateUser` - and run it:

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

It passes *immediately*! Yay! `->patch()` is already working. To dive deeper into other ways we can hide or show fields, also send a `flameThrowingDistance` field in the JSON set to 999. And down here, `->dump()` the response.

tests/Functional/UserResourceTest.php

```
// ... lines 1 - 10
class UserResourceTest extends ApiTestCase
{
// ... lines 13 - 40
    public function testPatchToUpdateUser(): void
    {
// ... lines 43 - 44
        $this->browser()
// ... lines 46 - 53
            ->dump()
            ->assertStatus(200);
    }
// ... lines 57 - 90
}
```

Before we try this, find `EntityClassDtoStateProcessor`. Right after we set the `id`, `dump($data)`. Those two dumps will help us understand *exactly* how this all works.

src/State/EntityClassDtoStateProcessor.php

```
// ... lines 1 - 15
class EntityClassDtoStateProcessor implements ProcessorInterface
{
// ... lines 18 - 27
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
    {
// ... lines 30 - 40
        $data->id = $entity->getId();
        dump($data);
// ... lines 43 - 44
    }
// ... lines 46 - 68
}
```

Now run the test:

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

And... awesome. The first dump on top - from the state processor - shows `flameThrowingDistance` 999, which means the field is *writable*. And below, the response returned 999, which means the field is also *readable*. Yup... this is a normal, boring field. If the user sends the field in JSON, that new value *is* deserialized onto the object.

Ok, experimentation time! In `UserApi`, above the property, start with the same `#[ApiProperty()]` and `readable: false`. We've already seen this.

src/ApiResource/UserApi.php

```
// ... lines 1 - 24
class UserApi
{
// ... lines 27 - 44
    #[ApiProperty(readable: false)]
    public int $flameThrowingDistance = 0;
}
```

When we run the test, on top, the "999" was *written* onto the `UserApi`, but it doesn't show up in the response. It's writable, but not readable.

If we *also* pass `writable: false`... and try again. On top, the value is just "10". The field is *not* writable, so the field in the JSON was ignored.

src/ApiResource/UserApi.php

```
// ... lines 1 - 24
class UserApi
{
// ... lines 27 - 44
    #[ApiProperty(readable: false, writable: false)]
    public int $flameThrowingDistance = 0;
}
```

It's also not in the response: it's not readable or writable.

The readable/writable options alone are probably going to solve most situations. But next, let's learn some other tricks and see why you probably want to make sure that your identifier is *not* writable.

Chapter 23: Other Conditional Field Strategies

Let's keep playing with how we can hide or show fields. Remove the `#[ApiProperty]` attribute. Then, on top, set the `normalizationContext` option. We used this in previous tutorials... but this time, instead of `groups`, set a key called `AbstractNormalizer::IGNORED_ATTRIBUTES` and *then* set that to an array. Inside, put `flameThrowingDistance`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 24
25 class UserApi
26 {
↕ // ... lines 27 - 44
45     #[ApiProperty(readable: false, writable: false)]
46     public int $flameThrowingDistance = 0;
47 }
```

Whether a field is readable or writable really comes down to the serializer. This tells the serializer:

> *"Yo! When you're normalizing - so going to JSON - ignore this property."*

This should make it *writable*, but not *readable*. When we try it...

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

That's exactly what happens! To wrap it in a "do not write" sign, duplicate this move with `denormalizationContext`.

src/ApiResource/UserApi.php

```
// ... lines 1 - 10
use Symfony\Component\Serializer\Normalizer\AbstractNormalizer;
// ... lines 12 - 15
#[ApiResource(
// ... line 17
    normalizationContext: [AbstractNormalizer::IGNORED_ATTRIBUTES => ['flameThrowingDistance']],
    denormalizationContext: [AbstractNormalizer::IGNORED_ATTRIBUTES => ['flameThrowingDistance']],
// ... lines 20 - 23
)]
// ... lines 25 - 27
class UserApi
{
// ... lines 30 - 48
}
```

Copy that, put a "de" on the front of it, and now when we try it:

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

Yup! `flameThrowingDistance` is "1" - so it is *not* writable, and down here... it's not readable either. *Sweet*.

So this is just a different option that should work the same as `ApiProperty`... though I *have* seen complex cases where this context option worked when the `ApiProperty` solution did not. Anyway, delete those.

The #[Ignore] Attribute

The last way to ignore a field - if you want to ignore it completely - is to add an attribute called... `#[Ignore]`! This comes from Symfony's serializer system.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 10
11 use Symfony\Component\Serializer\Annotation\Ignore;
↕ // ... lines 12 - 25
26 class UserApi
27 {
↕ // ... lines 28 - 45
46     #[Ignore]
47     public int $flameThrowingDistance = 0;
48 }
```

When we try the test:

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

Perfect: It is *not* writable nor readable. Cool!

Alrighty, let's hit the reset button on all that dummy code. Get rid of the `#[Ignore]`... and let's see if we have any extra `use` statements up here. Then, over in our processor, remove the `->dump()`... and in our test, get rid of that extra field and the other `->dump()`. All clean!

Avoiding Writable on the Identifier

On this topic of readable and writable, right now, we can actually *change* the `id` field in a `PATCH` request. Watch: set this to `47`... which I just made up, and... it *fails* with a 500 error!

Open up the error:

> *"Entity 47 not found."*

That's coming from *our* state processor. It's coming from down here... it reads the `id` up here and tries to find that in the database... but it's not there. If we *had* used a valid `id`, it would have queried for that *other* `User` entity... then we would have updated the properties on *that*!. That's a *big* no-no. At least with how our code is written, by making `id` writable, we're allowing the user to *change* which user is being modified.

Let's look at the full flow. First, our provider found the original `User` entity with the `id` from the URL... and mapped that over to a `UserApi` object. Good so far. Then, during deserialization,

the `id` on the `UserApi` object was *changed* to `47`. Finally, in the state processor, we tried to query for an entity with `id=47`... which is *ultimately* what we would have saved to the database.

Over in `UserApi`, to fix this, above `id`, add `writable: false`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 24
25 class UserApi
26 {
27     #[ApiProperty(readable: false, writable: false, identifier: true)]
28     public ?int $id = null;
↕ // ... lines 29 - 45
46 }
```

Or we could use the `#[Ignore]` attribute that we saw a second ago... since we don't want this to be readable *or* writable. The `id` property helps generate the IRI... but it's not *really* part of our API.

If we run that test now... it *passes* because it's *ignoring* the new `id` field in the JSON. Life is *good*.

While we're here, in `UserApi`, there are two other properties that, for now, I want to make read-only. Above `$dragonTreasures`, make this `writable: false`... though we *are* going to make this writable later.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 24
25 class UserApi
26 {
↕ // ... lines 27 - 42
43     #[ApiProperty(writable: false)]
44     public array $dragonTreasures = [];
↕ // ... lines 45 - 47
48 }
```

Below, do the same for `$flameThrowingDistance`... because this is a fake property that we're generating as a random number.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 24
25 class UserApi
26 {
↕ // ... lines 27 - 42
43     #[ApiProperty(writable: false)]
44     public array $dragonTreasures = [];
↕ // ... line 45
46     #[ApiProperty(writable: false)]
47     public int $flameThrowingDistance = 0;
48 }
```

Using "security" to hide/show a field

Oh, and another way to control whether a field is readable or writable is the `security` attribute. For example, if `$flameThrowingDistance` were only readable or writable if you had a certain *role*, you could use the `security` attribute to check for that. We'll see this a bit later.

Different Input/Output Classes?

Finally, I want to mention one last strategy for conditional fields... even though we won't do it. If the input JSON and output JSON for your API resource start to look *really* different, it *is* possible to have separate classes for your input and your output. You could have something like a `UserApiRead` and a *separate* `UserApiWrite`. The `UserApiRead` would be used for the *read* operations like `GET` and `GET` collection. And `UserApiWrite` would be used for `PUT`, `PATCH`, and `POST` operations.

Though, full disclosure: I haven't actually played with this yet. It should work, but there are probably some road bumps and details along the way. One other thing to keep in mind is that, on `UserApiWrite`, you could, in theory, set the `output` to `UserApiRead`. That would allow the user to send data in the format of `UserApiWrite`, but be returned JSON from `UserApiRead`. But, to make this work, after saving the `UserApiWrite` in your state processor, you would need to turn it into a `UserApiRead` and return *that*.

Anyway, that's definitely more advanced, but if it's interesting, and you try it, let me know!

Next up: Let's polish our new API resource by re-adding validation and security.

Chapter 24: DTO Validation & Security

Let's talk about *validation*! When we `->post()` to our endpoint, the internal object will be our `UserApi` object... which means *that's* what will be validated. Watch. Send *no* fields to the `POST` request... and *run* that test:

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

Oh uh: 500 error! And... I bet you can guess why. It says:

> *"`User::setEmail()`: Argument #1 (`$email`) must be of type string"*

Coming from our state processor on line 59. Because there are *no* validation constraints *at all* on `UserApi`, the `email` property remains `null`. Then, over here on line 59, we try to transfer that null `email` onto our entity. It doesn't like that, there's a short fist fight, and we see this error. And even if it *did* accept a null value, it would eventually fail in the database because the email isn't allowed to be null *there*.

We're missing *validation*. Fortunately, it's easy to add... once you know that validation will happen on the `UserApi` object, not the entity.

Configuration the Operations

But before we run wild and add constraints, let's specify the `operations`... so we only have the ones we need: `new Get()`, `new GetCollection()`, `new Post()`... we'll add some config to *that* in a moment... as well as `new Patch()` and `new Delete()`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 9
10 use ApiPlatform\Metadata\Delete;
11 use ApiPlatform\Metadata\Get;
12 use ApiPlatform\Metadata\GetCollection;
13 use ApiPlatform\Metadata\Patch;
14 use ApiPlatform\Metadata\Post;
↕ // ... lines 15 - 20
21 #[ApiResource(
↕ // ... line 22
23     operations: [
24         new Get(),
25         new GetCollection(),
26         new Post(
↕ // ... line 27
28         ),
29         new Patch(),
30         new Delete(),
31     ],
↕ // ... lines 32 - 35
36 )]
↕ // ... lines 37 - 39
40 class UserApi
41 {
↕ // ... lines 42 - 66
67 }
```

Back when our `User` *entity* was the `#[ApiResource]`, the `Post()` operation had an extra `validationContext` option with `groups` set to `Default` and `postValidation`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 9
10 use ApiPlatform\Metadata\Delete;
11 use ApiPlatform\Metadata\Get;
12 use ApiPlatform\Metadata\GetCollection;
13 use ApiPlatform\Metadata\Patch;
14 use ApiPlatform\Metadata\Post;
↕ // ... lines 15 - 20
21 #[ApiResource(
↕ // ... line 22
23     operations: [
24         new Get(),
25         new GetCollection(),
26         new Post(
27             validationContext: ['groups' => ['Default',
   'postValidation']],
28         ),
29         new Patch(),
30         new Delete(),
31     ],
↕ // ... lines 32 - 35
36 )]
↕ // ... lines 37 - 39
40 class UserApi
41 {
↕ // ... lines 42 - 66
67 }
```

Thanks to that, when the `Post()` operation happened, it would run all the *normal* validators *plus* any that were in this `postValidation` group. We'll see *why* we need that in a moment.

Adding the Constraints

Ok, constraint time! `$id` isn't even writable... we want `$email` to be `#[NotBlank]`... and be an `#[Email]`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 18
19 use Symfony\Component\Validator\Constraints as Assert;
↕ // ... lines 20 - 39
40 class UserApi
41 {
↕ // ... lines 42 - 44
45     #[Assert\NotBlank]
46     #[Assert\Email]
47     public ?string $email = null;
↕ // ... lines 48 - 66
67 }
```

We want `$username` to be `#[NotBlank]`...

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 18
19 use Symfony\Component\Validator\Constraints as Assert;
↕ // ... lines 20 - 39
40 class UserApi
41 {
↕ // ... lines 42 - 44
45     #[Assert\NotBlank]
46     #[Assert\Email]
47     public ?string $email = null;
48
49     #[Assert\NotBlank]
50     public ?string $username = null;
↕ // ... lines 51 - 66
67 }
```

then `$password` is an interesting one. `$password` should be *allowed* to be blank if we're doing a `PATCH` request to edit it... but *required* on a `POST` request. To accomplish that, add `#[NotBlank]` but with a `groups` option set to `postValidation`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 18
19 use Symfony\Component\Validator\Constraints as Assert;
↕ // ... lines 20 - 39
40 class UserApi
41 {
↕ // ... lines 42 - 44
45     #[Assert\NotBlank]
46     #[Assert\Email]
47     public ?string $email = null;
48
49     #[Assert\NotBlank]
50     public ?string $username = null;
↕ // ... lines 51 - 55
56     #[Assert\NotBlank(groups: ['postValidation'])]
57     public ?string $password = null;
↕ // ... lines 58 - 66
67 }
```

This constraint will only be run when we're validating the `postValidation` group... which means it will only be run for the `Post()` operation.

Okay, that should do it! Run the test now:

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

And... a beautiful 422 status code!

UniqueEntity constraint?

By the way, one of the other validation constraints we had before on the `User` entity was `#[UniqueEntity]`. That prevented someone from creating *two* users with the same `email` or `username`. I *don't* have that on `UserApi`, but we *should*. The `#[UniqueEntity]` constraint, unfortunately, only works on *entities*... so we'd need to create a custom validator to have that on `UserApi`. *We're* not going to worry about that right, but I wanted to point it out.

Anyway, back over on the test, re-add the fields. Validation, check!

Adding Security

The next thing we need to *re-add* - code that *used to* live on `User` - is *security*. Up here on the API level, for the *entire* resource, require `is_granted("ROLE_USER")`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 20
21 #[ApiResource(
↕ // ... lines 22 - 35
36     security: 'is_granted("ROLE_USER")',
↕ // ... lines 37 - 39
40 )]
↕ // ... lines 41 - 43
44 class UserApi
45 {
↕ // ... lines 46 - 70
71 }
```

This means that we need to be logged in to use *any* of the operations for this resource... *by default*. Then we *overrode* that. In `Post()`, we definitely can't be logged in yet because we're *registering* our user. Say, `security` set to `is_granted("PUBLIC_ACCESS")` which is a special attribute that will always pass.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 20
21 #[ApiResource(
↕ // ... line 22
23     operations: [
↕ // ... lines 24 - 25
26         new Post(
27             security: 'is_granted("PUBLIC_ACCESS")',
↕ // ... line 28
29         ),
↕ // ... lines 30 - 33
34     ],
↕ // ... line 35
36     security: 'is_granted("ROLE_USER")',
↕ // ... lines 37 - 39
40 )]
↕ // ... lines 41 - 43
44 class UserApi
45 {
↕ // ... lines 46 - 70
71 }
```

Down here for `Patch()`, we had `security('is_granted("ROLE_USER_EDIT")')`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 20
21 #[ApiResource(
↕ // ... line 22
23     operations: [
↕ // ... lines 24 - 25
26         new Post(
27             security: 'is_granted("PUBLIC_ACCESS")',
↕ // ... line 28
29         ),
30         new Patch(
31             security: 'is_granted("ROLE_USER_EDIT")'
32         ),
↕ // ... line 33
34     ],
↕ // ... line 35
36     security: 'is_granted("ROLE_USER")',
↕ // ... lines 37 - 39
40 )]
↕ // ... lines 41 - 43
44 class UserApi
45 {
↕ // ... lines 46 - 70
71 }
```

In our app, we decided that you need to have this special tole to be able to edit users.

Ok! Let's run *all* the tests for `User`:

