Symfony RESTful API: Hypermedia, Links & Bonuses (Course 5)

With <3 from SymfonyCasts

Chapter 1: The Battle Resource

Hey guys! Finally part 5 - and it's *special*! It's my chance to take on some buzzwords directly: like hypermedia & HATEOAS.

Now, defining these terms is easy... but putting them into practice? For me, it was a disaster.

Let's take those ideas on directly in this tutorial, and clear up what's hype, and what is actually an evil ploy to prevent your awesome API from being released.

Code Setup

And hey, I just had an idea! You should code along with me! We've gone to all this trouble already to add a download for the code, so you might as well get it. Plus, there are some kitten gifs in the archive.. ok, not really. Once you've unzipped the code, move into the start/ directory, you'll have the exact code I already have here.

There's also a README file that has a few other setup instructions you'll need.

And if you've been coding along with previous courses, you're my favorite. But... download the new code: I made a few small changes to the project.

Once you're ready, start the built-in web server with:

```
$ ./bin/console server:run
```

Resources: Programmer + Project = (epic) Battle

Go to the frontend at http://localhost:8000 and login with weaverryan, password foo. The API for the programmer resource is done: you can create, edit, view and do other cool programmery things.

But the *real* point of the site is to create epic *battles*, and here's how: choose a programmer, click "Start Battle", choose a project and then watch the magic. Boom!

There are *three* resources in play: the programmer, the project and a battle, whose entire existence is based on being related to these other two resources. And it turns out, relating things in an API can get tricky.

The resources in your API don't always need to match up with tables in your database, but they often do. Our Entity directory holds a Programmer entity:

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

```
... lines 1 - 9
10  /**
11   * Programmer
12   *
13   * @ORM\Table(name="battle_programmer")
14   * @ORM\Entity(repositoryClass="AppBundle\Repository\ProgrammerRepository")
15   * @Serializer\ExclusionPolicy("all")
16   * @Link(
17   *  "self",
18   *  route = "api_programmers_show",
19   *  params = { "nickname": "object.getNickname()" }
20   * )
21   */
22  class Programmer
23  {
... lines 24 - 196
197 }
```

and a Project entity, which just has name and difficultyLevel fields:

57 lines | src/AppBundle/Entity/Project.php

```
... lines 1 - 6
/**
 * @ORM\Table(name="battle_project")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\ProjectRepository")
 */
class Project
{
... lines 13 - 19
    /**
     * @ORM\Column(type="string")
     */
    private $name;

    /**
     * 1-10 difficulty level of the project
     *
     * @ORM\Column(type="integer")
     */
    private $difficultyLevel;
... lines 31 - 55
}
```

Now check out Battle: it has a ManyToOne relationship to Programmer and Project:

107 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 6
/**
 * @ORM\Table(name="battle_battle")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\BattleRepository")
 */
class Battle
{
... lines 13 - 19
    /**
     * @ORM\ManyToOne(targetEntity="Programmer")
     * @ORM\JoinColumn(nullable=false)
     */
    private $programmer;

    /**
     * @ORM\ManyToOne(targetEntity="Project")
     * @ORM\JoinColumn(nullable=false)
     */
    private $project;
... lines 31 - 105
}
```

plus a few other fields like didProgrammerWin, foughtAt and notes:

107 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 6
 /**
  * @ORM\Table(name="battle_battle")
  * @ORM\Entity(repositoryClass="AppBundle\Repository\BattleRepository")
  */
 class Battle
 {
... lines 13 - 31
     /**
      * @ORM\Column(type="boolean")
      */
     private $didProgrammerWin;

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

     /**
      * @ORM\Column(type="text")
      */
     private $notes;
... lines 46 - 105
 }
```

Here's the goal: add an endpoint that will create a new battle resource. How should this endpoint look and act? You know the drill: let's design it first with a test.

Chapter 2: Designing (Testing) the Create Battle Endpoint

In the controller test directory, create a new class: BattleControllerTest. Make it extend the fancy ApiTestCase we've been working on:

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 2
namespace Tests\AppBundle\Controller\Api;

use AppBundle\Test\ApiTestCase;

class BattleControllerTest extends ApiTestCase
{
... lines 9 - 38
}
```

Start with public function testPOSTCreateBattle():

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 37
    }
}
```

Go steal some code from ProgrammerControllerTest(). Copy the setup method that creates a user so that we can send the Authorization header:

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
    protected function setUp()
    {
        parent::setUp();

        $this->createUser('weaverryan');
    }
... lines 15 - 38
}
```

Create a Project & Programmer

For this endpoint, there are only two pieces of information we need to send: which programmer and which project will battle. So before we start, we need to create these. Add $this->createProject() and give it a name: my_project - that doesn't matter:

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
        $project = $this->createProject('my_project');
... lines 19 - 37
    }
}
```

If you open this method, this method simply creates a new Project and flushes it to the database:

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

```
... lines 1 - 21
class ApiTestCase extends KernelTestCase
{
... lines 24 - 322
    /**
     * @param string $name
     * @return Project
     */
    protected function createProject($name)
    {
        $project = new Project();
        $project->setName($name);
        $project->setDifficultyLevel(rand(1, 10));

        $this->getEntityManager()->persist($project);
        $this->getEntityManager()->flush();

        return $project;
    }
... lines 338 - 371
}
```

Next, create the programmer: $this->createProgrammer(). This takes an array of information about that programmer. Hmm, let's call him Fred. Pass weaverryan as the second argument: that will be the user who *owns* Fred:

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
        $project = $this->createProject('my_project');
        $programmer = $this->createProgrammer([
            'nickname' => 'Fred'
        ], 'weaverryan');
... lines 22 - 37
    }
}
```

Eventually, we'll need to restrict this endpoint so that we can only start battles with *our* programmers.

Sending the Related Fields

Next, send a POST request! In ProgrammerControllerTest, it was easy: we sent 3 scalar fields:

290 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTProgrammerWorks()
    {
        $data = array(
            'nickname' => 'ObjectOrienter',
            'avatarNumber' => 5,
            'tagLine' => 'a test dev!'
        );
... lines 23 - 35
    }
... lines 37 - 288
}
```

But now, it's a little different: we want to send data that identifies the related programmer and project *resources*. Should we send the id? Or the programmer's nickname?

Well, like normal with API's... it doesn't matter. But sending the id's makes sense. Create a new $data array with two fields: project set to $project->getId() and programmer set to $programmer->getId():

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
        $project = $this->createProject('my_project');
        $programmer = $this->createProgrammer([
            'nickname' => 'Fred'
        ], 'weaverryan');

        $data = array(
            'project' => $project->getId(),
            'programmer' => $programmer->getId()
        );
... lines 27 - 37
    }
}
```

I'm calling the keys programmer and project for obvious reasons: but they could be anything. *We* are in charge of naming the fields whatever we want. Just be sane and consistent: please don't call the fields bob and larry - everyone will hate you.

Finally, make the request: $response = $this->client->post(). For programmers, the URL is /api/programmers. Stay consistent with /api/battles. Pass an array as the second argument with a body key set to json_encode($data) and a headers key set to $this->getAuthorizedHeaders('weaverryan'):

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
        $project = $this->createProject('my_project');
        $programmer = $this->createProgrammer([
            'nickname' => 'Fred'
        ], 'weaverryan');

        $data = array(
            'project' => $project->getId(),
            'programmer' => $programmer->getId()
        );

        $response = $this->client->post('/api/battles', [
            'body' => json_encode($data),
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 32 - 37
    }
}
```

That will send a valid JSON web token for the weaverryan user - we created that in the previous course.

Asserts!

Whew, okay. That's it. So even though Battle is dependent on two other resources, it works pretty much the same. Add some basic asserts: $this->assertEquals() that 201 will be the response status code:

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
        $project = $this->createProject('my_project');
        $programmer = $this->createProgrammer([
            'nickname' => 'Fred'
        ], 'weaverryan');

        $data = array(
            'project' => $project->getId(),
            'programmer' => $programmer->getId()
        );

        $response = $this->client->post('/api/battles', [
            'body' => json_encode($data),
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);

        $this->assertEquals(201, $response->getStatusCode());
... lines 34 - 37
    }
}
```

In Battle, one of the fields that should be returned is didProgrammerWin. Make sure that exists with $this->asserter()->assertResponsePropertyExists() and look for didProgrammerWin:

40 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
        $project = $this->createProject('my_project');
        $programmer = $this->createProgrammer([
            'nickname' => 'Fred'
        ], 'weaverryan');

        $data = array(
            'project' => $project->getId(),
            'programmer' => $programmer->getId()
        );

        $response = $this->client->post('/api/battles', [
            'body' => json_encode($data),
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);

        $this->assertEquals(201, $response->getStatusCode());
        $this->asserter()
            ->assertResponsePropertyExists($response, 'didProgrammerWin');
        // todo for later
        //$this->assertTrue($response->hasHeader('Location'));
    }
}
```

I'll also add a todo to check for the Location header later. Remember, when you create a resource, you're *supposed* to return a Location header to the URL where the client can view the new resource. We don't have a GET endpoint for a battle yet, so we'll skip this.

The hard work is done: we've designed the new endpoint. Let's bring it to life!

Chapter 3: Saving Related Resources in a Form

In the Controller/Api directory, create a new BattleController. Make it extend the same BaseController as before: we've put a lot of shortcuts in this:

19 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 2
namespace AppBundle\Controller\Api;

use AppBundle\Controller\BaseController;
... lines 6 - 8
class BattleController extends BaseController
{
... lines 11 - 17
}
```

Then, add public function newAction(). Set the route above it with @Route - make sure you hit tab to autocomplete this: it adds the necessary use statement. Finish the URL: /api/battles. Do the same thing with @Method to restrict this to POST:

19 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 5
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class BattleController extends BaseController
{
    /**
     * @Route("/api/battles")
     * @Method("POST")
     */
    public function newAction()
    {
    }
}
```

Awesome! Our API processes input through a form - you can see that in ProgrammerController:

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

```
... lines 1 - 22
class ProgrammerController extends BaseController
{
    /**
     * @Route("/api/programmers")
     * @Method("POST")
     */
    public function newAction(Request $request)
    {
        $programmer = new Programmer();
        $form = $this->createForm(ProgrammerType::class, $programmer);
        $this->processForm($request, $form);
... lines 34 - 52
    }
... lines 54 - 149
}
```

This form modifies the Programmer entity directly and we save it. Simple.

BattleManager Complicates Things...

Well, not so simple in this case. What? I know, I like to make things as difficult as possible.

To create battles on the frontend, our controller uses a service class called BattleManager. It's kind of nice: it has a battle() function:

55 lines | src/AppBundle/Battle/BattleManager.php

```
... lines 1 - 9
class BattleManager
{
... lines 12 - 18
    /**
     * Creates and wages an epic battle
     *
     * @param Programmer $programmer
     * @param Project $project
     * @return Battle
     */
    public function battle(Programmer $programmer, Project $project)
    {
... lines 28 - 53
    }
}
```

We pass it a Programmer and Project and it takes care of all of the logic for creating a Battle, figuring out who won, and saving it to the database. We even gave Battle a __construct() function with two required arguments:

107 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
class Battle
{
... lines 13 - 46
    /**
     * Battle constructor.
     * @param $programmer
     * @param $project
     */
    public function __construct(Programmer $programmer, Project $project)
    {
        $this->programmer = $programmer;
        $this->project = $project;
        $this->foughtAt = new \DateTime();
    }
... lines 58 - 105
}
```

This is a really nice setup, so I don't want to change it. But, it doesn't work well with the form system: it prefers to instantiate the object and use setter functions.

Tip

Actually, it *is* possible to use the form system with the Battle entity by taking advantage of data mappers.

But that's ok! In fact, I like this complication: it shows off a very nice workaround. Just create a *new* model class for the form. In fact, I recommend this whenever you have a form that stops looking like or working nicely with your entity class.

Adding the BattleModel

In the Form directory, create a Model directory to keep things organized. Inside, add a new class called BattleModel:

33 lines | src/AppBundle/Form/Model/BattleModel.php

```
... lines 1 - 2
namespace AppBundle\Form\Model;
... lines 4 - 7
class BattleModel
{
... lines 10 - 32
}
```

Give *it* the two properties we're expecting as API input: $project and $programmer. Hit command+N - or go to the "Code"->"Generate" menu - and generate the getter and setter methods for both properties:

33 lines | src/AppBundle/Form/Model/BattleModel.php

```
... lines 1 - 4
use AppBundle\Entity\Programmer;
use AppBundle\Entity\Project;

class BattleModel
{
    private $project;

    private $programmer;

    public function getProject()
    {
        return $this->project;
    }

    public function setProject(Project $project)
    {
        $this->project = $project;
    }

    public function getProgrammer()
    {
        return $this->programmer;
    }

    public function setProgrammer(Programmer $programmer)
    {
        $this->programmer = $programmer;
    }
}
```

To be extra safe and make your code more hipster, type-hint setProgrammer() with the Programmer class and setProject() with Project. The form system will *love* this class.

Designing the Form

In the Form directory, create a new class for the form: BattleType. Make this extend the normal AbstractType and then hit command+N - or "Code"->"Generate" - and go to "Override Methods". Select the two we need: buildForm and configureOptions:

33 lines | src/AppBundle/Form/BattleType.php

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

... lines 5 - 6
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 15 - 22
    }

    public function configureOptions(OptionsResolver $resolver)
    {
... lines 27 - 30
    }
}
```

Take out the parent calls - the parent methods are empty.

