Symfony RESTful API: Course 1

With <3 from SymfonyCasts

Chapter 1: POST To Create

Tip

In this course we're using Symfony 2, but starting in episode 4, we use Symfony 3. If you'd like to see the finished code for this tutorial in Symfony 3, download the code from episode 4 and check out the start directory!

Well hey guys! I've wanted to write this series for *years*, and now that it's here, I'm so *pumped*! That's because even though building an API can be really tough, the system we're about to build feels simple, and really a bit beautiful.

We have another REST series on the site where we build the API in Silex and learn the short list of REST concepts like resources, representations, what status codes to return, what headers to set, how to format your JSON and a few other buzzwords like hypermedia, HATEOAS and of course, don't forget about our favorite: idempotency.

But in this series, I'll assume you have a basic grasp of this stuff and we'll get straight to work. If you're confused by a term, head back to that series to fill in any gaps.

The Project

Ok, I've got the "start" directory for the project downloaded, I've configured parameters.yml and I've already run composer install. So let's launch the built-in web server:

```
$ php app/console server:start
```

Hey, it's Code Battles! This is the same awesome project we built in Silex for the other REST series. It already has a slick web interface - so we're going to build the API. To make sure we can login, let's create the database and load the fixtures:

```
$ php app/console doctrine:database:create
$ php app/console doctrine:schema:create
$ php app/console doctrine:fixtures:load
```

Now login with a fixtures user: weaverryan and the very secure password foo.

The Code Battles Web Interface

To understand the API we're going to build, let me give you a quick 60-second tour. And please keep your hands and arms inside the project at all times.

The first resource is a programmer, and we start by creating one. Give it a name, a clever tag line, choose one of the avatars and compile! Next, a programmer has energy, and you can change that by powering them up. Sometimes good things happen that give you power, sometimes bad things happen -- like a case of the Mondays.

With some power, you can start a battle. These are projects, and projects are the second resource. And when you select one, it creates our third resource: a battle. Our programmer killed it! Each battle is between one programmer resource and one project resource. On the homepage, you can see a list of all the battles our programmer has bravely fought.

POST to /api/programmers

So where do we start with the API? Well, other than logging in - which we'll talk about later - the first thing we do on the web is create the programmer. That's where we should start. Building an API is no different than building for the web: you need to step back and *think* about your user-flow and build things piece-by-piece in that order.

Open up app/config/routing.yml. I'm loading annotation routes from a Controller/Web sub-directory. I put all my web stuff there because now I can create an Api directory right next it and keep things organized.

In routing.yml, I'll keep two separate route imports: one for Web/ and I'll add a new one for Api/. Trust me - this will come in handy later:

8 lines | app/config/routing.yml

```
app_web:
    resource: "@AppBundle/Controller/Web"
    type:     annotation

app_api:
    resource: "@AppBundle/Controller/Api"
    type:     annotation
```

Now create the new ProgrammerController - and make it extend Symfony's Controller like normal:

21 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
<?php

namespace AppBundle\Controller\Api;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
... lines 6 - 9
class ProgrammerController extends Controller
{
... lines 12 - 19
}
```

Our first endpoint will be for *creating* Programmers, so let's start with public function newAction(). Above it, setup the @Route annotation with the URL /api/programmers. Let's also make it only respond to POST requests:

21 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 4
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
... lines 8 - 9
class ProgrammerController extends Controller
{
    /**
     * @Route("/api/programmers")
     * @Method("POST")
     */
    public function newAction()
    {
... line 18
    }
}
```

URL Structures and HTTP Methods

Ok, we just made 2 interesting architectural decisions:

First, we're going to start all our API URI's with /api. That's opinionated, and RESTfully speaking, it's wrong. REST says that if we want to return an HTML *or* JSON representation of a programmer resource, we should have just *one* URI - like /programmers/HappyCoderCat. This one URI should be able to return both formats based on a header the client sends.

If you want to do this, awesome - go for it! But it's not easy to do, and I'm not sure it's worth it. That's why we've separated the Web and Api stuff into different controllers and URIs. Now we can focus *just* on getting our API right.

The second architectural decision we made was to create a new resource by sending a POST request to that resource's collection URI - so /api/programmers. If you're curious why, watch our other screencast and learn about idempotency. And, in REST, you can make your URLs look however you want. But in practice, we're going to use a very consistent pattern.

Because even though you can make your URLs super weird you probably shouldn't.

"Testing" the POST Endpoint

We'll return a new Response from the controller: Let's do this!

21 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\Response;
... lines 9 - 11
    /**
     * @Route("/api/programmers")
     * @Method("POST")
     */
    public function newAction()
    {
        return new Response('Let\'s do this!');
    }
... lines 20 - 21
```

Ok, so the easy days of just refreshing our browser to try this out are gone: we can't POST here directly in a browser. Now, a lot of people use Postman or something like it to test their API. And while it's great, I think there's a better way.

For now, create a new file - testing.php - right at the root of the project. Inside, require Composer's autoloader:

16 lines | testing.php

```
<?php

require __DIR__.'/vendor/autoload.php';
... lines 4 - 16
```

We're going to use the Guzzle library to hit our new endpoint and make sure it's working. I already installed it into the project - so go directly to $client = new Client([]) and pass it some configuration:

16 lines | testing.php

```
... lines 1 - 4
$client = new \GuzzleHttp\Client([
    'base_url' => 'http://localhost:8000',
    'defaults' => [
        'exceptions' => false
    ]
]);
... lines 11 - 16
```

Tip

We are using Guzzle 5. If you are using Guzzle 6 (or newer), there are (at least) 2 important changes:

- The base_url option has changed to base_uri
- The defaults.exceptions option has changed to http_errors
- echo'ing a response object doesn't work anymore. Instead, use echo $response->getBody();

The first is base_url set to localhost:8000. Next, pass it a defaults key - these are options that'll be passed, by default, to each request. Set one option - exceptions - to false. Normally, if our server returns a 400 or 500 status code, Guzzle blows up with an Exception. This makes it act normal - it'll return a Response *always*. Trust me, that's nice!

Now make the request - $response = $client->post('/api/programmers'). Echo the $response - it's an object, but has a really pretty __toString method on it:

16 lines | testing.php

```
... lines 1 - 11
$response = $client->post('/api/programmers');

echo $response;
echo "\n\n";
```

Try it by hitting this file from the command line:

```
$ php testing.php
```

Ok, let's fill in the guts and make this work!

Chapter 2: Finish POST with a Form

To create a programmer, our client needs to send up some data. And while you can send that data as JSON, XML, form-encoded or any insane format you dream up, you'll probably want your clients to send JSON... unless you work for the banking or insurance industry. They love XML.

For the JSON, we can design it to have any keys. But since our Programmer entity has the properties nickname, avatarNumber and tagLine, let's use those.

These don't have to be the same, but it makes life easier if you can manage it.

Back in testing.php, create a $nickname - but make it a little bit random: this has a unique index in the database and I don't want everything to blow up if I run the file twice. Make a $data array and put everything in it. The avatarNumber is *which* built-in avatar you want - it's a number from 1 to 6. And add a tagLine:

```
25 lines | testing.php
... lines 1 - 11
$nickname = 'ObjectOrienter'.rand(0, 999);
$data = array(
    'nickname' => $nickname,
    'avatarNumber' => 5,
    'tagLine' => 'a test dev!'
);
... lines 18 - 25
```

To send this data, add an options array to post. It has a key called body, and it's literally the raw string you want to send. So we need to json_encode($data):

```
25 lines | testing.php
... lines 1 - 11
$nickname = 'ObjectOrienter'.rand(0, 999);
$data = array(
    'nickname' => $nickname,
    'avatarNumber' => 5,
    'tagLine' => 'a test dev!'
);

$response = $client->post('/api/programmers', [
    'body' => json_encode($data)
]);
... lines 22 - 25
```

Reading the Request Body

This looks good - so let's move to our controller. To *read* the data the client is sending, we'll need the Request object. So add that as an argument:

24 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 7
use Symfony\Component\HttpFoundation\Request;
... lines 9 - 12
    /**
     * @Route("/api/programmers")
     * @Method("POST")
     */
    public function newAction(Request $request)
    {
... lines 19 - 21
    }
... lines 23 - 24
```

To get the JSON string, say $body = $request->getContent(). And to prove things are working, just return the POST'd body right back in the response:

24 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 16
    public function newAction(Request $request)
    {
        $data = $request->getContent();

        return new Response($data);
    }
... lines 23 - 24
```

The client is sending a JSON string and our response is just sending that right back. Try it!

```
$ php testing.php
```

Hey, that's prefect! We get a 200 status code response and its content is the JSON we sent it. Time to pack it up and call it a day. Just kidding.

Create the Programmer

Now that we've got the JSON, creating a Programmer is ridiculously simple. First, json_decode the $body into an array:

33 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 17
    public function newAction(Request $request)
    {
        $data = json_decode($request->getContent(), true);
... lines 21 - 30
    }
... lines 32 - 33
```

For now, we'll trust the JSON string has a valid structure. And the second argument to json_decode makes sure we get an array, not a stdClass object.

Now for the most obvious code you'll see $programmer = new Programmer(), and pass it $data['nickname'] and $data['avatarNumber'] - I gave this entity class a __construct() function with a few optional arguments. Now, $programmer->setTagLine($data['tagLine']):

33 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 17
18      public function newAction(Request $request)
19      {
20          $data = json_decode($request->getContent(), true);
21
22          $programmer = new Programmer($data['nickname'], $data['avatarNumber']);
23          $programmer->setTagLine($data['tagLine']);
... lines 24 - 30
31      }
... lines 32 - 33
```

The only tricky part is that the Programmer has a relationship to the User that created it, and this is a required relationship. On the web, I'm logged in, so that controller sets my User object on this when I create a Programmer. But our API doesn't have any authentication yet - it's all anonymous.

We'll add authentication later. Right now, we need a workaround. Update the controller to extend BaseController - that's something *I* created right in AppBundle/Controller that just has some handy shortcut methods. This will let me say $programmer->setUser($this->findUserByUsername('weaverryan')):

33 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 4
5   use AppBundle\Controller\BaseController;
... lines 6 - 11
12  class ProgrammerController extends BaseController
13  {
... lines 14 - 17
18      public function newAction(Request $request)
19      {
20          $data = json_decode($request->getContent(), true);
21
22          $programmer = new Programmer($data['nickname'], $data['avatarNumber']);
23          $programmer->setTagLine($data['tagLine']);
24          $programmer->setUser($this->findUserByUsername('weaverryan'));
... lines 25 - 30
31      }
32  }
```

So we're cheating big time... for now. At least while developing, that user exists because it's in our fixtures. I'm not proud of this, but I promise it'll get fixed later.

Finish things off by persisting and flushing the Programmer:

33 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 17
18      public function newAction(Request $request)
19      {
... lines 20 - 23
24          $programmer->setUser($this->findUserByUsername('weaverryan'));
25
26          $em = $this->getDoctrine()->getManager();
27          $em->persist($programmer);
28          $em->flush();
... lines 29 - 30
31      }
... lines 32 - 33
```

Enjoy this easy stuff... while it lasts. For the Response, what should we return? Ah, let's worry about that later - return a reassuring message, like It worked. Believe me, I'm an API!:

33 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 17
18      public function newAction(Request $request)
19      {
... lines 20 - 27
28          $em->flush();
29
30          return new Response('It worked. Believe me - I\'m an API');
31      }
... lines 32 - 33
```

The whole flow is there, so go back and hit the script again:

```
$ php testing.php
```

And... well, I think it looks like that probably worked. Now, let's add a form.

Chapter 3: Handling data with a Form

So what's different between this API controller and one that handles an HTML form submit? Really, not much. The biggest difference is that an HTML form sends us POST parameters and an API sends us a JSON string. But once we decode the JSON, both give us an array of submitted data. Then, everything is the same: create a Programmer object and update it with the submitted data. And you know who does this kind of work really well? Bernhard Schussek err Symfony forms!

Create a new directory called Form/ and inside of that, a new class called ProgrammerType. I'll quickly make this into a form type by extending AbstractType and implementing the getName() method - just return, how about, programmer.

Now, override the two methods we *really* care about - setDefaultOptions() and buildForm():

43 lines | src/AppBundle/Form/ProgrammerType.php

```
<?php

namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ProgrammerType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 13 - 29
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
... lines 34 - 36
    }

    public function getName()
    {
        return 'programmer';
    }
}
```

In Symfony 2.7, setDefaultOptions() is called configureOptions() - so adjust that if you need to.