```
symfony php bin/phpunit tests/Functional/UserResourceTest.php
```

And... *oh*. Not bad! Three out of four! The failure comes from `testTreasuresCannotBeStolen()`. That doesn't sound good!

If we check that out... this is a interesting test: we `->patch()` to update a `$user`, and then try to set the `dragonTreasures` property to a treasure that is owned by a *different* user. You can see that this `$dragonTreasure` is owned by `$otherUser`... but we're currently updating `$user`.

What we're attempting to do is *steal* this `$dragonTreasure` from `$otherUser` and make it part of `$user`. Dragons do *not* appreciate being robbed, so we're asserting that this is a 422 status code... because *previously*, we had a custom validator that prevented this.

Well, it still exists - it's this `TreasuresAllowedOwnerChangeValidator` - but it's not being applied to `UserApi`... and it needs to be *updated* to work with it. We'll *do* this later.

More importantly right now, the `dragonTreasures` property isn't even *writable*! In `UserApi`, above `$dragonTreasures`, we have `writable: false`. In a bit, we're going to change that so that we *can* write `dragonTreasures` again. And when we do, we'll bring back that validator and make sure this test passes.

Next: If you look at the processor *or* the provider we created, these classes are pretty generic. They could *almost* work for `UserApi` *and* a future `DragonTreasureApi` class... and *any* other DTO class we create that's tied to an entity. The only part that's *specific* to `User` is the code that maps *to* and *from* the `User` entity and the `UserApi` class.

If we could handle that mapping... in some system that lives *outside* our provider and processor... we *could* reuse them. Let's make this a reality next!

Chapter 25: MicroMapper: Central DTO Mapping

Doing the data transformation, from `UserApi` to the `User` entity, or the `User` entity to `UserApi`, is the *only* part of our provider and processor that *isn't* generic and reusable. Rats! If it wasn't for that code, we could create a `DragonTreasureApi` class and do this whole thing over again with, like almost no work! *Fortunately*, this is a well-known problem called "data mapping".

For this tutorial, I tried a few data mapping libraries, most notably `jane-php/automapper-bundle`, which is super-fast, advanced, *and* fun to use. However, it isn't *quite* as flexible as I needed... and extending it looked complex. Honestly... I got stuck in a few places... though I know that work *is* being done to make this package even friendlier.

The point is, we're not going to use that library. *Instead*, to handle the mapping, I created a small package of my own. It's easy to understand, and gives us *full* control... even if it's not *quite* as cool as jane's automapper.

Installing micro-mapper

So let's get it installed! Run:

```
composer require symfonycasts/micro-mapper
```

That kind of sounds like a superhero. Now that we have this in our app, we have one new micromapper service that's good at converting data from *one* object to another. Let's start by using it in our *processor*.

Using the MicroMapper Service

Up on top, autowire a `private MicroMapperInterface $microMapper`.

src/State/EntityClassDtoStateProcessor.php

```
// ... lines 1 - 14
use Symfonycasts\MicroMapper\MicroMapperInterface;

class EntityClassDtoStateProcessor implements ProcessorInterface
{
    public function __construct(
// ... lines 20 - 23
        private MicroMapperInterface $microMapper
    )
    {

    }
// ... lines 29 - 51
}
```

And down here, for all the mapping stuff, copy the existing logic, because we'll need it in a minute. Replace it with `return $this->microMapper->map()`. This has two main arguments: The `$from` object, which will be `$dto` and the *toClass*, so `User::class`.

src/State/EntityClassDtoStateProcessor.php

```
// ... lines 1 - 14
use Symfonycasts\MicroMapper\MicroMapperInterface;

class EntityClassDtoStateProcessor implements ProcessorInterface
{
    public function __construct(
// ... lines 20 - 23
        private MicroMapperInterface $microMapper
    )
    {

    }
// ... lines 29 - 47
    private function mapDtoToEntity(object $dto): object
    {
        return $this->microMapper->map($dto, User::class);
    }
}
```

Done! Well... not *quite*, but let's try running `testPostToCreateUser` anyway.

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

And... it *fails* with a *500* error. The interesting thing is *what* that 500 error says. Let's "View Page Source" so we can read this even better. It says

> *"No mapper found for `App\UserResource\UserApi` -> `App\Entity\User`"*

And this comes from `MicroMapper`. This basically says:

> *"Hey, I don't know how to convert a `UserApi` object to a `User` object! Halp!"*

Creating a Mapper

MicroMapper *isn't* magic... it's really the opposite. To teach micro mapper how to do this conversion, we need to create a class that *explains* what we want. That's called a *mapper class*. And these are fun!

Let me start by closing a few things... and then creating a new `Mapper/` directory in `src/`. Inside of *that*, add a new PHP class called... how about `UserApiToEntityMapper`, because we're going from `UserApi` to the `User` entity.

This class needs 2 things. First, to implement `MapperInterface`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 2
3 namespace App\Mapper;
↕ // ... lines 4 - 7
8 use Symfonycasts\MicroMapper\MapperInterface;
↕ // ... lines 9 - 10
11 class UserApiToEntityMapper implements MapperInterface
12 {
↕ // ... lines 13 - 22
23 }
```

And second, above the class, to describe what it's mapping *to* and *from*, we need an `#[AsMapper()]` attribute with `from: UserApi::class` and `to: User::class`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 2
3 namespace App\Mapper;
4
5 use App\ApiResource\UserApi;
6 use App\Entity\User;
7 use Symfonycasts\MicroMapper\AsMapper;
8 use Symfonycasts\MicroMapper\MapperInterface;
9
10 #[AsMapper(from: UserApi::class, to: User::class)]
11 class UserApiToEntityMapper implements MapperInterface
12 {
↕ // ... lines 13 - 22
23 }
```

To help the interface, go to "Code Generate" (or "command" + "N" on a Mac) and generate the two *methods* it needs: `load()` and `populate()`. For starters, let's `dd($from, $toClass)`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 2
3 namespace App\Mapper;
4
5 use App\ApiResource\UserApi;
6 use App\Entity\User;
7 use Symfonycasts\MicroMapper\AsMapper;
8 use Symfonycasts\MicroMapper\MapperInterface;
9
10 #[AsMapper(from: UserApi::class, to: User::class)]
11 class UserApiToEntityMapper implements MapperInterface
12 {
13     public function load(object $from, string $toClass, array $context):
   object
14     {
15         dd($from, $toClass);
16         // TODO: Implement load() method.
17     }
18
19     public function populate(object $from, object $to, array $context):
   object
20     {
21         // TODO: Implement populate() method.
22     }
23 }
```

Now, *just* by creating this and giving it `#[AsMapper]`, when we use MicroMapper to do this transformation, it *should* call our `load()` method. Let's see if it does!

Run the test:

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

And... got it! *There's* the `UserApi` object we're passing, and *it's* passing *us* the `User` class. The purpose of `load()` is to *load* the `$toClass` object and return it, like by querying for a `User` entity or creating a new one.

To do the query, on top, add `public function __construct()` and inject the normal `UserRepository $userRepository`. Down here, this will hold the same code that we saw earlier. I like to say `$dto = $from` and `assert($dto instanceof UserApi)`. That helps my brain *and* my editor.

Next, *if* our `$dto` has an `id`, then call `$this->userRepository->find($dto->id)`. *Else*, create a brand `new User()` object.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 6
7  use App\Repository\UserRepository;
↕ // ... lines 8 - 11
12 class UserApiToEntityMapper implements MapperInterface
13 {
14     public function __construct(
15         private UserRepository $userRepository,
16     )
17     {
18     }
19
20     public function load(object $from, string $toClass, array $context):
   object
21     {
22         $dto = $from;
23         assert($dto instanceof UserApi);
24
25         $userEntity = $dto->id ? $this->userRepository->find($dto->id) :
   new User();
↕ // ... lines 26 - 30
31     }
↕ // ... lines 32 - 36
37 }
```

It's *that* simple. And if, for some reason, we don't have a `$userEntity`, `throw new \Exception('User not found')`, similar to what we did before. Down here, `return $userEntity`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 6
7  use App\Repository\UserRepository;
↕ // ... lines 8 - 11
12 class UserApiToEntityMapper implements MapperInterface
13 {
14     public function __construct(
15         private UserRepository $userRepository,
16     )
17     {
18     }
19
20     public function load(object $from, string $toClass, array $context):
   object
21     {
22         $dto = $from;
23         assert($dto instanceof UserApi);
24
25         $userEntity = $dto->id ? $this->userRepository->find($dto->id) :
   new User();
26         if (!$userEntity) {
27             throw new \Exception('User not found');
28         }
29
30         return $userEntity;
31     }
↕ // ... lines 32 - 36
37 }
```

So we've initialized our `$to` object and returned it. And that's the point of `load()`: to do the *least* amount of work to get the `$to` object... but *without* populating the data.

Internally, after calling `load()`, micro mapper will *then* call `populate()` and pass us the `User` entity object that we just returned. To see this, let's `dd($from, $to)`.

src/Mapper/UserApiToEntityMapper.php

```
// ... lines 1 - 6
use App\Repository\UserRepository;
// ... lines 8 - 11
class UserApiToEntityMapper implements MapperInterface
{
    public function __construct(
        private UserRepository $userRepository,
    )
    {
    }

    public function load(object $from, string $toClass, array $context): object
    {
        $dto = $from;
        assert($dto instanceof UserApi);

        $userEntity = $dto->id ? $this->userRepository->find($dto->id) : new User();
        if (!$userEntity) {
            throw new \Exception('User not found');
        }

        return $userEntity;
    }

    public function populate(object $from, object $to, array $context): object
    {
        dd($from, $to);
    }
}
```

Run that test:

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

Perfect! Here's our "from" `UserApi` object, and the *new* `User` entity.

Now... you might be wondering why we have both a `load()` method *and* a `populate()` method... when it seems like these could just be *one* method. And you'd mostly be right! But there's a technical reason why they're separated, and it'll come in handy later when we talk

about relationships. But for now, you can imagine these two methods are really just one, continuous process: `load()` is called, then `populate()`.

And no surprise, *this* is where we will take the data from the `$from` object and put it onto the `$to` object. Once again, to keep me sane, I'll say `$dto = $from` and `assert($dto instanceof UserApi)`... then `$entity = $to` and `assert($entity instanceof User)`.

The code down here is going to be really boring... so I'll paste it. At the bottom, `return $entity`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 12
13 class UserApiToEntityMapper implements MapperInterface
14 {
↕ // ... lines 15 - 34
35     public function populate(object $from, object $to, array $context): object
36     {
37         $dto = $from;
38         assert($dto instanceof UserApi);
39         $entity = $to;
40         assert($entity instanceof User);
41
42         $entity->setEmail($dto->email);
43         $entity->setUsername($dto->username);
44         if ($dto->password) {
45             $entity->setPassword($this->userPasswordHasher->hashPassword($entity, $dto->password));
46         }
47         // TODO dragonTreasures if we change them to writeable
48
49         return $entity;
50     }
51 }
```

We're using `$this->userPasswordHasher` here... so we *also* need to make sure, at the top, to add `private UserPasswordHasherInterface $userPasswordHasher`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 7
8  use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;
↕ // ... lines 9 - 12
13 class UserApiToEntityMapper implements MapperInterface
14 {
15     public function __construct(
↕ // ... line 16
17         private UserPasswordHasherInterface $userPasswordHasher,
18     )
19     {
20     }
↕ // ... lines 21 - 34
35     public function populate(object $from, object $to, array $context): object
36     {
37         $dto = $from;
38         assert($dto instanceof UserApi);
39         $entity = $to;
40         assert($entity instanceof User);
41
42         $entity->setEmail($dto->email);
43         $entity->setUsername($dto->username);
44         if ($dto->password) {
45             $entity->setPassword($this->userPasswordHasher->hashPassword($entity, $dto->password));
46         }
47         // TODO dragonTreasures if we change them to writeable
48
49         return $entity;
50     }
51 }
```

So this is basically the same code we had before... but in a different spot.

Let's see what the test thinks!

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

It *passes*! This is *huge*! We've offloaded this work to our mapper... which means our processor is almost completely generic. Now we can remove the `UserPasswordHasher` that we don't need anymore... and the `UserRepository` up here. We can even remove those `use` statements.

We still *do* need to write the mapping code, but now it lives in a nice, central location.

Mapping the Other Direction

Ready to *repeat* this for the provider. Close the processor... and open it up. This time, we're going from the `User` entity to `UserApi`. Copy all of this code, *delete* it and, just like before, autowire `MicroMapperInterface $microMapper`. Down here, this simplifies to `return $this->microMapper->map()` going from our `$entity` to `UserApi::class`.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 11
12 use App\ApiResource\UserApi;
↕ // ... line 13
14 use Symfonycasts\MicroMapper\MicroMapperInterface;
↕ // ... line 15
16 class EntityToDtoStateProvider implements ProviderInterface
17 {
18     public function __construct(
↕ // ... lines 19 - 20
21         private MicroMapperInterface $microMapper
22     )
23     {
24
25     }
↕ // ... lines 26 - 54
55     private function mapEntityToDto(object $entity): object
56     {
57         return $this->microMapper->map($entity, UserApi::class);
58     }
59 }
```

Sweet! If we tried this now, we'd get a 500 error because we don't have a mapper for it. Back in `src/Mapper/`, create a new class called `UserEntityToApiMapper`... implement `MapperInterface`... and above the class, add `#[AsMapper()]`. In this case, we're going `from: User::class`, `to: UserApi::class`.

src/Mapper/UserEntityToApiMapper.php

```
↕ // ... lines 1 - 2
3 namespace App\Mapper;
4
5 use App\ApiResource\UserApi;
6 use App\Entity\User;
7 use Symfonycasts\MicroMapper\AsMapper;
8 use Symfonycasts\MicroMapper\MapperInterface;
9
10 #[AsMapper(from: User::class, to: UserApi::class)]
11 class UserEntityToApiMapper implements MapperInterface
12 {
↕ // ... lines 13 - 37
38 }
```

Implement both of the methods we need... and we start pretty much the same way as before, with `$entity = $from` and `assert($entity instanceof User)`.

src/Mapper/UserEntityToApiMapper.php

```
↕ // ... lines 1 - 2
3 namespace App\Mapper;
4
5 use App\ApiResource\UserApi;
6 use App\Entity\User;
7 use Symfonycasts\MicroMapper\AsMapper;
8 use Symfonycasts\MicroMapper\MapperInterface;
9
10 #[AsMapper(from: User::class, to: UserApi::class)]
11 class UserEntityToApiMapper implements MapperInterface
12 {
13     public function load(object $from, string $toClass, array $context):
   object
14     {
15         $entity = $from;
16         assert($entity instanceof User);
↕ // ... lines 17 - 21
22     }
↕ // ... lines 23 - 37
38 }
```

Down here, to create the DTO, we don't need to do any queries. We're *always* going to instantiate a fresh new `UserApi()`. Set the ID onto it with `$dto->id = $entity->getId()`... then `return $dto`.

src/Mapper/UserEntityToApiMapper.php

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

use App\ApiResource\UserApi;
use App\Entity\User;
use Symfonycasts\MicroMapper\AsMapper;
use Symfonycasts\MicroMapper\MapperInterface;

#[AsMapper(from: User::class, to: UserApi::class)]
class UserEntityToApiMapper implements MapperInterface
{
    public function load(object $from, string $toClass, array $context): object
    {
        $entity = $from;
        assert($entity instanceof User);

        $dto = new UserApi();
        $dto->id = $entity->getId();

        return $dto;
    }
// ... lines 23 - 37
}
```

Ok, the job of the `load()` method is *really* to create the `$to` object and... *at least* make sure it has its identifier if there is one.

Everything else we need to do is down here in `populate()`. Start our usual way: `$entity = $from`, `$dto = $to` and two asserts: `assert($entity instanceof User)` and `assert($dto instanceof UserApi)`. Below that, use the exact code we had before. We're just transferring the data. At the bottom, `return $dto`.

src/Mapper/UserEntityToApiMapper.php

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

use App\ApiResource\UserApi;
use App\Entity\User;
use Symfonycasts\MicroMapper\AsMapper;
use Symfonycasts\MicroMapper\MapperInterface;

#[AsMapper(from: User::class, to: UserApi::class)]
class UserEntityToApiMapper implements MapperInterface
{
    public function load(object $from, string $toClass, array $context): object
    {
        $entity = $from;
        assert($entity instanceof User);

        $dto = new UserApi();
        $dto->id = $entity->getId();

        return $dto;
    }

    public function populate(object $from, object $to, array $context): object
    {
        $entity = $from;
        $dto = $to;
        assert($entity instanceof User);
        assert($dto instanceof UserApi);

        $dto->email = $entity->getEmail();
        $dto->username = $entity->getUsername();
        $dto->dragonTreasures = $entity->getPublishedDragonTreasures()->getValues();
        $dto->flameThrowingDistance = rand(1, 100);

        return $dto;
    }
}
```

Phew! Let's try this! Head over to your browser, refresh this page, and... *oh*...

> *“Full authentication is required to access this resource.”*

Of course. That's because we added security! Head back over to the homepage, click this username and password shortcut... *boop*... and *now* try to refresh that page. It *works*! We *are* missing some of the data, though, which is my fault.

I said `$dto = new UserApi()`. So instead of *modifying* the `$to` object I'm being passed, I created a *new* one... and the original wasn't modified. There we go. If I try it again... *much* better.

So this is *huge* people! Our provider and processor are now generic! Let's finish the process of making them work for *any* API resource class *next*

Chapter 26: Reusable Entity->Dto Provider & Processor

Our `UserAPI` is now a *fully functional* API resource class! We've got our `EntityToDtoStateProvider`, which calls the core state provider from Doctrine, and *that* gives us all the good stuff, like querying, filtering, and pagination. Then, down here, we leverage the MicroMapper system to convert the `$entity` objects into `UserApi` objects.

And we do the *same* thing in the processor. We use MicroMapper to go *from* `UserApi` *to* our `User` entity... then call the core Doctrine state processor to let *it* do the saving or deleting. I love that!

Our *dream* is to create a `DragonTreasureApi` and repeat *all* of this magic. And if we can make these processor and provider classes *completely* generic... that's going to be *super* easy. So let's do it!

Making the Provider Generic

Start in the provider. If you search for "user", there's only one spot: where we tell MicroMapper which class to convert our `$entity` into. Can... we fetch this *dynamically*? Up here, our provider receives the `$operation` and `$context`. Let's dump *both* of these.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 15
16 class EntityToDtoStateProvider implements ProviderInterface
17 {
↕ // ... lines 18 - 26
27     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
28     {
29         dd($operation, $context);
↕ // ... lines 30 - 53
54     }
↕ // ... lines 55 - 59
60 }
```

Since this is in our *provider*... we can just go refresh the Collection endpoint and... *boom*! This is a `GetCollection` operation... and check it out. The operation object stores the ApiResource *class* that it's attached to!

