RESTful APIs in the Real World Course 2

With <3 from SymfonyCasts

Chapter 1: The Serializer: Swiss-Army Knife of APIs

THE SERIALIZER: SWISS-ARMY KNIFE OF APIS¶

Hey guys! Welcome to episode 2 of our RESTFUL API's in the real world series. In episode 1 we covered a lot of the basics. Phew! And explained really important confusing terms like representations, resources, idempotency, GET, POST, PUT, PATCH and a lot of other things that really make it difficult to learn REST. Because, really before that you had no idea what anybody was talking about. At This point I hope that you feel like you have a nice base because we're going to take that base and start doing some interesting things like talking about serializers, hypermedia HATEOAS, documentation and a ton more.

Tip

Hey, go download the starting code for this repository right on this page!

I've already started by downloading the code for Episode 2 onto my computer. So I'm going to move over to the `web/` directory, which is the document root for our project, and use the built-in PHP Web server to get things running:

```
cd /path/to/downloaded/code
cd web
php -S localhost:8000
```

Let's try that out in our browser... and there we go.

> http://localhost:8000

You can login as `ryan@knplabs.com` with password `foo` - very secure - and see the web side of our site. There's a web side of our site and an API version of our site, which we created in Episode 1.

The project is pretty simple, but awesome, you create programmers, give them a tag line, select from of our avatars and then battle a project. So we'll start a battle here, fight against "InstaFaceTweet"... some magic happens, and boom! Our happy coder has defeated the InstaFaceTweet project.

So from an API perspective, you can see that we have a number of resources: we have programmers, we have projects and we also have battles. And they all relate to each other, which will be really important for episode 2.

What We've Already Accomplished¶

The API code all lives inside of the `src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php` file. And in episode 1, we created endpoints for listing programmers, showing a single programmer, updating a programmer and deleting one:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

class ProgrammerController extends BaseController
{
    // ...

    public function newAction(Request $request)
    {
        // ...
    }

    // other methods for updating, deleting etc
}
```

We also used Behat to make some nice scenarios for this:

```
# features/api/programmer.feature
Feature: Programmer
  # ...

  Background:
    Given the user "weaverryan" exists

  Scenario: Create a programmer
    Given I have the payload:
      """
      {
        "nickname": "ObjectOrienter",
        "avatarNumber" : "2",
        "tagLine": "I'm from a test!"
      }
      """
    When I request "POST /api/programmers"
    Then the response status code should be 201
    And the "Location" header should be "/api/programmers/ObjectOrienter"

  # ... additional scenarios
```

This makes sure that our API doesn't break. It also lets us design our API, before we think about implementing it.

Whenever I have an API, I like to have an object for each resource. Before we started in episode 1, I created a `Programmer` object. And this is actually what we're allowing the API user to update, and we're using this object to send data back to the user:

```
<?php

// src/KnpU/CodeBattle/Model/Programmer.php
// ...

class Programmer
{
    public $id;

    public $nickname;

    public $avatarNumber;

    public $tagLine;

    // ... a few more properties
}
```

Serialization: Turning Objects into JSON or XML¶

So one of the key things we were doing was turning objects into JSON. For example, let's look in `showAction`. We're querying the database for a `Programmer` object, using a simple database-system I created in the background. Then ultimately, we pass that object into the `serializeProgrammer` function, which is a simple function we wrote inside this same class. It just takes a `Programmer` object and manually turns it into an array:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function showAction($nickname)
{
    $programmer = $this->getProgrammerRepository()->findOneByNickname($nickname);

    // ...
    $data = $this->serializeProgrammer($programmer);
    $response = new JsonResponse($data, 200);

    return $response;
}

// ...
private function serializeProgrammer(Programmer $programmer)
{
    return array(
        'nickname' => $programmer->nickname,
        'avatarNumber' => $programmer->avatarNumber,
        'powerLevel' => $programmer->powerLevel,
        'tagLine' => $programmer->tagLine,
    );
}
```

This transformation from an object into an array is really important because we're going to be doing it for every resource across all of our endpoints.

Installing JMS Serializer¶

The first thing we're going to talk about is a library that makes this a lot easier and a lot more powerful, called a serializer. The one I like is called jms/serializer, so let's Google for that. I'll show you how this works as we start using it. But the first step to installing any library is to bring it in via Composer.

I'm opening a new tab, and going back into the root of my project, and then copying that command:

```
composer require jms/serializer
```

If you get a "command not found" for `composer`, then you need to install it globally in your system. Or, you can go and download it directly, and you'll end up with a `composer.phar` file. In that case, you'll run `php composer.phar require` instead.

Creating/Configuring the Serializer Object¶

While we're waiting, let's go back and look at the usage a little bit. What we'll do is create an object called a "serializer", and there's this `SerializerBuilder` that helps us with this. Then we'll pass it some data, which for us will be a `Programmer` object or a `Battle` object. And then it returns to you the actual JSON string. So it takes an object and turns it into a string:

```
// from the serialization documentation
$serializer = JMS\Serializer\SerializerBuilder::create()->build();
$jsonContent = $serializer->serialize($data, 'json');
echo $jsonContent; // or return it in a Response
```

Now this is a little bit specific to Silex, which is the framework we're building our API on, but in Silex, you have a spot where you can globally configure objects that you want to be able to use. They're called services. I'll create a new global object called `serializer` and we'll use code similar to what you just saw to create the `serializer` object. We're doing this because it will let me access that object from any of my controllers:

```
// src/KnpU/CodeBattle/Application.php
// ...

private function configureServices()
{
    // ...

    $this['serializer'] = $this->share(function() use ($app) {
        // todo ...
    });
}
```

Before I start typing anything here, I'll make sure everything is done downloading. Yes, it is - so I should be able to start referencing the serializer classes. Start with the `SerializerBuilder` that we saw. We also need to set a cache directory, because this library caches annotations that we'll use a bit later. This is a fancy way in my app to tell it to use a `cache/serializer` directory at the root of my project.

There's also a debug flag, and when that's true, it'll rebuild the cache automatically. Finally, the last step tells the serializer to use the same property names that are on the `Programmer` object as the keys on the JSON. In other words, don't try to transform them in any way. And that's it!

```
// src/KnpU/CodeBattle/Application.php
// ...

private function configureServices()
{
    // ...

    $this['serializer'] = $this->share(function() use ($app) {
        return \JMS\Serializer\SerializerBuilder::create()
            ->setCacheDir($app['root_dir'].'/cache/serializer')
            ->setDebug($app['debug'])
            ->setPropertyNamingStrategy(new IdenticalPropertyNamingStrategy())
            ->build();
    });
}
```

Using the Serializer Object¶

The important thing here is that we have a `serializer` object and we can access it from any of our controllers. Let's open our `ProgrammerController` and rename `serializeProgrammer` to just `serialize` since it can serialize anything.

I've setup my application so that I can reference any of those global objects by saying `$this->container['serializer']`. This will look different for you: the important point is that we need to access that object we just configured.

Now, just call `serialize()` on it, just like the documentation told us. I'll put `json` as the second argument so we get JSON. The serializer can also give us another format, like XML:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

protected function serialize($data)
{
    return $this->container['serializer']->serialize($data, 'json');
}
```

Perfect! Now let's look to see where we used the `serializeProgrammer` function before. That old function returned an array, but `serialize` now returns the JSON string. So now we can return a normal `Response` object and just pass the JSON string that we want. The one thing we'll lose is the `Content-Type` header being set to `application/json`, but we'll fix that in a second:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function newAction(Request $request)
{
    $programmer = new Programmer();
    // ...

    $json = $this->serialize($programmer);
    $response = new Response($json, 201);

    // ... setting the Location header

    return $response;
}
```

Let’s go and make similar changes to the rest of our code.

In fact, when we have the collection of `Programmer` objects, things get much easier. We can pass the entire array of objects and it’s smart enough to know how to serialize that. You can already start to see some of the benefits of using a serializer:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function listAction()
{
    $programmers = $this->getProgrammerRepository()->findAll();
    $data = array('programmers' => $programmers);
    $json = $this->serialize($data);

    $response = new Response($json, 200);

    return $response;
}
```

Compared with what we had before, not a lot should have changed, because the serializer should give us a JSON structure with all the properties in `Programmer`. That’s practically the same as we were doing before.

So let’s run our tests!

```
php vendor/bin/behat
```

We’ve totally changed how a Programmer gets turned into JSON, but *almost* every test passes already! We’ll debug that failure next.

Chapter 2: Serializer Configuration (SerializationContext)

SERIALIZER CONFIGURATION (SERIALIZATIONCONTEXT)¶

Time to tackle that test failure! You can see that there is one strange problem. The “GET one programmer” scenario says it’s expecting a `tagLine` property, but you can see that the JSON response has a lot of other fields, but not `tagLine`. I know what the issue is, so we’ll fix it in a second.

But first, one thing the test didn’t show yet, is that we’re losing the `Content-Type` of `application/json`. We need to centralize the logic that creates our Response as much as possible so that all of our responses are very very consistent.

For example, right now - we’re creating the `Response` in every controller, and so every controller method is responsible for remembering to set the `Content-Type` header:

```
src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function newAction(Request $request)
{
    // ...
    $response = new Response($json, 201, array(
        'Content-Type' => 'application/json'
    ));
    // ...
}

// new Response() exists in several other methods too
```

That’s pretty error-prone, which means I am guaranteed to mess it up.

Central Method for Creating all API Responses¶

Instead, let’s go to `BaseController`, the parent class of `ProgrammerController`. Let’s create a couple of new functions. First, back in `ProgrammerController`, cut the `serialize` function, move it into the `BaseController` and change it to be `protected`:

```
// src/KnpU/CodeBattle/Controller/BaseController.php
// ...

protected function serialize($data, $format = 'json')
{
    return $this->container['serializer']->serialize($data, $format);
}
```

Now, when we have multiple controllers in the future, we can just re-use this method to serialize other resources, like Battles.

Second, create a new function called `createApiResponse` with 2 arguments: the data and the status code. The data could be a `Programmer` a `Battle` or anything else. Then we’ll let *it* call the `serialize` function. And finally, create the `Response` and make sure the `Content-Type` header is set perfectly:

```
// src/KnpU/CodeBattle/Controller/BaseController.php
// ...

protected function createApiResponse($data, $statusCode = 200)
{
    $json = $this->serialize($data);

    return new Response($json, $statusCode, array(
        'Content-Type' => 'application/json'
    ));
}
```

Back in `ProgrammerController`, we can simplify a lot of things. Let's search for `new Response`, because we can replace these. In `newAction` we can say `$response = $this->createApiResponse()` and pass it the `$programmer` object and the 201 status code. And we can still add any other headers we need:

```
src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function newAction(Request $request)
{
    $programmer = new Programmer();
    // ...

    $response = $this->createApiResponse($programmer, 201);

    $programmerUrl = $this->generateUrl(
        'api_programmers_show',
        ['nickname' => $programmer->nickname]
    );
    $response->headers->set('Location', $programmerUrl);

    return $response;
}
```

I'll copy this code and change the other spots in this controller. The `deleteAction` returns a 204, which is the blank response. So there's no need to use this fancy serialization stuff here.

Now let's try the tests again, so we can make sure we see *just* that same one failure:

```
php vendor/bin/behat
```

And we do!

To Serialize or Not Serialize Null Values?¶

This failure is caused by something specific to the serializer. In this test, the programmer doesn't actually have a `tagLine` - we could see this if we looked in the database:

```
# features/api/programmer.feature
# ...

Scenario: GET one programmer
  Given the following programmers exist:
    | nickname   | avatarNumber |
    | UnitTester | 3            |
  When I request "GET /api/programmers/UnitTester"
  Then the response status code should be 200
  And the following properties should exist:
    """
    nickname
    avatarNumber
    powerLevel
    tagLine
    """
  And the "nickname" property should equal "UnitTester"
```

When the serializer sees `null` values, it has 2 options: return the property with a `null` value, or omit the property entirely. By default, the serializer actually omits `null` properties. We'll fix this, because I like always having the same fields on a response, even if some of the data is empty.

Go back to the `serialize` function. To configure the serializer, create a `SerializationContext` object. Next, call `setSerializeNull` and pass `true`. Now, pass this context as the 3rd argument to `serialize`:

```
// src/KnpU/CodeBattle/Controller/BaseController.php
// ...

protected function serialize($data, $format = 'json')
{
    $context = new SerializationContext();
    $context->setSerializeNull(true);

    return $this->container['serializer']->serialize($data, $format, $context);
}
```

There's not a lot you can customize in this way, but `serializeNull` happens to be one of them.

Back to the tests!

```
php vendor/bin/behat
```

Boom, everything passes! We've changed to use the serializer, and it's now taking care of all of the heavy-lifting for us. This will be really powerful as we serialize more and more objects.

Chapter 3: Serializer Annotations

SERIALIZER ANNOTATIONS¶

Alright, now let me show you one other powerful thing about the serializer. The control you have over your objects! By default, it serializes *all* of your properties. But let's say we *don't* want the `userId` to be serialized. After all this isn't really an important field for the API, it's just the `id` of the user that created the programmer.

To start, open up the feature file and add a line to test that this property isn't in the response:

```
# features/api/programmer.feature
# ...

Scenario: GET one programmer
  Given the following programmers exist:
    | nickname   | avatarNumber |
    | UnitTester | 3            |
  When I request "GET /api/programmers/UnitTester"
  # ...
  And the "userId" property should not exist
```

We're using a custom Behat context that I created for this project with a lot of these nice sentences. To see a list of all of them, you can run behat with the `-dl` option, which stands for "definition list":

```
php vendor/bin/behat -dl
```

Try running the tests again. It *should* fail, and it does - saying that the `userId` property should not exist, but we can see that it in fact is there.

Configuring the Serializer with Annotations¶

As soon as you want to take control over what properties are returned, we're going to use annotations. Let's look at their documentation first and find the Annotation Reference section - this is by far the most interesting page. The first item on the list is what we need, but there's a huge list of annotations that give you all sorts of control.

Remember, whenever you use an annotation, you need a `use` statement for it. PHPStorm is tagging in to help me auto-complete the `ExclusionPolicy` class. Then I'll remove the last part and alias this to `Serializer`. This will allow us to use any of the JMS serializer annotations by starting with `@Serializer`:

```
// src/KnpU/CodeBattle/Model/Programmer.php
// ...

/**
 * @Serializer\ExclusionPolicy("all")
 */
class Programmer
{
    // ...
}
```

For example, on top of the class, we can say `@Serializer\ExclusionPolicy("all")`. We've now told the serializer to not serialize *any* of the properties in this class, unless we tell it to specifically. Whereas before, it was serializing *everything*.

To actually include things, we whitelist them with the `@Serializer\Expose` annotation. I'll copy this and use it on `nickname`, `avatarNumber`, `tagLine` and `powerLevel` fields:

```
// src/KnpU/CodeBattle/Model/Programmer.php
// ...

/**
 * @Serializer\ExclusionPolicy("all")
 */
class Programmer
{
    // ...

    public $id;

    /**
     * @Serializer\Expose
     */
    public $nickname;

    /**
     * @Serializer\Expose
     */
    public $avatarNumber;

    /**
     * @Serializer\Expose
     */
    public $tagLine;

    public $userId;

    /**
     * @Serializer\Expose
     */
    public $powerLevel = 0;
}
```

This is just *one* of the customizations you can make with annotations.

Now let's re-run the test:

```
php vendor/bin/behat
```

Success! This time the `userId` is *not* returned in our JSON.

If you want to know more, check out that annotation reference section. But we're also going to do more in the next videos.

Chapter 4: Requiring Authentication

REQUIRING AUTHENTICATION¶

Let's talk about API Authentication. The web part of our site *does* have authentication. I can login as `ryan@knplabs.com` password `foo`. By the way, if that ever doesn't work for you, go back and just delete the SQLite database we're using:

```
rm data/code_battles.sqlite
```

This is because when we run our Behat tests, it's messing with our database. That's not something we'd stand for. In a real project, we'd want to fix that!

When we're logged in as Ryan, any programmer we create is attached to the Ryan user in the database. The web side has authentication and we know who created which programmers.

In our API, that's not the case at all: we have no authentication. And so we're kind of faking the attachment of a programmer to a user. In `newAction` we call this function `handleRequest`, which lives right in this same class. Then down here on the bottom, you can see *how* we fake it:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

private function handleRequest(Request $request, Programmer $programmer)
{
    // update the programmer from the request data
    // ...

    $programmer->userId = $this->findUserByUsername('weaverryan')->id;
}
```

We assume there's a user called `weaverryan` in the database, grab it's `id` and attach that to every programmer. So obviously that's just hardcoded silliness and we need to get rid of it! We need some way for the API request to authenticate itself as a user.

Denying Access with a 401 (in Behat)¶

We're going to do this with tokens. You can create your own simple token system or you can do something bigger like OAuth. The important thing to realize is that the end results are the same: the request will send a token and you'll look that up in some database and figure out which user the token belongs to and if they have access. Using OAuth is just a different way to help you handle those tokens and get those to your users.

Before we start anything let's write a new Behat feature file, because we haven't done anything with authentication yet. For Behat newbies, this will feel weird, but stick with me. I'm going to describe here the business value of our authentication feature and who benefits from it, which is the API client. That part is not important technically, but we'd pity the fool who doesn't think about why they're building a feature:

```
# features/api/authentication.feature
Feature: Authentication
  In order to access protected resource
  As an API client
  I need to be able to authenticate
```

Below that, let's add our first scenario, which is going to be one where we try to create a programmer without sending a token. Ultimately we want to require authentication on most endpoints.

```
# features/api/authentication.feature
Feature: Authentication
  # ...

  Scenario: Create a programmer without authentication
    When I request "POST /api/programmers"
    Then the response status code should be 401
    And the "detail" property should equal "Authentication Required"
```

If I just post to `/api/programmers` with no token, the response I get should be 401. We don't know what the response is going to look like yet, but let's imagine that it's JSON and has some `detail` key on it which gives us a little bit of information about what went wrong.

So there we go! No coding yet, but we know how things should look if we don't send a token. If we try this right now - which is always a good idea - we should see it fail. By the way we can run one feature file at a time, and that's going to be really useful for us:

```
php vendor/bin/behat features/api/authentication.feature
```

And there we go! You can see it's giving us a 400 error because we're sending invalid JSON instead of the 401 we want.