In setDefaultOptions, the one thing we want to do is $resolver->setDefaults() and make sure the data_class is set so this form will definitely give us an AppBundle\Entity\Programmer object:

43 lines | src/AppBundle/Form/ProgrammerType.php

```
... lines 1 - 31
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Programmer'
        ));
    }
... lines 38 - 43
```

Building the Form

In build form, let's see here, let's build the form! Just like normal use $builder->add() - the first field is nickname and set it to a text type. The second field is avatarNumber. In this case, the value will be a number from 1 to 6. So we *could* use the number type. But instead, use choice. For the choices option, I'll paste in an array that goes from 1 to 6:

43 lines | src/AppBundle/Form/ProgrammerType.php

```
... lines 1 - 10
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nickname', 'text')
            ->add('avatarNumber', 'choice', [
                'choices' => [
                    // the key is the value that will be set
                    // the value/label isn't shown in an API, and could
                    // be set to anything
                    1 => 'Girl (green)',
                    2 => 'Boy',
                    3 => 'Cat',
                    4 => 'Boy with Hat',
                    5 => 'Happy Robot',
                    6 => 'Girl (purple)',
                ]
            ])
... line 28
        ;
    }
... lines 31 - 43
```

Using the choice Type in an API

Why choice instead of number or text? Because it has built-in validation. If the client acts a fool and sends something other than 1 through 6, validation will fail.

TIP To control this message, set the invalid_message option on the field.

For the API, we only care about the keys in that array: 1-6. The labels, like "Girl (green)", "Boy" and "Cat" are meaningless. For a web form, they'd show up as the text in the drop-down. But in an API, they do nothing and could be set to anything.

Finish with an easy field: tagLine and make it a textarea, which for an API, does the exact same thing as a text type:

43 lines | src/AppBundle/Form/ProgrammerType.php

```
... lines 1 - 10
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nickname', 'text')
            ->add('avatarNumber', 'choice', [
                'choices' => [
                    // the key is the value that will be set
                    // the value/label isn't shown in an API, and could
                    // be set to anything
                    1 => 'Girl (green)',
                    2 => 'Boy',
                    3 => 'Cat',
                    4 => 'Boy with Hat',
                    5 => 'Happy Robot',
                    6 => 'Girl (purple)',
                ]
            ])
            ->add('tagLine', 'textarea')
        ;
    }
... lines 31 - 43
```

So, there's our form. Can you tell this form is being used in an API? Nope! So yes, you *can* re-use forms for your API and web interface. Sharing is caring!

Using the Form

Back in the controller, let's use it! $form = $this->createForm() passing it a new ProgrammerType and the $programmer object. And now that the form is handling $data for us, get rid of the Programmer constructor arguments - they're optional anyways. Oh, and remove the setTagLine stuff, the form will do that for us too:

36 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 14
    /**
     * @Route("/api/programmers")
     * @Method("POST")
     */
    public function newAction(Request $request)
    {
        $data = json_decode($request->getContent(), true);

        $programmer = new Programmer();
        $form = $this->createForm(new ProgrammerType(), $programmer);
... lines 25 - 26
        $programmer->setUser($this->findUserByUsername('weaverryan'));

        $em = $this->getDoctrine()->getManager();
... lines 30 - 33
    }
... lines 35 - 36
```

Normally, this is when we'd call $form->handleRequest(). But instead, call $form->submit() and pass it the array of $data:

36 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 18
    public function newAction(Request $request)
    {
        $data = json_decode($request->getContent(), true);

        $programmer = new Programmer();
        $form = $this->createForm(new ProgrammerType(), $programmer);
        $form->submit($data);

        $programmer->setUser($this->findUserByUsername('weaverryan'));

        $em = $this->getDoctrine()->getManager();
... lines 30 - 33
    }
... lines 35 - 36
```

Ok, this is really cool because it turns out that when we call $form->handleRequest(), all *it* does is finds the form's POST parameters array and then passes that to $form->submit(). With $form->submit(), you're doing the same thing as normal, but working more directly with the form.

And that's all the code you need! So let's try it:

```
$ php testing.php
```

Yep! The server seems confident that still worked. That's all I need to hear!

Creating a Resource? 201 Status Code

On this create endpoint, there are 2 more things we need to do. First, whenever you create a resource, the status code should be 201:

36 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 14
    /**
     * @Route("/api/programmers")
     * @Method("POST")
     */
    public function newAction(Request $request)
    {
... lines 21 - 32
        return new Response('It worked. Believe me - I\'m an API', 201);
    }
... lines 35 - 36
```

That's our first non-200 status code and we'll see more as we go. Try that:

```
$ php testing.php
```

Cool - the 201 status code is hiding up top.

Creating a Resource? Location Header

Second, when you create a resource, best-practices say that you should set a Location header on the response. Set the new Response line to a $response variable and then add the header with $response->headers->set(). The value should be the URL to the new resource... buuuut we don't have an endpoint to view one Programmer yet, so let's fake it:

39 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 18
19      public function newAction(Request $request)
20      {
... lines 21 - 32
33          $response = new Response('It worked. Believe me - I\'m an API', 201);
34          $response->headers->set('Location', '/some/programmer/url');
35
36          return $response;
37      }
... lines 38 - 39
```

We'll fix it soon, I promise! Don't forget to return the $response.

Try it once more:

```
$ php testing.php
```

Just like butter, we're on a roll!

Chapter 4: GET one Programmer

Creating a programmer check! Next let's add the endpoint to return a single programmer. Add a public function showAction(). And even though it *technically* could be anything we dream up, the URL for this will follow a predictable pattern: /api/programmers/ and then some identifier. This might be an {id}, but for us each programmer has a unique nickname, so we'll use that instead. Don't forget the @Method("GET"):

48 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 38
39      /**
40       * @Route("/api/programmers/{nickname}")
41       * @Method("GET")
42       */
43      public function showAction($nickname)
44      {
... line 45
46      }
... lines 47 - 48
```

A client will use this to GET this programmer resource.

Add the $nickname argument and kick things off just by returning a new Response that says hi to our programmer:

48 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 38
39      /**
40       * @Route("/api/programmers/{nickname}")
41       * @Method("GET")
42       */
43      public function showAction($nickname)
44      {
45          return new Response('Hello '.$nickname);
46      }
... lines 47 - 48
```

Testing this Endpoint

Every time we create a new endpoint, we're going to add a new test. Ok, we don't *really* have tests yet, but we will soon! Right now, go back to testing.php and make a second request. The first is a POST to create the programmer, and the second is a GET to fetch that programmer.

Change the method to get() and the URI will be /api/programmers/ plus the random $nickname variable. And we don't need to send any request body:

29 lines | testing.php

```
... lines 1 - 11
$nickname = 'ObjectOrienter'.rand(0, 999);
... lines 13 - 17

// 1) Create a programmer resource
$response = $client->post('/api/programmers', [
    'body' => json_encode($data)
]);

// 2) GET a programmer resource
$response = $client->get('/api/programmers/'.$nickname);

echo $response;
echo "\n\n";
```

Alright, let's try it!

```
php testing.php
```

Well Hello ObjectOrienter564 and Hello also ObjectOrienter227. Nice to meet both of you.

Returning the Programmer in JSON

Instead of saying Hello, it would probably be more helpful if we sent the Programmer back in JSON. So let's get to work. First, we need to query for the Programmer: $this->getDoctrine()->getRepository('AppBundle:Programer') and I already have a custom method in there called findOneByNickname():

59 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 38
    /**
     * @Route("/api/programmers/{nickname}")
     * @Method("GET")
     */
    public function showAction($nickname)
    {
        $programmer = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
... lines 47 - 56
    }
... lines 58 - 59
```

Don't worry about 404'ing on a bad nickname yet. To turn the entity object into JSON, we'll eventually use a tool called a serializer. But for now, keep it simple: create an array and manually populate it: a nickname key set to $programmer->getNickname() and avatarNumber => $programmer->getAvatarNumber(). Also set a powerLevel key - that's the energy you get or lose when powering up - and finish with the tagLine:

59 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 44
        $programmer = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
            ->findOneByNickname($nickname);

        $data = array(
            'nickname' => $programmer->getNickname(),
            'avatarNumber' => $programmer->getAvatarNumber(),
            'powerLevel' => $programmer->getPowerLevel(),
            'tagLine' => $programmer->getTagLine(),
        );
... lines 55 - 59
```

Return whatever fields you want to: it's your API, but consistency is king.

Now all we need to do is json_encode that and give it to the Response:

58 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 41
    public function showAction($nickname)
    {
... lines 44 - 47
        $data = array(
            'nickname' => $programmer->getNickname(),
            'avatarNumber' => $programmer->getAvatarNumber(),
            'powerLevel' => $programmer->getPowerLevel(),
            'tagLine' => $programmer->getTagLine(),
        );

        return new Response(json_encode($data), 200);
    }
... lines 57 - 58
```

Our tools *will* get more sophisticated and fun than this. But keep this simple controller in mind: if things get tough, you can always back up to creating an array manually and json_encoding the results.

Let's try the whole thing:

```
$ php testing.php
```

Hey, nice JSON.

Chapter 5: Tightening up the Response

This endpoint is missing two teeny-tiny details.

Setting Content-Type: application/json

First, we're returning JSON, but the Response Content-Type is still advertising that we're returning text/html. That's a bummer, and will probably confuse some clients, like jQuery's AJAX function.

It's easy to fix anyways: set new Response to a $response variable like we did earlier and call $response->headers->set() with Content-Type and application/json:

61 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 41
    public function showAction($nickname)
    {
... lines 44 - 54
        $response = new Response(json_encode($data), 200);
        $response->headers->set('Content-Type', 'application/json');

        return $response;
    }
... lines 60 - 61
```

Check out the new Content-Type header:

```
$ php testing.php
```

404'ing

The second teeny-tiny thing we're missing is a 404 on a bad $nickname. Just treat this like a normal controller - so if (!$programmer), then throw $this->createNotFoundException(). And we might as well give ourselves a nice message:

68 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 41
    public function showAction($nickname)
    {
        $programmer = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
            ->findOneByNickname($nickname);

        if (!$programmer) {
            throw $this->createNotFoundException(sprintf(
                'No programmer found with nickname "%s"',
                $nickname
            ));
        }
... lines 54 - 65
    }
... lines 67 - 68
```

Use a fake nickname in testing.php temporarily to try this:

29 lines | testing.php

```
... lines 1 - 23
// 2) GET a programmer resource
$response = $client->get('/api/programmers/abcd'.$nickname);

echo $response;
echo "\n\n";
```

Then re-run:

```
$ php testing.php
```

Woh! That exploded! This is Symfony's HTML exception page. It *is* our 404 error, but it's in HTML instead of JSON. Why? Internally, Symfony has a request format, which defaults to html. If you change that to json, you'll get JSON errors. If you're curious about this, google for Symfony request _format.

But I'll show you this later in the series. And we'll go one step further to *completely* control the format of our errors. And it will be awesome.

Change the URL in testing.php back to the real nickname.

Setting the Location Header

Ok, remember that fake Location header on the POST endpoint? Good news! We can get rid of that fake URL.

First, give the GET endpoint route a name - api_programmers_show:

73 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 42
    /**
     * @Route("/api/programmers/{nickname}", name="api_programmers_show")
     * @Method("GET")
     */
    public function showAction($nickname)
... lines 48 - 73
```

Copy that, call $this->generateUrl(), pass it api_programmers_show and the array with the nickname key set to the nickname of this new Programmer. Then just set this on the Location header... instead of our invented URL:

72 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 18
    public function newAction(Request $request)
    {
... lines 21 - 32
        $response = new Response('It worked. Believe me - I\'m an API', 201);
        $programmerUrl = $this->generateUrl(
            'api_programmers_show',
            ['nickname' => $programmer->getNickname()]
        );
        $response->headers->set('Location', $programmerUrl);

        return $response;
    }
... lines 42 - 72
```

Why are we doing this again? Just because it might be helpful to your client to have the address to the new resource. That would be especially true if you used an auto-increment id that the server just determined.

To try this in testing.php, copy the echo $response stuff, put it below the first $response, then let's die:

33 lines | testing.php

```
... lines 1 - 18
// 1) Create a programmer resource
$response = $client->post('/api/programmers', [
    'body' => json_encode($data)
]);

echo $response;
echo "\n\n";
die;
... lines 27 - 33
```

Now, try php testing.php:

```
$ php testing.php
```

Now we have a *really* clean Location header we could use to fetch or edit that Programmer.

Use the Location Header

Heck, we can even use this and get rid of the hardcoded URL in testing.php. Set $programmerUrl to $response->getHeader('Location'). Pop that in to the next get() call:

31 lines | testing.php

```
... lines 1 - 23
$programmerUrl = $response->getHeader('Location');

// 2) GET a programmer resource
$response = $client->get($programmerUrl);

echo $response;
echo "\n\n";
```

I like that! When you're testing your API, you're really eating your own dog food. And that's a perfect time to think about the user-experience of getting work done with it.

Try it one last time:

```
$ php testing.php
```

That looks great!

Chapter 6: GET a Collection of Programmers

Our API client will need a way to fetch a collection of programmers. Piece of cake. Start with public function listAction(). For the URI, use /api/programmers and add an @Method("GET"):

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 67
68      /**
69       * @Route("/api/programmers")
70       * @Method("GET")
71       */
72      public function listAction()
73      {
... lines 74 - 85
86      }
... lines 87 - 98
```

So, the URI you POST to when creating a resource - /api/programmers - will be the same as the one you'll GET to fetch a collection of programmer resources. And yes, you can filter and paginate this list - all stuff we'll do later on.