So over here, it's simple: `$resourceClass = $operation->getClass()`. Now that we've got that, down here, make it an argument - `string $resourceClass` - and pass *that* instead. Finally, we need to add `$resourceClass` as the argument when we call `mapEntityToDto()` there... *and* right there. Remove the `use` statement we don't need anymore and... just like that... it *still* works!

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 14
15 class EntityToDtoStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 25
26     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
27     {
28         $resourceClass = $operation->getClass();
29         if ($operation instanceof CollectionOperationInterface) {
↕ // ... lines 30 - 33
34             foreach ($entities as $entity) {
35                 $dtos[] = $this->mapEntityToDto($entity, $resourceClass);
36             }
↕ // ... lines 37 - 43
44         }
↕ // ... lines 45 - 51
52         return $this->mapEntityToDto($entity, $resourceClass);
53     }
54
55     private function mapEntityToDto(object $entity, string
   $resourceClass): object
56     {
57         return $this->microMapper->map($entity, $resourceClass);
58     }
59 }
```

Making the Processor Generic

We're on a roll! Head to the *processor* and search for "user". Ah, we have the *same* problem except, this time, we need the `User` entity class.

Ok! Up on top, `dd($operation)`. And for this, we need to run one of our tests:

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 14
15 class EntityClassDtoStateProcessor implements ProcessorInterface
16 {
↕ // ... lines 17 - 25
26     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
27     {
28         dd($operation);
↕ // ... lines 29 - 43
44     }
↕ // ... lines 45 - 49
50 }
```

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

And... got it! We see the `Post` operation... and the class is, of course, `UserApi`. But this time we need the `User` class. Remember: in `UserApi`, we use `stateOptions` to say that `UserApi` is tied to the `User` entity. And now, we can *read* this info from the operation. If we scroll down a bit... there it is: the `stateOptions` property with the `Options` object, and `entityClass` inside.

Cool! Back in the processor, towards the top... remove the `dd()` and start with `$stateOptions = $operation->getStateOptions()`. Then, to help my editor (and also in case I misconfigure something), `assert($stateOptions instanceof Options)` (the one from Doctrine ORM).

You *can* use *different* `Options` classes for `$stateOptions`... like if you're getting data from ElasticSearch, but *we* know we're using *this* one from Doctrine. Below, say `$entityClass = $stateOptions->getEntityClass()`.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 15
16 class EntityClassDtoStateProcessor implements ProcessorInterface
17 {
↕ // ... lines 18 - 26
27     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
28     {
29         $stateOptions = $operation->getStateOptions();
30         assert($stateOptions instanceof Options);
31         $entityClass = $stateOptions->getEntityClass();
↕ // ... lines 32 - 46
47     }
↕ // ... lines 48 - 52
53 }
```

And... we don't need this `assert()` down here, then pass `$entityClass` to `mapDtoToEntity()`. Finally, use that with `string $entityClass`... and also pass it here.

src/State/EntityClassDtoStateProcessor.php

```
↕ // ... lines 1 - 15
16 class EntityClassDtoStateProcessor implements ProcessorInterface
17 {
↕ // ... lines 18 - 26
27     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
28     {
29         $stateOptions = $operation->getStateOptions();
30         assert($stateOptions instanceof Options);
31         $entityClass = $stateOptions->getEntityClass();
↕ // ... lines 32 - 46
47     }
↕ // ... lines 48 - 52
53 }
```

When we search for "user" now... we can get rid of the two `use` statements... and... we're clean! It's generic! Try the test!

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

That's it! We're ready! We have a reusable provider and processor! *Next*, let's create a `DragonTreasureApi` class, repeat this magic, and see how quickly we can get things to fall

into place!

Chapter 27: Quick! Create a DragonTreasure DTO

Time to convert our `DragonTreasure` ApiResource into a *proper* DTO class! We'll start by deleting a *ton* of stuff: *everything* related to API Platform in `DragonTreasure`... so we have a clean slate to start from. We'll add back what we need little-by-little. Goodbye filter stuff... the validators... all the serialization group stuff... and then we can do some cleanup on our properties. We had some *fairly* complex code in here... and while we won't add *all* of it back, we *will* add the most important things.

src/Entity/DragonTreasure.php

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

use App\Repository\DragonTreasureRepository;
use Carbon\Carbon;
use Doctrine\DBAL\Types\Types;
use Doctrine\ORM\Mapping as ORM;
use function Symfony\Component\String\u;

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

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

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

    /**
     * The estimated value of this treasure, in gold coins.
     */
    #[ORM\Column]
    private ?int $value = 0;

    #[ORM\Column]
    private ?int $coolFactor = 0;

    #[ORM\Column]
    private \DateTimeImmutable $plunderedAt;

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

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

    /**
     * @var bool Non-persisted property to help determine if the treasure is owned by the authenticated user
     */
```

```
47      private bool $isOwnedByAuthenticatedUser = false;
↕   // ... lines 48 - 169
170 }
```

Lemme scroll down to make sure we got everything. Yea, that should be it! We *now* have a good old-fashioned, *boring* entity class. In `src/ApiPlatform/`, let's also delete `AdminGroupsContextBuilder`. This was a complex way to make fields readable or writable by our admin... but we're going to solve that with `ApiProperty` security. Also get rid of the custom normalizer... which added a field and an extra group. And finally, remove the custom `DragonTreasureStateProvider` and `DragonTreasureStateProcessor` classes.

Query Extensions are Still Called!

But we *did* keep *one* thing: `DragonTreasureIsPublishedExtension`. Because the new system will *still* use the core Doctrine `CollectionProvider`, this query extension stuff will *continue* to work and be called. That's just one less thing we need to worry about.

Head over and refresh the documentation. Ok! Only `Quest` and `User`. Though, you may notice some `DragonTreasure` stuff down here... because `UserApi` has a relation to the `DragonTreasure` entity. So even though `DragonTreasure` isn't an API resource, API Platform still tries to document what that *field* is on `User`. It doesn't really matter, because we're going to fix that and *completely* use API classes everywhere

Creating the DTO Class

In `src/ApiResource/`, create the new class: `DragonTreasureApi`.

src/ApiResource/DragonTreasureApi.php

```
↕   // ... lines 1 - 2
3   namespace App\ApiResource;
↕   // ... lines 4 - 18
19  class DragonTreasureApi
20  {
↕   // ... lines 21 - 24
25  }
```

Next, in `UserApi`, steal some of the basic code from our `#[ApiResource]`... paste that over here, and, for now, delete `operations`. We can also get rid of these `use` statements. Perfect!

We *will* use a `shortName` - `Treasure` - give this `10` items per page, and remove the `security` line. The *most* important thing is that we have `provider` and `processor` (just as they are here), and `stateOptions`, which will point to `DragonTreasure::class`.

src/ApiResource/DragonTreasureApi.php

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

use ApiPlatform\Doctrine\Orm\State\Options;
// ... line 6
use ApiPlatform\Metadata\ApiResource;
use App\Entity\DragonTreasure;
use App\State\EntityClassDtoStateProcessor;
use App\State\EntityClassDtoStateProvider;

#[ApiResource(
    shortName: 'Treasure',
    paginationItemsPerPage: 10,
    provider: EntityClassDtoStateProvider::class,
    processor: EntityClassDtoStateProcessor::class,
    stateOptions: new Options(entityClass: DragonTreasure::class),
)]
class DragonTreasureApi
{
// ... lines 21 - 24
}
```

Also grab the `$id` property. Like before, we don't *really* want this to be part of our API, so it's `readable: false` and `writable: false`. Down here, add `public ?string $name = null`.

src/ApiResource/DragonTreasureApi.php

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

use ApiPlatform\Doctrine\Orm\State\Options;
use ApiPlatform\Metadata\ApiProperty;
use ApiPlatform\Metadata\ApiResource;
use App\Entity\DragonTreasure;
use App\State\EntityClassDtoStateProcessor;
use App\State\EntityClassDtoStateProvider;

#[ApiResource(
    shortName: 'Treasure',
    paginationItemsPerPage: 10,
    provider: EntityClassDtoStateProvider::class,
    processor: EntityClassDtoStateProcessor::class,
    stateOptions: new Options(entityClass: DragonTreasure::class),
)]
class DragonTreasureApi
{
    #[ApiProperty(readable: false, writable: false, identifier: true)]
    public ?int $id = null;

    public ?string $name = null;
}
```

Great start! We have one tiny class and... what the heck, let's just go try it! Refresh the docs. Yes! Our Treasure operations are here! If we try the collection endpoint... we get:

> *"No mapper found for `DragonTreasure` -> `DragonTreasureApi`"*

Adding the Mapper Class

That's fantastic! The only real work we need to do is implement those mappers. So let's go!

In the `src/Mapper/` directory, create a class called `DragonTreasureEntityToApiMapper`. We've done this before: implement `MapperInterface` and add the `#[AsMapper()]` attribute. We're going `from: DragonTreasure::class` `to: DragonTreasureApi::class`.

src/Mapper/DragonTreasureEntityToApiMapper.php

```
↕ // ... lines 1 - 2
3 namespace App\Mapper;
4
5 use App\ApiResource\DragonTreasureApi;
6 use App\Entity\DragonTreasure;
7 use Symfonycasts\MicroMapper\AsMapper;
8 use Symfonycasts\MicroMapper\MapperInterface;
9
10 #[AsMapper(from: DragonTreasure::class, to: DragonTreasureApi::class)]
11 class DragonTreasureEntityToApiMapper implements MapperInterface
12 {
↕ // ... lines 13 - 34
35 }
```

And *just* like that, micro mapper knows to use this. Generate the two methods for the interface: `load()` and `populate()`. For sanity, add `$entity = $from`, and `assert()` that `$entity` is an `instanceof DragonTreasure`.

src/Mapper/DragonTreasureEntityToApiMapper.php

```
↕ // ... lines 1 - 2
3 namespace App\Mapper;
4
5 use App\ApiResource\DragonTreasureApi;
6 use App\Entity\DragonTreasure;
7 use Symfonycasts\MicroMapper\AsMapper;
8 use Symfonycasts\MicroMapper\MapperInterface;
9
10 #[AsMapper(from: DragonTreasure::class, to: DragonTreasureApi::class)]
11 class DragonTreasureEntityToApiMapper implements MapperInterface
12 {
13     public function load(object $from, string $toClass, array $context):
   object
14     {
15         $entity = $from;
16         assert($entity instanceof DragonTreasure);
↕ // ... lines 17 - 21
22     }
↕ // ... lines 23 - 34
35 }
```

Down here, create the DTO object with `$dto = new DragonTreasureApi()`. And remember, the job of `load()` is to create the object *and* put an identifier on it if there is one. So add `$dto->id = $entity->getId()`. Finally, `return $dto`.

src/Mapper/DragonTreasureEntityToApiMapper.php

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

use App\ApiResource\DragonTreasureApi;
use App\Entity\DragonTreasure;
use Symfonycasts\MicroMapper\AsMapper;
use Symfonycasts\MicroMapper\MapperInterface;

#[AsMapper(from: DragonTreasure::class, to: DragonTreasureApi::class)]
class DragonTreasureEntityToApiMapper implements MapperInterface
{
    public function load(object $from, string $toClass, array $context): object
    {
        $entity = $from;
        assert($entity instanceof DragonTreasure);

        $dto = new DragonTreasureApi();
        $dto->id = $entity->getId();

        return $dto;
    }
// ... lines 23 - 34
}
```

For `populate()`, steal a few lines from above that set the `$entity` variable... then also say `$dto = $to`, and add one more `assert()` that `$dto` is an `instanceof DragonTreasureApi`.

src/Mapper/DragonTreasureEntityToApiMapper.php

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

use App\ApiResource\DragonTreasureApi;
use App\Entity\DragonTreasure;
use Symfonycasts\MicroMapper\AsMapper;
use Symfonycasts\MicroMapper\MapperInterface;

#[AsMapper(from: DragonTreasure::class, to: DragonTreasureApi::class)]
class DragonTreasureEntityToApiMapper implements MapperInterface
{
    public function load(object $from, string $toClass, array $context): object
    {
        $entity = $from;
        assert($entity instanceof DragonTreasure);

        $dto = new DragonTreasureApi();
        $dto->id = $entity->getId();

        return $dto;
    }

    public function populate(object $from, object $to, array $context): object
    {
        $entity = $from;
        $dto = $to;
        assert($entity instanceof DragonTreasure);
        assert($dto instanceof DragonTreasureApi);
// ... lines 30 - 33
    }
}
```

The only property we have on our DTO right now is `name`, so all we need is `$dto->name = $entity->getName()`. At the end, `return $dto`.

src/Mapper/DragonTreasureEntityToApiMapper.php

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

use App\ApiResource\DragonTreasureApi;
use App\Entity\DragonTreasure;
use Symfonycasts\MicroMapper\AsMapper;
use Symfonycasts\MicroMapper\MapperInterface;

#[AsMapper(from: DragonTreasure::class, to: DragonTreasureApi::class)]
class DragonTreasureEntityToApiMapper implements MapperInterface
{
    public function load(object $from, string $toClass, array $context): object
    {
        $entity = $from;
        assert($entity instanceof DragonTreasure);

        $dto = new DragonTreasureApi();
        $dto->id = $entity->getId();

        return $dto;
    }

    public function populate(object $from, object $to, array $context): object
    {
        $entity = $from;
        $dto = $to;
        assert($entity instanceof DragonTreasure);
        assert($dto instanceof DragonTreasureApi);

        $dto->name = $entity->getName();

        return $dto;
    }
}
```

And, people! We just created a class that maps from the entity to the DTO... and our state provider is using micro mapper internally... so I think this should... just work!

And... it *does*! Wow! With *just* the API Resource class and this *one* mapper, we now have a database-powered *custom* API Resource class. Woo!

Adding A Relation Field

Now things get *interesting*. Every `DragonTreasure` entity has an *owner*, which is a *relationship* to the `User` entity. In our API, we're going to have the same relationship. But instead of this being a relation from `DragonTreasureApi` to a `User` *entity* object, it will be to a `UserApi` object.

Check it out! Say `public ?UserApi $owner = null`.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 18
19 class DragonTreasureApi
20 {
↕ // ... lines 21 - 25
26     public ?UserApi $owner = null;
27 }
```

Then let's go populate that in the mapper. Down here, say `$dto->owner =`... but... hold on a second. This isn't as simple as saying `$entity->getOwner()`, because that's a *user entity object*. We need a `UserApi` object! Can you think of anything that's really good at converting a `User` entity to `UserApi`? That's right, MicroMapper!

Up here on top, inject `private MicroMapperInterface $microMapper`... and, down here, say `$dto->owner = $this->microMapper->map()` to map from `$entity->getOwner()` - the `User` entity object - to `UserApi::class`.

src/Mapper/DragonTreasureEntityToApiMapper.php

```
↕ // ... lines 1 - 9
10 use Symfonycasts\MicroMapper\MicroMapperInterface;
↕ // ... lines 11 - 12
13 class DragonTreasureEntityToApiMapper implements MapperInterface
14 {
15     public function __construct(
16         private MicroMapperInterface $microMapper,
17     )
18     {
19     }
↕ // ... lines 20 - 31
32     public function populate(object $from, object $to, array $context):
   object
33     {
↕ // ... lines 34 - 39
40         $dto->owner = $this->microMapper->map($entity->getOwner(),
   UserApi::class);
↕ // ... lines 41 - 48
49     }
50 }
```

How cool is that? One thing to be aware of is that if, in *your* system, `$entity->getOwner()` might be `null`, you should code for that. Like, if you have an owner, call the mapper, else just set `owner` to `null`... or don't set it at all. For us, we're *always* going to have an owner, so this should be safe.