Denying Access in your Controller¶

So let's go into `ProgrammerController` and fix this as easily as possible up in `newAction`. I already have a function called `getLoggedinUser()`, which will either give us a `User` object or give us `null` if the request isn't authenticated. What we can do is check if `!getLoggedInUser()` and then return the access denied page. In Silex, you do this by throwing a special exception called `AccessDeniedException`. Make sure you have the right `use` statement for this, which is in the `Security` component:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function newAction(Request $request)
{
    if (!$this->isUserLoggedIn()) {
        throw new AccessDeniedException();
    }
    // ..
}
```

And there we go! Since that should deny access, let's try our test:

```
php vendor/bin/behat features/api/authentication.feature
```

Boom! And this time you can see that we *are* passing the 401 test. Get outta here ya tokenless request! The only issue is that the response body is coming back slightly different than we expected. There *is* a `detail` property, but instead of it being set to “authentication required”, it's set to this `not privileged to request the resource` string. And yea, it's not really obvious where that's coming from.

What Happens when you Deny Access Anyways?¶

But first, how is this even working behind the scenes? If you were with us for episode 1, we did some pretty advanced error handling stuff at the bottom of this `Application` class. What we basically did was add a function that's called anytime there's an exception thrown *anywhere*. Then, we transform that into a consistent JSON response that follows the API problem format. So any exception gives us a nice consistent response:

```
src/KnpU/CodeBattle/Application.php
// ...

$this->error(function(\Exception $e, $statusCode) use ($app) {
    // ...

    $data = $apiProblem->toArray();

    $response = new JsonResponse(
        $data,
        $apiProblem->getStatusCode()
    );
    $response->headers->set('Content-Type', 'application/problem+json');

    return $response;
});
```

And hey, when we deny access, *we're throwing an exception*! But bad news, our exception is the *one* weird guy in the *whole* system: instead of being handled here, it's handled somewhere else entirely.

ApiEntryPoint: Where Security Responses are Created¶

Without getting too far into things, I've already written *most* of the logic for our token authentication system, and I'll show you the important parts but not cover how this all hooks up. If you have more detailed questions, just ask them in the comments.

Let's open up a class in my `Security/Authentication` directory called `ApiEntryPoint`. When we deny access, the `start()` method is called. Here, it's our job to return a Response that says "hey, get outta here!":

```
// src/KnpU/CodeBattle/Security/Authentication
// ...

use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Http\EntryPoint\AuthenticationEntryPointInterface;
// ...

class ApiEntryPoint implements AuthenticationEntryPointInterface
{
    // ...

    public function start(Request $request, AuthenticationException $authException = null)
    {
        $message = $this->getMessage($authException);

        $response = new JsonResponse(array('detail' => $message), 401);

        return $response;
    }

    /**
     * Gets the message from the specific AuthenticationException and then
     * translates it. The translation process allows us to customize the
     * messages we want - see the translations/en.yml file.
     */
    private function getMessage(AuthenticationException $authException = null)
    {
        $key = $authException ? $authException->getMessageKey() : 'authentication_required';
        $parameters = $authException ? $authException->getMessageData() : array();

        return $this->translator->trans($key, $parameters);
    }
}
```

So *that's* where the `detail` key is coming from, and it's set to some internal message that comes from the deep dark core of Silex's security that describes what went wrong.

Fortunately, I'm also translating that message, and we setup translations in episode 1. in this `translations/` directory. This is just a little key value pair. So let's copy that unhelpful "this not privileged request" thing that Silex's gives us and translate it to something nicer:

```
# translations/en.yml
authentication_required: Authentication Required
"Not privileged to request the resource.": "Authentication Required"
```

Ok, rerun the tests!

```
php vendor/bin/behat features/api/authentication.feature
```

Fantastic!

Chapter 5: Authorization via a Token

AUTHORIZATION VIA A TOKEN¶

So we can deny access and turn that into a nice response. Cool. Now we need to create something that'll look for a token on the request and authenticate us so we can actually create programmers!

You should know where to start: in one of our feature files. We're going to modify an existing scenario. See the `Background` of our programmer feature file? One of the things that we do before every single scenario is to make sure the `weaverryan` user exists in the database. We aren't sending authentication headers, just guaranteeing that the user exists in the database:

```
# features/api/programmer.feature
# ...

Background:
  Given the user "weaverryan" exists

# ... scenarios
```

Sending an Authorization Header on each Request¶

In the background, I already have a database table for tokens, and each token has a foreign-key relation to one user. So I'm going to extend the `Background` a little bit to create a token in that table that relates to `weaverryan`. And this is the important part, this says that on whatever request we make inside of our scenario, I want to send an `Authorization` header set to token, a space then `ABCD123`:

```
# features/api/programmer.feature
# ...

Background:
  Given the user "weaverryan" exists
  And "weaverryan" has an authentication token "ABCD123"
  And I set the "Authorization" header to be "token ABCD123"

# ... scenarios
```

Why did I choose to set the `Authorization` header or this "token space" format? Technically, none of this is important. In a second, you'll see us grab and parse this header. If you use OAuth, it has directions on the type of names you should give these things. So I'm just using authorization header and the word token, space and the actual authentication token that we're sending.

Hey You! Use HTTPS. No Excuses.¶

By the way, we aren't doing it in this tutorial, but one thing that that's really important for authentication across your API is that you only do it over HTTPS. The easiest way to do this is to require HTTPS across your entire API. Otherwise, these authentication tokens are flying through the internet via plain text, and that's a recipe for disaster.

If we rerun one of our tests right now, it's not going to make any difference. To prove it, let's rerun the first scenario of `programmer.feature`, which starts at line 11. So we say `:11` and it's going to fail:

```
php vendor/bin/behat features/api/programmer.feature:11
```

It *is* setting that `Authorization` header, but we aren't actually doing anything with it yet in our app. So we're getting that 401 authentication required message.

Authenticating a User via a Token¶

So let’s hook this up! Some of this is specific to Silex’s security system, but in case you’re using something else, we’ll stay high level enough to see what types of things you need to do in your system to make it happen. As always, if you have questions, just ask them in the comments!

Inside this `Security/` directory here, I’ve already set up a bunch of things for an API token authentication system

1) ApiTokenListener: Gets the Token from the Request¶

The first thing we’re going to do is open this `ApiTokenListener`. I’ve written some fake code in here as you can see:

```php
// src/KnpU/CodeBattle/Security/Authentication/ApiTokenListener.php
// ...

class ApiTokenListener implements ListenerInterface
{
    // ...

    public function handle(GetResponseEvent $event)
    {
        // ...
        $request = $event->getRequest();

        // there may not be authentication information on this request
        if (!$request->headers->has('Authorization')) {
            return;
        }

        // TODO - remove this return statement and add real code!
        return;
        // format should be "Authorization: token ABCDEFG"
        $tokenString = 'HARDCODED';

        if (!$tokenString) {
            // there's no authentication info for us to process
            return;
        }

        // some code that sends the tokenString into the Silex security system
        // ...
    }

    // ...
}
```

The job of the listener is to look at the request object and get the token information off of it. And hey, since we’re sending the token on the `Authorization` header, we are going to look for it there. So let’s get rid of this hard coded text and instead go get that `Authorization` header. You can say `$request->headers->get('Authorization')`. That’s going to get you the actual raw `token ABCD123` type of thing:

```php
// src/KnpU/CodeBattle/Security/Authentication/ApiTokenListener.php
// ...

public function handle(GetResponseEvent $event)
{
    // ...
    $request = $event->getRequest();

    $authorizationHeader = $request->headers->get('Authorization');
    // ...
}
```

Next, since the actual token is the second part of that, we need to parse it out. I'll say, `$tokenString = $this->parseAuthorizationHeader()`, which is a function I've already created down here. It's a private function that expects a format of token space and gets the second part for you:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiTokenListener.php
// ...

public function handle(GetResponseEvent $event)
{
    // ...
    $request = $event->getRequest();

    $authorizationHeader = $request->headers->get('Authorization');
    $tokenString = $this->parseAuthorizationHeader($authorizationHeader);
    // ...
}

/**
 * Takes in "token ABCDEFG" and returns "ABCDEFG"
 */
private function parseAuthorizationHeader($authorizationHeader)
{
    // ...
}
```

Perfect!

At this point the `$tokenString` is `ABCD123`. So that's all I want to talk about in this `TokenListener`, it's the only job of this class.

1) ApiTokenProvider: Uses the Token to find a User¶

Next, I'm going to open up the `ApiTokenProvider`. Its job is to take the token string `ABCD123` and try to look up a valid `User` object in the database for it. First, remember how I have an `api_token` table in my database? Let me show you some of the behind-the-scenes magic:

```
// src/KnpU/CodeBattle/DataFixtures/FixturesManager.php
// this is an internal class that creates our database tables

$tokenTable = new Table('api_token');
$tokenTable->addColumn('id', 'integer'();
$tokenTable->addColumn('token', 'string', array('length' => 32));
$tokenTable->addColumn('userId', 'integer');
$tokenTable->addColumn('notes', 'string', array('length' => 255));
$tokenTable->addColumn('createdAt', 'datetime');
// ...
```

You can see here I am creating an `api_token` table. It has a token column which is the string and a `user_id` column which is the user it relates to. So you can imagine a big table full of tokens and each token is related to exactly one user. For example, if we look up the entry in the token table, we can figure out "yes" this is a valid token and it is a valid token for a user whose `id` is `5`.

So here, the first thing we'll do is actually go and look up the token row. I don't want to get into the details of exactly how this all hooks up because I want to focus on REST. But I've already configured this class and created some code behind the scenes to take in a token string, which is the `ABCD123` thing in our case and return to me an `ApiToken` object, which represents a row in that table:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiTokenProvider.php
// ...

class ApiTokenProvider implements AuthenticationProviderInterface
{
    // ...

    public function authenticate(TokenInterface $token)
    {
        // the actual token string value from the header - e.g. ABCDEFG
        $tokenString = $token->getCredentials();

        // find the ApiToken object in the database based on the TokenString
        $apiToken = $this->apiTokenRepository->findOneByToken($tokenString);

        if (!$apiToken) {
            throw new BadCredentialsException('Invalid token');
        }

        // ... finishing code that's already written ...
    }
    // ...
}
```

So we’ve taken the string and we’ve queried for a row in the table. If we don’t have that row, we throw an exception which causes a 401 bad credentials error.

Next, when we have that, we just need to look up the `User` object from it. Remember, the job of this class is start with the token string and eventually give us a `User` object. And it does that by going through the `api_token` table:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiTokenProvider.php
// ...

class ApiTokenProvider implements AuthenticationProviderInterface
{
    // ...

    public function authenticate(TokenInterface $token)
    {
        // the actual token string value from the header - e.g. ABCDEFG
        $tokenString = $token->getCredentials();

        // find the ApiToken object in the database based on the TokenString
        $apiToken = $this->apiTokenRepository->findOneByToken($tokenString);

        if (!$apiToken) {
            throw new BadCredentialsException('Invalid token');
        }

        $user = $this->userRepository->find($apiToken->userId);

        // ... finishing code that's already written ...
    }
    // ...
}
```

And that’s the job of this `ApiTokenProvider` class. It’s technical and at the core of Silex’s security system, so I just want you to internalize what it does.

It Works! Get the Logged-In User¶

At this point - between these two classes and a few other things I’ve setup - if we send this `Authorization` header with a valid token, by the time we get it to our `ProgrammerController`, `$this->getLoggedInUser()` will actually return to us the `User` object that’s attached to the token that was sent:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function newAction(Request $request)
{
    // will return the User related to the token form the Authorization header!
    if (!$this->isUserLoggedIn()) {
        throw new AccessDeniedException();
    }
    // ...
}
```

In the case of our scenario, since we're sending a token of `ABCD123`, it means that we'll get a `User` object that represents this `weaverryan` user. We will actually be logged in, except we're logged in via the API token. So, let's try this out.

```
php vendor/bin/behat features/api/programmer.feature:11
```

And there it is!

The guts for getting this all working can be complicated, but the end result is so simple: send an `Authorization` header with the api token and use that to look in your database and figure out which `User` object if any this token is attached to.

So now, in `handleRequest()`, I have this ugly hard-coded logic that assumed that there is a user called `weaverryan`. Replace this garbage with `$this->getLoggedInUser()` to get the real user object that's attached to our token:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

private function handleRequest(Request $request, Programmer $programmer)
{
    // ...

    $programmer->userId = $this->getLoggedInUser()->id;
}
```

Chapter 6: Securing More Endpoints

SECURING MORE ENDPOINTS¶

Hey! We have this great system where users are actually being authenticated! Now we can start checking for security everywhere we need it. In `newAction` we're requiring that you are logged in:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function newAction(Request $request)
{
    if (!$this->isUserLoggedIn()) {
        throw new AccessDeniedException();
    }
    // ...
}
```

Awesome! In `showAction` and `listAction` we are going to leave those anonymous. In `updateAction`, we *do* need some extra security. It's more than just being logged in: we need to check to see if our user is actually the owner of that programmer or not. So we just add some `if` statement logic: `if ($programmer->userId != $this->getLoggedInUser()->id`, then `throw new AccessDeniedException`:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function updateAction($nickname, Request $request)
{
    $programmer = $this->getProgrammerRepository()->findOneByNickname($nickname);

    if ($programmer->userId != $this->getLoggedInUser()->id) {
        throw new AccessDeniedException();
    }

    // ...
}
```

Easy enough!

Centralizing Security Logic Checks¶

Since we're also going to use this in `deleteAction` let's go into our `BaseController` and make this a generic function. Open up the `BaseController`, create a new protected function `enforceProgrammerOwnershipSecurity`. Let's copy the logic in there and don't forget to add your `AccessDeniedException` `use` statement:

```
// src/KnpU/CodeBattle/Controller/BaseController.php
// ...
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

protected function enforceProgrammerOwnershipSecurity(Programmer $programmer)
{
    if ($this->getLoggedInUser()->id != $programmer->userId) {
        throw new AccessDeniedException();
    }
}
```

Perfect, so now go back to our `ProgrammerController`. It's a lot easier to just reuse this logic. Let's also use this down in `deleteAction`:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function updateAction($nickname, Request $request)
{
    $programmer = $this->getProgrammerRepository()->findOneByNickname($nickname);
    // ...

    $this->enforceProgrammerOwnershipSecurity($programmer);

    // ...
}

public function deleteAction($nickname)
{
    $programmer = $this->getProgrammerRepository()->findOneByNickname($nickname);
    // ...

    $this->enforceProgrammerOwnershipSecurity($programmer);

    // ...
}
```

Now the only other thing that could go wrong, is if the user is not logged in at all and they hit `updateAction`. Then, we would die inside this function. The problem is that `$this->getLoggedInUser` would be null and we'll call the `id` property on a null object:

```
// src/KnpU/CodeBattle/Controller/BaseController.php
// ...

protected function enforceProgrammerOwnershipSecurity(Programmer $programmer)
{
    if ($this->getLoggedInUser()->id != $programmer->userId) {
        throw new AccessDeniedException();
    }
}
```

So before we call this function, we need to make sure the user is at least logged in. If they aren't, then they are definitely not the owner of this programmer.

So let's create another function here called `enforceUserSecurity`. In this case, go back to `ProgrammerController` and grab the logic right here:

```
// src/KnpU/CodeBattle/Controller/BaseController.php
// ...

protected function enforceUserSecurity()
{
    if (!$this->isUserLoggedIn()) {
        throw new AccessDeniedException();
    }
}
```

There we go. And from inside `enforceProgrammerOwnershipSecurity` we can just make sure that the user is actually logged in:

```
// src/KnpU/CodeBattle/Controller/BaseController.php
// ...

protected function enforceProgrammerOwnershipSecurity(Programmer $programmer)
{
    $this->enforceUserSecurity();

    if ($this->getLoggedInUser()->id != $programmer->userId) {
        throw new AccessDeniedException();
    }
}
```

And in `ProgrammerController`, we can do the same thing and save ourselves a little bit of code:

```
// src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
// ...

public function newAction(Request $request)
{
    $this->enforceUserSecurity();
    // ...
}
```

Between these two new methods, we have a really easy way to go function by function inside of our controller to make sure that we're enforcing the right type of security.

Because we're sending our `Authorization` header in the background of our scenarios, we should be able to run our entire `programmer.feature` and see it pass:

```
php vendor/bin/behat features/api/programmer.feature
```

Perfect! And just like that, we have our entire application locked down.

Chapter 7: Authentication Error Format

AUTHENTICATION ERROR FORMAT¶

It's finally time to create a scenario to check and see what happens if we send an *invalid* token. So let's do that right now.

In this case, I'm *not* going to add a user to the database with this token. I'm just going to send a token that doesn't exist:

```
# features/api/authentication.feature
# ...

Scenario: Invalid token gives us a 401
  Given I set the "Authorization" header to be "token ABCDFAKE"
  When I request "POST /api/programmers"
  Then the response status code should be 401
  And the "detail" property should equal "Invalid Credentials"
```

Then, we'll just make any request that requires authentication. The response status code should be 401 and remember we're always returning that API problem format that has a `detail` property on it. And here, we can say whatever we want. To be nice to the users, let's set it to "Invalid Credentials" so they know what went wrong. It's not like they forgot the token, it just wasn't valid.

Let's try this out. Again we can run Behat on one particular scenario. This one starts on line 11:

```
php vendor/bin/behat features/api/authentication.feature:11
```

In fact you can see it *almost* passed, so out of the box things are working. We are denying access, sending a 401, and because of our security error handling in that `ApiEntryPoint` class, we're sending a nice api problem format with the actual `detail` set to "Invalid Credentials." Like before, this message comes from deep inside Silex and is describing what's going wrong.

And because I want people to be excited about our API, I'm even going to add an exclamation point to this:

```
# features/api/authentication.feature
# ...

Scenario: Invalid token gives us a 401
  # ...
  And the "detail" property should equal "Invalid Credentials!"
```

We'll see the difference this makes. We're expecting "Invalid Credentials!" and we are getting it with a period. So let's go find our translation file and change this to our version:

```
# translations/en.yml
# ...

"Invalid credentials.": "Invalid credentials!""
```

That should do it! Let's rerun things. Woops! I made a mistake - take that extra quote off. And I made one other mistake: it's catching me on a case difference. So this is why it is good to have tests, they have closer eyes than we do. So I'll say "Invalid Credentials!" and a capital letter:

```
# translations/en.yml
# ...

"Invalid credentials.": "Invalid Credentials!""
```

Perfect!

Returning application/problem+json for Security Errors¶

Next, we need all of our errors to *always* return that same API problem response format. And when we return this format, we should always send back its special `Content-Type` so let's make sure it's correct:

```
# features/api/authentication.feature
# ...