EntityType to the Rescue!

Okay, let's think about this. The API client will send programmer and project fields and both will be ids. Ultimately, we want to turn those into the *entity* objects corresponding to those ids before setting the data on the BattleModel object.

Well, this is *exactly* what the Entity type does. Use $builder->add() with project set to EntityType::class. To tell it *what* entity to use, add a class option set to AppBundle\Entity\Project:

33 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 5
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
... lines 7 - 10
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('programmer', EntityType::class, [
                'class' => 'AppBundle\Entity\Programmer'
            ])
... lines 19 - 21
        ;
    }
... lines 24 - 31
}
```

Do the same for programmer and set its class to AppBundle\Entity\Programmer:

33 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 10
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('programmer', EntityType::class, [
                'class' => 'AppBundle\Entity\Programmer'
            ])
            ->add('project', EntityType::class, [
                'class' => 'AppBundle\Entity\Project'
            ])
        ;
    }
... lines 24 - 31
}
```

In a web form, the entity type renders as a drop-down of projects or programmers. But it's *perfect* for an API: it transforms the project id into a Project object by querying for it. That's *exactly* what we want.

In configureOptions(), add $resolver->setDefaults() and pass it two things: first the data_class set to BattleModel::class:

33 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 4
use AppBundle\Form\Model\BattleModel;
... lines 6 - 10
class BattleType extends AbstractType
{
... lines 13 - 24
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => BattleModel::class,
... line 29
        ]);
    }
}
```

Make sure PhpStorm adds the use statement for that class. Then, set csrf_protection to false because we can't use normal CSRF protection in an API:

33 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 10
class BattleType extends AbstractType
{
... lines 13 - 24
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => BattleModel::class,
            'csrf_protection' => false,
        ]);
    }
}
```

Form, ready! Now let's hit the controller.

Chapter 4: Finishing the Battle

Head to the controller. We've got this form flow mastered! Step 1: create a new BattleModel object: $battleModel = new BattleModel():

37 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 11
class BattleController extends BaseController
{
    /**
     * @Route("/api/battles")
     * @Method("POST")
     */
    public function newAction(Request $request)
    {
        $battleModel = new BattleModel();
... lines 21 - 34
    }
}
```

Step 2: create the form: $form = $this->createForm() with BattleType::class:

37 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 5
use AppBundle\Form\BattleType;
... lines 7 - 11
class BattleController extends BaseController
{
... lines 14 - 17
    public function newAction(Request $request)
    {
        $battleModel = new BattleModel();
        $form = $this->createForm(BattleType::class, $battleModel);
... lines 22 - 34
    }
}
```

On my version of PhpStorm, I need to go back and re-type the e to trigger auto-completion so that the use statement is added above.

For the second argument to createForm: pass it $battleModel.

Step 3: Use $this->processForm():

37 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 11
class BattleController extends BaseController
{
... lines 14 - 17
    public function newAction(Request $request)
    {
        $battleModel = new BattleModel();
        $form = $this->createForm(BattleType::class, $battleModel);
        $this->processForm($request, $form);
... lines 23 - 34
    }
}
```

Remember, this is a method we added in BaseController:

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

```
... lines 1 - 19
abstract class BaseController extends Controller
{
... lines 22 - 142
    protected function processForm(Request $request, FormInterface $form)
    {
        $data = json_decode($request->getContent(), true);
        if ($data === null) {
            $apiProblem = new ApiProblem(400, ApiProblem::TYPE_INVALID_REQUEST_BODY_FORMAT);

            throw new ApiProblemException($apiProblem);
        }

        $clearMissing = $request->getMethod() != 'PATCH';
        $form->submit($data, $clearMissing);
    }
... lines 155 - 185
}
```

it decodes the Request body and submits it into the form, which is what we do on *every* endpoint that processes data.

Type-hint the Request argument for the controller and pass this to processForm().

If the form is *not* valid, we need to send back errors. Use another method from earlier:
$this->throwApiProblemValidationException() and pass it the $form object:

37 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 11
12  class BattleController extends BaseController
13  {
... lines 14 - 17
18      public function newAction(Request $request)
19      {
20          $battleModel = new BattleModel();
21          $form = $this->createForm(BattleType::class, $battleModel);
22          $this->processForm($request, $form);
23
24          if (!$form->isValid()) {
25              $this->throwApiProblemValidationException($form);
26          }
... lines 27 - 34
35      }
36  }
```

This will grab the validation errors off and create that response.

At this point, we have a BattleModel object that's populated with the Programmer and Project objects sent in the request. To create the battle, we need to use the BattleManager. Do that with $this->getBattleManager() - that's just a shortcut to get the service - ->battle() and pass it $battleModel->getProgrammer() and $battleModel->getProject():

37 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 11
12  class BattleController extends BaseController
13  {
... lines 14 - 17
18      public function newAction(Request $request)
19      {
20          $battleModel = new BattleModel();
21          $form = $this->createForm(BattleType::class, $battleModel);
22          $this->processForm($request, $form);
23
24          if (!$form->isValid()) {
25              $this->throwApiProblemValidationException($form);
26          }
27
28          $battle = $this->getBattleManager()->battle(
29              $battleModel->getProgrammer(),
30              $battleModel->getProject()
31          );
... lines 32 - 34
35      }
36  }
```

Put a little $battle = in the beginning of all of this to get the new Battle object. Perfect!

Now that the battle has been heroically fought, let's send back the gory details. Use return $this->createApiResponse() and pass it $battle and the 201 status code:

37 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 11
class BattleController extends BaseController
{
... lines 14 - 17
    public function newAction(Request $request)
    {
        $battleModel = new BattleModel();
        $form = $this->createForm(BattleType::class, $battleModel);
        $this->processForm($request, $form);

        if (!$form->isValid()) {
            $this->throwApiProblemValidationException($form);
        }

        $battle = $this->getBattleManager()->battle(
            $battleModel->getProgrammer(),
            $battleModel->getProject()
        );

        // todo - set Location header
        return $this->createApiResponse($battle, 201);
    }
}
```

We aren't setting a Location header yet, so let's at least add a todo for that.

We are done! Controller, model and form: these are the *only* pieces we need to create a robust endpoint. Try the test:

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

We are in prime battling shape.

Now, let's complicate things and learn how to *really* take control of every field in our endpoint. And, learn more about relations.

Chapter 5: VirtualProperty: Add Crazy JSON Fields

The test passes, but let's see what the response looks like. Add $this->debugResponse():

41 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 35
        $this->debugResponse($response);
        // todo for later
        //$this->assertTrue($response->hasHeader('Location'));
    }
}
```

and re-run the test:

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

Check it out! It has the fields we expect, but it's *also* embedding the entire programmer and project resources. That's what the serializer does when a property is an object. This might be cool with you, or maybe not. For me, this looks like overkill. Instead of having the Programmer and Project data right here, it's probably enough to just have the programmer's *nickname* and the project's id.

But hold on: I want to mention something *really* important. Whenever you need to make a decision about *how* your API should work, the *right* decision should always depend on *who* you're making the API for. If you're building your API for an iPhone app, will having these extra fields be helpful? Or, if you're API is for a JavaScript frontend like ReactJS, then build your API to make React happy.

Adding an ExclusionPolicy

Let's assume that we do *not* want those embedded objects. First, hide them! In the Battle entity, we need to add some serialization exclusion rules. Since we do this via annotations, we need a use statement. Here's an easy way to get the correct use statement without reading the docs. I know that one of the annotations is called ExclusionPolicy. Add use ExclusionPolicy and let it autocomplete. Now, remove the ExclusionPolicy ending and add as Serializer:

114 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 6
use JMS\Serializer\Annotation as Serializer;

/**
 * @ORM\Table(name="battle_battle")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\BattleRepository")
... line 12
 */
class Battle
... lines 15 - 114
```

Now, above the class, add @Serializer\ExclusionPolicy("all"):

114 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 6
use JMS\Serializer\Annotation as Serializer;

/**
 * @ORM\Table(name="battle_battle")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\BattleRepository")
 * @Serializer\ExclusionPolicy("all")
 */
class Battle
... lines 15 - 114
```

now *no* properties will be used in the JSON, until we expose them. Expose id, skip programmer and project, and expose didProgrammerWin, foughtAt and notes:

114 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 13
class Battle
{
    /**
... lines 17 - 19
     * @Serializer\Expose()
     */
    private $id;
... lines 23 - 27
    private $programmer;
... lines 29 - 33
    private $project;

    /**
... line 37
     * @Serializer\Expose()
     */
    private $didProgrammerWin;

    /**
... line 43
     * @Serializer\Expose()
     */
    private $foughtAt;

    /**
... line 49
     * @Serializer\Expose()
     */
    private $notes;
... lines 53 - 112
}
```

Run the same test

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

Ok, awesome - the JSON has *just* these 4 fields.

Adding Fake Properties

Let's go to the next level. Now, I *do* want to have a programmer, but set to the username instead of the whole object. And I also *do* want a project field, set to its id.

Update the test to look for these. Use $this->asserter()->assertResponsePropertyEquals() and pass it $response. Look for a project field that's set to $project->getId():

45 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 33
        $this->asserter()
            ->assertResponsePropertyExists($response, 'didProgrammerWin');
        $this->asserter()
            ->assertResponsePropertyEquals($response, 'project', $project->getId());
... lines 38 - 42
    }
}
```

Copy that line and do the same thing for programmer: it should equal Fred:

45 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 33
        $this->asserter()
            ->assertResponsePropertyExists($response, 'didProgrammerWin');
        $this->asserter()
            ->assertResponsePropertyEquals($response, 'project', $project->getId());
        $this->asserter()
            ->assertResponsePropertyEquals($response, 'programmer', 'Fred');
... lines 40 - 42
    }
}
```

We could also have this return the id - it's up to you and what's best for your client.

But, how can we bring this to life? We're in a weird spot, because these fields *do* exist on Battle, but they have the wrong values. How can we do something custom?

By using something called a virtual property. First, create a new public function called getProgrammerNickname(). It should return $this->programmer->getNickname():

```
132 lines | src/AppBundle/Entity/Battle.php
    ... lines 1 - 13
14  class Battle
15  {
    ... lines 16 - 117
118     public function getProgrammerNickname()
119     {
120         return $this->programmer->getNickname();
121     }
    ... lines 122 - 130
131 }
```

VirtualProperty

Simple. But that will *not* be used by the serializer yet. To make that happen, add @Serializer\VirtualProperty above the method. As soon as you do this, it will be exposed in your API. But it will be called programmerNickname:

```
132 lines | src/AppBundle/Entity/Battle.php
    ... lines 1 - 13
14  class Battle
15  {
    ... lines 16 - 113
114     /**
115      * @Serializer\VirtualProperty()
    ... line 116
117      */
118     public function getProgrammerNickname()
119     {
120         return $this->programmer->getNickname();
121     }
    ... lines 122 - 130
131 }
```

the serializer generates the field name by taking the method name and removing get.

SerializedName

Since we want this to be called programmer add another annotation: @Serializer\SerializedName() and pass it programmer:

```
132 lines | src/AppBundle/Entity/Battle.php
    ... lines 1 - 13
14  class Battle
15  {
    ... lines 16 - 113
114     /**
115      * @Serializer\VirtualProperty()
116      * @Serializer\SerializedName("programmer")
117      */
118     public function getProgrammerNickname()
119     {
120         return $this->programmer->getNickname();
121     }
    ... lines 122 - 130
131 }
```

Now we have a programmer field set to the return value of this method.