Let's try it! Refresh and... *oooh*. We have an `owner` field and it's an IRI. Why *is* that showing up as an IRI? Because API Platform recognizes that the `UserApi` object is an API *resource*. And how does it show API resources that are relations? That's right! It sets them as an IRI. So that's *exactly* what we wanted to see.

Adding More Fields

Let's fill in the rest of the fields we need: I'll go through this super-fast. One of the fields I'm adding is `$shortDescription`. That was a *custom* field before... but it'll be simpler now. Another custom field we had was `$isMine`, which will *also* just be a normal property.

src/ApiResource/DragonTreasureApi.php

```
// ... lines 1 - 18
class DragonTreasureApi
{
// ... lines 21 - 25
    public ?string $description = null;

    public int $value = 0;

    public int $coolFactor = 0;

    public ?UserApi $owner = null;

    public ?string $shortDescription = null;

    public ?string $plunderedAtAgo = null;

    public ?bool $isMine = null;
}
```

Over in our mapper, let's set everything. I'll speed through the boring parts. But `$shortDescription` *is* a bit interesting. Before, in `DragonTreasure`, we had a `getShortDescription()` method and *that* was exposed directly as the API field.

With the new setup, it's a normal property like anything else, and we handle setting the custom data in our mapper: `$shortDescription` is equal to `$entity->getShortDescription()`. Finally, for `$dto->isMine`, temporarily hardcode that to `true`.

src/Mapper/DragonTreasureEntityToApiMapper.php

```
↕ // ... lines 1 - 12
13 class DragonTreasureEntityToApiMapper implements MapperInterface
14 {
↕ // ... lines 15 - 31
32     public function populate(object $from, object $to, array $context): object
33     {
↕ // ... lines 34 - 40
41         $dto->description = $entity->getDescription();
42         $dto->value = $entity->getValue();
43         $dto->coolFactor = $entity->getCoolFactor();
44         $dto->shortDescription = $entity->getShortDescription();
45         $dto->plunderedAtAgo = $entity->getPlunderedAtAgo();
46         $dto->isMine = true;
↕ // ... lines 47 - 48
49     }
50 }
```

Let's check it! Refresh and... that's *beautiful*!

In `tests/Functional/`, we have `DragonTreasureResourceTest`. In *here*, we have `testGetCollectionOfTreasures()`, which tests to make sure that we only see *published* items. If our query extension is still working, this will pass. This also checks to make sure we see the correct keys.

Let's see if this works:

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

It *does*. Mind blown.

Populating the Weird isMine Field

Before we finish, let's fix the hard coded `true` on `isMine`. This is easy, but shows off just how nice it is to work with custom fields. In our mapper, this is a *service*, so we can inject *other* services like the `$security` service. *Then*, we can populate that with whatever data we want. So `isMine` is true if `$this->security->getUser()` equals the `DragonTreasure`, `getOwner()` (which is a `User` entity object).

src/Mapper/DragonTreasureEntityToApiMapper.php

```
↕ // ... lines 1 - 7
8 use Symfony\Bundle\SecurityBundle\Security;
↕ // ... lines 9 - 13
14 class DragonTreasureEntityToApiMapper implements MapperInterface
15 {
16     public function __construct(
↕ // ... line 17
18         private Security $security,
19     )
20     {
21     }
↕ // ... lines 22 - 33
34     public function populate(object $from, object $to, array $context): object
35     {
↕ // ... lines 36 - 47
48         $dto->isMine = $this->security->getUser() && $this->security->getUser() === $entity->getOwner();
↕ // ... lines 49 - 50
51     }
52 }
```

Try the test one more time to make sure this is working, and... it *is*. Woo!

Next: I want to dive deeper into relationships in our DTO-powered API. Because, if you're not careful, we can get the dreaded infinite recursion!

Chapter 28: Dtos, Mapping & Max Depth of Relations

Head to `/api/users.jsonld` to see... a circular reference coming from the serializer. Yikes! Let's think: API Platform serializes whatever we return from the state provider. So head there.... and find where the collection is created. Dump the DTOs. These are what's being serialized, so the problem must be here.

src/State/EntityToDtoStateProvider.php

```
↕ // ... lines 1 - 14
15 class EntityToDtoStateProvider implements ProviderInterface
16 {
↕ // ... lines 17 - 25
26     public function provide(Operation $operation, array $uriVariables =
   [], array $context = []): object|array|null
27     {
↕ // ... line 28
29         if ($operation instanceof CollectionOperationInterface) {
↕ // ... lines 30 - 36
37             dd($dtos);
↕ // ... lines 38 - 44
45         }
↕ // ... lines 46 - 53
54     }
↕ // ... lines 55 - 59
60 }
```

Refresh and... no surprise: we see 5 `UserApi` objects. Ah, but *here's* the problem: the `dragonTreasures` field holds an array of `DragonTreasure` *entity* objects... and each has an `owner` that points to a `User` entity... and that points *back* to a collection of `DragonTreasure` entities... which causes the serializer to serializer forever and ever. But that's not even the *real* problem! I know, I'm full of good news. The real problem is that the `UserApi` object should *really* relate to a `DragonTreasureApi`, not a `DragonTreasure` entity.

Over in `UserApi`, this will now be an `array` of `DragonTreasureApi`. Once we start going the DTO route, for maximum smoothness, we should relate DTOs to other DTOs... instead of mixing them with entities.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 42
43 class UserApi
44 {
↕ // ... lines 45 - 61
62     /**
63      * @var array<int, DragonTreasureApi>
64      */
65     #[ApiProperty(writable: false)]
66     public array $dragonTreasures = [];
↕ // ... lines 67 - 69
70 }
```

To populate the DTO objects, go to the mapper: `UserEntityToApiMapper`. Down here, for `dragonTreasures`, we can't do *this* anymore because that will give us `DragonTreasure` *entity* objects. What we basically want to do is convert *from* `DragonTreasure` to `DragonTreasureApi`. And so, once again, it's micro mapper to the rescue!

Micro-Mapping DragonTreasure -> DragonTreasureApi

Add `public function __construct()` with `private MicroMapperInterface $microMapper`. Down here, add some *fancy* code: `$dto->dragonTreasures =` set to `array_map()`, with a function that has a `DragonTreasure` argument. We'll finish that in a second... but first pass the array that it will loop over: `$entity->getPublishedDragonTreasures()->toArray()`.

So: we get an array of the published `DragonTreasure` objects and PHP loops over them and calls our function for each one - passing the `DragonTreasure`. Whatever we return will become an item inside a new array that's set onto `dragonTreasures`. And what we want to return is a `DragonTreasureApi` object. Do that with `$this->microMapper->map($dragonTreasure, DragonTreasureApi::class)`.

src/Mapper/UserEntityToApiMapper.php

```
// ... lines 1 - 4
use App\ApiResource\DragonTreasureApi;
// ... line 6
use App\Entity\DragonTreasure;
// ... lines 8 - 10
use Symfonycasts\MicroMapper\MicroMapperInterface;
// ... lines 12 - 13
class UserEntityToApiMapper implements MapperInterface
{
    public function __construct(
        private MicroMapperInterface $microMapper,
    )
    {
    }
// ... lines 21 - 41
        $dto->dragonTreasures = array_map(function(DragonTreasure $dragonTreasure) {
            return $this->microMapper->map($dragonTreasure, DragonTreasureApi::class);
        }, $entity->getPublishedDragonTreasures()->getValues());
// ... lines 45 - 47
    }
}
```

Circular Relationships

Cool! When we refresh to try it... we're greeted with a *different* circular reference problem. Fun! This one comes from MicroMapper... and it's a problem that will happen whenever you have relationships that refer to each other.

Think about it: we ask Micro Mapper to convert a `DragonTreasure` entity to `DragonTreasureApi`. *Simple.* To do that, it uses our mapper. And guess what? In our mapper, we ask it to convert the `owner` - a `User` entity - to an instance of `UserApi`. To do that, micro mapper goes back to `UserEntityToApiMapper` and... the process repeats. We're in a loop: to convert a `User` entity, we need to convert a `DragonTreasure` entity... which means we need to convert its `owner`... which is that same `User` entity.

Setting Mapping Depth

The fix lives in your mapper, when calling the `map()` function. Pass a *third* argument, which is a "context"... kind of an array of options. You can pass whatever you want, but Micro Mapper itself only has 1 option that it cares about. Set `MicroMapperInterface::MAX_DEPTH` to 1.

src/Mapper/UserEntityToApiMapper.php

```
↕ // ... lines 1 - 4
5  use App\ApiResource\DragonTreasureApi;
↕ // ... line 6
7  use App\Entity\DragonTreasure;
↕ // ... lines 8 - 10
11 use Symfonycasts\MicroMapper\MicroMapperInterface;
↕ // ... lines 12 - 13
14 class UserEntityToApiMapper implements MapperInterface
15 {
16     public function __construct(
17         private MicroMapperInterface $microMapper,
18     )
19     {
20     }
↕ // ... lines 21 - 41
42         $dto->dragonTreasures = array_map(function(DragonTreasure
   $dragonTreasure) {
43             return $this->microMapper->map($dragonTreasure,
   DragonTreasureApi::class, [
44                 MicroMapperInterface::MAX_DEPTH => 1,
45             ]);
46         }, $entity->getPublishedDragonTreasures()->getValues());
↕ // ... lines 47 - 49
50     }
51 }
```

Let's see what that does. When we refresh... look at the dump, which comes from the state provider. It maps the `User` entities to `UserApi` objects... and we see 5. We can *also* see that the `dragonTreasures` property *is* populated with `DragonTreasureApi` objects. So it *did* do the mapping from `DragonTreasure` to `DragonTreasureApi`. But when it went to map the `owner` of that `DragonTreasure` to a `UserApi`, it's there... but it's *empty*. It's a *shallow* mapping.

When we pass `MAX_DEPTH => 1`, we're saying:

> *"Yo! I want you to fully map this `DragonTreasure` entity to `DragonTreasureApi`. That is depth 1. But if the micro mapper is called again to map any deeper, skip that."*

Well, not exactly *skip*. When the mapper is called the 2nd time to map the `User` entity to `UserApi`, it calls the `load()` method on that mapper... but *not* `populate()`. So we end up with a `UserApi` object with an `id`... but nothing else. That fixes our circular loop. And, we don't really care that the `owner` property is an empty object... because our JSON never renders that deeply!

Watch. Remove the `dd()` so we can see the results. And... perfect! The result is *exactly* what we expect! For `DragonTreasures`, we're only showing the IRI.

So, as a rule, when calling micro mapper from inside a mapper class, you'll probably want to set `MAX_DEPTH` to `1`. Heck, we *could* set `MAX_DEPTH` to `0`! Though the only reason to do that would be a *slight* performance improvement.

This time, when we map `$dragonTreasure` to `DragonTreasureApi`, try `MAX_DEPTH => 0`.

src/Mapper/UserEntityToApiMapper.php

```
↕ // ... lines 1 - 13
14 class UserEntityToApiMapper implements MapperInterface
15 {
↕ // ... lines 16 - 32
33     public function populate(object $from, object $to, array $context): object
34     {
↕ // ... lines 35 - 41
42         $dto->dragonTreasures = array_map(function(DragonTreasure $dragonTreasure) {
43             return $this->microMapper->map($dragonTreasure, DragonTreasureApi::class, [
44                 MicroMapperInterface::MAX_DEPTH => 0,
45             ]);
46         }, $entity->getPublishedDragonTreasures()->getValues());
↕ // ... lines 47 - 49
50     }
51 }
```

This will cause the depth to be hit *immediately*. When it goes to map the `DragonTreasure` entity to `DragonTreasureApi`, it will use the mapper, but *only* call the `load()` method. The `populate()` method will *never* be called. Put the `dd()` back. What we end up with is a *shallow* object for `DragonTreasureApi`.

This might seem weird, but it's *technically* okay... because this `dragonTreasures` array is going to be rendered as IRI strings... and the only thing API Platform needs to build that IRI is... the `id`! Check it out! Remove the dump and reload the page. It looks *exactly* the same. We just saved ourselves a tiny bit of work.

So, to be on the safe side - in case you embed the object - use `MAX_DEPTH => 1`. But if you know that you're using IRIs, you *can* set `MAX_DEPTH` to `0`.

Over here, let's do the *same* thing: `MicroMapperInterface::MAX_DEPTH` set to 0 because we know that we're only showing the IRI here as well.

src/Mapper/DragonTreasureEntityToApiMapper.php

```
↕ // ... lines 1 - 13
14 class DragonTreasureEntityToApiMapper implements MapperInterface
15 {
↕ // ... lines 16 - 33
34     public function populate(object $from, object $to, array $context): object
35     {
↕ // ... lines 36 - 41
42         $dto->owner = $this->microMapper->map($entity->getOwner(), UserApi::class, [
43             MicroMapperInterface::MAX_DEPTH => 0,
44         ]);
↕ // ... lines 45 - 52
53     }
54 }
```

Forcing a JSON Array

One *other* thing you may have noticed is that `dragonTreasures` suddenly looks like an *object* - with its squiggly braces instead of square brackets. Well, in PHP it *is* an array - `array_map` returns an array with the `0` key set to something and the `2` key to set to something. But because of the missing `1` key, when it's serialized to JSON it looks like an *associative* array, or an "object" in JSON.

If we change the `toArray()` to `getValues()` and refresh the page... perfect! We're back to a regular array of items.

Next: We can *read* from our new `DragonTreasureApi` resource, but we can't *write* to it yet. Let's create a `DragonTreasureApiToEntityMapper` and re-add things like security and

validation.

Chapter 29: Making DragonTreasureApi Writable

Let's get our *write* endpoints working for `DragonTreasureApi`! If you look down here, we have a test called `testPostToCreateTreasure()`. That sounds like a good one! Over in your terminal, run it:

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

And... it goes kaboom! It ran a *few* tests... and they all say the same thing:

> *"No mapper found for `DragonTreasureApi` -> `DragonTreasure`"*

Ok, when we POST, it *deserializes* the JSON into a new `DragonTreasureApi` object and then calls our processor. Our processor takes that API object and tries to use MicroMapper to map it to the `DragonTreasure` entity. Since we're *missing* the mapper from `DragonTreasureApi` to `DragonTreasure`, kablooie!

Creating the Mapper

We know the drill! In `src/Mapper/`, create a new `DragonTreasureApiToEntityMapper`. Inside, implement `MapperInterface`, use `#[AsMapper()]` to say that we are mapping `from: DragonTreasureApi::class`, `to: DragonTreasure::class`... and add the two methods.

src/Mapper/DragonTreasureApiToEntityMapper.php

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

use App\ApiResource\DragonTreasureApi;
use App\Entity\DragonTreasure;
use App\Repository\DragonTreasureRepository;
use Symfonycasts\MicroMapper\AsMapper;
use Symfonycasts\MicroMapper\MapperInterface;

#[AsMapper(from: DragonTreasureApi::class, to: DragonTreasure::class)]
class DragonTreasureApiToEntityMapper implements MapperInterface
{
// ... lines 14 - 45
}
```

This will be very similar to our `UserApiToEntityMapper`. In `load()`, if we have an ID, we want to *query* for that object. Add a constructor, with `private DragonTreasureRepository $repository`. Down here, include the now-familiar `$dto = $from`, and `assert` that `$dto` is an `instanceof DragonTreasureApi`. To make life even easier, steal some code from our other mapper. Copy this... and plop it over here. But Hit "Cancel" because we don't need that `use` statement... and rename this to just `$entity`. So if the `$dto` has an `id`, it means we're editing it and we want to find the existing one. *Else*, we're going to create a `new DragonTreasure()`. And while it *shouldn't* happen, we have an `Exception` in case the treasure wasn't found.