Scenario: Create a programmer without authentication
  # ...
  And the "Content-Type" header should be "application/problem+json"
```

Ahh! It's not coming back with that. We are getting an application/problem-like format, but without the right `Content-Type` header. It's coming back as a simple `application/json`.

In our app, when an exception is thrown, there are 2 different places that take care of things. Most errors are handled in the `Application` class. We added this in episode 1. But security errors are handled in `ApiEntryPoint`, and it's responsible for returning some helpful response:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiEntryPoint.php
// ...

public function start(Request $request, AuthenticationException $authException = null)
{
    $message = $this->getMessage($authException);

    $response = new JsonResponse(array('detail' => $message), 401);

    return $response;
}
```

So for example here, you can see why we get the `detail` and why we get the 401. If I change this to 403, this proves that this class is responsible for the error responses. Let's add the `application/problem+json` `Content-Type` header:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiEntryPoint.php
// ...

public function start(Request $request, AuthenticationException $authException = null)
{
    $message = $this->getMessage($authException);

    $response = new JsonResponse(array('detail' => $message), 401);
    $response->headers->set('Content-Type', 'application/problem+json');

    return $response;
}
```

Using the ApiProblem Class For Security Errors¶

For consistency, one of the things we did in Episode 1 is actually create an `ApiProblem` class. The idea was whenever you had some sort of error response you needed to send back, you could create this `ApiProblem` object, which will help you structure things and avoid typos in any keys.

Right now inside of the `ApiEntryPoint`, we're kind of creating the API problem structure by hand, which is something I don't want to do. Let's leverage our `ApiProblem` class instead.

So first, I'm closing a couple of these classes. Inside `ApiProblem` there is a `type` property. The spec document that describes this format says that we should have a `type` field and that it should be a unique string for each error in your application. Right now we have two: `validation_error` as one unique thing that can go wrong and `invalid_body_format` as another:

```
// src/KnpU/CodeBattle/Api/ApiProblem.php
// ...

class ApiProblem
{
    const TYPE_VALIDATION_ERROR = 'validation_error';
    const TYPE_INVALID_REQUEST_BODY_FORMAT = 'invalid_body_format';

    // ...
}
```

That's if the client sends us json, but the json is malformed. Now we have a third type of error, which is when you send us bad credentials. So let's add a new constant here called `authentication_error`. And I'm just making up this string, it's not terribly important. And then down here is a map from those types to a human readable text that will live on the `title` key:

```
// src/KnpU/CodeBattle/Api/ApiProblem.php
// ...

class ApiProblem
{
    // ...
    const TYPE_AUTHENTICATION_ERROR = 'authentication_error';

    private static $titles = array(
        // ...
        self::TYPE_AUTHENTICATION_ERROR => 'Invalid or missing authentication',
    );
}
```

The purpose of this is that when we create a new `ApiProblem`, we are forced to pass in a `type` and then that has a nice little map to the title. So given a certain `type`, you always get this nice same identical human readable explanation for it. You don't have to duplicate the titles all around your codebase.

Back in `ApiEntryPoint`, instead of this stuff, you can create a new `ApiProblem` object. Add our `use` statement for that. The status code we know is 401 and the `type` is going to be our new `authentication_error` type:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiEntryPoint.php
// ...

public function start(Request $request, AuthenticationException $authException = null)
{
    $message = $this->getMessage($authException);

    $problem = new ApiProblem(401, ApiProblem::TYPE_AUTHENTICATION_ERROR);
    $problem->set('detail', $message);

    $response = new JsonResponse($problem->toArray(), 401);
    $response->headers->set('Content-Type', 'application/problem+json');

    return $response;
}
```

So it's a nice way to make sure we don't just invent new types all over the place.

And then, we set the `detail`. The `detail` is going to be the message that comes from Silex whenever something goes wrong related to security. Based on what went wrong, we will get a different message here and we can use the translator to control it.

Then down here for the response, we can say just `new JsonResponse`. For the content, we can say `$problem->toArray()`. This is a function we used earlier: it just takes all those properties and turns them into an array. Now we’ll use `$problem->getStatusCode()`. And we’ll keep the response headers already set.

So this is a small improvement. I’m more consistent in my code, so my API will be more consistent too. If I need to create an api problem response, I won’t do it by hand. The `ApiProblem` class does some special things for us, attaching the title and making sure we have a few defined types. If we try this, we should get the same result as before and we do. Perfect.

Chapter 8: Centralizing Error Response Creation

CENTRALIZING ERROR RESPONSE CREATION¶

Let's go further! Even the creation of the response is error prone. So right now, in both the `Application` class where we have our error handler *and* inside of this `ApiEntryPoint`, we create the `JsonResponse` and we set the `Content-Type` header to `application/problem+json` by hand. I don't want to have a lot of these laying around: I want to go through one central spot.

The fix for this has nothing to do with Silex or API's: we're just going to do a bit of refactoring and repeat ourselves a little bit less.

Lets create a new PHP class called `APIProblemResponseFactory` and its job will be to create API problem responses:

```
// src/KnpU/CodeBattle/Api/ApiProblemResponseFactory.php
namespace KnpU\CodeBattle\Api;

class ApiProblemResponseFactory
{
}
```

So we'll create a single function called `createResponse` and it will take in an `ApiProblem` object and create the response for that. And most of this we can just copy from our error handler code:

```
// src/KnpU/CodeBattle/Api/ApiProblemResponseFactory.php
// ...
use Symfony\Component\HttpFoundation\JsonResponse;

class ApiProblemResponseFactory
{
    public function createResponse(ApiProblem $apiProblem)
    {
        $data = $apiProblem->toArray();
        // making type a URL, to a temporarily fake page
        if ($data['type'] != 'about:blank') {
            $data['type'] = 'http://localhost:8000/docs/errors#'.$data['type'];
        }
        $response = new JsonResponse(
            $data,
            $apiProblem->getStatusCode()
        );
        $response->headers->set('Content-Type', 'application/problem+json');

        return $response;
    }
}
```

I'll make sure that I add a couple of `use` statements here. Perfect, it takes in the `ApiProblem`, transforms that into json, and makes sure that the `Content-Type` header is set. So if we can use this instead of repeating that logic elsewhere, it's going to save us some trouble.

Creating an api.response_factory Service¶

Like we saw before, inside of Silex there is a way to create global objects called services. We did this for the `serializer`, which let us use it in multiple places. So I'm going to do the same thing with the `api.response_factory`. And we'll just `return new ApiProblemResponseFactory`. Of course, like anything else don't forget to add the `use` statement for that:

```
// src/KnpU/CodeBattle/Application.php
// ...

use KnpU\CodeBattle\Api\ApiProblemResponseFactory;
// ...

private function configureServices()
{
    // ...

    $this['api.response_factory'] = $this->share(function() {
        return new ApiProblemResponseFactory();
    });
}
```

Yes, this class is getting a little crazy. And that’s it!

Down inside this class we’ll use that key to access the object and make use of it. I have that same `$app` variable, so I can get rid of all this stuff here. Pass the `ApiProblem` object to `createResponse` and there we go!

```
private function configureListeners()
{
    $app = $this;

    $this->error(function(\Exception $e, $statusCode) use ($app) {
        // $apiProblem = ...
        // existing code ...

        /** @var \KnpU\CodeBattle\Api\ApiProblemResponseFactory $factory */
        $factory = $app['api.response_factory'];

        return $factory->createResponse($apiProblem);
    });
}
```

Injecting ApiProblemResponseFactory into ApiEntryPoint¶

We can do the same thing inside the `ApiEntryPoint`. I need to practice a little bit of dependency injection, and if this is a new idea to you or going over your head, we have a free tutorial about dependency injection. I highly recommend you check it out, it’s going to change the way you code.

So in `Application`, I’m going to find the entry point and I’m actually going to go past that new factory object right to it as the second argument to the `__construct` function of our `ApiEntryPoint`:

```
// src/KnpU/CodeBattle/Application.php
// ...

private function configureSecurity()
{
    $app = $this;

    // ...

    $this['security.entry_point.'.$name.'.api_token'] = $app->share(function() use ($app) {
        return new ApiEntryPoint($app['translator'], $app['api.response_factory']);
    });

    // ...
}
```

This means here I will now have a second argument. Don’t forget the `use` statement for that and we’ll just set that on a new property:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiEntryPoint.php
// ...

use KnpU\CodeBattle\Api\ApiProblemResponseFactory;

class ApiEntryPoint implements AuthenticationEntryPointInterface
{
    private $translator;

    private $responseFactory;

    public function __construct(Translator $translator, ApiProblemResponseFactory $responseFactory)
    {
        $this->translator = $translator;
        $this->responseFactory = $responseFactory;
    }

    // ...
}
```

So now, when this object is created we're going to have access to this `ApiProblemResponseFactory`. Down below, we can just use it:

```
// src/KnpU/CodeBattle/Security/Authentication/ApiEntryPoint.php
// ...

class ApiEntryPoint implements AuthenticationEntryPointInterface
{
    private $responseFactory;

    // ...

    public function start(Request $request, AuthenticationException $authException = null)
    {
        $message = $this->getMessage($authException);

        $problem = new ApiProblem(401, ApiProblem::TYPE_AUTHENTICATION_ERROR);
        $problem->set('detail', $message);

        return $this->responseFactory->createResponse($problem);
    }
}
```

So we still create the `ApiProblem` object, but I don't want to do any of this other stuff. And that's it! We just reduced duplication, let's try our tests:

```
php vendor/bin/behat features/api/authentication.feature
```

Those pass too! Let's try all of our tests for the programmer.

```
php vendor/bin/behat features/api/programmer.feature
```

Sahweet! They're passing too! So there's no chance of duplication because everything is going through that same class.

Chapter 9: Creating Token Resources in the API

CREATING TOKEN RESOURCES IN THE API¶

Our API clients can send a token with their request to become authenticated. But how are they supposed to get that token in the first place?

Actually, this is already possible via the web interface. First, let me delete our SQLite database, which will reset our users. Now, we can login as ryan@knplabs.com password `foo`. If you go to the url `/tokens`, you see I have a little interface here. I can add a token, put a message, click *`tokenify-me`* and there we go, I've got a shiny new token!

And this is something we can use right now in an API request to authenticate as the `ryan@knplabs` user. This is great and might actually be enough for you. But for me, I also want a way to do this through my API. I want to make a request through an endpoint that says "give me a new API token."

As always, let's start with creating a new Behat Scenario. In fact this is a whole new feature! Create a new behat feature file and call it `token.feature` Let me get my necktie on here and start describing the business value for this. In this case it's pretty easy: you need to get tokens so you can use our API.

```
# features/api/token.feature
Feature: Token
  In order to access restricted information
  As an API client
  I can create tokens and use them to access information
```

Perfect.

Next, let's put our first scenario here, which is going to be the working scenario for creating a token. Even though a token relates to security it's really no different than any other resource we're creating, like a programmer resource. So the scenario for this is going to look really similar. The only difference is that we can't authenticate with an API token, because that's what we are trying to get. So instead we're going to authenticate with HTTP Basic. First, let's make sure there is a user in the database with a certain password. And just like you saw with the web interface, every token has a note describing how it's being used. So in our request body, we'll send JSON with a little note.

```
# features/api/token.feature
Feature: Token
  # ...

  Scenario: Creating a token
    Given there is a user "weaverryan" with password "test"
    And I have the payload:
      """
      {
        "notes": "A testing token!"
      }
      """
```

Great! To send HTTP basic headers with my request I have a built in line for this. Then after that, we can make a request just like normal:

```
# features/api/token.feature
Feature: Token
  # ...

  Scenario: Creating a token
    Given there is a user "weaverryan" with password "test"
    And I have the payload:
      """
      {
        "notes": "A testing token!"
      }
      """
    And I authenticate with user "weaverryan" and password "test"
    When I request "POST /api/tokens"
```

I'm making up this URL but you can see I'm being consistent because we also have `/api/programmers`.

After this, it's just like our programmer endpoint: we know the status code should be 201, that there should be a `Location` header and we'll expect that the token resource is going to be returned to us and that it will have a key called `token`, which is the newly generated string:

```
# features/api/token.feature
Feature: Token
  # ...

  Scenario: Creating a token
    Given there is a user "weaverryan" with password "test"
    And I have the payload:
      """
      {
        "notes": "A testing token!"
      }
      """
    And I authenticate with user "weaverryan" and password "test"
    When I request "POST /api/tokens"
    Then the response status code should be 201
    And the "Location" header should exist
    And the "token" property should be a string
```

Awesome!

So let's try this: we know it's going to fail but we want to confirm that:

```
php vendor/bin/behat features/api/token.feature
```

Failure, sweet! And it does with a 404 because we don't have an endpoint for this.

Creating the TokenController¶

To get this working, I'm going to create an entirely new controller class and make it look a bit like my `ProgrammerController`. Make it extend the `BaseController` class which we've been adding more helper methods into. Notice that I did just add a `use` statement for that:

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
namespace KnpU\CodeBattle\Controller\Api;

use KnpU\CodeBattle\Controller\BaseController;

class TokenController extends BaseController
{
}
```

And it expects us to have one method called `addRoutes`. This is special to my implementation of Silex, but you'll remember that we have this at the top of `ProgrammerController` and that's just where we build all of our endpoints. We can do the same things here. We'll add a new `POST` endpoint for `/api/tokens` that will execute a method called `newAction` when we hit it:

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

class TokenController extends BaseController
{
    protected function addRoutes(ControllerCollection $controllers)
    {
        $controllers->post('/api/tokens', array($this, 'newAction'));
    }

    public function newAction()
    {
        return 'foo';
    }
}
```

So let's go back and rerun the tests. Look at that, it *is* working. The test still fails, but instead of a 404, we see a 200 status code because we're returning `foo`.

Creating the Token Resource¶

So let's do as little work as possible to get this going. The first thing to know is that we *do* have a token table. And just like with our other tables like the `programmer` table where we have a `Programmer` class, I've also created an `ApiToken` class:

```
// src/KnpU/CodeBattle/Security/Token/ApiToken.php
namespace KnpU\CodeBattle\Security\Token;

use Symfony\Component\Validator\Constraints as Assert;

class ApiToken
{
    public $id;

    public $token;

    /**
     * @Assert\NotBlank(message="Please add some notes about this token")
     */
    public $notes;

    public $userId;

    public $createdAt;

    public function __construct($userId)
    {
        $this->userId = $userId;
        $this->createdAt = new \DateTime();
        $this->token = base_convert(sha1(uniqid(mt_rand(), true)), 16, 36);
    }
}
```

If we can create this new `ApiToken` object, then we can use some ORM-magic I setup to save a new row to that table.

So let’s start doing that: `$token = new ApiToken();`. I’ll add the `use` statement for that:

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

public function newAction()
{
    $token = new ApiToken();
}
```

You can see immediately it’s angry with me because I need to pass the `userId` to the constructor. Now, what is the `id` of the current user? Remember, in our scenario, we’re passing HTTP Basic authentication. So here we need to grab the HTTP Basic username and look that up in the database. I’m not going to worry about checking the password yet, we’ll do that in a second.

In Silex, whenever you need request information you can just type hint a `$request` variable in your controller and it will be passed in. Am I sounding like a broken record yet? Don’t forget your `use` statement!

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

use Symfony\Component\HttpFoundation\Request;
// ...

public function newAction(Request $request)
{
    $token = new ApiToken();
}
```

You may or may not remember this - I had to look it up - but if you want to get the HTTP Basic username that’s sent with the request, you can say `$request->headers->get('PHP_AUTH_USER')`. Oops don’t forget your equals sign. Next I’ll look this up in our user table. For now we’ll just assume it exists: I’m living on the edge by not doing any error handling. And then, we’re going to say `$user->id`:

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

use Symfony\Component\HttpFoundation\Request;
// ...

public function newAction(Request $request)
{
    $username = $request->headers->get('PHP_AUTH_USER');
    $user = $this->getUserRepository()->findUserByUsername($username);

    $token = new ApiToken($user->id);
}
```

Perfect!

Decoding the Request Body¶

Next, we need to set the notes. In our scenario we're sending a JSON body with a notes field. So here, what we can do is just grab that from the request. We did this before in Episode 1: `$request->getContent()` gets us the raw JSON and `json_decode` will return an array. So, we'll get the notes key off of that. And that's really it! All we need to do now is save the token object, which I'll do with my simple ORM system:

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

public function newAction(Request $request)
{
    $username = $request->headers->get('PHP_AUTH_USER');

    $user = $this->getUserRepository()->findUserByUsername($username);

    $data = json_decode($request->getContent(), true);

    $token = new ApiToken($user->id);
    $token->notes = $data['notes'];

    $this->getApiTokenRepository()->save($token);
}
```

Now, we need to return our normal API response. Remember we're using the Serializer at this point and in the last couple of chapters we created a nice new function in our `BaseController` called `createApiResponse`. All we need to do is pass it the object we want to serialize and the status code - 201 here - and that's going to build and return the response for us:

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

public function newAction(Request $request)
{
    // ...

    $this->getApiTokenRepository()->save($token);

    return $this->createApiResponse($token, 201);
}
```

That's as simple as Jean-Luc Picard sending the Enterprise into warp! Engage.

Head over to the terminal:

```
php vendor/bin/behat features/api/token.feature
```

Awesome...ish! So it's failing because we don't have a `Location` header set, but if you look at what's being returned from the endpoint, you can tell it's actually working and inserting this in the database. We're missing the `Location` header and we *should* have it, but for now I'm just going to comment that line out. I don't want to take the time to build the endpoint to view a single token. I'll let you handle that:

```
# features/api/token.feature
# ...

Scenario: Creating a token
  # ...
  When I request "POST /api/tokens"
  Then the response status code should be 201
  # And the "Location" header should exist
  And the "token" property should be a string
```

Let's run the test. Perfect it passes!

Testing for a Bad HTTP Basic Password¶

Since we're not checking to see if the password is valid, let's add another scenario for that. We can copy most of the working scenario but we'll change a couple of things. Instead of the right password we'll send something different. And instead of 201 this time it's going to be a 401:

```
# features/api/token.feature
# ...

Scenario: Creating a token with a bad password
  Given there is a user "weaverryan" with password "test"
  And I have the payload:
   """
   {
    "notes": "A testing token!"
   }
   """
  And I authenticate with user "weaverryan" and password "WRONG"
  When I request "POST /api/tokens"
  Then the response status code should be 401
```

Remember whenever we have an error response, we are always returning that API Problem format. Great! So let's run just this one scenario which starts on line 21. And again, we're expecting it to fail, but I like to see my failures before I actually do the code:

```
php vendor/bin/behat features/api/token.feature:21
```

Yes, failing!

Activating Silex's HTTP Basic Authentication¶

In our controller we need to check to see if the password is correct for the user. But hey, let's not do that, Silex can help us with some of this straightforward logic. In my main `Application` class, where I configure my security, I've already setup things to allow http basic to happen. By adding this little key here, when the http basic username and password come into the request, the Silex security system will automatically look up the user object and deny access if they have the wrong password:

```
// src/KnpU/CodeBattle/Application.php
// ...

private function configureSecurity()
{
    $app = $this;

    $this->register(new SecurityServiceProvider(), array(
        'security.firewalls' => array(
            'api' => array(
                // ...

                // add this line to the bottom of the array
                'http' => true,
            ),
            // ...
        )
    ));
    // ...
}
```

It’s kind of like our API token system: but instead of sending a token it’s going to be reading it off of the HTTP Basic username and password headers. That’s pretty handy.

That means that in the controller, if we need the actual user object we don’t need to query for it - the security system already did that for us. We can just say `$this->getLoggedInUser()` :

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

public function newAction(Request $request)
{
    // ...