Do the same thing for project: public function getProjectId(). This will return $this->project->getId():

132 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 13
class Battle
{
... lines 16 - 126
    public function getProjectId()
    {
        return $this->project->getId();
    }
}
```

Above this, add the @Serializer\VirtualProperty to activate the new field and @Serializer\SerializedName("project") to control its name:

132 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 13
class Battle
{
... lines 16 - 122
    /**
     * @Serializer\VirtualProperty()
     * @Serializer\SerializedName("project")
     */
    public function getProjectId()
    {
        return $this->project->getId();
    }
}
```

Head to the terminal and try the test:

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

We've got it! This trick is a *wonderful* way to take control of exactly how you want your representation to look.

Chapter 6: Form Voodoo: property_path

Remember, we want to design our API to work well with whomever is using it: whether that's a third-party API client, a JavaScript front end, or another PHP app that's talking to us. That's why we just changed how our Battle output looks.

But you might also want to control how the input looks: what the client needs to send to your API. For example, right now, to create a new battle, you send a project field and a programmer field: each set to their ID:

45 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 22
        $data = array(
            'project' => $project->getId(),
            'programmer' => $programmer->getId()
        );
... lines 27 - 42
    }
}
```

But what if we wanted to call these fields projectId and programmerId?

45 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 22
        $data = array(
            'projectId' => $project->getId(),
            'programmerId' => $programmer->getId()
        );
... lines 27 - 42
    }
}
```

After all, those are *IDs* that are being sent. If we change this in the test, everything will explode. Prove it by running things:

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

Yep, a big validation error: the form should not contain extra fields: these two new fields are *not* in the form we built.

The easiest fix is to simply rename these fields in the form to projectId and programmerId:

35 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 9
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('programmerId', EntityType::class, [
... lines 17 - 18
            ])
            ->add('projectId', EntityType::class, [
... lines 21 - 22
            ])
        ;
    }
... lines 26 - 33
}
```

But then, we would *also* need to change the property names in BattleModel to match these:

33 lines | src/AppBundle/Form/Model/BattleModel.php

```
... lines 1 - 7
class BattleModel
{
    private $project;

    private $programmer;
... lines 13 - 32
}
```

And that sucks: because these properties do *not* hold ID's: they hold objects. I'd rather *not* need to make my class ugly and confusing to help out the API.

Using property_path

Here is the very simple, elegant, amazing solution. In the form, you *do* need to update your fields to projectId and programmerId so they match what the client is sending. But then, add a property_path option to projectId set to project:

35 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 10
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('programmerId', EntityType::class, [
                'class' => 'AppBundle\Entity\Programmer',
                'property_path' => 'programmer'
            ])
... lines 20 - 23
        ;
    }
... lines 26 - 33
}
```

Do the same thing to the programmerId field: 'property_path' => 'programmer':

35 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 10
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
... lines 16 - 19
            ->add('projectId', EntityType::class, [
                'class' => 'AppBundle\Entity\Project',
                'property_path' => 'project'
            ])
        ;
    }
... lines 26 - 33
}
```

That's the key! The form now expects the client to send projectId and programmerId. But when it sets the final data on BattleModel, it will call setProject() and setProgrammer().

This is a little known way to have a field name that's different than the property name on your class. Bring on the test!

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

Awesome! Another useful option I want you to know about is called mapped. You can use this to allow an *extra* field in your input, without needing to add a corresponding property to your class.

Chapter 7: Adding Battle Validation

It doesn't make *any* sense to create a battle without a Programmer or a Project. But guess what - you can! Or at least, you kind of can: we don't have validation to prevent that yet!

The validation system we created in earlier courses is air-tight: as long as we add the constraint annotations, it just works. So normally, I might *not* write a test for failing validation. But I will now... because we're going to add a twist.

Testing for Validation

Add a new public function testPOSTBattleValidationErrors():

67 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
... lines 47 - 64
    }
}
```

Copy the first bits from the previous function that create the data and make the request:

67 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
        $programmer = $this->createProgrammer([
            'nickname' => 'Fred'
        ], 'weaverryan');

        $data = array(
            'projectId' => null,
            'programmerId' => $programmer->getId()
        );

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

... lines 62 - 64
    }
}
```

But, *don't* actually create a project! Instead, send null for the projectId. Since starting a battle against *nothing* is nonsense, assert that 400 is the response status code. This follows the pattern we did before in ProgrammerControllerTest.

And actually, that test shows off the validation errors response format: there should be an errors key with field names for the errors below that. Each field could technically have multiple errors, so that's an array:

290 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 211
    public function testValidationErrors()
    {
... lines 214 - 230
        $this->asserter()->assertResponsePropertyExists($response, 'errors.nickname');
        $this->asserter()->assertResponsePropertyEquals($response, 'errors.nickname[0]', 'Please enter a clever nickname');
... lines 233 - 234
    }
... lines 236 - 288
}
```

Check for the error in our code with $this->asserter()->assertResponsePropertyExists(): the field should be errors.projectId:

67 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
... lines 47 - 61
        $this->assertEquals(400, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyExists($response, 'errors.projectId');
... line 64
    }
}
```

Next, check for the exact message: assertResponsePropertyEquals() with errors.projectId[0] - so the *first* and only error - set to This value should not be blank.:

67 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
... lines 47 - 61
        $this->assertEquals(400, $response->getStatusCode());
        $this->asserter()->assertResponsePropertyExists($response, 'errors.projectId');
        $this->asserter()->assertResponsePropertyEquals($response, 'errors.projectId[0]', 'This value should not be blank.');
    }
}
```

Why that message? That's the default message for Symfony's NotBlank constraint.

Before we code this up, copy the method name and run the test:

```
$ ./vendor/bin/phpunit --filter testPOSTBattleValidationErrors
```

It explodes with a 500 error! This is what happens when you're lazy and forget to add validation: the BattleManager panics because there is no Project. We do *not* want 500 errors, they are not hipster.

Adding Basic Validation

We know how to fix this! Go to BattleModel. Remember, this is the class that's bound to the form: so the annotations should go here. First, add the use statement. Type use NotBlank, let it auto-complete, delete the last part and add the normal as Assert:

```
40 lines | src/AppBundle/Form/Model/BattleModel.php
... lines 1 - 6
use Symfony\Component\Validator\Constraints as Assert;
... lines 8 - 40
```

That's my shortcut to get the use statement.

Now, above project, add @Assert\NotBlank(). Do the same above programmer: @Assert\NotBlank():

```
40 lines | src/AppBundle/Form/Model/BattleModel.php
... lines 1 - 8
class BattleModel
{
    /**
     * @Assert\NotBlank()
     */
    private $project;

    /**
     * @Assert\NotBlank()
     */
    private $programmer;
... lines 20 - 39
}
```

Done! Now run the test:

```
$ ./vendor/bin/phpunit --filter testPostBattleValidationErrors
```

We're awesome! Or are we... there's a deeper problem! What prevents an API client from starting a battle with a Programmer that they do *not* own? Right now - *nothing*, besides karma and trusting that humankind will do the right thing. Unfortunately, that doesn't usually pass a security audit. Let's be heros and fix this security hole!

Chapter 8: EntityType Validation: Restrict Invalid programmerId

In the test, the Programmer is *owned* by weaverryan and then we authenticate as weaverryan. So, we're starting a battle using a Programmer that *we* own. Time to mess that up. Create a new user called someone_else:

70 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
        // create a Programmer owned by someone else
        $this->createUser('someone_else');
... lines 49 - 67
    }
}
```

There still *is* a user called weaverryan:

70 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
    protected function setUp()
    {
        parent::setUp();

        $this->createUser('weaverryan');
    }
... lines 15 - 68
}
```

But now, change the programmer to be owned by this sketchy someone_else character:

70 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
        // create a Programmer owned by someone else
        $this->createUser('someone_else');
        $programmer = $this->createProgrammer([
            'nickname' => 'Fred'
        ], 'someone_else');
... lines 52 - 67
    }
}
```

With this setup, weaverryan will be starting a battle with someone_else's programmer. This should cause a validation error: this is an *invalid* programmerId to pass.

Form Field Sanity Validation

But how do we do that? Is there some annotation we can use for this? Nope! This validation logic will live in the form. "What!?" you say - "Validation always goes in the class!". Not true! Every field type has a little bit of built-in validation logic. For example, the NumberType will fail if a mischievous - or confused - user types in a word. And the EntityType will fail if someone passes an id that's not found in the database. I call this sanity validation: the form fields at least make sure that a sane value is passed to your object.

If we could restrict the valid programmer id's to *only* those owned by our user, we'd be in business.

But first, add the test: assertResponsePropertyEquals() that errors.programmerId[0] should equal some dummy message:

70 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
... lines 47 - 66
        $this->asserter()->assertResponsePropertyEquals($response, 'errors.programmerId[0]', '???');
    }
}
```

Run the test to see the failure:

```
$ ./vendor/bin/phpunit --filter testPOSTBattleValidationErrors
```

Yep: there's no error for programmerId yet.

Let's fix that. Right now, the client can pass *any* valid programmer id, and the EntityType happily accepts it. To shrink that to a smaller list, we'll pass it a custom query to use.

Passing the User to the Form

To do that, the form needs to know who is authenticated. In BattleController, guarantee that first: add $this->denyAccessUnlessGranted('ROLE_USER'):

41 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 11
class BattleController extends BaseController
{
    /**
     * @Route("/api/battles")
     * @Method("POST")
     */
    public function newAction(Request $request)
    {
        $this->denyAccessUnlessGranted('ROLE_USER');
... lines 21 - 38
    }
}
```

To pass the user to the form, add a third argument to createForm(), which is a little-known options array. Invent a new option: user set to $this->getUser():

41 lines | src/AppBundle/Controller/Api/BattleController.php

```
... lines 1 - 11
class BattleController extends BaseController
{
... lines 14 - 17
    public function newAction(Request $request)
    {
... lines 20 - 22
        $form = $this->createForm(BattleType::class, $battleModel, [
            'user' => $this->getUser()
        ]);
... lines 26 - 38
    }
}
```

This isn't a core Symfony thing: we're creating a new option.

To allow this, open BattleType and find configureOptions. Here, you need to say that user is an allowed option. One way is via $resolver->setRequired('user'):

42 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 11
class BattleType extends AbstractType
{
... lines 14 - 32
    public function configureOptions(OptionsResolver $resolver)
    {
... lines 35 - 38
        $resolver->setRequired(['user']);
    }
}
```

This means that whoever uses this form is allowed to, and in fact *must*, pass a user option.

With that, you can access the user object in buildForm(): $user = $options['user']:

42 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 11
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $user = $options['user'];
... lines 17 - 30
    }
... lines 32 - 40
}
```

None of this is unique to API's: we're just giving our form more power!

Passing the query_builder Option

Let's filter the programmer query: add a query_builder option set to an anonymous function with ProgrammerRepository as the only argument. Add a use for $user so we can access it:

42 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 4
use AppBundle\Repository\ProgrammerRepository;
... lines 6 - 11
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 16 - 17
        $builder
            ->add('programmerId', EntityType::class, [
... lines 20 - 21
                'query_builder' => function(ProgrammerRepository $repo) use ($user) {
... line 23
                },
            ])
... lines 26 - 29
        ;
    }
... lines 32 - 40
}
```

We could write the query right here, but you guys know I don't like that: keep your queries in the repository! Call a new method createQueryBuilderForUser() and pass it $user:

42 lines | src/AppBundle/Form/BattleType.php

```
... lines 1 - 11
class BattleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
... lines 16 - 17
        $builder
            ->add('programmerId', EntityType::class, [
... lines 20 - 21
                'query_builder' => function(ProgrammerRepository $repo) use ($user) {
                    return $repo->createQueryBuilderForUser($user);
                },
            ])
... lines 26 - 29
        ;
    }
... lines 32 - 40
}
```

Copy that method name and shortcut-your way to that class by holding command and clicking ProgrammerRepository. Add public function createQueryBuilderForUser() with the User $user argument:

48 lines | src/AppBundle/Repository/ProgrammerRepository.php

```
... lines 1 - 8
class ProgrammerRepository extends EntityRepository
{
... lines 11 - 19
    public function createQueryBuilderForUser(User $user)
    {
... lines 22 - 24
    }
... lines 26 - 46
}
```

Inside, return $this->createQueryBuilder() and alias the class to programmer. Then, just andWhere('programmer.user = :user') with ->setParameter('user', $user):

48 lines | src/AppBundle/Repository/ProgrammerRepository.php

```
... lines 1 - 5
use AppBundle\Entity\User;
... lines 7 - 8
class ProgrammerRepository extends EntityRepository
{
... lines 11 - 19
    public function createQueryBuilderForUser(User $user)
    {
        return $this->createQueryBuilder('programmer')
            ->andWhere('programmer.user = :user')
            ->setParameter('user', $user);
    }
... lines 26 - 46
}
```

Done! The controller passes the User to the form, and the form calls the repository to create the custom query builder. Now, if someone passes a programmer id that we do *not* own, the EntityType will automatically cause a validation error. Security is built-in.