Inside listAction(), start like we always do: with a query.
$programmers = $this->getDoctrine()->getRepository('AppBundle:Programmer') and for now, we'll find everything with findAll():

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 71
72      public function listAction()
73      {
74          $programmers = $this->getDoctrine()
75              ->getRepository('AppBundle:Programmer')
76              ->findAll();
... lines 77 - 85
86      }
... lines 87 - 98
```

Next, we need to transform this array of Programmers into JSON. We'll want to re-use some logic from before, so let's create a new private function called serializeProgrammer() and add a Programmer argument. Inside, we can steal the manual logic that turns a Programmer into an array and just return it:

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 87
88      private function serializeProgrammer(Programmer $programmer)
89      {
90          return array(
91              'nickname' => $programmer->getNickname(),
92              'avatarNumber' => $programmer->getAvatarNumber(),
93              'powerLevel' => $programmer->getPowerLevel(),
94              'tagLine' => $programmer->getTagLine(),
95          );
96      }
... lines 97 - 98
```

That's a small improvement - at least we can re-use this stuff from inside this controller. In showAction(), use

$this->serializeProgrammer() and pass it the variable.

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 46
47      public function showAction($nickname)
48      {
... lines 49 - 59
60          $data = $this->serializeProgrammer($programmer);
61
62          $response = new Response(json_encode($data), 200);
... lines 63 - 65
66      }
... lines 67 - 98
```

Back in listAction(), we'll need to loop over the Programmers and serialize them one-by-one. So start by creating a $data array with a programmers key that's also set to an empty array. We're going to put the programmers there:

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 71
72      public function listAction()
73      {
74          $programmers = $this->getDoctrine()
75              ->getRepository('AppBundle:Programmer')
76              ->findAll();
77          $data = array('programmers' => array());
... lines 78 - 98
```

Avoid JSON Hijacking

Why? You can structure your JSON however you want, but by putting the collection inside a key, we have room for more root keys later like maybe count or offset for pagination. Second, your outer JSON should always be an object, not an array. So, curly braces instead of square brackets. If you have square brackets, you're vulnerable to something called JSON Hijacking. While not as bad as a car hijacking is something you want to avoid.

Turning the Programmers into JSON

Loop through $programmers, and one-by-one, say $data['programmers'][] and push on $this->serializeProgrammer(). The end is the same as showAction(), so just copy that:

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 71
72      public function listAction()
73      {
74          $programmers = $this->getDoctrine()
75              ->getRepository('AppBundle:Programmer')
76              ->findAll();
77          $data = array('programmers' => array());
78          foreach ($programmers as $programmer) {
79              $data['programmers'][] = $this->serializeProgrammer($programmer);
80          }
81
82          $response = new Response(json_encode($data), 200);
83          $response->headers->set('Content-Type', 'application/json');
84
85          return $response;
86      }
... lines 87 - 98
```

That ought to do it. Update testing.php to make another call out to /api/programmers. Let's see what that looks like:

```
$ php testing.php
```

Woh, ok! We've got a database full of programmers. NERDS! Creating a new endpoint is getting easier - that trend will continue.

Returning JSON on Create

Remember how we're returning a super-reassuring text message from our POST endpoint? Well, you *can* do this, but usually, you'll return the resource you just created. That's easy now - so let's do it. Just, $data = $this->serializeProgrammer(). Then json_encode() that in the Response. And don't forget to set the Content-Type header:

100 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 18
19      public function newAction(Request $request)
20      {
... lines 21 - 32
33          $data = $this->serializeProgrammer($programmer);
34          $response = new Response(json_encode($data), 201);
... lines 35 - 39
40          $response->headers->set('Content-Type', 'application/json');
41
42          return $response;
43      }
... lines 44 - 100
```

To see if this is working, dump again right after the first request:

35 lines | testing.php

```
... lines 1 - 18
19  // 1) Create a programmer resource
20  $response = $client->post('/api/programmers', [
21      'body' => json_encode($data)
22  ]);
23  echo $response;
24  echo "\n\n";die;
... lines 25 - 35
```

Hit it!

```
$ php testing.php
```

That's a helpful response: it has a Location header *and* shows you the resource immediately. Because, why not?

JsonResponse

We're about to move onto testing - which makes this all *so* much fun. But first, we can shorten things. Each endpoint json_encodes the data *and* sets the Content-Type header. Use a class called JsonResponse instead. And instead of passing it the JSON string, just pass it the data array. The other nice thing is that you don't need to set the Content-Type header, it does that for you:

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 19
20      public function newAction(Request $request)
21      {
... lines 22 - 33
34          $data = $this->serializeProgrammer($programmer);
35          $response = new JsonResponse($data, 201);
... lines 36 - 41
42          return $response;
43      }
... lines 44 - 98
```

API consistency is king, and this is just one less spot for me to mess up and forget to set that header. Make sure you update the other spots, which is pretty much a copy-and-paste operation - be careful to keep the right status code.

Take out the extra die() statement in testing.php and let's try this *whole* thing out:

```
$ php testing.php
```

It's lovely. To make sure it doesn't break, we need to add tests! And that will be a whole lot more interesting than you think.

Chapter 7: Add a Test!

This testing.php file is basically already a test... except it's missing the most important part: the ability to start shouting when something breaks.

To test our API, we'll use PHPUnit! Yes! Awesome! I'm excited because even though PHPUnit isn't the most exciting tool, it's solid - and we're going to do some cool stuff with our tests.

TIP In our other REST tutorial, we tested with Behat. Both are great, and really the same under the surface.

Create that Test

Create a Tests directory inside AppBundle. Now mimic your directory structure. So, add a Controller directory, then an API directory, and finish it with a new PHPUnit test class for ProgrammerController. Be a good programmer and fill in the right namespace. All these directories: technically unnecessary. But now we've got a sane setup.

Of course, we'll test our POST endpoint - so create public function testPOST():

28 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
<?php
namespace AppBundle\Tests\Controller\Api;

class ProgrammerControllerTest extends \PHPUnit_Framework_TestCase
{
    public function testPOST()
    {
... lines 8 - 25
    }
}
```

I'm being inconsistent - the controller is newAction, but this method is testPOST - it would be cool to have these match - maybe even with a mixture of the two - like postNewAction().

Anyways, let's go steal our first request code from testing.php and paste it into testPOST:

28 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 5
    public function testPOST()
    {
        $client = new \GuzzleHttp\Client([
            'base_url' => 'http://localhost:8000',
            'defaults' => [
                'exceptions' => false
            ]
        ]);

        $nickname = 'ObjectOrienter'.rand(0, 999);
        $data = array(
            'nickname' => $nickname,
            'avatarNumber' => 5,
            'tagLine' => 'a test dev!'
        );

        // 1) Create a programmer resource
        $response = $client->post('/api/programmers', [
            'body' => json_encode($data)
        ]);
    }
... lines 27 - 28
```

Ok cool. No asserts yet - but let's see if it blows up. I already installed PHPUnit into this project, so run php bin/phpunit -c app then the path to the test:

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Pretty green! No assertions yet, but also no explosions. Solid start team!

Be Assertive

Ok, what should we assert? Always start with the status code - $this->assertEquals() that the expected 201 equals $response->getStatusCode():

33 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 21
        // 1) Create a programmer resource
        $response = $client->post('/api/programmers', [
            'body' => json_encode($data)
        ]);

        $this->assertEquals(201, $response->getStatusCode());
... lines 28 - 33
```

Second: what response header should we send back whenever we create a resource? Location! Right now, just assertTrue that $response->hasHeader('Location'). Soon, we'll assert the actual value.

33 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 26
        $this->assertEquals(201, $response->getStatusCode());
        $this->assertTrue($response->hasHeader('Location'));
... lines 29 - 33
```

And to put a bow on things, let's json_decode the response body into an array, and just assert that is *has* a nickname key

with assertArrayHasKey, with nickname and $data:

33 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 26
        $this->assertEquals(201, $response->getStatusCode());
        $this->assertTrue($response->hasHeader('Location'));
        $finishedData = json_decode($response->getBody(true), true);
        $this->assertArrayHasKey('nickname', $finishedData);
... lines 31 - 33
```

In a second, we'll assert the actual value. It's not a super-tight test yet, but let's give it a shot:

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Yes! This time we deserve that green.

Chapter 8: Test Code Reuse

We'll have a bunch of test classes and they'll all need to create a Guzzle Client with these options. So let's just get organized now.

Create a new Test directory in the bundle and a new class called ApiTestCase. This will be a base class for all our API tests. Make *it* extend the normal PHPUnit_Framework_TestCase:

30 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 2
namespace AppBundle\Test;
... lines 4 - 6
class ApiTestCase extends \PHPUnit_Framework_TestCase
{
... lines 9 - 29
}
```

Right now, the thing I want to move *out* of each test class is the creation of the Guzzle Client. So copy that code. In ApiTestCase, override a method called setupBeforeClass() - it's static. PHPUnit calls this *one* time at the beginning of running your whole test suite.

Paste the $client code here. Because really, even if we run A LOT of tests, we can probably always use the same Guzzle client. Create a private static property called $staticClient and put the Client there with self::$staticClient. And give Client a proper use statement:

30 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 6
class ApiTestCase extends \PHPUnit_Framework_TestCase
{
    private static $staticClient;
... lines 10 - 15
    public static function setUpBeforeClass()
    {
        self::$staticClient = new Client([
            'base_url' => 'http://localhost:8000',
            'defaults' => [
                'exceptions' => false
            ]
        ]);
    }
... lines 25 - 29
}
```

Tip

In case you are using Guzzle 6, you would need to use the base_uri key instead of base_url to configure Guzzle client properly.

Cool. So now the Client is created once per test suite. Now, create a protected $client property that is *not* static with some nice PHPDoc above it. Woops - make sure you actually make this protected: this is what we'll use in the sub-classes. Then, override setup() and say $this->client = self::$staticClient:

30 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 6
class ApiTestCase extends \PHPUnit_Framework_TestCase
{
... lines 9 - 10
    /**
     * @var Client
     */
    protected $client;
... lines 15 - 25
    protected function setUp()
    {
        $this->client = self::$staticClient;
    }
}
```

setupBeforeClass() will make sure the Client is created just once and setup() puts that onto a non-static property, just because I like non-static things a bit better. Oh, and if we *did* need to do any clean up resetting of the Client, we could do that in setup() or tearDown().

Extend the Base Class

Back in the actual test class, get rid of the $client code and simply reference $this->client. Ooooo, and don't forget to extend ApiTestCase like I just did:

28 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 3
use AppBundle\Test\ApiTestCase;

class ProgrammerControllerTest extends ApiTestCase
{
    public function testPOST()
    {
... lines 10 - 16
        // 1) Create a programmer resource
        $response = $this->client->post('/api/programmers', [
            'body' => json_encode($data)
        ]);
... lines 21 - 25
    }
}
```

Make sure we didn't break anything:

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Hey, still green!