    $token = new ApiToken($this->getLoggedInUser()->id);
    // ...
}
```

We don’t really know if the user logged in via HTTP basic or passed a token, and frankly we don’t care. And since we need our user to be logged in for this endpoint, we can use our nice `$this->enforceUserSecurity()` function:

```
// src/KnpU/CodeBattle/Controller/Api/TokenController.php
// ...

public function newAction(Request $request)
{
    $this->enforceUserSecurity();

    // ...
}
```

Perfect, let’s try that out. And it passes with almost no effort!

Chapter 10: Reuse and Consistency

Now I want to do a couple of other cleanup things. First: whenever we need to read information off of the request that the client is sending us, we're going to use this same json_decode for the content:

```
31 lines | src/KnpU/CodeBattle/Controller/Api/TokenController.php
    ... lines 1 - 16
    public function newAction(Request $request)
    ... lines 18 - 20
        $data = json_decode($request->getContent(), true);
    ... lines 22 - 28
    }
    ... lines 30 - 31
```

In fact, we use this in ProgrammerController inside of the handleRequest method.:

```
168 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
    ... lines 1 - 123
    private function handleRequest(Request $request, Programmer $programmer)
    {
        $data = json_decode($request->getContent(), true);
    ... lines 127 - 154
    }
    ... lines 156 - 168
```

So we have the same logic duplicated there. Let's centralize this by putting it into our BaseController. Open this up and create a new protected function at the bottom called decodeRequestBodyIntoParameters. I know, really short name. And we'll take in a Request object as an argument. And at first this is going to be really simple. We can go to the TokenController, grab this json_decode line, go back to BaseController and return it:

```
290 lines | src/KnpU/CodeBattle/Controller/BaseController.php
    ... lines 1 - 269
    protected function decodeRequestBodyIntoParameters(Request $request)
    {
    ... lines 272 - 275
        $data = json_decode($request->getContent(), true);

        if ($data === null) {
            $problem = new ApiProblem(
                400,
                ApiProblem::TYPE_INVALID_REQUEST_BODY_FORMAT
            );
            throw new ApiProblemException($problem);
        }
    ... lines 285 - 287
    }
    ... lines 289 - 290
```

So now in the TokenController, $data = $this->decodeRequestBodyIntoParameters($request) and we've got it.

So just to be sure, let's go back and run our entire feature:

```
php vendor/bin/behat features/api/token.feature
```

Consistently Erroring on Invalid JSON

Everything still passes so we're good. So, why did we do this? Back in ProgrammerController, when we decoded the body in handleRequest, we also checked to see if maybe the json that was sent to us had a bad format. If the JSON *is* bad, then json_decode is going to return null which is what we're checking for here:

```
168 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 123
    private function handleRequest(Request $request, Programmer $programmer)
    {
        $data = json_decode($request->getContent(), true);
        $isNew = !$programmer->id;

        if ($data === null) {
            $problem = new ApiProblem(
                400,
                ApiProblem::TYPE_INVALID_REQUEST_BODY_FORMAT
            );
            throw new ApiProblemException($problem);
        }
... lines 136 - 154
    }
... lines 156 - 168
```

So let's move that into our new BaseController method, because that is a really nice check. And then it's creating an ApiProblem and throwing an ApiProblemException so we can have that really nice consistent format. We just need to add the use statements for both of these:

```
290 lines | src/KnpU/CodeBattle/Controller/BaseController.php
... lines 1 - 269
    protected function decodeRequestBodyIntoParameters(Request $request)
    {
... lines 272 - 275
            $data = json_decode($request->getContent(), true);

            if ($data === null) {
                $problem = new ApiProblem(
                    400,
                    ApiProblem::TYPE_INVALID_REQUEST_BODY_FORMAT
                );
                throw new ApiProblemException($problem);
            }
... lines 285 - 287
    }
... lines 289 - 290
```

Perfect. Let's rerun those again to make sure things are happy ... and they are!

Don't Blow up on an Empty Request Body!

One other little detail here is that if the request body is blank this is going to blow up with an invalid request body format because json_decode is going to return null. Now technically sending a blank request is not invalid json so I don't want to blow up in that way. This doesn't affect anything now but it's planning for the future. So if !$request->getContent(), then just set $data to an array. Else, we'll do all of our logic down here that actually decodes the json:

```
290 lines | src/KnpU/CodeBattle/Controller/BaseController.php
... lines 1 - 269
    protected function decodeRequestBodyIntoParameters(Request $request)
    {
        // allow for a possibly empty body
        if (!$request->getContent()) {
            $data = array();
        } else {
            $data = json_decode($request->getContent(), true);

            if ($data === null) {
                $problem = new ApiProblem(
                    400,
                    ApiProblem::TYPE_INVALID_REQUEST_BODY_FORMAT
                );
                throw new ApiProblemException($problem);
            }
        }

... line 287
    }
... lines 289 - 290
```

And just to make sure we didn't screw anything up, we'll rerun the tests.

```
php vendor/bin/behat features/api/token.feature
```

A ParameterBag Makes Life Nicer

One last little thing that is going to make our code even easier to deal with. Back in TokenController, because the decodeRequestBodyIntoParameters returns an array, we need to code a bit more defensively here. What if they don't actually send a notes key, we don't want some sort of PHP error:

```
31 lines | src/KnpU/CodeBattle/Controller/Api/TokenController.php
... lines 1 - 16
    public function newAction(Request $request)
    {
... lines 19 - 20
        $data = json_decode($request->getContent(), true);
... lines 22 - 23
        $token->notes = $data['notes'];
... lines 25 - 28
    }
... lines 30 - 31
```

The horror!

And that's not that big of a deal but it's kind of annoying and error prone. So instead, in our new function I want to return a different type of object called a ParameterBag:

```
290 lines | src/KnpU/CodeBattle/Controller/BaseController.php
... lines 1 - 269
    protected function decodeRequestBodyIntoParameters(Request $request)
    {
        // allow for a possibly empty body
        if (!$request->getContent()) {
            $data = array();
        } else {
            $data = json_decode($request->getContent(), true);
... lines 277 - 284
        }
... line 286
        return new ParameterBag($data);
    }
... lines 289 - 290
```

This comes from a component inside of Symfony that Silex uses. It's an object but it acts just like an array with some extra nice methods. Let me show you what I mean, back in TokenController instead of using it like an array we can now say $data->get() and if that key doesn't exist it's not going to throw some bad index warning. We can also use the second argument as the default value:

```
31 lines | src/KnpU/CodeBattle/Controller/Api/TokenController.php
... lines 1 - 16
    public function newAction(Request $request)
    {
... lines 19 - 20
        $data = $this->decodeRequestBodyIntoParameters($request);
... line 22
        $token = new ApiToken($this->getLoggedInUser()->id);
        $token->notes = $data->get('notes');
... lines 25 - 28
    }
... lines 30 - 31
```

Nice, so once again let's rerun the tests and everything's happy!

Use decodeRequestBodyIntoParameters in all the Places

We have this really nice new function inside of our BaseController and I also want to take advantage of it inside of our ProgrammerController. We're going down to handleRequest and now we can just say $data = $this->decodeRequestBodyIntoParameters() and pass the $request object:

```
159 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 123
    private function handleRequest(Request $request, Programmer $programmer)
    {
        $data = $this->decodeRequestBodyIntoParameters($request);
... lines 127 - 145
    }
... lines 147 - 159
```

Next, this big if block is no longer needed. And now because data is an object instead of an array, we need to update our two or three usages of it down here, which is going to make things simpler. So instead of using the isset function, we can say if(!$data->has($property)) because that's one of the methods on that ParameterBag object. And down here instead of having to code defensively using isset, we can just say $data->get($property). In fact let's just do this all in one line:

```
159 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 123
    private function handleRequest(Request $request, Programmer $programmer)
    {
        $data = $this->decodeRequestBodyIntoParameters($request);
... lines 127 - 134
        // update the properties
        foreach ($apiProperties as $property) {
            // if a property is missing on PATCH, that's ok - just skip it
            if (!$data->has($property) && $request->isMethod('PATCH')) {
                continue;
            }

            $programmer->$property = $data->get($property);
        }
... lines 144 - 145
    }
... lines 147 - 159
```

Lovely! Now that was a fairly fundamental change so there is a good chance that we broke something. So let's go back and run our entire programmer feature:

```
php vendor/bin/behat features/api/programmer.feature
```

Beautiful! We didn't actually break anything which is so good to know.

Chapter 11: Validate that Token Resource

It's finally that time: to add a little bit of validation to our token.

How We Validated Programmer

This should start to feel easy because we did all of this before with our programmer. Now let me remind you, in the PogrammerController, to validate things we call this validate() function, which is something I created for this project before we even started:

159 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 36
    public function newAction(Request $request)
... lines 38 - 43
        if ($errors = $this->validate($programmer)) {
            $this->throwApiProblemValidationException($errors);
        }
... lines 47 - 56
    }
... lines 58 - 159
```

But the way the validation works is that on our Programmer class we have these @Assert things:

52 lines | src/KnpU/CodeBattle/Model/Programmer.php

```
... lines 1 - 5
use Symfony\Component\Validator\Constraints as Assert;
... lines 7 - 11
class Programmer
... lines 13 - 17
    /**
     * @Assert\NotBlank(message="Please enter a clever nickname")
     * @Serializer\Expose
     */
    public $nickname;
... lines 23 - 50
}
```

So when we call the validate() function on our controller, it reads this and makes sure the nickname isn't blank. It's as simple as that!

Now if that validate() function returns an array that has at least one error in it, then we'll call this throwApiProblemValidationException function, which is something that we created inside this controller. You can see it further down inside this same file:

159 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 147
    private function throwApiProblemValidationException(array $errors)
    {
        $apiProblem = new ApiProblem(
            400,
            ApiProblem::TYPE_VALIDATION_ERROR
        );
        $apiProblem->set('errors', $errors);

        throw new ApiProblemException($apiProblem);
    }
... lines 158 - 159
```

What does it do? No surprises, it creates a new ApiProblem object, sets the errors as a nice property on it then throws a new ApiProblemException. We can see this if we look inside the programmer.feature class. I'll search for 400 because our validation errors return a 400 status code:

130 lines | features/api/programmer.feature

```
... lines 1 - 24
  Scenario: Validation errors
... lines 26 - 33
    Then the response status code should be 400
... lines 35 - 40
    And the "errors.nickname" property should exist
    But the "errors.avatarNumber" property should not exist
    And the "Content-Type" header should be "application/problem+json"
... lines 44 - 130
```

You can see this is an example of us testing our validation situation. We're checking to see that there are nickname and avatarNumber properties on errors.

Validating ApiToken

The ApiToken class also has one of these not blank things on it:

34 lines | src/KnpU/CodeBattle/Security/Token/ApiToken.php

```
... lines 1 - 4
use Symfony\Component\Validator\Constraints as Assert;

class ApiToken
{
... lines 9 - 14
    /**
     * @Assert\NotBlank(message="Please add some notes about this token")
     */
    public $notes;
... lines 19 - 32
}
```

So, all we need to do in our controller is call these same methods. First, let's move that throwApiProblemValidationException into our BaseController, because that's going to be really handy. And of course we'll make it protected so we can use it in the sub classes:

301 lines | src/KnpU/CodeBattle/Controller/BaseController.php

```
... lines 1 - 289
    protected function throwApiProblemValidationException(array $errors)
    {
        $apiProblem = new ApiProblem(
            400,
            ApiProblem::TYPE_VALIDATION_ERROR
        );
        $apiProblem->set('errors', $errors);

        throw new ApiProblemException($apiProblem);
    }
... lines 300 - 301
```

Perfect!

Next, let's steal a little bit of code from our ProgrammerController and put that into our TokenController. So once we're done updating our token object, we'll just call the same function, pass it the token instead of the programmer and throw that same error:

36 lines | src/KnpU/CodeBattle/Controller/Api/TokenController.php

```
... lines 1 - 16
    public function newAction(Request $request)
... lines 18 - 25
        $errors = $this->validate($token);
        if ($errors) {
            $this->throwApiProblemValidationException($errors);
        }
... lines 30 - 33
    }
... lines 35 - 36
```

Great, so this actually should all be setup. Of course what I forgot to do was write the scenario first, shame on me! Let's write the scenario to make sure this is in full operating order.

I'll copy most of the working version. Here, we won't pass any request body. Fortunately we've made our decode function able to handle that. We know the status code is going to be 400. We can check to see that the errors.notes property will equal the message that is on the ApiToken class. It will be this message right here:

38 lines | features/api/token.feature

```
... lines 1 - 31
  Scenario: Creating a token without a note
    Given there is a user "weaverryan" with password "test"
    And I authenticate with user "weaverryan" and password "test"
    When I request "POST /api/tokens"
    Then the response status code should be 400
    And the "errors.notes" property should equal "Please add some notes about this token"
```

Alright!

This starts on line 33, so let's run just this scenario:

```
php vendor/bin/behat features/api/token.feature:33
```

Oh no, and it actually passes! Instead of the 400 we want, it is giving us the 201, which means that things are not failing validation. You can see for the note it says default note. If you look back in our TokenController... Ah ha! It's because I forgot to take off the default value. So now it's either going to be set to whatever the note is or null:

```
36 lines | src/KnpU/CodeBattle/Controller/Api/TokenController.php
... lines 1 - 16
    public function newAction(Request $request)
    {
... lines 19 - 20
        $data = $this->decodeRequestBodyIntoParameters($request);
... lines 22 - 23
        $token->notes = $data->get('notes');
... lines 25 - 33
    }
... lines 35 - 36
```

And if it's set to null we should see our validation kick in. And we do, perfect!

Chapter 12: New Battle Resource (the Scenario)

Let's login to the site. Don't forget, our tests like to mess with our database, so I'm going to delete the SQLite database file and it'll regenerate with some nice test data:

```
rm data/code_battles.sqlite
```

We'll login as ryan@knplabs.com password foo.

We already know that I'm able to create a programmer. And we even have some really nice API endpoints for this. The other part of the site is all about battles. If I click "Start Battle", this is a list of projects that are in the database right now. If I dare to select one of those project, it starts an EPIC CODE *Battle* OF HISTORY between the programmer and the project and picks a winner.

A battle is another type of resource, but it can't be created yet in the API. Let's fix that!

New Battle Feature

Like my other resources, I already have a class that models this. You can see there's a programmer, a project, the outcome didProgrammerWin and it even stores the date it was fought and some extra notes:

26 lines | src/KnpU/CodeBattle/Model/Battle.php

```php
<?php

namespace KnpU\CodeBattle\Model;

class Battle
{
    /* All public properties are persisted */
    public $id;

    /**
     * @var Programmer
     */
    public $programmer;

    /**
     * @var Project
     */
    public $project;

    public $didProgrammerWin;

    public $foughtAt;

    public $notes;
}
```

Let's make the endpoint to create new battles. We're going to start like always by creating a new feature - battle.feature. The API clients are going to want to create battles to see if their programmers can take on and defeat these projects. After the business value, the next line is the person that's benefiting from the new feature and finally we have a little description:

```
25 lines | features/api/battle.feature
Feature:
  In order to prove my programmers' worth against projects
  As an API client
  I need to be able to create and view battles
... lines 5 - 25
```

Create Battle Scenario

Let's add the first Scenario: Creating a new Battle. If we go back to programmer.feature, we can copy a lot of this. First, in order to create a battle, we're probably going to need to be authenticated. So, I'll copy this background:

```
25 lines | features/api/battle.feature
Feature:
  In order to prove my programmers' worth against projects
... lines 3 - 5
  Background:
    Given the user "weaverryan" exists
    And "weaverryan" has an authentication token "ABCD123"
    And I set the "Authorization" header to be "token ABCD123"
... line 10
  Scenario: Create a battle
... lines 12 - 25
```

I'm going to go back and copy the entire scenario for creating a programmer. After all, this is an API, so creating a resource should always look pretty much the same.

Let's work from the end backwards and think about how we want the response to look. We know there's going to be a Location header, because there's always a Location header after creating a resource. But we don't know what URL that's going to be yet, because we don't have an endpoint yet for viewing a single battle. So we'll just say that the Location header should exist. And if you look at the Battle class, you'll see there's a didProgrammerWin property. Let's just make sure that exists as well - we don't know if it's going to be true or false, because there's some randomness. Let's update the URL to /api/battles and the status code of 201 looks perfect:

```
25 lines | features/api/battle.feature
... lines 1 - 10
  Scenario: Create a battle
... lines 12 - 20
    When I request "POST /api/battles"
    Then the response status code should be 201
    And the "Location" header should exist
    And the "didProgrammerWin" property should exist
```

Creating a Programmer and Project First

In order to create a Battle - we'll need to send a programmer and a project. And probably the way we'll want the client to do that is by sending the programmer and the project's ids. So let's send programmerId and projectId - but we don't know yet what these should be set to.

Next, in order for us to start a battle, there needs to already be a programmer and a project sitting in the database. So before this line, we'll need to say Given there is a programmer called, and we'll create a new programmer called Fred. Again, these are all built-in Behat definitions that I created before we started working and they all live in either ApiFeatureContext or ProjectContext. If you want to know what I'm doing behind the scenes, just open up those classes. There's another one for And there is a project called, and we'll say "my_project":

```
25 lines | features/api/battle.feature
... lines 1 - 10
  Scenario: Create a battle
    Given there is a project called "my_project"
    And there is a programmer called "Fred"
... lines 14 - 25
```

A little problem still exists: we don't know what the id's are of the programmer and project we just created. So I don't know what to put in the request body - we really want whatever ids those new things have. This is a really difficult problem with testing API's. So one of the things I've put into my testing system already, is the ability to do things like this:

```
25 lines | features/api/battle.feature
... lines 1 - 10
  Scenario: Create a battle
    Given there is a project called "my_project"
    And there is a programmer called "Fred"
    And I have the payload:
      """
      {
        "programmerId": "%programmers.Fred.id%",
        "projectId": "%projects.my_project.id%"
      }
      """
... lines 21 - 25
```

It's a special syntax. And what this will do is go find a programmer whose nickname is "Fred" and give us its id. It'll create a query for that dynamically. This syntax is totally special - it's not something built into Behat. If you want to know how it works, open ApiFeatureContext, scroll all the way to the bottom and find the processReplacements() function. It parses out that "%" syntax, looks for these wildcards, and lets us do some of that magic. This will be really handy, and we'll use it a few more times.

We'll do the same thing for projects. This looks great!