Head back to the terminal to try it!

```
$ ./vendor/bin/phpunit --filter testPOSTBattleValidationErrors
```

Awesome! Well, it failed - but look! It's just because we don't have the real message yet: it returned This value is not valid.. That's the standard message if any field fails the "sanity" validation.

Tip

You can customize this message via the invalid_message form field option.

Copy that string and paste it into the test:

70 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 44
    public function testPOSTBattleValidationErrors()
    {
... lines 47 - 66
        $this->asserter()->assertResponsePropertyEquals($response, 'errors.programmerId[0]', 'This value is not valid.');
    }
}
```

Run it!

```
$ ./vendor/bin/phpunit --filter testPOSTBattleValidationErrors
```

So that's "sanity" validation: it's form fields watching your back to make sure mean users don't start sending crazy things to us. And it happens automatically.

Chapter 9: HATEOAS & Hypermedia: The Buzzwords Level

These days, you expect a tutorial on REST to immediately throw out some buzzwords: like *hypermedia* and *HATEOAS*. But I've tried *not* to mention these because honestly, in practice: they're more trouble than good.

But, there are some cool parts! It's time to learn what these mean, what parts are good, and what parts almost drove me to drinking.

Hypermedia: Really Caffeinated DVD's

First, hypermedia! This does *not* involve slamming red bulls and binge-watching Netflix. Here's the story: JSON is known as a media type. If I tell you that I'm giving you a JSON string, you understand how to parse it. XML is another media type: if I give you XML, you can understand its format.

The difference between media and *hypermedia* is that hypermedia is a format that includes links. For example, if you used a JSON structure that consistently put links under a _links key, well, you could claim that you just created your own *hypermedia* format: a JSON structure where I know - semantically - what _links means: it's not data: it's links to *other* resources.

The most famous hypermedia format is ... drumroll HTML! It's basically an XML media type that has built-in tags for links: the <a> tag. Actually, <form>, <img> <link> and <script> are also considered "links" to other resources.

So calling something a hypermedia format is just a way of saying:

> Hey, when we return the JSON data, what if we added some rules that say any links we want to include always live in the same place?

And even though I avoided the word - *hypermedia* - in a sense, we're already returning a hypermedia format because we always include links under an _links key. We'll talk more about this a little later: there are actually "official" hypermedia formats you can adopt for your API.

HATEOAS

Ready for buzzword #2? HATEOAS... or HAT-E-OAS... or H-ATE-OAS... nobody really knows how to say it. Anyways, the throat-clearing acronym stands for:

> Hypermedia As The Engine Of Application State

Whoa. Let's unpack that monster. It's actually a cool idea.

Application state refers to *which* endpoint a client is currently using. As the client does more things - like creating a programmer and then starting a battle - they're moving through different "application" states, the same way that someone moves through the pages on your web site.

Normally, the client figures out *which* endpoint - or state - they need next by reading some API documentation. But HATEOAS says:

> What if we didn't write API documentation and instead, every response we send back self-documents what endpoints you might want next via links?

So when we send back a programmer resource, it would contain a link for every other possible thing the client might want to do next - like starting a battle with that programmer.

In an ideal world, you would stop writing documentation and just say:

> Hey, use my API! Every time you make a request, I'll include details about what do to next.

In reality, HATEOAS is a fantasy, at least for now. For it to work, links would need to be able to contain what HTTP method to use, what fields to send, and what those fields mean. It's hard, way too hard for most people right now, including me.

Here's the right way to navigate this mine field. Instead of saying:

I'm going to include so many links that I don't have to document anything!

You should say:

I'm going to add links that might be helpful, but also write documentation.

So let's do that: and take this idea in our app up to a new level.

Chapter 10: Link from Battle to Programmer

The Battle representation includes the username of the programmer that fought. If you want more information about Fred, you can make a request to /api/programmers/Fred:

70 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 37
        $this->asserter()
            ->assertResponsePropertyEquals($response, 'programmer', 'Fred');
... lines 40 - 42
    }
... lines 44 - 68
}
```

That's something we'll document.

But! Wouldn't it be even more convenient if we added a link to that URL inside of the Battle representation? Then, instead of needing to go look up and hardcode the URL, the client could simply read and follow the link.

Whenever a link like this would be helpful, add it! First, look for it in the test: $this->asserter()->assertResponsePropertyEquals(). For consistency, we decided to put links under an _links key. So, look for _links.programmer. This should equal $this->adjustUri('/api/programmers/Fred):

76 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 37
        $this->asserter()
            ->assertResponsePropertyEquals($response, 'programmer', 'Fred');

        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.programmer',
            $this->adjustUri('/api/programmers/Fred')
        );
... lines 46 - 76
```

All this method does is help account for the extra app_test.php that's in the URL when testing:

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

```
... lines 1 - 21
class ApiTestCase extends KernelTestCase
{
... lines 24 - 358
    /**
     * Call this when you want to compare URLs in a test
     *
     * (since the returned URL's will have /app_test.php in front)
     *
     * @param string $uri
     * @return string
     */
    protected function adjustUri($uri)
    {
        return '/app_test.php'.$uri;
    }

}
```

Perfect! Now, let's go add that link. First, open up the Programmer entity. We added the self link earlier via a cool annotation system we created:

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

```
... lines 1 - 9
/**
... lines 11 - 15
 * @Link(
 *  "self",
 *  route = "api_programmers_show",
 *  params = { "nickname": "object.getNickname()" }
 * )
 */
class Programmer
... lines 23 - 198
```

In Battle, add something similar: @Link - let that auto-complete for the use statement. Set the name - or rel - of the link to programmer. This is the significance of the link: it could be anything, as long as you consistently use programmer when linking to a programmer:

138 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 9
/**
... lines 11 - 13
 * @Link(
 *      "programmer",
... lines 16 - 17
 * )
 */
class Battle
... lines 21 - 138
```

For the route, use api_programmers_show: the route name to a single programmer:

138 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 9
/**
... lines 11 - 13
 * @Link(
 *     "programmer",
 *     route="api_programmers_show",
... line 17
 * )
 */
class Battle
... lines 21 - 138
```

Finally, add params: the wildcard parameters that need to be passed to the route. This route has a nickname wildcard. Set it to an expression: object.getProgrammerNickname():

138 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 9
/**
... lines 11 - 13
 * @Link(
 *     "programmer",
 *     route="api_programmers_show",
 *     params={"nickname": "object.getProgrammerNickname()"}
 * )
 */
class Battle
... lines 21 - 138
```

That's the method we created down below earlier:

138 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 19
class Battle
{
... lines 22 - 119
    /**
     * @Serializer\VirtualProperty()
     * @Serializer\SerializedName("programmer")
     */
    public function getProgrammerNickname()
    {
        return $this->programmer->getNickname();
    }
... lines 128 - 136
}
```

And that's all we need. Copy the method name again - testPostCreateBattle() - and run the test:

```
$ ./vendor/bin/phpunit --filter testPostCreateBattle
```

And it works.

Now, let me show you an awesome library that makes adding links even easier. In fact, I stole the @Link annotation idea from it.

Chapter 11: The Great Hateoas PHP Library

Google for "HATEOAS PHP" to find a fun library that a friend of mine made. This library has a bundle that integrates it into Symfony: so click to view the BazingaHateoasBundle and go straight to its docs. Before we talk about what it does: get it installed.

Installing BazingaHateoasBundle

Copy the composer require statement and then flip over to your terminal and paste that:

```
$ composer require willdurand/hateoas-bundle
```

This is a bundle, so grab the new bundle statement, open AppKernel and pop that at the bottom:

app/AppKernel.php

```
... lines 1 - 5
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
... lines 11 - 21
            new Bazinga\Bundle\HateoasBundle\BazingaHateoasBundle(),
        );
... lines 24 - 33
    }
... lines 35 - 52
}
```

Perfect. Currently, we have our own super sweet annotation system for adding links. In Battle, we use @Link to create a programmer link:

src/AppBundle/Entity/Battle.php

```
... lines 1 - 9
/**
... lines 11 - 13
 * @Link(
 *     "programmer",
 *     route="api_programmers_show",
 *     params={"nickname": "object.getProgrammerNickname()"}
 * )
 */
class Battle
... lines 21 - 138
```

Guess what! I completely stole that idea from this library. But now, to make our app a little simpler and to get some new features, let's replace our @Link code with *this* library.

Adding the HATEOAS Annotation Links

Go back to the library itself, and scroll down to the first coding example. This uses an annotation system that looks pretty similar to ours. Copy the use statement on top, open Battle and paste that:

141 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 8
use Hateoas\Configuration\Annotation as Hateoas;
... lines 10 - 141
```

Next, change the annotation to @Hateoas\Relation:

141 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
/**
... lines 12 - 14
 * @Hateoas\Relation(
... lines 16 - 20
 * )
 */
class Battle
... lines 24 - 141
```

Keep programmer: that will still be the link's rel. But add href=@Hateoas\Route and pass that the name of the route: api_programmers_show:

141 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
/**
... lines 12 - 14
 * @Hateoas\Relation(
 *     "programmer",
 *     href=@Hateoas\Route(
 *         "api_programmers_show",
... line 19
 *     )
 * )
 */
class Battle
... lines 24 - 141
```

Update params to parameters, and inside, set nickname equal, and wrap the expression in expr():

141 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
/**
... lines 12 - 14
 * @Hateoas\Relation(
 *     "programmer",
 *     href=@Hateoas\Route(
 *         "api_programmers_show",
 *         parameters={"nickname"= "expr(object.getProgrammerNickname())"}
 *     )
 * )
 */
class Battle
... lines 24 - 141
```

That translates our annotation format to the one used by the bundle. And the result is *almost* the same. Open BattleControllerTest and copy the first method name:

76 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
... lines 17 - 74
}
```

we have a test for a link near the bottom of this:

76 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 40
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.programmer',
            $this->adjustUri('/api/programmers/Fred')
        );
... lines 46 - 48
    }
... lines 50 - 74
}
```

Change over to the terminal and, as long as Composer is done, run:

```
$ vendor/bin/phpunit --filter testPOSTCreateBattle
```

Check it out! It fails - but *barely*. This library *still* puts links under an _links key, but instead of listing the URLs directly, it wraps each inside an object with an href key. That's causes the failure.

Ok, fair enough. Let's fix that by updating the test to look for _links.programmer.href:

76 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 40
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.programmer.href',
            $this->adjustUri('/api/programmers/Fred')
        );
... lines 46 - 48
    }
... lines 50 - 74
}
```

Run the test again:

```
$ vendor/bin/phpunit --filter testPOSTCreateBattle
```

And now we're green.

Holy Toledo Batman: This is HAL JSON!

But guess what? It's no accident that this library used this exact format: with an _links key and an href below that. This is a semi-official standard format called HAL JSON.

Chapter 12: The HAL JSON Standard

Google for "How to remove a mustard stain from a white shirt". I mean, Google for "HAL JSON" - sorry, it's after lunch.

This is one of a few competing *hypermedia* formats. And remember, *hypermedia* is one of our favorite buzzwords: it's a media type, or format, - like JSON - plus some rules about how you should semantically organize things inside that format. In human speak, HAL JSON says:

> Hi I'm HAL! If you want to embed links in your JSON, you should put them under an _links key and point to the URL with href. Have a lovely day!

If you think about it, this idea is similar to HTML. In HTML, there's the XML-like format, but then there are rules that say:

> Hi, I'm HTML! If you want a link, put it in an <a> tag under an href attribute.