Chapter 9: Tests with the Container

Using a random nickname in a test is weird: we should be explicit about our input and output. Just set it to ObjectOrienter. Now it's easy to make our asserts more specific, like for the Location header using assertEquals, which should be /api/programmers/ObjectOrienter. And now use the method getHeader():

29 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8       public function testPOST()
9       {
10          $data = array(
11              'nickname' => 'ObjectOrienter',
12              'avatarNumber' => 5,
13              'tagLine' => 'a test dev!'
14          );
... lines 15 - 22
23          $this->assertEquals('/api/programmers/ObjectOrienter', $response->getHeader('Location'));
... lines 24 - 26
27      }
... lines 28 - 29
```

And at the bottom, assertArrayHasKey is good, but we really want to say assertEquals() to really check that the nickname key coming back is set to ObjectOrienter:

29 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 24
25          $this->assertArrayHasKey('nickname', $finishedData);
26          $this->assertEquals('ObjectOrienter', $finishedData['nickname']);
... lines 27 - 29
```

This test makes me happier. But does it pass? Run it!

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Sawheet! All green. Untilllllll you try it again:

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Now it explodes - 500 status code and we can't even see the error. But I know it's happening because nickname is unique in the database, and now we've got the nerve to try to create a second ObjectOrienter.

Booting the Container

Ok, we've gotta take control of the stuff in our database - like by clearing everything out before each test.

If we had the EntityManager object, we could use it to help get that done. So, let's boot the framework right inside ApiTestCase. But not to make any requests, just so we can get the container and use our services.

Symfony has a helpful way to do this - it's a base class called KernelTestCase:

56 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 6
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;

class ApiTestCase extends KernelTestCase
{
... lines 11 - 55
}
```

Inside setupBeforeClass(), say self::bootKernel():

56 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 17
    public static function setUpBeforeClass()
    {
... lines 20 - 26
        self::bootKernel();
    }
... lines 29 - 56
```

The kernel is the heart of Symfony, and booting it basically just makes the service container available.

Add the tearDown() method... and do nothing. What!? This is important. I'm adding a comment about why - I'll explain in a second:

56 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 36
    /**
     * Clean up Kernel usage in this test.
     */
    protected function tearDown()
    {
        // purposefully not calling parent class, which shuts down the kernel
    }
... lines 44 - 56
```

But first, create a private function getService() with an $id argument. Woops - make that protected - the whole point of this method is to let our test classes fetch services from the container. To do that, return self::$kernel->getContainer()->get($id):

56 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 50
    protected function getService($id)
    {
        return self::$kernel->getContainer()
            ->get($id);
    }
```

The whole point of that KernelTestCase base class is to set and boot that static $kernel property which has the container on it. Now normally, the base class actually shuts down the kernel in tearDown(). What I'm doing - on purpose - is booting the kernel and creating the container just once per my whole test suite.

That'll make things faster, though in theory it could cause issues or even slow things down eventually. You can experiment by shutting down your kernel in tearDown() and booting it in setup() if you want. Or even just clearing the EntityManager to avoid a lot of entities getting stuck inside of it after a bunch of tests.

Clearing Data

Because we have the container, we have the EntityManager. And that also means we have an easy way to clear data.

Create a new private function called purgeDatabase(). Because we have the Doctrine DataFixtures library installed, we can use a great class called ORMPurger. Pass it the EntityManager - so $this->getService('doctrine')->getManager(). To clear things out, say $purger->purge():

56 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 4
use Doctrine\Common\DataFixtures\Purger\ORMPurger;
... lines 6 - 8
class ApiTestCase extends KernelTestCase
{
... lines 11 - 44
    private function purgeDatabase()
    {
        $purger = new ORMPurger($this->getService('doctrine')->getManager());
        $purger->purge();
    }
... lines 50 - 55
}
```

Now we just need to call this before every test - so calling this in setup() is the perfect spot - $this->purgeDatabase():

56 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 29
    protected function setUp()
    {
        $this->client = self::$staticClient;

        $this->purgeDatabase();
    }
... lines 36 - 56
```

This should clear the ObjectOrienter out of the database and hopefully get things passing. Try the test!

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Drumroll! Oh no - still a 500 error. And we still can't see the error. Time to take our debugging tools up a level.

Chapter 10: Mad Test Debugging

When we mess up in a web app, we see Symfony's giant exception page. I want that same experience when I'm building an API.

At the root of the project there's a resources/ directory with an ApiTestCase.php file. This has all the same stuff as *our* ApiTestCase plus some pretty sweet new debugging stuff.

Copy this and paste it over our class.

First, check out onNotSuccessfulTest():

205 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 64
    protected function onNotSuccessfulTest(Exception $e)
    {
        if (self::$history && $lastResponse = self::$history->getLastResponse()) {
            $this->printDebug('');
            $this->printDebug('<error>Failure!</error> when making the following request:');
            $this->printLastRequestUrl();
            $this->printDebug('');

            $this->debugResponse($lastResponse);
        }

        throw $e;
    }
... lines 78 - 205
```

If you have a method with this name, PHPUnit calls it whenever a test fails. I'm using it to print out the last response so we can see what just happened.

I also added a few other nice things, like printLastRequestUrl().

200 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 90
    protected function printLastRequestUrl()
    {
        $lastRequest = self::$history->getLastRequest();

        if ($lastRequest) {
            $this->printDebug(sprintf('<comment>%s</comment>: <info>%s</info>', $lastRequest->getMethod(), $lastRequest->getUrl()));
        } else {
            $this->printDebug('No request was made.');
        }
    }
... lines 101 - 200
```

Next up is debugResponse() use it if you want to see what a Response looks like:

200 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 101
102     protected function debugResponse(ResponseInterface $response)
103     {
104         $this->printDebug(AbstractMessage::getStartLineAndHeaders($response));
105         $body = (string) $response->getBody();
... lines 106 - 172
173     }
... lines 174 - 200
```

This crazy function is something I wrote - it knows what Symfony's error page looks like and tries to extract the important parts... so you don't have to stare at a giant HTML page in your terminal. I hate that. It's probably not perfect - and if you find an improvement and want to share it, you'll be my best friend.

And finally, whenever this class prints something, it's calling printDebug(). And right now, it's about as dull as you can get:

200 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 179
180     protected function printDebug($string)
181     {
182         echo $string."\n";
183     }
... lines 184 - 200
```

I think we can make that way cooler. But first, with this in place, it *should* print out the last response so we can see the error:

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Ah hah!

```
Catchable Fatal Error: Argument 1 passed to Programmer::setUser() must
be an instance of AppBundle\Entity\User, null given in ProgrammerController.php
on line 29.
```

So the problem is that when we delete our database, we're also deleting our hacked-in weaverryan user:

33 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 17
18     public function newAction(Request $request)
19     {
... lines 20 - 23
24         $programmer->setUser($this->findUserByUsername('weaverryan'));
... lines 25 - 30
31     }
... lines 32 - 33
```

Let's deal with that in a second - and do something cool first. So, remember how some of the app/console commands have really pretty colored text when they print? Well, we're not inside a console command in PHPUnit, but I'd *love* to be able to print out with colors.

Good news! It turns out, this is really easy. The class that handles the styling is called ConsoleOutput, and you can use it directly from anywhere.

Start by adding a private $output property that we'll use to avoid creating a bunch of these objects. Then down in printDebug(), say if ($this->output === null) then $this->output = new ConsoleOutput();. This is the $output variable you're passed in a normal Symfony command. This means we can say $this->output->writeln() and pass it the $string:

209 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 12
use Symfony\Component\Console\Output\ConsoleOutput;
... lines 14 - 15
class ApiTestCase extends KernelTestCase
{
... lines 18 - 29
    /**
     * @var ConsoleOutput
     */
    private $output;
... lines 34 - 184
    protected function printDebug($string)
    {
        if ($this->output === null) {
            $this->output = new ConsoleOutput();
        }

        $this->output->writeln($string);
    }
... lines 193 - 207
}
```

I'm coloring some things already, so let's see this beautiful art! Re-run the test:

```
$ php bin/phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Hey! That error is hard to miss!

Seeing the Exception Stacktrace!

Ok, *one* more debugging trick. What if we really need to see the full stacktrace? The response headers are printed on top - and one of those actually holds the profiler URL for this request. And to be even nicer, my debug code is printing that at the bottom too.

Pop that into the browser. This is the profiler for that API request. It has cool stuff like the database queries, but most importantly, there's an Exception tab - you can see the full, beautiful exception with stacktrace. This is huge.

Chapter 11: Test Fixtures and the PropertyAccess Component

Howdy big error! Now that I can see you, I can fix you! Remember, back in ProgrammerController, we're *always* assuming there's a weaverryan user in the database:

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 19
20      public function newAction(Request $request)
21      {
... lines 22 - 25
26          $form->submit($data);
27
28          $programmer->setUser($this->findUserByUsername('weaverryan'));
29
30          $em = $this->getDoctrine()->getManager();
31          $em->persist($programmer);
32          $em->flush();
... lines 33 - 42
43      }
... lines 44 - 98
```

We'll fix this later with some proper authentication, but for now, when we run our tests, we need to make sure that user is cozy and snug in the database.

Creating a test User

Create a new protected function called createUser() with a required username argument and one for plainPassword. Make that one optional: in this case, we don't care what the user's password will be:

I'll paste in some code for this: it's pretty easy stuff. I'll trigger autocomplete on the User class to get PhpStorm to add that use statement for me. This creates the User and gives it the required data. The getService() function we created lets us fetch the password encoder out so we can use it, what a wonderfully trained function:

259 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 212
213     protected function createUser($username, $plainPassword = 'foo')
214     {
215         $user = new User();
216         $user->setUsername($username);
217         $user->setEmail($username.'@foo.com');
218         $password = $this->getService('security.password_encoder')
219             ->encodePassword($user, $plainPassword);
220         $user->setPassword($password);
... lines 221 - 226
227     }
... lines 228 - 259
```

Let's save this! Since we'll need the EntityManager a lot in this class, let's add a protected function getEntityManager(). Use getService() with doctrine.orm.entity_manager. And since I *love* autocomplete, give this PHPDoc:

235 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 226
    /**
     * @return EntityManager
     */
    protected function getEntityManager()
    {
        return $this->getService('doctrine.orm.entity_manager');
    }
... lines 234 - 235
```

Now $this->getEntityManager()->persist() and $this->getEntityManager()->flush(). And just in case whoever calls this needs the User, let's return it.

235 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 210
    protected function createUser($username, $plainPassword = 'foo')
    {
... lines 213 - 219
        $em = $this->getEntityManager();
        $em->persist($user);
        $em->flush();

        return $user;
    }
... lines 226 - 235
```

We could just go to the top of testPOST and call this there. But really, our entire system is kind of dependent on this user. So to truly fix this, let's put it in setup(). Don't forget to call parent::setup() - we've got some awesome code there. Then, $this->createUser('weaverryan'):

36 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
    protected function setUp()
    {
        parent::setUp();

        $this->createUser('weaverryan');
    }
... lines 14 - 34
}
```

I'd say we've earned a greener test - let's try it!

```
$ phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Yay!

Testing GET one Programmer

Now, let's test the GET programmer endpoint:

54 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 35
    public function testGETProgrammer()
    {
... lines 38 - 51
    }
}
```

Hmm, so we have another data problem: before we make a request to fetch a single programmer, we need to make sure there's one in the database.

To do that, call out to an imaginary function createProgrammer() that we'll write in a second. This will let us pass in an array of whatever fields we want to set on that Programmer:

54 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 35
    public function testGETProgrammer()
    {
        $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
... lines 42 - 51
    }
... lines 53 - 54
```

The Programmer class has a few other fields and the idea is that if we don't pass something here, createProgrammer() will invent some clever default for us.

Let's get to work in ApiTestCase: protected function createProgrammer() with an array of $data as the argument. And as promised, our first job is to use array_merge() to pass in some default values. One is the powerLevel - it's required - and if it's not set, give it a random value from 0 to 10. Next, create the Programmer:

258 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 228
    protected function createProgrammer(array $data)
    {
        $data = array_merge(array(
            'powerLevel' => rand(0, 10),
... lines 233 - 235
        ), $data);
... lines 237 - 238
        $programmer = new Programmer();
... lines 240 - 247
    }
... lines 249 - 258
```

Ok, maybe you're expecting me to iterate over the data, put the string set before each property name, and call that method. But no! There's a better way.