```
25 lines | features/api/battle.feature
... lines 1 - 10
  Scenario: Create a battle
    Given there is a project called "my_project"
    And there is a programmer called "Fred"
    And I have the payload:
      """
      {
        "programmerId": "%programmers.Fred.id%",
        "projectId": "%projects.my_project.id%"
      }
      """
    When I request "POST /api/battles"
    Then the response status code should be 201
    And the "Location" header should exist
    And the "didProgrammerWin" property should exist
```

You know I like watching my tests fail first, so let's try it out. We'll just run this new battle.feature file:

```
php vendor/bin/behat features/api/battle.feature
```

Instead of 201, we get the 404 because the endpoint doesn't exist. That's awesome. Now let's make this work!

Chapter 13: Start (Create) an Epic Battle (Resource)

What's cool is that because we've done so much work up to this point, coding up our API is going to get really easy. I'll copy TokenController - I like having one resource per Controller. Update the class name. And this already has some code we want. So let's change the URL to /api/battles:

20 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```php
<?php

namespace KnpU\CodeBattle\Controller\Api;

use KnpU\CodeBattle\Controller\BaseController;
use Silex\ControllerCollection;

class BattleController extends BaseController
{
    protected function addRoutes(ControllerCollection $controllers)
    {
        $controllers->post('/api/battles', array($this, 'newAction'));
    }
... lines 14 - 18
}
```

In newAction, we're going to reuse a lot of this. To create a Battle, you *do* need to be logged in, so we'll keep the enforceUserSecurity(). This decodeRequestBodyIntoParameters() is what goes out and reads the JSON on the request and gives us back this array-like object called a ParameterBag. So that's all good too.

I *am* going to remove most of the rest of this, because it's specific to creating a token.

Finding the Battle and Project

What we need to do is read the programmer and project id's off of the request and then create and save a new Battle object based off of those. So first, let's go get the projectId. We're able to use this nice get function, because the decodeRequestBodyIntoParameters function gives us that ParametersBag object. Let's also get the programmerId.

36 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```php
... lines 1 - 15
    public function newAction(Request $request)
    {
        $this->enforceUserSecurity();

        $data = $this->decodeRequestBodyIntoParameters($request);

        $programmerId = $data->get('programmerId');
        $projectId = $data->get('projectId');
... lines 24 - 33
    }
... lines 35 - 36
```

Perfect!

And with these 2 things, I need to query for the Project and Programmer objects, because the way I'm going to create the Battle will need them, not just their ids. Plus, this will tell us if these ids are even real. I'll use one of my shortcuts to query for the Project:

```
36 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php
... lines 1 - 15
    public function newAction(Request $request)
    {
... lines 18 - 22
        $projectId = $data->get('projectId');
... lines 24 - 25
        $project = $this->getProjectRepository()->find($projectId);
... lines 27 - 33
    }
... lines 35 - 36
```

All this is doing is going out and finding the Project with that id and returning the Project model object that has the data from that row. We'll do the same thing with Programmer:

```
36 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php
... lines 1 - 15
    public function newAction(Request $request)
    {
... lines 18 - 21
        $programmerId = $data->get('programmerId');
... lines 23 - 24
        $programmer = $this->getProgrammerRepository()->find($programmerId);
... lines 26 - 33
    }
... lines 35 - 36
```

The BattleManager

Normally, to create a battle, you'd expect me to instantiate it manually. We've done that before for Tokens and Programmers. But for Battles, I have a helper called BattleManager that will do this for us. So instead of creating the Battle by hand, we'll call this battle() function and pass it the Programmer and Project:

```
58 lines | src/KnpU/CodeBattle/Battle/BattleManager.php
... lines 1 - 4
use KnpU\CodeBattle\Model\Battle;
use KnpU\CodeBattle\Model\Programmer;
use KnpU\CodeBattle\Model\Project;
... lines 8 - 10
class BattleManager
{
... lines 13 - 29
    public function battle(Programmer $programmer, Project $project)
    {
        $battle = new Battle();
        $battle->programmer = $programmer;
        $battle->project = $project;
        $battle->foughtAt = new \DateTime();

        if ($programmer->powerLevel < $project->difficultyLevel) {
            // not enough energy
            $battle->didProgrammerWin = false;
            $battle->notes = 'You don\'t have the skills to even start this project. Read the documentation (i.e. power up) and try again!';
        } else {
            if (rand(0, 2) != 2) {
                $battle->didProgrammerWin = true;
                $battle->notes = 'You battled heroically, asked great questions, worked pragmatically and finished on time. You\'re a hero!';
            } else {
                $battle->didProgrammerWin = false;
                $battle->notes = 'Requirements kept changing, too many meetings, project failed :(';
            }

            $programmer->powerLevel = $programmer->powerLevel - $project->difficultyLevel;
        }

        $this->battleRepository->save($battle);
        $this->programmerRepository->save($programmer);

        return $battle;
    }
```

It takes care of all of the details of creating the Battle, figuring out who won, setting the foughtAt time, adding some notes and saving all of this. So we *do* need to create a Battle object, but this will do it for us.

Creating the Battle

Back in BattleController, I already have a shortcut method setup to give us the BattleManager object. Then we'll use the battle() function we just saw and pass it the Programmer and Project:

36 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```
... lines 1 - 15
    public function newAction(Request $request)
    {
... lines 18 - 24
        $programmer = $this->getProgrammerRepository()->find($programmerId);
        $project = $this->getProjectRepository()->find($projectId);

        $battle = $this->getBattleManager()->battle($programmer, $project);
... lines 29 - 33
    }
... lines 35 - 36
```

And that's it - the Battle is created and saved for us. Now all we need to do is pass the Battle to the createApiResponse() method. And that will take care of the rest:

36 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```
... lines 1 - 15
    public function newAction(Request $request)
    {
... lines 18 - 27
        $battle = $this->getBattleManager()->battle($programmer, $project);

        $response = $this->createApiResponse($battle, 201);
... lines 31 - 32
        return $response;
    }
... lines 35 - 36
```

The createApiResponse method uses the serializer object to turn the Battle object into JSON. We haven't done any configuration on this class for the serializer, which means that it's serializing all of the fields. And for now, I'm happy with that - we're getting free functionality.

This looks good to me - so let's try it!

```
php vendor/bin/behat features/api/battle.feature
```

Oh! It *almost* passes. It gets the 201 status code, but it's missing the Location header. In the response, we can see the created Battle, with notes on why our programmer lost.

Adding the Location Header

Back in newAction, we can just set createApiResponse to a variable and then call $response->headers->set() and pass it Location add a temporary todo:

36 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```
... lines 1 - 15
    public function newAction(Request $request)
    {
... lines 18 - 29
        $response = $this->createApiResponse($battle, 201);
        $response->headers->set('Location', 'TODO');

        return $response;
    }
... lines 35 - 36
```

Remember, this is the location to view a single battle, and we don't have an endpoint for that yet. But this will get our tests to pass for now:

```
php vendor/bin/behat features/api/battle.feature
```

Perfect!

Battle Validation

So let's add some validation. Since the battle() function is doing all of the work of creating the Battle, we don't need to worry about it too much. We just need to make sure that projectId and programmerId are valid. I'll just do validation manually here by creating an $errors variable and then check to see if we didn't find a Project in the database for some reason. If that's the case, let's add an error with a nice message. And we'll do the same thing with the Programmer:

50 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```
... lines 1 - 15
    public function newAction(Request $request)
    {
... lines 18 - 24
        $programmer = $this->getProgrammerRepository()->find($programmerId);
        $project = $this->getProjectRepository()->find($projectId);

        $errors = array();
        if (!$project) {
            $errors['projectId'] = 'Invalid or missing projectId';
        }
        if (!$programmer) {
            $errors['programmerId'] = 'Invalid or missing programmerId';
        }
        if ($errors) {
            $this->throwApiProblemValidationException($errors);
        }
... lines 38 - 47
    }
... lines 49 - 50
```

Finally at the bottom, if we actually have at least one thing in the $errors variable, we're going to call a nice method we made in a previous chapter called throwApiProblemValidationException and just pass it the array of errors. It's just that easy.

We don't have a scenario setup for this, so let's tweak ours temporarily to try it - foobar is definitely not a valid id:

25 lines | features/api/battle.feature

```
... lines 1 - 10
  Scenario: Create a battle
    Given there is a project called "my_project"
    And there is a programmer called "Fred"
    And I have the payload:
      """
      {
        "programmerId": "foobar",
        "projectId": "%projects.my_project.id%"
      }
      """
... lines 21 - 25
```

```
php vendor/bin/behat features/api/battle.feature:11
```

Now, we can see that the response code is 400 and we have this beautifully structured error response. So that's why we went to *all* of that work of setting up our error handling correctly, because the rest of our API is so easy and so consistent.

Let's change the scenario back and re-run the tests to make sure we haven't broken anything:

```
php vendor/bin/behat features/api/battle.feature:11
```

Perfect! This is a really nice endpoint for creating a battle.

Chapter 14: GET Your (One) Battle On

Next, let's keep going with viewing a single battle. Scenario: GETting a single battle. And thinking about this, we're going to need to make sure that there's a battle in the database first. I'm going to use similar language as before to create a Fred programmer and a project called project_facebook. I also have another step that allows me to say And there has been a battle between "Fred" and "project_facebook":

34 lines | features/api/battle.feature

```
... lines 1 - 25
Scenario: GET one battle
  Given there is a project called "projectA"
  And there is a programmer called "Fred"
  And there has been a battle between "Fred" and "projectA"
... lines 30 - 34
```

By the way, the nice auto-completion I'm getting is from the new PHPStorm 8 version, which has integration with Behat. I highly recommend it.

Great, so this makes sure there's something in the database. Next, we'll make the GET request to /api/battles/something. Here's the problem: the only way we can really identify our Battles are by their id. They're not like Programmer, where each has a unique nickname that we can use.

The Special %battles.last.id% Syntax

Here, we know there's a Battle in the database, but just like before when we were building the request body, we have no idea what that id was going to be. Fortunately, we can use that same magic % syntax. This time we can say %battles.last.id%:

34 lines | features/api/battle.feature

```
... lines 1 - 25
Scenario: GET one battle
  Given there is a project called "projectA"
  And there is a programmer called "Fred"
  And there has been a battle between "Fred" and "projectA"
  When I request "GET /api/battles/%battles.last.id%"
... lines 31 - 34
```

Before, we used this syntax to query for a Programmer by its nickname. But it also has a special "last" keyword, which queries for the last record in the table. Again, this is *me* adding special things to *my* Behat project. Which is really handy for testing the API.

Next, go to programmer.feature and find its "GET one programmer". We'll copy the endpoint and "Then" lines and do something similar. The status code looks good. The Battle has a didProgrammerWin field and we'll also make sure that the notes field is returned in the response:

34 lines | features/api/battle.feature

```
... lines 1 - 25
Scenario: GET one battle
  Given there is a project called "projectA"
  And there is a programmer called "Fred"
  And there has been a battle between "Fred" and "projectA"
  When I request "GET /api/battles/%battles.last.id%"
  Then the response status code should be 200
  And the "notes" property should exist
  And the "didProgrammerWin" property should exist
```

You guys know the drill. We're going to try this first to make sure it fails. This is on line 26, so we'll add :26 to only run this scenario:

```
php vendor/bin/behat features/api/battle.feature:26
```

And there we go - we get the 404 instead of the 200 and that's perfect.

Creating the GET Endpoint

Let's get this working! In BattleController, add a new GET endpoint for /api/battles/{id} and change the method to showAction. Because we have a {id} in the path, the showAction will have an $id argument.

63 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```
... lines 1 - 10
    protected function addRoutes(ControllerCollection $controllers)
    {
... line 13
        $controllers->get('/api/battles/{id}', array($this, 'showAction'));
    }
... lines 16 - 50
    public function showAction($id)
    {
... lines 53 - 60
    }
... lines 62 - 63
```

From here, life is really familiar. First, do we need security? - always ask yourself that. I'm going to decide that anyone can fetch battle details out without being authenticated. So we won't add any protection.

We *will* need to go and query for the Battle object that represents the given id. We always want to check if that matches anything, and if it doesn't, we want to return a really nice 404 response. In episode 1, we did that by using a function called throw404. That's going to throw a special exception, which is mapped to a 404 status code, and because we have our nice error handling, the ApiProblem response format will be returned:

63 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```
... lines 1 - 50
    public function showAction($id)
    {
        $battle = $this->getBattleRepository()->find($id);
        if (!$battle) {
            $this->throw404('No battle with id '.$id);
        }
... lines 57 - 60
    }
... lines 62 - 63
```

We have the object and we know we want to serialize it to get that consistent response. Once again, this is really easy, because we can just re-use the createApiResponse method, and that's going to do all the work for us. We don't need the 2nd argument, because that defaults to 200 already:

63 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```php
// ... lines 1 - 50
    public function showAction($id)
    {
        $battle = $this->getBattleRepository()->find($id);
        if (!$battle) {
            $this->throw404('No battle with id '.$id);
        }

        $response = $this->createApiResponse($battle, 200);

        return $response;
    }
// ... lines 62 - 63
```

That's it guys - let's run the test:

```
php vendor/bin/behat features/api/battle.feature:26
```

Wow, and it already passes. This is getting *really really* easy, which is why we put in all the work before this.

Don't Forget to Fix the Location Header

Now that we have a proper showAction(), we can go back and fix the "todo" in the header. First, we'll need to give the route an internal name - api_battle_show:

65 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```php
// ... lines 1 - 10
    protected function addRoutes(ControllerCollection $controllers)
    {
// ... line 13
        $controllers->get('/api/battles/{id}', array($this, 'showAction'))
            ->bind('api_battle_show');
    }
// ... lines 17 - 65
```

In newAction(), we'll use generateUrl() to make the URL for us:

65 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```php
// ... lines 1 - 17
    public function newAction(Request $request)
    {
// ... lines 20 - 46
        $url = $this->generateUrl('api_battle_show', array('id' => $battle->id));
        $response->headers->set('Location', $url);

        return $response;
    }
// ... lines 52 - 65
```

Again, these shortcuts are things I added to *my* project, but this is just using the standard method in Silex to generate the URL based on the name of the route. And you can see what all of the shortcut methods really do by opening up the BaseController class.

First, let's make sure we didn't break anything by re-running the entire feature.

```
php vendor/bin/behat features/api/battle.feature
```

Green green green!

Chapter 15: Battles and Programmer - Link them!

Let's do something really interesting. First I want to see what the Battle response looks like - so I'll say "And print last response":

```
35 lines | features/api/battle.feature
... lines 1 - 25
  Scenario: GET one battle
... lines 27 - 29
    When I request "GET /api/battles/%battles.last.id%"
... lines 31 - 33
    And print last response
```

Now, run Behat.

```
php vendor/bin/behat features/api/battle.feature
```

At the bottom, you can see that response has programmer and project information right inside of it. That's because the Battle class has two properties that hold these objects. The serializer sees these objects, and serializes them recursively.

Controlling (Removing) Embedded Resources

In a couple of chapters, we're going to talk about embedding resources where we do this on purpose. But for now, I want to avoid it: if I'm getting a Battle, I only want to retrieve that Battle. So like before, we need to take control of how this is serialized.

I'll copy the Serializer use statement from Programmer into Battle. Next, let's also copy the ExclusionPolicy annotation into Battle, which tells the serializer to *only* serialize properties that we explicitly expose with the @Serializer\Expose annotation. Which properties you want to expose is totally up to you. I'll do it with the $id, of course $didProgrammerWin, $foughtAt and we'll also expose the $notes property:

```
42 lines | src/KnpU/CodeBattle/Model/Battle.php
... lines 1 - 4
use JMS\Serializer\Annotation as Serializer;

/**
 * @Serializer\ExclusionPolicy("all")
 */
class Battle
{
    /**
     * @Serializer\Expose()
     */
    public $id;
... lines 16 - 26
    /**
     * @Serializer\Expose()
     */
    public $didProgrammerWin;

    /**
     * @Serializer\Expose()
     */
    public $foughtAt;

    /**
     * @Serializer\Expose()
     */
    public $notes;
}
```

You guys know the drill - let's run just line 26 to make sure things still pass:

```
php vendor/bin/behat features/api/battle.feature:26
```

We're still printing out the last response, but nothing is broken, so that's good. You can see that it's in fact not printing out the programmer or project anymore.

Adding a Link to the JSON Response

But now that we did that, if we think about somebody who *is* retrieving a single battle, they might want to know who the programmer was. I can hear our user now:

```
"ok, I see this battle, but what programmer fought in it and how can
get more information about them?"
```

So what I'll do is add a line to the scenario and look for a new, invented field:
And the "programmerUri" field should equal "/api/programmers/Fred":

```
35 lines | features/api/battle.feature
... lines 1 - 25
26  Scenario: GET one battle
... lines 27 - 33
34    And the "programmerUri" property should equal "/api/programmers/Fred"
```

So we're saying that the response will have this extra field that's not *really* on the Battle class. What's cool about this is that as an API client, I'll see this and say:

```
"Oh, Fred was the programmer, and I can just go to that URL to get his details".
```

First, let's run this and watch it fail:

```
php vendor/bin/behat features/api/battle.feature:26
```

Using a VirtualProperty

So how can we add this? The problem is that there really *is no* programmerUri property on Battle. So one of the cool features from JMS serializer is the ability to have virtual properties.

Create a new function called getProgrammerUri - the name of the method is important - and for right now, I'm just going to hardcode in the URL instead of generating it from the name like we have been doing. I'll fix that later:

50 lines | src/KnpU/CodeBattle/Model/Battle.php

```
... lines 1 - 9
class Battle
{
... lines 12 - 44
    public function getProgrammerUri()
    {
        return '/api/programmers/'.$this->programmer->nickname;
    }
}
```

But just because you have this method does not means it's going to be served out to your API. You can use an annotation called @Serializer\VirtualProperty:

50 lines | src/KnpU/CodeBattle/Model/Battle.php

```
... lines 1 - 41
    /**
     * @Serializer\VirtualProperty()
     */
    public function getProgrammerUri()
    {
        return '/api/programmers/'.$this->programmer->nickname;
    }
... lines 49 - 50
```

And just like that, it's going to call getProgrammerUri, strip the get off of there, and look like a programmerUri field. And when I run my test, it does exactly that.

Congratulations! We just added our first link. And we're going to add a bunch more!

Chapter 16: I <3 HATEOAS Installation

We just added a link to our API endpoint for getting a single battle. I want to see the response again, so let's add "And print last response" then run the test - it starts on line 26.

```
vendor/bin/behat features/api/battle.feature:26
```

We just invented this idea to put a field on a link called programmerUri. Part of the issue is that we have this link mixed up with other *real* data fields on this property. It's not totally obvious if we can follow this link, or if maybe this is just a field that happens to be a URL, and that we could actually change that URL by sending a PUT request if we wanted to.