One interesting thing about the `DragonTreasure` entity is that it has a constructor argument: the *name*. And we *don't* have a `setName()` method: the only way to set it is through the constructor. So, to transfer the `name` from the `$dto` *onto* the entity, pass it to the constructor.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
// ... lines 1 - 6
use App\Repository\DragonTreasureRepository;
// ... lines 8 - 11
class DragonTreasureApiToEntityMapper implements MapperInterface
{
    public function __construct(
        private DragonTreasureRepository $repository,
    )
    {

    }

    public function load(object $from, string $toClass, array $context): object
    {
        $dto = $from;
        assert($dto instanceof DragonTreasureApi);

        $entity = $dto->id ? $this->repository->find($dto->id) : new DragonTreasure($dto->name);
        if (!$entity) {
            throw new \Exception('DragonTreasure not found');
        }

        return $entity;
    }
// ... lines 33 - 45
}
```

Two quick notes about this. Yes, this means that you can't *change* the name of an existing treasure via the API. And that's expected: if we've written our `DragonTreasure` without a `setName()` method, then we're intending for the name to be set once and never changed. Second, this is the *one* case where we *do* populate a *bit* of data inside `load()`. We normally save that work for `populate()`, but it can't be avoided here, and that's ok.

Head down to `populate()` and start with the same code from `load()`. Also add `$entity = $to`... and one more `assert()` that `$entity instanceof DragonTreasure`. Just say `TODO` for a moment.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 11
12 class DragonTreasureApiToEntityMapper implements MapperInterface
13 {
↕ // ... lines 14 - 33
34     public function populate(object $from, object $to, array $context): object
35     {
36         $dto = $from;
37         $entity = $to;
38         assert($dto instanceof DragonTreasureApi);
39         assert($entity instanceof DragonTreasure);
↕ // ... lines 40 - 43
44         return $entity;
45     }
46 }
```

I want to make sure our mapper is at least being called. Earlier, when we ran the test, it executed *three* tests that match the name. So let's make the method a bit more unique. This is called `testPostToCreateTreasure()` and it uses the normal login mechanism, so add `WithLogin` at the end. When we run the test with the new name:

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

A 500 error! Let's see what's going on. Okay, good! We got further! It's *now* exploding when it hits the *database*. So it *is* trying to save, and it's complaining because `owner_id` is null.

Adding Validation Constraints

Reminder time: the `owner` field is *supposed* to be optional. If we *don't* pass an owner, it should automatically be set to the authenticated user. We *had* code for that before, and we'll re-add it in a moment.

But this failure is actually coming from *earlier*: from line 71, right here. This test starts by checking our validation. It submits *no* JSON, and makes sure that our validation constraints save the day. We don't *have* any validation constraints, so instead of *failing* validation, it tries to save. Boo.

Let's re-add the constraints... this time to our API class. For `$name`, `#[NotBlank]`, `$description`, `#[NotBlank]`, `$value` will be `#[GreaterThanOrEqual(0)]` and `$coolFactor` will be `#[GreaterThanOrEqual(0)]` and *also* `#[LessThanOrEqual(10)]`.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 10
11 use Symfony\Component\Validator\Constraints\GreaterThanOrEqual;
12 use Symfony\Component\Validator\Constraints\LessThanOrEqual;
13 use Symfony\Component\Validator\Constraints\NotBlank;
↕ // ... lines 14 - 21
22 class DragonTreasureApi
23 {
↕ // ... lines 24 - 26
27     #[NotBlank]
28     public ?string $name = null;
29
30     #[NotBlank]
31     public ?string $description = null;
32
33     #[GreaterThanOrEqual(0)]
34     public int $value = 0;
35
36     #[GreaterThanOrEqual(0)]
37     #[LessThanOrEqual(10)]
38     public int $coolFactor = 0;
↕ // ... lines 39 - 46
47 }
```

Try the test again.

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

We're probably going to hit that same error, and... yep - 500 error. But look! Now it's coming from line 78! That means we *are* getting the validation error status code here. Then, below, when we POST valid data, it attempts to save it to the database, but *can't* because, like we saw a second ago, the `owner_id` is *still* null.

Automatically Setting the Owner

This is one of the great things about these mapper objects. In `DragonTreasureApiToEntityMapper`, *normally*, we're going to do things like `$entity->setValue($dto->value)`: just transferring data from one to the other. But we can *also* do custom things - like setting weird fields that require calculations or... setting the owner to the currently-authenticated user.

Check it out: `if ($dto->owner)`, then we're going to set that onto the entity. Well, we won't do it *yet*, just `dd()` for now. This is the case where we *do* include the `owner` field in the JSON... and we'll talk more about that soon.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 12
13 class DragonTreasureApiToEntityMapper implements MapperInterface
14 {
↕ // ... lines 15 - 35
36     public function populate(object $from, object $to, array $context): object
37     {
↕ // ... lines 38 - 42
43         if ($dto->owner) {
44             dd($dto->owner);
↕ // ... lines 45 - 46
47         }
↕ // ... lines 48 - 52
53     }
54 }
```

For the `else`, this is when the user does *not* send an `owner` field. To set it to the currently authenticated user, on top, inject the `Security` service onto a new property. Then back below, set `owner` to `$this->security->getUser()`.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 12
13 class DragonTreasureApiToEntityMapper implements MapperInterface
14 {
15     public function __construct(
↕ // ... line 16
17         private Security $security,
18     )
19     {
20
21     }
↕ // ... lines 22 - 35
36     public function populate(object $from, object $to, array $context):
   object
37     {
↕ // ... lines 38 - 42
43         if ($dto->owner) {
44             dd($dto->owner);
45         } else {
46             $entity->setOwner($this->security->getUser());
47         }
↕ // ... lines 48 - 52
53     }
54 }
```

Beautiful! We *are* still missing the other field setting... so if we try to run the test... it will *still* hit a 500. *But*, if you check out the error, it's failing because `description` is null. The `owner` *is* being set.

So let's fill in the other fields: `$entity->setDescription($dto->description)`, `$entity->setCoolFactor($dto->coolFactor)`, and `$entity->setValue($dto->value)`.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 12
13 class DragonTreasureApiToEntityMapper implements MapperInterface
14 {
15     public function __construct(
↕ // ... line 16
17         private Security $security,
18     )
19     {
20
21     }
↕ // ... lines 22 - 35
36     public function populate(object $from, object $to, array $context): object
37     {
↕ // ... lines 38 - 42
43         if ($dto->owner) {
44             dd($dto->owner);
45         } else {
46             $entity->setOwner($this->security->getUser());
47         }
48
49         $entity->setDescription($dto->description);
50         $entity->setCoolFactor($dto->coolFactor);
51         $entity->setValue($dto->value);
52
53         // TODO: set published
54
↕ // ... lines 55 - 58
```

Boring but clear work. Also include a `TODO` down for `published`. We'll talk more about that shortly.

Ok, run the test now:

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

And... it *passes*. Woo! Try *all* the tests from `DragonTreasure`:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

And... ooo. We have several failures, related to missing headers, security, validation, etc. Let's make this green next.

Chapter 30: DTO & Security

Our `DragonTreasureApi` is looking great! Back when this resource was an *entity*, we added *quite* a few cool customizations *and* included tests for those. Past "us" rocks.

The plan *now* is to put those thing back piece-by-piece and see how we can simplify the implementation inside our new DTO-powered setup.

Be crazy and run *all* the dragon treasure tests:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

Quite a few fail... and *one* of them says:

> *“Current response status code is 422, but 403 expected.”*

This `testPostToCreateTreasureDeniedWithoutScope` is related to security, and that makes sense. `DragonTreasureApi` is *entirely* missing security!

Adding Security Back

Start like we did with `UserApi`: by specifying the operations we want. Start with `new Get()`, `new GetCollection()`, and `new Post()`. In the original system, `Post()` had a `security` option set to `'is_granted("ROLE_TREASURE_CREATE")`.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 12
13 class DragonTreasureApiToEntityMapper implements MapperInterface
14 {
15     public function __construct(
↕ // ... line 16
17         private Security $security,
18     )
19     {
20
21     }
↕ // ... lines 22 - 35
36     public function populate(object $from, object $to, array $context):
   object
37     {
↕ // ... lines 38 - 42
43         if ($dto->owner) {
44             dd($dto->owner);
45         } else {
46             $entity->setOwner($this->security->getUser());
47         }
48
49         $entity->setDescription($dto->description);
50         $entity->setCoolFactor($dto->coolFactor);
51         $entity->setValue($dto->value);
52
53         // TODO: set published
54
↕ // ... lines 55 - 58
```

This is directly related to that test failure, which checks to make sure that our API token has that role. Well... if I spell "create" correctly, at least.

We also had a `Patch()` operation and that *also* had a `security` option. This leveraged a custom voter to check if the current user can `EDIT` this treasure. More on that in a minute.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 8
9  use ApiPlatform\Metadata\Get;
10 use ApiPlatform\Metadata\GetCollection;
11 use ApiPlatform\Metadata\Patch;
12 use ApiPlatform\Metadata\Post;
↕ // ... lines 13 - 19
20 #[ApiResource(
↕ // ... line 21
22     operations: [
23         new Get(),
24         new GetCollection(),
25         new Post(
26             security: 'is_granted("ROLE_TREASURE_CREATE")',
27         ),
28         new Patch(
29             security: 'is_granted("EDIT", object)',
30         ),
↕ // ... lines 31 - 33
34     ],
↕ // ... lines 35 - 38
39 )]
40 class DragonTreasureApi
41 {
↕ // ... lines 42 - 64
65 }
```

And *finally*, we had `new Delete()`, which we decided only admins could do. Enforce that with `is_granted("ROLE_ADMIN")`.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 7
8  use ApiPlatform\Metadata\Delete;
9  use ApiPlatform\Metadata\Get;
10 use ApiPlatform\Metadata\GetCollection;
11 use ApiPlatform\Metadata\Patch;
12 use ApiPlatform\Metadata\Post;
↕ // ... lines 13 - 19
20 #[ApiResource(
↕ // ... line 21
22     operations: [
23         new Get(),
24         new GetCollection(),
25         new Post(
26             security: 'is_granted("ROLE_TREASURE_CREATE")',
27         ),
28         new Patch(
29             security: 'is_granted("EDIT", object)',
30         ),
31         new Delete(
32             security: 'is_granted("ROLE_ADMIN")',
33         )
34     ],
↕ // ... lines 35 - 38
39 )]
40 class DragonTreasureApi
41 {
↕ // ... lines 42 - 64
65 }
```

Okay, we had *six* failures earlier and now:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

We're down to five. Progress! Let's zoom in on `testPatchToUpdateTreasure` and run *just* that:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

Back over here... check out what it's doing. Ok, it creates a `User`, a treasure, logs in as the owner, tries to change the *value* of that treasure, makes sure we get a 200 status code, and finally, checks that we see the updated value. Right now, we're getting a 403 instead of 200.

Updating the Security Voter for the DTO

A 403 status is a *security* failure. For some reason, we're not allowed to make a `Patch()` request to this treasure... even though we're the owner! Rude!

Ok: `Patch()` is using `is_granted("EDIT", object)`. This `"EDIT", object` thing is handled by a custom voter called `DragonTreasureVoter` that we created in a previous tutorial. So, either this voter is not being called or its saying that we shouldn't have access.

To see what's going on under the hood, `dump($attribute, $subject)`. This `supports()` method is called *any* time a security decision is made across the *entire* system, so it *should* get hit.

When we run the test again:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

There's the dump! It dumps `EDIT`, which comes from the `Patch()` operation. But here's the kicker: the *object* is now a `DragonTreasureApi`, which makes sense! *But* our `DragonTreasureVoter` was written to work with the *entity*, *not* `DragonTreasureApi`.

No problem! Let's update this voter to work with the DTO. For clarity, *rename* this to `DragonTreasureApiVoter`. *Then*, we'll support if `DragonTreasureApi` is the `$subject`. And down here, this `$subject` should also be `DragonTreasureApi`. `dd($subject)`... and below, let's fix the code. This says that if the user doesn't have this role (actually a *scope*, which relates to the token scopes), return `false`.

src/Security/Voter/DragonTreasureApiVoter.php

```
↕ // ... lines 1 - 4
5 use App\ApiResource\DragonTreasureApi;
↕ // ... lines 6 - 10
11 class DragonTreasureApiVoter extends Voter
12 {
↕ // ... lines 13 - 18
19     protected function supports(string $attribute, mixed $subject): bool
20     {
21         return in_array($attribute, [self::EDIT])
22             && $subject instanceof DragonTreasureApi;
23     }
24
25     protected function voteOnAttribute(string $attribute, mixed $subject,
   TokenInterface $token): bool
26     {
↕ // ... lines 27 - 36
37         assert($subject instanceof DragonTreasureApi);
38         dd($subject);
↕ // ... lines 39 - 53
54         return false;
55     }
56 }
```

The most important part is this: if the `$subject` - which is a `DragonTreasureApi` - has an owner that equals `$user` - the currently authenticated user - then return true: access granted!

Comment out this `dd()` real quick. What we need now is `$subject->owner`.

Well, that's not *quite* right... and if we put that `dd()` back, we can see why. Run the test:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

This dump - the `$subject` - is, of course, a `DragonTreasureApi`. But remember, its `owner` property *isn't* a `User` entity: it's a `UserApi` object. So we can't just compare the `UserApi` object to the `$user` entity object.

We *also* need to be careful because of our mapper. Thanks to the depth, the `UserApi` *isn't* populated: it's a *shallow* object. That's okay - we can compare the id of the objects - just keep this in mind.

So, the tl;dr is: compare the `id` property to `$user->getId()`. Oh, and it didn't autocomplete `getId()`... but we can help our editor by making this `instanceof` check specifically that this is a `User` entity, which it always will be in our app.

Now use `getId()`... and I'll code defensively by adding a `?`... in case this `DragonTreasureApi` doesn't have an owner: like for a treasure we're creating right now.

src/Security/Voter/DragonTreasureApiVoter.php

```
↕ // ... lines 1 - 4
5  use App\ApiResource\DragonTreasureApi;
6  use App\Entity\User;
↕ // ... lines 7 - 10
11
12 class DragonTreasureApiVoter extends Voter
↕ // ... lines 13 - 19
20     protected function supports(string $attribute, mixed $subject): bool
21     {
22         return in_array($attribute, [self::EDIT])
23             && $subject instanceof DragonTreasureApi;
24     }
25
26     protected function voteOnAttribute(string $attribute, mixed $subject,
   TokenInterface $token): bool
27     {
↕ // ... lines 28 - 29
30         if (!$user instanceof User) {
31             return false;
32         }
↕ // ... lines 33 - 37
38         assert($subject instanceof DragonTreasureApi);
↕ // ... lines 39 - 40
41         switch ($attribute) {
42             case self::EDIT:
↕ // ... lines 43 - 46
47                 if ($subject->owner?->id === $user->getId()) {
48                     return true;
49                 }
↕ // ... lines 50 - 51
52         }
53
54         return false;
55     }
56 }
```

Phew! Head over and try it now!

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

Adding the application/merge-patch+json Header

Progress! The current response status code is now *415*. This is thanks to a small detail we talked about a few times:

> *"The content-type `application/json` is not supported. Supported MIME types are `application/merge-patch+json`."*

When we make a PATCH request, we need to have a `headers` key with `Content-Type` set to `application/merge-patch+json`. The reason we didn't need that before, as I mentioned in a previous tutorial... is due to some funny business with formats which made that, accidentally, unnecessary for this resource. *But* now we *do* need it.

Let's quickly add that to all of our `patch()` requests. There's a *bunch* of them. Zoom!

tests/Functional/DragonTreasureResourceTest.php

```
↕ // ... lines 1 - 14
15 class DragonTreasureResourceTest extends ApiTestCase
16 {
↕ // ... lines 17 - 116
117     public function testPatchToUpdateTreasure()
118     {
↕ // ... lines 119 - 121
122         $this->browser()
↕ // ... line 123
124             ->patch('/api/treasures/'.$treasure->getId(), [
↕ // ... lines 125 - 127
128                 'headers' => ['Content-Type' => 'application/merge-
    patch+json']
129             ])
↕ // ... lines 130 - 134
135         $this->browser()
↕ // ... line 136
137             ->patch('/api/treasures/'.$treasure->getId(), [
↕ // ... lines 138 - 142
143                 'headers' => ['Content-Type' => 'application/merge-
    patch+json']
144             ])
↕ // ... line 145
146         ;
↕ // ... line 147
148         $this->browser()
↕ // ... line 149
150             ->patch('/api/treasures/'.$treasure->getId(), [
↕ // ... lines 151 - 154
155                 'headers' => ['Content-Type' => 'application/merge-
    patch+json']
156             ])
↕ // ... line 157
158         ;
159     }
↕ // ... line 160
161     public function testPatchUnpublishedWorks()
162     {
↕ // ... lines 163 - 168
169         $this->browser()
↕ // ... line 170
171             ->patch('/api/treasures/'.$treasure->getId(), [
↕ // ... lines 172 - 174
175                 'headers' => ['Content-Type' => 'application/merge-
    patch+json']
176             ])
```