The advantage of having standards is that - since the entire Internet follows them - we can create a browser that understands the significance of the <a> tag, and renders them clickable. In theory, if all API's followed a standard, we could create clients that easily deal with the data.

Updating Programmer to use the new Links

So let's also update the Programmer entity to use the new system. Copy the whole @Relation from Battle:

141 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
/**
... lines 12 - 14
 * @Hateoas\Relation(
 *     "programmer",
 *     href=@Hateoas\Route(
 *         "api_programmers_show",
 *         parameters={"nickname"= "expr(object.getProgrammerNickname())"}
 *     )
 * )
... line 22
class Battle
... lines 24 - 141
```

And replace the @Link inside of Programmer. Change the rel back to self and update the expression to object.getNickname():

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

```
... lines 1 - 8
use Hateoas\Configuration\Annotation as Hateoas;

/**
... lines 12 - 16
 * @Hateoas\Relation(
 *      "self",
 *      href=@Hateoas\Route(
 *          "api_programmers_show",
 *          parameters = { "nickname"= "expr(object.getNickname())" }
 *      )
 * )
 */
class Programmer
... lines 26 - 201
```

Make sure you've got all your parenthesis in place. Oh, and don't forget to bring over the use statement from Battle.

In ProgrammerControllerTest, the testGETProgrammer method looks for _links.self:

290 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 37
    public function testGETProgrammer()
    {
... lines 40 - 55
        $this->asserter()->assertResponsePropertyEquals(
... line 57
            '_links.self',
... line 59
        );
    }
... lines 62 - 288
}
```

Add .href to this to match the new format:

290 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 37
    public function testGETProgrammer()
    {
... lines 40 - 55
        $this->asserter()->assertResponsePropertyEquals(
... line 57
            '_links.self.href',
... line 59
        );
    }
... lines 62 - 288
}
```

Try it out!

```
$ vendor/bin/phpunit --filter testGETProgrammer
```

Yes!

Should I Use HAL JSON?

So why use a standardized format like Hal? Because now, we can say:

> Hey, our API returns HAL JSON responses!

Then, they can go read its documentation to find out what it looks like. Or better, they might already be familiar with it!

Advertising that you're using Hal

So now that we are using Hal, we should advertise it! In fact, that's what this application/hal+json means in their documentation: it's a custom Content-Type. It means that the format is JSON, but there's some extra rules called Hal. If a client sees this, they can Google for it.

In ProgrammerControllerTest, assert that application/hal+json is equal to $response->getHeader('Content-Type')[0]:

291 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTProgrammerWorks()
    {
... lines 18 - 30
        $this->assertEquals('application/hal+json', $response->getHeader('Content-Type')[0]);
... lines 32 - 36
    }
... lines 38 - 289
}
```

Guzzle returns an array for each header - there's a reason for that, but yea, I know it looks ugly.

To actually advertise that our API returns HAL, open BaseController and search for createApiResponse() - the method we're calling at the bottom of *every* controller. Change the header to be application/hal+json:

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

```
... lines 1 - 19
abstract class BaseController extends Controller
{
... lines 22 - 117
    protected function createApiResponse($data, $statusCode = 200)
    {
        $json = $this->serialize($data);

        return new Response($json, $statusCode, array(
            'Content-Type' => 'application/hal+json'
        ));
    }
... lines 126 - 185
}
```

Nice! Copy the test name and re-run the test:

```
$ ./vendor/bin/phpunit --filter testPOSTProgrammerWorks
```

Congratulations! Your API is no longer an island: welcome to the club.

Chapter 13: Embedding Objects with Hal?

Check out the Hal example on their docs. There are actually *three* different sections of the json: the actual data - like currentlyProcessing, _links and _embedded.

Relations: Links versus Embedding

Here's a cool idea: we know that it's nice to add links to a response. These are called *relations*: they point to *related* resources. But there's *another* way to add a relation to a response: *embed* the related resource right in the JSON.

Remember: the *whole* point of adding link relations is to make life easier for your API clients. If embedding the data is *even* easier than advertising a link, do it.

In fact, let's pretend that when we return a Battle resource, we still want to include a link to the related Programmer, but we also want to embed that Programmer entirely.

Adding an Embedded Relation

To do that, after href, add an embedded="expr()" with object.getProgrammer():

142 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
/**
... lines 12 - 14
 * @Hateoas\Relation(
 *     "programmer",
 *     href=@Hateoas\Route(
 *         "api_programmers_show",
 *         parameters={"nickname"= "expr(object.getProgrammerNickname())"}
 *     ),
 *     embedded = "expr(object.getProgrammer())"
 * )
 */
class Battle
... lines 25 - 142
```

Let's see what this looks like! Open BattleControllerTest and right at the bottom, add our handy $this->debugResponse($response):

77 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 40
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.programmer.href',
            $this->adjustUri('/api/programmers/Fred')
        );
        $this->debugResponse($response);
... lines 47 - 49
    }
... lines 51 - 75
}
```

Perfect! Copy that method name and run it:

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

Oh, cool: we still have the relation in _links, but now we *also* have an entire programmer resource in _embedded. So when you setup these @Hateoas\Relation annotations:

142 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
/**
... lines 12 - 14
 * @Hateoas\Relation(
 *     "programmer",
 *     href=@Hateoas\Route(
 *         "api_programmers_show",
 *         parameters={"nickname"= "expr(object.getProgrammerNickname())"}
 *     ),
 *     embedded = "expr(object.getProgrammer())"
 * )
 */
class Battle
... lines 25 - 142
```

You can choose whether you want this to be a link or an embedded object.

And OK, we cheated on this test by looking at it first, but *now* I guess we should specifically have a test for it. Add: $this->asserter()->assertResponsePropertyEquals() with $response. Look for _embedded.programmer.nickname to be equal to our friend Fred:

81 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 40
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.programmer.href',
            $this->adjustUri('/api/programmers/Fred')
        );
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_embedded.programmer.nickname',
            'Fred'
        );
... lines 51 - 53
    }
... lines 55 - 79
}
```

Run that!

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

It passes! Now let's customize how these links render.

Chapter 14: Customize how your Links Render

As cool as all this HAL JSON stuff is, you need to build your API for whoever is using it - maybe a JavaScript frontend, a mobile app or your customers themselves. And honestly, I don't think that the standardized formats - like Hal - are all that understandable or useful. This _embedded thing? To me it's just ugly.

I also don't love hiding the URL under an object with an href key. So let's suppose that we're building a JavaScript frontend, and it'll work better if the link URL's appeared *directly* under the _links key - without the href.

Let's do that! The HATEOAS library we installed really just helps you add relations to a class: both link relations and embedded relations. And fortunately, the library let's you control exactly how these are added to your response.

Custom Serializer

In AppBundle, in the Serializer directory, create a new class called CustomHATEOASJsonSerializer:

13 lines | src/AppBundle/Serializer/CustomHATEOASJsonSerializer.php

```
... lines 1 - 2
namespace AppBundle\Serializer;
... lines 4 - 6
use Hateoas\Serializer\JsonHalSerializer;
... lines 8 - 10
class CustomHATEOASJsonSerializer extends JsonHalSerializer
{
}
```

Make it extend a class called JsonHalSerializer: this is the current class responsible for adding links in the HAL format. In fact: open up the class.

It has two method. serializeLinks() is responsible for reading the Relation annotations and adding them to the JSON with _links. serializeEmbeddeds() adds any embedded relations under the _embedded key.

For now, let's focus on changing how the links render only. Go to the "Code"->"Generate" menu - command+N on a Mac - and hit "Override Methods". Override serializeLinks():

28 lines | src/AppBundle/Serializer/CustomHATEOASJsonSerializer.php

```
... lines 1 - 7
use JMS\Serializer\JsonSerializationVisitor;
use JMS\Serializer\SerializationContext;

class CustomHATEOASJsonSerializer extends JsonHalSerializer
{
... lines 13 - 17
    public function serializeLinks(array $links, JsonSerializationVisitor $visitor, SerializationContext $context)
    {
... lines 20 - 25
    }
}
```

Re-open the parent method and then the interface: I want to copy all that good PHPDoc so we get auto-complete. Paste it above our method and auto-complete the Link to get its use statement:

28 lines | src/AppBundle/Serializer/CustomHATEOASJsonSerializer.php

```
... lines 1 - 5
use Hateoas\Model\Link;
... lines 7 - 10
class CustomHATEOASJsonSerializer extends JsonHalSerializer
{
    /**
     * @param Link[] $links
     * @param JsonSerializationVisitor $visitor
     * @param SerializationContext $context
     */
    public function serializeLinks(array $links, JsonSerializationVisitor $visitor, SerializationContext $context)
    {
... lines 20 - 25
    }
}
```

Alright: this should be easy.

Create a $serializedLinks array and foreach over the $links variable:

28 lines | src/AppBundle/Serializer/CustomHATEOASJsonSerializer.php

```
... lines 1 - 10
class CustomHATEOASJsonSerializer extends JsonHalSerializer
{
    /**
     * @param Link[] $links
     * @param JsonSerializationVisitor $visitor
     * @param SerializationContext $context
     */
    public function serializeLinks(array $links, JsonSerializationVisitor $visitor, SerializationContext $context)
    {
        $serializedLinks = array();
        foreach ($links as $link) {
... line 22
        }
... lines 24 - 25
    }
}
```

Each of these is a Link object, and contains the configuration for one annotation. Now, just create the format we want: $serializedLinks[], with $link->getRel(). Instead of setting this to an array with an href key, simply set it to $link->getHref():

28 lines | src/AppBundle/Serializer/CustomHATEOASJsonSerializer.php

```
... lines 1 - 10
class CustomHATEOASJsonSerializer extends JsonHalSerializer
{
    /**
     * @param Link[] $links
     * @param JsonSerializationVisitor $visitor
     * @param SerializationContext $context
     */
    public function serializeLinks(array $links, JsonSerializationVisitor $visitor, SerializationContext $context)
    {
        $serializedLinks = array();
        foreach ($links as $link) {
            $serializedLinks[$link->getRel()] = $link->getHref();
        }
... lines 24 - 25
    }
}
```

Perfect! Finally, at the bottom, we need to add the _links property. Do that with: $visitor->addData('_links', $serializedLinks):

28 lines | src/AppBundle/Serializer/CustomHATEOASJsonSerializer.php

```
... lines 1 - 10
class CustomHATEOASJsonSerializer extends JsonHalSerializer
{
    /**
     * @param Link[] $links
     * @param JsonSerializationVisitor $visitor
     * @param SerializationContext $context
     */
    public function serializeLinks(array $links, JsonSerializationVisitor $visitor, SerializationContext $context)
    {
        $serializedLinks = array();
        foreach ($links as $link) {
            $serializedLinks[$link->getRel()] = $link->getHref();
        }

        $visitor->addData('_links', $serializedLinks);
    }
}
```

With any luck, that should give us a simpler format without that href.

Registering the Serializer

To hook this up. You guys can probably guess step 1: in app/config/services.yml, register this as a service. How about: custom_hateoas_json_serializer. Set its class to that same thing:

38 lines | app/config/services.yml

```
... lines 1 - 5
services:
... lines 7 - 36
    custom_hateoas_json_serializer:
        class: AppBundle\Serializer\CustomHATEOASJsonSerializer
```

And we don't have any constructor args yet.

Finally, copy the service name. To tell the bundle to use *our* class instead of the existing one, open up config.yml. Now, without even looking at its docs, we can get a list of the configuration for this bundle by going to the terminal and running:

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

Thanks to the bundle, there's a new valid config key called bazinga_hateoas. Pass that to the same command:

```
$ ./bin/console debug:config bazinga_hateoas
```

Ah, that serializer.json key looks like our target.

Back in config.yml, add bazinga_hateoas, serializer, json and then paste our service name:

82 lines | app/config/config.yml

```
... lines 1 - 78
bazinga_hateoas:
    serializer:
        json: custom_hateoas_json_serializer
```

That should do it!