HAL JSON: Standardizing How Links Look

A lot of smart people have thought about this and have invented different standardized formats for how your data and links should be organized inside of JSON or XML. One popular one right now is called HAL. I'll click into a document that has a really nice example:

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/user/matthew"
        }
    }
    "id": "matthew",
    "name": "Matthew Weier O'Phinney"
}
```

You can see that down here is the data, and above that, you have a _links property that holds the links. You'll also notice that there's this link called self, and that's something we're going to see over and over again. self is kind of a standard thing where each resource has a link to itself, and you keep that on a key called self. What's cool about this is that it's used on a lot of APIs. So if you ever see a link with the key self, you already know what they're talking about. We're going to add this to our stuff too.

Right now, we're using the serializer to create our JSON. And it works just by looking at the class of whatever object we're serializing and grabs the properties off of it. And of course we have some control over which properties to use.

But if you look at the _links.self thing, you might be wondering how we're going to do this. Are we going to need a bunch of these VirtualProperty things for _links?

Installing HATEOAS

Fortunately, there's a really nice library that integrates with the serializer and helps us add links. It's called HATEOAS, which is a REST term that we'll talk about later.

Before we look at how this library works, let's get it installed. Copy the name, then run composer require and the library name:

```
$ composer require willdurand/hateoas
```

Composer will figure out the best version to bring into the project.

Tip

Since the latest version of willdurand/hateoas is incompatible with some of our dependencies, double-check that you got installed the version ^2.3

How HATEOAS Works

Just like the serializer, this library works via annotations:

```
use Hateoas\Configuration\Annotation as Hateoas;

/**
 * @Hateoas\Relation("self", href = "expr('/api/users/' ~ object.getId())")
 */
class User
{
    private $id;
    private $firstName;
    private $lastName;

    public function getId() {}
}
```

It basically says the User class has a relation called self, and its href should be set to /api/users, and then the id of the user. And we'll talk about this syntax in a second. The end result will be something that looks like this:

```
{
    "_links": {
        "self": {
            "href": "/api/users/12"
        }
    }
    "id": "12",
    "firstName": "Leanna",
    "lastName": "Pelham"
}
```

It'll create the _links key with self and any other links you add below it.

HATEOAS Setup

Getting this setup is pretty easy. Find the HateoasBuilder code and copy it. Quickly, I'll make sure Composer is done downloading the library.

There's always a place inside a Silex application - and most frameworks - to define services, which are just re-usable objects. You might remember from earlier, that in my project, this is done inside this Application class. A few chapters ago, we used this to configure the serializer object. The HATEOAS library hooks right into this, so we just need to modify a few things. Instead of returning the serializer, we'll set it to a variable and take off the call to build(). Now we'll paste the new code in, return it, and pass the $serializerBuilder into the create() method, which is what ties the two libraries together:

```
349 lines | src/KnpU/CodeBattle/Application.php
... lines 1 - 5
use Hateoas\HateoasBuilder;
... lines 7 - 149
    private function configureServices()
    {
        $app = $this;
... lines 153 - 216
        $this['serializer'] = $this->share(function() use ($app) {
            // configure the underlying serializer
            $jmsBuilder = \JMS\Serializer\SerializerBuilder::create()
                ->setCacheDir($app['root_dir'].'/cache/serializer')
                ->setDebug($app['debug'])
                ->setPropertyNamingStrategy(new IdenticalPropertyNamingStrategy());

            // create the Hateoas serializer
            return HateoasBuilder::create($jmsBuilder)->build();
        });
... lines 227 - 230
    }
... lines 232 - 349
```

Now of course, my editor is angry because I'm missing my use statement, so I'll use a shortcut to add that, which puts use statement at the top of this file.

So that's it. We're already using the serializer object everywhere, and now the HATEOAS library will be working with that to add these links for us.

Update the Scenario for Links

Before we add our first link, let's add a scenario to look for it. Go into programmer.feature and find the scenario for getting one programmer. As I mentioned before. it's always a really good idea to have a self link. And if we look at the structure of HAL, this means we're going to have a _links.self.href property, and it'll be set to the URI of our programmer.

So let's add that here: And the "_links.self.href" property should equal - and we know what the URL of this is going to be - /api/programmers/UnitTester:

```
131 lines | features/api/programmer.feature
... lines 1 - 65
  Scenario: GET one programmer
... lines 67 - 69
    When I request "GET /api/programmers/UnitTester"
... lines 71 - 78
    And the "userId" property should not exist
    And the "nickname" property should equal "UnitTester"
    And the "_links.self.href" property should equal "/api/programmers/UnitTester"
... lines 82 - 131
```

And as always, let's run this to see it fail. This scenario is on line 66 of programmer.feature:

```
vendor/bin/behat features/api/programmer.feature:66
```

And it fails because the property doesn't even exist yet.

Adding your First Link

Go back to the HATEOAS docs and scroll back up. Grab the use statement and put it inside of the Programmer class. I'll go back and copy the HATEOAS\Relation annotation - beautiful. This says self, because we want this to be the self link and we'll change the href to be /api/programmers/ and then object.nickname:

54 lines | src/KnpU/CodeBattle/Model/Programmer.php

```
... lines 1 - 7
use Hateoas\Configuration\Annotation as Hateoas;
... line 9
/**
 * @Serializer\ExclusionPolicy("all")
 * @Hateoas\Relation("self", href = "expr('/api/programmers/' ~ object.nickname)")
 */
class Programmer
{
... lines 16 - 52
}
```

The Expression Language

Now, what the heck is this expr thing? This comes from Symfony's expression language, which is a small library that gives you a mini, PHP-like language for writing simple expressions. It has things like strings, numbers and variables. There are also functions and operators - very similar to PHP, but with some different syntax. You only have access to a few variables and a few functions, so you're sandboxed a bit.

In this case, we're saying the URL is going to be this string, then the tilde (~) is the concatenation character, so like the dot (.) in PHP. After that, we have object.nickname. When you use the HATEOAS\Relation, it takes whatever object you're serializing - so Programmer in this case - and makes it available as a variable in the expression called object. So by saying object.nickname, we're saying go get the nickname property.

Let's try this test!

```
vendor/bin/behat features/api/programmer.feature:66
```

Awesome, it passes that easily. Let's print out the response temporarily. And you can see that we *do* have that _links property with self and href keys inside of it. That transformation is all being taken care of by the HATEOAS library.

Chapter 17: HATEOAS Loves Routers

This is great, except that I don't like this hardcoded URL in the Relation:

```
54 lines | src/KnpU/CodeBattle/Model/Programmer.php
... lines 1 - 7
use Hateoas\Configuration\Annotation as Hateoas;
... line 9
/**
 * @Serializer\ExclusionPolicy("all")
 * @Hateoas\Relation("self", href = "expr('/api/programmers/' ~ object.nickname)")
 */
class Programmer
{
... lines 16 - 52
}
```

What I'd rather do is generate that URL from the internal name api_programmers_show like we normally do.

Fortunately, the HATEOAS library allows us to do that, and this can be hooked up to work with any framework, since they all generate URLs. As part of that HateoasBuilder::create() step, you can set a URL generator that does whatever you need.

Hooking up the SymfonyUrlGenerator

If you're using Silex or Symfony, life is a little bit easier because there's a built-in class called SymfonyUrlGenerator:

```
// from the HATEOAS docs
use Hateoas\UrlGenerator\SymfonyUrlGenerator;

$hateoas = HateoasBuilder::create()
    ->setUrlGenerator(null, new SymfonyUrlGenerator($app['url_generator']))
    ->build()
;
```

I'll copy this line. Go back into Application, move build() onto the next line and paste this:

```
352 lines | src/KnpU/CodeBattle/Application.php
... lines 1 - 6
use Hateoas\UrlGenerator\SymfonyUrlGenerator;
... lines 8 - 150
    private function configureServices()
    {
        $app = $this;
... lines 154 - 217
        $this['serializer'] = $this->share(function() use ($app) {
... lines 219 - 224
            // create the Hateoas serializer
            return HateoasBuilder::create($jmsBuilder)
                ->setUrlGenerator(null, new SymfonyUrlGenerator($app['url_generator']))
                ->build();
        });
... lines 230 - 233
    }
... lines 235 - 352
```

And don't forget that we need a use statement for that SymfonyUrlGenerator, so I'll click "import" to have PhpStorm add this class to the top of the file for me. This class comes from the HATEOAS library, and we're just passing it the url_generator object, which in Silex, is the object responsible for generating URLs. In Symfony, it's called router.

Using the Router in Annotations

With this, we can go back into Programmer. First, I'm going to move things onto multiple lines for my sanity. Instead of setting the href to a URL, we'll say @HATEOAS\Route . I'll make sure I have all my parenthesis in the right place. In the Route, we'll have 2 arguments - the first is the name of the route. The second argument is whatever variables we need to pass into the route, in a parameters key. Because the route has the {nickname}, we're going to pass that here using the expression language again. This time, we'll say object - because that represents the Programmer - .nickname. That's fancy way of saying: "generate me a URL, and here's the nickname to use in that URL.":

60 lines | src/KnpU/CodeBattle/Model/Programmer.php

```
... lines 1 - 9
/**
 * @Serializer\ExclusionPolicy("all")
 * @Hateoas\Relation(
 *      "self",
 *      href = @Hateoas\Route(
 *          "api_programmers_show"
 *          parameters = { "nickname" = expr("object.nickname") }
 *      )
 * )
 */
class Programmer
... lines 21 - 60
```

Unless I've messed something up, the test should pass like before:

```
php vendor/bin/behat features/api/programmer.feature:66
```

Ah, and it doesn't! I messed up some syntax. Anytime you see the Doctrine\Common\Annotations T_CLOSE_PARENTHESIS type of thing, this is a syntax error in your annotation. I'm missing a comma between my arguments. Let's try that one more time. Ah, I messed up again! If you look back at the docs, which I've been ignoring, you can see that the quotes should be around the entire nickname value. I'll fix that, and learn that it's always good to follow the docs:

60 lines | src/KnpU/CodeBattle/Model/Programmer.php

```
... lines 1 - 9
/**
 * @Serializer\ExclusionPolicy("all")
 * @Hateoas\Relation(
 *      "self",
 *      href = @Hateoas\Route(
 *          "api_programmers_show",
 *          parameters = { "nickname" = "expr(object.nickname)" }
 *      )
 * )
 */
... lines 20 - 60
```

And *this* time it finally passes. So other than my syntax error, that was easy to fix up. And if this looks overwhelming to you, that's ok. From now on, we're just going to be copying and pasting this and customizing it for whatever links we need.

Chapter 18: Adding Real Links with HATEOAS

Let's add more links! Back in battle.feature, we're returning a programmerUri, which was our way of creating a link before we knew there was a good standard to follow:

35 lines | features/api/battle.feature

```
... lines 1 - 25
  Scenario: GET one battle
... lines 27 - 33
    And the "programmerUri" property should equal "/api/programmers/Fred"
```

So now we can say: And the "_links.programmer.href" property should equal "/api/programmers/Fred":

35 lines | features/api/battle.feature

```
... lines 1 - 25
  Scenario: GET one battle
... lines 27 - 33
    And the "_links.programmer.href" property should equal "/api/programmers/Fred"
```

This time, instead of using self, we're using programmer. There are some special names like self that mean something, but when you're linking from a battle to a programmer, we'll just invent something new. We'll want to use this consistently in our API: whenever we're linking to a programmer, we'll use that same string programmer so that our API clients learn that whenever they see this link link they know what type of resource to expect on the other side.

First, let's run our test - line 26 - and make sure that it fails:

```
php vendor/bin/behat features/api/battle.feature:26
```

Let's go in and add that relation. Open up Battle and also open up Programmer so we can steal the Relation from there as promised. And don't forget, every time you use an annotation for the first time in a class, you need a use statement for it.

And also, since we have this relationship now, I'm going to remove our VirtualProperty down below. So this is really good - we're linking to a Programmer like before. So the route name is good and the nickname is good. The only thing that needs to change is that in order to get the nickname of the programmer for this Battle, we need to say object.programmer.nickname so that it uses the programmer field below. Let's try our test. Ah, and it fails! I got caught by copying and pasting - we *do* have a link, but its name is self. Change that to be programmer:

51 lines | src/KnpU/CodeBattle/Model/Battle.php

```
... lines 1 - 6
use Hateoas\Configuration\Annotation as Hateoas;
... line 8
/**
 * @Hateoas\Relation(
 *      "programmer",
 *      href = @Hateoas\Route(
 *          "api_programmers_show",
 *          parameters = { "nickname" = "expr(object.programmer.nickname)" }
 *      )
 * )
 * @Serializer\ExclusionPolicy("all")
 */
class Battle
... lines 20 - 51
```

And now, we'll get that to pass. Awesome.

Consistency = New Behat Step

Because we're *always* putting links under a _links key, I have a new piece of language that we can use in Behat to check for links:

```
35 lines | features/api/battle.feature
... lines 1 - 25
  Scenario: GET one battle
... lines 27 - 33
    And the link "programmer" should exist and its value should be "/api/programmers/Fred"
```

Why would I do this? It's just proving how consistent we are. This new sentence will look for the _links property, so there's no reason to repeat it in all of our scenarios. So let's try the test again - perfect.

```
php vendor/bin/behat features/api/battle.feature:26
```

We can repeat this same thing over in programmer.feature when we're checking the self link. I'll comment out the old line for reference:

```
131 lines | features/api/programmer.feature
... lines 1 - 65
  Scenario: GET one programmer
... lines 67 - 80
    And the link "self" should exist and its value should be "/api/programmers/UnitTester"
... lines 82 - 131
```

If we run our entire test suite, things keep passing:

```
php vendor/bin/behat
```

I love to see all of that green!

Chapter 19: Hypermedia vs. Media (Buzzwords!)

It's time to talk about a big term in REST -- hypermedia. It's one of those terms that seems like it was invented to scare people, but it's really quite underwhelming. We all know that every response has a Content-Type, like text/html or application/json. So when you hear "media" or "media types" and "content types", they're referring to the same idea.

We have two things: media and we have hypermedia.

Media is...

Media is any format: text/html, text/plain, application/json. These all contain data, and it's as simple as that.

Hypermedia is... Links!

Some of these formats are also called *hypermedia*. What's the difference between media and hypermedia? Hypermedia is a format that has a place for *links* to live. And that's it. The classic case you'll hear people talk about for hypermedia is HTML. HTML is the o.g. (original) hypermedia format. It contains data - like we see here - but it also has a way to express links, via anchor tags and forms are also a type of link. So these are links here and these are links. So implicit in the HTML format is a way to separate links from the rest of your data.

JSON is not Hypermedia

JSON is **not** hypermedia. That may seem confusing, because you might be thinking:

```
"but didn't we just add links to our JSON - isn't that hypermedia?"
```

And that answer is no, because if you read the official JSON specification, all it will talk about is where your curly braces, quotes, colons and commas should go. JSON is about the structure of the data - it says nothing about what's actually inside of the data. So by itself, JSON is just a media type, because there's nothing in there that says where your links should live.

Hal+JSON IS Hypermedia

But what's cool is that we've adopted this HAL JSON. This is something that's built on *top* of JSON: it starts with the JSON structure and then adds extra rules about where links should live. So when you talk about JSON, that's a media format. But when you talk about HAL, that's a hypermedia format, because it's spec tells you that links live below _links.

So that's really it: hypermedia is just a way to say: I have a structure that returns links, and there are rules about where those links live.

Advertising your Hypermedia Type (Content-Type Header)

As soon as you adopt a hypermedia format, instead of returning a Content-Type of application/json, you can return a Content-Type of something different, like application/hal+json. At the bottom of this page, there's an example of what a response looks like:

```
HTTP/1.1 201 Created
Content-Type: application/hal+json
Location: http://example.org/api/user/matthew

{
    ...
}
```

And you can see that the response comes back with a Content-Type of application/hal+json. This is a signal to the client that the response has a JSON structure but has some additional semantic rules on top of it. What's awesome is that if we return this in our API and someone looks at that header, they're going to say:

```
"Oh, what's this application/hal+json format?".
```

If they haven't heard of it, they can Google it and read about the structure and say:

```
"oh, they're using a format where the links live in an `_links` key"
```

along with some other rules.

Globally Setting the Content-Type

Because we're already following HAL, returning this Content-Type header on all of our endpoints is an easy win. In battle.feature, let's add a new scenario line to test this. My editor isn't happy with my language here. If you can't remember your definitions, run behat with the -dl option:

```
php vendor/bin/behat -dl
```

I'll grep this for header because I know I have a definition. Ah, and my language is slightly off. And now PHPStorm is very happy:

36 lines | features/api/battle.feature

```
... lines 1 - 25
  Scenario: GET one battle
... lines 27 - 29
    When I request "GET /api/battles/%battles.last.id%"
    Then the response status code should be 200
... lines 32 - 34
    And the "Content-Type" header should be "application/hal+json"
```

Oh, and we actually want to look for application/hal+json. I'll run the test first, and it's failing:

```
php vendor/bin/behat features/api/battle.feature:26
```

Remember, this is served from BattleController, so let's go back there. And all of our endpoints call this same createApiResponse method:

65 lines | src/KnpU/CodeBattle/Controller/Api/BattleController.php

```
... lines 1 - 52
    public function showAction($id)
    {
        $battle = $this->getBattleRepository()->find($id);
... lines 56 - 59
        $response = $this->createApiResponse($battle, 200);

        return $response;
    }
... lines 64 - 65
```

If we click into this, it opens up the BaseController and this is a method we created earlier. It uses the serializer then creates a Response. So let's just update that Content-Type header:

```
301 lines | src/KnpU/CodeBattle/Controller/BaseController.php
... lines 1 - 236
    protected function createApiResponse($data, $statusCode = 200)
    {
        $json = $this->serialize($data);

        return new Response($json, $statusCode, array(
            'Content-Type' => 'application/hal+json'
        ));
    }
... lines 245 - 301
```

Run the test, and it passes perfectly:

```
php vendor/bin/behat features/api/battle.feature:26
```

Now, API clients can see this header and know that we're using some extra rules on top of the JSON structure.

Chapter 20: We can Embed Resources Too

When we made our Battle endpoint, we decided it might be convenient to have a link to Programmer. It's just a nice thing to help our API clients. But to get that programmer's information, they're going to need to make a second request out to that URI. If it's *really* common to need the programmer information when you GET a battle, you may *choose* to put the Programmer's data, right inside the Battle's response.

What's really nice here is that HAL already has rules about how this should work:

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/user/matthew"
        }
    }
    "id": "matthew",
    "name": "Matthew Weier O'Phinney",
    "_embedded": {
        "contacts": [
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mac_nibblet"
                    }
                },
                "id": "mac_nibblet",
                "name": "Antoine Hedgecock"
            },
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/spiffyjr"
                    }
                },
                "id": "spiffyjr",
                "name": "Kyle Spraggs"
            }
        ]
    }
}
```

There's the _links section, but there's also an _embedded section. And our HATEOAS library will help us put stuff there.