```
↕ // ... lines 177 - 178
179         ;
180     }
↕ // ... lines 181 - 182
183     public function testAdminCanPatchToEditTreasure(): void
184     {
↕ // ... lines 185 - 189
190         $this->browser()
↕ // ... line 191
192             ->patch('/api/treasures/'.$treasure->getId(), [
↕ // ... lines 193 - 195
196                 'headers' => ['Content-Type' => 'application/merge-
    patch+json']
197             ])
↕ // ... lines 198 - 200
201         ;
202     }
203
204     public function testOwnerCanSeeIsPublishedAndIsMineFields(): void
205     {
↕ // ... lines 206 - 211
212         $this->browser()
↕ // ... line 213
214             ->patch('/api/treasures/'.$treasure->getId(), [
↕ // ... lines 215 - 217
218                 'headers' => ['Content-Type' => 'application/merge-
    patch+json']
219             ])
↕ // ... lines 220 - 223
224         ;
225     }
↕ // ... line 226
227     public function testPublishTreasure(): void
228     {
↕ // ... lines 229 - 234
235         $this->browser()
↕ // ... line 236
237             ->patch('/api/treasures/'.$treasure->getId(), [
↕ // ... lines 238 - 240
241                 'headers' => ['Content-Type' => 'application/merge-
    patch+json']
242             ])
↕ // ... lines 243 - 244
245         ;
↕ // ... lines 246 - 247
248     }
249 }
```

Let's see if we have any luck!

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

And... *ooh*... it *dies*. It hit our dump! That's coming from `DragonTreasureApiToEntityMapper`: when the `owner` is sent in the JSON. Comment this out for a moment so we can see the full picture. Run the test again:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

> *"Current response status code is 200, but 422 expected."*

Coming from down on line 157. So, looking at our test, *most* of it passes. Line 157 is *way* down here. This means that we *are* able to send a `patch()` request and have that update!

And the full *flow* here is fascinating! When we make a `patch()` request to a treasure, API Platform starts by using our data provider to find the `DragonTreasure` entity. Then we *map* that to a `DragonTreasureApi` object. Next, the new `value` is deserialized onto that `DragonTreasureApi`. Finally, in our processor, we map the *updated* `DragonTreasureApi` *back* to a `DragonTreasure` entity, and *that* is ultimately what saves. The `DragonTreasureApi` is then *serialized* and returned as JSON.

So this *is* working... and I *love* how all the pieces come together.

Updating the Custom Validator

Where we're *failing* is all the way down here. This checks to see if we're allowed to change the `owner` to someone else. We log in as `$user` and edit our *own* treasure... but try to change the treasure to *another* owner! This is like a dragon Santa Claus that sneaks into other dragon's caves for a late-night delivery of treasure. That's super nice... but not something we want to allow.

Previously, we had a custom validator that prevented this. So let's re-add that!

Open `DragonTreasureApi` and find the `$owner` property. Add `#[IsValidOwner]`: a validator we created in an earlier tutorial.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 15
16 use App\Validator\IsValidOwner;
↕ // ... lines 17 - 40
41 class DragonTreasureApi
42 {
↕ // ... lines 43 - 58
59     #[IsValidOwner]
60     public ?UserApi $owner = null;
↕ // ... lines 61 - 66
67 }
```

You'll find it in `src/Validator/`. Previously, this validator expected its constraint to be used above a property that held a `User` *entity*. *We're* putting it on a property that holds a `UserApi`. So like with the voter, we need to update it for the new reality.

Right here, `assert()` that `$value` is an `instanceof UserApi`.

src/Validator/IsValidOwnerValidator.php

```
↕ // ... lines 1 - 4
5 use App\ApiResource\UserApi;
↕ // ... lines 6 - 10
11 class IsValidOwnerValidator extends ConstraintValidator
12 {
↕ // ... lines 13 - 16
17     public function validate($value, Constraint $constraint): void
18     {
↕ // ... lines 19 - 25
26         assert($value instanceof UserApi);
↕ // ... lines 27 - 41
42     }
43 }
```

Down here, we need to check if the value (meaning the `UserApi` that's on this property) is *not* equal to the currently authenticated user. Once again, we'll use the `id`s to compare this. And... *also* once again, I'll use `assert()` to help my editor. Now... it's happy about `getId()`... but not about my missing semicolon!

src/Validator/IsValidOwnerValidator.php

```
↕ // ... lines 1 - 4
5  use App\ApiResource\UserApi;
↕ // ... lines 6 - 10
11 class IsValidOwnerValidator extends ConstraintValidator
12 {
↕ // ... lines 13 - 16
17     public function validate($value, Constraint $constraint): void
18     {
↕ // ... lines 19 - 25
26         assert($value instanceof UserApi);
↕ // ... lines 27 - 37
38         if ($value->id !== $user->getId()) {
↕ // ... lines 39 - 40
41         }
42     }
43 }
```

Moment of truth! Run that test:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

It passes! Try *everything*:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

And... ah! We're down to just *three* failures. And they're all related to the same thing: the `isPublished` property. Our `DragonTreasureApi` doesn't even have an `isPublished` property yet. We saved *that* for last because it's a *little* different and interesting. Let's tackle it *next*.

Chapter 31: Field Security with Patch

In a heroic twist of bravery, we've decided to run *all* the dragon treasure tests:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

And... we have *three* failures, including one from `testAdminCanPatchToEditTreasure` on line 200... which says `->assertJsonMatched('isPublished', true)`. That's failing because... we don't have an `isPublished` field in our `DragonTreasureApi` *at all*!

Adding the isPublished Field

That's because this is an *interesting* field. Previously, this field was readable *only* by admin users or the owner. Let's re-add this field and keep that behavior. Say `public bool $isPublished = false`.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 40
41 class DragonTreasureApi
42 {
↕ // ... lines 43 - 58
59     public bool $isPublished = false;
↕ // ... lines 60 - 68
69 }
```

Then... head into the mapper to populate this. Down here, get rid of the `TODO` and say `$entity->setIsPublished($dto->isPublished)`.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 12
13 class DragonTreasureApiToEntityMapper implements MapperInterface
14 {
↕ // ... lines 15 - 35
36     public function populate(object $from, object $to, array $context): object
37     {
↕ // ... lines 38 - 51
52         $entity->setIsPublished($dto->isPublished);
↕ // ... lines 53 - 54
55     }
56 }
```

So if we *change* `isPublished` in the API call, the new value will sync back to the entity.

On the other side... it doesn't matter where... say `$dto->isPublished = $entity->getIsPublished()`.

src/Mapper/DragonTreasureEntityToApiMapper.php

```
↕ // ... lines 1 - 13
14 class DragonTreasureEntityToApiMapper implements MapperInterface
15 {
↕ // ... lines 16 - 33
34     public function populate(object $from, object $to, array $context): object
35     {
↕ // ... lines 36 - 41
42         $dto->isPublished = $entity->getIsPublished();
↕ // ... lines 43 - 53
54     }
55 }
```

Cool! We don't have any security yet... so when we run the tests:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

A few pass, but the *original* one still fails - `testGetCollectionOfTreasures` - because it's not expecting the `isPublished` to be there.

Conditionally Showing isPublished via Security

Check it out: this is the *first* test, and at the bottom, we've asserted that these are the *exact* properties we should have if we fetch treasures as an anonymous user. So since we're *not* the owner or an admin, we should not see `isPublished`

How can do we that? Earlier, we worked on `DragonTreasureApiVoter`. When we call this with the `EDIT` attribute, it checks to see if we're an admin, and if we *are*, it grants access. It *also* checks to see if we're the *owner*. This is *exactly* the logic we want to use to determine if the `isPublished` field should be serialized.

So... let's use it! Above this property, say `#[ApiProperty(security: 'is_granted("EDIT", object)')]`.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 40
41 class DragonTreasureApi
42 {
↕ // ... lines 43 - 58
59     #[ApiProperty(security: 'is_granted("EDIT", object)')]
60     public bool $isPublished = false;
↕ // ... lines 61 - 69
70 }
```

If you want, you can change this attribute to something else - like `OWNER` - if that's more clear. `EDIT` sounds a little funny here... since we're just deciding if we should *include* this field in the response... not "edit" it... but it's up to you.

More importantly, let's see if this does the trick. Run the tests:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

It *fixed* our first test! The `isPublished` field is no longer being shown. *But*, curiously, we made *another* test *fail*. Whac-A-Mole! Now it's `testPublishTreasure` - failing on line 244.

Let's pop over and search for that. Okay, as the name suggests, we're testing to see if we can publish this treasure. We create a treasure that is `'isPublished' => false`, log in as its owner, then send a `patch()` request to set `isPublished` to `true`. Finally, we assert that the JSON in the response *has* `isPublished` true. And *that's* what's failing.

The ApiProperty Security Option on Patch Operations

Why? It took me a bit of debugging to unravel this mystery. The problem is that, when the JSON is deserialized, `isPublished` is *not* writable.

The `security` expression is called both when serializing and *deserializing*: when taking the JSON from the request and updating the object. For some reason, during deserialization, our `security` expression is returning *false*!

The reason is... *arguably* a bug: I have an issue open on API Platform. When you make a `patch()` request, our data provider first loads the object from the database. Despite this, when the expression is called during deserialization, `object` is *always* null. And because our voter only supports if `object` is a `DragonTreasureApi`, this returns `false`. Ultimately, *no* voters support this, and when that happens, access is *denied*. The end result is that `isPublished` is *not* writable.

The workaround is a bit weird, but stay with me here. We're basically going to allow access to this field if `object === null` or `is_granted("EDIT", object)`.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 40
41 class DragonTreasureApi
42 {
↕ // ... lines 43 - 58
59     // Object is null ONLY during deserialization: so this allows isPublished
60     // to be writable in ALL cases (which is ok because the operations are secured).
61     // During serialization, object will always be a DragonTreasureApi, so our
62     // voter is called.
63     #[ApiProperty(security: 'object === null or is_granted("EDIT", object)')]
64     public bool $isPublished = false;
↕ // ... lines 65 - 73
74 }
```

Think about this. If we're *reading* a `DragonTreasure`, then `object` is *never* `null`. We will *always* have an object, so the voter will *always* be called. This `object === null` will only happen during *deserialization*: when we're checking to see if we can *write* this field. This *effectively* makes the field *always writable*. That *seems* like a problem, but it's not, because we already have `security` up here on `Post()` and `Patch()`. For `Patch`, only the *owner* can

edit this object. So once you've passed the `Patch` security, we already know that you can edit this object. So, down here, it's okay to let us *always* edit this field.

If this looks too weird to you, another strategy is to leave API security off of the field *entirely*. Then, we would use the *mapper* to handle conditionally setting the `isPublished` field. We could put some security logic right here that basically says:

> *"Only set the `isPublished` field on the DTO if you're the owner. Otherwise, leave `isPublished` null as the default."*

It's good to remember that we *do* have full control of the data via our mappers.

Okay, let's go back and re-add our security expression. Oh! And go back to the mapper as well: I just realized that we also want to keep that `isPublished` code... just not in the `if` statement.

All right, *now* head over and rerun all the tests.

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

And... oooh! *So close*! We're down to just *one* failure in `testPublishTreasure`. This tests that, when a treasure is published, we send a notification. Let's see how we can tackle that in our new system next!

Chapter 32: Triggering a "Publish"

We're down to just one test failure: it's in `testPublishTreasure`. Let's check it out. Ok, this tests to make sure that a notification is created in the database when the status of a treasure changes from `'isPublished' => false` to `'isPublished' => true`. Previously, we implemented this via a custom state processor.

But now, we *could* put this into our mapper class! In `DragonTreasureApiToEntityMapper`, we could check to see if the entity *was* `'isPublished' => false` and is now *changing* to `'isPublished' => true`. If it *is*, create a notification right there. If this sounds good to you, go for it!

However, for me, putting the logic here doesn't *quite* feel right... just because it's a "data mapper", so it smells a bit strange to do something *beyond* just mapping the data.

Creating the State Processor

So, let's go back to our original solution: creating a *state processor*. Over at you terminal, run:

```
php bin/console make:state-processor
```

Call it `DragonTreasureStateProcessor`. Our goal should feel familiar: we'll add some custom logic here, but call the *normal* state processor to let it do the heavy lifting.

src/State/DragonTreasureStateProcessor.php

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

use ApiPlatform\Metadata\Operation;
use ApiPlatform\State\ProcessorInterface;

class DragonTreasureStateProcessor implements ProcessorInterface
{
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = []): void
    {
        // Handle the state
    }
}
```

To do that, add a `__construct()` method with `private EntityClassDtoStateProcessor $innerProcessor`. Down here, use that with `return $this->innerProcessor->process()` passing the arguments it needs: `$data`, `$operation`, `$uriVariables`, and `$context`. Ah, and you can see this is highlighted in red. This isn't really a `void` method, so remove that.

src/State/DragonTreasureStateProcessor.php

```
// ... lines 1 - 7
class DragonTreasureStateProcessor implements ProcessorInterface
{
    public function __construct(
        private EntityClassDtoStateProcessor $innerProcessor,
    )
    {
    }

    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
    {
        return $this->innerProcessor->process($data, $operation, $uriVariables, $context);
    }
}
```

Ok, let's hook up our API resource to use this! Inside `DragonTreasureApi`, change the processor to `DragonTreasureStateProcessor`.

src/ApiResource/DragonTreasureApi.php

```
↕ // ... lines 1 - 13
14 use App\State\DragonTreasureStateProcessor;
↕ // ... lines 15 - 20
21 #[ApiResource(
↕ // ... lines 22 - 37
38     processor: DragonTreasureStateProcessor::class,
↕ // ... line 39
40 )]
41 class DragonTreasureApi
42 {
↕ // ... lines 43 - 73
74 }
```

At this point, we haven't really changed anything: the system will call our new processor... but then it just calls the *old* one. And so when we run the tests:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php
```

Everything still works except for that last failure.

Detecting the isPublished Change

So let's add our notification code! Originally, we figured out if `isPublished` was changing from `false` to `true` by using the "previous data" that's inside the `$context`. Dump `$context['previous_data']` to see what that looks like.

src/State/DragonTreasureStateProcessor.php

```
↕ // ... lines 1 - 7
8 class DragonTreasureStateProcessor implements ProcessorInterface
9 {
↕ // ... lines 10 - 15
16     public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
17     {
18         dd($context['previous_data']);
↕ // ... lines 19 - 20
21     }
22 }
```

Now, run *just* this test:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

Cool! The previous data is the `DragonTreasureApi` with `isPublished: false`.. because that's the value our entity starts with in the test. Let's also dump `$data`.

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

Okay, the original one has `isPublished: false`, and the *new* one has `isPublished: true`! And *that* makes us dangerous.

Back over, we wrote the notification code in a previous tutorial... so I'll just paste it in. This is delightfully boring! We use `$previousData` and `$data` to detect the state change from `isPublished` false to true... then create a `Notification`.

src/State/DragonTreasureStateProcessor.php

```
// ... lines 1 - 12
class DragonTreasureStateProcessor implements ProcessorInterface
{
// ... lines 15 - 22
    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
    {
        assert($data instanceof DragonTreasureApi);
        $result = $this->innerProcessor->process($data, $operation, $uriVariables, $context);

        $previousData = $context['previous_data'] ?? null;
        if ($previousData instanceof DragonTreasureApi
            && $data->isPublished
            && $previousData->isPublished !== $data->isPublished
        ) {
            $entity = $this->repository->find($data->id);
            $notification = new Notification();
            $notification->setDragonTreasure($entity);
            $notification->setMessage('Treasure has been published!');
            $this->entityManager->persist($notification);
            $this->entityManager->flush();
        }

        return $result;
    }
}
```

The only thing that's *kind of* interesting is that the `Notification` entity is related to a `DragonTreasure` entity... so we query for the `$entity` using the `repository` and the `id` from the DTO class.

Let's inject the services we need: `private EntityManagerInterface $entityManager` so we can save and `private DragonTreasureRepository $repository`.

src/State/DragonTreasureStateProcessor.php

```
// ... lines 1 - 8
use App\Entity\Notification;
use App\Repository\DragonTreasureRepository;
use Doctrine\ORM\EntityManagerInterface;

class DragonTreasureStateProcessor implements ProcessorInterface
{
    public function __construct(
        private EntityClassDtoStateProcessor $innerProcessor,
        private EntityManagerInterface $entityManager,
        private DragonTreasureRepository $repository,
    )
    {
    }

    public function process(mixed $data, Operation $operation, array $uriVariables = [], array $context = [])
    {
        assert($data instanceof DragonTreasureApi);
        $result = $this->innerProcessor->process($data, $operation, $uriVariables, $context);

        $previousData = $context['previous_data'] ?? null;
        if ($previousData instanceof DragonTreasureApi
            && $data->isPublished
            && $previousData->isPublished !== $data->isPublished
        ) {
            $entity = $this->repository->find($data->id);
            $notification = new Notification();
            $notification->setDragonTreasure($entity);
            $notification->setMessage('Treasure has been published!');
            $this->entityManager->persist($notification);
            $this->entityManager->flush();
        }

        return $result;
    }
}
```

There we go! Moment of truth:

```
symfony php bin/phpunit tests/Functional/DragonTreasureResourceTest.php --filte
```

The test *passes*! Heck, at this point, *all* of our treasure tests pass! We've completely converted this complex API resource to our DTO-powered system! High five!