Getting down with PropertyAccess

Create an $accessor variable that's set to ProperyAccess::createPropertyAccessor(). Hello Symfony's PropertyAccess component! *Now* iterate over data. And instead of the "set" idea, call $accessor->setValue(), pass in $programmer, passing $key - which is the property name - and pass in the $value we want to set:

258 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 228
229     protected function createProgrammer(array $data)
230     {
... lines 231 - 237
238         $accessor = PropertyAccess::createPropertyAccessor();
239         $programmer = new Programmer();
240         foreach ($data as $key => $value) {
241             $accessor->setValue($programmer, $key, $value);
242         }
... lines 243 - 247
248     }
... lines 249 - 258
```

The PropertyAccess component is what works behind the scenes with Symfony's Form component. So, it's great at calling getters and setters, but it also has some *really* cool superpowers that we'll need soon.

The Programmer has all the data it needs, *except* for this $user relationship property. To set that, we can just add user to the defaults and query for one. I'll paste in a few lines here: I already setup our UserRepository to have a findAny() method on it:

258 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 228
229     protected function createProgrammer(array $data)
230     {
231         $data = array_merge(array(
232             'powerLevel' => rand(0, 10),
233             'user' => $this->getEntityManager()
234                 ->getRepository('AppBundle:User')
235                 ->findAny()
236         ), $data);
... lines 237 - 247
248     }
... lines 249 - 258
```

And finally, the easy stuff! Persist and flush that Programmer. And return it too for good measure:

258 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 228
229     protected function createProgrammer(array $data)
230     {
... lines 231 - 242
243
244         $this->getEntityManager()->persist($programmer);
245         $this->getEntityManager()->flush();
246
247         return $programmer;
248     }
... lines 249 - 258
```

Finishing the GET Test

Phew! With that work done, finishing the test is easy. Make a GET request to /api/programmers/UnitTester. And as always, we want to start by asserting the status code:

54 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 35
    public function testGETProgrammer()
    {
        $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));

        $response = $this->client->get('/api/programmers/UnitTester');
        $this->assertEquals(200, $response->getStatusCode());
... lines 45 - 51
    }
... lines 53 - 54
```

I want to assert that we get the properties we expect. If you look in ProgrammerController, we're serializing 4 properties: nickname, avatarNumber, powerLevel and tagLine. To avoid humiliation let's assert that those actually exist.

I'll use an assertEquals() and put those property names as the first argument in a moment. For the second argument - the *actual* value - we can use array_keys() on the json decoded response body - which I'll cleverly call $data. Guzzle can decode the JSON for us if we call $response->json(). This gives us the decoded JSON and array_keys gives us the field names in it. Back in the first argument to assertEquals(), we'll fill in the fields: nickname, avatarNumber, powerLevel and tagLine - even if it's empty:

54 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 35
    public function testGETProgrammer()
    {
... lines 38 - 42
        $response = $this->client->get('/api/programmers/UnitTester');
        $this->assertEquals(200, $response->getStatusCode());
        $data = $response->json();
        $this->assertEquals(array(
            'nickname',
            'avatarNumber',
            'powerLevel',
            'tagLine'
        ), array_keys($data));
    }
... lines 53 - 54
```

Ok, time to test-drive this:

```
$ phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Great success! Now let's zero in and make our assertions a whole lot more ...assertive :)

Chapter 12: The ResponseAsserter!

In *every* test, we're going to decode the JSON response and then assert some stuff - like "does the nickname property exist?" or "is it equal to UnitTester?"

Find the resources directory at the root of your project. I'm switching my PhpStorm mode up here temporarily, because I marked this directory as "excluded" so that Storm wouldn't try to autocomplete from stuff in it. See that ResponseAsserter.php file? Yea, copy that - it's good stuff.

Paste it into the Test directory right next to ApiTestCase. And now I'll re-hide that resources folder in PhpStorm.

Hello ResponseAsserter! This class is really good at reading properties off of a JSON response:

181 lines | src/AppBundle/Test/ResponseAsserter.php

```
... lines 1 - 13
class ResponseAsserter extends \PHPUnit_Framework_Assert
{
... lines 16 - 179
}
```

We won't read through this now, but you should. It uses the same PropertyAccess component internally - and we'll use its superpowers through this.

Setting things up in ApiTestCase

To use this in ApiTestCase, create a new private property called $responseAsserter:

272 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 19
class ApiTestCase extends KernelTestCase
{
... lines 22 - 43
    private $responseAsserter;
... lines 45 - 270
}
```

And then *way* down at the bottom - make a protected function asserter(). We'll use the property to avoid making multiple asserters. So, if $this->responseAsserter === null then set that to a new ResponseAsserter(). Finish by returning it:

272 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 19
20  class ApiTestCase extends KernelTestCase
21  {
... lines 22 - 251
252     /**
253      * @return ResponseAsserter
254      */
255     protected function asserter()
256     {
257         if ($this->responseAsserter === null) {
258             $this->responseAsserter = new ResponseAsserter();
259         }
260
261         return $this->responseAsserter;
262     }
... lines 263 - 270
271 }
```

Assert!

Now let's use this! Instead of having Guzzle decode the JSON for us, we can just say $this->asserter()->responsePropertiesExist() and pass it the $response we want it to look at and the array of properties that should exist in its JSON:

72 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 35
36      public function testGETProgrammer()
37      {
... lines 38 - 42
43          $response = $this->client->get('/api/programmers/UnitTester');
44          $this->assertEquals(200, $response->getStatusCode());
45          $this->asserter()->assertResponsePropertiesExist($response, array(
46              'nickname',
47              'avatarNumber',
48              'powerLevel',
49              'tagLine'
50          ));
... line 51
52      }
... lines 53 - 72
```

That gets rid of a nice block of code. Inside the new function, it just loops over each property and reads their value using the PropertyAccess component. And it *is* still just using json_decode internally. It's just an easier way to look into the JSON response.

Since we're responsible coders, let's assert that it all works:

```
$ phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Excellent! Let's add one more - an assert that the nickname is set to UnitTester. Use assertResponsePropertyEquals() - always pass the $response first. Then, nickname and it should equal UnitTester:

72 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 35
36      public function testGETProgrammer()
37      {
... lines 38 - 50
51          $this->asserter()->assertResponsePropertyEquals($response, 'nickname', 'UnitTester');
52      }
... lines 53 - 72
```

Run that!

```
$ phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

And *that* passes. Nothing scares me more than when things are green on the first try, ... well that and snakes on a plane. So, let's assert UnitTester2 and see it fail.

```
$ phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Phew, ok good!

```
Property "nickname": Expected "UnitTester2" but response was "UnitTester"
```

And like *all* failures, it prints out the raw response above this.

Testing /api/programmers

Our testing setup is, well, pretty sweet. So testing the GET collection endpoint should be easy. Create a testGETProgrammersCollection() method. Grab the createProgrammer() code from above, but paste it twice to create a new CowboyCoder:

72 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 53
54      public function testGETProgrammersCollection()
55      {
56          $this->createProgrammer(array(
57              'nickname' => 'UnitTester',
58              'avatarNumber' => 3,
59          ));
60          $this->createProgrammer(array(
61              'nickname' => 'CowboyCoder',
62              'avatarNumber' => 5,
63          ));
... lines 64 - 69
70      }
... lines 71 - 72
```

Now grab the lines that makes the request and asserts the status code. Update the URL to just /api/programmers:

72 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 53
54      public function testGETProgrammersCollection()
55      {
... lines 56 - 64
65          $response = $this->client->get('/api/programmers');
66          $this->assertEquals(200, $response->getStatusCode());
... lines 67 - 69
70      }
... lines 71 - 72
```

No assertions yet, but let's make sure it doesn't blow up:

```
$ phpunit -c app src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Great! Ok, so what do we want to assert? I don't know, what do you want to assert? Think about how the endpoint works: we're returning an associative array with a programmers key and *that* actually holds the collection of programmers:

98 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 72
73      public function listAction()
74      {
... lines 75 - 77
78          $data = array('programmers' => array());
79          foreach ($programmers as $programmer) {
80              $data['programmers'][] = $this->serializeProgrammer($programmer);
81          }
82
83          $response = new JsonResponse($data, 200);
84
85          return $response;
86      }
... lines 87 - 98
```

Let's first assert that there's a programmers key in the response and that it's an array. Use $this->asserter()->assertResponsePropertyIsArray(): pass it the $response and the property: programmers. Next, let's assert that there are *two* things on this array. There's a method for that called assertResponsePropertyCount() - pass it the $response, programmers and the number 2:

72 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 53
54      public function testGETProgrammersCollection()
55      {
... lines 56 - 64
65          $response = $this->client->get('/api/programmers');
66          $this->assertEquals(200, $response->getStatusCode());
67          $this->asserter()->assertResponsePropertyIsArray($response, 'programmers');
68          $this->asserter()->assertResponsePropertyCount($response, 'programmers', 2);
... line 69
70      }
... lines 71 - 72
```

Now let's run this - but copy the method name first. On the command line, before the filename, add --filter then paste the method name to *just* run this test:

```
$ phpunit -c app --filter testGETProgrammersCollection src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Yep - one little dot and the PHPUnit gnomes are pleased.

Deep Assertions with Property Path

Let's go further. We know the programmers property will have 2 items in it: the 0 index should be the UnitTester data and the 1 index should be the CowboyCoder data. Copy the assertResponsePropertyEquals() method and paste it here. But instead of just nickname, use programmers[1].nickname. And this should be CowboyCoder:

72 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 53
54      public function testGETProgrammersCollection()
55      {
... lines 56 - 64
65          $response = $this->client->get('/api/programmers');
... lines 66 - 68
69          $this->asserter()->assertResponsePropertyEquals($response, 'programmers[1].nickname', 'CowboyCoder');
70      }
... lines 71 - 72
```

And that's the super-power of the PropertyAccess component: it lets you walk down through the response data. This is really fun, give this a try:

```
$ phpunit -c app --filter testGETProgrammersCollection src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

We're still passing. If you change that to CowboyCoder2, we get that really clear failure message and the dumped JSON response right above it. We're dangerous. Change that test back so it passes.

Chapter 13: Using a Test Database

We're using the built-in PHP web server running on port 8000. We have that hardcoded at the top of ApiTestCase: when the Client is created, it *always* goes to localhost:8000. Bummer! All of our fellow code battlers will need to have the exact same setup.

We need to make this configurable - create a new variable $baseUrl and set it to an environment variable called TEST_BASE_URL - I'm making that name up. Use this for the base_url option:

273 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 45
46      public static function setUpBeforeClass()
47      {
48          $baseUrl = getenv('TEST_BASE_URL');
49          self::$staticClient = new Client([
50              'base_url' => $baseUrl,
51              'defaults' => [
52                  'exceptions' => false
53              ]
54          ]);
... lines 55 - 59
60      }
... lines 61 - 273
```

There are endless ways to set environment variables. But we want to at least give this a default value. Open up app/phpunit.xml.dist. Get rid of those comments - we want a php element with an env node inside. I'll paste that in:

36 lines | app/phpunit.xml.dist

```
... lines 1 - 3
4   <phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5            xsi:noNamespaceSchemaLocation="http://schema.phpunit.de/4.1/phpunit.xsd"
6            backupGlobals="false"
7            colors="true"
8            bootstrap="bootstrap.php.cache"
9   >
... lines 10 - 17
18      <php>
19          <env name="TEST_BASE_URL" value="http://localhost:8000" />
20      </php>
... lines 21 - 34
35  </phpunit>
```

If you have our setup, everything just works. If not, you can set this environment variable or create a phpunit.xml file to override everything.