Embedding a Resource

So let's try to embed our programmer into the battle. First, let's add a line to the scenario that looks for this. Let's look for _embedded, and we know it's going to be called programmer, and the data will live below this and we know one of the fields on a programmer is nickname. And we know this should be equal to Fred:

37 lines | features/api/battle.feature

```
... lines 1 - 25
  Scenario: GET one battle
    Given there is a project called "projectA"
    And there is a programmer called "Fred"
    And there has been a battle between "Fred" and "projectA"
    When I request "GET /api/battles/%battles.last.id%"
... lines 31 - 34
    And the "_embedded.programmer.nickname" property should equal "Fred"
... lines 36 - 37
```

Let's make sure that fails first - and it does:

```
php vendor/bin/behat features/api/battle.feature:26
```

To make this work, we'll add more annotations to Battle. When you think of one resource relating to another - like how our Battle relates to the Programmer resource - there are 2 ways to express that relation. You can either link to it *or* you can embed it. Those are both just valid ways to think about expressing a link between 2 resources.

This @HATEOAS\Relation lets you do whichever you want. For a link, use the href. To embed something, use the embedded key and set it to an expression that points to which object you want to embed:

52 lines | src/KnpU/CodeBattle/Model/Battle.php

```
... lines 1 - 8
/**
 * @Hateoas\Relation(
 *     "programmer",
 *     href = @Hateoas\Route(
 *         "api_programmers_show",
 *         parameters = { "nickname" = "expr(object.programmer.nickname)" }
 *     ),
 *     embedded = "expr(object.programmer)"
 * )
... line 18
 */
class Battle
... lines 21 - 52
```

And actually, if you include both href and embedded, it'll create a link *and* embed it.

Before we run the test, add a "And print last response", because I like to see how my endpoints look. Let's run it:

```
php vendor/bin/behat features/api/battle.feature:26
```

Awesome it passes! If you look - HATEOAS is doing all the work for us. We still have _links, but we also have _embedded. What's cool is that it goes out to the Programmer resource and serializes it. You end up with all the same properties as normal, and you even end up with its links. So a lot of things are falling into place accidentally.

Prize: New Behat Definition

And just like with links, since embedded data always lives under _embedded, I have a built-in definition you can choose to use if you want to:

37 lines | features/api/battle.feature

```
... lines 1 - 25
  Scenario: GET one battle
... lines 27 - 34
    And the embedded "programmer" should have a "nickname" property equal to "Fred"
... lines 36 - 37
```

Behind the scenes, this knows to look for all of this on the _embedded property.

And the test still passes. Now I'll take out the print last response. When it comes to linking and embedding, I hope you're feeling dangerous!

Chapter 21: Fun with the HAL Browser!

By using HAL, it gives our responses a nice, predictable structure. And while that's *really* awesome, I want to show you something that's even *more* awesome that we get for free by using Hal.

I'll google for Hal Browser. If you haven't seen this before, I'm not going to tell you want it is until we see it in action. I'll copy the Git URL:

```
git@github.com:mikekelly/hal-browser.git
```

Move into the web/ directory of the project and clone this into a browser/ directory:

```
cd web
git clone git@github.com:mikekelly/hal-browser.git browser
```

Great!

Opening up the Browser

Inside of this directory is a browser.html file - so let's surf to that instead of our actual project:

```
http://localhost:8000/browser/browser.html
```

And let me show you the directory - it's just that HTML file, some CSS and some JS.

What we get is the HAL Browser. This is an API client that knows all about dealing with HAL responses. We can put in the URL. Let's guess that there's a battle whose id is 1 in the database, and make a *real* request to that URL:

```
Put this into the Hal Browser:

http://localhost:8000/api/battles/1
```

We're actually getting a 404 right now, because apparently there's no battle with id 1. I'll re-run a test that puts battles into the database so we have something to play with:

```
php vendor/bin/behat features/api/battle.feature:26
```

Properties, Links and Embedded

Perfect, this time it finds a battle. On the right you see the raw response headers and raw JSON. But on the left, it's actually parsing through those a little bit. It sees that *these* are the properties. So even though it has this big response body on the right, the properties are *only* the id and didProgrammerWin type of fields. It also parses out the links and puts them in a nice "Links" section. And lets us follow that link, which I'll do in a second. Down here, it also has the embedded resource. It shows us that we have an embedded programmer, so let's click that and then it does the same thing. It says: here are the properties, and even says, this embedded resource has a link to itself.

Let's follow the link to the programmer. This actually makes a new GET request to this URL - which is awesome - parses out the properties, and once again, we see the self link. So just by using HAL, we get this for free, which makes it really easy to learn your own API and you can even expose this if you want, to your end users on your site. If you want to try our API, just go to this URL and navigate your way around it.

Chapter 22: Collections: The HAL Way

Let's go to one other endpoint - /api/programmers:

```
Put this into the Hal Browser:

http://localhost:8000/api/programmers
```

Here, things don't quite work out as well. The response body on the right is the same as the properties on the left. That might not look wrong to you, but the format for our collection is not correct yet according to HAL.

Hal Collection Format

If we look back at the HAL Primer doc, you'll see that *this* is what a collection resource should look like:

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/user?page=3"
        },
        "first": {
            "href": "http://example.org/api/user"
        },
        "prev": {
            "href": "http://example.org/api/user?page=2"
        },
        "next": {
            "href": "http://example.org/api/user?page=4"
        },
        "last": {
            "href": "http://example.org/api/user?page=133"
        }
    }
    "count": 3,
    "total": 498,
    "_embedded": {
        "users": [
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mwop"
                    }
                },
                "id": "mwop",
                "name": "Matthew Weier O'Phinney"
            },
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mac_nibblet"
                    }
                },
                "id": "mac_nibblet",
                "name": "Antoine Hedgecock"
            }
        ]
    }
}
```

Collections are "Collection Resources"

By the way, we've talked about resources like Programmer and Battle, but with the collection endpoints - like /api/programmers - the collection itself is known as a resource. So this is a collection resource. And in HAL, the way this works is that all of the items in the collection are considered to be embedded resources. If you imagine that this is /api/users, all the users live under the _embedded key. The only true properties - if you have any - relate to the collection itself: things like the total number of items in the collection or what page you're on.

And above, you'll see that this collection has pagination and it's using the links to help the client know how to get to other pages. We're going to do this exact same thing later, which is awesome and it'll be really easy.

Structuring Our Collections

But for now, I want to get things into this structure. First, before we implement it, you guys know what we're going to do, we're going to update our test. In our programmer.feature, the scenario for the collection *was* looking for a programmers property to be the array. But now, it's going to be _embedded.programmers. Let's go even one step further and say that the first item in this should be the UnitTester:

132 lines | features/api/programmer.feature

```
    ... lines 1 - 82
83    Scenario: GET a collection of programmers
84      Given the following programmers exist:
85        | nickname    | avatarNumber |
86        | UnitTester  | 3            |
87        | CowboyCoder | 5            |
88      When I request "GET /api/programmers"
89      Then the response status code should be 200
90      And the "_embedded.programmers" property should be an array
91      And the "_embedded.programmers" property should contain 2 items
92      And the "_embedded.programmers.0.nickname" property should equal "UnitTester"
    ... lines 93 - 132
```

So its nickname should equal UnitTester. This is line 84, so let's run this test first to make sure it fails:

```
php vendor/bin/behat features/api/programmer.feature:84
```

And it does.

The HATEOAS CollectionRepresentation

The HATEOAS library has its own way of dealing with collections. If you scroll to the top of its docs, you'll see a section all about this:

```
// From the HATEOAS documentation
use Hateoas\Representation\PaginatedRepresentation;
use Hateoas\Representation\CollectionRepresentation;

$paginatedCollection = new PaginatedRepresentation(
    new CollectionRepresentation(
        array($user1, $user2, ...),
        'users', // embedded rel
        'users' // xml element name
    ),
    'user_list', // route
    array(), // route parameters
    1, // page
    20, // limit
    4, // total pages
    'page',  // page route parameter name, optional, defaults to 'page'
    'limit', // limit route parameter name, optional, defaults to 'limit'
    false    // generate relative URIs
);
```

First of all, don't worry about this PaginatedRepresentation thing, we're going to talk about that later - it's not nearly as scary as it looks here. If we have a collection that doesn't need pagination, all we need to do is create this CollectionRepresentation object.

Open ProgrammerController and go down to listAction. Right now we're taking the programmers, putting them onto a programmer key and serializing that. Instead, create a new CollectionRepresentation - I'll use my IDE to auto-complete that, which automatically adds a use statement at the top of the file. The CollectionRepresentation takes two arguments: the items that are inside the collection and the key that these should live under. I could use anything here, but I'll use programmers:

```
154 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 72
    public function listAction()
    {
        $programmers = $this->getProgrammerRepository()->findAll();

        $collection = new CollectionRepresentation(
            $programmers,
            'programmers',
            'programmers'
        );

        $response = $this->createApiResponse($collection, 200, 'json');

        return $response;
    }
... lines 87 - 154
```

Just choose something that makes sense, then be consistent so that whenever your API clients see a programmers key, they know what it contains.

Now, we'll just pass this new object directly to createApiResponse. And hey, that's it! Let's try the test:

```
php vendor/bin/behat features/api/programmer.feature:84
```

And this time it passes! So whenever you have a collection resource, use this object. If it's paginated, stay tuned.

Celebrate with Hal (Browser)

And now that we're following the rules of HAL, if we go back to the HAL browser and re-send the request, you get a completely different page. This time it says we have no properties, since our collection has no keys outside of the embedded stuff, and below it shows our embedded programmers and we can see the data for each. Each programmer even has a self link, which we can follow to move our client to that new URL and see the data for that one programmer. If it has any other links, we could keep going by following those. So hey, this is getting kind of fun! And as you'll see, the HAL browser can help figure out what new links can be added to make life better.

Chapter 23: Link to a Subordinate Resource!

I'm playing with my API and looking at the collection of programmers. And of course, I can follow the self link to GET just that one resource. But now that I'm here, it occurs to me that it would be really cool if I had a battles link we could follow that would return a collection resource of all of the battles that this programmer has been in. So let's do that.

I'm going to add a new scenario inside programmer.feature, since it'll be showing me all the battles for a programmer. I'll call the scenario: "GET a collection of battles of the programmer". At the start of this scenario, we'll need a few projects, one programmer and a few battles between them. We used some similar language in battle.feature. I'll make sure there's a projectA in the database first and then repeat that to make a projectB. And let's make sure that our favorite programmer Fred exists as well. Finally, we'll add two more lines to create 2 battles between Fred and each project:

```
144 lines | features/api/programmer.feature
     ... lines 1 - 93
94     Scenario: GET a collection of battles for a programmer
95       Given there is a project called "projectA"
96       Given there is a project called "projectB"
97       And there is a programmer called "Fred"
98       And there has been a battle between "Fred" and "projectA"
99       And there has been a battle between "Fred" and "projectB"
     ... lines 100 - 144
```

Cool - so that's all the setup work.

This will return a collection resource, so we can steal a lot of the scenario from above, since all collection resources pretty much look the same.

Simple Guide to URL Structures

For the URI, one of the things you'll hear is that URIs don't matter. In theory, you can make whatever URIs you want. So if you're stressing out about how a URI should look, just choose something, because it ultimately doesn't matter.

That being said, you typically follow a pattern. So far we've seen URLs like /api/programmers for a collection and /api/programmers/Fred for a single programmer. And that's a decent pattern. In this case, this is actually what we call a "subordinate" resource - it's the collection of battles *under* a specific programmer. So a good URL for this is the URL to a specific programmer, plus /battles to get the subordinate battles collection resource for Fred. After that, everything will be pretty much the same, changing programmers to battles. We'll even check that the first battle has a didProgrammerWin property, since every battle has that. We don't know what it's going to be set to, but it should definitely be there:

```
144 lines | features/api/programmer.feature
      ... lines 1 - 93
94      Scenario: GET a collection of battles for a programmer
      ... lines 95 - 99
100       When I request "GET /api/programmers/Fred/battles"
101       Then the response status code should be 200
102       And the "_embedded.battles" property should be an array
103       And the "_embedded.battles" property should contain 2 items
104       And the "_embedded.battles.0.didProgrammerWin" property should exist
      ... lines 105 - 144
```

Great!

This starts on line 95, so let's run this and make sure it fails with a 404:

```
php vendor/bin/behat features/api/programmer.feature:95
```

Cool!

Coding up the Programmer's Battles Endpoint

Let's get to work! Open ProgrammerController. We'll need a new route and I'll copy the "show" route, since the URL will be really similar. We'll add the /battles in the end and change the method to listBattlesAction:

```
179 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 19
    protected function addRoutes(ControllerCollection $controllers)
    {
... lines 22 - 36
        $controllers->get('/api/programmers/{nickname}/battles', array($this, 'listBattlesAction'))
            ->bind('api_programmers_battles_list');
    }
... lines 40 - 179
```

The route name isn't important yet, but we'll use it later to link. Let's call it api_programmers_battles_list.

Implementing this is going to be really easy! I'll put it right between showAction and listAction so I can steal from both. Ok, let's think about what we need to do. First, we need to find the Programmer for this nickname. We have code for this, so let's steal it:

```
179 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 90
    public function listBattlesAction($nickname)
    {
        $programmer = $this->getProgrammerRepository()->findOneByNickname($nickname);

        if (!$programmer) {
            $this->throw404('Oh no! This programmer has deserted! We\'ll send a search party!');
        }
... lines 98 - 105
        );
... lines 107 - 179
```

If you find yourself repeating a lot of code like this, you can always create a private function inside your controller class and put it there. That's similar to what we've been doing by putting functions inside of BaseController.

The second thing we need to do is to find all of the battles that are linked to this programmer. I have a shortcut for this that I'll use:

```
179 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 90
    public function listBattlesAction($nickname)
    {
        $programmer = $this->getProgrammerRepository()->findOneByNickname($nickname);
... lines 94 - 98
        $battles = $this->getBattleRepository()
            ->findAllBy(array('programmerId' => $programmer->id));
... lines 101 - 110
    }
... lines 112 - 179
```

The code might be different in your project, but this is just saying:

```
"Hey, go query the battle table where programmerId is equal to the id
of the programmer that we have."
```

So what's cool is that from here, this is exactly like the listAction, because it's just a collection resource. So I'm going to grab everything from it, change the variable to $battles, change the key to battles, and that's it!

```
179 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 90
    public function listBattlesAction($nickname)
    {
... lines 93 - 98
        $battles = $this->getBattleRepository()
            ->findAllBy(array('programmerId' => $programmer->id));

        $collection = new CollectionRepresentation(
            $battles,
            'battles',
            'battles'
        );

        $response = $this->createApiResponse($collection);

        return $response;
    }
... lines 112 - 179
```

So with almost no work, we'll run the test again, and it passes!

```
php vendor/bin/behat features/api/programmer.feature:95
```

Adding the battles Relation

Back on the Hal Browser, if we hit Go, we *still* don't have that link. *We* know that we can just add /battles onto the URL, but there's no link yet. Let's add it!

Open up the Programmer class and copy and paste to create a second Relation. This time the key will be battles, and below we'll grab the route name we created for the endpoint:

```
67 lines | src/KnpU/CodeBattle/Model/Programmer.php
... lines 1 - 7
use Hateoas\Configuration\Annotation as Hateoas;
... line 9
/**
... lines 11 - 18
 * @Hateoas\Relation(
 *     "battles",
 *     href = @Hateoas\Route(
 *         "api_programmers_battles_list",
 *         parameters = { "nickname" = "expr(object.nickname)" }
 *     )
 * )
 */
class Programmer
... lines 28 - 67
```

Then, everything else looks good, because this route *does* need the nickname. So you *may* want to write a test for this if it's really important, but I'm just going to go back to the Hal browser, click "Go", and boom! We have a new battles link, which we can follow and see the collection resource. We can open up one of the battles, follow a link back to the related programmer and click to go back to the battles once again. We're surfing through our API, which is really cool!

Adding the Battle self Relation

If we click to look at a battle, you'll notice that we're missing one little thing. It has a programmer link, but no self link, which we really want every resource to have because it's a nice standard and it comes in handy. So let's go add this, which is *really* easy.

Open the Battle class and copy the relation. Let's remove the embedded option. We just want this to be a normal link. Change the link name to self and go find the route name from inside BattleController. For this, it's api_battle_show. In this case, the route needs the id of the battle. So on the relation, we can simply say object.id:

59 lines | src/KnpU/CodeBattle/Model/Battle.php

```
... lines 1 - 6
use Hateoas\Configuration\Annotation as Hateoas;
... line 8
/**
 * @Hateoas\Relation(
 *      "self",
 *      href = @Hateoas\Route(
 *          "api_battle_show",
 *          parameters = { "nickname" = "expr(object.id)" }
 *      )
 * )
... lines 17 - 25
 */
class Battle
... lines 28 - 59
```

Awesome!

If we re-GET this request, we see a huge error! This is no bueno! But hey, let's run our test for this to see if it helps us:

```
php vendor/bin/behat features/api/programmer.feature:95
```

And you can see that we're missing some "id" parameter when generating the URL. I made a mistake in the Relation. You probably saw me do it, but I'm still passing a nickname instead of passing the id:

59 lines | src/KnpU/CodeBattle/Model/Battle.php

```
... lines 1 - 8
/**
 * @Hateoas\Relation(
 *      "self",
 *      href = @Hateoas\Route(
 *          "api_battle_show",
 *          parameters = { "id" = "expr(object.id)" }
 *      )
 * )
... lines 17 - 25
 */
class Battle
... lines 28 - 59
```

So now, things work. Thank God for our tests, because that was really easy to debug. And every battle *now* has that self link.

Chapter 24: A Homepage for your API?

If I tell you to go to a webpage, I'll probably tell you to go to its homepage - like KnpUniversity.com - because I know once you get there, you'll be able see links, follow them and find whatever you need. And if you think about it, there's no reason an API has to be any different. And this is an idea that's catching on.

So right now, if I go to just /api in the Hal browser we get a 404 response, because we haven't built anything for this yet. I'm thinking, why not build a homepage that people can go to and get information about how to use our API?

API Homepage: Start Simple

I'll build this in ProgrammerController just for convenience. Let's add the new /api route and point it at a new method called homepageAction:

187 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 17
class ProgrammerController extends BaseController
{
    protected function addRoutes(ControllerCollection $controllers)
    {
        // the homepage - put in this controller for simplicity
        $controllers->get('api', array($this, 'homepageAction'));
... lines 24 - 41
    }
... lines 43 - 185
}
```

Inside of here, I'm not going to return anything yet - let's just reuse that same createApiResponse, and I'll pass it an empty array:

187 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 43
    public function homepageAction()
    {
        return $this->createApiResponse(array());
    }
... lines 48 - 187
```

If we try this, we get an empty response, but it has a valid application/hal+json. So that's all we really need to do to get an endpoint working.

A Model Class for Every Resource

We know that every URL is a resource in the philosophical sense: /api/programmers is a collection resource and /api/programmers/Fred represents a programmer resource. And really, the /api endpoint is no different. By the way, I did not mean to leave the / off of my path - it doesn't matter if you have it, but I'll add it for consistency.

187 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 19
    protected function addRoutes(ControllerCollection $controllers)
    {
        // the homepage - put in this controller for simplicity
        $controllers->get('/api', array($this, 'homepageAction'));
... lines 24 - 41
    }
... lines 43 - 187
```

So far, every time we have a resource, we have a model class for it. Why not do the same thing for the Homepage resource? Create a new class called Homepage:

20 lines | src/KnpU/CodeBattle/Model/Homepage.php

```
<?php

namespace KnpU\CodeBattle\Model;
... lines 4 - 5

/**
 * A model to represent the homepage resource
 *
... lines 10 - 15
 */
class Homepage
{
}
```

And without doing anything else, I'll create a new object called $homepage. Don't forget to add your use statement whenever you reference a new class in a file. And instead of the empty array, we'll pass createApiResponse() the Homepage object:

191 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 45
    public function homepageAction()
    {
        $homepage = new Homepage();

        return $this->createApiResponse($homepage);
    }
... lines 52 - 191
```

So every time we have a resource, we have a class for it. It doesn't really matter if the class is being pulled from the database, being created manually or being populated with data from something like Elastic Search.

Homepages Love Links

If we hit Go on the browser, we get the exact same response back: no difference yet. But now that we have a model class, we can start adding things to it. And since every resource has a self link, let's add that to Homepage too. I'll grab the Relation from programmer for convenience. And of course, grab the use statement for it: I know I'm repeating myself over and over again!

Now, we need the name of the route. So let's go give this new route a name: api_homepage:

```
191 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 20
    protected function addRoutes(ControllerCollection $controllers)
    {
... line 23
        $controllers->get('/api', array($this, 'homepageAction'))
            ->bind('api_homepage');
... lines 26 - 43
    }
... lines 45 - 191
```

We'll use this in the Relation. And because there aren't any wildcards in the route, we don't need any parameters.