Next: Let's make it possible to *write* the `$owner` property on dragon treasure. This involves a trick that will help us better understand how API Platform loads relation data.

Chapter 33: Writable Relation Fields

Open up `DragonTreasureResourceTest` and check out `testPostToCreateTreasureWithLogin()`. We've talked a lot about making our resources able to return relation fields. The main trick is simply to populate those fields from inside our data mapper. Then API Platform handles transforming them into IRIs.

One thing we *haven't* talked about is being able to *write* to one of these relation fields.

Writing to the owner Property

When we use this `post()` endpoint, we don't *need* to send an `owner` field. That's because, nestled in `DragonTreasureApiToEntityMapper`, we have code that says:

> *"If an `owner` is not sent in the JSON, automatically set it to the currently authenticated user."*

But, you are *allowed* to send the `owner` property and set it to *yourself*. Let's try that. Set `owner` to `'/api/users/'.$user->getId()`.

tests/Functional/DragonTreasureResourceTest.php

```
↕ // ... lines 1 - 14
15 class DragonTreasureResourceTest extends ApiTestCase
16 {
↕ // ... lines 17 - 61
62     public function testPostToCreateTreasureWithLogin(): void
63     {
↕ // ... lines 64 - 65
66         $this->browser()
↕ // ... lines 67 - 71
72             ->post('/api/treasures', HttpOptions::json([
↕ // ... lines 73 - 76
77                 'owner' => '/api/users/'.$user->getId(),
78             ]))
↕ // ... lines 79 - 80
81         ;
82     }
↕ // ... lines 83 - 251
```

How Relation Fields are Deserialized

When we do that, it *should* hit *this* part of our code. Battle stations! Run `symfony php bin/phpunit` and execute *just* this test:

```
symfony php bin/phpunit --filter=testPostToCreateTreasureWithLogin()
```

Perfect! It hits and dumps a `UserApi` object. *This* is *cool*. Actually, dump the entire `$dto` so we can see things in more detail.


```
symfony php bin/phpunit --filter=testPostToCreateTreasureWithLogin()
```


Fantabulous. When we send this JSON data, the serializer *deserializes* all of this into a `DragonTreasureApi` object. This string goes onto the `name` property, *this* string goes onto the `description` property, and so on. Over here, we see that: string... string... 1,000... and 5. *Super simple*.

But something special happens when the field you're sending is a *relation*, meaning the property holds an object that is an `#[ApiResource]`. Specifically, this IRI string is *transformed* into a `UserApi` object! But... how and *who* does that? The *answer* is: a bit of team work between the serializer system and the *state provider*.

Until now, as far as we know, the only time that the state provider is used is when we *fetch* a resource... like if we fetch a user here or here, or if we `PATCH` or `DELETE` a user. In all of those cases, API Platform leverages the user state provider to find the one or many users.

But there's one *other* spot where a state provider is used: when someone sends JSON that contains an IRI string on a relation field.

During the deserialization process, the serializer takes this IRI string, *sees* that it's for a `UserApi` object, then it calls *its* state provider to load that. Whatever that state provider returns will ultimately be set onto the `owner` property of `DragonTreasureApi`. This magic has *always* been happening... but I just *love* understanding the mechanics behind it. Nerd alert!

Mapping the Relation Field

Anyway, in our mapper, our job is pretty straightforward. We know that `$dto->owner` is a `UserApi` object. And what we *ultimately* need is a `User` entity. So, once again, we'll use the mapping system to go from `UserApi` over to `User`. Up here, inject a `MicroMapperInterface $microMapper`.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 11
12 use Symfonycasts\MicroMapper\MicroMapperInterface;
↕ // ... lines 13 - 14
15 class DragonTreasureApiToEntityMapper implements MapperInterface
16 {
17     public function __construct(
↕ // ... lines 18 - 19
20         private MicroMapperInterface $microMapper,
21     )
22     {
23
24     }
↕ // ... lines 25 - 60
61 }
```

And below, say `$entity->setOwner()`... but use `$this->microMapper->map()` to go from `$dto->owner` to `User::class`. And remember, any time we map a relationship, we should add a `MAX_DEPTH` as well. Set `MicroMapperInterface::MAX_DEPTH` to `0`.

src/Mapper/DragonTreasureApiToEntityMapper.php

```
↕ // ... lines 1 - 11
12 use Symfonycasts\MicroMapper\MicroMapperInterface;
↕ // ... lines 13 - 14
15 class DragonTreasureApiToEntityMapper implements MapperInterface
16 {
17     public function __construct(
↕ // ... lines 18 - 19
20         private MicroMapperInterface $microMapper,
21     )
22     {
23
24     }
↕ // ... lines 25 - 38
39     public function populate(object $from, object $to, array $context):
   object
40     {
↕ // ... lines 41 - 45
46         if ($dto->owner) {
47             $entity->setOwner($this->microMapper->map($dto->owner,
   User::class, [
48                 MicroMapperInterface::MAX_DEPTH => 0,
49             ]));
50         } else {
51             $entity->setOwner($this->security->getUser());
52         }
↕ // ... lines 53 - 59
60     }
61 }
```

Using `0` is enough because that will cause our mapper to query for the `User` object... it just won't continue and populate the individual property data from `UserApi` to `User.` We would only need to do that if we were allowing `owner` to be an embedded object, like creating a new one on the fly.... *or* if we were doing something crazy like adding the `@id` to load a user... then modifying that user all at once. Crazy, probably-not-realistic things that we talked about in previous tutorials.

And even if a user *did* try this right now, API Platform wouldn't allow it because you can only *write* embedded data on a field if we've set up the serialization groups for this.

Anyway, the only thing *we're* concerned about is making sure that we're loading the correct `User` entity object. Run the test again and...

```
symfony php bin/phpunit --filter=testPostToCreateTreasureWithLogin()
```

It's *good*! We are *now* allowed to *write* the `owner` field!

Next: Let's shift our focus to making the `dragonTreasures` field on `User` writable. This is a relation field... but because it's a collection, it'll need an extra trick.

Chapter 34: Writing to a Collection Relation

We are *so close* to completely re-implementing our API using these custom classes. So excited!

Let's run *every* test to see where we stand.

```
symfony php bin/phpunit
```

And... everything passes except *one*. This trouble-maker test is `UserResourceTest::testTreasuresCannotBeStolen`. Let's go check it out!

Open `tests/Functional/UserResourceTest.php` and search for `testTreasuresCannotBeStolen()`. Here it is.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 56
57     public function testTreasuresCannotBeStolen(): void
58     {
59         $user = UserFactory::createOne();
60         $otherUser = UserFactory::createOne();
61         $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
   $otherUser]);
62 
63         $this->browser()
64             ->actingAs($user)
65             ->patch('/api/users/' . $user->getId(), [
66                 'json' => [
67                     'username' => 'changed',
68                     'dragonTreasures' => [
69                         '/api/treasures/' . $dragonTreasure->getId(),
70                     ],
71                 ],
72                 'headers' => ['Content-Type' => 'application/merge-
   patch+json']
73             ])
74             ->assertStatus(422);
75     }
↕ // ... lines 76 - 89
90 }
```

Let's read the story. We update a user and attempt to change its `dragonTreasures` property to contain a treasure owned by someone else. The test looks for a 422 status code - because we want to prevent stealing treasures - but the test fails with a 200.

But apart from the whole stealing thing, this is the first test that we've seen that *writes* to a collection relation field. And *that* is an interesting topic all on its own.

Avoid Writable Collection Fields?

First, if you can, I'd recommend *against* allowing collection relationship fields like this to be writable. I mean, you absolutely *can*... but it adds complexity. For example, like this test shows, we need to worry about how setting the `dragonTreasures` property changes the *owner* on that treasure. And there's already a *different* way to do this: make a `patch()` request to this treasure and... change the `owner`. Simple!

But, if you still want to allow your collection relation to be writable in your DTO system, *fine*, here's how to do it. I'm kidding - it's not too bad.

Testing the Collection Write

Start by duplicating this test. Rename it to `testTreasuresCanBeRemoved`. I totally typo'ed that - mine says `cannot`, which is the *opposite* of what I want to test - so make sure you get that right in your code.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 56
57     public function testTreasuresCanBeRemoved(): void
58     {
↕ // ... lines 59 - 74
75     }
↕ // ... lines 76 - 109
110 }
```

Now we can dress this up a bit. Make the first `$dragonTreasure` owned by `$user`. Then create a *second* `$dragonTreasure` *also* owned by `$user`, but we won't need a variable for it... you'll see. Finally, add a *third* `$dragonTreasure` called `$dragonTreasure3` that's owned by `$otherUser`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 56
57     public function testTreasuresCanBeRemoved(): void
58     {
59         $user = UserFactory::createOne();
60         $otherUser = UserFactory::createOne();
61         $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
   $user]);
62         DragonTreasureFactory::createOne(['owner' => $user]);
63         $dragonTreasure3 = DragonTreasureFactory::createOne(['owner' =>
   $otherUser]);
↕ // ... lines 64 - 82
83     }
↕ // ... lines 84 - 117
118 }
```

So we have *three* `dragonTreasures`, *two* owned by `$user`, and one by `$otherUser`. Down here, we patch to modify `$user`. Remove `username` - we don't care about that - then send *two* `dragonTreasures`: the *first* and the *third*: `/api/treasures/$dragonTreasure3->getId()`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 56
57     public function testTreasuresCanBeRemoved(): void
58     {
59         $user = UserFactory::createOne();
60         $otherUser = UserFactory::createOne();
61         $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
   $user]);
62         DragonTreasureFactory::createOne(['owner' => $user]);
63         $dragonTreasure3 = DragonTreasureFactory::createOne(['owner' =>
   $otherUser]);
64
65         $this->browser()
66             ->actingAs($user)
67             ->patch('/api/users/' . $user->getId(), [
68                 'json' => [
69                     'dragonTreasures' => [
70                         '/api/treasures/' . $dragonTreasure->getId(),
71                         '/api/treasures/' . $dragonTreasure3->getId(),
72                     ],
73                 ],
74                 'headers' => ['Content-Type' => 'application/merge-
   patch+json']
75             ])
↕ // ... lines 76 - 81
82         ;
83     }
↕ // ... lines 84 - 117
118 }
```

We're going to test for *two* things. First, that the second treasure is removed from this user. Think about it: `$user` *started* with these two treasures... and the fact that this *second* treasure's IRI is *not* sent means that we want it to be *removed* from `$user`.

Second, I added `$dragonTreasure3` *temporarily* to prove that treasures *can* be stolen. This is currently owned by `$otherUser`, but we pass it to `dragonTreasures`... and we're going to verify that the *owner* of `$dragonTreasure3` changes from `$otherUser` to `$user`. That's not the end behavior we want, but it'll help us get all the relation writing working. *Then* we'll worry about *preventing* that.

Down here, `->assertStatus(200)` then extend the test by saying `->get('/api/users/' . $user->getId())` and `->dump()`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 56
57     public function testTreasuresCanBeRemoved(): void
58     {
59         $user = UserFactory::createOne();
60         $otherUser = UserFactory::createOne();
61         $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
   $user]);
62         DragonTreasureFactory::createOne(['owner' => $user]);
63         $dragonTreasure3 = DragonTreasureFactory::createOne(['owner' =>
   $otherUser]);
64
65         $this->browser()
66             ->actingAs($user)
67             ->patch('/api/users/' . $user->getId(), [
68                 'json' => [
69                     'dragonTreasures' => [
70                         '/api/treasures/' . $dragonTreasure->getId(),
71                         '/api/treasures/' . $dragonTreasure3->getId(),
72                     ],
73                 ],
74                 'headers' => ['Content-Type' => 'application/merge-
   patch+json']
75             ])
76             ->assertStatus(200)
77             ->get('/api/users/' . $user->getId())
78             ->dump()
↕ // ... lines 79 - 81
82         ;
83     }
↕ // ... lines 84 - 117
118 }
```

I want to see what the user looks like *after* the update. Finally, assert that the `length` of the `dragonTreasures` field - I need quotes on that - is 2, for treasures 1 and 3. Then assert that `dragonTreasures[0]` is equal to `'/api/treasures/'.`, followed by `$dragonTreasure->getId()`. Copy that, paste, and assert that the 1 key is `$dragonTreasure3`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 56
57     public function testTreasuresCanBeRemoved(): void
58     {
59         $user = UserFactory::createOne();
60         $otherUser = UserFactory::createOne();
61         $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
   $user]);
62         DragonTreasureFactory::createOne(['owner' => $user]);
63         $dragonTreasure3 = DragonTreasureFactory::createOne(['owner' =>
   $otherUser]);
64
65         $this->browser()
66             ->actingAs($user)
67             ->patch('/api/users/' . $user->getId(), [
68                 'json' => [
69                     'dragonTreasures' => [
70                         '/api/treasures/' . $dragonTreasure->getId(),
71                         '/api/treasures/' . $dragonTreasure3->getId(),
72                     ],
73                 ],
74                 'headers' => ['Content-Type' => 'application/merge-
   patch+json']
75             ])
76             ->assertStatus(200)
77             ->get('/api/users/' . $user->getId())
78             ->dump()
79             ->assertJsonMatches('length("dragonTreasures")', 2)
80             ->assertJsonMatches('dragonTreasures[0]', '/api/treasures/' .
   $dragonTreasure->getId())
81             ->assertJsonMatches('dragonTreasures[1]', '/api/treasures/' .
   $dragonTreasure3->getId())
82         ;
83     }
↕ // ... lines 84 - 117
118 }
```

Lovely! That test took some work, but it'll be *super* useful. Let's... just run it and see what happens! Copy the method name and, over at your terminal, run:

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

And by "*cannot* be removed", I, of course, mean that it *can* be removed. That was some good 'ol copy/paste madness right there. There we go. And... it *fails*, on line 81. This means that the request was successful... but the `dragonTreasures` are still the original two: `/api/treasures/2` instead of `/api/treasures/3`. No changes were made to the treasures.

Why? Let's find out next and leverage the property accessor component to make sure the changes save correctly.

Chapter 35: Writable Collection via the PropertyAccessor

To see what's going on here, head to the mapper: `UserApiToEntityMapper`. The `patch()` request will take this data, populate it onto `UserApi`... then *we* map it *back* to the entity in this mapper.

And... the reason the test fails is pretty obvious: we're not mapping the `dragonTreasures` property from the DTO to the entity!

Let's `dump($dto)` so we can see what it looks like after deserializing the data.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 12
13 class UserApiToEntityMapper implements MapperInterface
14 {
↕ // ... lines 15 - 34
35     public function populate(object $from, object $to, array $context): object
36     {
↕ // ... lines 37 - 46
47         dump($dto);
↕ // ... lines 48 - 50
51     }
52 }
```

Run the test again:

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

And... whoa. The `dragonTreasures` in the DTO is *still* the original two. This tells me that this field is being completely ignored: it's *not* being deserialized. And I bet you know the reason. Inside `UserApi`, the `$dragonTreasures` property *isn't* `writable`! But it *is* pretty cool to see `writable: false` doing its job.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 42
43 class UserApi
44 {
↕ // ... lines 45 - 61
62     /**
63      * @var array<int, DragonTreasureApi>
64      */
65     public array $dragonTreasures = [];
↕ // ... lines 66 - 68
69 }
```

When we run the test again, you'll see the difference.

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

Yup! Still two treasures but the IDs are "1" and "3". So `UserApi` looks correct.

Going from DragonTreasureApi -> DragonTreasure

Now, we need to take this array of `DragonTreasureApi` objects and map them to `DragonTreasure` entity objects so we can set them onto the `User` entity. Once again, we need micro mapper!

You know the drill: add `private MicroMapperInterface $microMapper`... and back down here... start with `$dragonTreasureEntities = []`. I'm going to keep this simple and use a good old-fashioned `foreach`. Loop over `$dto->dragonTreasures` as `$dragonTreasureApi`. Then say `$dragonTreasureEntities[]` equals `$this->microMapper->map()`, passing `$dragonTreasureApi` and `DragonTreasure::class`. And as you may have already guessed, we're going to pass `MicroMapperInterface::MAX_DEPTH` set to `0`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 11
12 use Symfonycasts\MicroMapper\MicroMapperInterface;
↕ // ... lines 13 - 14
15 class UserApiToEntityMapper implements MapperInterface
16 {
17     public function __construct(
↕ // ... lines 18 - 19
20         private MicroMapperInterface $microMapper,
21     )
22     {
23     }
↕ // ... lines 24 - 37
38     public function populate(object $from, object $to, array $context): object
39     {
↕ // ... lines 40 - 50
51         $dragonTreasureEntities = [];
52         foreach ($dto->dragonTreasures as $dragonTreasureApi) {
53             $dragonTreasureEntities[] = $this->microMapper->map($dragonTreasureApi, DragonTreasure::class, [
54                 MicroMapperInterface::MAX_DEPTH => 0,
55             ]);
56         }
↕ // ... lines 57 - 59
60     }
61 }
```