Let's double-check that this all works:

```
$ phpunit -c app --filter testGETProgrammersCollection src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Tests Killed our Database

One *little* bummer is that the tests are using our development database. Since those create a weaverryan user with password foo, that still works. But the cute programmer we created earlier is gone - they've been wiped out, sent to /dev/null... hate to see that.

Configuring the test Environment

Symfony has a test environment for *just* this reason. So let's use it! Start by copying app_dev.php to app_test.php, then change the environment key from dev to test. To know if this all works, put a temporary die statement right on top:

31 lines | web/app_test.php

```
<?php
die('working?');
... lines 3 - 24
$kernel = new AppKernel('test', true);
... lines 26 - 31
```

We'll setup our tests to hit *this* file instead of app_dev.php, which is being used now because Symfony's server:run command sets up the web server with that as the default.

Once we do that, we can setup the test environment to use a different database name. Open config.yml and copy the doctrine configuration. Paste it into config_test.yml to override the original. All we really want to change is dbname. I like to just take the real database name and suffix it with _test:

21 lines | app/config/config_test.yml

```
... lines 1 - 17
doctrine:
    dbal:
        dbname:   "%database_name%_test"
```

Ok, last step. In phpunit.xml.dist, add a /app_test.php to the end of the URL. In theory, all our API requests will now hit *this* front controller.

Run the test! This *shouldn't* pass - it should hit that die statement on every endpoint:

```
$ phpunit -c app --filter testGETProgrammersCollection src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

They fail! But not for the reason we wanted:

```
Unknown database `symfony_rest_recording_test`
```

Woops, I forgot to create the new test database. Fix this with doctrine:database:create in the test environment and doctrine:schema:create:

```
$ php app/console doctrine:database:create --env=test
$ php app/console doctrine:schema:create --env=test
```

Try it again:

```
$ phpunit -c app --filter testGETProgrammersCollection src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Huh, it passed. *Not* expected. We should be hitting this die statement. Something weird is going on.

Debugging Weird/Failing Requests

Go into ProgrammerControllerTest to debug this. We *should* be going to a URL with app_test.php at the front, but it *seems* like that's not happening. Use $this->printLastRequestUrl() after making the request:

73 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 53
54      public function testGETProgrammersCollection()
55      {
... lines 56 - 64
65          $response = $this->client->get('/api/programmers');
66          $this->printLastRequestUrl();
... lines 67 - 70
71      }
... lines 72 - 73
```

This is one of the helper functions I wrote - it shows the *true* URL that Guzzle is using.

Now run the test:

```
$ phpunit -c app --filter testGETProgrammersCollection src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Huh, so there's *not* app_test.php in the URL. Ok, so here's the deal. With Guzzle, if you have this opening slash in the URL, it takes that string and puts it right after the domain part of your base_url. Anything after that gets run over. We *could* fix this by taking out the opening slash everywhere - like api/programmers - but I just don't like that: it looks weird.

Properly Prefixing all URIs

Instead, get rid of the app_test.php part in phpunit.xml.dist:

36 lines | app/phpunit.xml.dist

```
... lines 1 - 3
4   <phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
... lines 5 - 17
18      <php>
19          <env name="TEST_BASE_URL" value="http://localhost:8000" />
20      </php>
... lines 21 - 34
35  </phpunit>
```

We'll solve this a different way. When the Client is created in ApiTestCase, we have the chance to attach listeners to it. Basically, we can hook into different points, like right before a request is sent or right after. Actually, I'm already doing that to keep track of the Client's history for some debugging stuff.

I'll paste some code, and add a use statement for this BeforeEvent class:

283 lines | src/AppBundle/Test/ApiTestCase.php

```
... lines 1 - 10
use GuzzleHttp\Event\BeforeEvent;
... lines 12 - 20
class ApiTestCase extends KernelTestCase
{
... lines 23 - 46
    public static function setUpBeforeClass()
    {
... lines 49 - 59
        // guaranteeing that /app_test.php is prefixed to all URLs
        self::$staticClient->getEmitter()
            ->on('before', function(BeforeEvent $event) {
                $path = $event->getRequest()->getPath();
                if (strpos($path, '/api') === 0) {
                    $event->getRequest()->setPath('/app_test.php'.$path);
                }
            });
... lines 68 - 69
    }
... lines 71 - 281
}
```

Ah Guzzle - you're so easy to understand sometimes! So as you can probably guess, this function is called *before* every request is made. All we do is look to see if the path starts with /api. If it does, prefix that with /app_test.php. This will make every request use that front controller, without ever needing to think about that in the tests.

Give it another shot:

```
$ phpunit -c app --filter testGETProgrammersCollection src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php
```

Errors! Yes - it doesn't see a programmers property in the response because all we have is this crumby die statement text. Now that we know things hit app_test.php, go take that die statement out of it. And remove the printLastRequestUrl(). Run the entire test suite:

```
$ phpunit -c app
```

Almost! There's 1 failure! Inside testPOST - we're asserting that the Location header is this string, but now it has the app_test.php part in it. That's a false failure - our code *is* really working. Let's soften that test a bit. How about replacing assertEquals() with assertStringEndsWith(). Now let's see some passing:

```
$ phpunit -c app
```

Yay!

Chapter 14: PUT is for Updating

Suppose now that someone using our API needs to *edit* a programmer: maybe they want to change its avatar. What HTTP method should we use? And what should the endpoint return? Answering those questions is one of the reasons we always start by writing a test - it's like the design phase of a feature.

Create a public function testPUTProgrammer() method:

92 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 71
72      public function testPUTProgrammer()
73      {
... lines 74 - 89
90      }
... lines 91 - 92
```

Usually, if you want to edit a resource, you'll use the PUT HTTP method. And so far, we've seen POST for creating and PUT for updating. But it's more complicated than that, and involves PUT being idempotent. We have a full 5 minute video on this in our original REST screencast (see PUT versus POST), and if you don't know the difference between PUT and POST, you should geek out on this.

Inside the test, copy the createProgrammer() for CowboyCoder from earlier. Yep, this programmer definitely needs his avatar changed. Next copy the request and assert stuff from testGETProgrammer() and add that. Ok, what needs to be updated. Change the request from get() to put(). And like earlier, we need to send a JSON string body in the request. Grab one of the $data arrays from earlier, add it here, then json_encode() it for the body. This is a combination of stuff we've already done:

92 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 71
72      public function testPUTProgrammer()
73      {
74          $this->createProgrammer(array(
75              'nickname' => 'CowboyCoder',
76              'avatarNumber' => 5,
77              'tagLine' => 'foo',
78          ));
79
80          $data = array(
... lines 81 - 83
84          );
85          $response = $this->client->put('/api/programmers/CowboyCoder', [
86              'body' => json_encode($data)
87          ]);
... lines 88 - 89
90      }
... lines 91 - 92
```

For a PUT request, you're supposed to send the *entire* resource in the body, even if you only want to update one field. So we need to send nickname, avatarNumber *and* tagLine. Update the $data array so the nickname matches CowboyCoder, but change the avatarNumber to 2. We won't update the tagLine, so send foo and add that to createProgrammer() to make sure this is CowboyCoder's starting tagLine:

92 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 71
    public function testPUTProgrammer()
    {
        $this->createProgrammer(array(
... lines 75 - 76
            'tagLine' => 'foo',
        ));

        $data = array(
            'nickname' => 'CowboyCoder',
            'avatarNumber' => 2,
            'tagLine' => 'foo',
        );
        $response = $this->client->put('/api/programmers/CowboyCoder', [
            'body' => json_encode($data)
        ]);
... lines 88 - 89
    }
... lines 91 - 92
```

This will create the Programmer in the database then send a PUT request where only the avatarNumber is different. Asserting a 200 status code is perfect, and like most endpoints, we'll return the JSON programmer. But, we're already testing the JSON pretty well in other spots. So here, just do a sanity check: assert that the avatarNumber has in fact changed to 2:

92 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 71
    public function testPUTProgrammer()
    {
... lines 74 - 84
        $response = $this->client->put('/api/programmers/CowboyCoder', [
            'body' => json_encode($data)
        ]);
... line 88
        $this->asserter()->assertResponsePropertyEquals($response, 'avatarNumber', 2);
    }
... lines 91 - 92
```

Ready? Try it out, with a --filter testPUTProgrammer to only run *this* one:

```
$ phpunit -c app --filter testPUTProgrammer
```

Hey, a 405 error! Method not allowed. That makes perfect sense: we haven't added this endpoint yet. Test check! Let's code!

Adding the PUT Controller

Add a public function updateAction(). The start of this will look a lot like showAction(), so copy its Route stuff, but change the method to PUT, and change the name so it's unique. For arguments, add $nickname and also $request, because we'll need that in a second:

130 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 87
88      /**
89       * @Route("/api/programmers/{nickname}")
90       * @Method("PUT")
91       */
92      public function updateAction($nickname, Request $request)
93      {
... lines 94 - 116
117     }
... lines 118 - 130
```

Ok, we have two easy jobs: query for the Programmer then update it from the JSON. Steal the query logic from showAction():

130 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 91
92      public function updateAction($nickname, Request $request)
93      {
94          $programmer = $this->getDoctrine()
95              ->getRepository('AppBundle:Programmer')
96              ->findOneByNickname($nickname);
97
98          if (!$programmer) {
99              throw $this->createNotFoundException(sprintf(
100                 'No programmer found with nickname "%s"',
101                 $nickname
102             ));
103         }
... lines 104 - 116
117     }
... lines 118 - 130
```

The updating part is something we did in the original POST endpoint. Steal *everything* from newAction(), though we don't need all of it. Yes yes, we *will* have some code duplication for a bit. Just trust me - we'll reorganize things over time. Get rid of the new Programmer() line - we're querying for one. And take out the setUser() code too: that's just needed on create. And because we're not creating a resource, we don't need the Location header and the status code should be 200, not 201:

130 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 91
92      public function updateAction($nickname, Request $request)
93      {
... lines 94 - 104
105         $data = json_decode($request->getContent(), true);
106         $form = $this->createForm(new ProgrammerType(), $programmer);
107         $form->submit($data);
108
109         $em = $this->getDoctrine()->getManager();
110         $em->persist($programmer);
111         $em->flush();
112
113         $data = $this->serializeProgrammer($programmer);
114         $response = new JsonResponse($data, 200);
115
116         return $response;
117     }
... lines 118 - 130
```

Done! And if you look at the function, it's really simple. Most of the duplication is for pretty mundane code, like creating a form and saving the Programmer. Creating endpoints is already really easy.

Before I congratulate us any more, let's give this a try:

```
$ phpunit -c app --filter testPUTProgrammer
```

Uh oh! 404! But check out that really clear error message from the response:

```
No programmer found for username UnitTester
```

Well yea! Because we should be editing CowboyCoder. In ProgrammerControllerTest, I made a copy-pasta error! Update the PUT URL to be /api/programmers/CowboyCoder, not UnitTester:

92 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 71
72      public function testPUTProgrammer()
73      {
... lines 74 - 84
85          $response = $this->client->put('/api/programmers/CowboyCoder', [
86              'body' => json_encode($data)
87          ]);
... lines 88 - 89
90      }
... lines 91 - 92
```

Now we're ready again:

```
$ phpunit -c app --filter testPUTProgrammer
```

We're passing!

Centralizing Form Data Processing

Before we go on we need to clean up some of this duplication. It's small, but each write endpoint is processing the request body in the same way: by fetching the content from the request, calling json_decode() on that, then passing it to $form->submit().

Create a new private function called processForm(). This will have two arguments - $request and the form object, which is a FormInterface instance, not that that's too important:

133 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 116
117     private function processForm(Request $request, FormInterface $form)
118     {
... lines 119 - 120
121     }
... lines 122 - 133
```

We'll move two things here: the two lines that read and decode the request body and the $form->submit() line:

133 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 116
117     private function processForm(Request $request, FormInterface $form)
118     {
119         $data = json_decode($request->getContent(), true);
120         $form->submit($data);
121     }
... lines 122 - 133
```

If this looks small to you, it is! But centralizing the json_decode() means we'll be able to handle invalid JSON in one spot, really easily in the next episode.

In updateAction(), call $this->processForm() passing it the $request and the $form. Celebrate by removing the json_decode lines. Do the same thing up in newAction:

133 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 20
21      public function newAction(Request $request)
22      {
23          $programmer = new Programmer();
24          $form = $this->createForm(new ProgrammerType(), $programmer);
25          $this->processForm($request, $form);
... lines 26 - 41
42      }
... lines 43 - 90
91      public function updateAction($nickname, Request $request)
92      {
... lines 93 - 103
104         $form = $this->createForm(new ProgrammerType(), $programmer);
105         $this->processForm($request, $form);
... lines 106 - 114
115     }
... lines 116 - 133
```

Yay! We're just a little cleaner. To really congratulate ourselves, try the whole test suite:

```
$ phpunit -c app
```

Wow!

Chapter 15: Read-Only Fields

What if we don't want the nickname to be changeable? After all, we're using it almost like a primary key for the Programmer. Yea, I want an API client to set it on create, but I don't want them to be able to change it afterwards.