Changing our Tests Back

But don't run the tests *quite* yet: we *know* some things will be broken. In BattleControllerTest, take off the href we just added: it should be _links.programmer:

81 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 40
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.programmer',
            $this->adjustUri('/api/programmers/Fred')
        );
... lines 46 - 53
    }
... lines 55 - 79
}
```

And in ProgrammerControllerTest, under testGETProgrammer, do the same:

291 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 6
class ProgrammerControllerTest extends ApiTestCase
{
... lines 9 - 38
    public function testGETProgrammer()
    {
... lines 41 - 56
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_links.self',
            $this->adjustUri('/api/programmers/UnitTester')
        );
    }
... lines 63 - 289
}
```

Phew! That's a lot of changes, so let's re-run the *entire* test suite:

```
$ ./vendor/bin/phpunit
```

Hey, it passes! I must've left a debugResponse() in there somewhere: but that's nothing to worry about - we're green!

Chapter 15: Customizing (making less ugly) Embeddeds!

Let me show you something else I don't really like. In BattleControllerTest, we're checking for the embedded programmer. Right now it's hidden under this _embedded key:

81 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 45
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            '_embedded.programmer.nickname',
            'Fred'
        );
... lines 51 - 53
    }
... lines 55 - 79
}
```

Hal does this so that a client knows which data is for the Battle, and which data is for the embedded programmer. But what if it would be more convenient for our client if the data was *not* under an _embedded key? What if they want the data on the root of the object like it was before?

Well, that's fine! Just stop using the embedded functionality from the bundle. Delete the assert that looks for the string and instead assert that the programmer.nickname is equal to Fred:

79 lines | tests/AppBundle/Controller/Api/BattleControllerTest.php

```
... lines 1 - 6
class BattleControllerTest extends ApiTestCase
{
... lines 9 - 15
    public function testPOSTCreateBattle()
    {
... lines 18 - 43
        $this->asserter()->assertResponsePropertyEquals(
            $response,
            'programmer.nickname',
            'Fred'
        );
... lines 49 - 51
    }
... lines 53 - 77
}
```

In other words, I want to change the root programmer key from a string to the whole object. And we'll eliminate the _embedded key entirely.

In Battle.php, remove the embedded key from the annotation:

139 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 10
/**
... lines 12 - 14
 * @Hateoas\Relation(
 *     "programmer",
 *     href=@Hateoas\Route(
 *         "api_programmers_show",
 *         parameters={"nickname"= "expr(object.getProgrammerNickname())"}
 *     ),
... line 21
 * )
 */
class Battle
... lines 25 - 139
```

OK, _embedded is gone! Next, on the programmer property, add @Expose:

139 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 23
class Battle
{
... lines 26 - 33
    /**
... lines 35 - 36
     * @Serializer\Expose()
     */
    private $programmer;
... lines 40 - 137
}
```

The serializer will serialize that whole object. We originally *didn't* expose that property because we added this cool @VirtualProperty above the getProgrammerNickname() method:

142 lines | src/AppBundle/Entity/Battle.php

```
... lines 1 - 23
class Battle
{
... lines 26 - 123
    /**
     * @Serializer\VirtualProperty()
     * @Serializer\SerializedName("programmer")
     */
    public function getProgrammerNickname()
    {
        return $this->programmer->getNickname();
    }
... lines 132 - 140
}
```

Get rid of that entirely.

In BattleControllerTest, let's see if this is working. First dump the response. Copy the method name, and give this guy a try:

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

Ah! It explodes!

> Warning: call_user_func_array() expects parameter 1 to be a valid callback. Class Battle does not have a method getProgrammerNickname().

Whoops! I think I was too aggressive. Remember, at the top of Battle.php, we have an expression that references this method:

```
139 lines | src/AppBundle/Entity/Battle.php
... lines 1 - 10
11  /**
... lines 12 - 14
15   * @Hateoas\Relation(
16   *     "programmer",
17   *     href=@Hateoas\Route(
... line 18
19   *         parameters={"nickname"= "expr(object.getProgrammerNickname())"}
20   *     ),
... line 21
22   * )
23   */
24  class Battle
... lines 25 - 139
```

So... let's undo that change: put back getProgrammerNickname(), but remove the @VirtualProperty:

```
139 lines | src/AppBundle/Entity/Battle.php
... lines 1 - 23
24  class Battle
25  {
... lines 26 - 123
124
125     public function getProgrammerNickname()
126     {
127         return $this->programmer->getNickname();
128     }
... lines 129 - 137
138 }
```

All right, try it again:

```
$ ./vendor/bin/phpunit --filter testPOSTCreateBattle
```

It passes! And the response looks exactly how *we* want: no more _embedded key.

We're not HAL-JSON'ing Anymore

But guess what, guys! We're breaking the rules of Hal! And this means that we are *not* returning HAL responses anymore. And that's OK: I want you to feel the freedom to make this choice.

We *are* still returning a consistent format that I want my users to know about, it's just not HAL. To advertise this, change the Content-Type to application/vnd.codebattles+json:

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

```
... lines 1 - 19
20  abstract class BaseController extends Controller
21  {
... lines 22 - 117
118     protected function createApiResponse($data, $statusCode = 200)
119     {
... lines 120 - 121
122         return new Response($json, $statusCode, array(
123             'Content-Type' => 'application/vnd.codebattles+json'
124         ));
125     }
... lines 126 - 185
186 }
```

This tells a client that this is still JSON, but it's some custom vendor format. If we want to make friends, we should add some extra documentation to *our* API that explains how to expect the links and embedded data to come back.

Copy that and go into ProgrammerControllerTest and update our assertEquals() that's checking for the content type property:

291 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 6
7   class ProgrammerControllerTest extends ApiTestCase
8   {
... lines 9 - 15
16      public function testPOSTProgrammerWorks()
17      {
... lines 18 - 30
31          $this->assertEquals('application/vnd.codebattles+json', $response->getHeader('Content-Type')[0]);
... lines 32 - 36
37      }
... lines 38 - 289
290 }
```

Finally, copy the test method name and let's make sure everything is looking good:

```
$ ./vendor/bin/phpunit --filter testPOSTProgrammerWorks
```

All green!

I really love this HATEOAS library because it's so easy to add links to your API. But it doesn't mean that you have to live with HAL JSON. You can use a different official format or invent your own.

Chapter 16: Subordinate URL Structure

When an API client fetches information about a programmer, they might also want a quick way to get details about all of the battles the programmer has fought. OK, we could add a link on programmer to an endpoint that returns all of that programmer's battles.

This collection of battles is called a subordinate resource because we're looking at the battles that belong to a programmer. It *feels* like a parent-child relationship. Truthfully, this whole idea of **subordinate resources** isn't that important - and it's usually subjective. But, if you're creating an endpoint and you realize that it *feels* like a subordinate resource, a few things usually change.

To start: how should we setup the URL? Is it /api/battles?username= or /api/battles/{nickname}? If you read up on REST API stuff, they'll tell you the URL structure never matters. Ok, let's use /hamburger! No, that's stupid... unless your app is about delicious hamburgers. For the rest of us, there *are* some sensible rules we should follow.

Adding the Link

First, in Programmer, let's add a new link from the programmer to the battles for that programmer, and then we'll create that endpoint. For the rel, let's use battles:

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

```
... lines 1 - 10
/**
 * Programmer
 *
... lines 14 - 16
 * @Hateoas\Relation(
 *      "self",
 *      href=@Hateoas\Route(
 *          "api_programmers_show",
 *          parameters = { "nickname"= "expr(object.getNickname())" }
 *      )
 * )
 * @Hateoas\Relation(
 *      "battles",
... lines 26 - 29
 * )
 */
class Programmer
... lines 33 - 208
```

That could be anything: just be consistent. Whenever you link to a collection of battles, use battles.

Everything else looks good. The route will *probably* need the nickname of the programmer... we're not sure yet - because this endpoint doesn't exist. Let's create it.

The URL Structure

But wait! Which controller should it go into: ProgrammerController or BattleController? There's no right answer to this, but because these are battles for a specific programmer, the battle is subordinate to the programmer. In these situations, I tend to put the code in the parent resource's controller: ProgrammerController.

And actually, the biggest reason I do this is because of how we're going to structure the URL. Make a public function battlesListAction():

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

```
... lines 1 - 22
class ProgrammerController extends BaseController
{
... lines 25 - 153
    public function battlesListAction()
    {

    }
}
```

Above that, add @Route() and the URL, which of course, *could* be anything. Make it /api/programmers/{nickname}/battles:

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

```
... lines 1 - 22
class ProgrammerController extends BaseController
{
... lines 25 - 150
    /**
     * @Route("/api/programmers/{nickname}/battles", name="api_programmers_battles_list")
     */
    public function battlesListAction()
    {

    }
}
```

Check this out: the first three parts of the URL identify a specific programmer resource. Then, /battles looks almost like a battles *property* on programmer. That *feels* right... and that's all that matters.

For the name, use api_programmers_battles_list and copy that. Every part of this is consistent and almost self-documenting.

Head back to Programmer and paste the route name. The big lesson about subordinate resources is that it's OK to have them and that this is the best URL structure to use. But if some other organization feels better to you, do it. This is one of those REST topics you should *not* lose time thinking about.

Chapter 17: Coding the Subordinate Resource Endpoint

Before we code up the endpoint, start with the test. But wait! This test is going to be *pretty* cool: we'll make a request for a programmer resource and *follow* that link to its battles.

Following the Link in a Test

In ProgrammerControllerTest, add a new public function testFollowProgrammerBattlesLink():

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
class ProgrammerControllerTest extends ApiTestCase
{
... lines 10 - 64
    public function testFollowProgrammerBattlesLink()
    {
... lines 67 - 87
    }
... lines 89 - 315
}
```

Copy the first 2 parts from testGETProgrammer() that create the programmer and make the request. Add those here:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
class ProgrammerControllerTest extends ApiTestCase
{
... lines 10 - 64
    public function testFollowProgrammerBattlesLink()
    {
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
... lines 71 - 78
        $response = $this->client->get('/api/programmers/UnitTester', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 82 - 87
    }
... lines 89 - 315
}
```

Okay: before the request, we need to add some battles to the database so we have something results to check out. Create a project first with $this->createProject('cool_project'):

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 66
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
        $project = $this->createProject('cool_project');
... lines 72 - 317
```

Now, let's add 3 battles. And remember! To do that, we need the BattleManager service. Set that up with $battleManager = $this->getService() - that's a helper method in ApiTestCase - and look up battle.battle_manager:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 66
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
        $project = $this->createProject('cool_project');

        /** @var BattleManager $battleManager */
        $battleManager = $this->getService('battle.battle_manager');
... lines 75 - 317
```

Let's add some inline PHPDoc so PhpStorm auto-completes the next lines.

Love it!

Now, life is easy. Add, $battleManager->battle() and pass it $programmer:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 66
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
        $project = $this->createProject('cool_project');

        /** @var BattleManager $battleManager */
        $battleManager = $this->getService('battle.battle_manager');
        $battleManager->battle($programmer, $project);
... lines 76 - 317
```

And, whoops - make sure you have a $programmer variable set above. Now, add $project. Copy that and paste it 2 more times:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 66
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
        $project = $this->createProject('cool_project');

        /** @var BattleManager $battleManager */
        $battleManager = $this->getService('battle.battle_manager');
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);
... lines 78 - 317
```

And we *are* setup! After we make the request for the programmer, we *should* get back a link we can follow. Get that link with $uri = $this->asserter()->readResponseProperty(). Read _links.battles:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 66
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
        $project = $this->createProject('cool_project');

        /** @var BattleManager $battleManager */
        $battleManager = $this->getService('battle.battle_manager');
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);

        $response = $this->client->get('/api/programmers/UnitTester', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
        $url = $this->asserter()
            ->readResponseProperty($response, '_links.battles');
... lines 84 - 317
```

Make sure you pass $response as the first argument.

Now, follow that link! Be lazy and copy the $response = code from above, because we still need that Authorization header. But change the url to be our dynamic $uri:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 66
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
        $project = $this->createProject('cool_project');

        /** @var BattleManager $battleManager */
        $battleManager = $this->getService('battle.battle_manager');
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);

        $response = $this->client->get('/api/programmers/UnitTester', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
        $url = $this->asserter()
            ->readResponseProperty($response, '_links.battles');
        $response = $this->client->get($url, [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
... lines 87 - 317
```

Before we assert anything, let's dump the response and decide later how this should all *exactly* look:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 66
        $programmer = $this->createProgrammer(array(
            'nickname' => 'UnitTester',
            'avatarNumber' => 3,
        ));
        $project = $this->createProject('cool_project');