`0` is fine here because we just need to make sure that the dragon treasure mapper queries for the correct `DragonTreasure` entity. If it has a relation, like `owner`, we don't care if *that* object is fully mapped & populated. Down here, `dd($dragonTreasureEntities)`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 11
12 use Symfonycasts\MicroMapper\MicroMapperInterface;
↕ // ... lines 13 - 14
15 class UserApiToEntityMapper implements MapperInterface
16 {
17     public function __construct(
↕ // ... lines 18 - 19
20         private MicroMapperInterface $microMapper,
21     )
22     {
23     }
↕ // ... lines 24 - 37
38     public function populate(object $from, object $to, array $context): object
39     {
↕ // ... lines 40 - 50
51         $dragonTreasureEntities = [];
52         foreach ($dto->dragonTreasures as $dragonTreasureApi) {
53             $dragonTreasureEntities[] = $this->microMapper->map($dragonTreasureApi, DragonTreasure::class, [
54                 MicroMapperInterface::MAX_DEPTH => 0,
55             ]);
56         }
57         dd($dragonTreasureEntities);
↕ // ... lines 58 - 59
60     }
61 }
```

Try it out!

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

And... looks good! We have 2 treasures with `id: 1`... and way down here `id: 3`.

Calling the Adder/Remover Methods

So all we need to do now is set that onto the `User` object. Say `$entity->set`... but... uh oh. We don't have a `setDragonTreasures()` method! And that's on purpose! Look inside the `User` entity. It has a `getDragonTreasures()` method, but no `setDragonTreasures()`. *Instead*, it has `addDragonTreasure()` and `removeDragonTreasure()`.

I won't dive too deeply into why we can't have a setter, but it relates to the fact that we need to set the *owning* side of the Doctrine relationship. We talk about this in our Doctrine relations tutorial.

The point is, if we *were* able to just call `->setDragonTreasures()`, it wouldn't save correctly. We need to call the adder and remover methods.

And this is tricky! We need to look at `$dragonTreasureEntities`, compare that with the *current* `dragonTreasures` property, then call the correct adders and removers for which ever treasures are new or removed. In our case, we need to call `removeDragonTreasure()` for the middle one and `addDragonTreasure()` for this third one.

Writing this code sounds... *annoying*... and complicated. *Fortunately*, Symfony *has* something that does this! It's a service called the "Property Accessor".

Head up here... and add `private PropertyAccessorInterface $propertyAccessor`.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\PropertyAccess\PropertyAccessorInterface;
↕ // ... lines 11 - 15
16 class UserApiToEntityMapper implements MapperInterface
17 {
18     public function __construct(
↕ // ... lines 19 - 21
22         private PropertyAccessorInterface $propertyAccessor,
23     )
24     {
25     }
↕ // ... lines 26 - 63
64 }
```

Property Accessor is good at *setting properties*. It can detect if a property is public... or if it has a setter method... or even adder, or remover methods. To use it, say `$this->propertyAccessor->setValue()` passing the object that we're setting data onto - the `User` `$entity`, the property we're setting - `dragonTreasures` - and finally, the *value*: `$dragonTreasureEntities`.

Down here, let's `dd($entity)` so we can see how it looks.

src/Mapper/UserApiToEntityMapper.php

```
↕ // ... lines 1 - 9
10 use Symfony\Component\PropertyAccess\PropertyAccessorInterface;
↕ // ... lines 11 - 15
16 class UserApiToEntityMapper implements MapperInterface
17 {
18     public function __construct(
↕ // ... lines 19 - 21
22         private PropertyAccessorInterface $propertyAccessor,
23     )
24     {
25     }
↕ // ... lines 26 - 39
40     public function populate(object $from, object $to, array $context): object
41     {
↕ // ... lines 42 - 58
59         $this->propertyAccessor->setValue($entity, 'dragonTreasures', $dragonTreasureEntities);
60         dd($entity);
↕ // ... lines 61 - 62
63     }
64 }
```

Deep breath. Try it:

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

Scroll up... to the `User` object. Look at `dragonTreasures`! It has *two* items with `id: 1` and `id: 3`! It correctly updated the `dragonTreasures` property! How the heck did it do that? By calling `addDragonTreasure()` for id 3 and `removeDragonTreasure()` for id 2.

I can prove it. Down here, add `dump('Removing treasure'.$treasure->getId())`.

When we run the test again...

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

There it is! Removing treasure 2! Life is good. Remove this `dump()`... as well as the other one over here.

Let's see some green. Run the test one last time... hopefully:

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

It *passes*! The final response contains treasures `1` and `3`. What happened to treasure `2`? It was actually *deleted* from the database *entirely*. Behind the scenes, its owner was set to `null`. *Then*, thanks to `orphanRemoval`, any time the *owner* of one of these `dragonTreasures` is set to `null`, it gets *deleted*. That's something we talked about in a previous tutorial.

Before we move on, we need to clean up the test. Remove the part where we are *stealing* `$dragonTreasure3`. We'll get rid of that object there, the part where we set it down here, change the length to `1`, and just test *that one*. So this now truly is a test for *removing* a treasure.

Celebrate by removing this `->dump()`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 56
57     public function testTreasuresCanBeRemoved(): void
58     {
59         $user = UserFactory::createOne();
60         $otherUser = UserFactory::createOne();
61         $dragonTreasure = DragonTreasureFactory::createOne(['owner' =>
   $user]);
62         DragonTreasureFactory::createOne(['owner' => $user]);
63 
64         $this->browser()
65             ->actingAs($user)
66             ->patch('/api/users/' . $user->getId(), [
67                 'json' => [
68                     'dragonTreasures' => [
69                         '/api/treasures/' . $dragonTreasure->getId(),
70                     ],
71                 ],
72                 'headers' => ['Content-Type' => 'application/merge-
   patch+json']
73             ])
74             ->assertStatus(200)
75             ->get('/api/users/' . $user->getId())
76             ->assertJsonMatches('length("dragonTreasures")', 1)
77             ->assertJsonMatches('dragonTreasures[0]', '/api/treasures/' .
   $dragonTreasure->getId())
78         ;
79     }
↕ // ... lines 80 - 113
114 }
```

But... treasures *can* still be stolen, which is lame. Let's fix the validator for this... but also make it a lot simpler, thanks to the DTO system, next.

Chapter 36: Simpler Validator for Checking State Change

We're down to *one failing test*. Apparently we *can* steal treasures by patching a user and sending `dragonTreasures` set to a treasure that's owned by someone else. This should give us a `422` status code, but we get *200*.

But no huge deal: we fixed this in the previous tutorial. *Now* we just need to reactivate and *adapt* that validator.

Re-Adding the Constraint

In `UserApi`, above the `$dragonTreasures` property, we can remove `#[ApiProperty]` and add `#[TreasuresAllowedOwnerChange]`.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 17
18 use App\Validator\TreasuresAllowedOwnerChange;
↕ // ... lines 19 - 43
44 class UserApi
45 {
↕ // ... lines 46 - 65
66     #[TreasuresAllowedOwnerChange]
67     public array $dragonTreasures = [];
↕ // ... lines 68 - 70
71 }
```

In the last tutorial, we put this above that same `$dragonTreasures` property, but inside the `User` entity. The validator would loop over each `DragonTreasure`, use Doctrine's `UnitOfWork` to get the `$originalOwnerId`, and *then* check to see if the `$newOwnerId` is different from the original. If it *was*, it would build a violation.

Adapting the Validator

First things first: the constraint will *not* be used on a property that holds a `Collection` object anymore: the new property holds a simple array. Also `dd($value)`.

src/Validator/TreasuresAllowedOwnerChangeValidator.php

```
↕ // ... lines 1 - 9
10 class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
11 {
↕ // ... lines 12 - 15
16     public function validate($value, Constraint $constraint): void
17     {
↕ // ... lines 18 - 23
24         dd($value);
↕ // ... lines 25 - 41
42     }
43 }
```

Over in the test, on top, put a `dump()` that says `Real owner is` with `$otherUser->getId()`. That'll help us track if it's stolen.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 80
81     public function testTreasuresCannotBeStolen(): void
82     {
↕ // ... lines 83 - 85
86         dump('Real owner is ' . $otherUser->getId());
↕ // ... lines 87 - 99
100     }
↕ // ... lines 101 - 114
115 }
```

Okay, run *just* this test:

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

And... perfect! The "Real owner" is supposed to be `2`, and the dump from the validator shows a single `DragonTreasureApi` object.

Reminder: this dump is the `dragonTreasures` property for the `UserApi` that's being updated. And, though we can't see it here, that user's id is 1. But, in the dump, look at the

owner: it's still `2`! That's *still* the correct owner!

When we make the PATCH request, this treasure is loaded from the database, transformed into a `DragonTreasureApi`, then set onto the `dragonTreasures` property of the `UserApi`. *But*, nothing has - yet - *changed* the treasure's `owner`: it still has the original `owner`.

The *problematic* part comes later when our state processor, really, `UserApiToEntityMapper`, maps the `dragonTreasures` property from `UserApi` to the `User` entity. That causes `User.addDragonTreasure()` to be called... and *that* causes `DragonTreasure.setOwner()` to be called... with the *new* `User` object.

So even though things *kind of* seem ok right now in the validator - the owner is still the original - the treasure *will* ultimately be stolen. Watch: add a `return` to the validator so it always passes. And in `UserResourceTest`, `->get('/api/users/'.$otherUser->getId())` and `->dump()`.

tests/Functional/UserResourceTest.php

```
↕ // ... lines 1 - 10
11 class UserResourceTest extends ApiTestCase
12 {
↕ // ... lines 13 - 80
81     public function testTreasuresCannotBeStolen(): void
82     {
↕ // ... lines 83 - 85
86         dump('Real owner is ' . $otherUser->getId());
87
88         $this->browser()
↕ // ... lines 89 - 98
99             ->get('/api/users/' . $otherUser->getId())->dump()
100            ->assertStatus(422);
101    }
↕ // ... lines 102 - 115
116 }
```

Run the test:

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

And... yup! The `dragonTreasures` field is *empty* for `$otherUser` because their treasure was stolen! They're mad!

Changing the Constraint to be above the Class

To sort out this mess in the validator, we need to know *two* things. First, what the *original* owner is for each treasure. And we have that: each `DragonTreasureApi` object stills has its original owner. *Second*, we need to know which *user* these treasures belong to now: which `UserApi` object this property belongs to. And we *don't* have that.

To get it, we can move the constraint from this specific property - where all we have access to are the `DragonTreasureApi` objects - up to the *class*. That will give us access to the entire `UserApi` object.

src/ApiResource/UserApi.php

```
↕ // ... lines 1 - 43
44 #[TreasuresAllowedOwnerChange]
45 class UserApi
46 {
↕ // ... lines 47 - 63
64     /**
65      * @var array<int, DragonTreasureApi>
66      */
67     public array $dragonTreasures = [];
↕ // ... lines 68 - 70
71 }
```

Step 1 is easy... move the constraint to be above the class! To allow this, open the constraint class. Get rid of the annotation stuff - since annotations are dead... and we're not using them. Then change this from `TARGET_PROPERTY` and `TARGET_METHOD` to `TARGET_CLASS`.

src/Validator/TreasuresAllowedOwnerChange.php

```
↕ // ... lines 1 - 6
7 #[\Attribute(\Attribute::TARGET_CLASS | \Attribute::IS_REPEATABLE)]
8 class TreasuresAllowedOwnerChange extends Constraint
9 {
↕ // ... lines 10 - 19
20 }
```

For some reason, my editor adds an extra `\` there, so delete that. We *also* need to override a method. I'm not sure why we have to specify the target in both places... this method is specific to the validation system, but no big deal: `return self::CLASS_CONSTRAINT`.

Also add a return type - `string|array`. That'll avoid a deprecation notice.

src/Validator/TreasuresAllowedOwnerChange.php

```
↕ // ... lines 1 - 6
7  #[\Attribute(\Attribute::TARGET_CLASS | \Attribute::IS_REPEATABLE)]
8  class TreasuresAllowedOwnerChange extends Constraint
9  {
↕ // ... lines 10 - 15
16     public function getTargets(): string|array
17     {
18         return self::CLASS_CONSTRAINT;
19     }
20 }
```

Back over in the validator, `dd($value)`... then rerun the test:

src/Validator/TreasuresAllowedOwnerChangeValidator.php

```
↕ // ... lines 1 - 9
10 class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
11 {
↕ // ... lines 12 - 15
16     public function validate($value, Constraint $constraint): void
17     {
↕ // ... lines 18 - 23
24         dd($value);
↕ // ... lines 25 - 41
42     }
43 }
```

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

Let's see... yes! It dumps the *entire* `UserApi` object with ID `1`. Good stuff! The `dragonTreasures` property holds that single treasure... and down here, we see its original owner! Now we can just check to see if the *new* owner is different from the *original* owner. Easy!

Back in the validator, `assert()` that `$value` is an `instanceof UserApi`.

src/Validator/TreasuresAllowedOwnerChangeValidator.php

```
↕ // ... lines 1 - 9
10 class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
11 {
12     public function validate($value, Constraint $constraint): void
13     {
↕ // ... lines 14 - 19
20         assert($value instanceof UserApi);
↕ // ... lines 21 - 35
36     }
37 }
```

Then, `foreach` over `$value->dragonTreasures as $dragonTreasureApi`.

src/Validator/TreasuresAllowedOwnerChangeValidator.php

```
↕ // ... lines 1 - 9
10 class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
11 {
12     public function validate($value, Constraint $constraint): void
13     {
↕ // ... lines 14 - 19
20         assert($value instanceof UserApi);
21
22         foreach ($value->dragonTreasures as $dragonTreasureApi) {
↕ // ... lines 23 - 34
35         }
36     }
37 }
```

The positively *lovely* thing is that we don't need *any* of this `$unitOfWork` stuff anymore. Delete it! Then say `$originalOwnerId = $dragonTreasureApi->owner->id`. The `$newOwnerId` will be `$value->id`. That's it!

To code defensively, you can add a `?` here... in case there *isn't* an owner... like if this is a new treasure.

src/Validator/TreasuresAllowedOwnerChangeValidator.php

```
↕ // ... lines 1 - 9
10 class TreasuresAllowedOwnerChangeValidator extends ConstraintValidator
11 {
12     public function validate($value, Constraint $constraint): void
13     {
↕ // ... lines 14 - 19
20         assert($value instanceof UserApi);
21
22         foreach ($value->dragonTreasures as $dragonTreasureApi) {
↕ // ... lines 23 - 24
25             $originalOwnerId = $dragonTreasureApi->owner?->id;
26             $newOwnerId = $value->id;
↕ // ... lines 27 - 34
35         }
36     }
37 }
```

The logic down here ain't broke, so nothing to fix: if we *don't* have the `$originalOwnerId` or the `$originalOwnerId` equals `$newOwnerId`, everything is cool. *Else*, build this violation. Remove this `$unitOfWork` line here as well, those `use` statements... and this `EntityManagerInterface` constructor. Thanks to the new DTO system, we now have a *very* boring custom validator.

Try the test again... and cross your fingers and toes for good luck:

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

We got it! High-five something, then remove this `->dump()` from the top. Deep breath: run the *entire* test suite:

```
symfony php bin/phpunit
```

All green! We have *completely* rebuilt our system using DTOs! Woohoo!

And... we're done! It took a bit of work to get this all set up, but that's the whole point of DTOs! There's more groundwork in the beginning in exchange for more flexibility and clarity later on, *especially* if you're building a really robust API that you want to keep stable.

As always, if you have questions, comments, or want to POST about the cool stuff you're building, we're here for you down in the comments. All right friends, seeya next time!

With <3 from SymfonyCasts