```
20 lines | src/KnpU/CodeBattle/Model/Homepage.php
... lines 1 - 4
use Hateoas\Configuration\Annotation as Hateoas;

/**
 * A model to represent the homepage resource
 *
 * @Hateoas\Relation(
 *     "self",
 *     href = @Hateoas\Route(
 *         "api_homepage"
 *     )
 * )
 */
class Homepage
{
}
```

Cool!

Let's try this out! We have a link! Now, I know this doesn't seem very useful yet, because we have an API homepage that's linking to itself, but now if we wanted to, we could link back to the homepage from other resources. So if we were on the /api/programmers resource, we could add a link back to the homepage. When an API client GET's that URL, they'll see that there's a place to get more information about the entire API.

Other Link Attributes (like title)

When you look at the Links section, there are a few other columns like title, name/index and docs. One of the things that this is highlighting is that your links can have more than just that href. So in Homepage, let's give the link a title of "The API Homepage":

```
21 lines | src/KnpU/CodeBattle/Model/Homepage.php
... lines 1 - 6
/**
 * A model to represent the homepage resource
 *
 * @Hateoas\Relation(
 *     "self",
 *     href = @Hateoas\Route(
 *         "api_homepage"
 *     ),
 *     attributes = {"title": "Your API starting point" }
 * )
 */
class Homepage
... lines 19 - 21
```

Let's go back to the browser to see that title. And now we can give any link a little bit more information.

Hi, Welcome! Homepage Message

Let's keep going! Since this is the API homepage, you probably want to give the client a nice welcome message and maybe even link to the human-readable documentation. Even though this Homepage class isn't being pulled from the database doesn't mean that we can't have properties. Let's create a $message property and set that to some hard-coded text. You could even put your documentation URL here:

```
22 lines | src/KnpU/CodeBattle/Model/Homepage.php
... lines 1 - 17
class Homepage
{
    private $message = 'Welcome to the CodeBattles API! Look around at the _links to browse the API. And have a crazy-cool day.';
}
```

And now, our API homepage is getting interesting! But the *real* purpose of this homepage is to have links to the actual resources that the API client will want. This is really easy as well. The most obvious resource the client may want is the programmers collection. So let's do this here. We'll say programmers and we'll link to /api/programmers. That route doesn't have a name yet, so let's call it api_programmers_list:

```
192 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 20
    protected function addRoutes(ControllerCollection $controllers)
    {
... lines 23 - 31
        $controllers->get('/api/programmers', array($this, 'listAction'))
            ->bind('api_programmers_list');
... lines 34 - 44
    }
... lines 46 - 192
```

And now we can use it in the Relation. To be super friendly, we'll give this link a nice title as well:

```
29 lines | src/KnpU/CodeBattle/Model/Homepage.php
... lines 1 - 6
/**
... lines 8 - 16
 * @Hateoas\Relation(
 *      "programmers",
 *      href = @Hateoas\Route(
 *          "api_programmers_list"
 *      ),
 *      attributes = {"title": "The list of all programmers" }
 * )
 */
class Homepage
... lines 26 - 29
```

So let's hit Go. Now we have a really great homepage. We can come here, we can see the message with a link to the documentation, we can visit all of the programmers, and now we're dangerous. From there we can follow links to a programmer, to that programmer's battles, and anything else our links lead us to.

Chapter 25: API Pagination Done Easily

Let's do some pagination! Technically, HAL doesn't have any opinion on how all of this should work. But fortunately, the rest of the REST world *does*. As you can see in this example, a great way to handle pagination is with links (from http://phlyrestfully.readthedocs.org/en/latest/halprimer.html#collections):

```
{
    "_links": {
        "self": {
            "href": "http://example.org/api/user?page=3"
        },
        "first": {
            "href": "http://example.org/api/user"
        },
        "prev": {
            "href": "http://example.org/api/user?page=2"
        },
        "next": {
            "href": "http://example.org/api/user?page=4"
        },
        "last": {
            "href": "http://example.org/api/user?page=133"
        }
    }
    "count": 3,
    "total": 498,
    "_embedded": {
        "users": [
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mwop"
                    }
                },
                "id": "mwop",
                "name": "Matthew Weier O'Phinney"
            },
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/mac_nibblet"
                    }
                },
                "id": "mac_nibblet",
                "name": "Antoine Hedgecock"
            },
            {
                "_links": {
                    "self": {
                        "href": "http://example.org/api/user/spiffyjr"
                    }
                },
                "id": "spiffyjr",
                "name": "Kyle Spraggs"
            }
        ]
    }
}
```

The keys here - first, prev, next and last - aren't accidental. These are Internet-wide standards like self. So if you use these names for your links, your API will be consistent with a lot of other APIs.

Pagination: Do it with Query Parameters

The other important thing to notice in this example is that pagination is done with query parameters. Technically, there are a lot of ways the client could tell us what page they want for a collection, like query parameters or request headers. But honestly, query parameters are the easiest way. In our case, we're going to follow what you see here exactly. And with the HATEOAS library, this will be easy.

Pagination Scenario

First, let's setup a scenario to test this in programmers.feature. In this scenario, we're going to do something really cool: we're going to follow the links for pagination. I want to be able to go to our collection resource, get the URL for this next link and make a second request to the next link and actually see what's on page 2.

For our pagination, we'll show 5 programmers per page. In the Given, we need to add a bunch of programmers to try this out - I'll paste in some very imaginative code that gives us 12 programmers in the database:

166 lines | features/api/programmer.feature

```
... lines 1 - 93
  # we will do 5 per page
  Scenario: Paginate through the collection of programmers
    Given the following programmers exist:
      | nickname    |
      | Programmer1 |
      | Programmer2 |
      | Programmer3 |
      | Programmer4 |
      | Programmer5 |
      | Programmer6 |
      | Programmer7 |
      | Programmer8 |
      | Programmer9 |
      | Programmer10 |
      | Programmer11 |
      | Programmer12 |
... lines 110 - 166
```

Everything after this will be very similar to the normal collection resource, so I'll grab a the second half of that scenario. I'll remove the status code 200, because we're already testing for this above. After I make the first GET request, we're going to parse through the response, find the next link and make a second GET request. I already have a built-in step definition to do exactly that. I'll just say: And I follow the "next" link:

166 lines | features/api/programmer.feature

```
... lines 1 - 94
  Scenario: Paginate through the collection of programmers
    Given the following programmers exist:
... lines 97 - 109
    When I request "GET /api/programmers"
    And I follow the "next" link
... lines 112 - 166
```

If you looked at the implementation behind this - which I wrote - it knows that we're using HAL, so it knows to look at _links, next, href and make a second GET request for that.

So on the second page, we'd expect there to be Programmer7, so we can say: And the "_embedded.programmers" property should contain "Programmer7", because we expect that word to be somewhere inside that JSON. And we expect there to *not* be Programmer2 and for there to also *not* be Programmer11:

166 lines | features/api/programmer.feature

```
... lines 1 - 93
  # we will do 5 per page
  Scenario: Paginate through the collection of programmers
... lines 96 - 111
    Then the "_embedded.programmers" property should contain "Programmer7"
    But the "_embedded.programmers" property should not contain "Programmer2"
    But the "_embedded.programmers" property should not contain "Programmer11"
... lines 115 - 166
```

Programmer2 should be on page 1 and Programmer11 should be on page 3.

This starts on line 96, so let's try the scenario out:

```
php vendor/bin/behat features/api.programmer.feature:96
```

It fails of course because it can't find the next link - our collection response doesn't have that because we haven't done any of the work for it yet.

Adding Pagination Links

If we look at the HATEOAS documentation, they talk about pagination. To do pagination, you'll return this PaginatedRepresentation resource. So let's create that in ProgrammerController. So here is listAction where we're getting our programmers. And remember, right now we're creating a CollectionResource, and that's what you return when you have a collection resource, but you don't need it to have pagination:

192 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 88
    public function listAction()
    {
        $programmers = $this->getProgrammerRepository()->findAll();

        $collection = new CollectionRepresentation(
            $programmers,
            'programmers',
            'programmers'
        );
... lines 98 - 101
    }
... lines 103 - 192
```

Below this, we'll say $paginated = new PaginatedRepresentation. My IDE just added the use statement at the top for us - make sure you have it:

```
205 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
    ... lines 1 - 5
use Hateoas\Representation\PaginatedRepresentation;
    ... lines 7 - 89
    public function listAction()
    {
    ... lines 92 - 97
        $collection = new CollectionRepresentation(
            $programmers,
            'programmers',
            'programmers'
        );
        $paginated = new PaginatedRepresentation(
    ... lines 104 - 109
        );
    ... lines 111 - 114
    }
    ... lines 116 - 205
```

This takes a number of different arguments. The first is the actual CollectionRepresentation. The second is the route name to the list endpoint, which for us is api_programmers_list. And it'll use this to generate the links like next, first and last. The third argument is any array of parameters that need to be passed to the route. So if the route had a nickname or id wildcard, you'd pass that here. But there aren't any wildcards in this route, so we'll pass an empty array. The next three arguments are the page we're on, the number of records we're showing per page, and the total number of pages. I'm going to invent a few variables and set them above in a moment:

```
205 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
    ... lines 1 - 89
    public function listAction()
    {
    ... lines 92 - 102
        $paginated = new PaginatedRepresentation(
            $collection,
            'api_programmers_list',
            array(),
            $page,
            $limit,
            $numberOfPages
        );
    ... lines 111 - 114
    }
    ... lines 116 - 205
```

Above, let's fill this in. Initially, I just want to get the next and last _links to show up, so I'm going to take some shortcuts and not worry about truly paginating the results quite yet. I'll hardcode these variables for now. I'll say that we're always on page 1, that the limit is always 5, and we can calculate the total number of pages. If we have 12 programmers, divided by 5 gives us 2.4, then we'll round that up with the ceil function:

205 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 89
    public function listAction()
    {
... lines 92 - 93
        $limit = 5;
        $page = 1;
        $numberOfPages = (int) ceil(count($programmers) / $limit);
... lines 97 - 102
        $paginated = new PaginatedRepresentation(
            $collection,
            'api_programmers_list',
            array(),
            $page,
            $limit,
            $numberOfPages
        );
... lines 111 - 113
        return $response;
... lines 115 - 205
```

Normally you'd use a library to help with pagination, but since I'm faking it, I'm just doing some manual work myself.

Finally, now that we have this $paginated object, we'll pass it to createApiResponse:

205 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
... lines 1 - 89
    public function listAction()
    {
... lines 92 - 102
        $paginated = new PaginatedRepresentation(
            $collection,
            'api_programmers_list',
            array(),
            $page,
            $limit,
            $numberOfPages
        );

        $response = $this->createApiResponse($paginated, 200, 'json');

        return $response;
    }
... lines 116 - 205
```

Cool. We'll still returning *all* of the programmers, so I don't expect our test to pass, but let's try it:

```
php vendor/bin/behat features/api.programmer.feature:96
```

Our test fails, but it *is* getting further. Ah, it actually *is* following the next link and it *is* actually seeing that the Programmer7 resource is in the response. But then it's failing because Programmer2 and every other programmer is still there.

Hal Browser <3's Pagination Links

What's really cool is that if we go back to our Hal browser and go to /api/programmers, those links are showing up. We also have some nice properties that tell us about the collection. And we can start paginating through the resources by following those links. And the way the links are done is just via ?page=1, ?page=2&limit=5.

Adding Real Pagination

Let's turn this into *real* pagination! Since the page and limit are being passed as query parameters, let's use those. In my application, when I need request information, I just add a Request $request argument to my controller:

```
208 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 89
    public function listAction(Request $request)
    {
... lines 92 - 117
    }
... lines 119 - 208
```

Then for query parameters, I can say $request->query->get('page') and the second argument is the default value if there is no page sent for some reasons. And the same for limit - we'll let the client control this, but we'll default to 5:

```
208 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 89
    public function listAction(Request $request)
    {
... lines 92 - 93
        $limit = $request->query->get('limit', 5);
        $page = $request->query->get('page', 1);
... lines 96 - 117
    }
... lines 119 - 208
```

Normally when you do pagination, you might do a LIMIT, OFFSET query to the database or pass which subset of records you want to some search engine, like Elastic Search. In both cases, we end up with *just* the 5 records we want, instead of all 10,000. Here, I'm going to be lazy and *not* do that. I'm just going to query for *all* of my programmers, then just use a little PHP array magic to only give us the ones we want. I'm just trying to keep things simple.

If I were doing this in a real project, I'd probably use a library called Pagerfanta. It helps you paginate and has built-in adapters already for Doctrine, and Elastic Search. You can give it things like, we're on page 2, we're showing 5 results per page, and then it'll do the work to find only the results we need.

Instead of that, I'm going to paste in some manual logic and use array_slice:

```
208 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php
... lines 1 - 89
    public function listAction(Request $request)
    {
... lines 92 - 93
        $limit = $request->query->get('limit', 5);
        $page = $request->query->get('page', 1);
        // my manual, silly pagination logic. Use a real library
        $offset = ($page - 1) * $limit;
        $numberOfPages = (int) ceil(count($programmers) / $limit);

        $collection = new CollectionRepresentation(
            // my manual, silly pagination logic. Use a real library
            array_slice($programmers, $offset, $limit),
... lines 103 - 104
        );
... lines 106 - 117
    }
... lines 119 - 208
```

So I'm querying for all of the programmers and then slicing those down to the ones I want. It's not an efficient way to do this, but we *should* have a functional, paginated endpoint now. So let's try the test again, and it works!

```
php vendor/bin/behat features/api.programmer.feature:96
```

Yes!

And to enjoy it a little bit more, we can go back to the Hal Browser, hit Go, and since we're on page 2, we see Programmers 6-10. We can follow links to the first page, then over to the last page to see only those 2 programmers. Now pagination isn't hard, and it'll be really consistent across your API.

Chapter 26: Filtering and HATEOAS (The Buzzword)

Similar to pagination is filtering. Let's say that I want to be able to search in the programmers collection by nickname. So I'll add a &nickname=2. Why 2? Well, my programmer nicknames aren't very interesting, but if I use 2, I'll want it to return Programmer2 and Programmer12. Of course right now, this doesn't have any affect yet.

Filtering: Use Query Parameters

Filtering is actually really easy. But the reason I wanted to cover it is that sometimes people wonder if they should get clever with their URLs when they're filtering and maybe come up with URLs like /api/programmers/filter/nickname2. Don't do that. If you're filtering, use a query parameter, end of story.

Coding up a Simple Filter

So how do we get this to work? It couldn't be simpler. At the top of listAction, I'll look for the nickname query parameter. And if it's present, we'll query in a special way. And if it's not, we'll do the normal findAll. I have a shortcut setup to query using MySQL LIKE. I'll pass it the value surrounded by the percent signs:

214 lines | src/KnpU/CodeBattle/Controller/Api/ProgrammerController.php

```
    ... lines 1 - 89
    public function listAction(Request $request)
    {
        $nicknameFilter = $request->query->get('nickname');
        if ($nicknameFilter) {
            $programmers = $this->getProgrammerRepository()
                ->findAllLike(array('nickname' => '%'.$nicknameFilter.'%'));
        } else {
            $programmers = $this->getProgrammerRepository()->findAll();
        }
    ... lines 99 - 123
    }
    ... lines 125 - 214
```

And that should be it!

If we go back to the Hal Browser and hit go, we get nothing back! But we're actually on page 3, so click to go back to page 1. Hmm, now we have too many results! That's because we lost the &nickname query parameter. That's because I was lazy - if I have extra filters I *should* pass those to the 3rd argument of PaginatedRepresentation. If I do that, it'll show up in our pagination links.

I'll re-add &nickname=2 manually and this time, we see it's *only* returning Programmer2 and Programmer12, and it knows that there's only going to be one page of them. So that's it for filtering: use query parameters, do whatever logic you need for filtering, and pass the filter to PaginatedRepresentation, even though I didn't do it here. It's that easy.

Hypermedia as the Engine of Application State (HATEOAS)

One quick thing to notice is that even though we now support this nickname filter, there isn't any way for the API client to know this just by looking at the links. We have links to help them go through the pages, but no link that says: "Hey, if you want, you could pass a nickname query parameter to filter this collection". There's not really a way with HAL to do that. There *are* other formats that give you more options and ways to say "Hey, you can add the following 4 query parameters to this URI, and here's what each will do".

And this is one of those spots where REST can get frustrating. The purpose of adding links is to *help* make your API client's life easier. They are *not* intended to replace human-readable documentation. So even if you *did* find a clever way to include filtering information in your response, this would just be as a "nice feature" for your API. You should still document which endpoints have which filters and what they mean.

The term HATEOAS means: hypermedia as the engine of application state. And as cool as it is, it's at the heart of this confusion. In its most pure form, HATEOAS is an idea that seems to suggest that if your API has really good links, it doesn't need human-readable technical documentation. Instead, a client can follow the links in your response and other details you return to figure everything out.

But this is a dream, not a reality. As cool as our API is, it lacks a *ton* of details. For example, it doesn't have any information about filtering. Second, it doesn't tell you which HTTP methods each URL supports, nor what fields you should POST to /api/programmers in order to create a new resource. The links we have are nice, but they're *nowhere* near giving the API client everything it needs.

So think of links as a nice *addition* to your API, but not something that'll replace really nice human documentation.