Send a new nickname in the body of the PUT request - CowgirlCoder. We want the server to just ignore that. At the end, assert that nickname *still* equals CowboyCoder, even though we're trying to mess with things:

94 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 71
72      public function testPUTProgrammer()
73      {
... lines 74 - 79
80          $data = array(
81              'nickname' => 'CowgirlCoder',
... lines 82 - 83
84          );
... lines 85 - 89
90          // the nickname is immutable on edit
91          $this->asserter()->assertResponsePropertyEquals($response, 'nickname', 'CowboyCoder');
92      }
... lines 93 - 94
```

Run just that test:

```
$ phpunit -c app --filter testPUTProgrammer
```

Womp womp - we're failing: the nickname *is* updated on the programmer. That makes perfect sense: our form will update *any* of the 3 fields we configured in PogrammerType:

43 lines | src/AppBundle/Form/ProgrammerType.php

```
... lines 1 - 8
9   class ProgrammerType extends AbstractType
10  {
11      public function buildForm(FormBuilderInterface $builder, array $options)
12      {
13          $builder
14              ->add('nickname', 'text')
15              ->add('avatarNumber', 'choice', [
... lines 16 - 27
28              ->add('tagLine', 'textarea')
29          ;
30      }
... lines 31 - 42
43  }
```

Using disabled Form Fields

So how can we make nickname *only* writeable when we're adding a new programmer. If you think about the HTML world, this would be like a form that had a functional nickname text box when creating, but a *disabled* nickname text box when editing. We can use this idea in our API by giving the nickname field a disabled option that's set to true.

In an API, this will mean that any value submitted to this field will just be ignored. If we can set this to true in edit mode only, that would do the trick!

To do that, reference a new option called is_edit:

47 lines | src/AppBundle/Form/ProgrammerType.php

```
... lines 1 - 10
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nickname', 'text', [
                // readonly if we're in edit mode
                'disabled' => $options['is_edit']
            ])
... lines 18 - 31
        ;
    }
... lines 34 - 47
```

If we're in "edit mode", then the field is disabled. To make this a valid form option, add a new entry in setDefaultOptions() and default it to false:

47 lines | src/AppBundle/Form/ProgrammerType.php

```
... lines 1 - 34
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Programmer',
            'is_edit' => false,
        ));
    }
... lines 42 - 47
```

Head back to ProgrammerController::updateAction() and give createForm() a third array argument. Pass is_edit => true.

135 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 90
    public function updateAction($nickname, Request $request)
    {
... lines 93 - 103
        $form = $this->createForm(new ProgrammerType(), $programmer, array(
            'is_edit' => true,
        ));
... lines 107 - 116
    }
... lines 118 - 135
```

Ok, try the test!

```
$ phpunit -c app --filter testPUTProgrammer
```

Yay! That was easy!

Creating a Separate Form Class

And now that it's working, I need to force one small change on us that'll help us *way* in the future when we talk about API documentation. Instead of passing is_edit in the controller, we'll create a second form type class. Copy ProgrammerType to UpdateProgrammerType. Make this extend ProgrammerType and git rid of buildForm(). In setDefaultOptions(), we only need to set is_edit to true and call the parent function above this. Make sure getName() returns something unique:

23 lines | src/AppBundle/Form/UpdateProgrammerType.php

```
... lines 1 - 2
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class UpdateProgrammerType extends ProgrammerType
{
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        parent::setDefaultOptions($resolver);

        // override this!
        $resolver->setDefaults(['is_edit' => true]);
    }

    public function getName()
    {
        return 'programmer_edit';
    }
}
```

The whole purpose of this class is to act just like ProgrammerType, but set is_edit to true instead of us passing that in the controller. Both approaches are fine - but I'm planning ahead to when we use NelmioApiDocBundle: it likes 2 classes better. In the controller, use new UpdateProgrammerType and get rid of the third argument:

134 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 91
    public function updateAction($nickname, Request $request)
    {
... lines 94 - 104
        $form = $this->createForm(new UpdateProgrammerType(), $programmer);
... lines 106 - 115
    }
... lines 117 - 134
```

Test out your handy-work:

```
$ phpunit -c app --filter testPUTProgrammer
```

Success!

Chapter 16: DELETE is for Saying Goodbye

So you have to part ways with your programmer, and we all know goodbyes are hard. So let's delete them instead. We're going to create an rm -rf endpoint to send a programmer to /dev/null.

Start with the test! public function testDELETEProgrammer, because we'll send a DELETE request to terminate that programmer resource:

105 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 5
class ProgrammerControllerTest extends ApiTestCase
{
... lines 8 - 93
    public function testDELETEProgrammer()
    {
... lines 96 - 102
    }
}
```

Copy the guts of the GET test - fetching and deleting a programmer are almost the same, except for the HTTP method, change it to delete():

105 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 93
    public function testDELETEProgrammer()
    {
        $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));

        $response = $this->client->delete('/api/programmers/UnitTester');
... line 102
    }
... lines 104 - 105
```

Now, what's the status code? What should we return? We can't return the JSON programmer, because we just finished truncating it's proverbial tables. I mean, it's deleted - so it doesn't make sense to return the resource. Instead, we'll return nothing and use a status code - 204 - that means "everything went super great, but I have no content to send back." Remove the asserts on the bottom... since there's nothing to look at:

105 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 93
94      public function testDELETEProgrammer()
95      {
96          $this->createProgrammer(array(
97              'nickname' => 'UnitTester',
98              'avatarNumber' => 3,
99          ));
100
101         $response = $this->client->delete('/api/programmers/UnitTester');
102         $this->assertEquals(204, $response->getStatusCode());
103     }
... lines 104 - 105
```

The Controller

Let's get straight to the controller: public function deleteAction(). Copy the route stuff from updateAction(). It's all the same again, except the method is different. Take out the route name - we don't need this unless we link here. And change the method to DELETE:

154 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 117
118     /**
119      * @Route("/api/programmers/{nickname}")
120      * @Method("DELETE")
121      */
122     public function deleteAction($nickname)
123     {
... lines 124 - 135
136     }
... lines 137 - 154
```

Grab the query code from updateAction() too, and make sure you have your $nickname argument:

154 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 121
122     public function deleteAction($nickname)
123     {
124         $programmer = $this->getDoctrine()
125             ->getRepository('AppBundle:Programmer')
126             ->findOneByNickname($nickname);
127
128         if (!$programmer) {
129             throw $this->createNotFoundException(sprintf(
130                 'No programmer found with nickname "%s"',
131                 $nickname
132             ));
133         }
134
135         // todo ...
136     }
... lines 137 - 154
```

So this will 404 if we don't find the programmer. Surprise! In the REST world, this is controversial! Since the job of this endpoint is to make sure the programmer resource is deleted, some people say that if the resource is already gone, then that's success! In other words, you should return the same 204 even if the programmer wasn't found. When you learn more

about idempotency, this argument makes some sense. So let's do it! But really, either way is fine.

Change the if statement to be if ($programmer), then we'll delete it. Grab the EntityManager and call the normal remove() on it, then flush():

156 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 121
122     public function deleteAction($nickname)
123     {
... lines 124 - 127
128         if ($programmer) {
129             // debated point: should we 404 on an unknown nickname?
130             // or should we just return a nice 204 in all cases?
131             // we're doing the latter
132             $em = $this->getDoctrine()->getManager();
133             $em->remove($programmer);
134             $em->flush();
135         }
... lines 136 - 137
138     }
... lines 139 - 156
```

And whether the Programmer was found or not, we'll always return the same new Response(null, 204):

156 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 121
122     public function deleteAction($nickname)
123     {
124         $programmer = $this->getDoctrine()
125             ->getRepository('AppBundle:Programmer')
126             ->findOneByNickname($nickname);
127
128         if ($programmer) {
... lines 129 - 134
135         }
136
137         return new Response(null, 204);
138     }
... lines 139 - 156
```

Try the test and send a programmer to the trash! Filter for testDELETE.

```
$ phpunit -c app --filter testPUTProgrammer
```

Another endpoint bites the dust!

Chapter 17: PATCH is (also) for Updating (basically)

The main HTTP methods are: GET, POST, PUT and DELETE. There's another one you hear a lot about: PATCH.

The simple, but not entirely accurate definition of PATCH is this: it's just like PUT, except you don't need to send up the entire resource body. If you just want to update tagLine, just send that field.

So really, PATCH is a bit nicer to work with than PUT, and we'll support both. Start with the test -
public function testPATCHProgrammer():

125 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 93
94      public function testPATCHProgrammer()
95      {
... lines 96 - 111
112     }
... lines 113 - 125
```

Copy the inside of the PUT test: they'll be almost identical.

If you follow the rules with PUT, then if you don't send tagLine, the server should nullify it. Symfony's form system works like that, so our PUT is acting right. Good PUT!

But for PATCH, let's *only* send tagLine with a value of bar. When we do this, we expect tagLine to be bar, but we also expect avatarNumber is still equal to 5. We're not sending avatarNumber, which means: don't change it. And change the method from put() to patch():

124 lines | src/AppBundle/Tests/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 93
94      public function testPATCHProgrammer()
95      {
96          $this->createProgrammer(array(
97              'nickname' => 'CowboyCoder',
98              'avatarNumber' => 5,
99              'tagLine' => 'foo',
100         ));
101
102         $data = array(
103             'tagLine' => 'bar',
104         );
105         $response = $this->client->patch('/api/programmers/CowboyCoder', [
106             'body' => json_encode($data)
107         ]);
108         $this->assertEquals(200, $response->getStatusCode());
109         $this->asserter()->assertResponsePropertyEquals($response, 'avatarNumber', 5);
110         $this->asserter()->assertResponsePropertyEquals($response, 'tagLine', 'bar');
111     }
... lines 112 - 124
```

In reality, PATCH can be more complex than this, and we talk about that in our other REST screencast (see The Truth Behind PATCH). But *most* API's make PATCH work like this.

Make sure the test fails - filter it for PATCH to run just this one:

```
$ phpunit -c app --filter PATCH
```

Sweet! 405, method not allowed. Time to fix that!

Support PUT and PATCH

Since PUT and PATCH are *so* similar, we can handle them in the same action. Just change the @Method annotation to have a curly-brace with PUT *and* PATCH inside of it:

158 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 87
88      /**
89       * @Route("/api/programmers/{nickname}")
90       * @Method({"PUT", "PATCH"})
91       */
92      public function updateAction($nickname, Request $request)
... lines 93 - 158
```

Now, this route accepts PUT or PATCH. Try the test again:

```
$ phpunit -c app --filter PATCH
```

Woh, 500 error! Integrity constraint: avatarNumber cannot be null. It *is* hitting our endpoint and because we're not sending avatarNumber, the form framework *is* nullifying it, which eventually makes the database yell at us.

The work of passing the data to the form is done in our private processForm() method. And when it calls $form->submit(), there's a *second* argument called $clearMissing. It's default value - true - means that any missing fields are nullified. But if you set it to false, those fields are ignored. That's perfect PATCH behavior. Create a new variable above this line called $clearMissing and set it to $request->getMethod() != 'PATCH':

158 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 139
140     private function processForm(Request $request, FormInterface $form)
141     {
142         $data = json_decode($request->getContent(), true);
143
144         $clearMissing = $request->getMethod() != 'PATCH';
... line 145
146     }
... lines 147 - 158
```

In other words, clear all the missing fields, *unless* the request method is PATCH. Pass this as the second argument:

158 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 139
140     private function processForm(Request $request, FormInterface $form)
141     {
142         $data = json_decode($request->getContent(), true);
143
144         $clearMissing = $request->getMethod() != 'PATCH';
145         $form->submit($data, $clearMissing);
146     }
... lines 147 - 158
```

Head back, get rid of the big error message and run things again:

```
$ phpunit -c app --filter PATCH
```

Boom! We've got PUT and PATCH support with about 2 lines of code.

Chapter 18: Using a Serializer

We're turning Programmers into JSON by hand inside serializeProgrammer():

158 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 147
    private function serializeProgrammer(Programmer $programmer)
    {
        return array(
            'nickname' => $programmer->getNickname(),
            'avatarNumber' => $programmer->getAvatarNumber(),
            'powerLevel' => $programmer->getPowerLevel(),
            'tagLine' => $programmer->getTagLine(),
        );
    }
}
```

That's pretty ok with just one resource, but this will be a pain when we have a lot more - especially when resources start having relations to other resources. It'll turn into a whole soap opera. To make this way more fun, we'll use a serializer library: code that's really good at turning objects into an array, or JSON or XML.

The one we'll use is called "JMS Serializer" and there's a bundle for it called JMSSerializerBundle. This is a *fanstatic* library and incredibly powerful. It *can* get complex in a few cases, but we'll cover those. You should also know that this library is not maintained all that well anymore and you'll see a little bug that we'll have to work around. But it's been around for years, it's really stable and has a lot of users.