        /** @var BattleManager $battleManager */
        $battleManager = $this->getService('battle.battle_manager');
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);
        $battleManager->battle($programmer, $project);

        $response = $this->client->get('/api/programmers/UnitTester', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
        $url = $this->asserter()
            ->readResponseProperty($response, '_links.battles');
        $response = $this->client->get($url, [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
        $this->debugResponse($response);
... lines 88 - 317
```

Coding the Subordinate Collection

Test, check! Let's hook this up. Open ProgrammerController. At first, it's pretty easy. Exchange the nickname for a Programmer object. I'll use a magic param converter for this: just type-hint the argument with Programmer, and it will magically make the query for us:

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

```
... lines 1 - 22
class ProgrammerController extends BaseController
{
... lines 25 - 150
    /**
     * @Route("/api/programmers/{nickname}/battles", name="api_programmers_battles_list")
     */
    public function battlesListAction(Programmer $programmer)
    {
... lines 156 - 159
    }
}
```

Next, get battles the way you always do: $this->getDoctrine()->getRepository('AppBundle:Battle'):

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

```
... lines 1 - 22
class ProgrammerController extends BaseController
{
... lines 25 - 153
    public function battlesListAction(Programmer $programmer)
    {
        $battles = $this->getDoctrine()->getRepository('AppBundle:Battle')
... lines 157 - 159
    }
}
```

Use findBy() to return an array that match programmer => $programmer:

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

```
... lines 1 - 22
class ProgrammerController extends BaseController
{
... lines 25 - 153
    public function battlesListAction(Programmer $programmer)
    {
        $battles = $this->getDoctrine()->getRepository('AppBundle:Battle')
            ->findBy(['programmer' => $programmer]);
... lines 158 - 159
    }
}
```

What now? Why not a simple return? return $this->createApiResponse() and pass it $battles:

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

```
... lines 1 - 22
class ProgrammerController extends BaseController
{
... lines 25 - 153
    public function battlesListAction(Programmer $programmer)
    {
        $battles = $this->getDoctrine()->getRepository('AppBundle:Battle')
            ->findBy(['programmer' => $programmer]);

        return $this->createApiResponse($battles);
    }
}
```

Right? Is it really that simple?

Well, let's find out! Go back to ProgrammerControllerTest, copy the new method name and run:

```
$ ./vendor/bin/phpunit --filter testFollowProgrammerBattlesLink
```

Consistency Anyone?

OK, cool - check out how this looks: it's a big JSON array that holds a bunch of JSON battle objects. At first glance, it's great! But there's a problem? It's totally inconsistent with our other endpoint that returns a collection of programmers.

Scroll down a little to testProgrammersCollection(). Here: we expect an items key with the resources inside of it:

317 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
class ProgrammerControllerTest extends ApiTestCase
{
... lines 10 - 105
    public function testGETProgrammersCollection()
    {
... lines 108 - 120
        $this->asserter()->assertResponsePropertyIsArray($response, 'items');
        $this->asserter()->assertResponsePropertyCount($response, 'items', 2);
        $this->asserter()->assertResponsePropertyEquals($response, 'items[1].nickname', 'CowboyCoder');
    }
... lines 125 - 315
}
```

We're also missing the pagination fields, making it harder for our API clients to guess how our responses will look.

Nope, we can do better, guys.

Chapter 18: Rock-Solid, Consistent Collection Endpoints

Go back to the test we're working on right now. First, every collection should have an items key for consistency. Assert that with $this->asserter()->assertResponsePropertyExists() for items:

318 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
class ProgrammerControllerTest extends ApiTestCase
{
... lines 10 - 64
    public function testFollowProgrammerBattlesLink()
    {
... lines 67 - 86
        $this->asserter()->assertResponsePropertyExists($response, 'items');
        $this->debugResponse($response);
    }
... lines 90 - 316
}
```

Pagination in the Past

Next, open ProgrammerController. The *whole* reason the other endpoint had an items key was because - in listAction() - we went through our fancy pagination system. Click into the pagination_factory. The *key* part is that this method eventually creates a PaginatedCollection object:

58 lines | src/AppBundle/Pagination/PaginationFactory.php

```
... lines 1 - 10
class PaginationFactory
{
... lines 13 - 19
    public function createCollection(QueryBuilder $qb, Request $request, $route, array $routeParams = array())
    {
... lines 22 - 33
        $paginatedCollection = new PaginatedCollection($programmers, $pagerfanta->getNbResults());
... lines 35 - 55
        return $paginatedCollection;
    }
}
```

This is what we feed to the serializer.

The PaginatedCollection object is something *we* created. And hey! It has an $items property! So this isn't rocket science. It also has a few other properties: total, count and the pagination links:

27 lines | src/AppBundle/Pagination/PaginatedCollection.php

```
... lines 1 - 4
class PaginatedCollection
{
    private $items;

    private $total;

    private $count;

    private $_links = array();
... lines 14 - 25
}
```

So if we want *every* collection endpoint to be identical, every endpoint should return a PaginatedCollection.

Creating a PaginatedCollection

We could do this the simple way: $collection = new PaginatedCollection() and pass it $battles and the total items - which right now is count($battles):

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 154
    public function battlesListAction(Programmer $programmer)
    {
        $battles = $this->getDoctrine()->getRepository('AppBundle:Battle')
            ->findBy(['programmer' => $programmer]);

        $collection = new PaginatedCollection($battles, count($battles));
... lines 161 - 162
    }
}
```

There's not *actually* any pagination going on.

At the bottom, pass that $collection to createApiResponse():

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 154
    public function battlesListAction(Programmer $programmer)
    {
        $battles = $this->getDoctrine()->getRepository('AppBundle:Battle')
            ->findBy(['programmer' => $programmer]);

        $collection = new PaginatedCollection($battles, count($battles));

        return $this->createApiResponse($collection);
    }
}
```

Done! Run that test:

```
$ ./vendor/bin/phpunit --filter testFollowProgrammerBattlesLink
```

Yes! *Now* we have an items key, *and* total, count and _links... which is empty.

Adding Real Pagination

And really: if we're going to all of this trouble to use the PaginatedCollection, shouldn't we go one extra half-step and actually add pagination? After all, it'll make this endpoint even more consistent by having those pagination links.

Change the $collection = line to $this->get('pagination_factory')->createCollection():

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

```
... lines 1 - 23
24  class ProgrammerController extends BaseController
25  {
... lines 26 - 154
155     public function battlesListAction(Programmer $programmer, Request $request)
156     {
... lines 157 - 159
160         $collection = $this->get('pagination_factory')->createCollection(
... lines 161 - 164
165         );
... lines 166 - 167
168     }
169 }
```

This needs a few arguments. The first is a query builder. So instead of making this full query for battles, we need to *just* return the query builder. Rename this to a new method called - createQueryBuilderForProgrammer() and pass it the $programmer object:

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

```
... lines 1 - 23
24  class ProgrammerController extends BaseController
25  {
... lines 26 - 154
155     public function battlesListAction(Programmer $programmer, Request $request)
156     {
157         $battlesQb = $this->getDoctrine()->getRepository('AppBundle:Battle')
158             ->createQueryBuilderForProgrammer($programmer);
... lines 159 - 167
168     }
169 }
```

I'll hold command and I'll click Battle to jump into BattleRepository. Add that method:
public function createQueryBuilderForProgrammer() with a Programmer $programmer argument:

17 lines | src/AppBundle/Repository/BattleRepository.php

```
... lines 1 - 4
5   use AppBundle\Entity\Programmer;
... lines 6 - 7
8   class BattleRepository extends EntityRepository
9   {
10      public function createQueryBuilderForProgrammer(Programmer $programmer)
11      {
... lines 12 - 14
15      }
16  }
```

Fortunately, the query is easy: return $this->createQueryBuilder('battle'), then ->andWhere('battle.programmer = :programmer') with setParameter('programmer', $programmer):

17 lines | src/AppBundle/Repository/BattleRepository.php

```
... lines 1 - 7
class BattleRepository extends EntityRepository
{
    public function createQueryBuilderForProgrammer(Programmer $programmer)
    {
        return $this->createQueryBuilder('battle')
            ->andWhere('battle.programmer = :programmer')
            ->setParameter('programmer', $programmer);
    }
}
```

Perfect! Back in ProgrammerController, rename the variable to $battlesQb and pass it to createCollection():

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 154
    public function battlesListAction(Programmer $programmer, Request $request)
    {
        $battlesQb = $this->getDoctrine()->getRepository('AppBundle:Battle')
            ->createQueryBuilderForProgrammer($programmer);

        $collection = $this->get('pagination_factory')->createCollection(
            $battlesQb,
... lines 162 - 164
        );
... lines 166 - 167
    }
}
```

The second argument is the request object. You guys know what to do: type-hint a new argument with Request and pass that in:

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

```
... lines 1 - 16
use Symfony\Component\HttpFoundation\Request;
... lines 18 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 154
    public function battlesListAction(Programmer $programmer, Request $request)
    {
        $battlesQb = $this->getDoctrine()->getRepository('AppBundle:Battle')
            ->createQueryBuilderForProgrammer($programmer);

        $collection = $this->get('pagination_factory')->createCollection(
            $battlesQb,
            $request,
... lines 163 - 164
        );
... lines 166 - 167
    }
}
```

The third argument is the name of the route the pagination links should point to. That's *this* route: api_programmers_battles_list. Finally, the last argument is any route parameters that need to be passed to the route. This route has a nickname, so pass nickname => $programmer->getNickname():

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 154
    public function battlesListAction(Programmer $programmer, Request $request)
    {
... lines 157 - 159
        $collection = $this->get('pagination_factory')->createCollection(
            $battlesQb,
            $request,
            'api_programmers_battles_list',
            ['nickname' => $programmer->getNickname()]
        );
... lines 166 - 167
    }
}
```

Done. We basically changed one line to create a *real* paginated collection. And now, we celebrate. Run the test:

```
$ ./vendor/bin/phpunit --filter testFollowProgrammerBattlesLink
```

That is *real* pagination pretty much out of the box. Yea, this only has three results and only one page: but if this programmer keeps having battles, we're covered.

We've really perfected a lot of traditional REST endpoints. Now, let's talk about what happens when endpoints get weird...

Chapter 19: Weird Endpoint: The tagline as a Resource?

Most of our endpoints are pretty straightforward: We create a programmer, we update a programmer, we create a battle, we get a collection of battles.

Reality check! In the wild: endpoints get weird. Learning how to handle these was one of the most *frustrating* parts of REST for me. So let's code through two examples.

Updating *just* the Tagline?

Here's the first: suppose you decide that it would be really nice to have an endpoint where your client can edit the tagline of a programmer directly.

Now, technically, that's already possible: send a PATCH request to the programmer endpoint and only send the tagline.

But remember: we're building the API for our API clients, and if they want an endpoint *specifically* for updating a tagline, give it to them.

Open ProgrammerControllerTest: let's design the endpoint first. Make a public function testEditTagline():

334 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8   class ProgrammerControllerTest extends ApiTestCase
9   {
... lines 10 - 317
318     public function testEditTagline()
319     {
... lines 320 - 331
332     }
333 }
```

Scroll to the top and copy the $this->createProgrammer() line that we've been using. Give this a specific tag line:
The original UnitTester:

334 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8   class ProgrammerControllerTest extends ApiTestCase
9   {
... lines 10 - 317
318     public function testEditTagline()
319     {
320         $this->createProgrammer(array(
321             'nickname' => 'UnitTester',
322             'avatarNumber' => 3,
323             'tagLine' => 'The original UnitTester'
324         ));
... lines 325 - 331
332     }
333 }
```

The URL Structure

Now, if we want an endpoint where the *only* thing you can do is edit the tagLine, how should that look?

One way to think about this is that the tagLine is a *subordinate string* resource of the programmer. Remember also that every

URI is supposed to represent a different resource. If you put those 2 ideas together, a great URI becomes obvious: /api/programmers/UnitTester/tagline. In fact, if you think of this as its own resource, then all of a sudden, you could imagine creating a GET endpoint to fetch *only* the tagline or a PUT endpoint to update *just* the tagline. It's a cool idea!

And that's what we'll do: make an update request with $this->client->put() to this URL: /api/programmers/UnitTester/tagline:

334 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8   class ProgrammerControllerTest extends ApiTestCase
9   {
... lines 10 - 317
318     public function testEditTagline()
319     {
320         $this->createProgrammer(array(
321             'nickname' => 'UnitTester',
322             'avatarNumber' => 3,
323             'tagLine' => 'The original UnitTester'
324         ));
325
326         $response = $this->client->put('/api/programmers/UnitTester/tagline', [
... lines 327 - 328
329         ]);
... lines 330 - 331
332     }
333 }
```

How to send the Data?