Symfony itself ships with a serializer, Symfony 2.7 has a lot of features that JMS Serializer has. There's a push inside Symfony to make it eventually replace JMS Serialize for most use-cases. So, keep an eye on that. Oh, and JMS Serializer is licensed under Apache2, which is a little bit less permissive than MIT, which is Symfony's license. If that worries you, look into it further.

With all that out of the way, let's get to work. Copy the composer require line and paste it into the terminal:

```
$ composer require jms/serializer-bundle
```

While we're waiting, copy the bundle line and add this into our AppKernel:

40 lines | app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 19
            new \JMS\SerializerBundle\JMSSerializerBundle(),
        );
... lines 22 - 31
        return $bundles;
    }
... lines 34 - 40
```

This gives us a new service calld jms_serializer, which can turn any object into JSON or XML. Not unlike a Harry Potter wizarding spell.... accio JSON! So in the controller, rename serializeProgrammer to serialize and make the argument $data, so you can pass it anything. And inside, just return $this->container->get('jms_serializer') and call serialize() on that, passing it $data and json:

151 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 15
16  class ProgrammerController extends BaseController
17  {
... lines 18 - 144
145     private function serialize($data)
146     {
147         return $this->container->get('jms_serializer')
148             ->serialize($data, 'json');
149     }
150 }
```

PhpStorm is angry, just because composer hasn't finished downloading yet: we're working ahead.

Find everywhere we used serializeProgrammer() and change those. The only trick is that it's not returning an array anymore, it's returning JSON. So I'll say $json = $this->serialize($programmer). And we can't use JsonResponse anymore, or it'll encode things twice. Create a regular Response instead. Copy this and repeat the same thing in showAction(). Use a normal Response here too:

151 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 21
22      public function newAction(Request $request)
23      {
... lines 24 - 33
34          $json = $this->serialize($programmer);
35          $response = new Response($json, 201);
... lines 36 - 39
40          $response->headers->set('Location', $programmerUrl);
41
42          return $response;
43      }
... lines 44 - 48
49      public function showAction($nickname)
50      {
... lines 51 - 61
62          $json = $this->serialize($programmer);
63
64          $response = new Response($json, 200);
65
66          return $response;
67      }
... lines 68 - 151
```

For listAction, life gets easier. Just put the $programmers array inside the $data array and then pass this big structure into the serialize() function:

151 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 72
    public function listAction()
    {
        $programmers = $this->getDoctrine()
            ->getRepository('AppBundle:Programmer')
            ->findAll();
        $json = $this->serialize(['programmers' => $programmers]);

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

        return $response;
    }
... lines 84 - 151
```

The serializer has no problem serializing arrays of things. Make the same changes in updateAction():

151 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 88
    public function updateAction($nickname, Request $request)
    {
... lines 91 - 108
        $json = $this->serialize($programmer);
        $response = new Response($json, 200);

        return $response;
    }
... lines 114 - 151
```

Great! Let's check on Composer. It's done, so let's try our entire test suite:

```
$ phpunit -c app
```

Ok, things are *not* going well. One of them says:

```
Error reading property "avatarNumber" from available keys
(id, nickname, avatar_number, power_level)
```

The responses on top show the same thing: all our properties are being underscored. The JMS Serializer library does this by default... which I kinda hate. So we're going to turn it off.

The library has something called a "naming strategy" - basically how it transforms property names into JSON or XML keys. You can see some of this inside the bundle's configuration. They have a built-in class for doing nothing: it's called the "identical" naming strategy. Unfortunately, the bundle has a bug that makes this not configurable in the normal way. Instead, we need to go kung-foo on it.

Open up config.yml. I'll paste a big long ugly new parameter here:

79 lines | app/config/config.yml

```
... lines 1 - 5
parameters:
    # a hack - should be configurable under jms_serializer, but the property_naming.id
    # doesn't seem to be taken into account at all.
    jms_serializer.camel_case_naming_strategy.class: JMS\Serializer\Naming\IdenticalPropertyNamingStrategy
... lines 10 - 79
```

This creates a new parameter called jms_serializer.camel_case_naming_strategy.class. I'm setting this to JMS\Serializer\Naming\IdenticalPropertyNamingStrategy. That is a total hack - I only know to do this because I went deep

enough into the bundle to find this. If you want to know how this works, check out our Journey to the Center of Symfony: Dependency Injection screencast: it's good nerdy stuff. The important thing for us is that this will leave our property names alone.

So now if we run the test:

```
$ phpunit -c app
```

we still have failures. But in the dumped response, our property names are back!

Chapter 19: Centralize that Response!

Check out the response - it's got a Content-Type of text/html. I thought we fixed that! Well, that's no surprise - when we switched from JsonResponse to Response, we lost that header. But more importantly, this mistake is too easy to make: we're calling serialize() and then creating the Response by hand in *every* controller. That means we'd need to set this header everywhere. That sucks. Let's centralize this across our entire project.

First, move serialize() out of ProgrammerController and into a class called BaseController. This is something I created and all controllers extend this. Paste this at the bottom and make it protected:

129 lines | src/AppBundle/Controller/BaseController.php

```
... lines 1 - 15
abstract class BaseController extends Controller
{
... lines 18 - 122
    protected function serialize($data, $format = 'json')
    {
        return $this->container->get('jms_serializer')
            ->serialize($data, $format);
    }
}
```

And while we're here - make another function: protected function createApiResponse(). Give it two arguments: $data and $statusCode that defaults to 200:

129 lines | src/AppBundle/Controller/BaseController.php

```
... lines 1 - 15
abstract class BaseController extends Controller
{
... lines 18 - 113
    protected function createApiResponse($data, $statusCode = 200)
    {
... lines 116 - 120
    }
... lines 122 - 127
}
```

Instead of creating the Response ourselves, we can just call this and it'll take care of the details. Inside, first serialize the $data - whatever that is. And then return a new Response() with that $json, that $statusCode and - most importantly - that Content-Type header of application/json so we don't forget to set that:

129 lines | src/AppBundle/Controller/BaseController.php

```
... lines 1 - 113
    protected function createApiResponse($data, $statusCode = 200)
    {
        $json = $this->serialize($data);

        return new Response($json, $statusCode, array(
            'Content-Type' => 'application/json'
        ));
    }
... lines 122 - 129
```

I love it! Let's use this everywhere! Search for new Response. Call $response = $this->createApiResponse() and pass the $programmer. Copy that line and make sure it's status code is 201. Remove the other stuff, but *keep* the line that sets the Location header:

140 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 15
class ProgrammerController extends BaseController
{
... lines 18 - 21
    public function newAction(Request $request)
    {
... lines 24 - 33
        $response = $this->createApiResponse($programmer, 201);
... lines 35 - 38
        $response->headers->set('Location', $programmerUrl);

        return $response;
    }
... lines 43 - 138
}
```

Ok, *much* easier. Find the rest of the new Response spots and update them. It's all pretty much the same - listAction() has a different variable name, but that's it. For deleteAction(), well, it's returning a null Response, so we can leave that one alone.

140 lines | src/AppBundle/Controller/Api/ProgrammerController.php

```
... lines 1 - 47
    public function showAction($nickname)
    {
... lines 50 - 60
        $response = $this->createApiResponse($programmer, 200);

        return $response;
    }
... lines 65 - 69
    public function listAction()
    {
... lines 72 - 75
        $response = $this->createApiResponse(['programmers' => $programmers], 200);

        return $response;
    }
... lines 80 - 84
    public function updateAction($nickname, Request $request)
    {
... lines 87 - 104
        $response = $this->createApiResponse($programmer, 200);

        return $response;
    }
... lines 109 - 140
```

Let's re-run the tests!

```
$ phpunit -c app
```

They still fail, but the responses have the right Content-Type header.

Time to fix these failures, *and* see how we can control the serializer.

Chapter 20: Taking Control of the Serializer

The serializer is... *mostly* working. But we have some test failures:

```
Error reading property "tagLine" from available keys (id, nickname
avatarNumber, powerLevel)
```

Huh. Yea, if you look at the response, tagLine is mysteriously absent! Where did you go dear tagLine???

So the serializer works like this: you give it an object, and it serializes every property on it. Yep, you *can* control that - just hang on a few minutes. But, if any of these properties is null, instead of returning that key with null, it omits it entirely.

Fortunately, that's easy to change. Go into BaseController. In serialize() create a new variable called $context and set that to a new SerializationContext(). Call setSerializeNull() on this and pass it true. To finish this off, pass that $context as the third argument to serialize():

133 lines | src/AppBundle/Controller/BaseController.php

```
... lines 1 - 123
    protected function serialize($data, $format = 'json')
    {
        $context = new SerializationContext();
        $context->setSerializeNull(true);

        return $this->container->get('jms_serializer')
            ->serialize($data, $format, $context);
    }
... lines 132 - 133
```

Think of the SerializationContext as serialization configuration. It doesn't do a lot of useful stuff - but it *does* let us tell the serializer to actually return null fields.

So run the *whole* test suite again and wait impatiently:

```
$ phpunit -c app
```

ZOMG! They're passing!

Serialization Annotations

But something extra snuck into our Response - let me show you. In testGETProgrammer(), at the end, add $this->debugResponse(). Copy that method name and run just it:

```
$ phpunit -c app --filter testGETProgrammer
```

Ah, the id field snuck into the JSON. Before, we only serialized the other four fields. So what if we *didn't* want id or some property to be serialized?

The solution is so nice. Go back to the homepage of the bundle's docs. There's one documentation gotcha: the bundle is a small wrapper around the JMS Serializer library, and most of the documentation lives there. Click the documentation link to check it out.

This has a *great* page called Annotations: it's a reference of *all* of the ways that you can control serialization.

@VirtualProperty and @SerializedName

One useful annotation is @VirtualProperty. This lets you create a method and have its return value serialized. If you use that with @SerializedName, you can control the serialized property name for this or anything.

For controlling *which* fields are returned, we'll use @ExclusionPolicy. Scroll down to the @AccessType code block and copy that use statement. Open the Programmer entity, paste this on top, but remove the last part and add as Serializer:

189 lines | src/AppBundle/Entity/Programmer.php

```
... lines 1 - 5
use JMS\Serializer\Annotation as Serializer;
... lines 7 - 189
```

This will let us say things like @Serializer\ExclusionPolicy. Add that on top of the class, with "all".

189 lines | src/AppBundle/Entity/Programmer.php

```
... lines 1 - 5
use JMS\Serializer\Annotation as Serializer;
... line 7
/**
... lines 9 - 12
 * @Serializer\ExclusionPolicy("all")
 */
class Programmer
... lines 16 - 189
```

This says: "Hey serializer, don't serialize *any* properties by default, just hang out in your pajamas". Now we'll use @Serializer\Expose() to whitelist the stuff we *do* want. We don't want id - so leave that. Above the $name property, add @Serializer\Expose(). Do this same thing above $avatarNumber, $tagLine and $powerLevel:

188 lines | src/AppBundle/Entity/Programmer.php

```
... lines 1 - 14
class Programmer
{
... lines 17 - 25
    /**
... lines 27 - 29
     * @Serializer\Expose
     */
    private $nickname;
... line 33
    /**
... lines 35 - 37
     * @Serializer\Expose
     */
    private $avatarNumber;
... line 41
    /**
... lines 43 - 45
     * @Serializer\Expose
     */
    private $tagLine;
... line 49
    /**
... lines 51 - 53
     * @Serializer\Expose
     */
    private $powerLevel = 0;
... lines 57 - 186
}
```

And my good buddy PhpStorm is telling me I have a syntax error up top. Whoops, I doubled my use statements - get rid of the

extra one.

With this, the id field *should* be gone from the response. Run the test!

```
$ phpunit -c app --filter testGETProgrammer
```

No more id! Take out the debugResponse(). Phew! Congrats! We only have one resource, but our API is kicking butt! We've built a system that let's us serialize things easily, create JSON responses and update data via forms.

Oh, and the serializer can also *deserialize*. That is, take JSON and turn it back into an object. I prefer to use forms instead of this, but it may be another option. Of course, if life gets complex, you can always just handle incoming data manually without forms or deserialization. Just keep that in mind.

We also have a killer test setup that let's us write tests first without any headache. We could just keep repeating what we have here to make a bigger API.

But, there's more to cover! In episode 2, we'll talk about errors: a fascinating topic for API's and something that can make or break how usable your API will be.

Ok, seeya next time!