Send the normal Authorization header:

334 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8   class ProgrammerControllerTest extends ApiTestCase
9   {
... lines 10 - 317
318     public function testEditTagline()
319     {
... lines 320 - 325
326         $response = $this->client->put('/api/programmers/UnitTester/tagline', [
327             'headers' => $this->getAuthorizedHeaders('weaverryan'),
... line 328
329         ]);
... lines 330 - 331
332     }
333 }
```

But how should we pass the new tagline data? Normally, we send a json-encoded array of fields. But this resource isn't a collection of fields: it's just *one* string. There's nothing wrong with sending some JSON data up like before, but you could also set the body to the plain-text New Tag Line itself:

334 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
class ProgrammerControllerTest extends ApiTestCase
{
... lines 10 - 317
    public function testEditTagline()
    {
... lines 320 - 325
        $response = $this->client->put('/api/programmers/UnitTester/tagline', [
            'headers' => $this->getAuthorizedHeaders('weaverryan'),
            'body' => 'New Tag Line'
        ]);
... lines 330 - 331
    }
}
```

And I think this is pretty cool.

Finish this off with $this->assertEquals() 200 for the status code:

334 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
class ProgrammerControllerTest extends ApiTestCase
{
... lines 10 - 317
    public function testEditTagline()
    {
... lines 320 - 325
        $response = $this->client->put('/api/programmers/UnitTester/tagline', [
            'headers' => $this->getAuthorizedHeaders('weaverryan'),
            'body' => 'New Tag Line'
        ]);
        $this->assertEquals(200, $response->getStatusCode());
... line 331
    }
}
```

But what should be returned? Well, whenever we edit or create a resource, we return the resource that we just edited or created. In this context, the tagline *is* its own resource... even though it's just a string. So instead of expecting JSON, let's look for the literal text: $this->assertEquals() that New Tag Line is equal to the string representation of $response->getBody():

334 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 329
        $this->assertEquals(200, $response->getStatusCode());
        $this->assertEquals('New Tag Line', (string) $response->getBody());
... lines 332 - 334
```

But you don't *need* to do it this way: you might say:

> Look, we all know that you're *really* editing the UnitTester programmer resource, so I'm going to return that.

And that's fine! This is an interesting *option* for how to think about things. Just as long as you don't spend your days dreaming philosophically about your API, you'll be fine. Make a decision and feel good about it. In fact, that's good life advice.

Adding the String Resource Endpoint

Let's finish this endpoint. At the bottom of ProgrammerController, add a new public function editTaglineAction():

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 173
    public function editTagLineAction(Programmer $programmer, Request $request)
    {
... lines 176 - 181
    }
}
```

We already know that the route should be /api/programmers/{nickname}/tagline. To be super hip, add an @Method annotation: we know this should only match PUT requests:

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 169
    /**
     * @Route("/api/programmers/{nickname}/tagline")
     * @Method("PUT")
     */
    public function editTagLineAction(Programmer $programmer, Request $request)
    {
... lines 176 - 181
    }
}
```

Like before, type-hint the Programmer argument so that Doctrine will query for it *for* us, using the nickname value. And, we'll also need the Request argument:

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

```
... lines 1 - 7
use AppBundle\Entity\Programmer;
... lines 9 - 16
use Symfony\Component\HttpFoundation\Request;
... lines 18 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 173
    public function editTagLineAction(Programmer $programmer, Request $request)
    {
... lines 176 - 181
    }
}
```

I *could* use a form like before... but this is just *so* simple: $programmer->setTagLine($request->getContent()):

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 173
    public function editTagLineAction(Programmer $programmer, Request $request)
    {
        $programmer->setTagLine($request->getContent());
... lines 177 - 181
    }
}
```

Literally: read the text from the request body and set that on the programmer.

Now, save: $em = $this->getDoctrine()->getManager(), $em->persist($programmer) and $em->flush():

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

```
... lines 1 - 175
        $programmer->setTagLine($request->getContent());
        $em = $this->getDoctrine()->getManager();
        $em->persist($programmer);
        $em->flush();
... lines 180 - 184
```

For the return, it's not JSON! Return a plain new Response() with $programmer->getTagLine(), a 200 status code, and a Content-Type header of text/plain:

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

```
... lines 1 - 17
use Symfony\Component\HttpFoundation\Response;
... lines 19 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 173
    public function editTagLineAction(Programmer $programmer, Request $request)
    {
... lines 176 - 180
        return new Response($programmer->getTagLine());
    }
}
```

Now, this is a good-looking, super-weird endpoint. Copy the test method name and try it out:

```
$ ./vendor/bin/phpunit --filter testEditTagLine
```

We're green! Next, let's look at a *weirder* endpoint.

Chapter 20: Weird Endpoint: Command: Power-Up a Programmer

On our web interface, if you select a programmer, you can start a battle, *or* you can hit this "Power Up" button. Sometimes our power goes up, sometimes it goes down. And isn't that just like life.

The higher the programmer's power level, the more likely they will win future battles.

Notice: all we need to do is click one button: Power Up. We don't fill in a box with the desired power level and hit submit, we just "Power Up"! And that makes this a weird endpoint to build for our API.

Why? Basically, it doesn't easily fit into REST. We're not sending or editing a resource. No, we're more issuing a command: "Power Up!".

Let's design this in a test: public function testPowerUp():

351 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8   class ProgrammerControllerTest extends ApiTestCase
9   {
... lines 10 - 333
334     public function testPowerUp()
335     {
... lines 336 - 348
349     }
350 }
```

Grab the $programmer and Response lines from above, but replace tagLine with a powerLevel set to 10:

351 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8   class ProgrammerControllerTest extends ApiTestCase
9   {
... lines 10 - 333
334     public function testPowerUp()
335     {
336         $this->createProgrammer(array(
337             'nickname' => 'UnitTester',
338             'avatarNumber' => 3,
339             'powerLevel' => 10
340         ));
341
342         $response = $this->client->post('/api/programmers/UnitTester/powerup', [
343             'headers' => $this->getAuthorizedHeaders('weaverryan')
344         ]);
... lines 345 - 348
349     }
350 }
```

Now we know that the programmer *starts* with this amount of power.

The URL Structure of a Command

From here, we have *two* decisions to make: what the URL should look like and what HTTP method to use. Well, we're

issuing a command for a specific programmer, so make the URL /api/programmers/UnitTester/powerup:

351 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
class ProgrammerControllerTest extends ApiTestCase
{
... lines 10 - 333
    public function testPowerUp()
    {
... lines 336 - 341
        $response = $this->client->post('/api/programmers/UnitTester/powerup', [
... line 343
        ]);
... lines 345 - 348
    }
}
```

Here's where things get ugly. This is a new URI... so philosophically, this represents a new resource. Following what we did with the tag line, we should think of this as the "power up" resource. So, are we editing the "power up" resource... or are we doing something different?

The "Power Up?" Resource???

Are you confused? I'm kind of confused. It just doesn't make sense to talk about some "power up" resource. "Power up" is *not* a resource, even though the rules of REST want it to be. We just had to create *some* URL... and this made sense.

So if this isn't a resource, how do we decide whether to use PUT or POST? Here's the key: when REST falls apart and your endpoint doesn't fit into it anymore, use POST.

POST for Weird Endpoints

Earlier, we talked about how PUT is idempotent, meaning if you make the same request 10 times, it has the same effect as if you made it just once. POST is *not* idempotent: if you make a request 10 times, each request *may* have additional side effects.

Usually, this is how we decide between POST and PUT. And it fits here! The "power up" endpoint is *not* idempotent: hence POST.

But wait! Things are *not* that simple. Here's the rule I want you to follow. *If* you're building an endpoint that fits into the rules of REST: choose between POST and PUT by asking yourself if it is idempotent.

But, if your endpoint does *not* fit into REST - like this one - always use POST. So even if the "power up" endpoint *were* idempotent, I would use POST. In reality, a PUT endpoint *must* be idempotent, but a POST endpoint is allowed to be either.

So, use ->post(). And now, remove the body: we are not sending any data. This is why POST fits better: we're not really *updating* a resource:

351 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 7
8   class ProgrammerControllerTest extends ApiTestCase
9   {
... lines 10 - 333
334     public function testPowerUp()
335     {
... lines 336 - 341
342         $response = $this->client->post('/api/programmers/UnitTester/powerup', [
343             'headers' => $this->getAuthorizedHeaders('weaverryan')
344         ]);
... lines 345 - 348
349     }
350 }
```

And the Endpoint Returns....?

Assert that 200 matches the status code:

351 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 341
342         $response = $this->client->post('/api/programmers/UnitTester/powerup', [
343             'headers' => $this->getAuthorizedHeaders('weaverryan')
344         ]);
345         $this->assertEquals(200, $response->getStatusCode());
... lines 346 - 351
```

And now, what should the endpoint return?

We're not in a normal REST API situation, so it matters less. You could return nothing, or you could return the power level. But to be as predictable as possible, let's return the entire programmer resource. Read the new power level from this with $this->asserter()->readResponseProperty() and look for powerLevel:

351 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 341
342         $response = $this->client->post('/api/programmers/UnitTester/powerup', [
343             'headers' => $this->getAuthorizedHeaders('weaverryan')
344         ]);
345         $this->assertEquals(200, $response->getStatusCode());
346         $powerLevel = $this->asserter()
347             ->readResponseProperty($response, 'powerLevel');
... lines 348 - 351
```

This *is* a property that we're exposing:

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

```
... lines 1 - 31
class Programmer
{
... lines 34 - 67
    /**
... lines 69 - 71
     * @Serializer\Expose
     */
    private $powerLevel = 0;
... lines 75 - 206
}
```

We don't know what this value will be, but it *should* change. Use assertNotEquals() to make sure the new powerLevel is no longer 10:

351 lines | tests/AppBundle/Controller/Api/ProgrammerControllerTest.php

```
... lines 1 - 341
        $response = $this->client->post('/api/programmers/UnitTester/powerup', [
            'headers' => $this->getAuthorizedHeaders('weaverryan')
        ]);
        $this->assertEquals(200, $response->getStatusCode());
        $powerLevel = $this->asserter()
            ->readResponseProperty($response, 'powerLevel');
        $this->assertNotEquals(10, $powerLevel, 'The level should change');
... lines 349 - 351
```

Implement the Endpoint

Figuring out the URL and HTTP method was the hard part. Let's finish this. In ProgrammerController, add a new public function powerUpAction():

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 187
    public function powerUpAction(Programmer $programmer)
    {
... lines 190 - 193
    }
}
```

Add a route with /api/programmers/{nickname}/powerup and an @Method set to POST:

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 183
    /**
     * @Route("/api/programmers/{nickname}/powerup")
     * @Method("POST")
     */
    public function powerUpAction(Programmer $programmer)
    {
... lines 190 - 193
    }
}
```

Once again, type-hint the Programmer argument:

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

```
... lines 1 - 7
use AppBundle\Entity\Programmer;
... lines 9 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 187
    public function powerUpAction(Programmer $programmer)
    {
... lines 190 - 193
    }
}
```

To power up, we have a service already made for this. Just say: $this->get('battle.power_manager') ->powerUp() and pass it the $programmer:

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 187
    public function powerUpAction(Programmer $programmer)
    {
        $this->get('battle.power_manager')
            ->powerUp($programmer);
... lines 192 - 193
    }
}
```

That takes care of everything. Now, return $this->createApiResponse($programmer):

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

```
... lines 1 - 23
class ProgrammerController extends BaseController
{
... lines 26 - 187
    public function powerUpAction(Programmer $programmer)
    {
        $this->get('battle.power_manager')
            ->powerUp($programmer);

        return $this->createApiResponse($programmer);
    }
}
```

Done! Copy the testPowerUp() method name and run that test:

```
$ ./vendor/bin/phpunit -—filter testPowerUp
```

Success!

And that's it - that's everything. I *really* hope this course will save you from some frustrations that I had. Ultimately, don't over-think things, add links when they're helpful and build your API for whoever will actually use it.

Ok guys - seeya next time!